REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l 'Enseignement Supérieur et de la Recherche Scientifique

Université Abderrahmane Mira de Bejaïa

Faculté des Sciences Exactes

Département d'Informatique

École Doctorale d'Informatique

Mémoire de Magistère en Informatique

Option : Réseaux et Systèmes Distribués

Thème

Concepts et Algorithmes pour les Mémoires Transactionnelles Logicielles

Présenté par

Samir BOULDJADJ

Devant le jury :

Président :	Abdelnasser DAHMANI,	Professeur,	Université de Béjaïa.
Rapporteur :	Moussa KERKAR,	Professeur,	Université de Béjaïa.
Examinateur :	Abdelkamel TARI,	MCA,	Université de Béjaïa.
Examinateur :	Ali MELIT,	Professeur,	Université de Jijel.
Invité :	Hamouma MOUMEN,	MAA,	Université de Béjaïa.

Promotion 2008/2009

Remerciements

*« Tout d'abord, je remercie **Dieu**, le tout puissant de nous avoir accordé le savoir, et qui nous a donné la volonté et le courage pour poursuivre mes études.».*

Je tiens à exprimer tous mes vifs remerciements et ma profonde gratitude à mon promoteur : Pr M. KERKAR et Mr, H. MOUMEN pour leurs disponibilités, aide et bonne humeur durant toutes les étapes de ce projet. Leurs dévouements, conseils scientifiques et suivis, m'ont permis de mener mon travail à terme.

Je remercie tous nos enseignants de l'école doctorale ReSyD. Merci pour Monsieur A. Tari, chef de département d'informatique de l'université de Bejaia, pour tout ce qu'il a fait pour l'école doctorale.

Mes sincères Remerciements et ma profonde gratitude vont également au président et aux membres de jury pour l'honneur qu'ils m'accordent, en acceptant de juger mon travail.

Et enfin merci à tous ceux qui ont contribué de prés ou de loin durant l'accomplissement de mes études.

Merci à tous.

Dédicace

Parce qu'il y a des choses plus faciles à écrire qu'à dire, Je dédie ce modeste travail:

A mes chers parents que Dieu me les garde.

A mes chers frères et sœurs chacun de son nom pour leur soutien morale et leurs sacrifices le long de ma formation.

A tous qui m'ont encouragé et aidé dans la réalisation de ce mémoire.

A tous mes amis chacun de son nom.

A tous mes collègues de promotion 2008.

A touts ceux que j'aime et m'aiment.

Samir 2011.

Résumé

La révolution multi-cœurs dans ces dernières années a mis les techniques utilisées pour la programmation parallèle en cause. Ces techniques sont largement fondées sur les verrous qui sont difficiles à gérer et qui souffrent d'un ensemble de problèmes comme l'inter-blocage, l'inversion de priorité, le Livelook, etc. qui réduisent leur utilisation et rendent le développement des applications parallèles une tâche très délicate même pour les experts. Ces problèmes ont poussé les chercheurs à trouver d'autres solutions pour réaliser la synchronisation dans les applications parallèles sans fait recours aux verrous. Pour cette fin, les deux dernières décennies ont connu la naissance de la *Mémoire Transactionnelle*, une nouvelle technique pour réaliser la synchronisation dans les applications parallèles qui vise à fournir une haute évolutivité en comparaison avec le verrouillage à grain gossier et à être plus facile à utiliser que le verrouillage à grain fin.

L'idée principale derrière cette technique est d'exécuter les séquences de code critiques d'une manière atomique dans des transactions. Les recherches indiquent que les mémoires transactionnelles puissent être plus performantes que les verrous traditionnels. Vu l'importance du sujet, le manque des travaux dans ce domaine et pour comprendre les différents concepts et implémentations de mémoire transactionnelle logicielle existants, une claire étude comparative de ces implémentations est nécessaire. Malgré l'existence de certains travaux dans ce sens, mais ces travaux sont très restreints et ne couvrent que quelques implémentations. Ce mémoire présente les différents concepts utilisés dans le domaine des STMs, il fournit une étude détaillée des STMs proéminents représentant l'état de l'art jusqu'à nos jours en discutant la différence entre leurs choix de conception et d'implémentation.

Mots clés : Multiprocesseur, Concurrence, Synchronisation, Mémoire transactionnelle.

Abstract

The multicore revolution in recent years has made the techniques used for parallel programming in question. These techniques are largely based on the locks that are difficult to manage and suffering a series of problems such as deadlock, priority inversion, LiveLook, etc. that reduce their use and make the development of parallel applications a very difficult task even for experts. These problems have pushed researchers to find other ways to achieve synchronization in parallel applications without use locks. To this end, the last two decades have witnessed the birth of *transactional memory*, a new technique to achieve synchronization in parallel applications that aims to provide high scalability in comparison with the coarse grained locking and be easier to use then the fine-grained locking.

The main idea behind this technique is to run the critical code sequences atomically in transactions. Research indicates that transactional memory can be more efficient than traditional locks. Given the importance of subject, lack of work in this field and to understand the different concepts and implementations of existing software transactional memory, a clear comparative study of these implementations is necessary. Despite the existence of certain work in this direction, but this works are very limited and cover only some implementations. This work presents the different concepts used in the field of STMs, it provides a detailed study of STMs prominent representing the state of the art until today discussing the difference between their design choices and implementation.

Keywords: Multiprocessor, Concurrency, Synchronization, Transactional memory.

Terminologie

A défaut de bonne traduction en français, nous emploierons dans ce mémoire le terme *commit* pour désigner la fin avec succès d'une transaction et le fait que ses modifications soient visibles par tous les autres threads, et le terme *abort* pour désigner l'interruption et l'annulation d'une transaction suite à un conflit. Par extension, nous utiliserons les verbes *commiter* et *aborter*. Enfin, nous étendrons ces actions aux processeurs, i.e. un processeur commite si la transaction qu'il est en train d'exécuter commite. Le terme transactionnel désigne ce qui a relation à une transaction. Ainsi, des données transactionnelles sont des données accédées durant une transaction.

Table des matières

Liste des Figures

Liste des Tableaux

1

Introduction

Depuis le lancement du premier ordinateur sur le marché, les demandes en capacité de calcul sont devenues de plus en plus importantes dans tous les domaines. Ces demandes sont satisfaites en augmentant la vitesse d'horloge des processeurs. Cependant, cette vitesse a atteint la limite et elle ne peut plus être augmentée. Le besoin continu de puissance de calcule mène à la révolution multi-cœurs où chaque machine est dotée de plus d'une unité de calcul. Une telle machine est capable d'exécuter plusieurs processus en parallèle, ce qui nécessite des applications robustes et évolutives pour pouvoir utiliser la pleine puissance de ces machines parallèles.

Les applications parallèles dans ce contexte utilisent une mémoire partagée entre plusieurs processus, la communication entre ces processus se fait en lisant et en écrivant des variables partagées ce qui nécessite une synchronisation entre ces processus. Le développement d'une telle application avec les méthodes classiques est une tâche très délicate même pour les experts. Les verrous et l'exclusion mutuelle sont les mécanismes utilisés pour réaliser cette synchronisation. Cependant, ces mécanismes sont connus par ses vulnérabilités aux erreurs et peuvent être un goulot d'étranglement pour la performance du système. Les deux dernières décennies ont connu à l'issu d'intenses recherches la naissance de la Mémoire Transactionnelle [67,34,6,23,52], une technique prometteuse pour le contrôle de concurrence dans les architectures multiprocesseurs modernes.

L'idée principale derrière la Mémoire Transactionnelle est d'exécuter les séquences de code critiques d'une manière atomique dans des transactions. Les transactions ont été utilisées long temps dans les bases de données et elles garantissent un partage de ressources sans famine ni inter-blocage [27]. Si la transaction s'exécute sans conflit elle sera commitée avec succès, si elle rencontre un conflit elle sera annulée et relancée à nouveau. Un conflit apparait lorsque deux transactions ou plus accèdent à la même donnée simultanément avec au moins un accès est une écriture. La mémoire transactionnelle maintient un log pour chaque transaction contenant les valeurs des emplacements mémoire modifiés par la transaction et ça à fin de pouvoir restituer leurs valeurs originales en cas d'abort de la transaction.

Une Mémoire Transactionnelle peut être implémentée complètement en logiciel (STM), complètement en matériel (HTM) ou d'une façon hybride (HyTM). L'idée d'utiliser les transactions pour faire la synchronisation dans la programmation parallèle a été introduite pour la première fois en 1993 par Maurice Herlihy et autres [41], ils ont réalisé une implémentation matérielle de la mémoire transactionnelle. Deux ans après Shavit et Touitou [79] ont introduit l'idée de la mémoire transactionnelle logicielle. Ensuite des implémentations hybrides ont été développées [19,46]. Dans le cadre de ce travail nous nous intéresserons à l'étude des implémentations logicielles de mémoire transactionnelle et certaines implémentations hybrides.

Les systèmes STMs sont d'intérêt particulier parce qu'ils n'exigent aucune modification de matériel de base. Ils peuvent être mis en application soit en tant qu'une partie du compilateur d'un langage de haut niveau, soit au niveau de l'environnement d'exécution virtuel, ou même comme une bibliothèque externe [32].

Vu l'importance du sujet, le manque des travaux dans ce domaine et pour comprendre les différents concepts et implémentations de mémoire transactionnelle logicielle existants, une claire étude comparative de ces implémentations est nécessaire. Une telle étude va aider les chercheurs et les praticiens pour développer des implémentions de mémoire transactionnelle logicielle plus efficaces en termes de performance. Malgré l'existence de certains travaux dans ce sens, ces travaux sont très restreints et ne couvrent que quelques implémentations. Une étude comparative a été réalisée par Marathe et Scott [62] dans laquelle ils ont comparé les implémentations FSTM [26], DSTM [44] et WSTM [35]. De même un livre écrit par J.Larus et R.Rajwar [59] dans lequel ils ont discuté tous les STMs existants jusqu'au milieu de l'an 2006. Cependant, ce mémoire présente un état de l'art des implémentations de mémoire transactionnelle logicielle où dix-neuf implémentations ont été étudiées dont dix implémentations n'ont pas été discutées dans le livre de J.Larus et R.Rajwar.

Organisation du mémoire

Le reste de ce mémoire est organisé comme suit : Le chapitre deux a pour objectif de cerner le contexte générale de notre étude centrée sur les architectures multiprocesseurs. Dans le chapitre trois, nous ferons une présentation de tous les concepts et notions utilisés dans le domaine de mémoire transactionnelle logicielle. Le chapitre quatre fournit une étude détaillée de dix-neuf implémentations de mémoire transactionnelle logicielle. Dans le chapitre cinq, nous discuterons les résultats de notre étude et nous identifierons certains problèmes et défis de la mémoire transactionnelle. Enfin, le chapitre six vient pour conclure notre travail.

Contributions

Notre travail à notre connaissance est le premier travail dans ce domaine en Algérie. Peu de travaux dans la littérature abordent ce sujet. A travers ce mémoire, les contributions suivantes sont apportées;

- Une présentation des systèmes distribués à mémoire partagée est faite avec une discussion de tous les problèmes et défis caractérisant ce type de systèmes.
- Un aperçu de la programmation parallèle, tout en discutant les problèmes et les difficultés rencontrées en utilisant les approches de synchronisation traditionnelles.
- L'importance des mémoires transactionnelles logicielles dans la programmation parallèle est présentée avec un aperçu des travaux de recherche réalisés dans ce domaine.
- Les concepts et les axes de conception de Mémoire Transactionnelle Logicielle sont discutés. Par exemple l'existence et l'importance des transactions dans la programmation de base de données, la sémantique et les constructions des mémoires transactionnelles logicielles. En outre, les choix de conceptions comme les transactions emboitées, la granularité, les stratégies de mise à jour, la détection des conflits et des politiques de gestion conflit sont présentées.
- Une étude détaillée de dix neufs systèmes de mémoire transactionnelle logicielle et hybrides est présentée dont dix systèmes n'ont pas été couverts par Larus et Rajwar dans [59].
- Les limitations de conception d'un certain nombre des systèmes de mémoire transactionnelle sont discutées, tout en proposant certaines solutions à ces limitations.
- Une discussion approfondie est effectuée sur les différentes caractéristiques techniques des systèmes de mémoire transactionnelle logicielle, tout en discutant les diverses tendances et leur impact sur la performance.
- Certains défis et problèmes de mémoire transactionnelle sont identifiés.

2

Systèmes distribués à mémoire Partagée

Nous présenterons dans ce chapitre le contexte général de notre étude, centrée sur les systèmes distribués à mémoire partagée (systèmes multiprocesseurs ou multi-cœurs) qui ont connu ces dernières années un essor remarquable. Un tel système est composé de plusieurs unités de calcul (processeurs ou cœurs) qui se communiquent en lisant et en écrivant un espace mémoire commun. Nous présenterons ici des généralités sur ce type de systèmes, les architectures existantes et les différents concepts relatifs à ces systèmes.

2.1 Définition d'un système distribué

Les systèmes distribués existent dès l'existence de l'univers, d'un troupeau des poissons à une bande des oiseaux et aux écosystèmes entiers des micro-organismes, il y a une communication entre les agents intelligents mobiles dans la nature. Avec la prolifération répandue de l'Internet et la naissance du village global, la notion des systèmes informatiques distribués comme un outil utile et largement déployé devient une réalité. Donc c'est quoi un système distribué ?

Plusieurs définitions de système distribué ont été données dans la littérature aucune d'elles n'est satisfaisante et aucune d'elles n'est en accord avec l'autre, on peut citer quelques définitions :

- Vous savez que vous utilisez un quand le crash d'un ordinateur que vous n'avez jamais entendu parler vous empêche d'effectuer le travail [Lamport].
- Un système distribué est une collection d'ordinateurs indépendants qui apparaît à ses utilisateurs comme un système unique cohérent [Tanenbaum].
- Un terme qui décrit une large gamme d'ordinateurs, de systèmes faiblement couplés tel que les réseaux de large porté *WANs*, aux systèmes fortement couplés tel que les réseaux locaux LANs, aux systèmes très fortement couplés tels que les systèmes multiprocesseurs [29].
- Un système distribué est une collection des unités de calcul qui peuvent communiquer entre elles [34].

Dans le cadre de ce mémoire, nous nous intéresserons aux *systèmes parallèles multiprocesseurs* (SMP). La section suivante va discuter les différents types de ces systèmes.

2.2 Les architectures des systèmes distribués

Les systèmes distribués sont classés en se basant sur le mécanisme de communication utilisé entre les différents nœuds en deux classes :

2.2.1 Les architectures à communication par envoie de messages

Dans cette architecture, chaque processeur à une mémoire locale accessible seulement par lui, et la communication entre les différents processeurs se fait par échange des messages à travers le support de communication. La Figure 2.1 schématise un tel système.

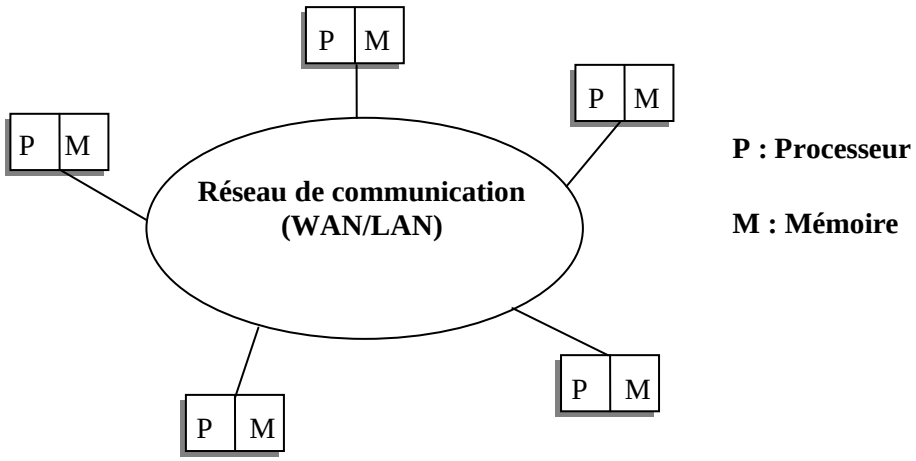

Figure 2.1 : Système distribué où la communication est par échange de messages

2.2.2 Les architectures à communication par mémoire partagée

Les systèmes à mémoire partagée ont au moins une partie de la mémoire accessible par tous les processeurs ce qui permet aux processeurs de communiquer directement entre eux par des opérations de lecture/écrire de cette mémoire partagée. La coordination et la synchronisation entre tous les processeurs sont réalisées via la mémoire partagée, Un système informatique à mémoire partagée se compose d'un ensemble de processeurs indépendants, d'un ensemble de modules mémoires, et d'un réseau d'interconnexion. On peut distinguer trois implémentations différentes;

2.2.2.1 Mémoire à accès non uniforme (NUMA)

Dans cette architecture, chaque processeur possède une partie de la mémoire partagée attachée à lui, la mémoire forme un espace d'adressage unique et chaque processeur peut accéder à n'importe case mémoire directement en utilisant l'adresse réelle. Cependant, le temps d'accès dépend de la distance entre le processeur et la casse mémoire, d'où il vient *NUMA*. Une telle organisation est schématisée dans la Figure 2.2.

2.2.2.2 Mémoire à accès uniforme (UMA)

Cette architecture correspond bien aux systèmes parallèles multiprocesseurs dans laquelle les processeurs ont un accès direct à la mémoire partagée (UMA) qu'elle forme un espace d'adressage commun même si la mémoire est organisée en banques multiples.

La forme la plus simple de cette architecture est constituée de deux processeurs partageant un seul module mémoire (Figure 2.3). Les requêtes d'accès arrivent au module mémoire à travers ses deux ports. Le module mémoire possède une unité d'arbitration qui intercepte les demandes et les fait passer au contrôleur mémoire. Si le module mémoire n'est pas occupé alors la première requête qui arrive sera servie et le module se place dans l'état occupé, cependant si le module mémoire reçoit une requête et il est dans l'état occupé (c.à.d. il est en train de servir une requête précédente) alors le module mémoire envoie un signal d'attente au processeur demandant à travers le contrôleur mémoire et en réponse à ce signal, le processeur demandant peut maintenir sa requête sur la ligne ou il peut la répéter plus tard. Cependant, si l'unité d'arbitration reçoit deux requêtes en même temps, elle doit choisir une et la faire passer au contrôleur tandis que l'autre requête sera retardée.

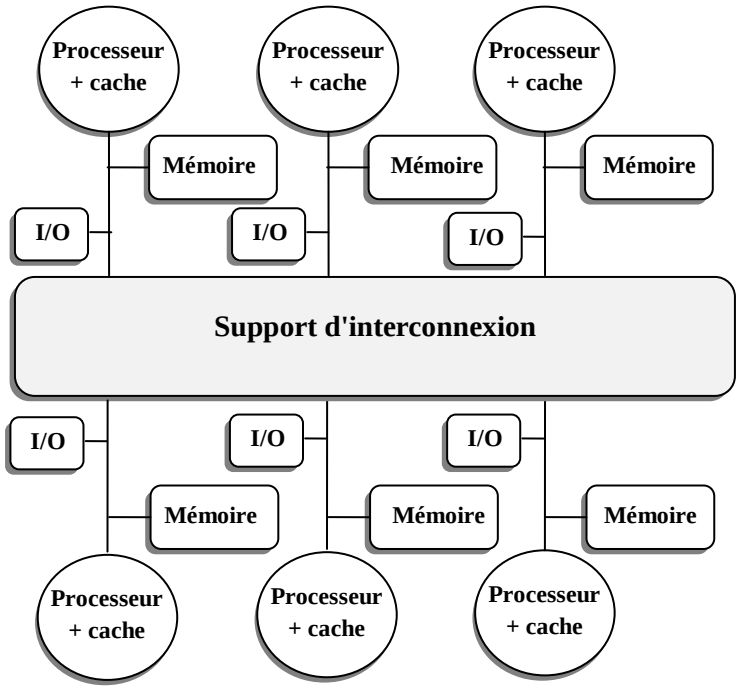

Figure 2.2 : L'architecture NUMA.

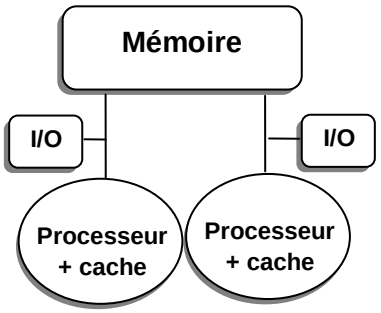

Figure 2.3 : Mémoire partagée entre deux processeurs

La Figure 2.4 montre la forme générale d'une architecture UMA où le système a une latence (temps d'accès) unique pour tous les processeurs et souvent les processeurs n'ont pas une horloge commune. Cette d'architecture est la plus populaire actuellement [40].

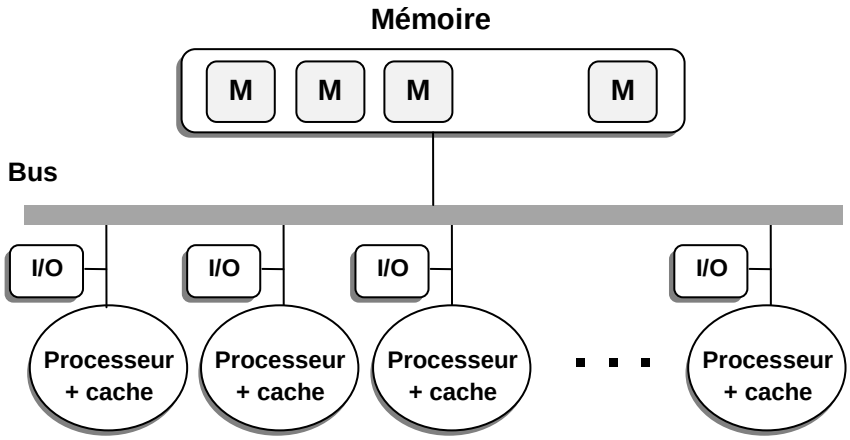

Figure 2.4 : L'Architecture UMA à base de bus

Les processeurs sont souvent de même type et ils sont installés dans le même boitier avec la mémoire partagée, la connexion entre les processeurs et la mémoire peut être un bus comme elle peut être un commutateur à barres croisées (pour des raisons d'efficacité).

2.2.2.3 Cache-Only Memory Architecture (COMA)

Cette architecture est semblable à l'architecture *NUMA*, sauf que l'espace mémoire est formé uniquement des mémoires caches, où chaque processeur possède une partie de cette mémoire cache partagée c.à.d. il n'existe aucune hiérarchie mémoire. Un système COMA exige que les données soient migrées vers le processeur demandant. Il y a un répertoire de cache *(D)* au niveau de chaque processeur qui aide dans l'accès à distance au cache. La Figure 2.5 affiche l'organisation du COMA.

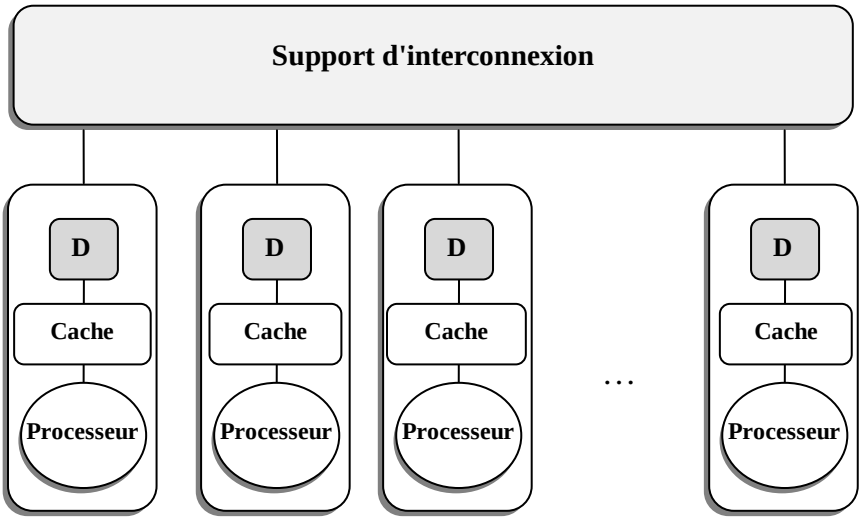

Figure 2.5: L'Architecture COMA

Deux problèmes principaux doivent être adressés dans les systèmes à mémoire partagée : la *dégradation de performance* due au conflit, et le *problème de cohérence*. La dégradation de performance pourrait se produire quand plus d'un processeur essaie d'accéder à la mémoire partagée simultanément. Un remède à ce problème consiste à employer des antémémoires (caches) pour résoudre le problème de contention. Cependant, avoir plusieurs copies des données

dispersées dans toutes les antémémoires pourrait mener à un problème de cohérence. Les copies dans les antémémoires sont cohérentes si elles sont toutes égales la même valeur. Cependant, si un processeur modifie sa copie, alors cette copie devient incohérente parce qu'elle n'égale plus la valeur des autres copies. On va discuter en plus de détails le problème de cohérence mémoire ultérieurement dans ce chapitre.

2.3 Espace de registres

2.3.1 Définition d'un Registre

Un *registre*[1] est un emplacement mémoire qui maintient une valeur binaire. Un registre de lecture/écriture (ou juste un registre), est un objet qui encapsule une valeur qui peut être observée par une méthode de lecture *read()* et peut être modifiée par une méthode d'écriture *write()*[53]. Les méthodes de lecture/écriture s'appellent souvent *load* et *Store*. Chaque registre implémente l'interface présentée dans la Figure 2.6 suivante.

```
1    public interface Register <T> {
2    T read();
3    void write(T  v);
4    }
```

Figure 2.6: L'interface d'un Registre <T>.

2.3.2 Types de registres

Les registres peuvent être classifiés en se basant sur les trois aspects suivants:

2.3.2.1 Suivant les valeurs autorisées

Chaque registre appartient à un type <*T*> qui spécifie le type des valeurs qui peut contenir. Ce type peut être un booléen, un entier d'ordre *M* ou une référence mémoire. Un registre booléen peut avoir uniquement deux valeurs ; *vraie* et *faux* (souvent on utilise *0* et *1*). Un registre de M-valeurs est un registre qui peut contenir un entier de taille max égale à M.

```
1    public class SequentialRegister<T> implements Register<T>{
2        private T value;
3         public T read() {
4         return value;
5        }
6        public void write(T v) {
7         value = v;
8       }
9    }
```

Figure 2.7: la classe SequentialRegister.

Si les appels des méthodes ne superposent pas, une implémentation d'un registre devrait se comporter comme il est indiqué dans la Figure 2.7. Cependant, ce n'est pas le cas dans une

[1] Appelé registre pour des raisons historiques.

architecture multiprocesseur, ou les méthodes peuvent superposer à tout moment. Donc il est nécessaire de spécifier ce que signifient les appels simultanés d'une méthode.

2.3.2.2 Suivant le nombre de lecteurs et rédacteurs

Un autre aspect très important dans l'implémentation des registres est le nombre de lecteurs et rédacteurs prévus. Il est clair qu'il est plus facile de mettre en application un registre *SRSW* (Single Reader Single Writer) qui supporte seulement un unique lecteur et un unique rédacteur qu'un registre *MRSW* (Multi Reader Single Writer) qui supporte des lecteurs multiples et un unique rédacteur ou un registre *MRMW* (Multi Reader Multi Writer) permettant des lecteurs multiples et des rédacteurs multiples.

2.3.2.3 Suivant le degré de consistance fournie

Le dernier aspect dans l'implémentation des registres est le degré de consistance qu'ils fournissent. Généralement il existe trois types:

1. Le registre atomique

Un registre *atomique* est une implémentation linearisable de la classe *SequentialRegister* affichée dans Figure 2.7. Un registre atomique se comporte exactement comme nous prévoyons; en d'autres termes chaque lecture d'un registre atomique retourne la dernière valeur écrite à ce registre. La communication entre les threads en effectuant des lectures et des écritures sur des registres atomiques était pendant longtemps le modèle standard du calcul concurrent [53]. Un registre atomique est de type MRMW.

2. Le registre sûr

Un registre MRSW est dit *sûr* (*safe*) [53] : (1) si une opération de lecture *Read()* qui ne superpose pas avec une opération d'écriture *Write()* retourne toujours la dernière valeur écrite par l'opération d'écriture la plus récente. (2) sinon, c.à.d. si une opération de lecture superpose avec une opération d'écriture, alors l'opération de lecture peut retourner n'importe valeur parmi les valeurs permises par le registre (par exemple de 0 à M-1 pour un registre M-valeurs).

Dans la Figure 2.8 suivante, si le registre est sûr, $R^1()$ retourne la valeur 0 car elle ne superpose pas avec une écriture, cependant $R^2()$ et $R^3()$ peuvent retourner n'importe valeur permise par le registre car elles superposent avec l'écriture *W(1)*.

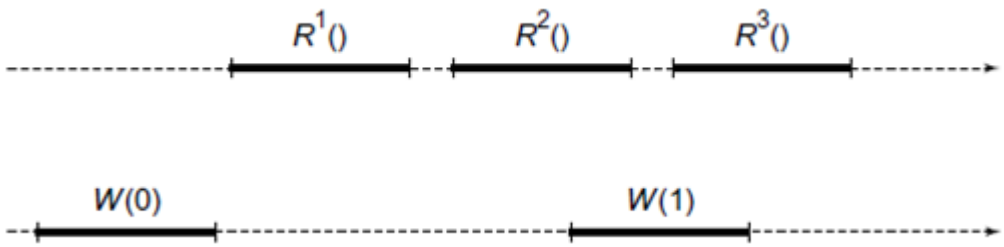

Figure 2.8: L'historique d'une exécution

3. Un registre régulier

Un registre *régulier* fournit un niveau de consistance intermédiaire entre le registre sûr et le registre atomique. Un registre régulier est un registre MRSW où les écritures ne se produisent pas d'une façon atomique [53]. En d'autre termes, quand un appel à l'opération *Write()* est en progrès, la valeur lue peut clignoter entre l'ancienne et la nouvelle valeur avant que l'ancienne

valeur soit remplacée définitivement par la nouvelle valeur. Dans l'exemple de la Figure 2.8, un registre régulier avec un lecteur unique doit se comporter comme suit : $R^1()$ retourne la plus ancienne valeur qui est 0. $R^2()$ et $R^3()$ peuvent retourner soit l'ancienne valeur 0, soit la nouvelle valeur 1.

Il est a noté que si l'exemple de la Figure 2.8 est exécuté sur un registre atomique de type SRSW, $R^1()$ retourne la plus ancienne valeur 0. Cependant, si $R^2()$ retourne 1 alors $R^3()$ doit retourner aussi 1 mais si $R^2()$ retourne un 0, $R^3()$ peut retourner 0 ou 1.

2.4 Consistance de la mémoire

Un modèle de consistance mémoire *(consistency model),* ou le modèle de mémoire *(memory model),* pour un multiprocesseur à mémoire partagée spécifie la vue logique de la mémoire partagée, en d'autres termes il spécifie comment la mémoire se comporte en ce qui concerne les opérations de lecture et d'écriture des différents processeurs. De point de vue du programmeur, un modèle mémoire permet un raisonnement correct sur les opérations mémoires dans un programme, et du point de vue des concepteurs système le modèle spécifie le comportement acceptable de la mémoire pour le système. Le modèle de consistance de mémoire influence beaucoup d'aspects de conception du système, y compris la conception des langages de programmation, des compilateurs, et du matériel fondamental [28].

2.4.1 Consistance Séquentielle

Une manière intuitive pour définir un modèle mémoire pour les multiprocesseurs est de se baser sur la sémantique séquentielle des opérations mémoire dans les monoprocesseurs et de voir les multiprocesseurs comme la multiprogrammation des monoprocesseurs. Lamport a définit formellement un tel modèle comme *consistance séquentielle* [55] :

" *… le résultat de n'importe exécution est identique au résultat obtenu si on a exécuté les opérations de tous les processeurs dans un certain ordre séquentiel, et les opérations de chaque processeur individuel apparaissent dans l'ordre spécifié par leur programme*"

La *consistance séquentielle* maintient le comportement de la mémoire qui est intuitivement prévu par la plupart des programmeurs [28]. La Figure 2.9 schématise une représentation conceptuelle de la consistance séquentielle. La mémoire est partagée entre plusieurs processeurs. Chaque processeur émet son opération mémoire dans l'ordre définit par le programme. Les opérations sont servies par la mémoire une par une et elles semblent se produire d'une façon atomique en ce qui concerne les autres opérations mémoires. L'ordre de service des opérations des différents processeurs peut être arbitraire ce qui mène à un ordre arbitraire des opérations mémoires à partir des différents processeurs dans un seul ordre séquentiel. Une exécution d'un programme est séquentiellement consistante s'il existe au moins une exécution sur un système séquentiellement consistant qui fournit le même résultat.

Les innovations dans les architectures et les compilateurs, telles que les mémoires tampons d'écriture et les caches, ont introduit plus de complexité en supportant la consistance séquentielle comme un modèle de choix pour les multiprocesseurs à mémoire partagée. Ces innovations ont mené à un travail étendu pour la spécification, la définition, et l'implémentation des différents modèles de consistance mémoire dans les multiprocesseurs à mémoire partagée modernes [28].

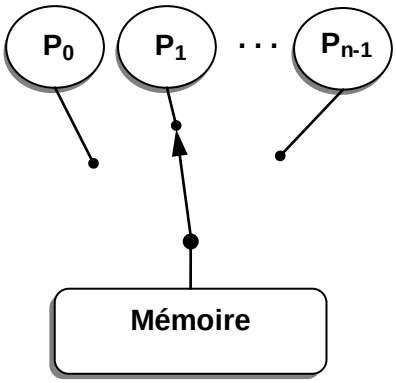

Figure 2.9: Vue conceptuelle de la consistance séquentielle

2.4.2 Modèles mémoire relaxés

Des modèles mémoire relaxés ont été proposés pour permettre l'utilisation de plus d'optimisations en relaxant les limitations sur l'ordre des opérations mémoires comme il est imposé par les modèles mémoire stricts tels que la consistance séquentielle. Contrairement à la consistance séquentielle qui exige l'illusion de l'ordre de programme et l'atomicité d'être maintenues pour toutes les opérations, les modèles mémoire relaxés permettent à certaines opérations mémoires d'être exécutées en dehors de l'ordre de programme et/ou d'une façon non atomique.

L'idée principale derrière les modèles de consistance relaxés est de permettre à des opérations de lecture et à des opérations d'écriture de se terminer sans respect de l'ordre de programme, mais en utilisant des opérations de synchronisation pour imposer l'ordre, de sorte qu'un programme synchronisé se comporte comme si le processeur était séquentiellement consistant. Il existe une variété de modèles relaxés qui sont classifiés selon comment l'ordre des lectures et d'écritures est relaxé.

L'ordre des opérations mémoires est spécifié par un ensemble des règles de la forme $X \rightarrow Y$, signifiant que l'opération X doit se terminer avant que l'opération Y soit faite. La consistance séquentielle exige le maintient de chacun des quatre ordres possibles : $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, *et* $W \rightarrow W$. les modèles relaxés sont définis suivant lesquels parmi ces quatre ensembles d'ordre ils relaxent [40]:

- relaxé l'ordre des $W \rightarrow R$ qui donne un modèle connu sous le nom d'*ordre totale des sauvegardes* (total store ordering) ou sous la *consistance de processeur* (processor consistency). Puisque cet ordre maintient l'ordre parmi les opérations d'écriture, plusieurs programmes qui fonctionnent sous la consistance séquentielle fonctionnent sous ce modèle, sans aucune synchronisation supplémentaire. La *consistance de processeur* est le modèle le moins strict car il exige que les opérations d'écriture doivent être observées dans le même ordre par tous les processeurs, ce qui permet aux opérations de lecture d'être exécuter avant les opérations d'écriture dans la file de service au niveau de la mémoire, en fournissant une meilleure performance [5].

- Relaxé l'ordre des $W \rightarrow W$ qui donne un modèle connu sous le nom d'*ordre partiel des sauvegardes* (partial store order).

- Relaxé l'ordre des $R \rightarrow W$ et des $R \rightarrow R$ qui donne une variété de modèles comprenant *l'ordre faible* (weak ordering), et la *consistance relâchée* (release consistency). La

cohérence faible sépare les accès mémoires ordinaires des accès de synchronisation et exige seulement que la mémoire devienne cohérente lors des accès de synchronisation. Ainsi, un accès de synchronisation doit attendre l'accomplissement de tous les accès précédents, alors que les accès ordinaires de lecture et d'écriture peuvent procéder tant qu'il n'y a aucun accès de synchronisation en suspension [5]. La *cohérence relâchée* est semblable à la cohérence faible mais elle identifie deux accès de synchronisation appelés *acquisition* et *libération*, avec des accès partagés protégés serrés entre eux. Dans ce modèle, les accès ordinaires de lecture et d'écriture peuvent procéder seulement quand il n'y a aucun accès d'acquisition suspendu du même processeur et un accès de libération doit attendre pour que toutes les lectures et écritures soient complétées [5].

En relaxant ces ordres, le processeur peut probablement obtenir des avantages de performance significatifs. Le modèle particulier de cohérence choisi affecte la conception et la performance du protocole de cohérence du cache exigé pour l'imposer [5]. La description des modèles de consistance relaxés introduit beaucoup de complexité, y compris les avantages et les inconvénients de relaxation des différents ordres, définir d'une manière précise ce que signifie pour qu'une écriture termine, et décider quand les processeurs peuvent voir les valeurs modifiées par un processeur. Pour plus d'informations sur les complexités, l'implémentation, et le potentiel d'exécution des modèles relaxés, il est recommandé de consulté l'excellent tutoriel de Adve et Gharachorloo [1].

2.5 Programmation de la mémoire partagée

La programmation parallèle de la mémoire partagée est peut être le modèle le plus facile à comprendre en raison de sa similitude avec la programmation des systèmes d'exploitation et la multiprogrammation d'une façon générale. La programmation de la mémoire partagée est faite en effectuant quelques extensions aux langages de programmation, aux systèmes d'exploitation, et aux bibliothèques de code existantes. Dans un programme parallèle de la mémoire partagée, il doit exister trois éléments de programmation principaux [20]: (1) la création des tâches, (2) la communication et (3) la synchronisation.

2.5.1 Création des tâches

Dans le cas d'une grande granularité, un système de mémoire partagé peut fournir le temps partagé traditionnel. Chaque fois qu'un nouveau processus est lancé, il sera exécuté dans les processeurs inactifs. Cependant, si le système est chargé (tous les processeurs sont occupés), le nouveau processus est assigné au processeur avec la moindre quantité de travail. Ces processus de grande granularité s'appellent souvent les *tâches lourdes* en raison de leur temps système élevé. Une tâche lourde dans un système multitâche comme l'UNIX se compose des tables de pages, de la mémoire, des descripteurs de fichier en plus le code du programme et les données.

Au niveau à grain fin, les threads *(processus légers)* rendent le parallélisme dans une seule application praticable. À ce niveau, une application est une série des constructions *fork-join* suivant les indications de la Figure 2.10. Un *fork* génère un nouvel thread d'exécution pour le processus concurrent. Quand tous les threads atteignent leur *join*, le traitement continue en mode séquentiel. Ce pattern de création de tâche s'appelle le modèle *superviseur-travailleurs*.

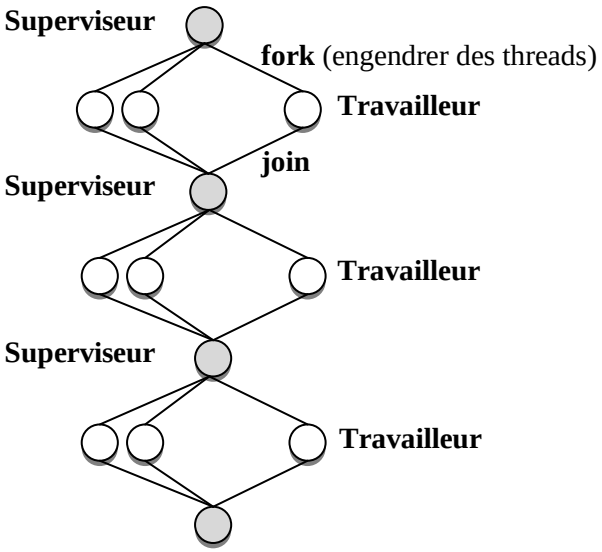

Figure 2.10: le modèle Superviseur-Travailleurs

utilisé dans la plupart des applications sur les systèmes à mémoire partagée [27].

2.5.2 La communication

Généralement l'espace d'adressage dans un processus en cours d'exécution est constitué de trois segments : segment de code, segment de données et la pile. Le segment de code est l'emplacement où le code binaire à exécuter est enregistré. Le segment de données contient les données du programme. La pile c'est l'emplacement où les enregistrements de lancement (activation) et les données dynamiques sont enregistrés. Les segments de données et de pile augmentent et se contractent pendant l'exécution du programme c'est pour cela un intervalle est laissé exprès entre ces deux segments.

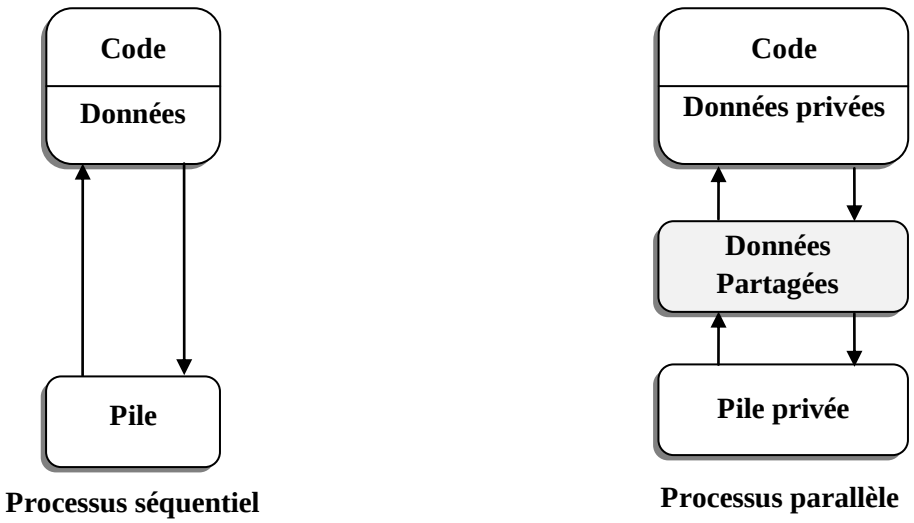

Figure 2.11: Processus séquentiel vs. Processus parallèle [27].

Dans le cas des processus séquentiels, chaque processus possède ses propres segments qui sont mutuellement indépendants. Le code de chaque processus est seulement autorisé d'accéder

aux segments de données et de pile propres à ce processus. Cependant, un processus parallèle est similaire à un processus séquentiel avec un segment supplémentaire qui est le *segment des données partagées*. Cette zone partagée est permise d'être étendue et elle est placée dans l'intervalle entre le segment de données privées et segment de pile. La Figure 2.11 montre la différence entre un processus séquentiel et un processus parallèle.

La communication entre les processus parallèles peut être réalisée en écrivant et en lisant des variables partagées dans les segments de données partagées suivant les indications de la Figure 2.12.

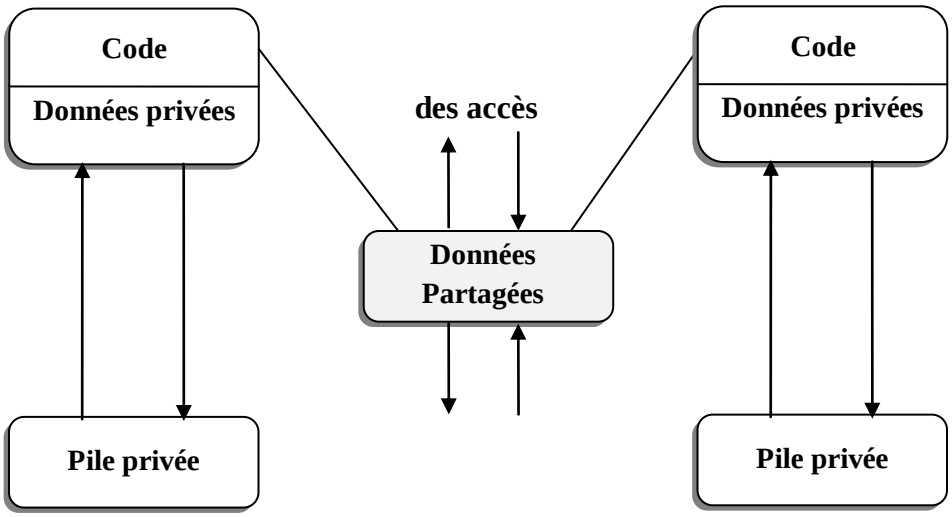

Figure 2.12: Communication entre les processus parallèles [27].

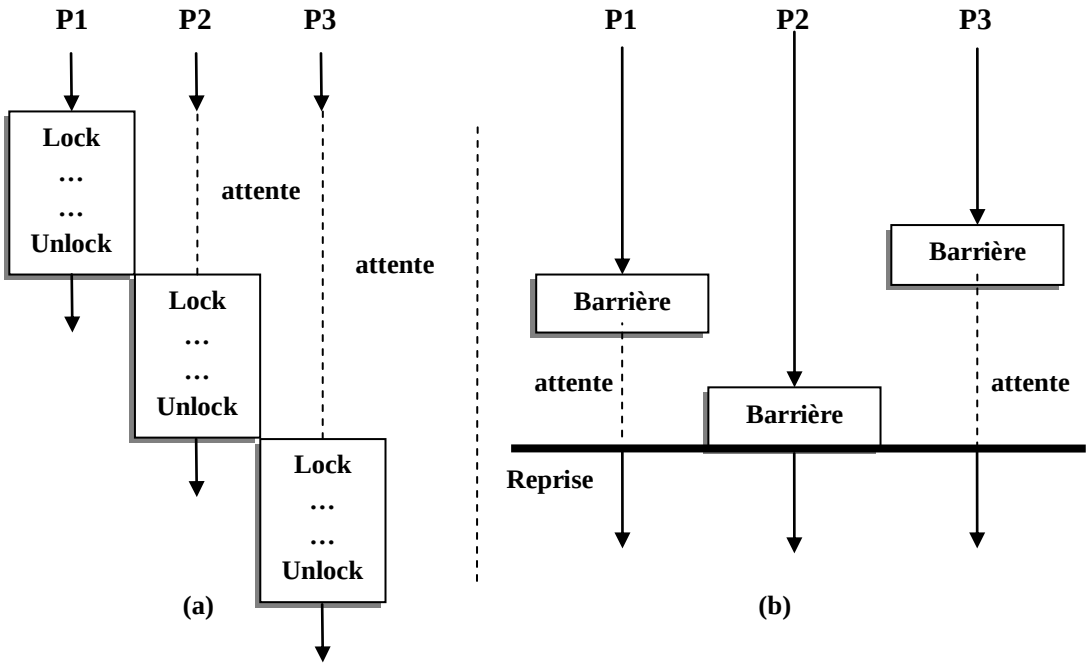

Figure 2.13: les verrous et les barrières.

2.5.3 La synchronisation

La synchronisation est nécessaire pour protéger les variables partagées en s'assurant qu'elles soient accédées uniquement par un processus à un moment donné (exclusion mutuelle). Les outils de synchronisation peuvent également être employés pour coordonner l'exécution des processus parallèles et pour les synchroniser à certains points dans l'exécution. Il existe deux outils principaux de synchronisation dans les systèmes à mémoire partagée [20]: (1) les verrous et (2) les barrières. La Figure 2.13 (a) montre l'exécution de trois processus parallèles en utilisant les verrous pour assurer l'exclusion mutuelle. Le processus P2 doit attendre jusqu'à ce que P1 déverrouille la section critique ; de même P3 doit attendre jusqu'à ce que P2 relâche le verrou.

Dans la Figure 2.13 (b), P3 et P1 atteignent leur bloc de barrière avant P2, et tous les deux doivent attendre jusqu'à ce que P2 atteigne sa barrière. Quand les trois processus atteignent le bloc de barrière, ils peuvent tous procéder en avant.

2.6 Problème de cohérence du cache

La performance d'un système multiprocesseur à mémoire partagée devient un problème quand le réseau d'interconnexion connectant les processeurs à la mémoire globale devient un goulot. La mise des données en mémoire cache locale est une technique très populaire pour remédier à ce goulot. Cependant, l'introduction des caches qui impliquent la création de plusieurs copies d'un même emplacement mémoire a créé un problème de *consistance* parmi les différents caches et entre la mémoire et les caches pour rendre les caches cohérents. Un système avec des caches cohérents est un système qui garantie que la valeur retournée par une opération de chargement *(load)* est toujours la dernière valeur générée par la dernière opération d'enregistrement *(store)* avec la même adresse [9].

Un cache est une mémoire de petite taille avec accès très rapide, chaque entrée dans le cache contient un groupe de mots voisins appelés *ligne* (bloc de cache). Un cache possède une certaine méthode pour mapper les adresses aux lignes. Considérant une architecture simple dans laquelle les processeurs et la mémoire communiquent à travers un support d'émission partagé appelé le bus. Chaque ligne de cache a une étiquette (*tag*), qui encode l'information d'état (statut) de cette ligne (Figure 2.14). Le statut d'une ligne suit le protocole standard MEPI (*En : MESI*), dans lequel chaque ligne du cache est identifiée par l'un des états suivants [53]:

- **Modifié :** la ligne dans le cache a été modifiée, et éventuellement elle doit par la suite être écrite à la mémoire. Cette ligne n'est pas cachée par aucun autre processeur.
- **Exclusivité :** la ligne n'a pas été modifiée, mais aucun autre processeur n'a cette ligne cachée. (une ligne est typiquement chargée dans le mode exclusif avant d'être modifiée).
- **partagé :** la ligne n'a pas été modifiée, et d'autres processeurs peuvent avoir cette ligne cachée.
- **Invalidé :** la ligne ne contient pas des données significatives.

Quand un processeur souhaite réaliser une lecture, son cache peut satisfaire cette lecture s'il contient la ligne requise et l'état relative à cette ligne est différent d'*invalidé*. Si le processeur souhaite réaliser une écriture, la ligne de cache correspondante doit être initialement dans l'état *modifié* ou *exclusif* et après l'écriture elle sera dans l'état *modifié*.

L'état d'une ligne de cache *X* doit être modifiée convenablement quand l'état d'une autre ligne équivalente d'un autre cache devient *modifié, exclusif* ou *partagé. X* peut devoir être écrite à la mémoire avant de laisser l'autre cache assumer la propriété de cette ligne.

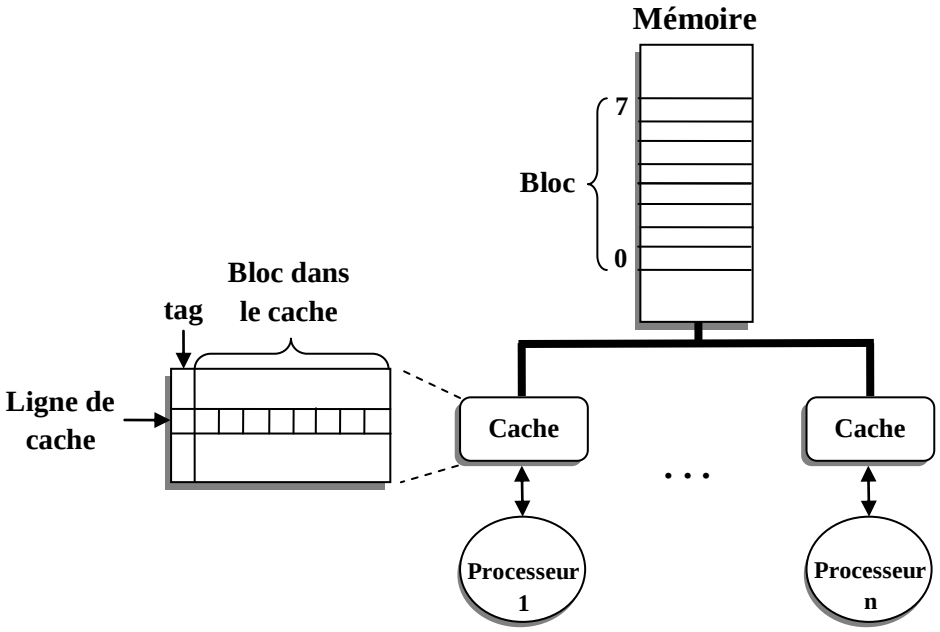

Figure 2.14: Structure du cache

2.6.1 Protocoles de cohérence du cache

Les protocoles de cohérence du cache peuvent être classés en deux catégories principales : protocoles basés sur le *répertoire* et les protocoles basés sur le *snoop* [40] :

2.6.1.1 Protocoles basés sur le répertoire

Dans ce type de protocoles, le statut partagé d'un bloc mémoire physique est maintenu uniquement dans un seul emplacement, appelé le *répertoire*. Cette technique a un surcoût légèrement plus élevé que le *snoop*, mais elle peut être utilisée avec un nombre élevé de processeurs. Un répertoire est une structure de données qui maintient l'information sur les processeurs qui partagent un bloc mémoire et sur son état. L'information maintenue dans le répertoire peut être centralisée ou distribuée. Un répertoire central maintient des informations sur tous les blocs dans une structure de données centrale. Dans ce cas-là, le répertoire central devient un goulot et souffre du grand temps de recherche car il inclut tous dans un seul emplacement. Pour alléger ce problème, la même information peut être traitée d'une manière distribuée en permettant à chaque module mémoire de maintenir un répertoire séparé. Dans un répertoire distribué, l'entrée liée à un bloc mémoire a seulement un pointeur vers le cache qui a demandé le bloc.

2.6.1.2 Protocoles basés sur le Snoop

Chaque cache qui a une copie d'un bloc mémoire physique possède également une copie du statut partagé de ce bloc, mais aucun état centralisé n'est gardé. Les caches sont tous accessibles par l'intermédiaire d'un certain support d'interconnexion (un bus ou un commutateur), et tous les

contrôleurs du cache surveillent *(snoop)* le bus pour déterminer s'ils ont une copie d'un bloc qui est requis sur un accès de bus.

Il existe deux méthodes pour maintenir la cohérence entre les caches et la mémoire. Une méthode consiste à s'assurer qu'un processeur a un accès exclusif à une donnée avant qu'il puisse écrire cette donnée. Ce protocole s'appelle l'*invalidation d'écriture* (write invalidate) car il invalide d'autres copies sur une écriture. C'est le protocole le plus commun pour les deux classes des protocoles *(snoop* et *répertoire).* L'accès exclusif assure qu'aucune autre copie ni en lecture ni en écriture n'existe pour la donnée quand l'opération d'écriture se produit. Toutes les autres copies cachées de la donnée sont invalidées. Par conséquent ce protocole force la sérialisation des opérations d'écriture c.à.d. Si deux processeurs simultanément veulent faire une opération d'écriture au même emplacement mémoire avec des valeurs différentes, la cohérence de cache s'assure qu'on observe les deux écritures dans le même ordre par tous les processeurs avec la même valeur persistante à toutes les copies.

L'autre méthode consiste à mettre à jour toutes les copies cachées d'une donnée quand cette donnée est écrite. Ce type de protocole s'appelle le protocole de *mise à jour d'écriture* (write update) ou de *diffusion d'écriture* (write broadcast). Ce protocole consomme considérablement plus de largeur de bande du support d'interconnexion car toutes les opérations d'écriture doivent être diffusées sur la ligne partagée par les caches. Pour cette raison, tous les multiprocesseurs récents ont choisi d'implémenter le protocole d'*invalidation d'écriture* [40]. Dans ce protocole les conflits de synchronisation sont détectés parmi les opérations *load* et *store* en s'assurant que les différents processeurs conviennent sur l'état de la mémoire partagée. Une description complète de ce protocole peut être complexe, mais voici les principales transitions qui nous intéressent :

- Quand un processeur demande de charger une ligne (bloc) dans le mode exclusif, les autres processeurs invalident toutes les copies de cette ligne. N'importe quel processeur avec une copie modifiée de cette ligne doit sauvegarder la ligne à la mémoire avant que le chargement puisse être accompli.
- Quand un processeur demande de charger une ligne dans son cache en mode partagé, n'importe quel processeur qui a une copie exclusive de cette ligne doit changer son état en partagé, et n'importe quel processeur avec une copie modifiée doit sauvegarder cette ligne à la mémoire avant que le chargement puisse être accompli.
- Si le cache devient plein, il peut être nécessaire d'*expulser* une ligne. Si la ligne est partagée ou exclusive, elle peut simplement être écrasée, mais si elle est modifiée, elle doit être sauvegardée à la mémoire.

Dans ce qui suit, on va voir comment adapter ce protocole pour supporter les transactions.

2.6.2 Cohérence du cache Transactionnel

Pour supporter les transactions, le protocole MEPI est gardé tel qu'il est sauf qu'on ajoute un bit supplémentaire à l'étiquette de chaque ligne de cache appelé le *bit transactionnel* [53]. Ce bit par défaut n'est pas utilisé. Cependant, Quand une valeur est placée dans le cache au nom d'une transaction, ce bit est utilisé pour indiquer que cette ligne contienne une *entrée transactionnelle*. On doit s'assurer seulement que les lignes transactionnelles modifiées ne doivent pas être écrites à la mémoire, et que l'invalidation d'une ligne transactionnelle doit interrompre la transaction. En d'autres termes :

- Si le protocole MEPI invalide une entrée transactionnelle, alors la transaction correspondante doit être interrompue. Une telle invalidation représente un conflit de synchronisation, entre deux *stores*, ou un *store* et un *load*.

- Si une ligne transactionnelle modifiée est invalidée ou elle est expulsée, alors sa valeur doit être jetée au lieu d'être écrite à la mémoire. Puisque n'importe quelle écriture transactionnelle est une tentative d'exécution, nous ne pouvons pas la laisser " s'échapper " tandis que la transaction est en activité. Au lieu de ça, on doit interrompre la transaction.

- Si le cache expulse une ligne transactionnelle, alors la transaction correspondante doit être interrompue, parce qu'une fois que la ligne n'est plus dans le cache, le protocole de cohérence du cache ne peut pas détecter les conflits de synchronisation.

- Quand une transaction termine, et aucune de ses lignes transactionnelles n'a été invalidée ou expulsée, alors elle peut commiter en effaçant les bits transactionnels dans ses lignes de cache. Si une invalidation ou une expulsion fait interrompre la transaction, ses lignes de cache transactionnelles sont invalidées. Ces règles s'assurent que le commit ou l'abort d'une transaction sont des étapes locales au processeur exécutant la transaction.

2.7 Techniques de synchronisation et contrôle de concurrence

Le *contrôle de concurrence* [71] est l'activité de coordonner les accès concurrents à un objet partagé ; c.à.d. de contrôler l'ordre d'exécution relatif des opérations contradictoires des différents threads. La *technique de synchronisation* [71] est l'algorithme utilisé pour réaliser une telle gestion des conflits d'accès (contrôle de concurrence).

2.7.1 Mécanismes à base de verrous

Dans les mécanismes à base de verrous, un thread acquiert un verrou protégeant une ressource partagée avant d'y accéder. Une fois le thread détient le verrou, il empêche d'autres threads d'accéder à la même ressource jusqu'à ce qu'il libère le verrou. Il existe un certain nombre de mécanismes à base de verrous, qui seront introduits brièvement.

- **Les verrous :** l'outil de synchronisation le plus connu jusqu'ici est le verrou [58]. Les verrous sont également connus comme verrous d'exclusion mutuelle ou verrous de *mutex*. Un thread acquiert un verrou avant d'accéder à une ressource partagée pour empêcher d'autres threads d'y accéder. Une fois le verrou est obtenu par un thread, les autres threads demandant le même verrou se bloquent jusqu'à ce que le verrou soit libéré par le thread qui le possède.

- **Les variables de condition :** les variables de condition permettent de faire attendre un thread (des threads) jusqu'à ce que la condition devienne vérifiée et du fait ils peuvent procéder. Généralement la condition peut être de type arbitraire tel que: *isEmpty==true, x==4.* En revanche, les verrous sont attachés à un seul état qui est la disponibilité du verrou.

- **Les moniteurs :** les moniteurs [50] sont des types de données abstraits c.à.d. une combinaison des structures de données et des opérations, qui permettent uniquement à un

seul thread à la fois d'être exécuté sur des ressources partagées. Cette fonction des moniteurs est semblable à celle des verrous. Les protections requises sont imposées par le compilateur en implémentant le moniteur. À la différence des verrous, les moniteurs peuvent éviter beaucoup de problèmes des verrous tels que l'association inconsistante verrou-ressource, les inter-blocages dus à l'échec de déverrouillage. Les moniteurs sont le mécanisme primaire de synchronisation utilisé dans le langage de programmation Java et qui sont implémentés par des variables de condition. Quand une fonction quitte, les moniteurs sont alors libérés automatiquement indépendamment du flux du programme, permettant à des exceptions d'être traitées dans des méthodes synchronisées de Java toute en évitant l'inter-blocage.

- **Les sémaphores:** les sémaphores sont introduits par Dijsktra [19] qui est le premier qui a décrit et résolu le problème de l'exclusion mutuelle. Les sémaphores ou parfois désignés sous le nom de compteur des sémaphores, sont des compteurs non négatifs. N'importe quel thread peut décrémenter le compteur pour verrouiller le sémaphore, mais en essayant de le décrémenter en dessous de zéro fait attendre le thread appelant qui doit attendre un autre thread pour le déverrouiller.

2.7.2 Les mécanismes non bloquants

Les mécanismes non bloquants sont une alternative aux mécanismes à base de verrous, ils sont développés pour éviter certains problèmes liés aux verrous [48]. Plusieurs propositions ont été faites dans la dernière décennie pour les mécanismes non bloquants. Dans les mécanismes à base de verrous traditionnels, un thread bloque une ressource partagée en détenant le verrou protégeant cette ressource. Les algorithmes non bloquants permettent à plusieurs threads de lire et d'écrire des données partagées simultanément sans blocage (en n'utilisant ni le verrouillage ni l'exclusion mutuelle).

Les algorithmes non bloquants utilisent les primitives atomiques non bloquantes, comme *compare-and-swap (CAS)* et le *Load Linked- Store Conditional (LL-SC).* L'oprération *CAS (a, o, n)* remplace d'une manière atomique le contenu de l'emplacement mémoire *a* par *n*, si l'emplacement contient la valeur *o,* Autrement l'emplacement est laissé inchangé. L'opération *LL-SC* originalement proposée par Jensen et autres [52] est composée de deux sous-opérations, le *LL (a)* qui renvoie la valeur de l'emplacement mémoire *a*. Un *SC (a, n)* ultérieur enregistre la valeur *n* dans l'emplacement *a*, si aucune opération d'écriture n'est effectuée sur l'emplacement *a* depuis l'opération *LL* précédente.

Dès la proposition de premier système de mémoire transactionnelle logicielle (STM) [79], on a largement accepté que les opérations atomiques non bloquantes étaient la seule voie pour construire des tels systèmes [65] et les implémentations non bloquantes ont dominé la recherche dans les STMs pendant plusieurs années. En fait, l'évaluation pratique a prouvé que les algorithmes non bloquants sont moins performants en comparaison avec leurs versions fondées sur les verrous les plus optimisés [46]. En 2005, le papier controversé de Robert Ennals [23] et les travaux postérieurs [22,18] ont indiqué que les systèmes STMs basés sur les verrous sont beaucoup plus rapides que les systèmes STMs non bloquants. Néanmoins, ils posent plus de défis pour les programmer en comparaison aux algorithmes bloquant conventionnels [43]. Jusqu'à présent, un nombre significatif de chercheurs dans le domaine de mémoire transactionnelle sont contre la conception des STMs en utilisant les verrous, à causes des problèmes inhérents qu'ils créent (voir la section 3.1.1.1).

2.8 Sûreté et Vivacité dans le contrôle de concurrence

Dans cette section on va discuter La sûreté et la vivacité qui ont été décrites pour la première fois par Lamport [56].

2.8.1 La sûreté

Lamport a défini formellement la sûreté *(safety)* en tant que "mauvaises choses ne se produisent pas" [56]. Une propriété de sûreté contraint les actions autorisées, et donc les modifications d'état permises à un programme c.à.d. les actions qu'un algorithme peut faire.

Généralement la spécification de la sûreté peut être n'importe quelle propriété de sûreté qui est maintenue pour une exécution si et seulement si elle est maintenue pour toutes les exécutions. L'exclusion mutuelle, la liberté d'inter-blocage, la serialisabilité, le traitement FIFO sont tous des propriétés de sûreté. Deux propriétés de sûreté qui nous intéressent ici sont la serialisabilité et l'absence d'inter-blocage qui seront discutées dans le chapitre suivant.

2.8.2 La vivacité

Lamport a défini formellement la vivacité *(liveness)* en tant que "quelque chose de *bonne* doit par la suite se produire pendant l'exécution" [56]. Les exemples des propriétés de vivacité incluent la liberté de famine, la terminaison, et le service garanti. La vivacité ne limite pas ce que peut être une "bonne chose". Les propriétés de vivacité ne peuvent pas stipuler qu'une certaine "*bonne chose*" se produit toujours, mais c'est uniquement éventuellement (dans un moment plus tard non spécifié) se produit [2]. Les deux propriétés de vivacité qui nous intéressent sont: l'absence de Livelock et l'absence de famine :

- **L'absence de livelock :** l'absence de livelock peut être paraphrasée comme : "Si un certain processus veut exécuter une transaction, un certain processus exécutera par la suite la transaction".
- **L'absence de la famine :** l'absence de famine peut être paraphrasée comme : " Si un certain processus veut exécuter une transaction, alors ce processus exécutera par la suite la transaction." Pour assurer la liberté de famine, tous les threads doivent par la suite réussir. Ceci est réalisé en utilisant un schéma approprié de résolution du conflit garantissant que tous les perdants de contention deviennent par la suite des gagnants.

2.9 Conclusion

Les systèmes distribués à mémoire partagée ont connu ces dernières années un essor remarquable notamment les systèmes multiprocesseurs ou multi-cœurs. Nous avons vu dans ce chapitre une présentation de ces systèmes et les différents concepts et notions que nous en aurons besoin pour comprendre le reste de ce mémoire. Le chapitre suivant va introduire le domaine de mémoire transactionnelle.

3

Concepts des STMs

Le domaine de mémoire transactionnelle logicielle à connu des intenses recherches ces dernières années et plusieurs concepts et constructions concernant ce domaine ont été définis, dans ce chapitre nous allons présenter tous ces concepts, les notions de base et les différents choix de conceptions et d'implémentation des STMs.

3.1 Techniques de synchronisation

3.1.1 La synchronisation bloquante

Les verrous d'exclusion mutuelle sont l'une des fondamentales abstractions les plus largement utilisés couramment pour réaliser la synchronisation. Cette popularité est quasiment due à leur modèle de programmation qui apparaît simple et à la disponibilité des implémentations qui sont efficaces et évolutives. Malheureusement, avec l'essor et l'évolution remarquable des multiprocesseurs et la nécessité de la programmation parallèle concurrente, les verrous sont mis en cause et leurs avantages sont difficilement maintenus même avec des experts de programmation pour des systèmes contenant plus qu'une poignée de verrous [25].

3.1.1.1 Problèmes de synchronisation avec les verrous conventionnels

Le verrouillage est la technique la plus utilisée pour assurer un accès mutuel aux données partagées, une entité demandant l'accès à une ressource partagée se bloque jusqu'à ce que cette ressource soit attribuée. Cependant, le verrouillage crée un goulot pour d'autres threads ou processus parallèles. D'autres processus doivent attendre jusqu'à ce que le thread qui détient le verrou termine son exécution. Cet état de blocage est la source de nombreux problèmes [41] :

3.1.1.1.1 Inversion de priorité

L'inversion de priorité a lieu quand un processus de priorité plus haute demande une ressource détenue par un processus de basse priorité, ce qui fait l'attente de processus de haute priorité jusqu'à ce que la ressource soit libérée.

3.1.1.1.2 Convoiement

Le problème de convoiement a lieu quand un processus détenant un verrou est déschedulé a cause des raisons différentes (par exemple si le processus a consommé son quantum de temps de traitement mais il n'est pas encore terminé son exécution ou peut être due à un défaut de page ou par un autre genre d'interruption) et en même temps, d'autres processus attendant dans la file d'attente pour acquérir le verrou ne pourront pas progresser en avant jusqu'à ce que ce processus relâche le verrou, par conséquence cela ralentira le traitement.

3.1.1.1.3 Inter-blocage

L'inter-blocage peut se produire si deux processus chacun détient une ressource et attend l'autre pour libérer la ressource qu'il détient, ou plus de deux processus attendent des ressources dans une chaîne circulaire. Donc comme résultat, aucun des processus impliqués dans l'inter-blocage ne peut progresser ni le système entier. Prenant l'exemple d'une application de base de données de type client-serveur. En supposant qu'une application A a acquis le verrou sur une table de base de données T1 et elle demande un accès exclusif à une autre table T2 qui est verrouillée par une autre application B et cette application attend l'application A pour libérer la table T1, qu'elle détient. De cette façon les deux applications A et B attendent l'une l'autre pour libérer le verrou sur la table qu'elle détient tandis qu'aucune d'elle ne le fait, ce qui mène à une situation d'inter-blocage.

3.1.1.1.4 Livelock

Le Livelock est semblable à l'inter-blocage sauf que les états des processus impliqués dans le Livelock se changent constamment en réponse aux modifications des autres processus. Donc les processus ne sont pas bloqués mais le système global ne progresse pas. Un bon exemple du Livelock peut être une boucle infinie. Il est analogue à l'inter-blocage du fait qu'aucun progrès réel n'est accompli mais il se diffère car aucun processus n'est bloqué en attendant une autre ressource. Un exemple de vie quotidienne du Livelock peut être la réunion de deux personnes dans le couloir. Tous les deux changent leur position pour permettre à l'autre personne de progresser, mais tous les deux ne peuvent pas réaliser aucun progrès réel parce que tous les deux déménagent le même côté en même temps.

3.1.1.1.5 Composition

La composition est la propriété de combiner des entités simples pour former une autre plus complexe. Par exemple, considérant une fonction qui supprime un élément d'une table et une autre fonction différente qui insère un élément dans une table. Dans la programmation séquentielle, on pourrait combiner les deux fonctions pour créer une fonction qui déplace un élément d'une table à une autre table. Ceci ne peut pas être fait avec les mécanismes de synchronisation explicites comme les verrous car ils ne sont pas *composables* [59].

3.1.1.1.6 Faible tolérance aux fautes

Si un processus détenant un verrou se termine en raison d'un défaut, d'autres processus attendant le verrou détenu par ce processus ne peuvent jamais se compléter comme le verrou ne sera jamais libre de nouveau. Ce problème est catastrophique dans un environnement orienté transaction où les processus sont largement indépendants tout en accédant à quelques structures critiques partagées. Les données modifiées dans la section critique sont laissées dans un état incohérent à cause d'une panne de l'application.

3.1.2 La synchronisation non bloquante

Les techniques de synchronisation non bloquantes ont été proposées comme une alternative de synchronisation pour éviter les problèmes de vivacité engendrées par les techniques bloquantes (les verrous traditionnels) discutés précédemment. Un système est non bloquant si la suspension ou la panne de n'importe threads n'empêche pas le reste du système de progresser en avant [25]. En d'autres termes, dans le mécanisme de verrouillage, si un processus détenant le verrou d'une ressource partagée est collé, alors les autres processus devront attendre jusqu'à ce que ce processus lâche le verrou, par conséquent, les mécanismes de verrouillage bloquent d'autres processus de progresser en avant. Les approches de synchronisation non bloquantes essaient d'éviter ce blocage (l'exclusion mutuelle).

L'utilisation des mécanismes non bloquants pour achever la synchronisation dans les systèmes parallèles mène à un débit élevé et une meilleure exécution en évitant les inter-blocages, les Livelocks et l'inversion de priorité et ceci est la raison pour le décalage des approches de synchronisation à base de verrous vers les approches non bloquantes dans les applications parallèles. Fondamentalement, les algorithmes de synchronisation non bloquants peuvent être classifiés en trois catégories suivant la garantie de progrès qu'ils achèvent ; *liberté de verrou* (Lock-freedom) [54], *liberté d'attente* (Wait-freedom) [48] et *liberté d'obstruction* (Obstruction-freedom) [4]. Dans cette section, nous décrivons ces techniques non bloquantes en plus de détail.

3.1.2.1 Liberté de verrou

La liberté de verrou [62] a été explorée la première fois par Lamport [54] et plus tard formalisée par Herlihy [46]. La Liberté de verrou s'assure que le système entier réalise un progrès malgré l'existence de contention. Donc plusieurs processus s'exécutent en même temps mais au moins un processus avance et termine son exécution dans un nombre fini du temps d'exécution et Les processus restants doivent attendre. La Liberté de verrou assure la prévention d'inter-blocage mais souffre de la famine. Dans la liberté de verrou, chaque processus essaie de compléter son exécution mais quand il identifie que les valeurs initiales ont été changées par un autre processus, il fait un retour arrière (roll-back) et commence son traitement de nouveau en se basant sur les nouvelles valeurs.

Il existe une méthode systématique qui peut transformer chaque objet concurrent en un objet libre de verrou [46]. D'abord, une copie de l'objet concurrent est créée et utilisée pour faire des modifications. La copie modifiée remplace la copié originale dans la structure de données en changeant le pointeur avec l'opération load-linked/store-conditional. Si le store-conditional échoue, l'opération est relancée car un autre thread doit avoir changé la structure de données après que la copie ait été créée. Les techniques libres de verrou garantissent un progrès vers l'avant puisqu'au moins un des threads essayant de modifier l'objet concurrent réussira.

3.1.2.2 Liberté d'attente

La liberté d'attente [62] a été proposée plus tard par Herlihy [48] pour avoir une garantie de progrès encore plus forte. Le Liberté d'attente est la propriété la plus forte d'un algorithme de synchronisation non bloquant en termes de garantie de progrès des processus concurrents. Cette propriété garantit que tous les processus contestant pour un ensemble commun d'objets concurrents accomplissent un progrès dans un nombre fini de leurs individuelles étapes de temps. La Liberté d'attente élimine l'occurrence des inter-blocages aussi bien que la famine.

Pour réaliser cette garantie de progrès plus forte, les objets libres d'attente sont plus complexes que les objets libres de verrou. Une méthodologie semblable à celle utilisée pour les objets libres de verrou peut être utilisée pour transformer un objet concurrent en un objet libre d'attente [46]. Le thread commence également par la création d'une copie modifiable en annonçant son envie de modification de l'objet. Chaque fois qu'un thread essaie d'apporter n'importe quelle modification, il réalise également toutes les modifications exceptionnelles annoncées par d'autres processeurs. Comme dans la synchronisation libre de verrou, au moins l'un des threads concurrents réussira. Le thread réussi, va également apporter les modifications pour les threads non réussis. Par conséquent, tous les threads seront réussis dans une quantité de temps finie.

3.1.2.3 Liberté d'obstruction

Dans un système concurrent, un algorithme de synchronisation non bloquant est dit libre d'obstruction [43] s'il garantit le progrès à un processus dans un nombre fini de ses propres étapes dont l'absence de contention d'accès à n'importe quel emplacement mémoire. C'est la propriété de garantie de progrès la plus faible d'un algorithme de synchronisation non bloquant. La Liberté d'obstruction élimine l'occurrence des inter-blocages, mais les Livelocks peuvent se produire [44]; si un ensemble de processus continuent à aborter ou interrompre les opérations atomiques de chacun et par conséquent aucun processus n'accomplit aucun progrès.

Pour éviter le Livelock, une technique de back-off est utilisée. D'ailleurs, un *gestionnaire de conflit* (contention manager) peut être utilisé pour permettre aux processus les plus prioritaires de s'exécuter tandis que les processus de priorité plus basse sont obstrués. Maurice Herlihy à illustré que les objets libres d'obstruction partagent plusieurs avantages avec les objets libres d'attente et les objets libres de verrou tout en étant plus facile à programmer. Malheureusement, il n'existe aucune transformation systématique qui peut être réalisée par le compilateur [43].

Quelques anciens travaux ont utilisé le terme *liberté de verrou* et le terme *non bloquant* comme des termes interchangeables. Cependant, dans ce mémoire nous suivrons l'utilisation récente de Herlihy et autres [43] en utilisant le terme *liberté de verrou* pour dénoter un genre particulier de garantie de progrès non bloquante.

3.1.3 Verrouillage et synchronisation Spéculatifs

Rajwar et Goodman [70] ont proposé une nouvelle approche pour réaliser la synchronisation dans les applications parallèles appelée *verrouillage spéculatif des données partagées* (Speculative Lock Elision -SLE). L'idée derrière SLE est que la sérialisation des threads due aux sections critiques est un goulot d'étranglement pour achever une haute performance dans les applications multithreads. Cependant, une telle sérialisation peut être non nécessaire car les sections critiques peuvent être exécutées d'une manière concurrente sans utilisation des verrous en respectant la sûreté. Malheureusement, les processeurs actuels ne peuvent pas exploiter un tel parallélisme car ils n'ont pas des mécanismes pour détecter dynamiquement un tel inter-thread indépendance.

SLE est une technique de synchronisation micro-architecturelle pour supprimer une telle sérialisation non nécessaire dans l'exécution de flux d'instruction dynamique. C'est une technique qui n'exige aucun changement ni dans le protocole de cohérence du cache, ni dans le jeu d'instruction. De plus elle ne nécessite aucun support de la part du programmeur ou du compilateur. Avec SLE, le matériel surveille dynamiquement les opérations de synchronisation et quand il détecte que la synchronisation est inutile alors elle sera éliminée. La détection de la synchronisation inutile est réalisée par le mécanisme de cohérence du cache. Cependant, si le matériel constatait que la synchronisation est exigée alors elle réalise une opération de reprise et explicitement acquiert le verrou.

L'exemple ci-dessous (Figure 3.1) montre un cas où les verrous sont non nécessaires. Dans cet exemple les deux threads travaillent sur la structure de données partagée Struct_Employee et ils modifient deux champs différents (Bonus et Rank) de cette structure, par conséquent le verrou n'est pas exigé pour synchroniser les opérations de modification. En outre, s'il y a un conflit en synchronisant l'opération parallèle (généralement pour garantir l'exactitude), l'opération est roulée en arrière et le verrou approprié est acquis pour compléter le processus. La détection de conflit et l'acquisition de verrou est faite à l'aide du mécanisme de cohérence du cache.

	Processus n° 1	Processus n° 2
1	**Lock** (employee_struct . lock)	**Lock** (employee_struct. lock)
2	Struct Struct_Employee;	Struct Struct_Employee;
3	**If** (Struct_Employee . Salary >1000)	**If** (Struct_Employee. Salary >1000)
4	{	{
5	Struct_Employee.	Struct_Employee. Rank="Manager";
6	Bonus=10/100*1000;	}
7	}	**Unlock** (employee_struct. lock)
	Unlock (employee_struct. lock)	

Figure 3.1: Extraction spéculative de verrou.

SLE a certaines similitudes avec la mémoire transactionnelle logicielle qui serait discutée dans les sections suivantes. Le problème surgit avec SLE quand les conflits de données ont lieu et les verrous sont acquis ce qui crée comme résultat un goulot d'étranglement et d'autres problèmes similaires qui sont associés au mécanisme de verrouillage classique.

3.2 Concepts de la mémoire Transactionnelle logicielle

Dans cette section, nous allons discuter les concepts de base utilisés dans le domaine de la mémoire transactionnelle logicielle. Une transaction est une séquence d'instructions exécutée d'une manière atomique et en isolation par rapport aux autres threads concurrents. L'atomicité garantit que l'état intermédiaire de la transaction est masqué par rapport aux autres threads. L'isolation garantit que les tâches s'exécutant de manière concurrente ne peuvent pas interférer avec la transaction, signifiant que le résultat de la transaction doit toujours être le même et ne doit pas dépendre des autres threads. Ces propriétés seront discutées en plus de détail plus tard dans ce chapitre.

3.1.1 Transactions dans les bases de données

Les avantages des transactions ont été connus dans la communauté de base de données et les transactions ont été adoptées comme une partie du modèle de programmation pour les bases de données pendant longtemps [31].

Les transactions offrent un modèle de programmation simple qui écarte du programmeur la grande partie de la difficulté de la programmation concurrente. Les programmeurs ne doivent pas s'inquiéter de l'inter-blocage, du livelock, de la cohérence de données, de l'atomicité, de l'inversion de priorité, ou du placement des verrous, en fait ils doivent à peine penser à la concurrence du tout [86].

Les transactions qui sont depuis longtemps une partie de programmation pour les bases de données ont maintenant rendu compte dans la programmation parallèle [59]. Dans les deux dernières décennies on a constaté qu'un modèle de programmation qui a été employé avec succès par les bases de données peut également être utilisé dans la programmation parallèle. Cependant, il existe quelques différences entre les transactions de base de données et les transactions mémoires.

3.2.1 Transactions mémoires vs. Transactions de bases de données

Dans cette section nous discuterons les différences et les similitudes entre les transactions de base de données et une transaction mémoire typique.

Les transactions de base de données sont habituellement très longues et peuvent durer tout à fait un long temps. Elles peuvent se composer des nombreuses instructions [11]. Cependant, une transaction mémoire typique est constituée habituellement d'un nombre peu élevé des instructions par conséquent elle s'exécuterait plus rapidement.

Dans une transaction de base de données, les données sont stockées sur les disques ce qui rend l'extraction de données lente et par conséquent le temps d'exécution de la transaction est plus long. En revanche, les transactions mémoires résident dans la mémoire centrale et le cache, par conséquent elles s'exécutent plus rapidement. De plus les transactions de base de données sont persistantes c.à.d. quand une transaction commite avec succès, ses modifications sont enregistrées sur disque cependant les transactions mémoires n'ont pas besoin d'être persistantes et ne comportent habituellement aucune E/S explicite vers le disque [47].

D'ailleurs les transactions de base de données sont souvent emboîtées alors que les transactions mémoires ne sont pas habituellement emboîtées [11]. Les transactions de base de données sont basées sur les propriétés ACID tandis que les transactions mémoire supportent la *linearisabilité* et *l'atomicité* [41]. Cependant, si la linearisabilité et l'atomicité sont réalisées alors la cohérence et l'isolement sont automatiquement atteints.

3.2.2 Critères d'exactitude

Il existe plusieurs critères d'exactitude dans la littérature adressant la propriété de *sûreté* des systèmes de mémoire transactionnelle. Ces critères incluent la linearisabilité, la serialisabilité et l'opacité.

3.2.2.1 Linearisabilité

Le concept de linearisabilité [49] dans le calcul parallèle a été introduit la première fois par Herlihy et Wing [49]. Ils ont dit qu'un calcul parallèle est *linearisable* si ses résultats sont égaux à sa version séquentielle. En d'autres termes si un calcul parallèle ou concurrent est exécuté séquentiellement et il fournit les mêmes résultats que le calcul concurrent, alors ce calcul a la propriété de linearisabilité.

Dans le calcul transactionnel, la linearisabilité est une propriété de sûreté dans laquelle chaque transaction devrait apparaître prendre effet instantanément pendant sa durée de vie c.à.d. à un certain moment *unique* entre son invocation et sa réponse.

Des travaux récents ont étendu la définition pour tenir compte les transactions interrompues. Généralement on parle du linearisabilité seulement si le résultat final d'une transaction est important. Cependant, puisque chaque transaction dans la mémoire transactionnelle est une partie interne d'une application, alors le résultat de chaque opération réalisée à l'intérieur d'une transaction est important et accessible par d'autres processus. Ainsi, la *serialisabilité* et ses dérivés sont des critères d'exactitude plus appropriés, comme recommandé par [49].

3.2.2.2 Serialisabilité

La serialisabilité est l'une des propriétés la plus généralement requise pas les transactions. L'exécution concurrente des transactions est *serialisable* si le résultat de cette exécution est

identique à une exécution séquentielle de ces transactions [59]. Le système est libre d'ordonner ou d'imbriquer les transactions mais il doit assurer que l'exécution reste serialisable.

En général, une histoire H des transactions (c.à.d. la séquence des opérations exécutées par toutes les transactions dans une exécution donnée) est dite serialisable si toutes les transactions commises dans H réalisent les mêmes opérations et reçoivent les mêmes réponses comme dans une certaine histoire séquentielle S qui est constituée seulement des transactions commises dans H [33]. (Intuitivement, une histoire séquentielle est une histoire où il n'est y a pas de concurrence entre les transactions).

La serialisabilité, comme il est exposé dans [33] n'est pas suffisante pour modeler une MT pour différentes raisons : (a) elle se base sur la supposition implicite qu'une opération de "lecture" d'un objet partagé X renvoie toujours la dernière valeur écrite à X ; (b) elle est limitée seulement aux opérations de lecture et d'écriture, et (c) elle n'indique aucune chose sur l'état accédé par les transactions courantes (ou interrompues). L'opacité est proposée pour adresser certains de ces points.

3.2.2.3 Opacité

L'Opacité [33] a été proposée récemment comme le critère d'exactitude le plus approprié pour les MT. Intuitivement, l'opacité garantit que les effets de chaque transaction commise apparaissent comme si elles sont produites à un certain point unique, indivisible pendant la durée de vie de la transaction. Les opérations réalisées par toutes les transactions abortées doivent ne jamais être visibles à d'autres transactions (comprenant celles vivantes). Chaque transaction observe toujours un état cohérent du système. Comme exposé par [33], la plupart des systèmes de mémoire transactionnelle tels que DSTM [44], ASTM [65], SXM [45], JVSTM [8], TL2 [16], LSA-STM [72] and RSTM [63] assurent l'Opacité.

3.2.3 Les propriétés ACI

Les transactions et leur sémantique associée ont été utilisées pendant longtemps dans les bases de données (par exemple voir [71]), les systèmes de mémoires transactionnelles logicielles sont basés sur ces idées. Les transactions de bases de données réelles ont les propriétés ACID cependant les systèmes STM ont seulement les propriétés ACI car les effets des transactions mémoires ne sont pas durables puisque les transactions effectuent seulement des opérations sur la mémoire centrale volatile. Dans ce qui suit, nous discuterons la signification de ces propriétés au sens des STMs.

3.2.3.1 Atomicité

L'atomicité (appelée aussi Failure atomicity [59]) s'assure qu'une transaction soit elle s'exécute complètement avec succès et commite ses résultats soit elle est entièrement annulée et n'a aucun effet c.à.d. considérée comme si elle n'a pas du tout s'exécutée. L'atomicité joue un rôle très important pour rendre le système d'exécution cohérent. En d'autres termes, si une transaction est interrompue, toutes les modifications apportées par cette transaction seront retournées en arrière.

3.2.3.2 Cohérence

La propriété de *cohérence* (Consistency) dans les STMs signifie que la mémoire et les structures de données sont dans un état valide après qu'une transaction commite ou aborte. En d'autres termes, chaque transaction commence son exécution à partir d'un état cohérent et laisse la mémoire après son exécution dans un état cohérent. La signification de la cohérence est entièrement dépendante de l'application. Elle se compose typiquement d'une collection des

invariants sur des structures de données. Par exemple, le champ NumClient dans une table Client ne doit pas contenir des entrées en double [59].

3.2.3.3 Isolation

L'isolation assure que les opérations exécutées par une transaction sur les données partagées n'ont jamais être visibles dans leurs états intermédiaires par une autre transaction. C.à.d. les modifications apportées par une transaction ne sont pas visibles à d'autres transactions jusqu'à ce que cette transaction commite. Elle exige que chaque transaction produise un résultat correct, indépendamment des autres transactions qui s'exécutent simultanément. C'est cette propriété qui fait évidemment aux transactions un modèle de programmation attrayant pour les ordinateurs parallèles [59].

3.3 Constructions et sémantiques des STMs

Dans cette section, nous discuterons les briques et les constructions de base de la mémoire transactionnelle logicielle et leurs détails d'utilisation.

3.3.1 Le bloc atomique

Une transaction mémoire est déclarée à l'intérieur d'un bloc appelé le *bloc atomique*. Ce bloc atomique définit les bornes d'une transaction mémoire [36]. La Figure 3.2 montre un exemple d'une transaction mémoire.

```
1    atomic {
2       if (x != null) x.foo();
3       y = true;
4    }
```

Figure 3.2: Exemple d'un bloc atomique

Une transaction s'exécute en respectant l'atomicité et la serialisabilité comme discuté précédemment. Ça signifié que La transaction mémoire expose ses résultats au reste du système (les autres transactions et threads) seulement après qu'elle commite avec succès c.à.d. après identification qu'aucun conflit d'accès aux données n'est détecté.

3.3.2 Le bloc Retry

Dans un système de mémoire transactionnelle logicielle, plusieurs transactions peuvent s'exécuter simultanément et il n'y a aucun ordre spécifique dans lequel les transactions devraient mettre à jour leurs résultats. Dans les transactions mémoires, si une transaction met à jour le résultat (commite), d'autres transactions conflictuelles doivent relancer et re-exécuter de nouveau en se basant sur les nouvelles valeurs (mises à jour). Donc, dans ce contexte il devrait y avoir un certain mécanisme qui pourrait re-exécuter une transaction. Harris et autres [37] ont proposé le bloc Retry pour synchroniser l'exécution des transactions. Un bloc Retry interrompt une transaction et puis l'exécute de nouveau. La Figure 3.3 affiche un exemple d'utilisation du bloc Retry.

Harris a proposé de retarder la réexécution jusqu'à ce que le système détecte des changements d'une ou de plusieurs valeurs lues par la transaction lors de son exécution précédente pour que le résultat de la réexécution soit différent. Cependant, les systèmes pratiques

peuvent avoir besoin d'un mécanisme pour limiter le nombre des essaies car une transaction ne peut pas maintenir le nombre de fois où elle réessaie par elle-même (chaque abort roule en arrière le compteur).

```
1    atomic {
2      if (buffer.isEmpty()) retry;
3      Object x = buffer.getElement();
4    ...
5    }
```

Figure 3.3 : Exemple d'un bloc Retry

3.3.3 Le bloc orElse

Harris et autres [37] ont également introduit l'opération orElse qui est un autre mécanisme pour coordonner l'exécution de deux transactions. Par exemple, si A et B sont deux transactions, le bloc orElse peut être très utile afin de les coordonner. La Figure 3.4 montre l'utilisation du bloc orElse.

```
1    atomic {      //Transaction A
2    {
3       x = Q1.getElement(); }
4      orElse      //Transaction B
5      {
6       x = Q2.getElement(); }
7    }
```

Figure 3.4 : Exemple d'un bloc orElse

Dans l'exemple de la Figure 3.4 d'abord la Transaction A est exécutée et si elle réessaie elle sera interrompue et le contrôle passera à la Transaction B. Si la Transaction B de sa part échoue et réessaie alors le bloc atomique en entier sera exécuté de nouveau à partir du début [37]. En d'autres termes :

- Si A commite ou explicitement aborte (à cause d'un bloc d'arrêt explicite ou une exception), le bloc d'orElse se termine sans exécuter le B.
- Si A exécute un bloc Retry, alors l'opération d'orElse commence à exécuter le B.
- Si B commite ou explicitement aborte, le bloc d'orElse termine l'exécution.
- Si B exécute un bloc Retry, le bloc d'orElse attend jusqu'à ce qu'un emplacement lu par l'une des deux transactions se change, et puis ré-exécute de nouveau de la même manière.

3.4 Espace de conception des STMs

Dans cette section nous présenterons les différents choix de conception associés aux mémoires transactionnelles logicielles et leurs influences sur les systèmes STMs. Plus tôt dans ce chapitre (section 3.2.2 et section 3.2.3) nous avons discuté les propriétés de base d'une transaction mémoire.

3.4.1 Isolement faible et Isolement fort

L'isolation comme elle est discutée dans la section 3.2.3.3 implique que l'exécution d'une transaction ne doit pas affecter d'autres transactions. Cependant, l'isolement est classé en un *isolement fort* (appelé aussi atomicité forte) et un *isolement faible* (appelé aussi atomicité faible) [7]. Dans un STM il peut y avoir deux types d'opérations ; les opérations transactionnelles et opérations non transactionnelle où les opérations non transactionnelles peuvent avoir un effet négatif sur les opérations transactionnelles ce qui peut mener aux *races* et aux incohérences de données.

3.4.1.1 L'isolement fort

L'isolement fort garantit les sémantiques transactionnelles entre les transactions elles mêmes et entre les transactions et le code non transactionnel. L'isolement fort convertit automatiquement toutes les opérations en dehors d'un bloc atomique en des opérations transactionnelles individuelles, en reproduisant le modèle de base de données dans lequel tous les accès aux ressources partagées s'exécutent dans des transactions.

3.4.1.2 L'isolement faible

L'isolement faible garantit les sémantiques transactionnelles uniquement entre les transactions. Avec l'isolement faible une référence mémoire contradictoire exécutée à l'extérieure d'une transaction ne suit pas le protocole de STM par conséquent la référence peut renvoyer une valeur incohérente ou perturber l'exécution correcte de la transaction. L'isolement faible est souvent compris que tous les accès aux données partagées se produisent à l'intérieur d'une transaction. Cependant, ça n'est pas juste et non plus nécessaire car un programme doit seulement s'assurer que les accès non-transactionnels ne sont pas en conflit avec les accès transactionnels en s'assurant que les accès ne superposent pas ni dans le temps ni dans l'emplacement mémoire accédé. Pour un programme sans races de données, l'isolement faible et fort sont équivalents.

3.4.2 Transactions emboîtées

Une transaction *emboîtée* (Nested) est une transaction qui est constituée d'une ou plusieurs transactions. Le comportement de la transaction interne avec la transaction externe peut être associé l'une avec l'autre de plusieurs manières. Initialement, la transaction externe lance la transaction interne, et selon le décalage de contrôle entre les deux transactions, les transactions emboîtées sont divisées en plusieurs catégories :

3.4.2.1 Transaction aplatie

Dans une transaction aplatie (*Flattened*), l'interruption de la transaction interne mène à l'interruption de la transaction externe, mais le commit de la transaction interne n'a aucun effet jusqu'à ce que la transaction externe commite, et à ce moment là, les modifications apportées par la transaction interne deviennent visibles à d'autres threads. La transaction externe voit les modifications apportées à l'état du programme par la transaction interne. Les transactions aplaties sont simples à implémenter mais elles peuvent diminuer la performance globale du système puisque elles font interrompre la transaction externe [67].

Dans l'exemple de la Figure 3.5 suivante, si la transaction interne est aplatie, la variable x est restée égale à 1 quand la transaction externe commite.

```
1    int x = 1;
2    atomic {
3        x = 2;
4        atomic Nested {
5          x = 3;
6          abort;
7        }
8    }
```

Figure 3.5: Exemple 1 de transactions emboitées.

Il existe deux autres catégories qui sont différentes de la transaction aplatie ; les transactions fermées et ouvertes [59].

3.4.2.2 Transaction fermée

Dans une transaction *fermée* (Closed), l'abort de la transaction interne ne mène par à la terminaison de la transaction externe. Si une transaction interne commite ou s'interrompt alors le contrôle est passé à la transaction externe. Dans le cas où la transaction interne commite, ses mises à jour sont visibles à la transaction externe (aux transactions environnantes). Cependant, une transaction non-environnante peut seulement voir les modifications quand la transaction la plus externe commite également ses modifications avec succès.

Dans l'exemple 1 précédant, si la transaction interne est fermée, la variable x est égal à 2 quand la transaction externe commite parce que l'affectation de la transaction interne est défaite par l'*Abort*.

3.4.2.3 Transaction ouverte

Dans les transactions *ouvertes* (Open), les modifications commises par la transaction interne deviennent visibles et demeurent permanentes à toutes les transactions courantes dans le système même avant le commit de la transaction père (externe) qu'elle peut même s'échouer et s'aborter.

Dans l'exemple de la Figure 3.6 suivante, même après que la transaction externe aborte, la variable x est laissé avec la valeur 3 :

```
1    int x = 1;
2    atomic {
3        x = 2;
4        atomic Open {
5          x = 3;
6         }
7       abort;
        }
```

Figure 3.6: Exemple 2 de transactions emboitées.

Les transactions emboîtées aplaties menacent réellement la propriété d'isolement des transactions. L'isolement implique que si une transaction s'échoue ou s'interrompre elle ne doit pas affecter d'autres transactions. De même, dans les transactions ouvertes les modifications apportées par les transactions internes ne peuvent pas être visibles par les transactions externes et vice versa du fait qu'il y a une violation de la propriété d'isolement qui souligne que les données

de la transaction interne ne peuvent pas être visibles à d'autres transactions à moins que la transaction commite complètement avec succès. Cependant, les transactions ouvertes permettent à une transaction d'apporter des modifications permanentes à l'état du programme, qui ne seront pas roulées en arrière si une transaction environnante s'interrompt. En outre, elles permettent à un code indépendant, tel qu'un *ramasse-miettes* (garbage collector) de s'exécuter à l'intérieur d'une transaction et d'apporter des modifications permanentes qui restent inchangées par les transactions environnantes. Ces utilisations apportées par les transactions ouvertes peuvent améliorer la performance des programmes [59].

3.4.3 Traitement des exceptions dans les transactions

Les exceptions sont conçues pour empêcher le système logiciel entier de tomber en panne. Le traitement d'exception est également une partie importante d'un système STM. Quand l'exception se produit, le système essaie de commiter la transaction en sauvegardant ses résultats et puis la faire quitter le système ou elle peut quitter le système sans que ses résultats soient sauvegardés. De plus la transaction peut s'exécuter de nouveau ou elle peut quitter définitivement. Cependant, sauvegarder des résultats incohérents peut être une source des défauts, donc l'approche la plus appropriée peut être quand l'exception se produit, la transaction s'aborte et s'exécute de nouveau.

3.4.4 Granularité de transaction

La granularité de transaction se rapporte à l'espace mémoire sur lequel le système STM détecte les conflits (les accès contradictoires aux données) [59]. La granularité peut être implémentée au niveau *objet* dans laquelle un accès contradictoire à un objet est détecté même si les transactions ont accédées à des champs différents de cet objet, au niveau *mot* dans laquelle le système détecte le conflit des accès à un mot mémoire, et au niveau *bloc* dans laquelle le système détecte le conflit des accès à un groupe adjacent de taille fixe des mots mémoire. Tous ces niveaux se rapportent à la quantité de données de la ressource partagée sur laquelle le système STM détecte les conflits.

Un système STM maintient les *métadonnées* de chaque objet en cours d'exécution afin de le gérer, et éviter les accès contradictoires possibles (maintenir la cohérence) [59]. Il existe deux possibilités pour maintenir les métadonnées ; d'abord les métadonnées devraient faire partie de chaque objet dans le niveau de *granularité objet* c.à.d. il suffit d'étendre l'objet avec un champ supplémentaire ce qui facilite l'accès aux métadonnées. Cependant, dans le cas d'une granularité de niveau *mot* ou *bloc*, les métadonnées peuvent être maintenues dans une structure de données séparée (une table auxiliaire) ce qui permet un partage plus précis et augmente les accès concurrents par les transactions car les structures de données Agrégées (comme les tableaux) peuvent être partagées logiquement [59]. Cependant, garder une structure de données séparée et mapper entre les adresses mémoires et cette table des métadonnées peut être coûteux. La granularité de niveau objet est plus compréhensible par le programmeur que la granularité de niveau bloc car les objets sont plus visibles au programmeur que des blocs mémoires.

3.4.5 Stratégies de mise à jour de données

Quand une transaction se termine avec succès (commite), elle met à jour les valeurs initiales avec les nouvelles valeurs. En se basant sur la stratégie de mise à jour utilisée, on peut distinguer deux approches, la *mise à jour directe* et *mise à jour différée* [59].

Dans La mise à jour directe (eager update), la transaction travaille directement avec les valeurs originales et fait les modifications sur ces valeurs. Cependant, le système maintient une

copie des valeurs initiales pour qu'il puisse les restaurer (fait un roll back) en cas d'échec de la transaction. Pour cela le système STM utilise un *log* (espace mémoire réservé à cet usage) des activités. Le coût d'aborter une transaction est élevé dans le cas de mise à jour directe car les valeurs originales doivent être restituées à partir du log et chaque transaction qui lit une valeur modifiée doit faire le retour-arrière.

Une autre approche pour mettre à jour les valeurs est la mise à jour différée (lazy update). Dans cette approche la transaction en exécution maintient une copie séparée (privée) des valeurs à mettre à jour et quand elle termine avec succès son exécution, elle copie toutes les valeurs à partir de la copie provisoire vers les emplacements mémoire originaux. Par contre lorsqu'elle aborte elle efface ces valeurs provisoires. Cependant, dans cette approche, le maintient d'une copie séparée des variables est un autre surcoût de plus le temps nécessaire pour copier les emplacements provisoires aux emplacements originaux lors de commit [59].

La mise à jour différée peut augmenter l'espace mémoire utilisé par un programme en créant des copies privées de chaque objet modifié par la transaction. D'une autre part, la mise à jour directe doit enregistrer la valeur initiale des objets modifiés. L'enregistrement peut exiger moins de mémoire si des objets sont partiellement modifiés.

Les premières implémentations des STMs ont utilisé la mise à jour différée, mais certains systèmes récents utilisent la mise à jour directe, qui semble être plus efficace. La mise à jour directe élimine un ou deux références mémoire par chaque lecture ou écriture [59]. En outre, elle peut réduire la quantité de travail requise pour commiter une transaction, en éliminant la nécessité de copier les copies privées vers l'emplacement original.

3.4.6 Contrôle de la Concurrence

Un autre aspect dans la conception des STMs est la gestion des conflits d'accès qui est une tâche très importante d'un STM afin de faciliter l'exécution concurrente des processus. Le contrôle de concurrence est également important pour les systèmes à mise à jour directe aussi bien que pour ceux à mise à jour différée.

Quand deux transactions essaient d'accéder au même emplacement mémoire partagé, avec au moins une transaction qu'elle doit modifier cet emplacement, alors un conflit surgisse [78]. Par conséquence, afin de résoudre ce conflit l'une des transactions doit attendre ou elle doit interrompre son traitement. La gestion des conflits d'accès est toujours basée sur trois événements, qui se produisent dans l'ordre. Tout d'abord, un conflit se produit, le système STM détecte ce conflit et dans la troisième étape il résout le conflit. Fondamentalement il y a deux catégories de gestion des conflits d'accès, la gestion des conflits d'accès *pessimiste* et la gestion des conflits d'accès *optimiste* [59].

La *gestion des conflits d'accès pessimiste* (appelée aussi eager) est basée sur le fait que tous les trois événements c.à.d. l'*occurrence de conflit, la détection de conflit* et la *résolution du conflit* aient lieu dans le même point d'exécution. Ceci signifie que dès qu'un conflit surgira il est détecté par le système STM et résolu. La gestion des conflits d'accès pessimiste accorde l'accès exclusif à un objet partagé à une seule transaction. Cet accès exclusif peut être acquis tout en interrompant d'autres transactions.

La *gestion des conflits d'accès optimiste* (appelée aussi Lazy) suppose que la *détection* et la *résolution de conflit* ont lieu après que le conflit surgisse. Dans ce type de gestion des conflits d'accès, plusieurs transactions ont la permission d'accéder à une ressource partagée simultanément. La gestion des conflits d'accès optimiste détecte et résout le conflit quand une

transaction essaie de commiter ses résultats. La résolution du conflit est faite en interrompant les transactions contradictoires ou en les mettant dans la file d'attente dépendamment de la décision du gestionnaire de conflit que nous le discuterons après.

L'approche pessimiste de détection de conflit peut interrompre les transactions qui pourraient avoir commiter avec L'approche *optimiste*, mais L'approche *optimiste* écrase trop de calcul, parce que des transactions sont interrompues plus tard [47]. Donc L'approche *optimiste* de gestion de conflits d'accès convient quand le conflit n'est pas fréquent ce qui donne comme résultat un taux de commit élevé.

3.4.7 Race de données

Dans une exécution concurrente, les données partagées peuvent être accédées par des transactions *(accès transactionnels)* comme elles peuvent être accédées par d'autres threads *(accès non transactionnels)*. Quand ces accès sont contradictoires ils engendrent ce qui s'appelle une *race de données* [59]. Le comportement d'un programme avec une race de données est imprévisible et indéterminé. Les races de données peuvent être prévenues en utilisant le verrouillage ou d'autres formes de synchronisation pour empêcher les accès concurrents.

Un autre aspect important dans le contrôle de concurrence est l'approche de synchronisation utilisée (bloquante ou non bloquante) qui est déjà discutée en détail plus haut dans ce chapitre (section 3.1).

3.4.8 Schémas de détection de Conflit

Dans un système de mémoire transactionnelle logicielle, la détection de conflit est le problème de détermination quand deux transactions ne peuvent pas toutes les deux commiter en respectant la sûreté [80]. La détection de conflit peut être faite dans des étapes différentes pendant l'exécution d'une transaction et chaque approche a ses avantages et ses inconvénients relatifs et ses influences potentielles sur la performance globale d'un STM.

Toutes les approches qui détectent le conflit avant la phase le commit sont classées dans la catégorie de *détection de conflit tôt* (early conflict detection). Tandis que celles qui détectent le conflit dans la phase de commit sont classées dans la catégorie de *détection conflit en retard* (late conflict detection). D'une façon générale Il existe les trois approches suivantes comme discutées par James Larus et Ravi Rajwar dans [59] :

3.4.8.1 La détection de conflit pendant l'ouverture

Dans cette approche, un conflit peut être détecté quand une transaction essaie d'accéder à un objet (emplacement mémoire) ou quand elle essaie d'acquérir le contrôle sur cet objet.

3.4.8.2 Détection de conflit pendant la validation

Dans cette méthode la transaction vérifie régulièrement si un objet de l'ensemble des objets lus ou modifiés par cette transaction est ouvert par une autre transaction pour une modification.

3.4.8.3 Détection de conflit pendant le Commit

Un autre point où un conflit peut être détecté est en commit. Quand une transaction essaie de commiter ses résultats elle peut (souvent doit) valider les valeurs qu'elle a lu ou modifié pour voir s'il y a des conflits avec d'autres transactions.

L'étude faite dans [80] a montré que la stratégie de détection de conflit peut affecter d'une manière significative la performance globale d'un système de mémoire transactionnelle

logicielle. Les auteurs ont proposé une nouvelle stratégie de détection de conflit appelée l'invalidation mixte.

3.4.8.4 L'invalidation mixte

Dans une stratégie de détection de conflit mixte, les conflits de type écriture/écriture sont détectés tôt (car c'est au plus une des transactions conflictuelles peut commiter) mais les conflits de type lecture/écriture sont détectés tard (car tous les deux peuvent commiter si la transaction de lecture commite en premier lieu).

La détection tôt de conflit réduit la quantité de calcul qui sera perdu en interrompant une transaction. D'une part, il y a des situations où une transaction pourrait avoir compléter son traitement si le conflit n'avait pas été détecté plus tôt. Considérant l'exemple où une transaction T1 et une transaction T2 sont toutes les deux en conflit avec la transaction T3 sur deux objets différents. Donc T1 s'interrompt puisqu'elle est en conflit avec T3. De même T3 s'interrompt parce qu'elle est en conflit avec T2. Cependant, dans cette situation T1 avait pu compléter son exécution car T3 a été interrompue. Donc avec la détection de conflit en retard les transactions T1 et T2 pourraient compléter leur traitement avec succès sans s'interrompre. Cependant, la détection de conflit en retard maximise la quantité de calcul perdu quand une transaction aborte.

3.4.9 Gestions des versions des données

La détection des conflits d'accès exige à un STM d'avoir un mécanisme qui lui permet de maintenir une association entre les transactions et les objets (emplacements mémoire) qu'elles accèdent. Une manière de faire c'est que chaque transaction garde une copie des emplacements mémoire qu'elle accède et lors de la phase de validation elle compare les valeurs privées avec les valeurs réelles. Cependant, cette solution ne peut pas détecter le problème ABA (Un objet O a la valeur A lorsque la transaction T1 a lu O, après une autre transaction T2 change O à la valeur B et commite et par la suite une troisième transaction T3 change O à la valeur A de nouveau et commite. Donc T1 commite et valide son ensemble de lecture du fait que O ait la valeur A malgré qu'elle a lu deux valeur incohérentes).

L'autre approche qui évite le problème ABA consiste à donner un numéro de version à chaque objet. Le numéro de version est fondamentalement un compteur qui est incrémenté sur chaque modification de l'objet. Pour détecter les conflits dans ce cas, la transaction maintient les numéros de version des objets accédés pour les comparer pendant le commit avec les numéros de version réels des objets en déterminant que les objets sont modifiés ou pas.

3.4.10 Gestionnaire de conflit

Afin de gérer les conflits entre les transactions exécutées dans un STM, le système STM devrait avoir un *gestionnaire de conflit* (Contention Manager). Le gestionnaire de conflit a pour tâche de résoudre les conflits entre deux transactions (ou plus) contradictoires en interrompant une transaction ou en la mettant dans une file d'attente. Un gestionnaire de conflit *dynamique* devrait incorporer plusieurs politiques de gestion de conflit pour des situations différentes afin de réagir efficacement. De plus un gestionnaire de conflit idéal devrait assurer le progrès vers l'avant c.à.d. Les transactions devraient s'exécuter et le système doit procéder en avant et il ne devrait pas être coincé dans un état où il ne peut pas procéder plus loin. Les études empiriques ont constaté que le choix de l'algorithme de gestion de conflit peut affecter la performance d'un STM d'une façon significative [55,61].

Il existe plusieurs politiques de gestion de conflit dans la littérature, la plupart d'eux ont été discutées en détail par William Scherer et Michael Scott dans [77]. Ici nous discuterons les gestionnaires de conflit les plus performants, en se basant sur leur travail.

3.4.10.1 Gestionnaire Polit

Le gestionnaire de conflit *Polit* (Polite manager) utilise le back-off exponentiel pour une quantité de temps, appelée temps de rotation (spinning time) pour résoudre le conflit. Le temps de rotation est calculé par la formule 2^{n+k} ns (voir la Figure 3.7). Le gestionnaire polit, compte le nombre de fois, qu'une transaction avait essayé d'accéder à un objet. Après au maximum *m* ronds de rotation tout en essayant d'accéder au même objet, le gestionnaire *polit*, arrête toutes les transactions contradictoires et donne l'accès à la transaction concurrente.

Temps de rotation = 2^{n+k} ns où :
n = nombre de fois où la transaction a essayée d'accéder à un objet.
K = constante dépendante de l'architecture.
m = le nombre maximum des ronds de rotation.
Valeurs idéales: m = 22, and k = 4

Figure 3.7: La formule de Back-off exponentiel.

3.4.10.2 Gestionnaire Karma

Le gestionnaire *karma* résout les conflits parmi les transactions conflictuelles en se basant sur la quantité des données traitées par une transaction particulière. Il préfère d'interrompre une transaction concurrente qui a juste commencé son traitement qu'une transaction qui est dans sa phase finale d'exécution. Cependant, estimer la quantité de données traitées par une transaction est une tâche difficile.

Le gestionnaire karma utilise le nombre des objets ouverts (consultés ou acquis) par une transaction comme mesure des données traitées. Ce nombre est considéré comme une priorité pour les transactions et il est maintenu dans un compteur avec chaque transaction. Si la transaction commite alors ce compteur est réinitialisé à zéro. Cependant, quand une transaction aborte, ce compteur n'est pas changé, ce qui donne une haute priorité à cette transaction pour compléter son traitement la prochaine fois.

Le gestionnaire karma résout le conflit en interrompant une transaction ennemie (contradictoire) quand le nombre de fois où une transaction a essayé d'ouvrir un objet dépasse la différence dans les priorités entre cette transaction et son ennemi. Quand le nombre de tentatives d'ouverture ne dépasse pas cette différence de priorités, le karma diminue la permission d'interrompre l'ennemie et faire un retour arrière pour une période du temps fixe pour cette dernière.

3.4.10.3 Gestionnaire Polka

Le gestionnaire *Polka* est une combinaison des deux gestionnaires précédents qui fusionne leurs meilleurs caractéristiques d'où la nomination Polka (**Pol**it et **ka**rma), il combine le back off exponentiel randomisé du gestionnaire polit avec le mécanisme d'accumulation de priorité du gestionnaire karma. Polka fait un back off pour un nombre qui appartient à l'intervalle de priorité (égal à la différence dans les priorités entre la transaction et son ennemi).

3.4.10.4 Gestionnaire Kindergarten

Le gestionnaire *Kindergarten* est basé sur la stratégie de résolution du conflit pour le problème de diner des philosophes [10]. Dans ce scénario, toutes les transactions essaient d'accéder à l'objet partagé. Chaque transaction maintient une liste de coups (hit list) (initialement vide) des transactions ennemies qui contient l'identité des transactions en faveur desquelles un thread s'est précédemment interrompu. Lors d'un conflit le gestionnaire Kindergarten arrête la transaction contradictoire si elle est dans la liste de coups de cette transaction. Si elle n'est pas dans la liste de coups, alors on l'ajoute à cette liste, et le gestionnaire Kindergarten fait un retour arrière pour un nombre limité des intervalles de temps fixes pour donner une chance à l'autre transaction pour terminer son exécution. Si la transaction ne peut pas encore procéder en avant, alors le gestionnaire Kindergarten interrompt sa propre transaction et la relance.

3.4.10.5 Gestionnaire de l'estampille

Le gestionnaire de l'estampille (Timestamp manager) essaie d'être aussi juste que possible avec les transactions. Il enregistre l'estampille de départ de chaque transaction et lors d'un conflit il interrompt les transactions ennemies qui ont une estampille plus nouvelle. Autrement, le gestionnaire de l'estampille marque un flag sur la transaction ennemie, en la considérant en tant qu'une transaction morte et attend pour un intervalle de temps fixe, à la fin de cette intervalle il contrôle le flag, s'il est resté marqué alors il abort cette transaction. Cependant, les transactions actives effacent leur flag.

3.4.10.6 Gestionnaire avide

Le gestionnaire avide [17] (Greedy) assigne à chaque transaction une seule estampille monotoniquement croissante à son début. La transaction avec l'estampille la plus inférieure gagne toujours. Une propriété importante d'avide est que, à la différence d'autres gestionnaires de conflit, il évite la famine des transactions. Il est plus performant pour les charges de travail de grande puissance.

3.4.10.7 Gestionnaire Serializer

Le Serializer [17] est très semblable à avide sauf qu'il assigne une nouvelle estampille à une transaction sur chaque relance, ce qui ne lui permet pas d'empêcher la famine ou même les livelocks des transactions.

3.5 Conclusion

Le domaine des mémoires transactionnelles a connu l'introduction de plusieurs concepts et notions. Dans ce chapitre nous venons de voir ces concepts en détail et les différents axes de conception des systèmes de mémoires transactionnelles logicielles, nous aurons besoin de ces concepts et axes dans le chapitre suivant pour comprendre les implémentations des STMs.

4

Implémentations des STMs

Dans ce chapitre nous présenterons des différentes implémentations des systèmes des mémoires transactionnelles logicielles, tout en discutant leurs choix de conception et d'implémentation. Initialement, la mémoire transactionnelle a été supportée seulement en matériel, mais en 1995, Shavit et Touitou ont introduit la mémoire transactionnelle logicielle, par la suite plusieurs systèmes STMs ont été introduits. La mémoire transactionnelle logicielle a beaucoup d'avantages par rapport aux systèmes de mémoire transactionnelle basés sur le matériel [59] :

- STM est facile à mettre en application, et il est plus flexible et divers.
- STM est plus évolutif et facile à modifier qu'une approche matérielle.
- STM peut être fait comme partie d'un langage de programmation ce qui augmente son utilisation.
- STM a moins de limitations internes en comparaison avec l'approche matérielle, par exemple, la taille limitée du cache.

Il existe actuellement plusieurs implémentations de mémoire transactionnelle logicielle. Dans ce mémoire nous nous concentrerons sur les implémentations logicielles (et certaines hybrides) des systèmes de mémoire transactionnelle mentionnées dans le Tableau 4.1.

4.1 Software Transactional Memory (STM)

Le système STM proposé par Shavit et Touitou [79] était le premier système de mémoire transactionnelle logicielle décrit dans la littérature, il identifie et essaie d'obtenir l'accès à tous les emplacements mémoire nécessaires pour l'exécution d'une transaction particulière. Chaque transaction doit définir à l'avance tous les emplacements mémoire dont elle aura besoin pour son exécution. L'unité mémoire de base sur laquelle ce système est basé est le *mot*. En d'autres termes, la granularité des transactions est au niveau mot. Les caractéristiques fondamentales du STM de Shavit et Touitou sont affichées dans le Tableau 4.1.1.

Tableau 4.1 : Brève Historique des STMs étudiés.

N°	Année	Auteurs	Système	Synchronisation
1	**1995**	Shavit et Touitou	STM	Liberté de verrous
2	**2003**	Fraser, Harris	WSTM	Liberté de verrous
3	**2003**	Herlihy, Luchangco, Moir et Scherer	DSTM	Liberté d'obstruction
5	**2003**	Fraser	OSTM	Liberté de verrous
4	**2003**	Ennals	ESTM	À base de verrous
6	**2006**	Dave Dice, Ori Shalev, Nir Shavit	TL2	À base de verrous
7	**2006**	Herlihy et autres	DSTM 2	Liberté d'obstruction
8	**2006**	Marathe et autres	RSTM	Liberté d'obstruction
9	**2006**	Riegel et autres	TbSTM	Liberté d'obstruction
10	**2006**	Kumar et autres	HybridTM	Hybrid TM
11	**2006**	Moir et autres	HyTM	Hybrid TM
12	**2006**	Saha et autres	McRT-STM	À base de verrous
13	**2006**	Tim Harris et autres	BSTM	À base de verrous
14	**2007**	Faud et autres	NZTM	Hybrid TM
15	**2007**	Mark Moir et autres	PhTM	Hybrid TM
16	**2007**	Gottschlich et Connors	DRACO-STM	À base de verrous
17	**2008**	Virendra J. Marathe et Mark Moir	NBSTM	Liberté d'obstruction
18	**2008**	Felber, Fetzer et Riegel	TinySTM	À base de verrous
19	**2009**	Dragojevi´c et autres	SuissTM	À base de verrous

4.1.1 Détails d'implémentation

Une transaction est représentée par une structure de données qui maintient les données nécessaires pour l'exécution de la transaction et l'information sur son statut (Figure 4.1.1). Le champ *addr* contient les adresses des emplacements mémoire accédés par la transaction et qui sont triées dans un ordre croissant pour éviter les inter-blocages et assurer l'exactitude de cet algorithme. Le champ *oldValues* enregistre le contenu des emplacements mémoire quand la transaction a acquis leur propriété. Les champs *version*, *status* et *executing* maintiennent l'état de la transaction.

Tableau 4.1.1 : les caractéristiques de base de STM.

STM	
Synchronisation	Liberté de verrous
Contrôle de concurrence	Pessimiste
Granularité	Mot
Stratégie de mise à jour	Directe
Détection de conflit	Tôt
Stratégie de résolution de conflit	Aide
Transactions emboitées	Non Supportées

Quand une transaction détient le contrôle sur un mot mémoire, elle devient le propriétaire de ce mot et donc elle est la seule capable de faire une modification sur ce mot. L'information de propriété est enregistrée séparément près des données réelles suivant les indications de la Figure 4.1.1.

L'enregistrement de propriété contient soit l'adresse du propriétaire actuel soit une valeur nulle qu'elle indique qu'aucune transaction n'est le propriétaire de la donnée. Bien que chaque transaction puisse accéder aux données partagées pour une consultation, mais uniquement une seule transaction à la fois détient un bloc mémoire ce qui implique que seulement cette transaction puisse faire des modifications sur ce bloc. La propriété d'un mot mémoire est acquise en utilisant le mécanisme de verrouillage à deux phases, l'acquisition des emplacements mémoire se fait dans un ordre croissant pour éviter la possibilité d'inter-blocage.

Si une transaction a échoué d'obtenir la propriété d'un emplacement mémoire, elle aborte et libère tous les emplacements mémoire qu'elle a déjà acquis. Si une transaction parvient à prendre tous les emplacements mémoire désirées alors elle termine son exécution et met à jour les résultats sans possibilité de retour-arrière. Ceci implique que le système utilise l'approche de *mise à jour directe.*

Le système assure un accès non bloquant avec une garantie de progrès vers l'avant. Même si quelques transactions s'échouent et s'arrêtent alors au moins une transaction parviendra à terminer son exécution après un nombre fini de tentatives. Le contrôle de concurrence est pessimiste c.à.d. le système garantit que l'occurrence de conflit, sa détection et les événements de résolution aient lieu dans le même point d'exécution, en d'autres termes dès qu'un conflit surgisse il est détecté par le système STM et résolu. Le système a un mécanisme de détection de conflit tôt et utilise une stratégie appelée *Aide* pour la résolution du conflit, qui implique que si une transaction ne puisse pas procéder plus loin (échec dans l'acquisition de propriété d'un emplacement mémoire) à cause d'un conflit avec d'autre transaction, elle devrait s'interrompre et aider la transaction contradictoire (détenant la propriété) pour terminer son exécution dans la supposition que le thread exécutant cette dernière à été déschedulé.

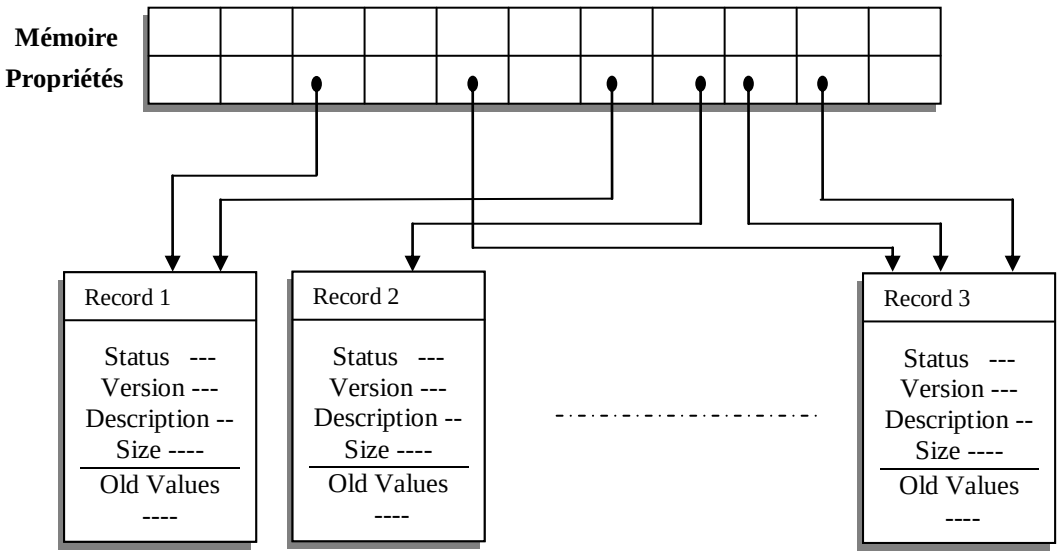

Figure 4.1.1: Modèle de Mémoire Partagée de STM de Shavit et Touitou [79].

4.1.2 Discussion

L'un des inconvenants majeurs du système STM de Shavit et Touitou est le mécanisme d'*aide* dans la résolution des conflits. Cette technique de résolution du conflit est basée sur le fait que si une transaction ne puisse pas procéder en avant en raison d'un certain conflit avec une autre transaction, elle devrait *aider* l'autre transaction pour compléter son traitement. Dans ce cas-la, il peut y avoir deux threads qui exécutent la même transaction. Ceci signifie que si la transaction *X* constate qu'elle est contradictoire avec la transaction *Y* alors la transaction *X* fait la mise à jour au nom de la transaction *Y*.

Shavit et Touitou ont initialement adopté le concept d'*aide* de Greg Barnes [4], qui a introduit le concept d'*aide récursive* dans laquelle une transaction étant aidée peut elle même aider une autre transaction. D'ailleurs, l'*aide consistante* peut détériorer l'exécution par des conflits inutiles [79]. Cependant, dans le STM de Shavit, l'aide est limitée seulement à un niveau spécifique; malgré ça elle mène à un grand niveau de complexité car elle expose les données à un ou plusieurs threads [39]. Cependant, les stratégies postérieures de résolution du conflit, par exemple le *vol (stealing)* de Harris et Fraser [35] ont surmonté les inconvénients de la stratégie d'*aide*. On va discuter cette technique dans la section 4.2.

Une autre limitation de ce système est la déclaration en avance des emplacements mémoire qu'une transaction ait besoin. Ceci limite l'acquisition des emplacements mémoire dynamiquement par la transaction. Cependant, les STMs récents ont surmonté cette limitation comme nous allons voir en permettant aux transactions d'acquérir les emplacements mémoire dynamiquement.

Un autre inconvénient trouvé dans le STM de Shavit et Touitou est le maintient de l'information de propriété dans des emplacements mémoire séparés près des données. C'est une stratégie peu efficace pour la sauvegarde des enregistrements de propriété. Considérant l'exemple où une donnée est étendue sur deux mots mémoire ou plus avec les mêmes enregistrements de propriété redondants. Ceci rend les besoins de stockage pour un élément de données doubles et redondants suivant les indications de la Figure 4.1.1.

4.2 Word based Software Transactional Memory (WSTM)

Harris et Fraser dans [35] en 2003 ont introduit un premier système STM qui est une partie intégré dans un langage de programmation orientée objets (Java). WSTM détecte les conflits au niveau mot (Word) d'où il vient le nom WSTM. Les caractéristiques de conception de base de WSTM sont affichées dans le Tableau 4.2.1.

Tableau 4.2.1 : les caractéristiques de base de WSTM.

WSTM	
Synchronisation	Liberté d'Obstruction
Contrôle de concurrence	Optimiste
Granularité	Mot
Stratégie de mise à jour	Différée
Détection de conflit	Tard
Stratégie de résolution de conflit	Aide ou abort
Transactions emboitées	Aplaties

Le WSTM assure un isolement faible avec un mécanisme de mise à jour différée. D'ailleurs, il supporte des transactions emboîtées *aplaties*. D'autres caractéristiques incluent une gestion de concurrence d'accès optimiste et une synchronisation non bloquante avec une liberté d'obstruction.

WSTM a le mécanisme de détection de conflit en retard. La résolution du conflit est réalisée par le mécanisme de *vol (stealing)* [35]. La stratégie de *vol* permet à une transaction de prendre la propriété des emplacements mémoire d'une transaction contradictoire en effectuant un CAS atomique. Cependant, cette stratégie assure que l'état logique des emplacements mémoire ne doive pas changer dans l'opération de vol. Après que la nouvelle transaction ait pris la propriété de l'emplacement mémoire, on veille que la nouvelle transaction propriétaire commite avant l'ancienne transaction propriétaire.

```
1    public int get_Item(int index)
2    {
3        atomic (index != 0)
4        {
5            index --;
6            return buffer[index];
7        }
8    }
```

Figure 4.2.1: La région critique conditionnelle [35].

WSTM n'exige pas à une transaction de déclarer en avance les emplacements mémoire dont elle aura besoin contrairement au système STM de Shavit et Touitou. WSTM est

fondamentalement inspiré à partir des régions critiques conditionnelles de Hoare *(RCCs)* [51] et les blocs atomiques utilisant les *RCCs* de Lomet [61]. Dans une *RCC*, un programmeur peut protéger une région de code particulière par une condition booléenne. La Figure 4.2.1 affiche un exemple d'une RCC. Dans cette figure le bloc atomique est protégé par une condition sur un index entier qui est différent de zéro. Le contrôle n'entrera pas la région critique conditionnelle que si l'index soit différent de zéro.

Harris et Fraser, ont construit leur système en améliorant le travail de Lomet en présentant un système qui empêche l'inter-blocage avec le mécanisme de verrouillage à deux phases, de plus il n'exige pas une déclaration à l'avance des variables conditionnelles pour le contrôle de synchronisation [35]. Contrairement à Lomet, WSTM a facilité le fonctionnement des RCCs sur la base de l'état du programme au lieu des variables spécifiques. WSTM fait une pause d'exécution d'une transaction jusqu'à ce que l'une des variables de condition soit modifiée par une autre transaction [35]. WSTM peut supporter des langages procéduraux aussi bien qu'il ait été implémenté en tant qu'une partie d'un langage orienté objet (Java). WSTM a introduit un nouvel bloc dans Java qui permet de déclarer un bloc de code atomique avec un gestionnaire de RCC approprié. La Figure 4.2.2 affiche la structure d'un bloc atomique de WSTM.

Une chose importante à noter dans le bloc atomique présenté sur cette figure, c'est qu'il commite la transaction quand l'exception aura lieu. Cependant, habituellement les exceptions sont seulement utilisées pour traiter les erreurs et empêcher le système entier de tomber en panne. L'idée principale derrière la mise en place du bloc de commit à l'intérieure de l'exception est de sauvegarder le traitement fait jusqu'ici dans la transaction. D'ailleurs, dans des transactions emboîtées aplaties (où l'interruption de la transaction interne cause l'interruption de la transaction externe), si la transaction interne s'échoue, alors afin de sauvegarder le travail réalisé par la transaction externe, le commit à l'intérieur de l'exception peut être très utile [35].

```
1   boolean done = false;
2   while (!done)
3   {
4       STMStart();
5       try
6       { if (<condition>)
7         {
8           <statements>;
9           done = STMCommit();
10        }
11        else {
12                STMWait();
13              }
14      }
15    catch (Exception t)
16    {
17      done = STMCommit();
18      if (done) { throw t; }
19    } }
```

Figure 4.2.2: La Structure d'un bloc atomique [35].

La bibliothèque de WSTM implémente les méthodes de base suivantes [35] :

1. ***void*** *STMStart()*
2. ***void*** *STMAbort()*
3. ***boolean*** *STMCommit()*
4. ***boolean*** *STMValidate()*
5. ***void*** *STMWait()*

WSTM utilise deux structures de données auxiliaires près de la structure de données principale c.à.d. le segment de données où les données sont maintenues. Une de ces structures de données est le *descripteur de transaction* qui est unique pour chaque transaction. Le descripteur garde le statut actuel de la transaction et dépiste les accès aux données faites part la transaction. Lors de démarrage d'une transaction son statut est mis à l'état *ACTIVE*, pendant l'exécution la transaction peut appeler les opérations *STMRead*, *STMWrite* pour faire des opérations de lecture et d'écriture. L'exécution de *STMWait* change le statut de transaction vers *ASLEEP*. Les méthodes *STMAbort* et *STMCommit* sont appelées pour aborter (changer le statut de la transaction à *ABORTED)* et commiter (changer le statut de la transaction à *COMMITTED*) une transaction respectivement. Les accès mémoire réalisés par une transaction sont dépistés dans une structure de données appelée *Transaction Entry* qui garde les adresses des différentes cases mémoires accédées par la transaction (les anciennes valeurs avec leurs versions et les nouvelles valeurs leurs versions) comme il est indiqué dans la Figure 4.2.3. Les numéros de version sont très utiles dans la gestion de conflit. Le descripteur contient un champ qui s'appelle *NestingDepth* qui sauvegarde le nombre de transactions (aplaties) partageant ce descripteur et qu'elles doivent commiter ou aborter en même temps.

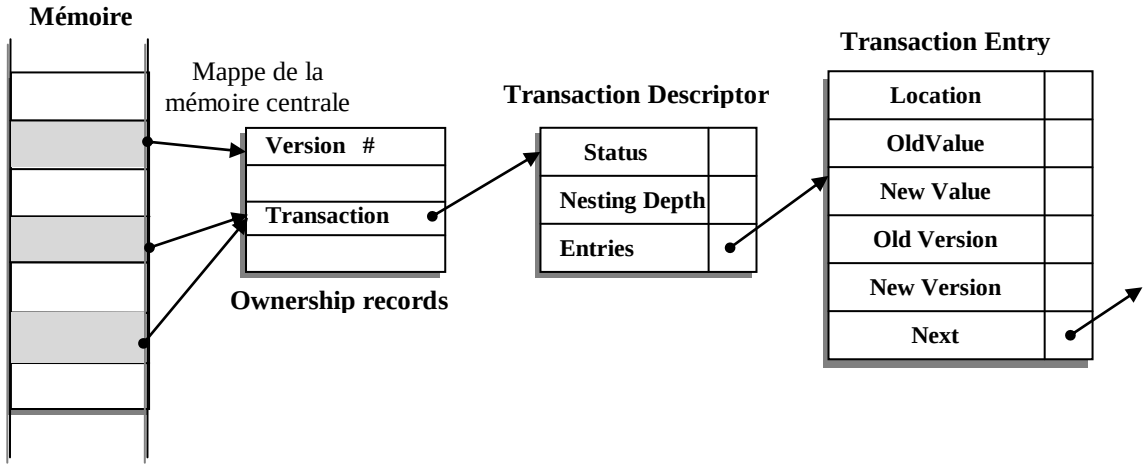

Figure 4.2.3: Structure de données de WSTM [35].

L'autre structure de données est une collection d'enregistrements de propriété *orecs* (ownership records). Ces enregistrements se servent pour deux rôles. Premièrement, un *orec* enregistre le numéro de version de l'emplacement mémoire produit par la dernière (la plus récente) transaction qui a commis en faisant une mise à jour de cet emplacement. Deuxièmement, quand une transaction est en cours de commit, l'*orec* garde un pointeur vers la transaction qui a acquis la propriété exclusive sur l'emplacement mémoire. En d'autres termes, chaque enregistrement de propriété contient soit un numéro de version soit un pointeur vers la transaction qui possède le contrôle sur l'emplacement.

4.2.1 Détails d'implémentation

L'opération *STMStart* est utilisée pour réserver un nouveau descripteur pour une nouvelle transaction et initialiser son statut à l'état *ACTIVE*. *STMAbort* aborte une transaction et met son statut à *ABORTED* et l'espace mémoire occupé par la transaction peut être récupéré immédiatement. Dans la méthode *STMRead,* il existe deux cas, d'abord si le descripteur de transaction contient déjà une entrée (*TransactionEntry)* pour l'emplacement mémoire requis, dans ce cas, la valeur de l'emplacement est trouvée dans le champ *newValue.* Autrement si le descripteur n'a aucune entrée pour l'emplacement mémoire requis alors le système crée une nouvelle entrée en y recopiant la valeur de l'emplacement mémoire dans les champs *newValue* et *OldValue* tandis que la version de la valeur serait également enregistrée dans les champs *NewVerion* et *OldVersion* [35].

La fonction *MemRead* est employée pour récupérer la valeur de l'emplacement mémoire et son numéro de version. Les informations sur la valeur et le numéro de version peuvent être dans des emplacements différents dépendamment du statut d'une autre transaction qui a y accédé. Si l'emplacement n'a pas été consulté par aucune transaction alors la valeur courante réside dans l'emplacement mémoire et l'enregistrement de propriété (*Orec*) contient le numéro de version. Si une transaction a accédé à l'emplacement mémoire et a commis ses résultats alors la dernière valeur serait trouvée dans le champ *newValue* et la version dans le champ *NewVerion* dans l'emplacement mémoire de cette transaction. De même si la transaction n'a pas encore commité mais elle est en traitement alors la valeur est maintenue dans le champ *OldValue* et la version dans le champ *OldVersion* de la transaction.

STMWrite est une autre méthode utilisée par la bibliothèque WSTM. *STMWrite*, s'assure d'abord qu'une entrée existe (dans le descripteur) pour l'emplacement mémoire en question. Ceci est réalisé tout simplement par une opération de lecture *(en appelant STMRead)*. La nouvelle valeur est écrite au-dessus de l'ancienne valeur (dans le champ *NewValue)* et l'ancienne version est incrémentée par un et mise en tant que nouvelle version (dans le champ *NewVerion)* après il y aura un copiage de la nouvelle version vers toutes les entées qui ont une relation avec le même *Orec* pour garder le descripteur bien formé [35].

L'opération *STMCommit* acquiert en premier lieu les enregistrements de propriété de tous les emplacements mémoire consultés par la transaction. Si c'est fait avec succès alors la transaction change son état à *COMMITTED,* met à jour les emplacements mémoire avec les nouvelles valeurs et libère les enregistrements de propriété acquis [35]. Ces trois opérations ont lieu d'une façon atomique et concurrente. WSTM détecte les conflits tout en commitant les transactions cependant il ne fait pas la validation de l'ensemble de lecture continuellement. Éventuellement certaine transaction peut entrer dans un état incohérent et elle doit être interrompue et ré-exécutée à nouveau, pour cela le WSTM utilise *STMValidate* pour assurer la cohérence des données (par exemple pour éviter les boucles infinies à l'intérieur d'une transaction). *STMValidate* est une opération de lecture seule qui vérifie que les enregistrements de propriété pour chaque emplacement mémoire contiennent exactement le même numéro de version retenu dans le descripteur de la transaction demandant (qu'elle exécute *STMValidate).* La validation réussit et *STMValidate* renvoi la valeur *true* si tous les enregistrements contiennent les valeurs prévues. Autrement la validation s'échoue, et elle renvoie *false*.

Le commit des transactions emboîtées est plus compliqué car la transaction externe peut exposer les valeurs mises à jour. De plus, la transaction interne peut causer l'abort de la transaction externe. Le descripteur de transaction dépiste le nombre de transactions emboîtées à l'intérieur d'une transaction et maintient l'opération de commit en suspension jusqu'à ce que la

transaction la plus extérieure commite [35]. Si la transaction interne s'aborte alors la transaction globale ne peut pas commiter du tout.

La bibliothèque de WSTM a *STMWait* qui est utilisé dans les RCCs pour suspendre une transaction jusqu'à ce qu'une autre transaction modifie l'emplacement mémoire consulté par la première transaction. Afin de le faire, d'abord elle acquiert tous les *Orecs* relatifs à cette transaction, met à jour le statut de la transaction à *ASLEEP* et arrête le thread exécutant la transaction. Quand une autre transaction met à jour l'un des emplacements mémoire, donc elle sera en conflit avec le thread arrêté mais le gestionnaire de conflit devrait permettre au thread actif de procéder en avant et puis plus tard laisser la transaction suspendue continuer. Comme résultat, le thread suspendu devra s'exécuter de nouveau.

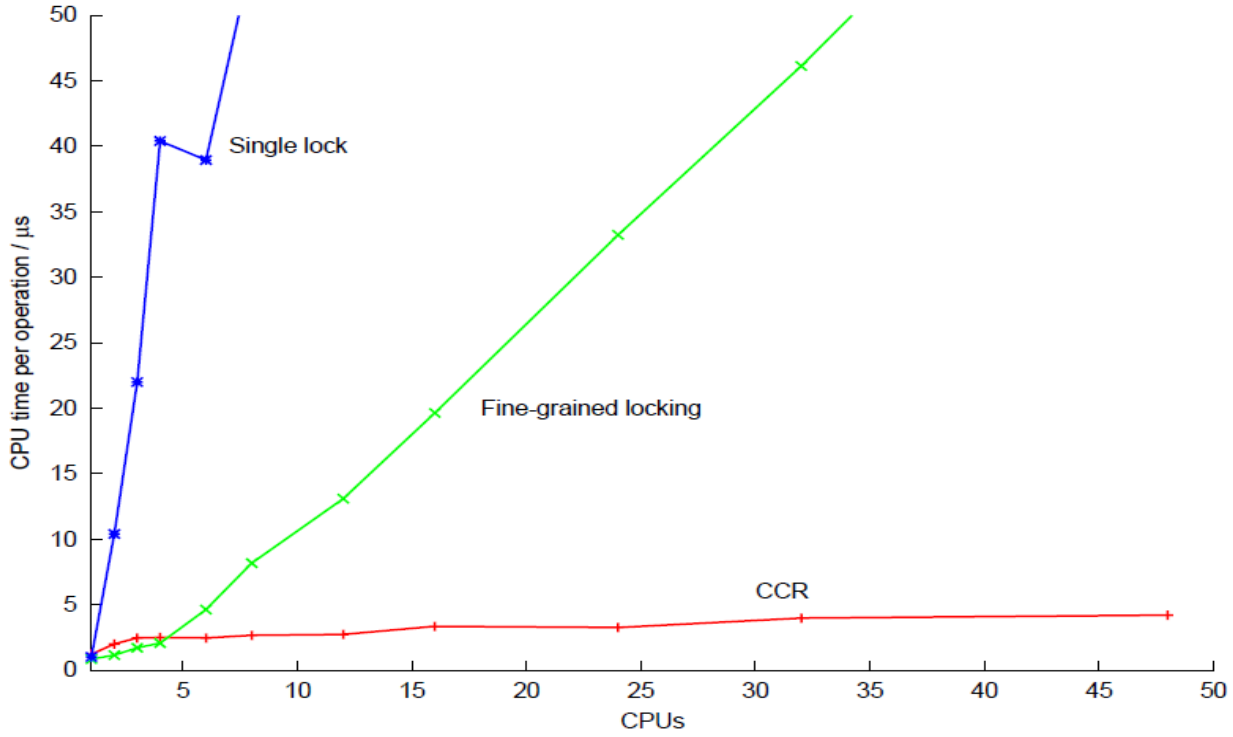

Figure 4.2.4: Comparaison de performance de WSTM avec les verrous [35]

4.2.2 Discussion

L'inconvénient principal de l'implémentation de WSTM est son coût élevé en termes de temps d'exécution et d'espace mémoire du fait que chaque mot mémoire consulté dans une transaction ait un surcoût jusqu'à sept mots, plus une certaine fraction de la table de hachage. En outre Haris et Fraser [35] ont identifié plusieurs points faibles dans leur conception de WSTM ;

- l'enregistrement de propriété peut être accédé par seulement une transaction.
- L'opération de lecture et d'écriture exige la recherche dans le descripteur de transaction pour l'entrée de transaction dans un *OREC* spécifique.
- Le traitement d'une entrée de transaction décrivant une lecture seule d'un emplacement mémoire nécessite de mettre à jour deux fois l'*orec* correspondant, ce qui est un surcoût (le premier accès pour acquérir la propriété et le deuxième pour sa libération).

- L'opération de relance *(Retry)* empêche le système d'être non bloquant.

Haris et Fraser ont donné un remède à ses problèmes qui est très compliqué à décrire. Nous évitons de présenter les détails ici à cause que d'autres systèmes que nous allons étudier par la suite réalisent ce but avec considérablement moins de complexité.

En fait, WSTM est bien adapté pour des applications où les opérations concurrentes sont probablement sans conflits [35]. La Figure 4.2.4 affiche l'exécution de WSTM (CCR dans la figure) en comparaison avec un seul verrou et avec un verrouillage granuleux sur une table de hachage.

4.3 Dynamic Software Transactional Memory (DSTM)

Herlihy, Luchangco, Moir et Scherer ont présenté le DSTM en 2003. DSTM a fondamentalement surmonté l'insuffisance des STMs précédents où la taille de transaction et les emplacements mémoire accédés par la transaction ont été statiquement définis à l'avance. Cependant, DSTM permet aux transactions et aux objets transactionnels d'être créé dynamiquement et permet à une transaction de déterminer la séquence d'objets à accéder dynamiquement, pour cela le DSTM est bien adapté pour l'implémentation des structures de données dynamiques comme les listes et les arbres d'où il intervient le nom de STM Dynamique [44].

DSTM est fondamentalement une *API* (*Applications Programming Interface*) pour synchroniser les données partagées sans utiliser des verrous. Le prototype de l'API est disponible en C++ et en Java. DSTM garantie une synchronisation non bloquante avec liberté d'obstruction. Cependant, la liberté d'obstruction est une propriété plus faible que la liberté de verrou ce qui rend simple l'implémentation de DSTM [44]. La liberté d'obstruction suppose qu'un thread arrêté (en attente) n'empêche pas les autres threads de faire un progrès.

Tableau 4.3.1 : Les caractéristiques de base de DSTM.

DSTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Objet
Stratégie de Mise à jour	Différée
Détection de Conflit	Tôt
Stratégie de résolution de conflit	Gestionnaire de conflit
Transactions emboitées	Aplaties

Contrairement à la liberté de verrou, la liberté d'obstruction n'empêche pas complètement l'occurrence des *Livelocks*. En d'autres termes, plusieurs threads en exécution peuvent interférer et se gêner pour progresser en avant [43]. La propriété de liberté d'obstruction fournit des techniques simples pour donner la priorité aux transactions du fait que n'importe quelle

transaction peut aborter une autre transaction à un moment particulier. Cependant, une transaction prioritaire aborte souvent une transaction de priorité plus basse. Dans DSTM, une transaction peut prévoir qu'elle va interrompre une autre transaction et dans cette situation c'est au gestionnaire de conflit de décider si elle devrait aborter la transaction ou attendre et la laisser compléter son traitement. DSTM a une implémentation modulaire du gestionnaire de conflit ce qui permet d'intégrer des nouvelles politiques de gestion de conflit sans aucun risque d'altérer l'exactitude du code de transaction [43]. Les propriétés conceptuelles de base de DSTM sont affichées dans le Tableau 4.3.1.

DSTM a introduit une nouvelle propriété qui est la possibilité de libérer un objet avant le commit d'une transaction. Cette propriété dans le scénario idéal peut être très efficace cependant elle peut causer des erreurs d'incohérence ce qui nécessite une bonne gestion soigneuse de la part du programmeur. DSTM fournit un isolement faible et utilise une granularité au niveau objet, il utilise une stratégie de mise à jour différée c.à.d. le système maintient et travaille sur des copies de données privées et lors de commit il y aura une mise à jour des données originales à partir de ces données privées. DSTM a un contrôle de concurrence optimiste avec une politique de détection de conflit tôt. Le conflit peut être détecté quand la transaction essaie d'ouvrir un objet qu'il est déjà ouvert pour modification par une autre transaction ou lors de la phase de validation de l'ensemble de lecture. Finalement DSTM autorise des transactions emboîtées de type aplaties.

4.3.1 Aperçu avec exemple

Dans cette section on va discuter comment utiliser le DSTM dans la programmation et quelles sont les méthodes et les procédures qu'il possède et pour quels buts.

DSTM maintient une collection des objets transactionnels qui sont concurremment consultés par des transactions différentes. Un objet transactionnel dans DSTM encapsule les objets simples de Java. En d'autres termes chaque fois qu'une transaction a besoin d'accéder à un objet régulier, elle ouvre l'objet transactionnel et par conséquent elle peut lire et modifier des objets réguliers. Les modifications apportées à un objet par une transaction particulière ne sont pas visibles en dehors de la transaction jusqu'à ce qu'elle commite. Cependant, si la transaction s'aborte ses modifications sont ignorées [44].

Les transactions peuvent être créées dynamiquement à n'importe quel moment. Cependant, la création d'une nouvelle transaction et son initialisation ne peut pas être faite comme partie d'une autre transaction [44]. La classe *TMThread* est responsable sur le traitement parallèle des threads. Elle est étendu (hérite) des threads réguliers de Java. La classe *TMThread* fournit des méthodes supplémentaires pour commencer, commiter, aborter et obtenir le statut d'une transaction. Les threads peuvent être créés et détruits dynamiquement. Les objets transactionnels peuvent être implémentés par la classe *TMObject*. Dans l'exemple de la Figure 4.3.1, on encapsule une instance de l'objet atomique *counter*, à l'intérieur d'un objet de la classe *TMObject*.

```
1    Counter   counter = new Counter (0);
2    TMObject  tmObject = new  TMObject (counter);
```

Figure 4.3.1: Encapsulation d'un Objet à l'intérieur de TMObject [44]

Pour chaque classe héritée de la classe *TMObject* (c.à.d. les objets de cette classe peuvent être encapsulés à l'intérieure d'un objet transactionnel), elle doit implémenter l'interface *TMCloneable*. Cette interface, oblige la classe qui l'implémente d'implémenter une méthode

publique *clone ()* qu'elle retourne une nouvelle copie disjointe de l'objet de la même classe. Cette méthode est utilisée lors de l'ouverture des objets transactionnels pour modification comme indiqué dans la Figure 4.3.2. Durant le clonage d'un objet, le système garantit que cet objet n'est pas été changé.

```
1   Counter counter = (Counter) tmObject. open(WRITE);
2   counter.inc (); // incrémentation de counter
```

Figure 4.3.2: Ouverture d'un objet dans le mode d'écriture [44]

Une transaction est commencée en appelant la méthode *beginTransaction()* et elle demeure active jusqu'à ce qu'elle aborte ou commite. Pendant l'exécution d'une transaction, elle peut ouvrir des objets transactionnels en exécutant la méthode *Open ()* qui accepte comme argument une constante indiquant le type d'ouverture (pour modification ou lecture). La méthode *Open ()* crée une copie (propre version) de l'objet à partir de la version originale en utilisant la méthode *clone ()* de l'objet. Par la suite la transaction peut réaliser son traitement sur sa propre copie sans aucun besoin de synchronisation car cette copie et accédée uniquement par la transaction créatrice [44].

Une transaction commite ses modifications en exécutant la méthode *commitTransaction()* qu'elle renvoie *True* si elle commite avec succès et *false* autrement. De même une transaction peut être abortée en utilisant la méthode *abortTransaction().* DSTM s'assure que les transactions commitent d'une façon linearisable [44]. En d'autres termes, on exécute des transactions comme si elles sont exécutées une par une.

Les lectures des objets (*TMObject)* peuvent mener aux problèmes de synchronisation. Par exemple une transaction *T* peut noter que le *TMObject* qui a ouvert ou qu'elle va l'accéder par la suite a été déjà modifiée par une autre transaction. Donc la transaction *T* voit un effet partiel de la deuxième transaction ce qui peut mener à des exécutions anormales (par exemple un pointeur nul ou violation de limite d'un tableau) [44]. Afin de traiter une telle situation DSTM utilise la *validation.* Les contrôles de validation sont déclenchés à chaque ouverture d'un objet transactionnel par une transaction pour vérifier si le même objet est ouvert par une autre transaction d'une façon contradictoire en même temps [44]. Quand il est ouvert alors la méthode *Open ()* engendre une exception au lieu de créer un clone de cet objet car la transaction ne peut pas commiter ultérieurement et donc il ne faut pas perdre des efforts pour rien. Le *processus de validation* évite les accès contradictoires aux objets transactionnels.

4.3.2 Détails d'implémentation

Dans DSTM, l'objet transaction possède un attribut nommé *status* qui est initialement mis à *ACTIVE,* et par la suite il est changé à *ABORTED* ou *COMMITTED* à l'aide d'un *CAS.* Rappelons qu'un objet transactionnel est un récipient pour d'autres objets des autres classes utilisateur. DSTM utilise deux niveaux d'indirection pour accéder à un objet. Un objet est référencié à travers un *TMObject* qui pointe vers un *repère (Locator).* Un *repère* possède trois champs, le champ *Transaction* qui pointe vers la transaction la plus récente qui a ouvert l'objet transactionnel dans le mode d'écriture, le champ *OldObject* qui pointe vers l'ancienne version de l'objet et le champ *NewObject* qui pointe vers la nouvelle version de l'objet qui est la valeur actuelle de l'objet (voir Figure 4.3.3).

La version courante d'un objet peut être déterminée à partir de son *repère*, si le pointeur transaction est nul c.à.d. L'objet est ouvert pour lecture, la valeur originale et dans le champ *OldObject*. Cependant, si le pointeur transaction pointe vers une transaction commise (son statut

à *COMMITTED*), la version courante est celle modifiée par la transaction (*NewObject*) et l'ancienne version sera écrasée. Si la transaction est abortée (son statut est à *ABORTED)* alors l'ancienne version (*OldObject*) devient la version courante et la nouvelle version devient inutile. Dans le cas où la transaction est en activité (son statut est en *ACTIVE)*, l'ancienne version est la version courante et la nouvelle version est la valeur de la transaction active. Cette version devient courante si la transaction commite avec succès, autrement elle sera écrasée.

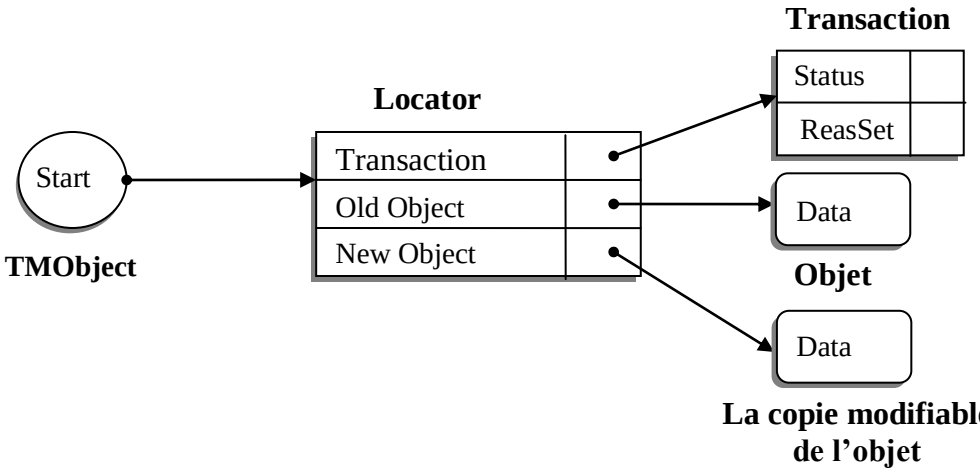

Figure 4.3.3: Structure d'un Objet Transactionnel [44].

Une transaction peut ouvrir plusieurs objets transactionnels pour modification pendant son exécution, quand cette transaction commite avec succès, tous ses objets privés (les clones créés par la transaction à travers la méthode *Open(Write)*) remplacent les versions courantes de ces objets. La Figure 4.3.4 et Figure 4.3.5 affichent l'adressage indirect de l'ancienne version à la nouvelle version.

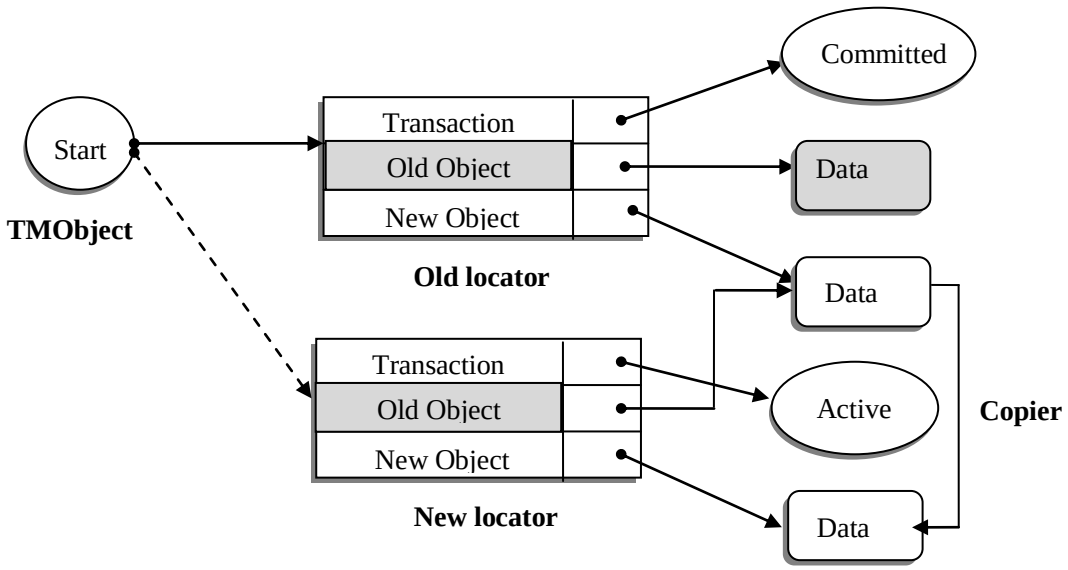

Figure 4.3.4: L'ouverture d'un objet transactionnel après un commit [44].

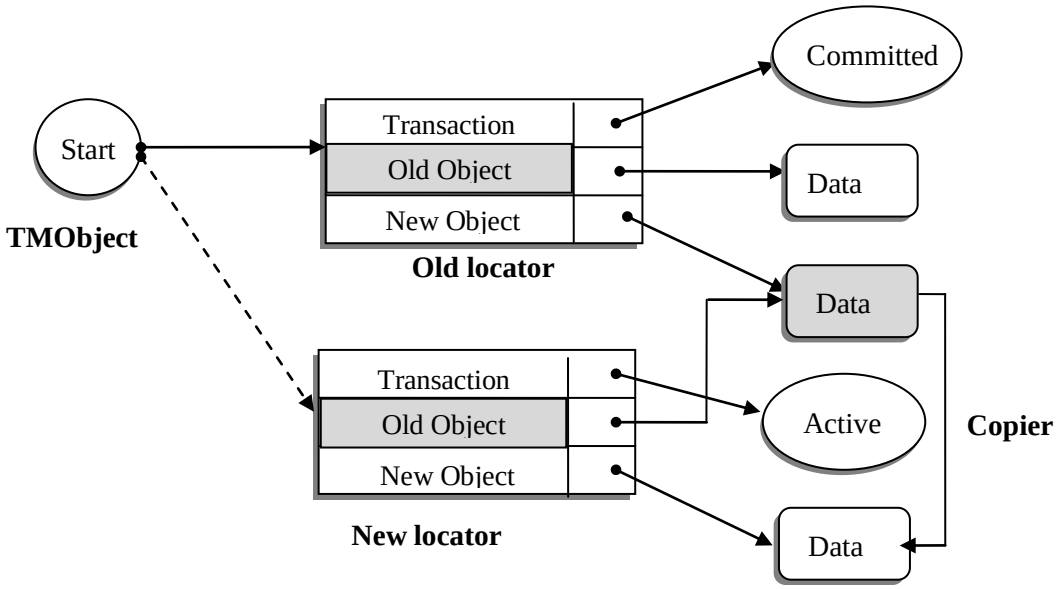

Figure 4.3.5: L'ouverture d'un objet transactionnel après un abort [44].

Lors de l'appariation d'un conflit, DSTM consulte le gestionnaire de conflit pour décider quelle est la transaction à suspendre ou aborte et laquelle à laisser continuer son traitement. Cependant, il n'existe par de règle pour permettre à une transaction particulière de progresser toujours en avant. DSTM implémente une interface pour le gestionnaire de conflit qui permet à plusieurs politiques de résolution du conflit d'agir en même temps. Le but essentiel du gestionnaire de conflit est d'assurer qu'une transaction particulière essayant d'accéder à un objet transactionnel doive avoir une chance pour progresser. En d'autres termes, si une transaction particulière essaie d'interrompre une autre transaction contradictoire alors elle doit avoir la permission après plusieurs tentatives. Chaque transaction en exécution a sa propre instance de gestionnaire de conflit qui permet à cette transaction de décider d'interrompre une transaction contradictoire ou d'attendre et la laisse finir d'abord. D'ailleurs les gestionnaires de conflit des différentes transactions peuvent communiquer entre eux pour gérer les priorités et prendre des décisions dépendamment des différentes situations [44].

Le gestionnaire de conflit met en application deux types de méthodes [44] : *notification* et *feedback***.** Les méthodes de notification indiquent au gestionnaire de conflit sur les différents événements, par exemple l'événement *commitTransactionSucceeded()* est déclenché quand certaine transaction termine son exécution avec succès. Une autre méthode de notification est *commitTransactionFailed()* qu'elle est déclenchée quand une transaction a échoué de commiter. Le gestionnaire de conflit appelle les méthodes de *feedback* pour décider ce qu'il doit faire dans les différentes situations contradictoires. Un bon exemple de la méthode de feedback est le *shouldAbort()* qui est appelée par le gestionnaire de conflit quand il découvre que l'objet demandé par la transaction est déjà ouvert par une autre transaction, alors il indique au transaction de s'aborter.

4.3.3 Discussion

Les politiques de gestion de conflit dans DSTM sont restées un domaine de recherche ouvert, où on peut contribuer et aller en avant avec des politiques de gestion de conflit plus efficaces. Une autre caractéristique de conception intéressante de DSTM est l'opération de libération qui permet à une transaction de libérer un objet transactionnel avant de commiter ses

résultats. La libération de l'objet de transaction abaisse le coût de validation, cependant il peut devenir une source de conflit menant à aborter des transactions.

4.4 Object Based Software Transactional Memory (OSTM)

Keir Fraser en 2003 a présenté le premier système STM libre de verrou avec une granularité au niveau objet dans sa thèse de PhD [26] Qui s'appelle également le STM de Fraser (FSTM) (il souvent nommé OSTM). Un objet dans OSTM est un bloc mémoire contigu. OSTM est tout à fait semblable à DSTM de Herlihy et autres [44], La différence de base entre les deux systèmes est que DSTM utilise la liberté d'obstruction comme technique de synchronisation tandis qu'OSTM est basé sur la liberté de verrou. D'ailleurs OSTM utilise un isolement faible et une approche de détection de conflit en retard. OSTM ne supporte pas les transactions emboîtées [42]. Les caractéristiques de base d'OSTM sont affichées dans le Tableau 4.4.1.

Tableau 4.4.1 : Les caractéristiques de base d'OSTM

OSTM	
Synchronisation	Liberté de verrou
Contrôle de Concurrence	Optimiste
Granularité	Objet
Stratégie de Mise à jour	différée
Stratégie de Détection de Conflit	Tard
gestionnaire de conflit	Abort
Transactions emboitées	Non Supportées

4.4.1 Détails d'implémentation

OSTM est une bibliothèque implémentée dans le langage C. les structures de données utilisées par OSTM sont très similaires à celles de DSTM. Un objet dans OSTM est un emplacement mémoire contigu (structure de données) qui est accédé à travers de son entête (*object header*). En d'autres termes, le contenu d'un objet n'est pas accessible directement dans une application, à moins qu'un pointeur à l'entête de cet objet soit obtenu. Une référence à un objet dans OSTM est en fait un pointeur à l'entête de l'objet qui est de taille d'un mot mémoire, l'entête contient un pointeur au bloc de données actuel suivant les indications de la Figure 4.4.1. Ce pointeur est mis à jour pour pointer vers un nouveau bloc de données après le succès d'un commit [26].

Une transaction peut ouvrir plusieurs objets pour les traiter. Chaque transaction a un descripteur de transaction. Le descripteur de transaction maintient un champ *STATUS* qui peut avoir des valeurs *Undecided, Read-checking, Committed* ou *Aborted*. Le descripteur de transaction aide en dépistant les objets ouverts pour lecture ou écriture par une transaction particulière. Le descripteur de transaction maintient deux listes chaînées à cette fin, c.à.d. une liste de *lecture seule* (read-only list) et une liste de *lecture/écriture* (read-write list).

La liste de *lecture-écriture* est utilisée pour les objets qui sont ouverts pour le but d'écriture. Chaque nœud dans cette liste contient un *object handle* qui contient les champs : *Object-ref* qui est une référence à l'en-tête de l'objet, *old-data* qui est un pointeur vers l'objet courant, *new-data* qui est une référence à la copie d'ombre de l'objet et finalement un pointeur vers le prochain objet, c.à.d. le prochain *handle*. Les mises à jour apportées à la copie d'ombre sont maintenues invisibles et privées jusqu'à ce que la transaction commite. La liste de *lecture seule* a une structure semblable mais elle n'a pas une copie d'ombre de l'objet [26].

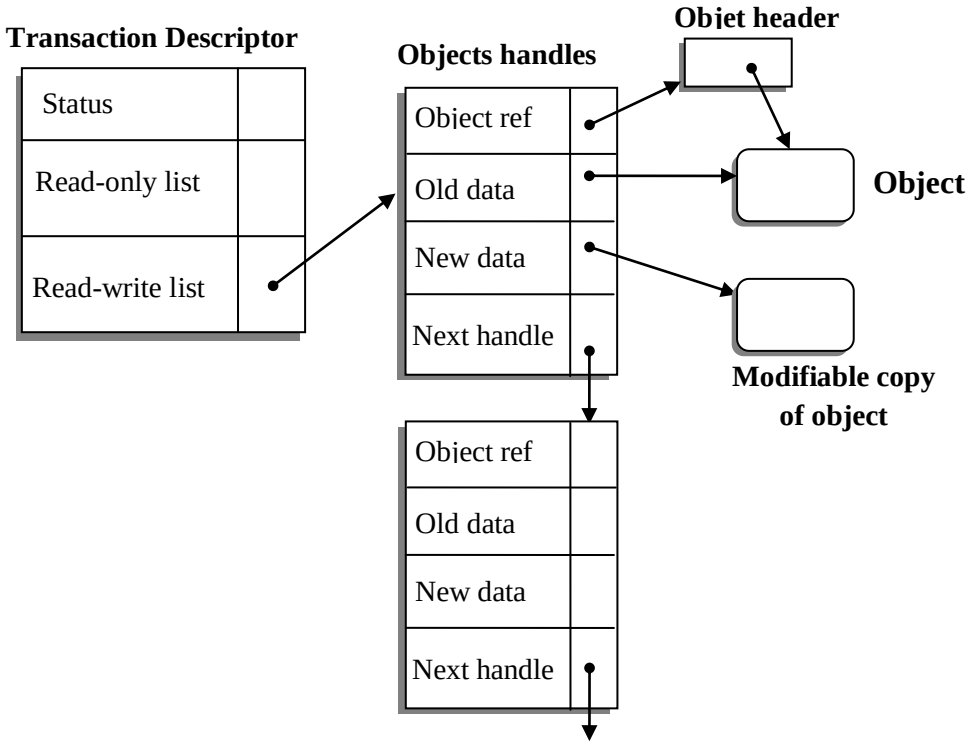

Figure 4.4.1: Structure de données d'OSTM [26].

Les objets ouverts pour une lecture seule, sont ajoutés à la liste de lecture seule. De même, Chaque fois qu'une transaction ouvre un objet de données pour une modification, on l'ajoute à la liste de lecture-écriture. L'OSTM utilise un commit à deux phases ; dans la première phase, la transaction acquiert le contrôle sur tous les objets de données dont elle a besoin pour commiter, alors que dans la deuxième phase les objets acquis sont libérés. Dans la *phase d'acquisition*, les objets sont acquis dans un ordre global logique en rendant l'entête de l'objet pointé vers le descripteur de transaction en utilisant un CAS[2]. Ensuite (*phase de décision)* le statut de transaction est changé suivant les résultats finaux de la transaction indiquant le succès ou l'échec. Une transaction commite avec succès quand elle réussit à mettre à jour à la mémoire partagée tous les objets figurant dans la liste de lecture-écriture. La deuxième phase est la *phase de libération* dans laquelle la transaction libère tous les objets qu'elle a acquis dans la phase d'acquisition [26].

[2] Si le CAS échoue à cause d'une autre transaction qui est entrain de commiter, la transaction aborte et essaie d'aider la transaction conflictuelle pour assurer le comportement liberté de verrou.

4.4.2 Discussion

L'un des goulots de performance principaux qu'OSTM souffre est son mécanisme de lecture seule. Les opérations de lecture seule doivent intervenir pendant les phases d'acquisition et de libération. Ceci peut introduire des conflits inutiles entre les transactions. En d'autres termes, il est inutile d'acquérir des objets concurrents pour un accès en lecture seule. Comme d'autres algorithmes de recherche qui ont un seul point d'entrée, c.à.d. la racine, pour chaque opération, souffrira du goulot d'exécution [26]. La structure de données d'arbre est un bon exemple pour expliquer cet inconvénient dont laquelle pour accéder à une feuille (nœud d'extrémité), il faut passer par la racine et traverser jusqu'à cette feuille particulière.

Afin de surmonter cet inconvénient, Fraser a modifié l'algorithme pour acquérir seulement les objets qui sont dans la liste de lecture-écriture d'une transaction. *La phase d'acquisition*, est suivie d'une *phase de lecture* dans laquelle on cherche l'objet actuel dans la liste de lecture seule de la transaction pour assurer la cohérence. Elle compare la version actuelle de l'objet avec la version qui a été identifiée lors de première ouverture. Si toutes les références s'assortissent alors la transaction commite avec succès, autrement, la transaction s'échoue. Si la *phase de lecture* note qu'un entête d'un objet est actuellement ouvert par une autre transaction alors il n'aidera pas la transaction contradictoire [26]. Fraser a discuté ce scénario par un exemple. Considérant les transactions *T1* et *T2* s'exécutent en même temps. Supposant que *T1* ouvre un objet de données *Xi* pour une lecture et *Yi* pour une écriture et *T2* ouvre *Xi* pour écriture et *Yi* pour lecture. Selon l'implémentation d'OSTM, *T1* et *T2* commiteront avec succès. Cependant, ces transactions violent les critères de cohérence et d'exactitude d'OSTM puisque les transactions ne sont pas serialisable, et les modifications apportées par *T2* ne sont pas visibles à *T1* ou vice versa. Fraser a résolu ce problème en introduisant les deux modifications suivantes [26] :

1. Il a introduit une nouvelle valeur de statut de transaction avec le nom *read-checking*. Cette valeur de statut indique qu'une transaction est dans la phase de lecture.

2. Une transaction commite avec succès ou aborte quand son statut est changé en *read-checking*.

Quand la transaction complète la phase d'acquisition, elle se tourne vers l'état de *read-checking*. Dans la phase de lecture, une transaction parcourt sa liste de lecture seule et contrôle la cohérence des objets dans cette liste. Si au moins un objet de données a été changé depuis sa dernière lecture par la transaction, et le statut de transaction est dans l'état *Undecided*, alors la transaction aborte. Si la transaction trouve par hasard un conflit dans l'état *read-checking*, alors selon la position des transactions dans l'ordre global total déterminé dans la mémoire; Si la transaction actuelle précède la transaction contradictoire alors la transaction actuelle interrompt la transaction contradictoire. Autrement, la transaction actuelle aide la transaction contradictoire.

OSTM autorise uniquement des objets de taille fixe. Les objets de taille fixe ne sont pas appropriés aux applications qui supportent une collection des objets de données hétérogènes ou les objets avec taille variable, par exemple, listes de saut (*skip list*). D'ailleurs l'OSTM ne supporte pas les transactions emboîtées et c'est une autre limitation dans sa conception.

4.5 Ennals STM

Robert Ennals dans [23] a argué que la prévention d'inter-blocage est la seule raison pour implémenter les transactions d'une manière non-bloquante, sinon il n'est y a aucun besoin pour fournir un tel mécanisme au niveau utilisateur. Ennals argue que la "*liberté d'obstruction*" n'est pas appropriée pour un STM. Il argue que si on ignore le principe de liberté d'obstruction, on peut produire un STM qui est considérablement plus rapide [23]. De plus il a évoqué que rendre un STM libre d'obstruction prévient le système d'appliquer plusieurs optimisations très importantes qui produisent un système plus performant.

Les systèmes implémentant la liberté d'obstruction ont besoin (obligatoirement) au moins d'un niveau d'indirection à partir des métadonnées pour accéder à la version courante de la donnée (Figure 4.5.1). Cependant, ces niveaux d'indirection sont indésirables du fait qu'elles requièrent à un programme d'effectuer plusieurs chargements (*load)* des lignes de cache pour chaque lecture ou écriture.

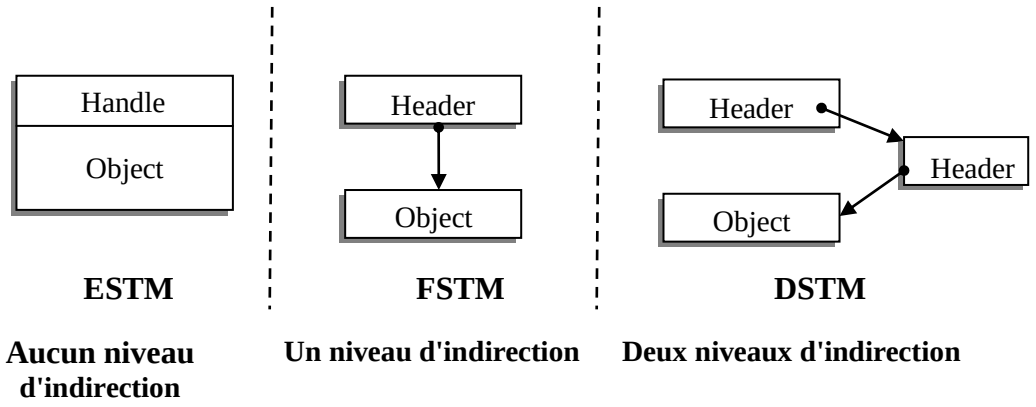

Figure 4.5.1: nombre d'indirection pour trouver un objet [23].

R. Ennals a proposé un STM [74 ,75] basé sur les verrous qui utilise une granularité au niveau objet[3] et a montré que son système est plus performant que les STMs prédécesseurs les plus performants qui utilisaient la liberté d'obstruction. ESTM utilise le protocole de verrouillage à deux phases révocable pour gérer les écritures et utilise un contrôle d'accès optimiste pour gérer les lectures. Les caractéristiques fondamentales du STM d'Ennals sont affichées dans le Tableau 4.5.1 suivant.

4.5.1 Détails d'implémentation

ESTM est un système basé sur les verrous qui utilise un verrouillage dés le premier accès en écriture et maintient des verrous sur tous les objets écrits jusqu'à ce que la transaction commite. Cependant, si l'accès est une lecture, l'objet ne sera pas verrouillé.

ESTM divise la mémoire en une région publique contient les objets accédée par toutes les transactions, et une région privée à chaque transaction accédée uniquement par cette transaction pour stockées les descripteurs de lecture et d'écriture (informations de *book-keeping*). Ces descripteurs sont libérés une fois la transaction commite ou aborte.

[3] Un objet ici est un bloc de données manipulé par le programme utilisateur.

Tableau 4.5.1 : les caractéristiques de base d'ESTM.

ESTM	
Synchronisation	Bloquante
Contrôle de Concurrence	Lectures Optimistes et Ecritures Pessimistes
Granularité	Objet
Stratégie de Mise à Jour	Différée
Détection de Conflit	Tôt
Stratégie de Résolution de Conflit	Abort

Chaque transaction dans ESTM est représentée par un descripteur de transaction (*TransactionDescriptor*) qui contient (entre autres) un ensemble de lecture (*Reads*) et un ensemble d'écriture (*Writes*) comme il est indiqué dans la Figure 4.5.2. Chaque entrée dans l'ensemble d'écriture (*WriteDescriptor*) possède trois champs; *Last-Version* pour maintenir la dernière version *v* de l'objet récupérée lors de l'acquisition de cet objet. Le champ *Object-Pointer* contenant l'adresse de l'objet. Le champ *Working-Copy-Of-Data* initialisé à la valeur courante de l'objet, pour enregistrer la nouvelle valeur de l'objet. Chaque entrée dans l'ensemble de lecture (*ReadDescriptor*) possède l'adresse de l'objet (*Object-Pointer*) et sa version lue (*Version-Seen*).

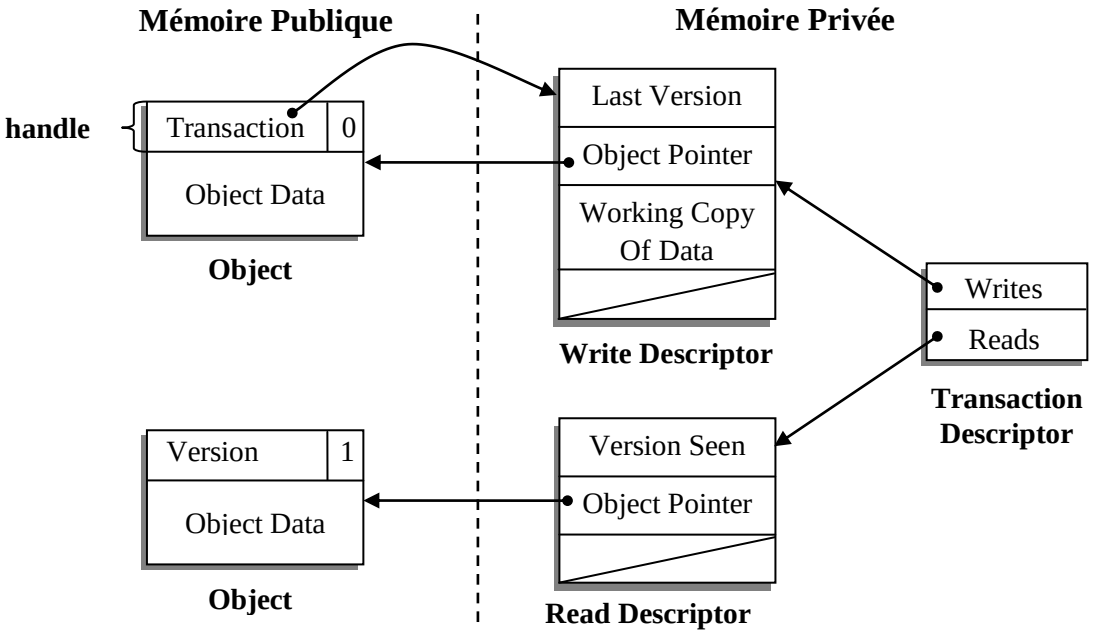

Figure 4.5.2: Structure de données d'ESTM [23].

Chaque objet dans ESTM (dans la mémoire publique) possède un *handle* (verrou) attaché à lui. Ce *handle* est un champ qui a deux rôles, si son bit de poids faible est mis à *1*, alors l'objet est libre et le champ *handle* contient le numéro de version *v* de l'objet, autrement l'objet est verrouillé et il contient un pointeur *w* vers le descripteur d'écriture dans la mémoire privée de la transaction possédant le verrou (Figure 4.5.2).

Lors d'une écriture à un objet *o* par une transaction *T*, *T* doit acquérir un accès exclusif à *o* et créer sa propre copie de *o*. si *o* est libre, *T* l' acquiert en effectuant un *CAS* sur le champ *handle* pour changer son contenu (numéro de version) par un pointeur vers son descripteur d'écriture en enregistrant la version courante de o dans le champs *Last-Version*. Cependant, si l'objet est déjà verrouillé par une autre transaction *S* (c.à.d. le champ *hanble* contient un pointeur *w*). *T* doit attendre une courte durée *d* pour que *S* libère l'objet en lui copiant le nouveau numéro de version. Après l'écoulement de *d* et si *S* n'a pas encore libéré l'objet *o*, si *S* est moins prioritaire que *T*, alors *T* demande à *S* de s'aborter (*T* peut trouver le descripteur de *S* en mettant les bits de poids faible du *w* à *0*). Si une situation d'inter-blocage se produit, le système aborte l'une des transactions [23].

Lors d'une lecture d'un objet *o* par une transaction *T*, si *T* est le propriétaire de *o* alors *T* lit la valeur courante de *o* à partir de *working-copy* créée précédemment dans le descripteur d'écriture, autrement, *T* doit attendre que le *handle* de l'objet soit une version, puis ajoute *o* dans l'ensemble de lecture de *T* en créant un nouvel descripteur de lecture en y sauvegardant le numéro de version de *o*. le système d'exécution aborte toutes les transactions qui s'avèrent incorrectes. Une transaction abortée peut être relancée ultérieurement.

Le commit dans ESTM est semblable à celui de DSTM. Une transaction qui va commiter doit d'abord s'assurer qu'elle est valide et par la suite rendre les copies de données privées publiques. En d'autres termes, la transaction vérifie que tous les objets qu'elle a lu n'ont pas été modifiés par d'autre transaction après les avoir lu, ceci est fait en validant l'ensemble de lecture, ensuite pour chaque objet écrit, la transaction copie sa nouvelle valeur de la copié privée vers l'emplacement réel de l'objet puis libère le verrou en remplaçant le *handle* de l'objet par la nouvelle version de l'objet.

4.5.2 Discussion

L'évaluation de performance de ESTM faite en comparant ESTM avec FSTM et DSTM a montré que ESTM et plus performant que les deux [23]. Ceci parce que ESTM n'a requis aucun niveau d'indirection contrairement aux autres (qui requièrent au moins un niveau d'indirection) et par conséquent il a éliminé plusieurs coups manqués du cache que les autres STMs en chargeant uniquement peu de lignes de cache à chaque processeur.

Ennals a montré que la liberté d'obstruction est une propriété non nécessaire pour un système de mémoire transactionnelle. Bien que ESTM puisse avoir plusieurs améliorations, mais il est le premier qui a implémenté un STM bloquant en utilisant les verrous qu'ultérieurement a été avéré qu'ils sont plus performants que les STMs non-bloquants.

4.6 Transactional Locking II (TL2)

Dice, Shalev et Shavit ont introduit le TL2 (*Transactional Locking II*) en 2006 [16]. Le TL2 est une amélioration de leur STM précédent (*TL*) développé en 2006 [15], par conséquent il est nommé TL2. TL2 est un système basé sur les verrous qui réalise un verrouillage de données au moment de commit, il utilise une horloge globale partagée pour estampiller les données. L'idée de l'estampillage dans les transactions n'est pas nouvelle car elle a été utilisée dans des transactions de base de données longtemps avant [84]. Cependant, l'estampillage utilisé dans TL2 est beaucoup plus rapide et plus efficace que les bases de données car le système STM est plus rapide [16]. Une idée semblable de l'estampillage a été introduite par Reigel et autres [73]. Mais le TL2 est différent du STM de Reigel car il est basé sur les verrous et il est simple tandis

que le STM de Reigel est non bloquant et complexe dans sa conception et il est plus coûteux [16].

Tableau 4.6.1 : Caractéristiques de base de TL2.

TL2	
Synchronisation	Bloquante
Contrôle de Concurrence	Optimiste
Granularité	Objet, Mot ou Région
Stratégie de Mise à jour	Différée
Détection de Conflit	Tôt ou Tard
Stratégie de Résolution de Conflit	Abort
Transactions Emboitées	Non Supportées

TL2 est un STM qui combine la mise à jour différée avec une technique de synchronisation bloquante. Les caractéristiques conceptuelles principales du TL2 sont affichées dans le Tableau 4.6.1. La résolution de conflit est réalisée en retardant des transactions ou en les abortant. La détection de conflit peut être tôt ou tard c.à.d. sélective. TL2 protège chaque emplacement mémoire par un verrou qui contient un pointeur vers la transaction qui le possède ou il contient le numéro de version de l'emplacement mémoire s'il est déverrouillé, ce numéro sera incrémenté quand une transaction modifie la valeur de l'emplacement. Les transactions commencent en lisant la valeur de l'*horloge globale de version*, cette valeur sera utilisée pour valider les emplacements mémoire lus. Quand une transaction commite ses mises à jour, la valeur de l'horloge globale est incrémentée et les verrous possédés par cette transaction seront libérés [16].

Dice et autres supportent également l'idée de Robert Ennals [23] qui a argué que dans les systèmes d'exploitation modernes, la prévention d'inter-blocage est la seule raison pour rendre les transactions non bloquantes, autrement il n'est y a aucun besoin pour fournir un tel mécanisme aux transactions au niveau utilisateur. Les verrous éliminent le besoin d'adressage indirect dans la mémoire partagée. En d'autres termes, les transactions modifient directement les données après avoir acquérir les verrous nécessaires en gardant des pointeurs vers un *log de défaire* (undo log) non partagé avec d'autres threads. Cependant, les verrous ont besoin toujours d'un *système mémoire fermé* [16]. Un système mémoire fermé permet à la mémoire d'être libérée automatiquement irrégulièrement. TL2 évite les inter-blocages et les Livelocks par des minuteries *(times-out)* en demandant à d'autres transactions de s'aborter.

Les études faites par R. Ennals dans [23] ont prouvé que les STMs bloquant (basés sur les verrous) ont l'avantage sur les STMs non bloquants à cause de sa nature et implémentation simples. Mais un tel système STM doit adresser les deux exigences suivantes;

- **Le système mémoire doit être fermé** : Le système à mémoire fermée implique que l'espace mémoire qu'il n'est plus utilisé doit être récupéré automatiquement c.à.d. il doit y exister un ramasse-miettes (mécanisme de récupération de place). En d'autres termes La mémoire utilisée par les transactions doit être recyclable pour être utilisée par d'autres

threads (non transactionnels) et vice versa [16]. Certaines langages de programmation incorporent ce service (comme Java) où la mémoire qu'elle n'est pas référenciée par aucun objet est automatiquement libérée, tandis que d'autres ne possèdent pas ce service mais on peut le faire explicitement à travers des commandes différentes par exemples les deux commandes *malloc()* et *free()* dans le langage C.

- **Environnement d'exécution spécialisé:** Les systèmes STM exigent un tel environnement qui peut surmonter les incohérences comme les boucles infinies, les accès mémoire illégaux et d'autres anomalies d'exécution [16]. Un tel genre d'incohérences peut être surmonté en interrompant et en relançant une transaction. D'ailleurs, des boucles infinies peuvent être résolues par des contrôles de validation *(validation checks).* Cependant, la validation de chaque ensemble de lecture est très coûteuse en terme de temps d'exécution ce qui influe la performance d'un système STM. Ce temps peut être réduit en vérifiant par intermittence les boucles infinies au lieu d'un contrôle continu [76].

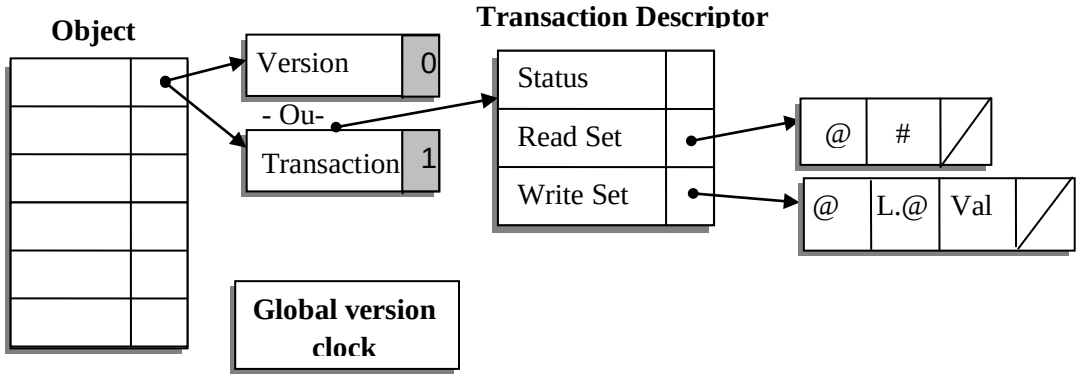

Figure 4.6.1: Structures de données de TL2.

Dans TL2 Une transaction qui va faire une écriture a besoin de maintenir un ensemble de lecture et un ensemble d'écriture contrairement aux transactions faisant uniquement des lectures. Chaque entrée dans l'ensemble de lecture contient l'adresse de l'objet et le numéro de version du verrou associé à l'objet. Une entrée dans l'ensemble d'écriture contient l'adresse de l'objet, l'adresse de son verrou et la nouvelle valeur de l'objet (Figure 4.6.1). Lors d'une écriture à un objet, la transaction doit d'abord vérifier si cet objet est dans son ensemble d'écriture. S'il n'est y pas, la transaction créera une nouvelle entrée pour cet objet et modifie l'objet dans l'ensemble d'écriture et non pas dans l'emplacement réel (mise à jour différée).

Pour commiter, une transaction doit d'abord acquérir tous les verrous pour tous les objets modifiés (figurant dans l'ensemble d'écriture), ensuite elle doit valider son ensemble de lecture. Si elle réussit alors elle peut compléter en copiant les valeurs modifiées de son ensemble d'écriture vers les objets réels, puis lâcher les verrous et libérer ses structures de données. Une fois elle commite, la transaction acquiert le verrou de l'horloge globale de version pour l'incrémenter [16].

Dice et autres croient que TL2 a surmonté la majorité des problèmes de sûreté et de performance qui créaient un goulot pour les implémentations des STM basées sur les verrous. TL2 a les améliorations conceptuelles suivantes par rapport à d'autres systèmes STM [16] :

- Contrairement aux anciennes implémentations STMs basées sur les verrous, TL2 évite les vulnérabilités liées aux lectures incohérentes des emplacements mémoire. Précédemment, une aide de la part du compilateur ou des vérifications explicites par le programmeur ont été exigées à cet égard.

- TL2 fait un recyclage de la mémoire automatiquement (ramasse miettes) en utilisant des opérations de style *malloc()* et *free()* et ceci est réalisé facilement avec une complexité non significative. La comparaison de performance de TL2 avec d'autres STMs et avec le verrouillage simple a montré que TL2 est plus concurrent et plus performant.

4.6.1 Détails d'implémentation

TL2 est une variante de verrouillage transactionnel (TL) basé sur *l'horloge de version globale* qui a été introduit par Dice et Shavit plus tôt [42]. En se basant sur cette approche de gestion des versions globales, TL2 a pu éliminer plusieurs problèmes de sureté en comparaison avec l'approche précédente de gestion de versions locales, de plus l'utilisation des versions globales améliore la performance des transactions réalisant uniquement des lectures [16]. TL2 utilise le schéma de verrouillage à deux phases pour verrouiller au moment de commit *(commit-time),* comme le TL et contrairement aux autres STMs (McRT-STM [76] et Ennals [23]) faisant le verrouillage dès le premier accès aux données *(encounter-time)*. Le verrouillage à deux phases implique que dans la première phase la transaction acquiert graduellement tous les verrous. Cette phase s'appelle *extension* et une fois tous les verrous sont acquis alors elle commite. Dans La deuxième phase, la transaction libère graduellement tous les verrous acquis. Cette phase s'appelle *rétrécissement*.

Chaque objet est associé à un verrou d'écriture *(write-lock).* Le verrou est un mot mémoire où un bit unique est utilisé pour indiquer que le verrou est acquis. Les bits du verrou restants sont utilisés pour maintenir un numéro de version si le verrou est libre et un pointeur vers le propriétaire de verrou s'il est acquis (Figure 4.6.1). Ce numéro de version est incrémenté après chaque modification de l'objet associé. Les verrous d'écriture avec version fournissent beaucoup de performance et d'exactitude [16]. TL2 peut être implémenté en utilisant les deux stratégies d'association des verrous d'écriture avec les données suivantes:

- assigner les verrous par objet *(PO -per object)* : où un verrou est associé à chaque objet.

- assigner des verrous par bloc *(PS -per stripe) :* où on alloue un ensemble de verrous (généralement sous forme d'une table) et on partitionne la mémoire en strippes en utilisant une fonction de hachage pour mapper chaque strippe à un verrou.

4.6.1.1 L'Algorithme TL2

Dans TL2, l'exécution d'une transaction d'écriture et différente de l'exécution d'une transaction réalisant uniquement des opérations de lecture :

A. Transactions d'écriture : La séquence des opérations suivantes est exécutée par une transaction d'écriture qui réalise des écritures dans la mémoire partagée :

1. **Lire l'horloge globale de version :** Une transaction lit la valeur actuelle de l'horloge globale de version et l'enregistre dans une variable locale, appelé le numéro de

version lu (*rv*). Le *rv* est utilisé plus tard pour détecter les éventuelles modifications apportées aux données en le comparant avec le numéro de version du verrou d'écriture de ces données.

2. **Exécution spéculative :** Dans TL2 le code d'une transaction est exécuté spéculativement. Elle est dite spéculative parce que pendant l'exécute de transaction, elle n'a aucun effet sur la mémoire partagée. Une transaction s'exécute séparément et plus tard commite ses résultats. Une transaction maintient localement un ensemble de lecture et un ensemble d'écriture (Figure 4.6.1). lors d'une lecture d'un emplacement mémoire, le système vérifie d'abord en utilisant le filtre de *Bloom* [6] si l'adresse demandée apparaît déjà dans l'ensemble d'écriture de la transaction et ça pour récupérer la dernière valeur écrite dans l'adresse demandée (*read-after-write*). La lecture d'un objet est précédée par une consultation de verrou d'écriture associé pour s'assurer que l'objet n'est par acquis actuellement pour modification par une autre transaction. si le verrou est libre, le système récupère le numéro de version correspondant et assure que ce numéro soit $\leq rv$ (lu dans l'étape 1). S'il est plus grand que *rv* il implique que l'emplacement mémoire ait été modifié après que le thread actuel ait réalisé l'étape 1, et donc la transaction aborte.

3. **Verrouillage de l'ensemble d'écriture :** Dans cette étape la transaction acquiert les verrous correspondants à tous les emplacements mémoire figurant dans l'ensemble d'écriture, si elle réussit alors elle procède à la prochaine étape autrement, elle s'aborte et réessaie à nouveau.

4. **Incrémentation de l'horloge globale de Version :** Si une transaction acquiert avec succès tous les verrous de l'ensemble d'écriture, alors elle incrémente l'horloge globale de version et enregistre la nouvelle valeur dans une variable locale appelée numéro de version d'écriture (*wv*).

5. **Validation de l'ensemble de lecture :** Valider pour chaque emplacement dans l'ensemble de lecture que le numéro de version de verrou d'écriture associé soit inférieur ou égale au numéro de version lu *(rv).* En d'autres termes, on vérifie que ces emplacements mémoire n'ont pas été modifiés ni verrouillés actuellement par d'autres threads. En cas d'échec de validation, la transaction s'aborte. La validation de l'ensemble de lecture garantit que les emplacements mémoire lus n'ont pas été modifiés pendant l'exécution des étapes 3 et 4. Dans le cas particulier où le numéro de version lu plus 1 soit égale au numéro de version d'écriture c.à.d. ($rv + 1 = wv$), alors il n'est pas nécessaire de valider l'ensemble de lecture, car on est sûr qu'aucune transaction concurrente ne pourrait l'avoir modifié.

6. **Commiter et libérer les verrous :** La dernière étape est la phase de commit, Pour chaque emplacement dans l'ensemble d'écriture, copier à l'emplacement mémoire réel la nouvelle valeur à partir de l'ensemble d'écriture et libérer le verrou correspondant en mettant sa version à la version d'écriture *wv* et en effaçant le bit de verrou d'écriture (ceci est fait en utilisant un simple sauvegarde [16]).

A. Transactions de lecture seule : L'un des buts derrière la conception de TL2 est l'exécution efficace des transactions de lecture seule, car elles sont le type dominant dans plusieurs applications. L'exécution d'une transaction de lecture seule implique les deux étapes suivantes :

1. **Échantillonner l'horloge globale de version :** Charger la valeur courante de l'horloge globale de version et l'enregistrer dans une variable locale appelée le numéro de version lu *rv*.

2. **Exécuter à travers une exécution spéculative :** Exécuter le code de transaction. Chaque instruction de chargement est validée postérieurement en vérifiant que le verrou d'écriture correspondant soit libre et le numéro de version qu'il contient soit ≤ *rv*. S'il est plus grand que le *rv*, la transaction est abortée, autrement elle commite.

Comme on peut constater, l'implémentation des transactions de lecture seule est plus efficace parce qu'elle ne construit pas ou ne valide pas un ensemble de lecture. La détection d'un comportement de lecture seule peut être faite au niveau de chaque partie spécifique d'une transaction (par exemple, méthode ou bloc atomique). Ceci peut être fait au moment de la compilation ou en exécutant simplement toutes les méthodes d'abord en mode de lecture seule, et lors de détection de la première écriture, on arrête la transaction et on met un indicateur pour indiquer qu'on devrait dorénavant exécuter cette méthode en mode d'écriture [16].

4.6.2 Discussion

La difficulté principale avec l'horloge de version globale est qu'elle introduit plus de contention et un partage très cher de cache. Cependant, Dice et autres ont proposé une amélioration [16] qui consiste à diviser les métadonnées (les estampilles) de chaque objet pour inclure le numéro de version et l'identification (ID) du dernier thread qui a réalisé une modification à cet objet.

Pour obtenir sa nouvelle version d'écriture *wv*, un thread compare l'horloge globale actuelle avec la version d'écriture de sa propre transaction la plus récente. Si les chiffres diffèrent, le thread utilise l'horloge globale sans incrémentation. Si elles ne diffèrent pas, alors le thread incrémente l'horloge globale comme d'habitude. La combinaison de cette identité avec *wv* offre un *ID* unique pour la transaction. Dans le meilleur des cas, ça permet à chaque thread d'utiliser la même version avant d'incrémenter l'horloge globale. Ceci mènera à réduire le nombre des incrémentations de l'horloge globale de version.

Dice, Shalev et Shavit croient que TL2 est au moins dix fois plus rapide et plus robuste en termes de performance que le verrouillage simple. D'ailleurs TL2 peut facilement être utilisé avec le mécanisme transactionnel matériel [16]. Le TL2 a donné un nouvel espoir d'utiliser le verrouillage dans les STMs. Plus de travail est exigé dans TL2, pour l'intégrer avec l'approche matérielle et rendre l'horloge globale de version et les ensembles de lectures plus efficaces.

4.7 Dynamic Software Transactional Memory II (DSTM2)

Herlihy et autres dans [42] ont introduit la mémoire transactionnelle logicielle dynamique II (DSTM2), en se basant sur leurs premiers travaux sur le DSTM [44] discuté dans la section 4.3. DSTM2 est une bibliothèque logicielle basée sur Java. Il fournit un Framework flexible pour

mettre en application la mémoire transactionnelle logicielle. DSTM2 introduit un nouveau concept qui s'appelle l'*usine transactionnelle (*transactional factorie). Une *usine transactionnelle* transforme les interfaces séquentielles stylisées en structures de données synchronisées d'une façon transactionnelle. Les usines transactionnelles seront discutées plus tard dans la section 4.7.1.1. DSTM2 fournit aux utilisateurs la capacité d'intégrer leurs propres mécanismes de synchronisation et techniques de gestion de conflit et les techniques et les stratégies de recouvrement.

Tableau 4.7.1 : Caractéristiques de base de DSTM2.

DSTM2	
Synchronisation	Liberté d'obstruction ou Verrous
Contrôle de Concurrence	Optimiste
Granularité	Méthode
Stratégie de Mise à jour	Différée
Détection de Conflit	Tôt
Stratégie de résolution de conflit	Gestionnaire de conflit
Transactions emboitées	Non Supportées

DSTM2 ne supporte pas les transactions emboîtées et non plus l'attente conditionnelle car les auteurs croient qu'elles sont importantes uniquement dans le cas des longues exécutions, elles seront intégrées dans le future [42]. DSTM2 utilise une granularité au niveau méthode. DSTM2 fournit un thread, appelé *dstm2* qui intercepte l'appel d'une méthode, contrôle sa validité et décide si cette méthode peut commiter ou non.

4.7.1 Caractéristiques de la bibliothèque DSTM2

Dans DSTM2 plusieurs threads concurrents partagent les objets de données créés à partir des classes atomiques. Afin de déclarer une classe atomique dans DSTM2, une interface séquentielle est définie avec une collection de méthodes pour maintenir la consistance dans l'exécution des transactions [42], ensuite cette interface est passée à un constructeur d'une usine transactionnelle. L'usine transactionnelle utilise cette interface et crée une version synchronisée d'une classe anonyme, en implémentant l'interface et ses méthodes. Plus tard, des objets différents de cette classe anonyme peuvent être créés.

Les usines transactionnelles utilisent deux techniques différentes pour la synchronisation et la gestion de conflit ; la première est non bloquante (liberté d'obstruction) et la deuxième est bloquante (utilise des verrous). Les programmeurs ont la possibilité d'utiliser leurs propres techniques de gestion de conflit et de synchronisation tout en les intégrant à la bibliothèque DSTM2 [42].

DSTM2 est inspirée par l'expérience acquise de l'API DSTM qui était un bon API pour des buts expérimentaux mais il possède plusieurs imperfections [42]. Certaines de ces imperfections sont mentionnées ci-dessous ;

- Quand une transaction ouvre un objet en lecture, les autres transactions ne peuvent pas modifier cet objet pendant cette période.

- Quand une transaction ouvre un objet en écriture, les modifications apportées à cet objet seraient visibles par la suite à d'autres transactions qui l'ouvriront pour une lecture. Mais si nous renversons l'ordre de ce scénario, alors DSTM ne peut pas donner des résultats corrects. En d'autres termes, si une transaction a lu des valeurs d'un emplacement mémoire et plus tard cet emplacement mémoire a été écrit par une autre transaction, alors ces modifications ne peuvent pas être notées par la première transaction.

- Les références aux objets ouverts pour des opérations de lecture ou d'écriture sont valables pendant la durée de vie des transactions. Par conséquent, le programmeur doit faire une attention particulière pour la gestion des pointeurs et des références.

4.7.1.1 Les usines transactionnelles

La synchronisation transactionnelle est réalisée dans les appels de méthode [42] car il n'existe pas d'autre manière pour intercepter les accès aux champs directement. La définition des types atomiques à travers des interfaces permet aux usines transactionnelles de fournir leurs propres implémentations pour les méthodes déclarées dans les interfaces atomiques définies par l'utilisateur [42]. Une interface peut être implémentée par plus d'une classe. De même des usines multiples peuvent être fusionnées l'uns avec les autres pour construire une nouvelle usine, à condition que les transactions créées par ces usines aient un ordre et une séquence logiques (l'ordre dont lequel les transactions prennent l'effet) [42]. DSTM2 possède un package appelé, *dstm2.Thread* qui gère toutes les transactions, il fournit les fonctionnalités suivantes [42]:

- La validation : vérifier si la transaction peut commiter.

- Aborter une transaction quand une méthode ne peut pas commiter.

- Identification qu'une méthode particulière peut commiter et l'aider pour commiter quand elle peut, autrement la nettoyer et l'interrompre.

4.7.1.1.1 L'usine de base

Toutes les usines fournies par DSTM2 héritent de la classe *BaseFactory* qui implémente les tâches de base communes pour toutes les usines transactionnelles. Dans DSTM2 pour toutes les usines transactionnelles (décries ultérieurement), quand une transaction fait un accès à un objet, elle l'ouvre puis elle vérifie que cet objet n'est pas ouvert par une autre transaction conflictuelle. Si c'est le cas alors elle consulte le gestionnaire de conflit pour décider d'attendre le commit de la transaction contradictoire ou de s'aborter et recommencer à nouveau [42].

Pour illustrer comment DSTM2 supporte des usines transactionnelles hétérogènes, nous décrivons maintenant les implémentations de deux usines qui représentent deux approches complètement différentes pour implémenter des objets atomiques.

4.7.1.1.2 L'usine libre d'obstruction

L'usine libre d'obstruction est basée sur l'algorithme de liberté d'obstruction introduit précédemment dans DSTM [44]. La structure des objets créés par l'usine libre d'obstruction sont représentés dans trois niveaux suivant les indications de la Figure 4.7.1. Le premier niveau est un entête (*start*) contenant la référence au repère. Le repère a trois attributs, l'un d'entre eux est une référence à la nouvelle version *(New Object)* de l'objet et le seconde est une référence à l'ancienne version de l'objet *(Old Object)*. Le troisième attribut contient la référence au descripteur de transaction de la dernière transaction qui ouvre l'objet pour une écriture. La valeur

logique de l'objet est à l'ancien objet si la transaction est active ou elle a aborté, et elle est au nouvel objet si elle a commité.

L'usine libre d'obstruction est responsable de l'exécution non bloquante d'une transaction. Le mécanisme de fonctionnement de cette usine est identique à celui discuté dans la section 4.3.2 pour l'exécution libre d'obstruction des transactions par DSTM [42].

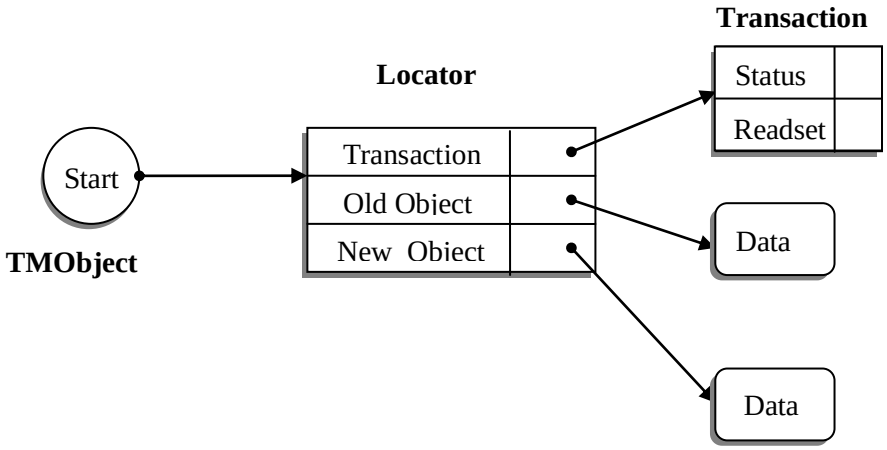

Figure 4.7.1: La Structure des objets créés par l'usine libre d'obstruction [42]

4.7.1.1.3 L'usine d'ombre

L'usine d'ombre (*Shadow Factory*) utilise une courte section critique gérée par le système (utilise des verrous) pour l'exécution des transactions. L'usine d'ombre maintient pour chaque attribut défini dans l'interface, des zones d'ombre *(shadow field)* près des attributs réguliers comme indiqué dans la Figure 4.7.2 [42]. L'usine d'ombre évite les surcoûts d'adressage indirect et d'allocation mémoire engendrés par l'usine libre d'obstruction. Cependant, l'usine d'ombre n'est pas bien adaptée pour la programmation multithread (où plusieurs transactions partagent un seul processeur) [42]. L'usine d'ombre utilise la méthode *backup()* pour copier chaque zone régulière définie dans une interface à sa zone d'ombre suivant les indications de la Figure 4.7.2. la méthode *restore()* fait un copiage dans le sens inverse. Quand une transaction ouvre un objet, elle vérifie si la dernière transaction qui a écrit à cet objet a commité ou a aborté. Si elle a commité alors ceci implique que les valeurs courantes sont dans les champs réguliers et donc la transaction appelle la méthode *backup()* pour les copier dans les champs d'ombre suivant les indications de la Figure 4.7.3. Cependant, si elle a aborté ceci implique que les zones d'ombre contiennent les valeurs courantes. Par conséquent, la méthode *restore()* est appelée pour copier les valeurs originales à partir des zones d'ombre aux zones régulières suivant les indications de la Figure 4.7.4. En d'autres termes, l'usine d'ombre a la capacité de restaurer et maintenir l'état de transaction dans les deux cas c.à.d. après un abort ou un commit.

DSTM2 possède les nouvelles caractéristiques et avantages suivants [42] :

- DSTM2 est le premier Framework flexible pour programmer des applications STM.
- DSTM2 ne dépend pas des modifications de compilateur ou du système d'exécution.
- DSTM2 est simple, flexible et portable.

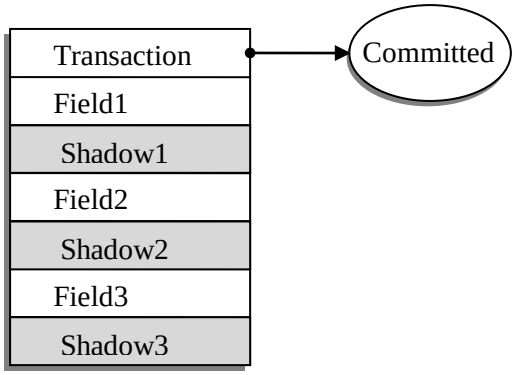

Figure 4.7.2: Structure d'un objet créé par l'usine d'ombre [42]

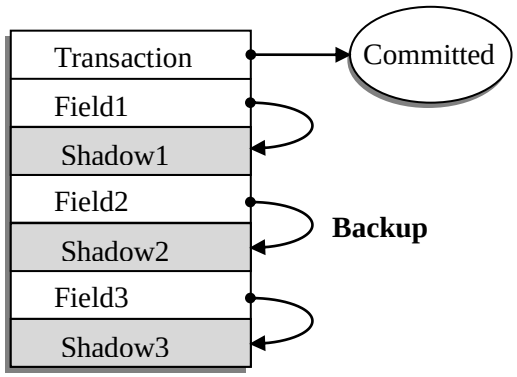

Figure 4.7.3: Ouverture d'un objet d'usine d'ombre après un commit récent [42]

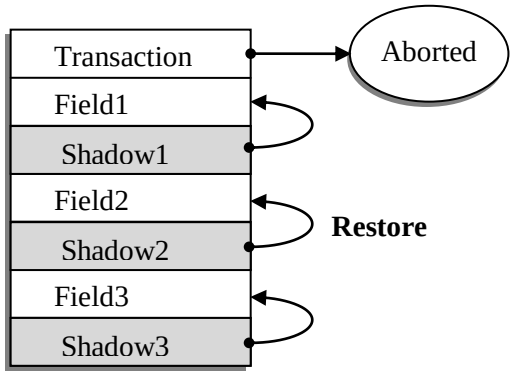

Figure 4.7.4: Ouverture d'un objet d'usine d'ombre après un abort récent [42]

4.7.2 Discussion

DSTM2 est implémentée dans Java qui a un mécanisme intrinsèque de récupération de place. En d'autres termes DSTM2 dépend de l'environnement d'exécution du langage Java concernant la gestion de la mémoire. D'ailleurs DSTM2 crée une copie de chaque classe et la transforme à une usine transactionnelle ce qui épuise par la suite la mémoire [18] bien qu'une seule version de chaque classe puisse être suffisante au lieu de créer deux versions. La réparation de ce problème a besoin d'un travail considérable sur DSTM2.

4.8 Rochester Software Transactional Memory (RSTM)

Le système de mémoire transactionnelle logicielle de Rochester (RSTM) [63] de Marathe et autres développé en 2006 est conçu pour les langages de programmation qui n'ont pas une ramasse miettes. RSTM est développé comme une bibliothèque C++ mais une bibliothèque équivalente peut être développée dans le langage C. RSTM était un effort pour réduire le plus possible le surcoût engendré par les systèmes STM précédant, par exemple OSTM, ASTM et DSTM tout en exploitant leurs bonnes qualités. Les caractéristiques conceptuelles de base de RSTM sont affichées dans le Tableau 4.8.1.

Tableau 4.8.1 : Caractéristiques conceptuelles de base de RSTM.

RSTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Objet
Stratégie de Mise à jour	Différée
Stratégie de Détection de Conflit	Tôt ou Tard (Selective)
gestionnaire de conflit	Gestionnaire de Conflit
Transactions emboitées	Aplaties

RSTM est un STM non bloquant (libre d'obstruction) avec une granularité de niveau objet, il utilise une mise à jour différée. Il peut utiliser les deux stratégies de détection de conflit (tôt et tard). RSTM possède les améliorations suivantes par rapport aux STMs précédents [63] ;

- RSTM utilise l'adressage indirect d'un seul niveau pour accéder aux objets de données au lieu de deux niveaux utilisés par ses prédécesseurs comme DSTM et ASTM, ceci implique un changement dans la structure des objets où deux champs supplémentaires sont ajoutés.

- RSTM utilise son propre ramasse miettes et évite l'allocation dynamique de la plupart de ses structures de données. Grâce à son propre ramasse miettes RSTM peut fonctionner avec des langages qui n'ont pas un tel mécanisme, par exemple le C++.

- RSTM emploie l'*invalidation* pour éviter les lectures inconsistantes. Il utilise une nouvelle heuristique pour dépister les lecteurs d'un objet, ce qui réduit le surcoût engendré par le *bookkeeping*[4] trouvé dans les systèmes de ce type précédents. RSTM

[4] Opération d'acquisition et de libération des objets et la gestion des ensembles de lectures et écriture privés.

utilise des listes de lecture et d'écriture statiquement allouées pour minimiser le surcoût de bookkeeping dans le cas générale.

- RSTM supporte plusieurs options pour la gestion de contention et la détection de conflit, ce qui lui permet de s'adapter avec les différentes charges de travail.

4.8.1 Détails d'implémentation

Dans RSTM, chaque objet est accédé à partir de l'*ObjectHeader* qui est unique pendant la durée de vie de l'objet. L'*ObjectHeader* pointe directement sur la version courante de l'objet comme il est indiqué sur la Figure 4.8.1. RSTM utilise le bit de poids faible du pointeur *Newdata* de l'*ObjectHeader* comme un flag *(clean bit)* qui indique si l'élément de données est la version courante ou pas. Si le flag est mis à zéro, il signifie qu'aucune transaction n'a ouvert l'objet pour écriture et donc l'*ObjectHeader* pointe directement sur la version courante de l'objet de données et les champs *Owner* et *Olddata* de l'objet de données n'ont pas d'importance c.à.d. *NewDataObject* est la version courante. Cependant, quand le flag est mis à un, il indique que cet objet de données est actuellement ouvert par une autre transaction, le champ *Owner* de l'objet de données est valide et pointe sur le *TransactionDescripteur* de la transaction propriétaire qui possède l'objet. Si le champ *status* de la transaction propriétaire est à *COMMITTED* alors l'objet *NewDataObject* est la version courante. Si le *status* de la transaction propriétaire est à *ABORTED* alors le pointeur *OldData* est valide et pointe vers la version courante de l'objet c.à.d. *OldDataObject* est la version courante. Si le *status* de la transaction propriétaire est à *ACTIVE* alors aucune autre transaction ne peut accéder à l'objet que ce soit pour une lecture ou écriture sans aborter la transaction propriétaire.

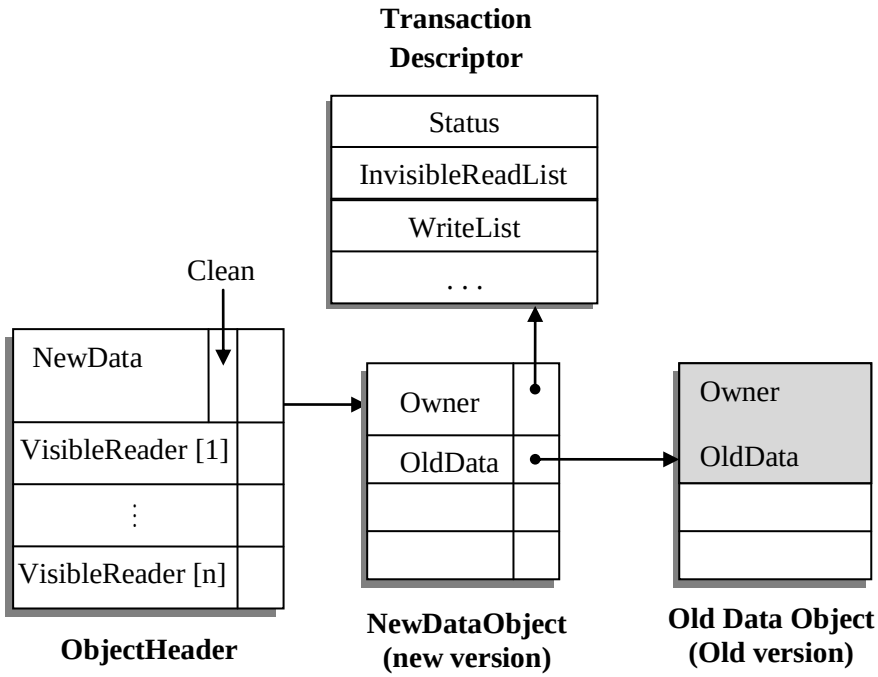

Figure 4.8.1: Les structures de données de RSTM [63].

Une autre caractéristique importante introduite par RSTM est les listes de lecture visibles et invisibles [63]. L'*ObjectHeader* possède une liste de lecture de taille fixe, appelée la *liste de lecture visible* (*VisibleReader[]*), qui contient des entrées pour les transactions qui lisent l'objet

actuellement. Quand une transaction ouvre un objet pour une lecture, elle sera ajoutée à la liste visible de lecture de cet objet. Quand une transaction ouvre un objet pour une modification elle aborte toutes les transactions figurant dans la liste de lecture visible de cet objet. Donc les transactions figurant dans les listes de lecture visibles des objets n'ont pas besoin de valider leurs lectures car une transaction d'écriture contradictoire aborte ces transactions. Cependant, s'il n'y a pas d'espace pour ajouter une transaction à la liste de lecture visible d'un objet (liste de lecture pleine), alors la transaction doit ajouter l'objet à sa propre liste privée de lecture, appelée la *liste invisible de lecture*. La transaction doit valider sa liste de lecture invisible. Si un objet de données a été changé depuis sa dernière lecture par une transaction, alors la liste invisible de lecture d'une transaction aidera la transaction pour valider sa version de lecture et pour s'interrompre s'il est nécessaire. Par conséquent, la liste visible de lecture peut minimiser la taille de la liste de lecture invisible et le coût de sa validation.

TransactionDescriptor maintient aussi une liste d'écriture qui contient les objets de données accédés en écriture par la transaction. Quand une transaction termine (commite avec succès), un thread parcourt sa liste d'écriture pour nettoyer les éléments de données qu'elle a accédé. En d'autres termes, quand une transaction réussit à commiter, elle utilise un CAS pour changer le flag *(clean bit)* de un à zéro pour tous les objets de données qu'elle a accédé pour une opération de commit particulière. Si la transaction aborte, le thread exécutant essaie de changer le pointeur de la copie sale vers la copié juste. Si l'opération de changement échoue, alors un autre thread a déjà réalisé l'opération de nettoyage ou a ultérieurement acquis l'objet. Dans les deux cas, le thread courant marque l'objet de données comme n'est plus valide pour une éventuelle récupération. Une fois que le thread atteint la fin de sa liste d'écriture, il sait qu'il n'existe aucune référence dans le *TransactionDescriptor* et donc il peut réutiliser ce descripteur pour la prochaine transaction [63].

Lors de début d'une transaction, elle met le statut de son descripteur à Active et ouvre les objets désirés avant de les y accédés, et par la suite :

1. Si la transaction ouvre l'objet pour une modification, la transaction doit acquérir la propriété de l'objet en exécutant les opérations suivantes :

   a. Lire le pointeur *NewData* de l'*ObjectHeader* de cet objet et s'assurer qu'aucune autre transaction ne le possède. S'il est déjà possédé, le gestionnaire de conflit est consulté (en supposant ici que le gestionnaire de conflit arrête l'autre transaction en faveur de cette transaction.)

   b. Allouer le *NewDataObject* et copier les valeurs à partir de la version courante de l'objet.

   c. Initialiser les pointeurs *Owner* et *OldData* du nouvel objet.

   d. utiliser un CAS pour faire un changement atomique du pointeur lu dans l'étape (a) avec un pointeur vers la nouvelle copie allouée.

   e. Ajouter l'objet à la liste privée d'écriture de la transaction.

   f. Réitérer sur la liste de lecteur visible de l'objet en interrompant toutes les transactions qu'elle contient.

2. Si la transaction ouvre l'objet pour une lecture, et la liste visible de lecture de l'objet n'est pas pleine, ajouter la transaction à cette liste. Si la liste est pleine ajouter l'objet à la liste de lecture privée de la transaction.

3. Contrôler le champ *Status* dans le descripteur de transaction, pour s'assurer que la transaction n'a pas été abortée par d'autre transaction.
4. Valider progressivement tous les objets figurant dans la liste de lecture privée de la transaction.

4.8.2 Discussion

La caractéristique conceptuelle la plus importante offert par RSTM est qu'il peut fonctionner sans ramasse miettes et peut récupérer ses structures de données à la fin d'une transaction [63]. Cependant, les listes de lecture introduites par RSTM produisent plus de trafic et deviennent coûteuses. Le papier présente une comparaison de performance de RSTM en utilisant des différents gestionnaires de conflit, et notamment les lectures *visibles* vs *invisibles* et les mécanismes d'acquisition de propriété *Tôt* vs *Tard*.

L'étude empirique de RSTM a montrée que malgré le surcoût de la validation de l'ensemble de lecture invisible, elle reste plus rapide dans la plupart des cas [63] car la liste visible introduit plus de contention au niveau de cache. De même la stratégie de détection de conflit en retard est plus efficace que la détection de conflit tôt car dans la détection de conflit en retard plus de transactions ont une chance de commiter avec succès. La mise en application de l'ensemble de lecture visible nécessite le développement d'un nouveau gestionnaire de conflit qui prend en considération les lecteurs visible.

4.9 Time Based Transactional Memory (TbSTM)

T. Riegel, C. Fetzer et P. Felber ont présenté en 2006 TbSTM [74], un nouveau système de mémoire transactionnelle logicielle basé sur le temps. TbSTM utilise la notion de temps pour maintenir la consistance dans les accès aux données et l'ordre dans lesquels les transactions commitent leurs résultats. Les anciennes implémentations des systèmes de mémoire transactionnelles basés sur le temps ont utilisé une horloge (compteur) unique qui est incrémentée chaque fois qu'une transaction commite avec succès. Cependant, cette technique est peu efficace dans le cas des grands systèmes avec un taux de commit élevé où la contention sur le compteur global unique devient un goulot important. Les caractéristiques fondamentales du TbSTM sont affichées dans le Tableau 4.9.1.

4.9.1 Détails d'implémentation

Le STM de Riegel utilise l'estampillage pour éviter le surcoût engendré par la validation [74]. En d'autres termes, TbSTM n'a pas besoin de valider chaque élément de données consulté pour assurer la consistance des données. Cependant, un STM basé sur le temps utilise la notion de temps pour raisonner sur l'ordre dont lequel les transactions commitent [74]. Ceci permet au système d'accéder toujours à des objets consistants (comme s'ils ont été validés après chaque lecture) sans souffrir du surcoût de validation [74].

TbSTM est un système de mémoire transactionnelle qui utilise des lectures invisibles mais qui garantit la consistance des données en maintenant toujours un *snapshot consistant* des données pour les transactions. Le STM de Riegel n'exige aucune implémentation fondamentale

spécifique, en d'autres termes, il peut être implémenté au niveau mot, au niveau objet ou même avec l'approche matérielle [74]. Cependant, l'information de temps (*l'estampille*) doit être stockée avec chaque objet.

Tableau 4.9.1 : Les caractéristiques de base de TbSTM.

TbSTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Objet, Mot (n'importe)
Stratégie de Mise à jour	différée
Stratégie de Détection de Conflit	Tôt
gestionnaire de conflit	Gestionnaire de Conflit
Transactions emboitées	Non Supportées

Les STMs basés sur le temps utilisent une base de temps (par exemple une horloge sous forme d'un compteur partagé) pour imposer un ordre total entre les transactions et les versions des objets. Cette horloge peut être incrémentée lors de commit d'une transaction ou elle peut fonctionner indépendamment comme les horloges de temps réel [74]. Cette technique est peu évolutive, pour cela, TbSTM utilise une base de temps plus évolutive qui ne souffre pas de la dégradation due à la contention.

Dans TbSTM, il existe deux types de transactions [74] : une transaction de *lecture seule* qui ne modifie aucun objet, et une transaction de *mise à jour* qui modifie au moins un objet. Un objet peut avoir une séquence de version lorsqu'il est modifié par une transaction. Une nouvelle version devient *valide* quand la transaction qui a modifié l'objet commite.

Chaque objet a une version v qui est *valable pour* une certaine durée de temps appelée l'*intervalle de validité* $(v.R)$. L'intervalle de validité est l'intervalle entre le moment où la version v devient *valide* et le moment où elle devient *périmée* par une autre version. Les bornes inférieures et supérieures de l'intervalle $v.R$ sont référées par $\lfloor v.R \rfloor$ et$\lceil v.R \rceil$. Une version qui est encore valide a une limite supérieure mise à ∞. De même une transaction T possède un intervalle de validité $T.R$ qui est l'intervalle de temps dans lequel toutes les versions des objets $T.O$ consultés par cette transaction sont valides c.à.d. $T.R$ est l'intersection des intervalles de validité des différentes versions des objets consultés par T (en supposant que les objets sont accédés uniquement par les transactions). $\lfloor T.R \rfloor$ et $\lceil T.R \rceil$ dénotent les limites inférieures et supérieures de $T.R$ réspectivement. On dit que les versions des objets consultées par la transaction T forment un *snapshot consistant* si l'intervalle de validité $T.R$ ne soit pas vide [74]. Quoique les différents processeurs dans TbSTM puissent avoir des références de temps distinctes, les valeurs de temps de commit des transactions et les intervalles de validité des objets sont convenues. Il est a noté que malgré que les valeurs sont convenues, un thread pourrait seulement pouvoir rapprocher une estampille distante ts avec sa propre horloge locale, par exemple, il pourrait pouvoir calculer que ts a été lu entre les estampilles locales $ts1$ et $ts2$.

Une transaction T peut essayer d'étendre son $T.R$ pour éviter qu'il devienne vide. Quand T accède aux versions les plus récentes des objets figurant dans $T.O$, la limite supérieure de son intervalle de validité est mise au temps actuel (et non pas à ∞) parce que T ne peut pas prévoir si ou quand les objets seront changés. En étendant $T.R$, la transaction vérifie si quelques objets ont été changés. S'ils ne sont pas changés, l'intervalle de validité peut être étendu au temps actuel (c.à.d. le temps avant que la vérification ait été réalisée). Autrement $\lceil T.R \rceil$ sera mis au temps de commit le plus tôt de toutes les versions des objets qui remplacent une version dans $T.O$ [74].

Riegel et autres ont présenté un nouvel algorithme STM appelé *LSA (Lazy Snapshot Algorithme)* qui utilise les horloges de temps réel parfaites[5] pour synchroniser d'une manière optimiste les transactions concurrentes (la Figure 4.9.1) [74]. (L'algorithme LSA peut être modifié pour utiliser des horloges extérieurement synchronisées quelles donnent à tous les threads l'accès à une horloge globale mais avec une erreur de lecture, cette erreur peut varier et peut ne pas être bornée). L'étude empirique a prouvé que les horloges de temps réel sont plus évolutives dans des grands systèmes [74]. L'horloge globale n'a pas besoin réellement d'être une horloge réelle. En d'autres termes, ni la vitesse ni les valeurs pour ces horloges ne doivent être synchronisées avec le temps réel. Les horloges locales des processeurs rapprochent leur temps avec les horloges de temps réel, qui aident pour obtenir une valeur de temps unique et minimise les erreurs de synchronisation [74].

LSA utilise les fonctions suivantes (Algorithme 1 de la Figure 4.9.1) que leurs implémentations dépendent de la base du temps utilisée [74]. Les deux méthodes *GETTIME* et *GETNEWTS* sont utilisées pour obtenir l'estampille. La méthode *GETTIME* retourne le temps actuel tandis que *GETNEWTS* retourne une estampille qui est plus grande que n'importe qu'autre temps utilisé par cette transaction. Cependant, les estampilles ne doivent pas nécessairement être propres pour une transaction, les autres transactions peuvent utiliser la même estampille. Dans certains cas, il peut être difficile de trouver quelle est l'estampille qui a été lue le plus tard et laquelle qui est lue le plus tôt. Supposant qu'un temps $t1$ a été lu non plus tard que le temps $t2$. Dans ce cas, éventuellement on peut dire que $t1$ et $t2$ sont égaux mais il n'est pas possible que $t1$ pourrait être plus grand que $t2$, noté $t2 \succcurlyeq t1$. Dans certains cas on peut uniquement dire que $t2$ est possiblement lu après $t1$, noté $t2 \succsim t1$. (Il est à noter que $t2 \succcurlyeq t1$ implique toujours que $t1 \succsim t2$ n'est pas vérifié et $t2 \succsim t1$ implique que $t1 \succcurlyeq t2$ n'est pas vérifié).

Les fonctions *max(t1, t2)* et *min(t1, t2)* ont la sémantique suivante. Pour n'importe quelle estampille $t3$, si $t3$ est garanti d'être plus tard que *max(t1, t2)* alors le $t3$ est garanti d'être après $t1$ et $t2$. De même, pour n'importe quelle estampille $t3$ qui est garantie pour être plus tôt que le *min(t1,t2)*, alors $t3$ est garanti d'être plus tôt que $t1$ et $t2$. L'algorithme 4 montre l'implémentation de ces fonctions.

La tache principale du LSA est de construire des *snapshots consistants* pendant l'exécution d'une transaction et d'étendre *paresseusement* l'intervalle de validité sur demande. Ce qui permet aux transactions de travailler sur des *snapshots consistants et* par conséquent de lire toujours des données *consistantes* en permettant à une transaction *T* de commiter seulement quand elle peut étendre son *intervalle de validité* jusqu'à un certain temps qui inclut le moment de commit, ce qui garantit qu'aucune autre transaction n'a modifié aucun des objets de *T* jusqu'au moment de commit [74]. Autrement la transaction sera abortée par le gestionnaire de conflit.

[5] Une horloge parfaite donne à tous les threads l'accès à une horloge globale sans aucune erreur, c.à.d. pas de différence entre la valeur lue et la valeur réelle.

4.9.1.1 Détails de l'algorithme LSA

L'ensemble des objets consultés par une transaction et leurs versions sont déterminés pendant l'exécution d'une transaction. L'intervalle de validité $T.R$ est donc construit progressivement. Quand une transaction T commence, la limite inférieure de son intervalle de validité est mise au temps actuel récupéré par la fonction *GETTIME* (la ligne 3), c.à.d. la transaction ne peut pas exécuter dans le passé. Les estampilles retournées par la fonction à n'importe quel thread sont garanties d'être incrémentées d'une manière mono-tonique, mais pas strictement (un thread peut lire plus d'une fois la même estampille).

En accédant à la version la plus récente d'un objet *o*, on ne connaît pas encore quand cette version sera remplacée par une nouvelle version. On obtient donc un intervalle approximatif de validité *r* en obtenant la dernière version de l'objet *o* (ligne 12) et en calculant une limite inférieure sur son intervalle de validité maximum appelée la *limite supérieure préliminaire* sur l'intervalle de validité (voir la ligne 29). La fonction *GETPRELIMUB* est utilisée pour recalculer la limite supérieure préliminaire d'une version d'un objet selon la référence du temps du thread actuel.

Pendant l'exécution d'une transaction, le temps pourrait avancer et les intervalles préliminaires de validité pourraient rester valide plus de temps. On peut essayer d'étendre $T.R$ en recalculant sa limite inférieure (ligne 23 de l'algorithme 2 et lignes de 1-6 dans algorithme 3). Les extensions ne sont pas exigées pour l'exactitude, mais elles augmentent la chance qu'une version appropriée de l'objet soit disponible.

Si l'intervalle de validité r de la dernière version de l'objet *o* n'a pas une intersection avec $T.R$ et la transaction est en lecture seule, on peut rechercher une version plus ancienne de *o* dont l'intervalle superpose avec $T.R$ (l'algorithme demande le plus récent parmi les versions superposâtes valides, mais n'importe version peut le faire). La nouvelle valeur de $T.R$ est calculée comme l'intersection de la valeur précédente et l'intervalle de validité de la version consultée (les lignes 28-29). La transaction doit s'aborter si aucune version appropriée ne peut être trouvée (ligne 31 de l'algorithme 2 et ligne 11 de l'algorithme 3). Par la construction de $T.R$, LSA garantit qu'une transaction qui a commencé au moment t possède un *snapshot* qui est valide au moment ou après que la transaction ait commencé c.à.d. $\lfloor T.R \rfloor \geqslant$ t. par conséquent, une transaction de lecture seule peut commiter si elle a utilisé un snapshot consistant, c.à.d. $T.R$ n'est pas vide.

Une transaction T peut commiter seulement quand elle peut étendre son *intervalle de validité* jusqu'à un certain temps qui inclut le moment de commit, ce qui garantit qu'aucune autre transaction n'a modifiée aucun des objets de T jusqu'au moment de commit [74]. De cette façon plusieurs transactions peuvent commiter tant que leurs *intervalles de validité* ne sont pas en conflit [74]. La limite supérieure préliminaire d'une version d'un objet écrit par T est surestimée par 1 (la ligne 27 dans l'algorithme 3). Pour simplifier le test dans le *COMMIT()* : on sait que T essaiera de commiter une nouvelle version de *o* à $T.CT$ mais, le plus important c'est qu'on sait également qu'aucune autre transaction ne peut commiter une nouvelle version de l'objet *o* jusqu'à $T.CT + 1$ si T peut en effet commiter.

Algorithme 1 -Fonctions d'utilité générique

```
1: use function GETTIME()                          ▷ Get current timestamp
                                                   ▷ (module-specific)
2: use function GETNEWTS()                         ▷ Get strictly greater timestamp
                                                   ▷ (module-specific)
3: use function ≽(t1, t2)                          ▷ Guaranteed later than or equal
                                                   ▷ (module-specific)

4: function ≿(t1, t2)                              ▷ Possibly later than
5:   return t2 ⋡ t1
6: end function
7: use function max(t1, t2)                        ▷ Maximum (module-specific)
8: use function min(t1, t2)                        ▷ Minimum (module-specific)
```

Algorithme 2 -LZA temps réel

```
1: procedure START(T)                              ▷ Initialize transaction attributes
2:   T.CT ← 0                                      ▷ T's commit time
3:   T.R ← [GETTIME(), ∞]                          ▷ T's validity range
4:   T.O ← ∅                                      ▷ Set of objects versions accessed by T
5:   T.update ← false                             ▷ T starts as a read-only transaction
6:   T.status ← active                            ▷ T is active
7: end procedure
8: procedure OPEN(T, o, m)                         ▷ Opens o in mode m (read/write)
   ▷ To simplify, we assume an object is opened at most once per T
9:   if m = write then
10:    T.update ← true
11:    repeat
12:      v_c ← GETVERSION(T, o, [⌊T.R⌋, ∞])
                                                   ▷ Get latest committed version
13:      v ← clone(v_c)                            ▷ Create new copy for writing
14:      v.T ← T                                   ▷ Current transaction is writer
15:      T_w ← o.writer
16:      if T_w ≠ null ∧ T_w.status ∉ {aborted, committed} then
17:        solveConflict(o, T, T_w)                ▷ Contention manager ...
18:      else                                      ▷ ...arbitrates and aborts the loser
19:        C&S(o.writer, T.w, T)                   ▷ Try registering as writer
20:      end if
21:    until o.writer = T
22:    if ⌊v.R⌋ ≿ ⌈T.R⌉ then                       ▷ Is the version too recent?
23:      EXTEND(T)                                 ▷ Extend as much as possible
24:    end if
25:  else
26:    v ← GETVERSION(T, o, T.R)
                                                   ▷ Get latest committed version in interval
27:  end if
28:  ⌊T.R⌋ ← max(⌊T.R⌋, ⌊v.R⌋)
29:  ⌈T.R⌉ ← min(⌈T.R⌉, GETPRELIMUB(T, o, v, ⌈T.R⌉))
30:  if ⌊T.R⌋ ≿ ⌈T.R⌉ then                         ▷ Possibly inconsistent?
31:    ABORT(T)                                    ▷ Yes: abort (and terminate execution)
32:  end if
33:  T.O ← T.O ∪ {(o, v)}                         ▷ Access object versions
34: end procedure
```

```
35: procedure COMMIT(T)                          ▷ Try to commit transaction
36:   if ¬T.update then
37:     C&S(T.status, active, committed)
                                                 ▷ Validation not necessary
38:   else
39:     C&S(T.status, active, committing)        ▷ Start committing
40:     if T.status = committing then
41:       t ← GETNEWTS()
                          ▷ Tentative commit time (may not be unique)
42:       C&S(T.CT, 0, t)                        ▷ Try imposing our timestamp
43:       for all (o, v) ∈ T.O do                ▷ Are versions still valid at t?
44:         ub ← GETPRELIMUB(T, o, v, T.CT)
45:         if T.CT ≻̃ ub then
46:           ABORT(T)                           ▷ No: abort (and terminate execution)
47:         end if
48:       end for
49:       C&S(T.status, committing, committed)   ▷ Yes: commit
50:     end if
51:   end if
52: end procedure

53: procedure ABORT(T)                           ▷ Abort transaction (unless committed)
54:   if ¬C&S(T.status, active, aborted) then    ▷ Still active?
55:     C&S(T.status, committing, aborted)       ▷ Committing?
56:   end if
57:   if T.status = aborted then                 ▷ Aborted?
58:     throw AbortedException in T              ▷ Terminate execution
59:   end if
60: end procedure
```

Algorithme 3 -Fonctions d'aide

```
1: procedure EXTEND(T)
                          ▷ Try to extend T's validity range to at least t
2:   ⌈T.R⌉ ← GETTIME()
3:   for all (o, v) ∈ T.O do
                          ▷ Recompute the upper bound on validity range
4:     ⌈T.R⌉ ← min(⌈T.R⌉, GEPRELIMTUB(o, v, ⌈T.R⌉))
5:   end for
6: end procedure

7: function GETVERSION(T, o, R)
                          ▷ Get latest version of o overlapping R
8:   loop
9:     v ← latest version of o s.t. ⌈v.R⌉ ⋡ ⌊R⌋ ∧ ⌈R⌉ ⋡ ⌊v.R⌋∧
         (v.T = null ∨ v.T.status ∈ {committing, committed})
10:    if v = null then                          ▷ Any valid version?
11:      ABORT(T)                  ▷ No: abort (and terminate execution)
12:    else if v.T ≠ null ∧ v.T.status = committing then
13:      COMMIT(v.T)  ▷ Help committing transaction to complete
14:    else
15:      return v                  ▷ Always return a commited version
16:    end if
17:  end loop
18: end function
```

```
19: function GETPRELIMUB(T, o, v, t)
                                  ▷ Get conservative estimate on ⌈v.R⌉
20:   T_w ← o.writer
21:   if ⌈v.R⌉ ≠ ∞ then                                  ▷ Still open?
22:     return ⌈v.R⌉                  ▷ No: return version upper bound
23:   else if T_w ≠ null then  ▷ Yes: only T_w may set UB before t
24:     if T_w.status ∈ {committing, committed} then
25:       if T_w.CT > 0 then
26:         if T_w = T then
27:           return T_w.CT        ▷ Off by 1 but simplifies COMMIT
28:         else
29:           return T_w.CT − 1▷ Version valid at least until then
30:         end if
31:       end if
32:     end if
33:   end if
34:   return t          ▷ Return caller's timestamp (GETTIME() ⩾ t)
35: end function
```

Figure 4.9.1: Le pseudo code de l'algorithme LZA-temps réel [74].

Le commit d'une transaction de mise à jour (les lignes 35-52) se fait en deux phases [74]. D'abord la transaction entre l'état de commit avant de déterminer si elle peut commiter ou elle doit s'aborter. La raison pour maintenir le statut de transaction en utilisant l'opération *CAS* est qu'un autre thread peut aider la transaction pour commiter ou la forcer pour s'aborter c.à.d. Un thread de commit essaiera de mettre l'estampille obtenue à partir de sa référence de temps local en tant que temps de commit de la transaction. S'il échoue c.à.d. un autre thread a mis le temps de commit à l'avance, alors le thread courant utilise le temps de commit *T.CT* définit précédemment. Le thread vérifiera par la suite si la limite supérieure de l'intervalle de validité de la transaction peut être étendue pour inclure *T.CT*. La transaction peut commiter seulement si ceci réussira parce qu'autrement quelques objets consultés par *T* pourraient avoir été modifiés par une autre transaction qui a commis avant *T.CT*.

S'il est possible de mettre à jour la version la plus récente d'un objet (c-à.d. $T.R$ reste non vide), *LSA-RT* marque d'une façon atomique l'objet *o* qu'il a écrit (écriture visible) par l'enregistrement de lui-même dans *o.writer*. Quand une autre transaction essaie d'écrire le même objet, elle verra la marque et détectera un conflit (lignes 16 - 17). Dans ce cas, le gestionnaire de conflit est consulté pour décider quelle est la transaction quelle doit attendre ou être abortée.

4.9.2 Discussion

Le STM basé sur le Temps utilise la notion de temps pour synchroniser les transactions concurrents. Les horloges de temps réel sont mieux que des compteurs globaux simples. Les horloges de temps réel (des horloges externes ou des horloges physiques synchronisés) évitent le conflit souffert par les compteurs globaux simples, qui est un goulot pour des transactions courtes ou quand le système est très grand et exécute plusieurs transactions [74]. L'algorithme de LSA utilise différentes bases de temps. Cependant, il y aurait une différence en l'utilisant pour des autres systèmes. Un simple compteur de temps partagé peut être suffisant pour les systèmes réduits. Tandis que dans des grands systèmes, les horloges externes réalisées par le matériel peuvent être plus efficaces.

4.10 Hybrid Transactional Memory (HybTM)

S. Kumar, M. Chu, C.J Hughes, P. Kundu et A. Nguyen Ont proposé en 2006 HybTM [62], un système de mémoire transactionnelle hybride basé sur l'objet. Une transaction dans Le HybTM s'exécute initialement comme une transaction matérielle et quand elle s'échoue (dépassement des capacités des ressources matérielles), elle sera exécutée comme une transaction logicielle dans un STM basé sur l'objet (Le DSTM) [62]. Le système bénéficie de la performance des transactions matérielles avec la flexibilité de l'approche logicielle. Une approche similaire (HyTM [12]) a été proposée par Moir et autres qui sera discutée dans la section 4.11 et NZTM [82] par Faud et autres qui sera présentée dans la section 4.12. Les détails de conception de base de l'HybTM de Kumar et autres sont affichés dans le Tableau 4.10.1.

Tableau 4.10.1 : Les caractéristiques de base de HybTM.

HybTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste et Pessimiste
Granularité	Objet et Ligne de cache
Stratégie de Mise à jour	Différée et Directe
Stratégie de Détection de Conflit	Tard et Tôt
Gestionnaire de conflit	Polit et abort
Transactions emboitées	Supportées

4.10.1 Détails d'implémentation

Une transaction peut choisir Le mode d'exécution matériel ou logiciel indépendamment des autres transactions et ouvre tous ses objets dans le même mode choisi au démarrage. Cependant, l'approche logicielle (STM) est totalement différente de l'approche matérielle (HTM). La différence de base entre les deux approches est que l'approche STM détecte les conflits au niveau objet tandis que l'approche HTM a la granularité au niveau ligne de cache [62]. Le protocole de cohérence du cache est employé pour détecter les conflits entre les transactions logicielles et matérielles [62].

HybTM est basé généralement sur la proposition initiale de Herlihy [41] mais la conception de la puce est légèrement plus complexe et plus large en termes d'espace pour pouvoir supporter le schéma d'exécution hybride. Un buffer est aussi utilisé pour stocker les données transactionnelles. L'architecture *ISA* (Instruction Set Architecture) a été utilisée pour plus de flexibilité. L'*ISA* est la partie d'un processeur qui est visible au programmeur ou au compilateur et se sert comme une borne entre le logiciel et le matériel. Le Tableau 4.10.2 montre les extensions apportées par HybTM au jeu d'instructions pour les deux modes [62].

XBA	Cette instruction commence l'exécution d'une transaction dans mode matériel. Tous les accès mémoire à l'intérieur de la transaction sont par défaut en mode transactionnel.
XBS	Cette instruction commence une transaction en mode logiciel. Tous les accès mémoire à l'intérieur de la transaction sont par défaut en mode non transactionnel.
XC	Pour commiter une transaction.
XA	Pour aborter une transaction.
LDX/STX	Les opérations explicites de chargement et d'enregistrement transactionnels *(transactionnel load et store)* qui sont utilisées à l'intérieur des transactions dans le mode logiciel.
LDR/STR	Les opérations explicites de chargement et d'enregistrement non transactionnels *(non transactionnel load et store)* qui sont utilisées à l'intérieur des transactions dans le mode matériel.
SSTATE	Enregistre l'état des registres dans un point de reprise (*Checkpoint*).
RSTATE	Restaurer l'état de registres à partir d'un point de reprise enregistré précédemment.
XHAND	Spécifier le traitant d'exception si une transaction a aborté a cause d'un conflit d'accès.

Tableau 4.10.2 : Les extensions apportées par HybTM au jeu d'instructions.

La partie logicielle de HybTM est également basée sur le modèle DSTM de Herlihy, discuté en détail dans la section 4.3, elle hérite de son interface de programmation. Cependant, Kumar et autres ont identifié deux actions dans DSTM qui contribuent à la réduction de la performance globale du système à cause du surcoût qu'elles engendrent et ont essayé de les remédier: la *gestion des versions* et *l'allocation et le copiage* requises quand une transaction ouvre un objet pour modification [62]. La partie HTM est utilisée pour enlever ces surcoûts si c'est possible en modifiant les objets sur place par conséquent l'algorithme DSTM doit être changé pour supporter le schéma hybride.

HybTM étend le modèle de DSTM et étend le processeur et ses caches. Sur le coté matériel, HybTM ajoute deux buffers, un buffer transactionnel qui enregistre deux versions pour une ligne de cache (la valeur modifiée transactionnellement et la valeur avant la modification) et Une table d'état transactionnel avec deux champs, un pour indiquer le type de transaction (STM ou HTM) et l'autre pour enregistrer son statut. Le statut permet à une autre transaction (matérielle ou logicielle) d'interrompre une transaction en changent son statut. Cependant, sur le coté logiciel, HybTM modifie le modèle DSTM de Herlihy en apportant des modifications sur la structure du *repère (Locator)*. La modification principale apportée par Kumar et autres est que le *repère* dépiste les lecteurs et les rédacteurs au lieu de maintenir juste les références aux anciens et nouveaux objets de données et leurs états suivant les indications de la Figure 4.10.1. Ces modifications facilitent la détection des conflits de lecture et d'écriture un peu tôt ce qui peut augmenter le taux de commit [62].

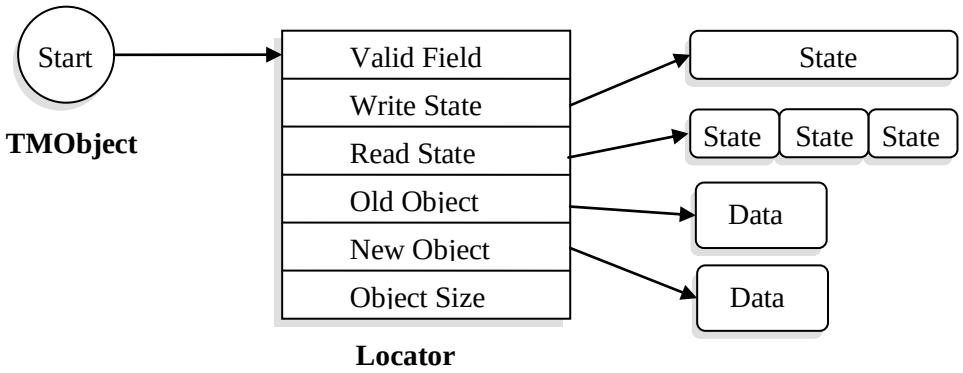

Figure 4.10.1: la structure de TMObject dans HybTM [62].

Dans l'HybMT une transaction qui va écrire, aborte directement tous les lecteurs quand elle ouvre un objet et du fait les transactions n'ont pas besoin de vérifier les versions des objets lors de commit [62]. En revanche, DSTM exige la vérification des objets de données qui étaient ouverts pour une lecture. Afin de supporter ces modifications, le *repère* de l'HybMT inclut un nouveau champ appelé *valid* qui indique si cet objet de données est actuellement ouvert pour une lecture seulement ou une écriture. Contrairement à l'unique variable d'état de DSTM, l'HybMT a des champs séparés pour *l'état de lecture* et *l'état d'écriture*. Dans l'HybMT il peut y avoir plusieurs lecteurs en même temps ou au maximum un seul rédacteur [62]. Le champ *valid* indique laquelle des deux zones d'état est actuellement valide. Le champ *ObjectSize* est utilisé pour sauvegarder la taille de l'objet transactionnel, cette information est utile pour créer des duplications[6] des objets.

Dans l'HybMT un objet peut être ouvert dans les deux modes (STM ou HTM), cependant tous les objets d'une transaction doivent être ouverts dans le même mode choisi initialement [62]. L'ouverture d'un objet dans le mode logiciel exige *l'allocation* et *la gestion de version* comme dans DSTM ce qui engendre plus de surcoût. Cependant, l'ouverture dans le mode matériel est plus simple et ne nécessite pas d'allocation mémoire pour le *repère* et la création des copies logicielles des objets car une transaction dans le mode matériel modifie les données sur place (directement dans le cache).

Lors de l'ouverture d'un objet dans le mode logiciel, la transaction crée son propre statut (*State*) (initialement mis à *ACTIVE*) et charge l'emplacement correspond d'une façon transactionnelle (en utilisant l'instruction *LDX)* dans le *buffer transactionnel*. Dans le mode logiciel, le statut d'un objet est la seule partie de l'objet qu'elle doit être chargée dans le buffer transactionnel et ça pour garantir la suffisance du buffer transactionnel dans le mode logiciel. Quand une autre transaction (matérielle ou logicielle) écrit dans cet emplacement *(State),* le buffer transactionnel détecte un conflit et aborte la transaction. Autrement, l'ouverture d'un objet dans le mode logiciel fonctionne d'une façon similaire à DSTM comme décrit dans la section 4.3.

Avant d'ouvrir un objet dans le mode logiciel (en écriture), on doit d'abord consulter les champs W*riteState* et *ReadState* du *repère* d'objet pour voir s'il y a des conflits avec d'autres transactions. A ce moment là, et comme dans DSTM, le repère d'un objet est alloué, initialisé, et le *TMObject* est atomiquement modifié pour pointer vers ce nouveau repère en utilisant un *CAS.*

[6] Dans DSTM, c'est au programmeur de fournit une méthode *dup* pour ce but.

Cependant, dans l'HybTM la copie de l'objet peut être créée seulement après qu'on s'assure qu'aucune transaction matérielle n'a l'objet ouvert car les transactions matérielles modifient les données sur place.

Dans le mode matériel, l'ouverture d'un objet est beaucoup plus simple. Comme dans le mode logiciel, le mode matériel doit d'abord vérifier et aborter les transactions contradictoires du mode logiciel en lisant les zones d'état du repère pour voir s'il est dans l'état ACTIVE, l'en remplace atomiquement par *ABORTED*. Une opération d'écriture entraîne un abort si l'emplacement mémoire à modifier a été chargé dans la mémoire tampon transactionnelle par d'autre transaction de mode logiciel. Après avoir aborter les transactions contradictoires du mode logiciel, l'ouverture dans le mode matériel peut renvoyer les données courantes valides. Contrairement au mode logiciel, le mode matériel ne réalise aucune allocation mémoire ou copiage et de plus il n'a pas un repère lié à lui. Une transaction en mode logiciel peut aborter une autre transaction de même mode ou d'autre mode et ainsi une transaction en mode matériel peut aborter une autre transaction de même mode ou d'autre mode.

Au début de chaque transaction, le schéma hybride choisit entre le mode d'exécution matériel et logiciel. Le mode matériel apparu le plus approprié du fait qu'il engendre moins de surcoût et garantit une exécution plus rapide. Pour cela dans HybMT une transaction essaie d'abord dans le mode matériel trois fois et si elle ne réussit pas à commiter à trois tentatives alors elle commute vers le mode logiciel et réessaie jusqu'à ce qu'elle réussira [62]. L'HybMT utilise le gestionnaire *Polit* pour la gestion de conflit dans le mode logiciel qui utilise un back-off exponentiel aléatoire. Cependant, dans Le mode matériel il utilise une politique *agressive* pour la gestion de conflit d'où une transaction interrompra automatiquement la transaction contradictoire.

4.10.2 Discussion

Un certain nombre d'améliorations peuvent être suggérées dans l'HyMT de Kumar et autres. Une amélioration peut être la classification des transactions échouâtes. Quelques transactions s'interrompent en raison des conflits avec d'autres transactions tandis que d'autres transactions s'interrompent en raison de la limitation de la ressource matérielle de l'HyMT. Les premières transactions peuvent être relancées dans le mode matériel pour une exécution efficace. Cependant, cette fonctionnalité exige l'implémentation du traitant d'exception dans le mode matériel qui indique quel est le type de transaction qui a échoué [62]. D'ailleurs, les transactions extrêmement longues créent un problème pour l'HyMT. Une transaction qui a un long temps ne pourra jamais se terminer avec succès dans le système actuel. Afin d'accommoder telles transactions un troisième mode est exigé où le matériel actuel ne peut pas être utilisé du tout. D'ailleurs, tous les threads courants peuvent être interrompus et laissent cette transaction pour compléter son exécution. De telles transactions sont très rares, cependant cette approche assurera la terminaison sans affecter la performance globale [81].

4.11 Hybrid Transactional Memory (HyTM)

Mark Moir et autres ont présenté la mémoire transactionnelle hybride (HyTM) en 2006 [12]. HyTM est une approche pour implémenter la mémoire transactionnelle en logiciel tout en utilisant le meilleur effort du matériel pour améliorer la performance mais elle ne dépend pas de la mémoire transactionnelle matérielle (HTM). Le prototype de HyTM inclut un compilateur et une bibliothèque STM. Le compilateur de HyTM permet à une transaction d'essayer d'utiliser le meilleur effort du matériel et de réessayer en utilisant le logiciel en cas d'échec. HyTM est basé

sue le mot et utilise un gestionnaire explicite de conflit pour résoudre les conflits [12]. Les détails de conception de base du HyTM sont affichés dans le Tableau 4.11.1.

Tableau 4.11.1: Les caractéristiques de base de HyTM.

HyTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Mot
Stratégie de Mise à jour	Différée
Stratégie de Détection de Conflit	Tard
Gestionnaire de Conflit	Polka
Nested Transaction	Aplaties

4.11.1 Détails d'implémentation

L'idée derrière le développement de HyTM est de construire un système de mémoire transactionnelle logicielle qui ne dépend pas du support matériel qui n'est pas disponible actuellement, mais c'est de fournir également la capacité d'exécuter des transactions en utilisant n'importe support HTM qui est disponible de telle manière que les deux types de transactions puissent coexister correctement. Cette approche permet de développer des programmes d'essai en utilisant les systèmes d'aujourd'hui, et puis de mieux exploiter successivement le meilleur effort des implémentations HTM pour améliorer la performance dans le futur. Pour réaliser cette fin, La synchronisation entre le STM et les transactions basées sur le HTM, est réalisée en imposant la condition qu'une transaction basée sur HTM ne commite pas si elle détecte un conflit avec une transaction basée sur le STM, ça est réalisé en ajoutant aux transactions matérielles un code pour contrôler les structures de données maintenues pas le STM [12].

Cependant, quand une transaction basée sur le STM a un conflit avec une transaction basée sur le HTM, alors la transaction matérielle est interrompue et relancée dans le STM ou le HTM [12]. Dans HyTM, on le présume que la plupart des transactions sont exécutées dans le HTM car le HTM est plus rapide que le STM. Quand une transaction HTM échoue alors un appel est fait au HyTM pour décider si la transaction peut être relancée dans le HTM ou dans le STM [12]. Cette méthode peut également mettre en application des politiques de gestion de conflit, telles que le back-off avant de ressayer car dans certains cas, il est plus raisonnable de relancer une transaction en matériel après un court délai pour réduire le conflit et pour augmenter les possibilités de commit dans le matériel. Un tel délai peut être déterminer en effectuant une technique de back-off simple ou en utilisant également des politiques de gestion de conflit plus sophistiquées. Cependant, quand une transaction HTM échoue à plusieurs reprises, alors elle est passée au STM où les limitations de matériel sont éliminées et on peut exécuter la transaction dans un environnement de gestion de conflit plus flexible [12].

HyTM utilise deux structures de données principales, le *descripteur de transaction* et l'*enregistrement de propriété (orec).* Le descripteur de transaction contient *l'identificateur de descripteur de transaction* (tdid), l'*en-tête de transaction* (transaction header)*, l'ensemble de lecture* (ReadSet) et *l'ensemble d'écriture* (WriteSet). L'en-tête de transaction est un mot de 64 bits qui contient un *numéro de version* et un *statut* qui peut avoir les valeurs *FREE, ACTIVE, ABORTED* et *COMMITTED.* L'ensemble de lecture contient un *snapshot* pour chaque *orec* qui correspond à un emplacement mémoire lu et n'est pas écrit par une transaction. L'ensemble d'écriture contient une entrée pour chaque emplacement mémoire que la transaction vise de le modifier, elle sauvegarde l'adresse de l'emplacement et la valeur la plus récente écrite à cet emplacement par cette transaction. La Figure 4.11.1 montre ces structures de données. L'enregistrement *OREC*, est un enregistrement des mots de 64 bits contenant les champs : *tdid, ver, mode* et *rdcnt.* Les champs *tdid* et *ver* sont utilisés pour indiquer la transaction la plus récente qui a possédé l'*orec* en mode d'écriture (dans ce mode l'*orec* est acquis d'une manière exclusive). Le champ *mode* indique si l'*orec* est possédé ou non et s'il est possédé dans quel mode, il peut avoir les valeurs : *UNOWNED, READ* et *WRITE*. Si *l'orec* est acquis en mode de lecture (le champ *mode* contient la valeur *READ*), le champ *rdcnt* indique le nombre des transactions qui sont en train de lire les emplacements mémoire mappés vers cet *orec* (dans ce mode l'*orec* est acquis d'une manière non exclusive).

Une transaction exécutée dans le mode logicielle (en utilisant la bibliothèque HyTM) commence en mettant son statut à *ACTIVE* et en initialisant les ensembles de lecture et d'écriture à vide. Elle exécute le code utilisateur en faisant appel à la bibliothèque STM lors de chaque accès mémoire. Avant d'écrire dans un emplacement mémoire, la transaction doit acquérir la propriété de cet emplacement dans le mode d'écriture et créer une entrée dans son ensemble d'écriture pour enregistrer la nouvelle valeur écrite à cet emplacement. Pour cela, la transaction change d'abord le champ *mode* de l'*orec* correspondant à cet emplacement à *WRITE* et enregistre son *identificateur de descripteur* (*tdid*) et son *numéro de version (ver)* dans l'*orec* correspondant. Les écritures ultérieures à cet emplacement (de la même transaction) trouvent l'entrée dans l'ensemble d'écriture, et écrasent la valeur dans cette entrée avec la nouvelle valeur à écrire. De même, avant de lire un emplacement, une transaction acquiert la propriété sur l'*orec* correspondant à cet emplacement, cette fois en mode de *LECTURE*. Si l'*orec* est déjà possédé en mode de *LECTURE* par d'autres transactions, cette transaction peut acquérir la propriété en incrémentant le champ *rdcnt* et en gardant tous les autres champs tels qu'ils sont. Autrement, la transaction acquiert l'*orec* en mode de LECTURE, en plaçant le champ *mode* à *READ* et le champ *rdcnt* à *1*. Dans les deux cas, la transaction enregistre dans son ensemble de lecture l'index de l'*orec* (récupéré à partir de la table *OREC)* et un *snapshot* (le contenu de l'*orec*) à ce moment-là.

Après chaque opération de lecture, une transaction *valide* son ensemble de lecture. La validation est la vérification que toutes les valeurs lues par une transaction n'ont pas été changées depuis la dernière lecture par cette transaction. Quand une transaction termine, elle essaie de commiter, elle valide son ensemble de lecture et si elle réussit, elle change son statut de *ACTIVE* à *COMMITTED* d'une manière atomique, en cas de succès, elle copie les valeurs à partir de l'ensemble d'écriture vers les emplacements mémoire appropriés avant de libérer la propriété de ces emplacements.

HyTM supporte des transactions emboîtées *aplaties* d'où une transaction externe entoure la transaction intérieure. En d'autres termes, une transaction intérieure commite seulement quand la transaction la plus externe commite avec succès. HyTM implémente le gestionnaire de conflit *Polka* qui est une combinaison des deux gestionnaires *Polit* et *Karma*, expliqué dans la section 3.4.10.3. Le gestionnaire *Polka* utilise la rotation du gestionnaire *Polit* et la stratégie de back-off

avec le schéma de priorité du gestionnaire *Karma*. Le schéma de priorité est basé sur la quantité de travail effectuée par une transaction particulière. Le nombre des objets de données ouverts par une transaction est utilisé comme unité pour la quantité des données traitées. Le gestionnaire Polka accorde la priorité aux transactions qui ont traité plus de données et les laisse compléter leur traitement [12].

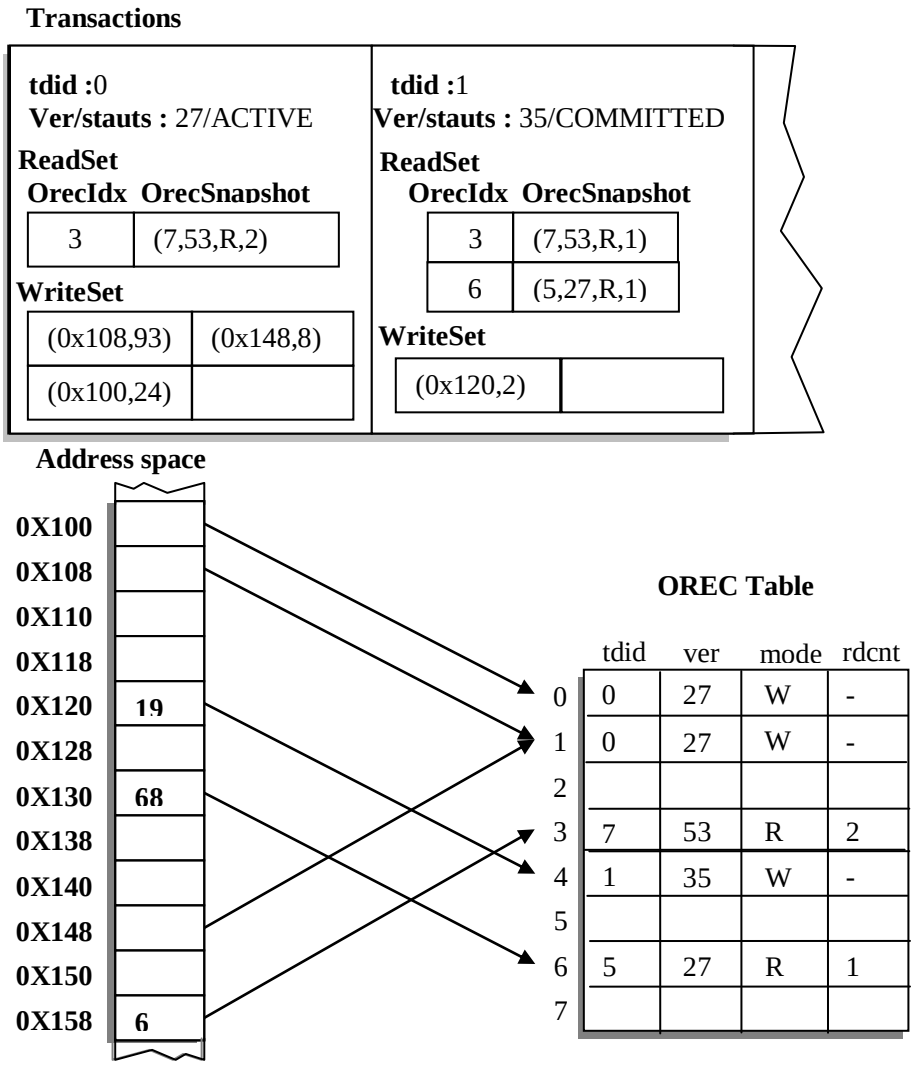

Figure 4.11.1 : Les structures de données utilisées par HyTM [12].

On discute maintenant comment HyTM modifie les transactions matérielles pour assurer une interaction correcte avec les transactions exécutées dans le mode logiciel en utilisant la bibliothèque HyTM. L'observation principale est que la valeur logique d'un emplacement mémoire diffère de son contenu physique seulement si une transaction logicielle modifie actuellement cet emplacement mémoire. Par conséquent, si aucune transaction logicielle n'est en exécution actuellement, alors on est sûr que la valeur logique n'existe pas et donc on peut appliquer directement une transaction matérielle aux emplacements désirés en utilisant le mode HTM. Le défi est d'assurer qu'une transaction matérielle ne s'exécute que seulement si aucune transaction logicielle contradictoire n'est en exécution. Le prototype HyTM modifie des transactions matérielles pour qu'elles détectent les conflits avec les transactions logicielles à la granularité des orecs. Plus précisément, le code d'une transaction matérielle est modifié de sorte que la transaction matérielle consulte l'orec associé à chaque emplacement mémoire consulté

pour détecter les transactions logicielles contradictoires. L'HyMT s'assure que le HTM aborte chaque transaction matérielle si cet orec se change avant que cette transaction matérielle commite.

4.11.2 Discussion

Le but principal du HyTM était d'augmenter l'évolutivité du STM avec le support HTM. Cependant, idéalement HyTM devrait donner une bonne performance sans support HTM dans le cas des conflits réduits. Quoique HyTM souffre de coût de validation où chaque transaction exige la vérification de l'*OREC* afin de découvrir les conflits possibles, néanmoins si les transactions logicielles sont fréquentes ou en d'autres termes s'il n'y a aucun support HTM alors les telles lectures de l'*OREC* ralentiront l'évolutivité [12]. Une approche différente pour les transactions HTM peut résoudre ce problème. Une manière peut être que la transaction HTM utilise des lectures invisibles ou en d'autres termes, gardent un *snapshot* de l'*OREC* pour la validation postérieure. Cependant, la conservation de l'*OREC* séparément entraînera un autre surcoût, donc la solution se base en faisant le bon choix.

4.12 Multi-Core RunTime STM (McRT-STM)

Saha et autres [76] ont développé un STM bloquant basé sur les verrous nommé *McRT (Multi-Core RunTime),* dans les laboratoires de recherches d'Intel en 2006. Le McRT est un compilateur (*JIT*) -*Just-in-Time* développé pour les transactions C/C++ et Java. Ce compilateur est un système STM complet basé sur les optimisations de compilateur et de système d'exécution (par exemple le scheduleur et le gestionnaire de la mémoire) pour les constructions de mémoire transactionnelles. McRT-STM supporte une granularité au niveau linge de cache ou objet avec un mécanisme de mise à jour direct. D'ailleurs, il a une stratégie de détection de conflit optimiste pour les lectures et une stratégie pessimiste pour les écritures. Le McRT utilise l'abort comme stratégie de résolution du conflit et supporte des transactions emboîtées fermées. Les détails de conception de base sont également affichés dans le Tableau 4.12.1.

Tableau 4.12.1 : Les caractéristiques de base de McRT-STM.

McRT-STM	
Synchronisation	Bloquante
Contrôle de Concurrence	Lecture Optimiste et écriture Pessimiste
Granularité	Objet ou ligne de cache
Stratégie de Mise à Jour	Directe
Stratégie de Détection de Conflit	Tôt et Tard
gestionnaire de Conflit	Abort
Transactions Emboitées	Fermées

4.12.1 Détails d'implémentation

Nous discuterons ici seulement les différents choix conceptuels de McRT-STM, car on va discuter en plus de détails un système similaire qui est BSTM dans la section 4.13.

McRT est basé sur les verrous et il utilise le protocole de verrouillage à deux phases pour contrôler l'accès à un objet partagé, dans la première phase une transaction acquiert tous les verrous nécessaires et dans la deuxième phase, elle libre ces verrous acquis après avoir commiter. Ce protocole permet à plusieurs transactions de lire un objet simultanément, mais il donne un accès exclusif à une seule transaction pour modifier l'objet. McRT-STM utilise deux types de verrou, un verrou de lecture et un autre d'écriture.

Une transaction maintient un ensemble de lecture contenant les numéros de version des objets lus après avoir acquérir le verrou de lecture correspondant, ces numéros de version sont utilisés ultérieurement dans la validation. Lors d'une écriture à un emplacement mémoire, la transaction doit acquérir le verrou d'écriture correspondant et enregistrer le numéro de version et la valeur de cet emplacement dans un *log de défaire* (undo-log) pour les restituer si la transaction aborte. Lors de la phase de commit, la transaction doit valider son ensemble de lecture pour détecter les éventuels conflits d'écritures avec d'autres transactions en assurant que les numéros de version des emplacements lus demeurent inchangés.

Chaque objet possède un verrou, ce verrou peut avoir deux états. Si l'objet est libre ou il est verrouillé pour une lecture, il contient le numéro de la version en cours de l'objet. Dans cet état, une autre transaction peut ouvrir l'objet pour une lecture en enregistrant tout simplement l'objet et sa version en cours dans son ensemble de lecture. Cependant, l'autre état indique que l'objet est ouvert pour une écriture et dans ce cas-ci, le verrou contient un pointeur vers le descripteur de la transaction modifiant l'objet.

Lors de l'acquisition d'un verrou par une transaction T pour une écriture. T enregistre le numéro de version de l'objet correspondant (contenu dans le verrou) dans son log et remplace ce numéro de version par un pointeur vers son descripteur. Quand la transaction T commite elle libère ce verrou, et remplace son contenu (actuellement un pointeur vers le descripteur de T) par la valeur de la nouvelle version de l'objet calculée en incrémentant l'ancienne version de l'objet (sauvegardée précédemment).

Dans la plupart des cas, une transaction T accède à un objet *o* pour une lecture et par la suite effectue une écriture. McRT permet à une transaction de changer le statut du verrou associé à *o* (d'un verrou de lecture à un verrou d'écriture) et au moment de commit, T doit vérifier que le numéro de version sauvegardé lors de lecture de *o* est égal à la valeur initiale de numéro de version d'écriture. Les lecteurs actifs n'empêchent pas un rédacteur d'acquérir un verrou et de modifier un objet (pas de coalition des lecteurs contre le rédacteur). Le conflit est détecté et résolu quand une transaction de lecteur essaie de commiter. De même quand une transaction ouvre un objet pour une écriture et modifie sa valeur, elle n'interrompt pas les transactions qui lisent précédemment le même objet. Cependant, chacune de ces transactions détectera le conflit pendant la phase de commit quand elle contrôle les numéros de version des objets lus.

McRT-STM utilise une stratégie de synchronisation bloquante qui rapproche au comportement des autres systèmes STMs qui adaptent la *liberté d'obstruction* comme mécanisme de synchronisation, en s'assurant qu'une transaction suspendue n'empêche pas des transactions actives de progresser vers l'avant. Cette stratégie consiste à : Quand un objet est ouvert pour une écriture, les autres transactions qui essaient d'ouvrir cet objet soit pour lecture ou écriture se bloquent. Après l'écoulement d'une minuterie *(time out)* McRT-STM suspend les

transactions contradictoires si le thread détenant le verrou d'écriture est encore en activité cependant il interrompt la transaction détenant le verrou d'écriture si son thread d'exécution est suspendu.

McRT-STM utilise une stratégie de mise à jour directe, qui consiste à modifier les objets sur place, cependant dans ce cas, les transactions peuvent lire des valeurs modifiées par une autre transaction et ainsi observer des états incohérents. Ceci peut entraîner à des exécutions anormales, (par exemple des exceptions inattendues ou des boucles infinies). Une exception inattendue peut être traitée en attrapant ces exceptions, en interrompant la transaction et en la relançant ultérieurement. Cependant, le problème des boucles infinies exige la consistance de l'ensemble de lecture d'une transaction qui peut être atteint en validant cet ensemble après chaque boucle. Malgré ces problèmes qui peuvent être causés par la stratégie de mise à jour directe, Saha et autres ont constaté que cette technique performe mieux (par un facteur de 2 à 6) que la stratégie de mise à jour différée en raison du surcoût introduit par la recherche des valeurs les plus récentes [76].

4.12.2 Discussion

Contrairement au verrouillage transactionnel, TL [15] et TL2 [16], le STM McRT verrouille un objet dès le premier accès jusqu'au moment de commit et utilise le mécanisme de mise à jour directe. Cependant, il est contre la notion de base du modèle de Dice et autres de la mémoire transactionnelle [79], où des transactions multiples pourraient accéder à une mémoire partagée en même temps. Une amélioration de la performance de McRT-STM peut être l'utilisation d'un verrouillage au moment de commit au lieu de la stratégie actuelle de verrouillage des objets de données dès le premier accès et ça en permettant plus de concurrence.

4.13 Microsoft's Bartok STM (BSTM)

BSTM est un système développé chez Microsoft par Tim Harris et autres en 2006 [38], il est très similaire à McRT-STM d'Intel dans beaucoup aspects. C'est un STM qui utilise une mise à jour directe en maintient un log de défaire pour d'éventuels conflits. BSTM utilise le protocole de verrouillage à deux phases pour assurer un accès exclusif aux objets à modifier. Il sauvegarde les versions des objets lus pour pouvoir détecter les conflits de modifications. BSTM utilise des optimisations de compilateur (*Bartok*) pour éliminer les opérations transactionnelles non nécessaires (par exemple en éliminant les mises à jour des objets transactionnels locales à une transaction) et essaie également de réduire agressivement la taille des logs, en filtrant les entrées inutiles et à travers une forte intégration avec le ramasse miettes *(Garbage Collector).* Les caractéristiques conceptuelles de base de BSTM sont affichées dans le Tableau 4.13.1 suivant.

4.13.1 Détails d'implémentation

BSTM est un système qui utilise deux stratégie de verrouillage ; il utilise une stratégie pessimiste lors d'une mise à jour (c.à.d. verrouille les objets dès le premier accès si l'accès est une écriture) ce choix est motivée par une supposition que les conflits sont rares. Cette technique permet au thread possédant le verrou de mettre à jour l'objet sur place. Cependant, Il utilise une stratégie de la gestion des conflits d'accès optimiste si les accès sont des lectures et ça pour offrir une grande performance [38].

Tableau 4.13.1: caractéristiques conceptuelles de base de BSTM

BSTM	
Synchronisation	Bloquante
Contrôle de Concurrence	Lectures Optimistes et écritures Pessimistes
Granularité	Object
Stratégie de Mise à jour	Directe
Détection de Conflit	Tôt conflit écriture– écriture Tard conflit écriture–lecture
Stratégie de résolution de conflit	Abort
Transactions emboitées	Fermées

Pour supporter la validation des objets ouverts pour une lecture seule et les opérations d'ouvertes et de fermeture sur les objets qui sont mis à jour. BSTM utilise des structures de données qui possèdent deux champs supplémentaires pour chaque objet : le champ *STMWord* et le flag *Snapshot.* Le champ *STMWord* est utilisé pour indiquer le type de l'objet (Figure 4.13.1). Ce champ contient deux valeurs, la première est un bit indiquant si l'objet est ouvert par une transaction pour une modification. Si ce bit est utilisé, l'autre valeur du champ contient un pointeur vers le descripteur de la transaction qui a ouvert l'objet pour modification. Si ce bit n'est pas utilisé (l'objet n'est pas ouvert pour une mise à jour), l'autre valeur contient le numéro de version de l'objet. Les opérations atomiques suivantes manipulent ce champ:

- ***word GetSTMWord****(Object o).*
- ***bool*** *OpenSTMWord (Object o, word prev, word next)*: essaie un CAS atomique sur le *STMword* de *prev* vers *next*
- ***void CloseSTMWord (Object*** *o,* ***word*** *next)*: met à jour le mot à une valeur spécifique.

BSTM utilise également le flag *Snapshot* pour indiquer si l'objet a été mis à jour. Ce flag change sa valeur quand un thread qui a ouvert l'objet pour modification appelle *CloseSTMWord*. Il est utilisé pour détecter les conflits. BSTM enregistre le *STMWord* dans un mot alloué dynamiquement.

Une transaction est représentée par une structure de donnée appelée *TMMgr* qui est manipulée par les opérations usuelles suivantes:

- *TMMgr* ***TMGetTMMgr*** *()*
- *void* ***TMStart*** *(TMMgr tx)*
- *void* ***TMAbort*** *(TMMgr tx)*
- *bool* ***TMCommit*** *(TMMgr tx)*
- *bool* ***TMIsValid*** *(TMMgr tx)*

Une transaction (*TMMgr)* maintient un statut et trois logs sous forme de listes chaînées qui sont écrits séquentiellement et ne seront jamais recherchés [38] (Figure 4.13.1), un log de lecture *(ReadLog)* qui maintient les objets lus par la transaction, un *log d'écriture* (WriteLog) qui maintient les objets modifiés par la transaction (ouverts pour modification) et un *log de défaire* (undo log) qui maintient les modifications qu'elles doivent être restaurées en cas d'abort de la transaction.

Lors de l'ouverture d'un objet par une transaction pour une lecture (en appelant *TMOpenForRead*), elle ajoute l'objet et sa version dans son log de lecture sans besoin de vérifier si l'objet est ouvert pour une modification pour détecter un conflit, cette vérification est faite lors de la phase de validation de l'ensemble de lecture pendant le commit de la transaction.

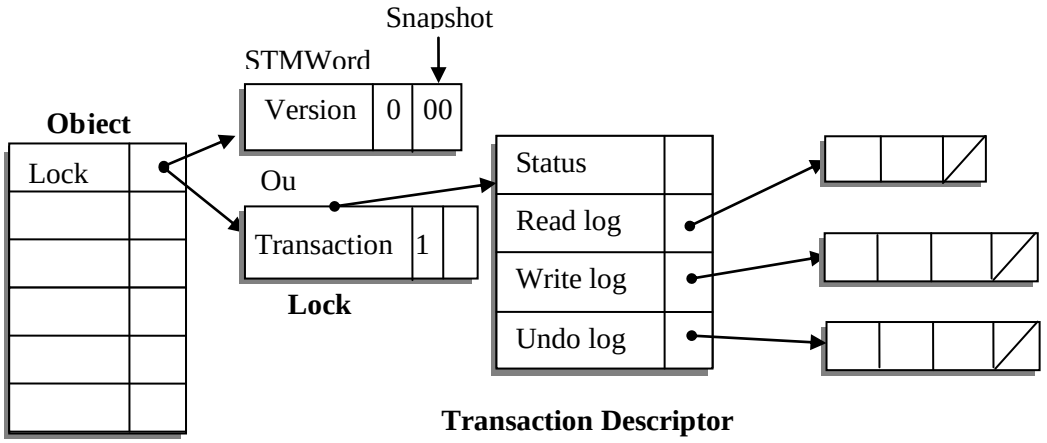

Figure 4.13.1 : Structure de données de BSTM [38].

Lors de l'ouverture d'un objet *o* par une transaction *T* pour une modification (en appelant *TMOpenForUpdate*), *T* doit acquérir un accès exclusif à *o* pour empêcher d'autres transactions de modifier l'objet. D'abord *T* doit s'assurer que l'objet n'est pas acquis par une autre transaction pour modification ni il est déjà modifié par une transaction qui n'a pas encore commité. Après avoir acquérir l'objet, une entrée pour *o* est ajoutée au log d'écriture de *T* et sa valeur sera sauvegardée dans le *log de défaire* pour pouvoir roller en arrière si elle ne réussira par à commiter.

Pour valider une transaction *T*, le thread exécutant fait appel à *STMCommit* qui commite la transaction en deux phases ; Il vérifie d'abord la consistance des objets lus et par la suite ferme les objets ouverts pour modification c.à.d. incrémenter le numéro de version de l'objet et libérer le verrou exclusif acquis. Le commit peut échouer uniquement si la première phase échoue puisque la deuxième phase est garantie de réussir car la transaction possède des verrous exclusifs sur les objets ouverts pour modification. La validation a pour but d'assurer qu'au moment où *T* a lu l'objet aucune autre transaction n'a eu l'objet ouvert pour modification, d'assurer qu'aucune autre transaction n'a ouvert l'objet pour modification après que *T* a l'ouvert pour lecture et qu'aucune autre transaction n'a l'objet ouvert pour une modification. Si la phase de commit s'échoue, la transaction doit s'aborter en exécutant *STMAbort* qui restaure les emplacements modifiés (à partir de log de défaire) et libère les verrous exclusifs acquis par la transaction. Cependant, les objets ouverts pour lecture ne demandent aucun traitement.

4.13.1.1 Optimisations de compilateur

Le compilateur *Bartok* réalise un certain nombre d'optimisations pour éliminer les appels au système STM inutiles, ces optimisations peuvent être classifiées dans les catégories suivantes :

- Les optimisations conventionnelles faites par les compilateurs peuvent optimiser les opérations réalisées par BSTM dont leurs sémantiques sont connues par l'optimiseur. Par exemple, *TMGetTMMgr* renvoie un résultat constant pendant l'exécution de toute la transaction [38], et donc elle peut être placée à l'extérieure d'une boucle (par le *loop-invariant code motion)*. De même on peut éliminer les appels doubles à *TMOpenForRead* et *TMOpenForUpdate* puisque se sont idempotent à l'intérieur d'une transaction [38].

- les objets locaux à une transaction (alloués dans une transaction) n'ont pas besoin d'être enregistrés (dans le log), car ils sont privés à cette transaction et ne peuvent pas être référenciés par d'autre transaction, ils sont écrasés et récupérés par le *GC* si la transaction s'aborte. Le compilateur Bartok utilise une analyse simple de flux de données pour déterminer quelles sont les variables qui contiennent toujours des objets alloués récemment pour les empêcher d'être enregistrés dans le log.

- Les opérations d'ouverture redondantes peuvent être éliminées (par exemple, une méthode ouvrant un objet déjà ouvert par son appelant) en déplaçant les appels à *TMOpenForRead* ou à *TMOpenForUpdate* de demandeur vers son appelant pour éliminer l'appel redondant ici.

- Les objets qui sont lus et par la suite modifiés, par exemple dans le cas d'une incrémentation d'une variable *O.x=O.x+1* quelle implique deux opérations d'ouverture. L'appel à *TMOpenForRead* peut être évité en ouvrant initialement l'objet pour la mise à jour car l'appel à *TMOpenForUpdate* englobe l'ouverture de l'objet.

- Le contrôle de débordement du buffer dans les opérations d'enregistrement (*logging*) dans une transaction peut être parfois optimisé en effectuant un test unique, qui vérifie s'il y a suffisamment d'espace pour toutes les données bufférisées dans la transaction.

Ces optimisations peuvent réduire le nombre d'entrées du log de 40-60% sur une variété de benchmarks, avec une amélioration similaire dans de temps d'exécution [38].

4.13.1.2 Optimisations de Run-time

Malgré les optimisations faites par le compilateur, les logs restent contiennent beaucoup d'entrées en doubles et inutiles. Pour les éliminer BSTM réalise des tests pendant l'exécution. Les surcoûts supplémentaires engendrés par ces tests doivent être compensés par la réduction dans l'utilisation de la mémoire et le coût de commit ou d'abort d'une transaction. Ces optimisations peuvent être classifiées dans les catégories suivantes :

- Les objets alloués à l'intérieure d'une transaction n'ont pas besoin d'être enregistrés dans le log de défaire, puisque ces objets deviennent inaccessibles si la transaction s'interrompt. L'analyse statique de compilateur ne peut pas identifier tous ces objets [38]. BSTM garde la trace des objets alloués à l'intérieure d'une transaction et ne les enregistre

pas dans le log de défaire. Cependant, ces objets exigent la lecture et la mise à jour des entrées dans le log, pour qu'ils puissent être commis.

- BSTM utilise un filtre pour détecter et éliminer les entrées en doubles dans les logs de lectures et de défaire.
- Le ramasse miettes de Bartok compacte les logs lors du ramassement. Ce compactage supprime les entrées des objets inaccessibles dans le log (des ordures rassemblées), il supprime aussi les entrées dans le log de lecture pour les objets qui sont ultérieurement ouverts pour une modification, et les entrées en doubles dans le log.

Bien que ces optimisations d'exécution puissent augmenter le temps d'exécution de quelques programmes, elles peuvent également réduire la période d'exécution d'autres applications d'une manière significative.

4.13.2 Discussion

BSTM a adopté quatre approches pour accélérer la mémoire transactionnelle basée sur le mot [38]; il modifie les données sur place pour éviter la recherche des valeurs récentes dans les logs, il introduit une décomposition et optimisation au moment de la compilation pour réduire l'utilisation des opérations de mise en log, il utilise un filtrage rapide pendant l'exécution pour supprimer les duplications dans les logs, et un compactage des logs fait par le GC pendant l'exécution pour retirer de manière déterministe les objets morts et tous les doubles qui ont été manqués par l'analyse statique du compilateur .

4.14 Non-bloquante Zero-Indirection Transactional Memory (NZTM)

F. Tabba, C. Wang et M. Moir ont présenté NZTM, un système de mémoire transactionnelle non bloquant sans adressage indirecte c.à.d. avec zéro niveau d'indirection d'où il intervient le nom *(Non-bloquante Zero-Indirection transactional memory)* en 2007[82]/2009[83]. NZTM est une mémoire transactionnelle logicielle hybride basée sur l'objet qui peut utiliser le meilleur effort de la mémoire transactionnelle matérielle si elle est disponible et elle peut également exécuter les transactions en utilisant la mémoire transactionnelle logicielle compatible (NZSTM) en cas d'échec. NZTM utilise une stratégie de mise à jour directe où il modifie les objets sur place et Il utilise un gestionnaire explicite de conflit pour résoudre les conflits. Kumar et autres ont également présenté une mémoire transactionnelle hybride semblable (section 4.10). D'ailleurs, NZTM a beaucoup de similarités avec DSTM de Herlihy et autres [44]. Les détails de conception de base de NZTM sont affichés dans le Tableau 4.14.1.

4.14.1 Détails d'implémentation

NZTM a une implémentation non bloquante et il modifie les objets de données sur place dans le cas générale, cependant la difficulté principale dans la conception d'un tel STM est *l'incertitude* qui surgisse quand une transaction *T1* met à jour un objet et une autre transaction *T2* différente souhaite accéder à cet objet. *T2* ne peut pas attendre *T1* pour se terminer parce que ceci aurait comme conséquence une implémentation bloquante. *T2* peut essayer d'informer *T1* qu'elle devrait arrêter la modification de l'objet, mais jusqu'à ce que *T2* puisse déterminer que *T1* s'est rendue compte qu'elle devrait s'arrêter, il n'est pas sûr pour *T2* ou d'autres transactions de mettre à jour l'objet de donnée sur place, parce que *T1* peut continuer l'écrasement de l'objet. Pour remédier à cette difficulté, NZTM fait recourt à l'adressage indirect d'un seul niveau

malgré le surcoût engendré pendant que *T1* ne répond pas pour éviter le blocage comme dans les systèmes STMs bloquants [82].

Tableau 4.14.1: Les caractéristiques de base de NZTM.

NZTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Object
Stratégie de Mise à Jour	Directe ou différée
Stratégie de Détection de Conflit	Tard
gestionnaire de Conflit	Spécifique
Transactions Emboitées	Non Supportées

Dans ce cas et contrairement aux autres systèmes STM non bloquants précédents où une transaction contradictoire est interrompue, la transaction dans NZTM arrête elle-même et prend peu de temps pour terminer le retour-arrière et par la suite l'autre transaction peut modifier les objets de données sur place et sans aucune incertitude [82]. Par conséquent, NZTM élimine le gap de performance introduit par les STMs précédents (bloquant et non bloquant) en éliminant leurs niveaux d'adressage indirect.

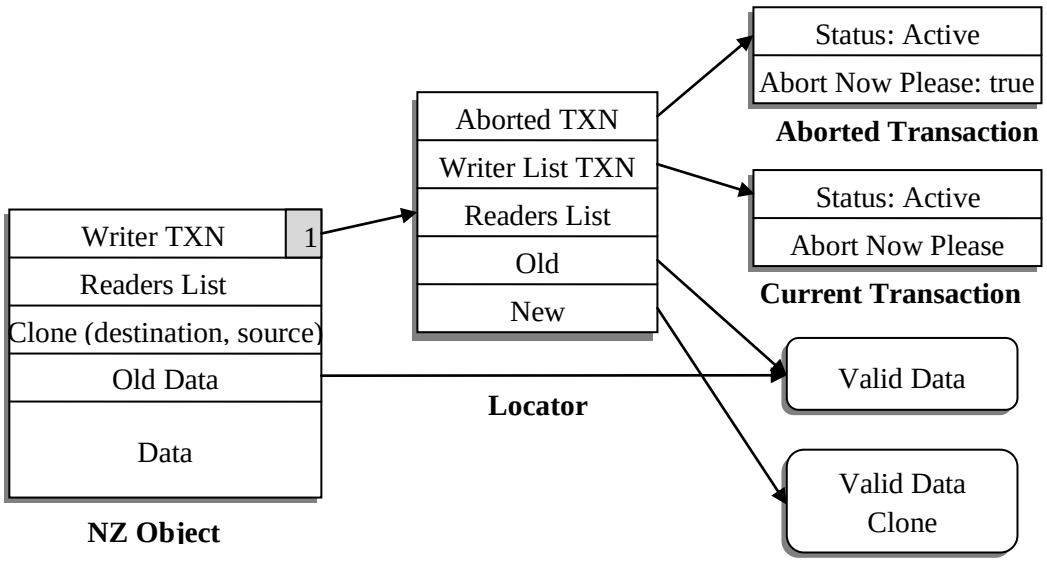

Figure 4.14.1 : Les structures de données de NZTM [82].

La structure de données et le modèle de programmation employés par NZTM est très semblable à ceux de DSTM [44], cependant l'objet *Transaction* dans NZTM contient un nouveau champ appelé *AbortNowPlease* qui est un flag utilisé pour demander à une transaction

de s'aborter, ce flag est enregistré avec le champ *Status* et il peut avoir les valeurs *True* et *False*. Ces deux champs sont modifiés d'une manière atomique [82].

L'objet *NZObject* possède une structure similaire à celle de *TMObject* de DSTM, il possède les champs : *WriterTXN* qu'il pointe (s'il n'est pas vide) vers la dernière transaction qui a ouvert cet objet pour écriture. Le champ *ReaderList* qui est un pointeur vers une liste chainée des lecteurs de l'objet. Le champ *Clone* qui pointe vers une fonction qu'elle peut être utilisée pour créer une copie de l'objet. Le champ *OldData* qui pointe vers une copie de backup de l'objet pendant que la transaction qui modifie l'objet est en progrès. Finalement le champ *Data* contenant la valeur actuelle de l'objet. NZTM n'a pas besoin d'aucun niveau d'adressage indirect pour accéder aux objets car ils sont stockés dans un offset fixe à partir de début de l'objet [82]. Après l'initialisation d'un *NZObject* tous les champs sont nul sauf les champs *Data* et *Clone* qui contiennent la valeur de l'objet [82].

Comme dans DSTM, un thread commence une transaction en créant un nouvel objet *Transaction* en mettant son statut à *ACTIVE*. Par la suite, La transaction ouvre des différents objets de données soit pour une lecture ou une écriture et quand elle termine son exécution, elle essaie de changer leur statut d'*ACTIVE* à *COMMITTED*. Quand une autre transaction détecte un conflit avec cette transaction, elle consulte le gestionnaire de conflit pour décider si elle doit l'attendre où d'essayer de l'interrompre [82]. Contrairement à DSTM et à d'autres systèmes STM, une transaction *T* dans NZTM n'interrompt pas la transaction contradictoire mais elle demande à elle de s'interrompre. Cette demande est lancée en plaçant l'indicateur *AbortNowPlease* de cette transaction à *True*. Quand une transaction note que le flag *AbortNowPlease* est mis à *True*, elle met à jour son statut à *ABORTED* [82]. La transaction *T* attend ceci avant de procéder en avant pour ouvrir l'objet sur lequel le conflit s'est produit. En raison de cette attente, le STM est bloquant. Cependant, nous discuterons la technique utilisée par NZTM pour rendre ce mécanisme non bloquant dans la section suivante.

Une transaction *T* qui veut modifier un objet doit d'abord acquérir un accès exclusif sur cet objet en plaçant un pointeur vers *T* dans le champ *WriterTXN* de l'objet, et avant de modifier la donnée, la transaction doit créer une copie de l'objet (en utilisant la fonction *Clone*) et rendre le champ *OldData* de l'objet pointé vers cette copie. Cette copie sera utile pour restaurer la valeur de l'objet dans le cas où la transaction aborte.

NZTM utilise une liste visible des lecteurs dans laquelle une transaction ouvrant un objet pour une lecture insère un pointeur vers elle-même dans la liste *ReaderList* de l'objet. De cette façon, les transactions qui ouvrent l'objet pour une écriture peuvent détecter des conflits avec les transactions qui lisent l'objet, et peuvent décider de les interrompre si nécessaire pour résoudre le conflit.

4.14.1.1 NZSTM la version non bloquante de NZTM

Nous décrivons maintenant comment transformer le STM décri précédemment d'un système bloquant vers un système non-bloquant. Pour cela, il faut éliminer l'attente engendrée quand un conflit entre deux transactions se produit, et comme les STMs non-bloquant précédents basés sur l'objet [23,36,47] qui impliquent au moins un niveau d'adressage indirect dans tous les cas, NZTM fait recourt à un adressage indirect d'un seul niveau en déplaçant temporairement les données logiques vers un emplacement différent que le champ *Data* du *NZObject*. Pour cela NZTM "étend'' un objet et utilise des techniques comme celles utilisées par les STMs basées sur l'objet quand un propriétaire d'un objet est interrompu et ne répond pas [82].

Pour le moment la conception de NZSTM quand un objet doit être étendu est similaire à celle de l'approche DSTM (on l'appelle *DSTM-Like*). Puisque l'objet de début (*Start*) de DSTM est juste un pointeur, il peut être intégré dans NZSTM en utilisant le champ *WriterTXN* comme pointeur de début, et en indiquant qu'il devrait être traité comme un pointeur en utilisant le bit de poids faible du pointeur.

Quand une transaction *T* a invité le propriétaire actuel d'un objet à s'interrompre, et le propriétaire n'a pas répondu. *T* peut décider de ne pas attendre plus de temps et d'étendre cet objet à un objet DSTM-Like. Pour cela, *T* crée un *repère* DSTM en rendant le champ *Transaction* de ce repère pointé vers l'objet *Transaction* de *T*, et le champ *Old* du repère pointé vers la copie de backup créée par la transaction qui ne repend pas ou vers le champ *Data* s'il n'y a pas de copie de backup, et finalement le champ *New* du repère pointé vers une copie privée de la copie de backup créée en utilisant la fonction *Clone*. L'objet du repère de NZSTM a un champ supplémentaire appelé *AbortedTXN*, qui pointe vers l'objet *Transaction* de la transaction non responsive.

Par la suite, *T* essaie de permuter *WriterTXN* de l'objet *Transaction* de la transaction insensible au nouveau repère créé, en plaçant le bit de poids faible à 1 pour indiquer que l'objet pointe maintenant a un repère de DSTM-like au lieu de l'objet *Transaction* de NZSTM. Ceci a comme conséquence un état comme celui illustré sur la Figure 4.14.1. Dorénavant, l'objet est traité comme un objet DSTM (en plus que chaque nouveau repère à introduire contient le champ *AbortedTXN* à partir du repère remplacé pour préserver l'identité de la transaction insensible).

Une fois la transaction insensible s'aborte finalement, on est sûr qu'elle ne modifie plus le champ *Data* de l'objet, donc il est désirable de restaurer l'objet pour qu'il soit un *NZObject* normal de sorte que les transactions ultérieures bénéficient de la performance d'exécution en accédant à l'objet sur place. Ceci peut être réalisé par une transaction *T* comme suit ; *T* ouvre l'objet pour une écriture selon la méthode DSTM normale, sauf qu'elle enregistre un pointeur vers sa propre *Transaction* dans le champ *AbortedTXN* du nouveau repère qu'elle installe. Ceci établit *T* comme la transaction qu'elle doit être interrompue et répondue avant qu'une tentative ultérieure pour restaurer l'objet puisse commencer, lorsque *T* ne termine pas la restauration.

Par la suite *T* enregistre dans *OldData* un pointeur vers la valeur courante de l'objet (indiquée par le pointeur *Old* dans le repère nouvellement installé), de sorte qu'il se serve comme une copie de backup lorsque *T* s'interrompt. Ensuite *T* essaie de substituer le repère qu'elle a juste installé dans *WriterTXN* avec un pointeur vers sa propre *Transaction* (en plaçant le bit de poids faible à 0 pour indiquer que l'objet est maintenant revient dans le mode normal). Si ceci réussit, *T* copie tout simplement la valeur actuelle de l'objet dans le champ *Data* et procède normalement.

Avant que *T* permute sa *Transaction* dans *WriterTXN*, d'autres transactions peuvent souhaiter accéder à l'objet, dans ce cas elles peuvent procéder comme d'habitude, en ouvrant l'objet selon la méthode DSTM modifiée décrite ci-dessus. Si une transaction a réussi à faire avant que *T* permute sa *Transaction* dans *WriterTXN*, alors le CAS du *T* sur *WriterTXN* échouera, ainsi *T* saura qu'elle n'a pas réussi à restaurer l'objet. Dans ce cas, elle s'interrompt afin de permettre à une tentative ultérieure de restauration de réussir (parce qu'elle est maintenant la transaction identifiée par *AbortedTXN*).

Un point subtil avec ce mécanisme est qu'une transaction lente qui a commencé à restaurer un objet peut enregistrer une valeur démodée dans *OldData*. Cependant, ce n'est pas un problème, parce que les tentatives ultérieures de restauration de l'objet ne commenceront pas

qu'après que cette transaction reconnaisse qu'elle est interrompue. Jusque là *OldData* n'a pas de valeur parce que l'objet reste étendu.

Comme NZTM est une mémoire transactionnelle hybride, donc le système utilise la mémoire transactionnelle logicielle non bloquante (NZSTM) pour des transactions logicielles. Les transactions essaient à plusieurs reprises en utilisant le matériel NZTM et si elles ne réussissent pas alors elles ressaieront en utilisant le NZSTM [82].

Faud et autres dans [83] ont réalisé plusieurs comparaisons de performance en testant NZTM avec plusieurs autres systèmes de mémoire transactionnels, par exemple, LogTM-SE, BZTM, DSTM2-SF, SCSS. Le test de performance a indiqué que l'édition de LogTM-Signature a surpassé NZTM. NZTM a un mécanisme matériel supplémentaire pour détecter les conflits avec les transactions logicielles alors que la mémoire transactionnelle basée sur le matériel pur telle que LogTM n'exige pas un tel mécanisme.

4.14.2 Discussion

NZSTM est un STM non bloquant qui évite les surcoûts des niveaux d'adressage indirect. NZSTM peut être utilisé avec les systèmes actuels sans aucun support matériel supplémentaire. Le thème de NZTM a été inspiré par le travail de Marathe et Moir mais leur HTM est basé sur le mot, en revanche, NZTM est basé sur l'objet. Cependant, NZTM utilise des méta-données complexes pour réaliser la liberté d'obstruction [81].

4.15 Phased Transactional Memory (PhTM)

La mémoire transactionnelle en phases PhTM a été présentée par M. Moir, Y. Lev et D. Nussbaum en 2007 [60]. PhTM commute entre différentes phases chacune est implémentée par une forme différente de mémoire transactionnelle. En d'autres termes, PhTM utilise des implémentations différentes de mémoire transactionnelles, dans des phases différentes ce qui permet de s'adapter suivant les différentes circonstances et charges de travail. PhTM est une mémoire transactionnelle hybride qu'elle a deux modes d'exécution matériel et logiciel. Dans certain cas elle utilise la mémoire transactionnelle matérielle et dans d'autres cas elle utilise la mémoire transactionnelle logicielle. La mémoire transactionnelle matérielle illimitée surpasse d'une manière significative la mémoire transactionnelle hybride parce que le HTM ne prend pas soin des conflits avec les transactions logicielles concurrentes et du fait elle ne souffre pas de surcoût de telle détection des conflits. De plus les transactions exécutées dans le mode logiciel exposent assez d'informations pour permettre aux transactions exécutées en utilisant le HTM de détecter des conflits ce qui engendre un surcoût et par conséquent il est difficile à une implémentation Hybride de mémoire transactionnelle de compéter d'autres STM qui n'ont pas cette condition (Par exemple, le prototype de HyTM de Mark Moir et autres (section 4.11) utilisé uniquement dans le mode logiciel est surpassé par les STMs existants par exemple le TL2) [60]. Le but de PhTM est de construire un système de mémoire transactionnelle hybride dans lequel les transactions qui s'exécutent avec succès en utilisant le meilleur effort de HTM réalisent une performance (presque) comme celle d'une HTM illimitée, alors que les transactions exécutées dans le mode logiciel sont compétitives avec les systèmes STMs les plus rapides [60].

4.15.1 Détails d'implémentation

PhTM supporte des différents modes pour l'exécution des transactions, chaque mode est optimal pour certaines situations en se basant sur la charge de travail et la disponibilité du support matériel. Dans PhTM le système tourne dans un mode pour certaine période de temps et

après permute vers un autre mode c.à.d. l'exécution se déroule dans des phases. Les défis principaux relevés en développant PhTM incluent l'identification du mode le plus approprié et le plus efficace pour une situation particulière, la gestion des transitions d'un mode vers un autre mode sans perdre des données ou des transactions et de décider quand il faut commuter vers l'autre mode [60]. Le prototype de PhTM est intégré avec le compilateur C/C++.

L'idée principale de PhTM est d'utiliser des implémentations de mémoires transactionnelles indépendantes et de réaliser la transition entre ces implémentations d'une manière simple et souple pour utiliser le mode le plus optimal pour la situation actuelle en se basant sur la charge de travail et l'environnement d'exécution actuels. Donc le changement vers un mode est fait en cas de changement dans la charge de travail ou si le mode actuel ne supporte pas une transaction [60]. En se basant sur ces paramètres, on peut identifier les différents modes d'exécution suivants pour des situations différentes [60] :

- Le support HTM est disponible et pertinent pour la charge de travail actuelle. Dans cette situation on exécute toutes les transactions dans le HTM (*HARDWARE Mode*)

- Le support HTM n'est pas disponible ou il ne peut pas être efficace pour la charge de travail actuelle. Dans ce cas il n'est y a aucune possibilité pour exécuter les transactions avec le support HTM. Par conséquent, le mode logiciel (*SOFTWARE Mode*) convient dans ce scénario. Toutes les transactions sont exécutées dans le mode logiciel et il n'y a aucun besoin d'inter-opérer avec le matériel. Par conséquent les STMs existant peuvent être utilisés dans ce cas-ci pour exécuter les transactions.

- Le support HTM est disponible mais il n'est pas efficace tout le temps. Dans ce scénario il aura une combinaison entre les deux modes d'exécution, matériel et logiciel (*HYBRIDE Mode*), c.à.d. la MT hybride conviendrait par exemple HyTM de Moir et autres [12]. Cependant, dans ce mode on doit s'assurer que les conflits entre les transactions matérielles et logicielles concurrentes soient détectés et résolus correctement.

- Il y a peu de transactions ou la charge de travail est mono-thread. Dans ce mode *(SEQUNETIAL mode)* on exécute une seule transaction à la fois et par conséquent il n'y a aucun besoin de détection des conflits car il n'est y a pas des transactions concurrentes. Par conséquent on élimine un surcoût significatif.

- La charge de travail est mono-thread ou peu de transactions sont exécutées à la fois et quelques transactions sont connues pour ne pas être interrompues explicitement. Dans ce mode *(SEQENTIAL-NOABORT)* les transactions peuvent s'exécuter sans aucune détection de conflits et comme la transaction ne sera pas interrompue à cause d'un conflit alors le besoin d'enregistrement (mettre dans le *log*) est éliminé. Ce mode est semblable au mode *SÉQUENTIEL*. Cependant, *SEQUENTIAL-NOABORT* contrairement à d'autres modes supporte la fonctionnalité d'exécuter les opérations d'E/S ou les appels système à l'intérieure des transactions.

La plupart des scénarios pratiques tombent dans les deux premières catégories. D'ailleurs, supporter des modes dynamiques multiples pour des scénarios différents permet d'équilibrer la charge de travail ce qui donne une exécution efficace des transactions.

L'un des principaux défis qu'on fait face en mettant en application le PhTM est la transition d'un mode à un autre une fois c'est requis. Une solution consiste à interrompre toutes les transactions s'exécutant dans le mode actuel ou attendre leur terminaison avant le décalage vers le nouveau mode. Cette solution peut être implémentée en utilisant une variable en lecture seule appelée *modeIndicator*. En consultant cette variable chaque transaction peut décider qu'elle devrait s'interrompre ou continuer. Les transactions exécuteront si le mode n'a pas été changé et quand le mode se change, les transactions s'interrompront. Les transactions lisent la variable *modeIndicator* une fois et la gardent avec elles-mêmes pour la validation postérieure. En d'autres termes, une copie de *modeIndicator* sera une partie d'une transaction et une autre sera global et en comparant les deux copies, on conclut si le mode a été changé ou pas. Une autre approche peut être que toutes les transactions actuelles devraient terminer leur exécution tandis que la nouvelle transaction puisse s'exécuter dans le prochain mode.

Un autre défi majeur dans PhTM est de décider quand il faut commuter vers un autre mode [60]. Une solution simple dans ce cas-ci peut être qu'on exécute les transactions en phases dans chaque mode d'exécution et on surveille la performance. Quand on note que les transactions s'exécutent efficacement dans un mode d'exécution particulier alors PhTM peut donner plus de temps à ce mode pour exécuter les transactions. D'ailleurs le gestionnaire de conflit peut être utilisé pour surveiller la performance d'un mode d'exécution particulier pour aider à décider dans le décalage vers un autre mode d'exécution.

4.15.2 Discussion

L'avantage majeur de PhTM est de permettre de bénéficier des différents avantages de plusieurs systèmes de mémoire transactionnelle en utilisant chaque implémentation dans une phase particulière où elle sera la plus appropriée pour la charge de travail actuelle, ce qui facilite l'équilibrage de charge. Cependant, décider quand il faut décaler et vers quel mode est un point critique dans la performance de PhTM. De plus le décalage d'un mode vers un autre mode peut causer un retard qui peut diminuer la performance globale du système, donc plus de travail est nécessaire à cet égard pour développer des stratégies de décalage plus efficaces. D'ailleurs, l'utilisation des différents systèmes STMs et HTMs en même temps exigerait plus de mémoire.

4.16 DracoSTM

Le DracoSTM a été présenté par Gottschlich et Connors en 2007 [30]. DracoSTM est une bibliothèque STM C++ à base de verrous. Les travaux de recherches récents ont prouvé que les STMs basés sur les verrous sont plus performants que les STMs non bloquants [33,58]. DracoSTM est le premier STM qui implémente une stratégie de mise à mise à jour directe et différée et supporte la commutation entre les deux stratégies pendant l'exécution. Les détails de conception de base du DracoSTM sont affichés dans le Tableau 4.16.1. La conception de DracoSTM est un effort pour montrer que le problème d'évolutivité et d'autres problèmes comme les inter-blocages, l'inversion de priorité peuvent être évités en faisant des bons choix de conception.

4.16.1 Détails d'implémentation

DracoSTM est un STM qui intègre plusieurs propriétés qui rendent la mémoire transactionnelle une solution idéale pour la programmation parallèle; Il implémente une mise à jour directe et différée avec une composition de deux stratégies de détection conflit tôt et tard. Il permet à ses utilisateurs d'intégrer leurs propres gestionnaires de conflit. Il incorpore des mécanismes de récupération et gestion d'espace mémoire. Toutes ces propriétés sont

implémentées en utilisant les schémas de conception orientées objets et sans d'autres mécanismes qui pourraient réduire la sûreté (par exemple pointeurs vides) ou la flexibilité du système (solutions spécifiques au compilateur).

Tableau 4.16.1 : Les caractéristiques de base de DracoSTM.

DracoSTM	
Synchronisation	bloquante
Contrôle de Concurrence	Optimiste
Granularité	Object
Stratégie de Mise à jour	Déférée et Directe
Stratégie de Détection de Conflit	Tard et Tôt
gestionnaire de conflit	Abort
Transactions emboitées	Fermées

DracoSTM utilise un verrou par thread pour implémenter les opérations transactionnelles (de lecture et d'écriture). Quand une transaction T est sur le point de commiter, toutes les transactions sont temporairement bloquées excepté la transaction T [30]. Cette stratégie de verrouillage global qui bloque temporairement toutes les transactions en exécution et les permet de reprendre après le commit fournit les gains de performance d'un système non bloquant du fait que quand il n'est y a pas de commit, les transactions ne se bloquent pas et sont garanties de progresser vers l'avant.

DracoSTM utilise *l'invalidation*[7] pour assurer la consistance de l'ensemble de lecture. C'est le premier système STM qui utilise l'invalidation au moment de commit [30]. La différence de base entre la validation et l'invalidation est que l'invalidation peut détecter l'inversion de priorité plus facilement que la *validation*. L'inversion de priorité se produit dans la MT quand une transaction de priorité plus basse fait interrompre une transaction de priorité plus élevée. L'inversion de priorité peut être évitée en permettant à une transaction d'interrompre uniquement les transactions de priorité plus basse.

D'ailleurs, l'invalidation peut sauvegarder beaucoup d'opérations gaspillées par une notification tôt des transactions condamnées tandis que la validation ne peut pas. Car les systèmes réalisant la validation doivent réitérer à travers toutes les opérations transactionnelles et déterminer la cohérence seulement au moment de commit. De cette manière, chaque transaction doit entièrement exécuter ses opérations transactionnelles. Cependant, une quantité de travail substantielle peut être sauvegardée par un système d'invalidation qui peut marquer des transactions condamnées tôt.

[7] La *validation* est le processus fait par une transaction sur ses ensembles de lecture et écriture pour vérifier la consistance, la transaction aborte elle-même en cas d'inconsistance. Cependant, avec *l'Invalidation* la transaction vérifier la consistance sur les ensembles de lecture et écriture des autres transactions, la transaction aborte les autres transactions en cas d'inconsistance.

L'une des nouvelles caractéristiques introduite par DracoSTM est sa capacité de permuter pendant l'exécution entre les deux techniques de mise à jour de données (directe et différée). Cependant, cette commutation exige qu'aucune transaction ne soit en exécution lors de commutation. Pour commuter entre les deux modes de mise à jour et identifier qu'elle est la stratégie active, DracoSTM utilise les quatre interfaces statiques suivantes fournies à partir de la classe *transaction* :

- *bool do_direct_updating():* essaie de commuter vers la mise à jour directe, elle revoie *True* en activant la mise à jour directe en cas de succès, elle s'échoue (renvoie *False*) en cas où des transactions sont en cours d'exécution lors de commutation.

- *bool do_deferred_updating()* essaie de commuter vers la mise à jour différée, elle revoie True en activant la mise à jour différée en cas de succès, elle s'échoue (renvoie False) en cas où des transactions sont en cours d'exécution lors de commutation.

- *bool direct_updating()* renvoie *True* si la mise à jour directe est active et *False* autrement.

- *bool deferred_updating()* renvoie *True* si la mise à jour différée est active et *False* autrement.

De même, DracoSTM peut commuter entre les deux stratégies de détection de conflit (tôt et tard) pendant l'exécution en s'assurant qu'aucune transaction n'est en exécution. Cependant, DracoSTM ne permet pas une mise à jour différée avec une stratégie de détection de conflit tôt et le système prévient un tel comportement. La classe transaction fournit Les quatre interfaces statiques suivantes pour gérer la commutation:

- *bool do_early_conflict_detection():* essaie de commuter vers la détection de conflit tôt, elle échoue et renvoie *False* en cas où des transactions sont en cours d'exécution ou la stratégie de mise à jour différée est active. Autrement elle commute avec succès vers la détection de conflit tôt et renvoie *True*.

- *bool do_late_conflict_detection():* essaie de commuter vers la détection de conflit tard, elle échoue et renvoie *False* en cas où des transactions sont en cours d'exécution. Autrement elle commute avec succès vers la détection de conflit tard et renvoie *True*.

- *bool early_conflict_detection() :* renvoie *True* si la mise à jour directe avec une détection de conflit tôt est active. Autrement elle renvoie *False*.

- *bool late_conflict_detection() :* renvoie *True* si la mise à jour différée est active ou si la mise à jour directe avec une détection de conflit tard est active. Autrement elle renvoie False.

DracoSTM peut utiliser les deux techniques de détection de conflit avec la stratégie de mise à jour directe. Mais dans le cas de mise à jour différée, le système permet uniquement une détection de conflit en retard, car la détection de conflit tôt dans ce cas peut empêcher beaucoup transactions de commiter avec succès et engendre un perd de plusieurs optimisations [30]. La mise à jour directe permet à une seule transaction d'écrire à un emplacement mémoire à la fois et par conséquent la détection de conflit tôt est plus pertinente dans ce cas. Cependant, la détection de conflit tard est activée uniquement au moment de commit car elle permet certaines optimisations [30].

DracoSTM utilise l'*invalidation* pour assurer la cohérence de l'ensemble de lecture. Dans cette technique, une transaction informe d'autres transactions qu'elles doivent aborter en mettant le flag *forced_to_abort* à True. Chaque fois qu'une transaction effectue une lecture, écriture ou essaie de commiter, elle doit d'abord vérifie son flag *forced_to_abort*. Si ce flag est activé, la transaction consulte immédiatement le gestionnaire de conflit pour décider ce qu'elle doit faire. Si le gestionnaire de conflit décide d'aborter une transaction, cette transaction réalise une courte attente pour éviter de créer d'autres contentions (au niveau de la mémoire ou les verrous) avant de lancer l'exception *aborted_transaction_exception* qui sera captée par le système et par la suite mettre fin à la transaction.

DracoSTM utilise les opérations *new_memory()*, *new_memory_copy()* et *delete_memory()* pour gérer les opérations mémoires. Chaque transaction maintient une liste des emplacements qu'elle accède, les éléments de cette liste seront supprimés et la liste sera vidée une fois la transaction commite ou aborte [30].

DracoSTM implémente un mécanisme de gestion de conflit extensible, qui permet aux utilisateurs d'intégrer leurs propres gestionnaires de conflit. Pour ce faire, les utilisateurs utilisent les deux interfaces suivantes fournies par a classe *transaction*:

- *void contention_manager (base_contention_manager *rhs)* : elle rend le gestionnaire de conflit de DracoSTM pointe vers le gestionnaire de conflit passé en paramètre en écrasant l'ancien gestionnaire de conflit.

- *base_contention_manager* get_contention_manager()* : elle revoie un pointeur vers le gestionnaire de conflit actuel. Cette méthode est généralement utilisée après la précédente pour s'assurer l'intégration avec succès du gestionnaire de conflit de la part de l'utilisateur.

DracoSTM met en application des transactions emboîtées fermées d'où les modifications faites par les transactions fils sont visibles uniquement pour leur père et vise versa. Les autres transactions ne peuvent pas voir les états intermédiaires c.à.d. les modifications seront visible après le commit de la transaction la plus externe.

4.16.2 Discussion

DracoSTM supporte une nouvelle caractéristique dans la stratégie de mise à jour, cependant la stratégie de mise à jour différée est mieux adaptée avec le mécanisme d'invalidation au moment de commit que la mise à jour directe [30], Bien que dans quelques systèmes [16] la mise à jour directe soit plus rapide que la mise à jour différée, mais ces systèmes n'utilisent pas l'invalidation au moment de commit. Donc La stratégie de mise à jour est étroitement attachée avec la stratégie de détection de conflit. Néanmoins, une limitation de DracoSTM est que les transactions ne peuvent pas commuter vers une autre stratégie de mise à jour pendant l'exécution. D'abord, toutes les transactions doivent être suspendues ou terminées pour que DracoSTM puisse initialiser la nouvelle stratégie de mise à jour. D'ailleurs, le changement d'une stratégie de mise à jour à une autre stratégie entraîne un retard, qui détériore par la suite le temps de réaction de système entier. DracoSTM utilise le verrouillage est montre qu'il surpasse d'une manière significative [30] (dans certains cas plus de deux fois) le RSTM qui est un STM non bloquant. Cependant, il utilise l'invalidation au moment de commit pour profiter des avantages de la synchronisation non bloquante c.à.d. empêche l'inversion de priorité, les inter-blocages et les livelocks.

4.17 Nonblocking Software Transactional Memory *(NBSTM)*

Virendra Marathe et Mark Moir [85] ont développé en 2008 un système STM non-bloquant qui intègre la plus part des techniques d'optimisation adaptées par les STMs bloquants. Le but de leur travail est de développer un STM non bloquant compétitif au STMs bloquants que les dernières années ont montré quels sont les plus performants. Les détails de conception de base sont également affichés dans le Tableau 4.17.1.

Tableau 4.17.1 : Les caractéristiques de base de NBSTM.

NBSTM	
Synchronisation	Liberté d'obstruction
Contrôle de Concurrence	Optimiste
Granularité	Mot
Stratégie de Mise à jour	Directe ou différée
Stratégie de Détection de Conflit	Tard
Stratégie de résolution de conflit	Vol

NBSTM (Phase2-STM) est le premier STM non-bloquant qui utilise la technique de validation à base d'estampille [85]. Il utilise une granularité de niveau mot (des blocs mémoire contigües) et garde les métadonnées *(orecs)* séparément des données. Il adapte le mécanisme non-bloquant de *copyback* de system de Harris et Fraser (WSTM section 4.2) avec quelques améliorations.

NBSTM utilise une nouvelle technique appelée *vol* pour permettre à une transaction de voler la propriété d'un emplacement mémoire d'une autre transaction, et des techniques de gestion de méta-données pour permettre l'intégration de plusieurs optimisations utilisées par les STMs bloquants récents telles que la validation basée sur les estampilles et la libération des propriétés par l'intermédiaire des simples instructions de sauvegarde (store) [85], tout en conduisant à un chemin d'exécution rapide et plus efficace.

Le NBSTM est implémenté avec un *undo log* (mise à jour directe) qui n'a jamais été utilisé avec les STMs non-bloquant [85], et une variante avec un *redo log* (mise à jour différée) *avec les deux techniques* d'acquisition de propriété (*tôt* et tard*)* pour les écritures faites par les transactions.

4.17.1 Détails d'implémentation

Le travail présenté dans le papier est procédé en deux phases : **dans la première phase** on essaie d'imiter le comportement des STMs bloquants le plus possible, et faire recourt à la gestion plus chère de déplacement de données et de métadonnées seulement dans les situations où les transactions ont des problèmes pour accomplir le progrès vers l'avant, ce qui donne une performance comparable au STMs bloquants dans le cas commun [85]. Dans cette phase, le STM non-bloquant permet à une transaction de voler la propriété d'un emplacement mémoire d'une autre transaction au lieu d'attendre son libération. L'accès aux emplacements volés est

plus compliqué et plus coûteux que L'accès aux emplacements non volés [85]. Cependant, malgré les coûts générés par cette technique, elle est très utile afin d'éviter d'attendre une transaction qui est retardée pendant longtemps, (par exemple en raison de préemption). De plus le système commute rapidement les emplacements *volés* de nouveau à l'état *non volé* afin de réduire au minimum le surcoût éventuel qui se produit pendant le vol en raison du conflit élevé [85].

Dans la deuxième phase, on se concentre sur l'incorporation des techniques d'optimisation telles que la validation en utilisant les estampilles adaptées par des STMs bloquants (par exemple TL2).

Phase-1 STM

Dans cette section on va décrire le *Phase-1 STM,* un STM *non-bloquant* qui est très similaire à le *HyTM* bloquant (décrit dans la section 4.11) cependant il inclut quelques optimisations, ce système sera modifié par la suite (phase-2) en y intégrant des techniques adaptées avec les STMs bloquants.

L'API du système *Phase-1 STM* implémente les méthodes suivantes qui sont similaires aux autres STMs basés sur le mot :

- ***stm_begin*** *(TxnDescriptor* my_txn)*
- ***stm_commit*** *(TxnDescriptor* my_txn)*
- ***Word_t stm_read*** *(TxnDescriptor* my_txn, Word_t* addr)*
- ***void stm_write*** *(TxnDescriptor* my_txn, Word_t* addr, Word_t value).*

Dans ce système, Chaque transaction est représentée par une structure de données appelée *descripteur de transaction* qui contient les champs ; *identificateur de transaction (TID)* qui est unique pour chaque transaction, un *numéro de version (version),* un *statut (Active, Committed, et Aborted),* et des *ensembles de lecture* et *d'écriture*. Chaque emplacement mémoire est associé à une entée dans une table de propriété *(orecs)* pour désigner la transaction propriétaire de cet emplacement. Les ensembles de lecture et d'écriture sont organisés suivant les lignes d'orec, telles que toutes les entrées relatives à des emplacements mémoire mappés au même orec sont enregistrées dans la même ligne. Chaque ligne dans les deux ensembles contient un identificateur d'orec *(orec_ID),* un *snapshot* de l'orec associé *(orec_snapshot),* et un tableau des entrées. Chaque entrée dans la liste de lecture contient l'adresse *(addr)* et la valeur *(old_value)* de l'emplacement associé à l'***orec*** indiqué. Cependant, une entrée dans la liste d'écriture contient l'adresse *(addr)* et l'ancienne et la nouvelle valeur *(old_value et new_value)* de l'emplacement associé à l'*orec* indiqué. La Figure 4.17.1 schématise la structure d'un descripteur de transaction.

Chaque entrée dans *l'orec* est un mot mémoire de 64 bits qui peut être modifié par un *CAS* atomique, ce mot contient les champs; *tid* et *version* utilisés pour identifier le propriétaire actuel de *l'orec,* et le champ *row_offset* pour identifier l'emplacement mémoire où la transaction propriétaire maintient les données mappées à cet *orec*. Le maintient de l'identificateur de la transaction propriétaire *(tid)* et sa version *(version)* dans l'*orec* est une *nouvelle technique de libération* de l'*orec* qui est optimisée et très rapide. Pour ce faire, la transaction peut simplement incrémenter sa version pour libérer implicitement la propriété des *orecs* qu'elle possède[8] [85].

[8] Le numéro de version dans le champ *version* d'un *orec* est utilisé pour déterminer si la *version* courante de descripteur de transaction pointé par *l'orec* est celui qui acquiert *l'orec*.

Ceci permet d'éliminer le surcoût généré par la libération explicite des *orecs* en utilisant des *CAS* coûteuses (comme ceux utilisés par WSTM section 4.2).

Une transaction en exécution a son statut dans l'état *Active* et utilise son ensemble d'écriture comme un log de refaire (redo log). Elle acquiert les Orecs dés le premier accès en écriture (tôt). Pour assurer la cohérence, elle valide son ensemble de lecture en entier lors de chaque accès à un nouvel *orec* (les techniques de validation basées sur l'estampille [16] n'ont pas été inventées quand cet algorithme était conçu). Pour commiter, une transaction commute atomiquement son statut d'*Active* à *Committed* et copie les valeurs de son ensemble d'écriture (redo log) aux emplacements mémoire correspondants, et puis libère les *orecs* possédés en incrémentant son numéro de version. Si une transaction *T* essaie d'accéder à un emplacement déjà possédé par une autre transaction S, *T* (voleur) peut "voler" l'orec *O* correspondant possédé par *S* (victime) dès que *S* change son statut à *Committed* (c.à.d *S* est entrain de copier les valeurs modifiées vers leurs emplacements à partir de son ensemble d'écriture *Rv*). Pour ce faire *T* utilise un *CAS* sur *O* (le champ *row_offset*) pour le rendre indiquer une ligne dans l'ensemble d'écriture du *T*. Cette ligne sera utilisée par *T* pour enregistrer ses mises à jour ultérieures apportées aux emplacements mémoire mappés à *O*. Cependant, le système doit prendre certaines précautions pour s'assurer que les valeurs logiques des emplacements mappés à *O* (qui sont dans une ligne appelée *Rv*) volés soient correctement préservées pendant le processus de vol. Pour cela, *T* doit d'abord fusionner *Rv* dans une ligne (qui soit *Rs*) dans son ensemble d'écriture. Si le vol réussit, *O* doit pointer vers *Rs* [85].

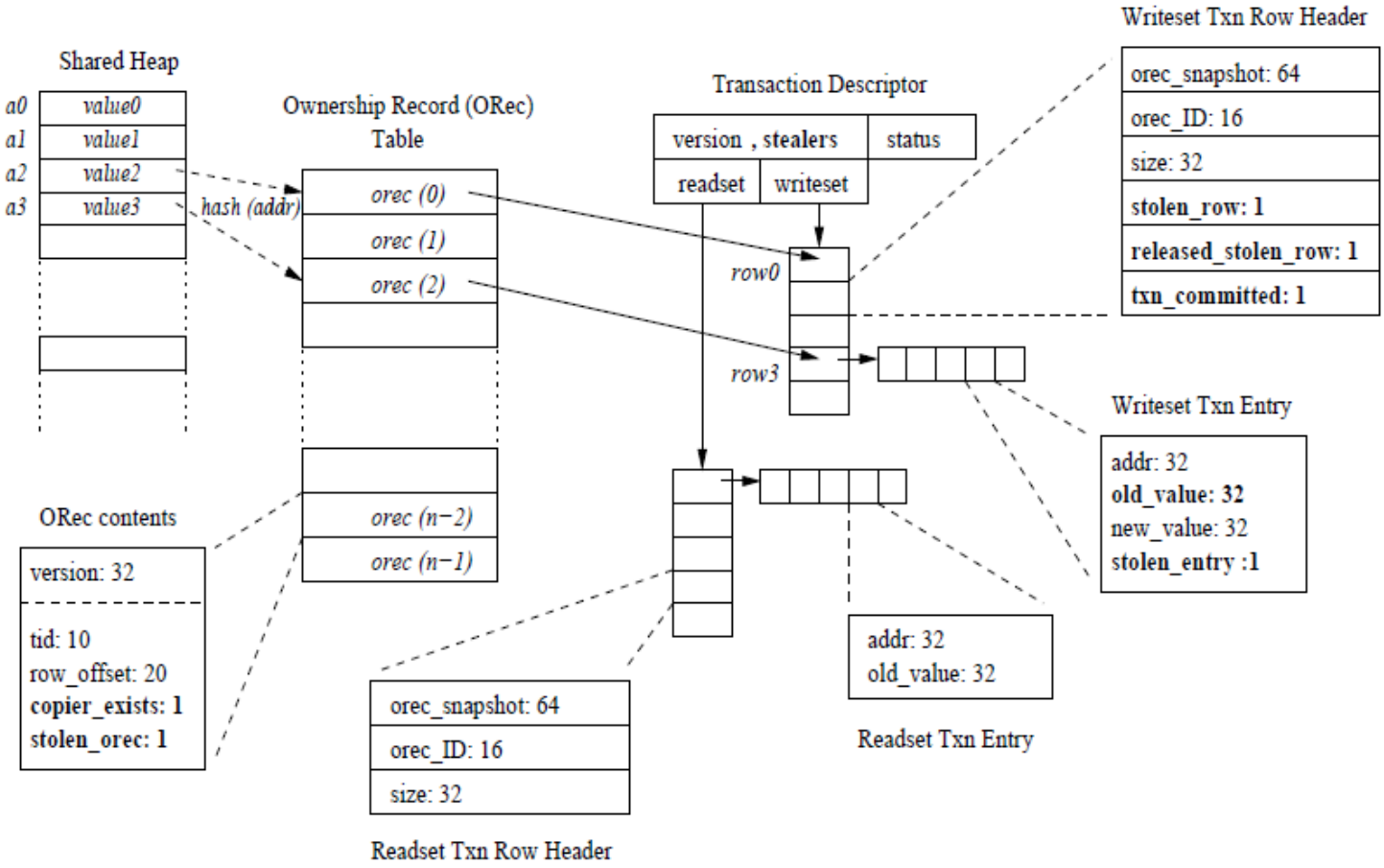

Figure 4.17.1 : Les structures du NBSTM

(Les champs en gras sont requis uniquement par le STM Non-bloquant) [85]

Après le vol d'*O*, le voleur *T* peut terminer avant que la victime *S* termine sa phase de copiage *(copyback)* vers les emplacements mappés à *O* car ce copiage peut être retardé arbitrairement [85]. Dans ce cas-ci, les valeurs logiques des emplacements mappés dans *O* résident dans le *redo log* de voleur (plus précisément dans *Rs*) et la victime *S* peut écraser les modifications apportées par *T* par des valeurs "périmées". Pour éviter un tel cas, le nouveau flag *(stolen_orec)* de l'orec volé (voir la Figure 4.17.1) est utilisé pour indiquer que l'orec *O* est dans l'état *volé*, et les valeurs logiques des emplacements mappés à *O* peuvent être résidées dans une ligne dans l'ensemble d'écriture pointé par *O* (*Rs* dans l'exemple), et par conséquence, *Rs* ne doit pas être réutilisé par une transaction ultérieure exécutée par le même thread (c.à.d. en utilisant le même descripteur de transaction *tid*); ce *Rs* peut être réutilisé seulement quand *O* ne sera plus pointé vers *Rs*. Le flag *stolen_row* est utilisé (Figure 4.17.1) pour ce but, c.à.d. il est utilisé dans une ligne d'un ensemble d'écriture pour indiquer qu'elle est pointée par un *orec O volé*, (*Rs* dans notre exemple). Ce flag *(stolen_row)* est activé pendant le processus de vol et il doit être effacé par une transaction ultérieure *B* lorsqu'elle veut *revoler* le *O* de *Rs*. La remise à l'état initial de l'indicateur *stolen_row* se produit indirectement, d'où le nouveau voleur active l'indicateur *released_stolen_row* dans *Rs*, qui sera utilisé ensuite par la victime afin de "récupérer" *Rs* pour une réutilisation ultérieure. Si le voleur réussit à commiter, alors les nouvelles valeurs logiques des emplacements mappés à *O* sont dans son redo log *(new_values)* et l'indicateur *txn_committed* est activé, autrement les anciennes valeurs *(old_values)* demeurent les valeurs logiques.

Il est à noter que puisque les valeurs logiques des emplacements mappés à un *orec* volé *O* sont maintenues dans une ligne dans l'ensemble d'écriture pointée par *O*, il n'y a aucun besoin pour les voleurs de recopier leurs mises à jour aux emplacements mappés à *O*. C'est seulement la première victime qui recopie ses mises à jour. Après que la première victime termine son *copyback*, elle vérifie que tous les *orecs* qu'elle possède n'étaient pas volés. (Ceci est fait avec une liste des voleurs (*stealers*) dans le descripteur de transaction (Figure 4.17.1) qui est accédée atomiquement avec sa zone de *version*. Le voleur s'ajoute atomiquement dans la liste *stealers* pour informer la victime au sujet du vol d'un *orec*. Plus de détails peuvent être trouvés dans [64]). Cependant, si un *orec O* était volé, la victime a la possibilité d'informer le système qu'elle a terminé son copyback pour *O*.

L'accès aux emplacements volés est cher. Par conséquent il est important de commuter un *orec* de nouveau à l'état *non volé* aussi rapidement que possible. Puisque la première victime est la seule transaction faisant un *copyback* des emplacements mappés à *O*, l'unique flag *copier_exists* de *O* suffit pour informer le système que le seul copieur d'*O* a terminé. Le premier voleur qui vole *O* active les deux flags *stolen_orec* et *copier_exists*. La première victime, après la fin de son *copyback*, remet à l'état initial le flag *copier_existes* pour indiquer au système qu'elle a terminé son *copyback*. Cet état de l'orec permet au système de commuter sans risque l'orec de nouveau à l'état *non volé* ; pour cela, le prochain voleur d'*O* peut sans risque assumer la responsabilité de faire le copyback car aucun voleur n'est en train de le faire (en activant l'indicateur *copier_exists* par un *CAS* pendant le vol). Il recopie les valeurs logiques les plus récentes des emplacements mappés à *O* dans sa ligne dans l'ensemble d'écriture, et ultérieurement reset les deux indicateurs *stolen_orec* et *copier_exists* (en utilisant un CAS). Ceci commutera *O* de nouveau à l'état *non volé*.

Phase-2 STM

Dans cette section, nous décrierons l'adaptation de certaines optimisations utilisées avec les STMs bloquants pour améliorer la performance du STM non-bloquant décrit précédemment (phase-1 STM). Ces améliorations consistent à utiliser la technique de libération de propriété

expliquée précédemment qui facilite l'intégration de la technique de validation basée sur les estampilles. Nous verrons aussi quelles sont les modifications apportées pour pouvoir utiliser une acquisition des propriétés en retard et l'undo log avec un système non-bloquant.

La technique de validation basée sur les estampilles consiste à utiliser une horloge globale partagée. Quand une transaction commence l'exécution elle lit cette horloge et enregistre sa valeur dans la variable locale *begin_timestamp*. Cette valeur est utilisée, pendant l'exécution de la transaction, pour déterminer si les emplacements accédés par la transaction sont cohérents. Chaque orec contient une estampille qui approxime "le temps logique" dans lequel cet *orec* était modifié pour la dernière fois par une transaction. Une transaction assure qu'elle visualise une version cohérente d'un *orec O* si la valeur de l'estampille de *O* est inférieure ou égal à celle enregistrée dans la variable *begin_timestamp* de cette transaction. Lors de la phase de commit, une transaction peut revalider son ensemble de lecture en entier.

Phase-2 STM utilise une technique de libération de propriété efficace, cette technique consiste à superposer l'estampille et le numéro de version de la transaction dans le champ *version* des *orecs*. Initialement, ce champ contient une estampille, et lors de son acquisition par une transaction *T*, *T* permute sa version[9] (la version est différenciée de l'estampille par le bit le moins significatif) dans le champ *version* de cet *orec* par un CAS. Pour la libération, *T* lit le temps global et écrase le champ *estampille* de tous les *orecs* acquis par cette valeur en utilisant une simple écriture.

Phase-2 STM utilise un ensemble d'écriture plus ou moins identique à celle de Phase-*1STM* (sauf l'ajout des deux nouvelles champs *copier_ID* et *stealing_time_version* dans la structure de données d'une ligne de l'ensemble d'écriture). Le champ *copier_ID* est mis par le premier voleur *d'orec* correspondant et il contient l'identificateur *ID* de la victime. Le champ *stealing_time _version* contient la *version* courante du voleur *d'orec* correspondant pendant l'opération de vol, chaque transaction qui accède à un *orec* dans l'état volé utilise ce champ pour déterminer la version courante du propriétaire d'orec (ou l'ancienne version d'un propriétaire qui a possédé cet orec).

Cependant, l'ensemble de lecture n'est pas identique à l'ensemble d'écriture, elle est sous forme d'une liste chainée de blocs contigus de 256 entrées chacune. Chaque entrée se compose d'une adresse *d'orec*, qu'elle est utilisée au moment de commit pour la validation de l'ensemble de lecture. Ainsi, les entrées redondantes pour le même emplacement peuvent exister dans l'ensemble de lecture dans le cas des grandes transactions, cette redondance pour être éliminée en filtrant périodiquement l'ensemble de lecture [85]. Lors d'une lecture par une transaction *T* d'un emplacement mémoire mappé à un *orec* possédé par cette dernière, *T* recherche la ligne correspondante à cet *orec* dans son ensemble d'écriture pour récupérer l'éventuelle valeur modifiée par *T (read-after-write lookup*). Cette recherche est inhérente dans les implémentations basées sur le redo log, mais pas dans celles basées sur l'undo log.

L'algorithme présenté jusqu'ici utilise une stratégie d'acquisition tôt pour les écritures. Pour la rendre tard (acquisition au moment de commit). Une liste linéaire d'écriture est introduite, cette liste est très semblable à celle utilisée pour l'ensemble de lecture (la seule différence étant que les entrées de cette liste d'écriture contiennent des paires *adresse-valeur* pour maintenir le redo log de la transaction. Cette liste est parcourue lors de commit en acquérant les orecs correspondants.

[9] Le champ *version* d'un *orec* libre contient une estampille et le flag *stolen_orec* correspondant est mis à false.

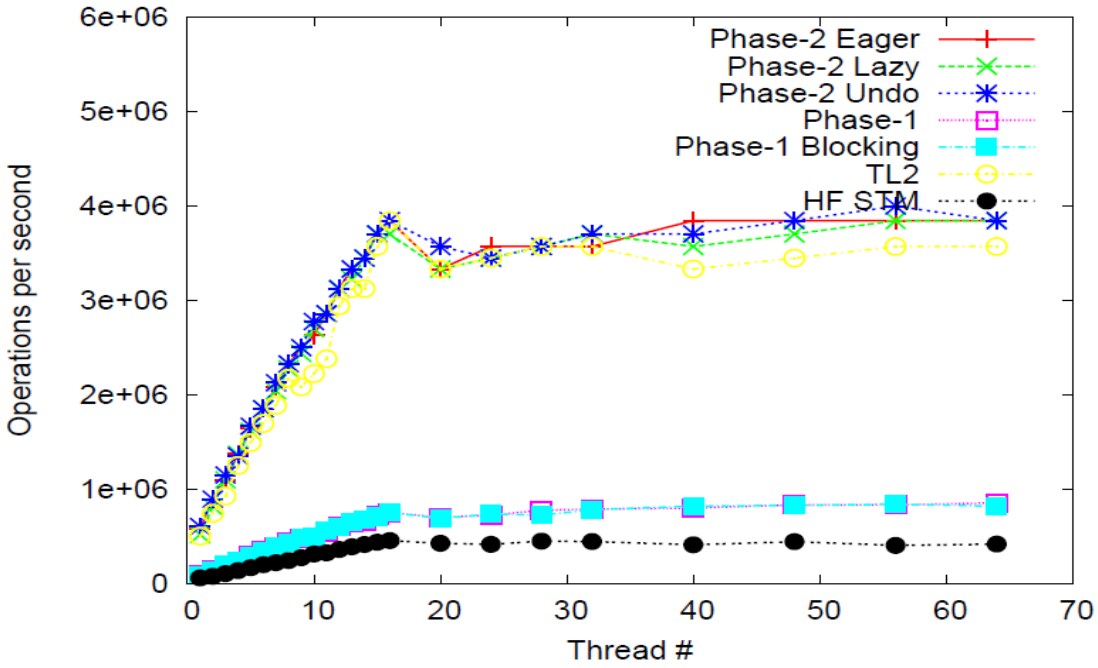

Figure 4.17.2 : Comparaison de performance de NBSTM avec TL2 et HF-STM

(phase-1 est la version bloquante et phase-2 est la version non bloquante) [85]

L'implémentation d'un STM bloquant avec un undo log (mise à jour directe) avec un système non bloquant peut donner une bonne performance car elle élimine le surcoût engendré par la recherche dans l'ensemble d'écriture pour récupérer des valeurs modifiés (*read-after-write*) faite par les STMs basés sur le redo log (mise à jour différée). Cependant, les STMs bloquants basés sur l'undo log souffrent d'un temps d'attente arbitraire en attendant une transaction interrompue à libérer les propriétés des emplacements acquis. Cependant, un STM non-bloquant basé sur l'undo log peut profiter des avantages de deux techniques [85].

L'implémentation d'un tel STM avec un undo log est très simple. Dans les STMs basés sur le redo log, le vol est nécessaire dans le cas où une transaction contradictoire a déjà commité puisque les valeurs logiques de toutes ses mises à jour peuvent résider dans son ensemble d'écriture. Le même raisonnement s'applique aux transactions contradictoires interrompues dans la variante du STM non-bloquant avec un undo log (les valeurs réelle des emplacements accédés par la transaction interrompue sont dans l'undo log), de plus dans tel STM, le vol se produit principalement quand les transactions s'interrompent [85]. Mais il peut être nécessaire même lorsqu'une transaction rencontre une transaction déjà commise. (C'est un outil de la politique de libération d'*orec* basée sur l'instruction store, qui est maintenue dans le STM utilisant l'undo log [85]). L'opération de libération d'*orec* par une transaction peut interférer avec l'acquisition du même *orec* par une autre transaction. Cependant, le vol est nécessaire dans ce cas-ci seulement pour maintenir une vue cohérente de l'état de l'*orec*. Les valeurs logiques des emplacements modifiés par la victime résident déjà dans les emplacements réels. Ainsi il n'est pas nécessaire de fusionner la ligne de l'ensemble d'écriture de la victime dans l'ensemble d'écriture du voleur. Toutes autres parties de l'algorithme de vol non-bloquant demeurent les mêmes dans le STM avec l'undo log.

Dans la Figure 4.17.2 il est montré le résultat de comparaison de performance de NBSTM avec TL2 et HF-STM. L'étude empirique a prouvé que ces améliorations ont rendu le système non bloquant compétitif au TL2 l'un des leaders des STMs bloquants.

4.17.2 Discussion

NBSTM est le premier STM non-bloquant qui a réussi à diminuer le gap de performance entre les STMs bloquants et ceux non-bloquant, c'est le premier STM non-bloquant qui utilise un undo log. Les résultats empiriques ont montré que toutes les variantes du NBSTM non-bloquantes sont compétitives aux STMs bloquants ce qui donne un espoir pour construire des STMs non bloquants plus performants.

4.18 Tuning of W-B Software Transactional Memory (TinySTM)

Felber, Fetzer et Riegel ont développé en 2008 TinySTM [72], un système STM basé sur les verrous qui a une granularité de niveau mot. Sa nomination (Tiny) est due à sa simplicité et sa performance [72]. TinySTM est un STM à base de temps, il utilise une version de l'algorithme LSA (section 4.9.1.1) et il est très similaire à TL2 (section 4.6) sauf qu'il utilise un verrouillage dès le premier accès en écriture *(encounter-time locking).* Ce choix est motivé par l'étude empirique, car le verrouillage dès le premier accès augmente le débit des transactions et permet de surmonter le problème de *lecture-après-écriture* sans besoin de mécanismes coûteux et complexes contrairement au verrouillage au moment de commit qui évite certaines conflits de lecture/écriture mais en générale les conflits détectés au moment de commit sont résolus en abortant au moins une transaction [72]. TinySTM introduit un nouveau mécanisme pour minimiser le coût de validation des grands ensembles de lecture. Les détails de conception de base sont également affichés dans le Tableau 4.18.1.

Tableau 4.18.1 : Les caractéristiques de base de TinySTM.

TinySTM	
Synchronisation	Bloquante
Contrôle de Concurrence	Mixte
Granularité	Mot
Stratégie de Mise à jour	Directe ou différée
Stratégie de Détection de Conflit	Écritures Tôt et Lectures Tard
Stratégie de résolution de conflit	Abort

4.18.1 Détails d'implémentation

TinySTM est un système bloquant avec une granularité au niveau mot et il peut utiliser soit une stratégie de mise à jour directe ou différée. Chaque bloc mémoire est mappé à une table de verrous (en utilisant une fonction de hash). Le bit le moins significatif de chaque verrou dans la table est utilisé pour indiquer si le verrou est possédé ou non ; ce bit est mis à *1* une fois il est verrouillé et les bits restant contiennent soit un pointeur vers la transaction propriétaire (dans le cas de *mise à jour directe*), ou une entrée dans la liste d'écriture de la transaction propriétaire (dans le cas de *mise à jour différée)* ce qui permet à une transaction de localiser rapidement dans son ensemble d'écriture les emplacements mémoire mis à jour couverts par ce verrou lorsqu'ils seront consultés de nouveau par la même transaction. Cette technique est une amélioration par rapport à TL2 qui est obligé de vérifier (dans l'ensemble d'écriture) lors de chaque accès à un

emplacement mémoire modifié par la transaction, cette recherche est couteuse dans le cas des grands ensembles d'écriture (TL2 utilise le filtre Bloom pour éviter de traverser inutilement l'ensemble d'écriture). Cette technique écarte le problème de *lecture-après-écriture* (dans le cas de mise à jour différée) car la mémoire contient (indique) toujours la dernière valeur écrite par la transaction active [72]. Cependant, si le verrou est libre, le bit le moins significatif est mis à *0* et les bits restants maintiennent une estampille (numéro de version) qui correspond à l'estampille lu lors de commit par la dernière transaction qui à écrit dans l'un des emplacements mappés à ce verrou.

Lors d'une opération d'écriture par une transaction *T* à un emplacement mémoire *m*, *T* d'abord identifier le verrou *o* correspondant à *m* et lire sa valeur atomiquement. Si le verrou est actif, *T* vérifie si elle est le propriétaire du verrou en utilisant l'adresse enregistrée dans les bits restants de verrou. Dans ce cas, elle écrit simplement la nouvelle valeur et retourne. Autrement, *T* s'interrompre immédiatement. Cependant, si le verrou *o* est libre, *T* essaie de l'acquérir en écrivant une nouvelle valeur dans *o* avec une *CAS*. L'échec de *CAS* indique qu'une autre transaction a acquis le verrou avant *T* et donc la procédure en entier sera relancée [72].

Lors d'une opération de lecture par une transaction *T* d'un emplacement mémoire *m. T* doit vérifier d'abord que le verrou *o* correspondant à *m* est libre et il n'est pas modifié concurremment. Pour cela, *T* lit le verrou *o*, puis *m* et finalement le verrou *o* de nouveau. Si le verrou n'est pas possédé et sa valeur (c.à.d. numéro de version) n'est pas changée entre les deux lectures, alors la lecture de la valeur est cohérente [72].

Avec la technique de mise à jour directe Il y a un problème subtil [72]. Considérant une transaction *T1* lisant le verrou *o*; puis d'autres transactions *T2* acquis le verrou *o* et écrit une nouvelle valeur à l'emplacement mémoire correspondant à *o* qui est lu par *T1* ; ensuite *T2* s'interrompt et restaure la valeur initiale du verrou ; en conclusion, *T1* lit le verrou *o* de nouveau et ne détecte pas un accès concurrent, bien qu'elle ait lu une valeur incohérente. Pour résoudre ce problème, on sauvegarde dans le verrou un *numéro d'incarnation* supplémentaire sur trois bits qui est incrémenté chaque fois une transaction s'aborte, ce numéro permet de détecter les accès concurrents dans tels scénarios. Dans le cas d'un débordement qui est peu probable, un nouveau numéro de version est simplement obtenu à partir de l'horloge globale. Noter que ce problème ne s'applique pas avec une mise à jour différée.

Lors d'une lecture d'une nouvelle valeur, TinySTM vérifie si elle peut être utilisée pour construire un snapshot cohérent. Similairement à LSA (section 4.9.1.1), si la version est plus récente que l'intervalle de validité actuel du snapshot de la transaction, on essaie "d'étendre" le snapshot. En d'autres termes en vérifiant que toutes les adresses dans l'ensemble de lecture sont encore valides et ne sont pas verrouillées par d'autres transactions. Si l'extension réussit, la borne supérieure de l'intervalle de validité du snapshot peut être mise à la valeur de l'horloge lue juste avant l'extension. Autrement, la transaction aborte. Comme on peut remarquer, les transactions de lecture seule sont particulièrement efficaces car aucune validation n'est nécessaire au moment de commit du fait qu'on construit incrémentalement un snapshot qui est valide à tout moment, par conséquent, il n'est y a aucun besoin de maintenir un ensemble de lecture [72].

TinySTM utilise une variable globale partagée (de 32 bits ou 64 bits) comme une horloge, cette variable est lue au démarrage de chaque transaction et incrémentée lors de commit d'une transaction d'écriture. Cette technique est simple et largement suffisante, cependant si ce compteur devient un goulot dans le cas des grands systèmes, TinySTM peut adapter une base de temps plus évolutive comme une horloge externe ou plusieurs horloges physiques synchronisés [72].

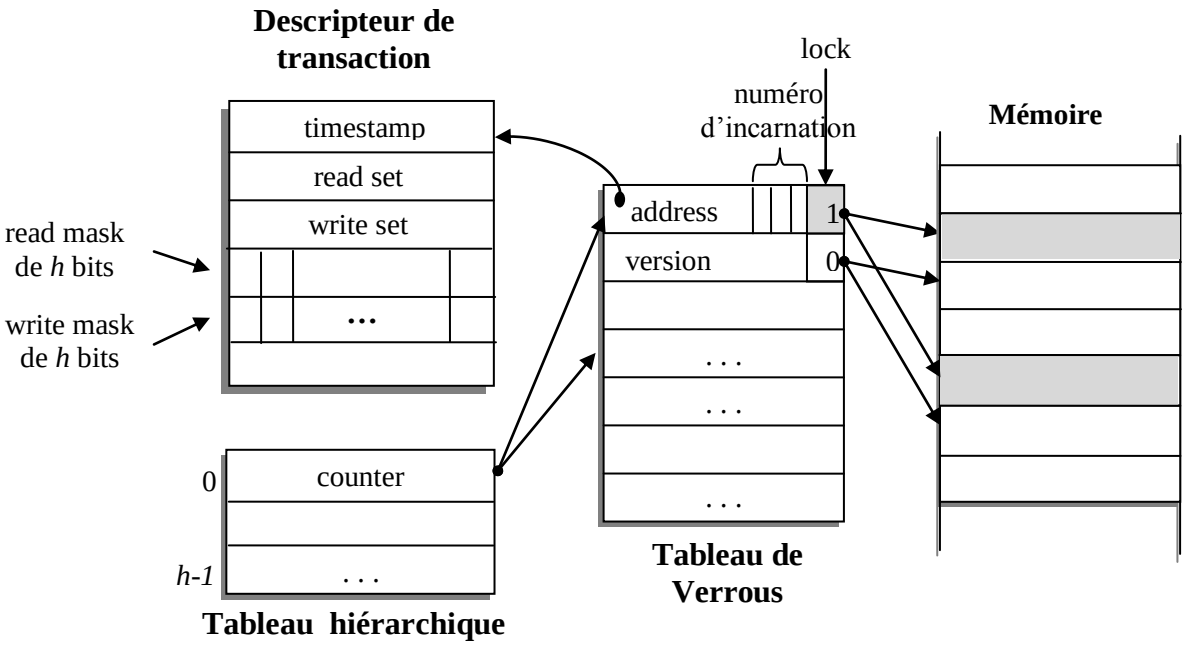

Figure 4.18.1 : Structures de données utilisées par TinySTM

La valeur maximale de l'horloge est 2^{31} sur une architecture de 32-bits, et 2^{63} sur une architecture 64-bits dans les systèmes de 32 bits avec taux de commit fréquent, cette valeur peut être rapidement atteinte. Par conséquent, TinySTM fournit un simple mécanisme de roll-over d'horloge: quand une transaction détecte que la valeur maximale d'horloge a été atteinte, elle s'interrompt et attend sur une barrière jusqu'à ce que toutes les transactions actives terminent leur exécution. Puis, on remit à l'état initial l'horloge et tous les numéros de version (Le surcoût de cette procédure est négligeable car elle s'exécute rarement).

Lors de la phase de commit, une transaction qui réalise des mises à jour doit valider son ensemble de lecture pour assurer la cohérence des données lues c.à.d. elle doit vérifier que les adresses lues ont resté avoir les mêmes numéros de version. Cependant, si le temps de commit d'une transaction égale à son temps de démarrage plus un alors cette transaction n'a pas besoin de valider son ensemble de lecture (car elle est sûr qu'aucune autre transaction n'a concurremment modifié la mémoire).

La validation des ensembles de lecture peut être coûteuse lorsqu'ils sont grands (les transactions effectuent beaucoup de lectures). Pour minimiser ce coût, TinySTM utilise une stratégie de *verrouillage hiérarchique* où une table supplémentaire de *h* compteurs est utilisée (h << *l* la taille de la table de verrou, typiquement *h*= de *4* à 16) (Figure 4.18.1). Chaque entrée dans cette table couvre un ensemble de verrous dans la table de verrou est par conséquent elle couvre les emplacements mémoire correspondants.

Chaque transaction maintient de plus deux structures de données privées: un *masque de lecture* et *un masque d'écriture* de *h* bits chacun et les ensembles de lecture sont partitionnées en *h* pièces indépendantes. Lors d'une lecture ou écriture d'un emplacement mémoire *m* par la transaction *T, T* déterminera d'abord le compteur *i* correspondant à *m* dans la table hiérarchique. Si le $i^{ième}$ bit correspondant dans le masque de lecture de *T* est égale à *0*, il sera mis à *1* et la valeur actuelle du compteur sera enregistrée localement. Si l'accès mémoire est une écriture, on contrôle le $i^{ième}$ bit correspondant dans le masque d'écriture de *T*: si c'est un *0*, il sera mis à *1* et

le compteur partagé correspondant sera incrémenté. Si l'accès mémoire est une lecture et la transaction possède un ensemble de lecture, on crée une nouvelle entrée dans la $i^{ième}$ pièce correspondante dans l'ensemble de lecture.

Lors de la validation, on vérifie chaque compteur *i* dont le $i^{ième}$ bit correspondant dans le masque de lecture est mis à 1. Si (1) la valeur actuelle du compteur est égale à la valeur précédemment enregistrée, ou (2) la valeur actuelle du compteur est plus grande par un que la valeur enregistrée et le *i ième* bit correspondant dans le masque d'écriture est mis à 1. Alors dans les deux cas, on n'est sûr qu'aucune transaction concurrente n'ait verrouillé une adresse mappée au compteur *i* et nous pouvons ignorer la validation de la $i^{ième}$ pièce de l'ensemble de lecture. Et du fait, on a divisé les verrous de sorte que la validation puisse s'appliquer seulement à une partie des emplacements mémoire lus par une transaction [72]. Ce schéma de verrouillage hiérarchique fournit des gains de performance seulement si (1) la validation d'ensemble de lecture est coûteuse, c.à.d, les transactions de mise à jour lisent beaucoup d'emplacements mémoire, et (2) il y a peu d'écritures de la part des transactions concurrentes. Cependant, les auteurs croient que ces conditions sont produites assez souvent dans les applications réelles pour que cette optimisation soit utile [72].

4.18.2 Discussion

L'étude empirique [72] a prouvée que TinySTM est plus performant que TL2 car le TL2 souffre de verrouillage au moment de commit du fait que les transactions conflictuelles souvent perdent du temps en continuant l'exécution malgré qu'elles sont condamnées à être interrompues.

Le papier présente une stratégie d'adaptation automatique pendant l'exécution des paramètres influant le débit de système en cherchant la meilleure configuration pour la charge de travail actuelle car la performance d'un système STM dépend généralement de la charge de travail. L'évaluation démontre que démarrer à partir d'une configuration initiale et dynamiquement s'adapter permet d'atteindre rapidement la meilleure configuration et que le gain de performance est très important. Les paramètres influant la performance de TinySTM sont la taille de la table hiérarchique h car une transaction écrivant à beaucoup d'emplacements pourrait devoir incrémenter au plus *h* compteurs. Les opérations d'incrémentation atomiques sont coûteuses sur la plupart des architectures pour cela la taille de tableau hiérarchique doit être choisie avec soin car les plus grandes valeurs de *h* réduisent le temps système de validation mais peuvent exiger plus d'opérations atomiques [72].

Les deux autres paramètres influant la performance de TinySTM sont le nombre de verrous utilisés (taille de la table de verrou) et la fonction de hachage utilisée pour mapper les emplacements mémoire à la table de verrous. TinySTM décale à droite l'adresse et calcule le modulo de reste à la taille de la table de verrou. Le nombre de décalages à droite permet de contrôler combien d'adresses contiguës seront mappées au même verrou. L'accord et l'adaptation automatiques sont particulièrement importants étant donné qu'il n'y a aucune base ou benchmark sur lequel on peut savoir une charge de travail typique pour la mémoire transactionnelle. Ils nous permettent d'exploiter toutes les capacités des conceptions actuelles de mémoire transactionnelle pour les différentes charges de travail actuelles ou qui vont être identifiées.

4.19 Stretching Transactional Memory (SwissTM)

A. Dragojevi´c, R. Guerraoui et M. Kapalka ont présenté en 2009 le *SwissTM* qui est une bibliothèque STM implémentée en C++ basée sur le mot et sur les verrous. Leur but était de concevoir et d'implémenter un STM qui performe particulièrement bien avec des charges de travail transactionnelles complexes tout en gardant une bonne performance pour celles de petite taille et des structures de données simples. SwissTM est le premier STM qui utilise une stratégie d'invalidation mixte, qui n'a jamais été utilisée avec les STMs bloquants ni avec ceux basés sur le mot [17]. La stratégie d'invalidation mixte pour la détection de conflit consiste à détecter les conflits d'écriture/écriture tôt et de détecter les conflits de lecture/écriture en retard. C.à.d. il utilise une stratégie de détection de conflit optimiste (détection au moment de commit) pour les conflits lecture/écriture et une stratégie de détection de conflit pessimiste (détection lors de premier accès) pour les conflits écriture/écriture. Le Tableau 4.19.1 suivant montre les caractéristiques conceptuelles de base de SwissTM.

SwissTM Utilise un nouveau gestionnaire de conflit à deux phases qui assure le progrès de longues transactions sans causer aucun surcoût sur les courtes transactions [17]. L'évaluation de SwissTM a prouvée qu'il surpasse les implémentations des algorithmes STMs représentant l'état de l'art, à savoir RSTM, TL2 et TinySTM, dans les expériences faites sur les benchmarks STMBench7, STAMP, Lee-TM et red-black tree.

Tableau 4.19.1 : les caractéristiques de base de SwissTM.

SwissTM	
Synchronisation	Bloquante
Contrôle de Concurrence	Mixte
Granularité	Mot
Stratégie de Mise à jour	Différée
Stratégie de Détection de Conflit	Mixte
Stratégie de résolution de conflit	Gestionnaire de conflit Spécifique
Isolation	Faible
Transactions emboitées	Non Supportées

4.19.1 Détails d'implémentation

Le *SwissTM* est un STM basé sur les verrous, il utilise un verrouillage au moment de commit comme celui utilisé par TL2 qui est efficace uniquement pour les courtes transactions contrairement aux grandes transactions car il peut faire perdre éventuellement un grand travail lors de l'abort des grandes transactions à cause d'un conflit de type écriture/écriture. Pour cela, ce type des conflits (écriture/écriture) qui mène habituellement à aborter les transactions sont détectés tôt pour éviter de continuer l'exécution des transactions qui sont condamnées a être abortées et de ne pas gaspiller les ressources système ; les transactions verrouillent les objets dès le premier accès en écriture, ce qui permet de détecter les conflits d'écriture/écriture dès qu'ils

apparaissent. Cependant, les conflits de lecture/écriture sont détectés en retard pour être optimiste en permettant plus de concurrence entre les transactions. Ça est fait en utilisant les lectures invisibles et en permettant à des transactions de lire des objets acquis pour écriture. SwissTM utilise un schéma basé sur l'estampille pour réduire le coût de validation de l'ensemble de lectures invisible.

SwissTM utilise un compteur global (*commit-ts)* partagé entre toutes les transactions. Ce compteur sera incrémenté par chaque transaction qui réalise une écriture lors de chaque commit. Une transaction *T* possède un descripteur de transaction *tx* qui contient (entre d'autres données) : (1) le champ *Valid-ts* pour enregistrer la valeur de *commit-ts* lue au début ou à la validation ultérieure de *T* (2) les logs de lecture et les logs d'écriture de *T* (Figure 4.19.1).

SwissTM utilise deux types de verrous : un verrou de lecture *r-lock* et un verrou d'écriture *w-lock* qui sont stockés dans une table globale de verrou. Chaque mot mémoire *m* est mappé à une paire de verrous dans cette table (r-lock et w-lock). Un verrou *w* est un mot mémoire. Le verrou d'écriture est égal à *0* lorsqu'il est libre et contient un pointeur à l'entrée correspondante dans le log d'écriture de la transaction possédant le verrou une fois il est verrouillé. Cependant, le verrou de lecture contient la valeur *1* lorsqu'il est verrouillé pendant que le bit de poids faible est mis à *0* une fois il déverrouillé, et les autres bits enregistrent le numéro de version de l'emplacement mémoire correspondant à *w*. Le verrou *w-lock* du mot mémoire *m* est acquis par une transaction d'écriture *T* tôt dès la première référence à *m* par *T* pour empêcher d'autres transactions d'y écrire (à *m)*. Le verrou *r-lock* associé à *m* est acquis par *T* au moment de commit pour empêcher d'autres transactions de lire le mot *m* et par conséquence, d'observer des états incohérents des mots écrits par *T*. Quand le *r-Lock* est déverrouillé, il contient le numéro de version du *m*.

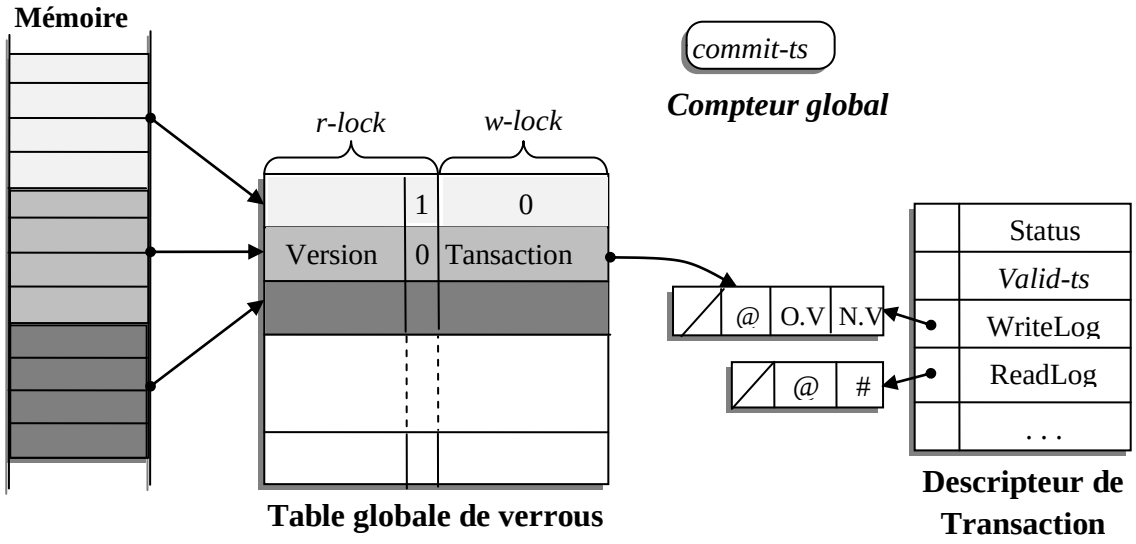

Figure 4.19.1: Structures de données de SwissTM.

A. Dragojevi´c et autres ont donné le pseudo code de son système STM sous forme d'un algorithme (Algorithme1 de la Figure 4.19.2). Lors de début de chaque transaction *T*, elle lit le compteur global *commit-ts* et enregistre sa valeur dans *tx.valid-ts* (ligne 2). Quand *T* lit un emplacement *addr*, elle doit lire d'abord la valeur du *w-lock* correspondant pour voir s'il est déjà détenu par une autre transaction. Si *T* est le propriétaire du *w-lock* de cet emplacement, alors *T* peut retourner immédiatement la dernière valeur écrite à *addr* (par *T*) depuis son log d'écriture

(ligne 6). Autrement, c.à.d. quand une autre transaction possède le *w-lock* ou quand le *w-lock* est déverrouillé (égale à 0), *T* lit la valeur du *r-lock*, puis la valeur de l'*addr*, et la valeur du *r-lock* une autre fois. La transaction *T* répète ces trois lectures jusqu'à ce que : (1) deux valeurs de r-*lock* soient identiques (signifiant que *T* a lu des valeurs consistantes de *r-lock* et d'*addr*), et (2) le *r-lock* soit déverrouillé (les lignes 8-15). C.à.d. Quand le *r-lock* est déverrouillé, il contient la version en cours *v* de l'*addr*. Si *v* est inférieure ou égale à l'estampille *tx.valid-ts* de validation de *T* (qui signifie que l'*addr* n'a pas été changé depuis la dernière validation ou le début du *T*), *T* retourne la valeur dans *addr* lue dans la ligne 18. Autrement, *T* revalide son ensemble de lecture. Si la revalidation ne réussit pas, *T* roule en arrière (ligne 17). Si elle réussit, l'opération de lecture retourne et *T* étend son estampille de validation *tx.valid-ts* à la valeur actuelle *commit-ts* (ligne 56).

Lors de chaque écriture à l'emplacement mémoire *addr* par la transaction *T*. *T* doit vérifier si elle est le propriétaire de verrou *w-lock* associé à *addr*. Si c'est le cas, *T* met à jour la valeur d'*addr* dans son log d'écriture et retourne (les lignes 21-23). Autrement, *T* essaie d'acquérir le *w-lock* en remplaçant de façon atomique (en utilisant l'opération *CAS*), son contenu (la valeur *0)* avec un pointeur vers l'entrée de log d'écriture du *T* qui contient la nouvelle valeur d'*addr* (ligne 29). Si le *CAS* ne réussit pas, *T* consulte le gestionnaire de conflit pour décider si elle doit faire un retour-arrière et recommencer ou elle doit attendre la terminaison du propriétaire actuel du verrou (ligne 26). Afin de garantir l'*opacité*, *T* doit revalider son ensemble de lecture si la version en cours de l'*addr* (contenu dans le *r-lock*) est plus grande que son estampille de validité *tx.valid-ts* (lignes 31-32). Pour le faire, *T* compare les versions de tous les emplacements mémoire lus jusqu'ici à leurs versions prises au moment où ils été initialement lus par T (lignes 51-52). Ces versions sont enregistrées dans le log de lecture de *T*. S'il y a une erreur dans la comparaison entre n'importe quels numéros de version, la validation s'échoue (ligne 52).

Dans la phase de commit, une transaction de lecture seule *T* peut commiter immédiatement, car son log de lecture est garanti pour être consistant (ligne 35). Une transaction *T* qui réalise des écritures doit d'abord verrouiller tous les verrous de lecture des emplacements mémoire qu'elle a écrit (ligne 36) pour empêcher les autres transactions de l'y accédé. Puis, *T* incrémente *commit-ts* (ligne 37) et revalide son log de lecture. Si la validation ne réussit pas, *T* fait un retour-arrière en libérant tous les verrous d'écriture acquis (linges 47–48), et par la suite recommence (lignes 38-41). Cependant, si la validation réussit, *T* parcourt son ensemble d'écriture, met à jour les valeurs de tous les emplacements mémoire écrits, et libère les verrous de lecture et d'écriture correspondants (lignes 42-45). En libérant les verrous de lecture, *T* écrit les nouvelles versions de ces emplacements en copiant la valeur de *commit-ts* dans ces verrous.

SwissTM implémente un gestionnaire de conflit à deux phases qui n'encourt aucun surcoût sur les transactions de lecture seule et les courtes transactions de lecture/écriture tout en favorisant le progrès des transactions qui ont réalisé un nombre significatif de commit. Pour ce type de transaction le schéma *timide* [17] est utilisé dont lequel les transactions sont abortées dès le premier conflit. Cependant, pour les transactions les plus complexes, *SwissTM* permute dynamiquement vers le mécanisme *avide* (*greedy)* qui implique plus de surcoût mais favorise ces transactions, en empêchant la famine. De plus, les transactions qui s'interrompent en raison d'un conflit de l'écriture/écriture fait un back-off pour une période proportionnelle au nombre de leurs arrêts successifs.

Algorithme 1: le Pseudo-code de SwissTM

```
1  function start(tx)
2      tx.valid-ts ← commit-ts;
3      cm-start(tx);
4  function read-word(tx, addr)
5      (r-lock,w-lock) ← map-addr-to-locks(addr);
6      if is-locked-by(w-lock, tx) then return get-value(w-lock, addr);
7      version ← read(r-lock);
8      while true do
9          if version = locked then
10             version ← read(r-lock);
11             continue;
12         value ← read(addr);
13         version2 ← read(r-lock);
14         if version = version2 then break;
15         version2 ← version;
16     add-to-read-log(tx, r-lock, version);
17     if version > tx.valid-ts and not extend(tx) then rollback(tx);
18     return value;
19 function write-word(tx, addr, value)
20     (r-lock,w-lock) ← map-addr-to-locks(addr);
21     if is-locked-by(w-lock, tx) then
22         update-log-entry(w-lock, addr, value);
23         return;
24     while true do
25         if is-locked(w-lock) then
26             if cm-should-abort(tx, w-lock) then rollback(tx);
27             else continue;
28         log-entry ← add-to-write-log(tx, w-lock, addr, value);
29         if compare&swap(w-lock, unlocked, log-entry) then
30             break;
31     if read(r-lock) > tx.valid-ts and not extend(tx) then
32         rollback(tx);
33     cm-on-write(tx);
```

```
34 function commit(tx)
35     if is-read-only(tx) then return;
36     for log-entry in tx.read-log do write(log-entry.r-lock, locked);
37     ts ← increment&get(commit-ts);
38     if ts > tx.valid-ts + 1 and not validate(tx) then
39         for log-entry in tx.read-log do
40             write(log-entry.r-lock, log-entry.version);
41         rollback(tx);
42     for log-entry in tx.write-log do
43         write(log-entry.addr, log-entry.value);
44         write(log-entry.r-lock, ts);
45         write(log-entry.w-lock, unlocked);
46 function rollback(tx)
47     for log-entry in tx.write-log do
48         write(log-entry.w-lock, unlocked);
49     cm-on-rollback(tx);
50 function validate(tx)
51     for log-entry in tx.read-log do
52         if log-entry.version ≠ read(log-entry.r-lock) and not
           is-locked-by(log-entry.r-lock, tx) then return false;
53     return true;
54 function extend(tx)
55     ts ← read(commit-ts);
56     if validate(tx) then tx.valid-ts ← ts; return true;
57     return false;
```

Algorithme 2: Le Pseudo-code de gestionnaire de conflit à deux phases ($\mathbf{W_n}$ est constant)

```
1 function cm-start(tx)
2     if not-restart(tx) then tx.cm-ts ← ∞ ;
3 function cm-on-write(tx)
4     if tx.cm-ts = ∞ and size(tx.write-log) = W_n then
      tx.cm-ts ← increment&get(greedy-ts) ;
5 function cm-should-abort(tx, w-lock)
6     if tx.cm-ts = ∞ then return true;
7     lock-owner = owner(w-lock);
8     if lock-owner.cm-ts < tx.cm-ts then return true;
9     else abort(lock-owner); return false;
```

```
10 function cm-on-rollback(tx)
11     wait-random(tx.succ-abort-count);
```

Figure 4.19.2: l'algorithme SwissTM.

4.19.2 Discussion

La comparaison de performance [17] du SwissTM avec TL2, TinySTM et RSTM a montré que SwissTM et plus performant que les autres STMs d'une manière significative pour des charges de travail mixtes caractérisées par des structures de données non-uniformes et dynamiques et des transactions de diverses tailles. La performance de SwissTM est expliquée par son adoption d'une compilation pertinente des choix de conception des systèmes de mémoire transactionnelle logicielle.

SwissTM est le premier STM qui adapte et utilise une stratégie de détection de conflit mixte avec une granularité au niveau mot que l'étude empirique a prouvé qu'elle est plus efficace que les deux autres stratégies (tôt et tard). De plus le schéma de gestion de conflit à deux phases a prouvé empiriquement qu'il est très efficace pour des charges de travail mixtes. Les expériences ont montré que le gestionnaire *avide* utilisé par SwissTM pour résoudre les conflits d'écriture/écriture est plus performant dans le cas des grandes transactions que le gestionnaire *Polka* mais moins performant dans le cas des courtes transactions que ce dernier (qui est considéré le plus efficace pour les courtes transactions [77]) car toutes les transactions incrémentent le compteur global partagé à leur début, ce qui entraîne beaucoup de coups manqués au niveau de cache et de manière significative dégrade l'exécution et l'évolutivité. Cependant, le schéma de gestion de conflit à deux phases a prouvé qu'il est plus performant que le gestionnaire avide dans le cas des charges mixtes parce qu'il permet aux courtes transactions et aux transactions de lecture seule de commiter sans incrémenter le compteur partagé employé par l'algorithme avide. Cependant, SwissTM ne supporte pas les transactions emboitées d'où l'étude empirique (sur un emboitement fermé) n'a pas permet d'observer aucun avantage clair de cette technique [17]. D'autres expériences pourraient être nécessaires dans cette direction.

Conclusion

La mémoire transactionnelle logicielle a connu dans ces dernières années une grande attention et les STMs ont été l'objet d'intenses recherches et du fait plusieurs implémentations des STMs ont été réalisées. Dans ce chapitre nous avons essayé de présenter ces implémentations en effectuant une étude détaillée de dix-neuf STMs représentant l'état de l'art de ce domaine tout en discutant leurs choix conceptuels et détails d'implémentation.

À un niveau élevé, il y a des attractions et des inconvénients apparentes pour chaque approche, il est clair que les implémentations modernes sont plus performantes que les premières implémentations, mais il est moins clair comment les performances relatives de ces implémentations sont comparées. Inévitablement, cela dépend beaucoup de la charge de travail et de la structure de la machine sous-jacente, par exemple, est que ce les threads partagent ou non un cache commun, si elle est ou non un système multi-cœurs, exactement quel modèle de mémoire est assuré par le matériel, quel modèle de programmation est fourni par le STM et les constructions de programmation construites sur lui. D'une façon générale Cette étude nous a permis de tirer un ensemble de résultats qui seront présentés dans le chapitre suivant.

5

Résultats et Discussion

Nous avons présenté dans les chapitres précédents une étude du domaine de mémoire transactionnelle notamment les implémentations logicielles. Le chapitre quatre présente une étude détaillée de dix-neuf implémentations en se basant sur leurs choix et caractéristiques conceptuels. Dans ce chapitre nous présenterons les résultats de cette étude tout en résumant les techniques et les choix d'implémentation des STMs. Nous présenterons aussi les défis et les problèmes ouverts dans le domaine des mémoires transactionnelles logicielles.

5.1 Résultats

Le Tableau 5.1 résume les systèmes de mémoire transactionnelle étudiés dans ce mémoire et leurs choix conceptuels. D'après l'étude faite, nous avons constaté que les implémentations de mémoire transactionnelle logicielle se diffèrent suivant les sept aspects suivants :

5.1.1 Technique de synchronisation

Les systèmes étudiés peuvent être classifiés suivant la technique de synchronisation utilisée en deux catégories ; les systèmes bloquants et les systèmes non bloquants. Nous avons remarqué que les implémentations les plus anciennes sont non bloquantes et ça est justifié par le fait que les développeurs ont évité d'utiliser les verrous vu les problèmes qu'ils posent (inter-blocage, inversion de priorité, etc.) (voir section 3.1.1.1). Généralement la liberté d'obstruction et le type dominant par rapport aux deux autres types d'implémentation non bloquante (liberté de verrou et liberté d'attente) et ça vu sa simplicité d'implémentation et performance par rapport aux autres types. Cependant, elle introduit le problème de livelook et d'inter-blocage, ces problèmes sont généralement prévenus en utilisant des rolls back et en faisant recours aux gestionnaires de conflit pour résoudre la contention entre les transactions.

Le deuxième type est les systèmes bloquants (à base de verrous). Ce type d'implémentation utilise des verrous gérés par le système pour assurer les accès exclusifs aux données. Robert Ennals dans son papier controversé [22] est le premier qui a réalisé une implémentation d'un STM bloquant, Ennals a montré que la liberté d'obstruction est une propriété non nécessaire pour un système de mémoire transactionnelle [23]. Par la suite la plupart des implémentations des STMs récents (TL2, MCRT, BSTM, TinySTM, SuissTM et d'autres) sont des systèmes bloquants. Ça est motivé par les études faites qui ont montré que les implémentations bloquantes sont plus performantes en comparaison avec les implémentations non bloquantes [16, 20,21] car elles éliminent les indirections dans l'adressage, les systèmes non bloquants ont besoin (obligatoirement) au moins d'un niveau d'indirection à partir des métadonnées pour accéder à la

version courante de la donnée. D'une façon générale la recherche dans les deux domaines c.à.d. bloquant et non bloquant est active pour développer des implémentations plus performantes.

5.1.2 Contrôle de concurrence

Le contrôle de concurrence est l'activité de coordonner les accès concurrents à un objet partagé [71] c.à.d. de contrôler l'ordre d'exécution relatif des opérations contradictoires des différents threads. Quand plusieurs transactions accèdent simultanément au même objet avec au mois un accès est une écriture un conflit surgisse. Suivant le moment de détection de conflit et de sa résolution, les implémentations des STMs peuvent être classifiées en deux catégories : (1) le contrôle d'accès optimiste et (2) le contrôle d'accès pessimiste.

Les STMs implémentant un contrôle d'accès pessimiste garantissent un accès exclusif à un objet à une seule transaction à la fois, et dès qu'un conflit surgisse, il sera détecté et résolu par le système. Cependant, dans les STMs implémentant un contrôle d'accès optimiste, plusieurs transactions peuvent accéder au même objet simultanément et lors du commit le système résout le conflit. La plupart des STMs étudiés dans ce mémoire utilisent un contrôle d'accès optimiste et ça pour garantir plus de concurrence en permettant à plusieurs transactions d'accéder simultanément aux données partagées. Cependant, le contrôle d'accès pessimiste est orienté performance car il résout le conflit dés qu'il surgisse et évite de perdre les ressources systèmes en effectuant des calculs qui seront perdus quand on aborte des transactions contradictoires. D'une autre part le contrôle d'accès optimiste peut permettre à certaines transactions de commiter contrairement au contrôle d'accès pessimiste notamment dans le cas des conflits de type lecture/écriture, si la transaction faisant la lecture de l'objet en conflit commite en premier lieu, les deux transactions conflictuelles peuvent commiter. Comme solution intelligente pour bénéficier des avantages des deux techniques, une troisième stratégie de contrôle de concurrence appelée la stratégie mixte est introduite, dans laquelle on utilise un contrôle d'accès pessimiste pour les conflits de type écriture/écriture et un contrôle d'accès optimiste pour les conflits de type lecture/écriture, c'est la stratégie adoptée par plusieurs implémentations récentes (par exemple : McRT-STM section 4.12, TinySTM section 4.18 et SuissTM section 4.19)

5.1.3 La Granularité

La granularité dans un STM définit l'espace mémoire sur lequel le système détecte les conflits d'accès. La plupart des implémentations étudiées dans le cadre de ce travail utilisent une granularité de niveau objet (exemples : OSTM, RSTM, BSTM, DracoSTM) ou de niveau mot (exemples : WSTM, HyTM, TinySTM). La granularité de niveau objet est la plus compréhensible de la part du programmeur, dans une telle granularité les métadonnées sont stockées avec les objets ce qui facilite leur accès. Cependant, dans les autres types de granularité (mot, bloc ou ligne de cache), les métadonnées sont stockées séparément dans des structures de données dédiées. D'une façon générale le choix de niveau de granularité doit être un compromis, car une granularité de petite taille fournit plus de partage mais elle nécessite plus d'espace mémoire pour sauvegarder les métadonnées, cependant une granularité de grande taille requiert moins d'espace mais peut empêcher la concurrence et engendrer de faux conflits.

5.1.4 Stratégie de mise à jour

La stratégie de mise à jour définit le moment où le système effectue les modifications incorporées dans la transaction. D'une façon générale il existe deux stratégies : une mise à jour directe est une mise à jour différée.

Dans une mise à jour directe, le système agit directement sur les données réelles et garde les donnés originales dans un log de défaire pour pouvoir les restituer si la transaction aborte. Cette stratégie est très efficace si les conflits sont rares, cependant si les conflits sont fréquents, cette stratégie peut affecter la performance globale du système vu le coût élevé pour aborter une transaction et toutes les transactions qui ont lu des données qui vont être roulées en arrières. C'est une technique qui est peu utilisée dans les STMs étudiés car elle est moins efficace dans le cas des charges de travails élevées où le taux d'abort est élevé.

L'autre stratégie est la mise à jour différée, dans cette technique chaque transaction à ses propres copies de données sur lesquelles elle travaille. Lors de commit, ces valeurs privées remplacent les valeurs originales, cependant si la transaction aborte, ces copiées privées sont tout simplement supprimées. C'est une technique qui est plus efficace si les conflits sont fréquents. Cette stratégie été très populaire dans les premières implémentions STMs, elle est utilisée par quasiment tous les STMs non bloquants.

Une troisième stratégie consiste à implémenter une mise à jour directe et une mise à jour différée au même temps et de basculer entre les deux suivant le taux des conflits observé. C'est la technique adoptée par PhTM étudié en section 4.15 et DracoSTM étudié en section 4.16. C'est une stratégie qui peut être utile si on peut adresser certains défis ; d'abord quand on doit basculer vers l'autre stratégie, ensuite comment faire le bascule vers l'autre stratégie ? Une solution consiste à suspendre toutes les transactions actives et de les ré-exécuter dans l'autre mode. Cette solution engendre une perte de travail, une autre solution consiste à attendre la terminaison de toutes les transactions actives avant de basculer, cette attente peut augmenter le temps de réponse du système et par conséquent réduire la performance globale du système.

5.1.5 Gestionnaire de conflit

Le gestionnaire de conflit est le module consulté par le système STM afin d'arbitrer la contention entre les transactions conflictuelles. Un bon gestionnaire de conflit est celui qui garantit la progression du système vers l'avant en essayant de fournir un bon débit c.à.d. en permettant de commiter le plus grand nombre possible des transactions. Les systèmes STMs étudiés dans le cadre de ce travail utilisent des différentes stratégies pour résoudre les conflits.

D'abord, la solution la plus simple consiste à aborter l'une des transactions conflictuelles. C'est la technique utilisée par la plupart des anciennes implémentations (DSTM, OSTM, ESTM, TL2) et compris certaines implémentations récentes (TinySTM). D'autres implémentations utilisent une autre technique appelée "aide", dans laquelle la transaction qui perd la contention à faveur d'une autre transaction essaie d'aider cette dernière pour compléter son exécution. C'est une technique qui a été très utilisée dans les premières implémentations (STM, WSTM), cette technique est rarement utilisée avec les nouvelles implémentations vu sa complexité comme elle peut introduire le problème d'aide récursive où la transaction aidante est aidée par une autre transaction.

L'étude faite dans ce mémoire montre que la plupart des implémentations récentes utilisent un gestionnaire de conflit et éventuellement des gestionnaires de conflit dynamiques qui incorporent plus d'une politique de gestion de conflit (PhTM, SuissTM, NZTM). Il existe plusieurs politiques de gestion de conflit, les plus importantes sont présentées dans la section 3.4.10. généralement il n'existe pas une politique qui est la meilleure pour toutes les situations, en d'autres termes une politique qui est puissante pour une situation peut être non pour une autre circonstance, donc il est très utile de continuer la recherche pour trouver des mécanismes qui permettent d'incorporer plus d'une politique de gestion de conflit dans un STM et d'utiliser la politique la plus adéquate pour chaque charge de travail.

Système ---------- **Année**	**Stratégie de Synchronisation**	**Contrôle de Concurrence**	**Granularité**	**Stratégie de mise à jour**	**Détection de Conflit**	**Résolution de Conflit**	**transactions Emboitées**
STM 1995	NB Liberté de verrou	Pessimiste	Mot	Directe	Tôt	Aide	Non Supportées
WSTM 2003	NB Liberté d'obstruction	Optimiste	Mot	Différée	Tard	Aide/Abort	Aplatie
DSTM 2003	NB Liberté de verrou	Optimiste	Objet	Différée	Tôt	Abort	Non Supportées
OSTM 2003	NB Liberté de verrou	Optimiste	Objet	Différée	Tard	Abort	Non Supportées
ESTM 2003	Bloquante	Optimiste-lecture & Pessimiste-écriture	Objet	Différée	Tôt	Abort	Non Supportées
TL2 2006	Bloquante	Optimiste	Objet /Mot ou bloc	Différée	Tôt ou Tard (Sélectif)	Abort	Non Supportées
DSTM2 2006	Liberté d'obstruction ou Verrouillage	Optimiste	Méthode	Différée	Tôt	Gestionnaire de Conflit	Non Supportées
RSTM 2006	NB Liberté d'obstruction	Optimiste	Objet	Différée	Tôt/ Tard (Sélectif)	Gestionnaire de Conflit	Aplaties
TbSTM 2006	NB Liberté d'obstruction	Optimiste	Objet/ Mot ou bloc	Différée	Tôt /Tard (Sélectif)	Gestionnaire de Conflit	Non Supportées
HybTM 2006	NB Liberté d'obstruction	Optimiste et Pessimiste	Objet et Ligne Cache	Différée et Directe	Tard /Tôt	Gestionnaire de Conflit	Supportées
HyTM 2006	NB Liberté d'obstruction	Optimiste	Mot	Différée	Tard	Gestionnaire de Conflit	Supportées
McRT-STM 2006	Bloquante	Optimiste et Pessimiste	Objet/ Mot	Directe	Tôt pour écritures/ tard pour lectures	Abort	Fermées
BSTM 2006	Bloquante	Optimiste et Pessimiste	Objet	Directe	Tôt pour écritures/ tard pour lectures	Abort	Fermées
NZTM 2007/2009	NB Liberté d'obstruction	Optimiste	Objet	Différée et Directe	Tard	Gestionnaire de Conflit	Non Supportées

Système ---------- **Année**	**Stratégie de Synchronisation**	**Contrôle de concurrence**	**Granularité**	**Stratégie de mise à jour**	**Détection de conflit**	**Résolution de Conflit**	**transactions emboitées**
PhTM 2007	Bloquante ou Non Bloquante	Optimiste et Pessimiste	Objet, Mot ou ligne de Cache	Différée ou Directe	Tard /Tôt	Gestionnaire de Conflit, Abort, Aide	Peuvent être Supportées
Draco-STM 2007	Bloquante	Optimiste	Objet	Différée ou Directe	Tard /Tôt	Abort	Fermées
NBSTM 2008	NB Liberté d'obstruction	Optimiste	Mot	Directe ou Différée	Tard	Vol	Non Supportées
TinySTM 2008	Bloquante	Optimiste et Pessimiste	Mot /bloc	Directe ou Différée	Tôt / tard	Abort	Non Supportées
SuissTM 2009	Bloquante	Optimiste et Pessimiste	Mot	Différée	Mixed	Gestionnaire de Conflit	Non Supportées

Tableau 5.1 : Comparaison des choix de conception des STMs

5.1.6 Isolation

L'isolation est la propriété qui garantit l'exécution parallèle de plusieurs transactions sans interférence ni chevauchement. En d'autres termes elle garantit une exécution indépendante pour chaque transaction. L'exécution d'une transaction avec isolation en parallèle doit fournir le même résultat obtenu si on l'exécute toute seule. Les opérations dans un système de mémoire transactionnelle peuvent être des opérations transactionnelles et d'autres non transactionnelles c.à.d. des opérations exécutées à l'intérieur des transactions et d'autres hors les transactions. Un STM fournit une *isolation forte* s'il garantit la cohérence des données que ce soit pour les accès transactionnels ou non transactionnels. Cependant, il fournit une *isolation faible* s'il garantit uniquement la cohérence entre les transactions. Tous les STMs étudiés dans ce mémoire ne garantissent que l'isolation faible car l'implémentation de l'isolation forte dans un STM et une tâche très compliquée. Généralement l'isolation forte est garantit par les implémentations de mémoire transactionnelle matérielle où le protocole de cohérence de cache peut détecter les accès transactionnels et non transactionnels conflictuels.

5.1.7 Transactions emboitées

La plupart des systèmes étudiés dans le cadre de ce travail supportent les transactions emboitées notamment les implémentations récentes. Une transaction emboitée est une transaction qui contient plusieurs sous transactions. L'avantage des transactions emboitées est de permettre la distribution des tâches entres toutes les transactions. De plus si une transaction s'échoue, les autres transactions peuvent continuer l'exécution ce qui permet une tolérance aux échecs. Les transactions emboitées aplaties peuvent menacer la propriété d'isolation d'un STM du fait l'abort de la transaction interne cause l'abort de la transaction externe. De même, une transaction ouverte peut menacer l'isolation car les données commitées par une transaction interne peuvent être visibles pour les autres transactions même avant le commit de la transaction père.

5.2 Défis et problèmes ouverts de mémoire transactionnelle

Malgré les recherches faites dans le domaine des mémoires transactionnelles logicielles dans presque deux décennies, plusieurs problèmes sont à adresser et plus de recherche est nécessaire dans ce sens pour remédier à certains problèmes et battre un ensemble des défis. Dans cette section nous allons identifier quelques problèmes et défis qui fait face les STMs.

5.2.1 Les appels systèmes et les entrés/sorties

Les opérations d'E/S, les actions irrévocables et la communication avec les entités qui sont hors le contrôle du système de STM peuvent être le défi majeur pour l'utilisation des MT. Plusieurs techniques ont été proposées pour rendre les E/S possibles dans les transactions ; une solution consiste à bufférisér les sorties et mettre en log les entrées et lors de la phase de commit, les sorties seront finalisées et les logs seront effacées. Cette solution peut être intéressante dans certains cas mais elle ne marche pas si une entrée atomique est nécessaire après une sortie, prenant l'exemple d'une transaction qui écrit sur une ligne de commande et l'utilisateur attend le résultat pour faire une entrée, et tant que la sortie est bufférisée l'utilisateur ne va pas produire l'entrée ce qui même au blocage de la transaction. D'autres solutions (par exemple [87]) ont introduit les notions des transactions irrévocables et inévitables, mais ces solutions ont des limitations sévères parce qu'elles exigent la sérialisation des transactions qui ont des effets non réversibles avec un verrou global ce qui peu affecter la performance globale du système.

5.2.2 L'intégration avec les codes existants

Plusieurs implémentations des systèmes de mémoire transactionnelle logicielle ont été proposées. Malgré que le nouveau code dans les applications multithreads soit écrit à partir de zéro, il ne peut pas être exécuté uniquement sur la MT, car le code doit interagir avec les protocoles de synchronisation enterrés dans les appels système, les bibliothèques et les codes existants. Ces codes s'appuient sur le verrouillage pour mettre en œuvre les structures de données ou pour réaliser proprement les opérations d'entrées/sorties et autres services système. Assurer une telle intégration est un grand défi pour la MT. Plus de recherche est nécessaire dans ce sens pour trouver des solutions et des techniques intéressantes pour assurer l'intégration et l'interopérabilité des STMs avec les codes existants.

5.2.3 Programmer avec les transactions

Rossbach et autres [75] ont essayé de répondre à la question : "Est-ce que réellement la programmation transactionnelle est plus facile ?". Pour cela ils ont réalisé une étude en examinant 147 étudiants de premier cycle. Le travail demandé avait été de fixer une série de défis de programmation en utilisant des verrous à grain fin, des moniteurs à grain grossier, et la Mémoire Transactionnelle. Dans chaque cas, le problème était de gérer l'accès aux différentes voies dans un champ de tir (un jeu). Les transactions ont été trouvées plus difficiles à utiliser que le verrouillage à grain grossier, mais elles étaient plus faciles que le verrouillage à grain fin. Cependant, les erreurs de synchronisation ont été réduites légèrement lors d'utilisation des transactions en comparaison avec le verrouillage à grains grossiers, mais d'une valeur significative en comparaison avec les verrous à grain fin. De même, Pankratius et autres ont étudié un groupe de 12 étudiants qui travaillent sur un moteur de recherche de bureau parallèle dans le langage C [68]. Les projets ont été construits à partir de zéro, en faisant l'indexation et des fonctions de recherche. Les étudiants ont d'abord été introduits à la programmation parallèle et le prototype de la MT. Trois paires utilisaient une combinaison des pthreads et les verrous, et trois paires utilisaient les pthreads et les blocs atomiques. Le temps de la meilleure implémentation à base de MT de l'indexeur était d'environ 30% que le temps de la meilleure

implémentation à base de verrou et plus rapide que celle des verrous sur 9 des 18 requêtes. Ces études ont montré que la MT a le potentiel d'être plus facile à utiliser mais d'abord il faut l'apprendre à utiliser. Alors un autre défi qui n'est pas moins important que les deux précédents est d'apprendre à programmer avec les transactions et d'enseigner la grande communauté des programmeurs, des étudiants et des professeurs cette nouvelle abstraction de programmation.

5.2.4 Les benchmarks de mémoire transactionnelle

La performance d'une implémentation de mémoire transactionnelle est mesurée en testant et en analysant cette implémentation sur des benchmarks. Cependant, les évaluations empiriques de ces implémentations souffrent de l'absence des benchmarks réalistes et même ceux existants (par exemple : TM Micro benchmarks [26], SPLASH-2 [88], STMBench7 [32], STAMP [66], Lee-TM [3], Haskell STM Benchmark suite [69], Worm Bench [89] et RMS-TM [13]) ne fournissent pas des charges de travail adéquates qui reflètent des charges réelles car la plus part de ces benchmarks se concentrent sur une seule structure de donnée simple (par exemple, listes, arbres rouge-noirs, skip-list) et du fait ils ne contiennent que des transactions effectuant uniquement une poignée des accès mémoires contrairement à une charge de travail réelle qui peut être plus complexe en travaillant sur plusieurs structures de données en même temps et du fait les transactions peuvent contenir des centaines ou même des milliers d'accès mémoires. Donc la mesure de performance d'un STM avec les benchmarks existants (simplifiés) peut être dans le meilleur des cas non informative et dans le pire des cas trompeuse, car elle peut orienter les chercheurs à essayer d'optimiser les aspects pertinents de leurs implémentations. Vue ce nombre réduit des benchmarks et leurs valeurs, il y a une grande nécessité pour développer d'autres benchmarks qui reflètent une utilisation réelle de la mémoire transactionnelle.

5.2.5 Comparaison de Performance

Afin de pouvoir comprendre les différences entre les différents choix de conception et d'implémentation des systèmes de mémoire transactionnelle logicielle, une comparaison de performance (quantitative) de ces implémentations est nécessaire. Tabba et autres dans [83] ont réalisé une comparaison de performance de leur système NZTM avec d'autres systèmes. A notre connaissance c'est l'unique papier dans ce sens, alors d'autres comparaisons sont nécessaires. Cependant, la comparaison de performance (l'évaluation empirique) des implémentations de mémoire transactionnelle logicielle est une tâche délicate et peut être même non significative pour deux raisons [32]. Premièrement, la comparaison directe est difficile pour des systèmes basés sur des langages de programmations différents ou exécutés sur des environnements d'exécutions différents. Deuxièmement, parce qu'il n'y a pas de benchmarks qui fournissent des charges de travail réelles pour les STMs comme il est discuté dans la section précédente. Alors les résultats obtenus sont moins significatifs.

5.3 Conclusion

Nous venons de présenter dans ce chapitre les résultats de notre étude faite sur les mémoires transactionnelles logicielles, nous avons énuméré les différents axes de conception des STMs tout en identifiant les différents choix existants et leurs avantages et inconvénients relatifs. Comme résultat on peut conclure qu'il n'existe pas de politique universellement reconnue supérieure pour tous les systèmes. Nous avons identifié par la suite quelques défis et problèmes qui fait face l'utilisation des mémoires transactionnelles, d'une façon générale, ces problèmes doivent être adressés pour que la mémoire transactionnelle devienne l'incontournable modèle de programmation pour le développement des applications parallèles.

6

Conclusion

Le domaine de mémoire transactionnelle est un sujet très populaire actuellement dans la communauté de recherché. L'essor multi-cœurs à mis la programmation parallèle en cause vu l'absence d'un modèle de programmation simple et efficace pour développer des applications parallèles qui peuvent profiter de la pleine puissance de calcule des multi-cœurs. Les mémoires transactionnelles est un nouveau modèle de programmation qui vise à améliorer la productivité des applications parallèles en écartant la grande partie de difficulté causée par le contrôle de concurrence du développeur des applications vers les concepteurs des systèmes. L'efficacité de cette nouvelle technique doit être prouvée par l'expérience. Durant presque deux décennies, les chercheurs ont essayé d'explorer cette technique en développant plusieurs implémentations de mémoire transactionnelle que ce soit logicielles, matérielles ou même hybrides.

Dans ce mémoire nous avons essayé de présenter cette nouvelle technique notamment les mémoires transactionnelles logicielles qui ont plus d'avantages en comparaison avec l'approche matérielle. Probablement l'avantage le plus significatif est que les STMs puissent être utilisés avec les machines actuelles et ne nécessitent aucun changement dans les infrastructures (par exemple : les caches et leurs protocoles de cohérence). Les différents concepts concernant le domaine de mémoire transactionnelle sont étudiés en détails, de plus un état de l'art de dix neuf implémentations des STMs est présenté tout en discutant leurs choix de conception et leurs limitations. Ce travail a été un effort qui vise à fournir une meilleure compréhension de domaine de mémoire transactionnelle logicielle.

Pour conclure, la mémoire transactionnelle est une technique prometteuse par rapport aux approches classiques à base de verrous qui peut simplifier le développement des applications parallèles. Il est encore difficile de savoir comment elle sera réussie. Le succès consiste à surmonter certaines difficultés et défis techniques importants. En commençant par fournir des systèmes de mémoire transactionnelle omniprésents, efficaces, performants et flexibles qui peuvent être intégrés dans les environnements d'exécution existants. Si ces difficultés peuvent être résolues bientôt, la mémoire transactionnelle va probablement devenir un pilier essentiel de la programmation parallèle.

Bibliographie

[1] Adve et Gharachorloo, Shared Memory Consistency Models: A Tutorial 1995. IEEE Computer, 29(12):66–76, December 1996.

[2] Bowen Alpern and Fred B. Schneider. Defining Liveness. Information Processing Letters, 21(4):181–185, October 1985.

[3] Ansari .M, Kotselidis .C, Jarvis .K, Luj´an .M, Kirkham .C, and Watson .I. Lee-tm: A non-trivial benchmark for transactional memory. In ICA3PP ’08, June 2008.

[4] Barnes .G. A method for implementing lock-free shared-data structures. In Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (Velen, Germany). SPAA '93. ACM, New York, NY, July 1993.

[5] Behrooz Parhami. Introduction to Parallel Processing Algorithms and Architectures. Kluwer Academic Publishers 2002.

[6] Bloom B.H. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13(7) (1970) 422–426

[7] Blundell .C, Lewis .E.C, Martin .M.M.K. Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters, pages 17-17.

[8] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL); in conjunction with the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA’05), 2005.

[9] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. IEEE Transactions on Computers, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

[10] Chandy, K. M. and Misra, J. The drinking philosophers problem. ACM Trans. Program. Lang. Syst. October 1984.

[11] Chung J. W, Minh C.C, McDonald A, Skare T, Chafi H, Carlstrom B. D, Kozyrakis C. and Olukotun K. Tradeoffs in Transactional Memory Virtualization.12th international conference on Architectural support for programming languages and operating systems, ACM Press, New York, USA, 2006.

[12] Damron P, Fedorova A, Lev Y, Luchangco V, Moir M and Nussbaum D. Hybrid transactional memory. In Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA. ASPLOS-XII. ACM, New York, NY. October, 2006.

[13] Derin H, Gokcen K, Pascal F, Vincent G. and Christof F. RMS-TM: a transactional memory benchmark for recognition, mining and synthesis applications. 4th ACM SIGPLAN Workshop on Transactional Computing TRANSACT 2009, Raleigh, North Carolina, USA. Feb 2009.

[14] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In

Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO), pages 21–33, March 2007.

[15] Dice D, and Shavit N. What Really Makes Transactions Fasler? Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), Ottawa, Canada, 2006 (http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf).

[16] Dice D, Shalev O and Shavit N. Transactional locking II. Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 204-213.

[17] Aleksandar, Rachid Guerraoui and Michał Kapałka. Stretching Transactional Memory. In PLDI 2009.

[18] Dragojevi´c A, Guerraoui R, Kapałka M. Dividing Transactional Memories by Zero. School of Computer and Communication Sciences, I&C, EPFL, 2008.

[19] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9):569, September 1965.

[20] Hesham El-Rewini, Mostafa Abd-El-Barr. Advanced computer architecture and parallel processing. Wiley-Interscience, 2005.

[21] Ennals J. R. Adaptive Evaluation of Non-Strict Programs. PhD thesis. July 2007.

[22] Robert Ennals. Efficient Software Transactional Memory, Technical Report Nr. IRC-TR-05-051 Intel Research Cambridge Tech Report, 2003.

[23] Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research, 2006

[24] Michael J. Flynn. Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21(9) :948–960, sep 1972.

[25] Fraser, K. and Harris, T. Concurrent programming without locks. ACM Trans. Comput. Syst. May 2007.

[26] Fraser, K. Practical Lock-Freedom. Ph.D. Thesis, King's College, University of Cambridge, pages 15-47, September 2003.

[27] Garcia-Molina, H., Ullman, J. D. and Widom, J. Database Systems: The Complete Book. Prentice Hall, page 13, 2002.

[28] Kourosh Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, Stanford University, Stanford, CA, 1995. Also appears as Technical Report CSLTR- 95-685, Computer Systems Laboratory, Stanford University, Stanford, CA, Dec 1995.

[29] A. Goscinski. Distributed Operating Systems: The Logical Design, Reading, MA, Addison-Wesley, 1991

[30] Gottschlich J. E. and Connors D. A. DracoSTM: a practical C++ approach to software transactional memory. In Proceedings of the 2007 Symposium on Library-Centric Software Design. LCSD '07. ACM, New York, NY, Montreal, Canada, Oct, 2007.

[31] Gray .J and Reuter .A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[32] Guerraoui .R, Kapalka .M., and Vitek J. Stmbench7: a benchmark for software transactional memory. SIGOPS ’07, 2007.

[33] R. Guerraoui. On the correctness of transactional memory. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 175–184, 2008.

[34] Hagit Attiya and Welch Jennifer. Distributed Computing. Fundamentals, Simulations and Advanced Topics. Wiley Interscience, 2nd Edition, 2004.

[35] Harris .T and Fraser .K. Language support for lightweight transactions. In Proc. 18th SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 03), Anaheim, CA, 2003, pages 288–402.

[36] Harris T. Exceptions and side-effects in atomic blocks. Sci. Comput. Program. December 2005.

[37] Harris T, Marlow S, Peyton-Jones S and Herlihy, M. Composable Memory Transactions. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA. PPoPP '05. ACM, New York, NY, June 2005.

[38] Harris T, Plesko M, Shinnar A, Tarditi D. Optimizing memory transactions. ACMSIGPLAN Conf on Programming Language Design and Implementation (PLDI 2006) Ottawa, ON: ACM. 2006:14–25

[39] B. He, W. N. Scherer, III and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In Proc. 12th Annu. IEEE Int. Conf. on High Performance Computing (HiPC), Goa, India, 2005 (http://www.cs.rochester.edu/scherer/papers/2005-HiPC-TPlocks.pdf).

[40] John L. Hennessy and David A. Patterson. Computer Architecture - A Quantitative Approach. Morgan Kaufmann, fourth edition, 2007.

[41] Herlihy M, Eliot J and Moss B. Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289-300, 1993.

[42] Herlihy, M., Luchangco, V., and Moir, M. A flexible framework for implementing software transactional memory. SIGPLAN Not. 2005.

[43] Herlihy M, Luchangco V, and Moir M. Obstruction-free synchronization: Double-ended queues as an example. In Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.

[44] Herlihy M, Luchangco V, Moir M, and Scherer W N. Software transactional memory for dynamic-sized data structures. In Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (Boston, Massachusetts). PODC '03. ACM, New York, NY, pages 1-2, 92-101, 2003.

[45] M. Herlihy. SXM software transactional memory package for C#. http://www.cs.brown.edu/~mph.

[46] Maurice Herlihy, Nir Shavit. The Art of Multiprocessor Programming. Elsevier (2008)

[46] Maurice Herlihy. A methodology for implementing highly concurrent data ob-jects. ACM Transactions on Programming 15(5):745–770, November 1993.Languages and

Systems (TOPLAS),

[47] Maurice Herlihy. Transactional Memory Today .Springer-Verlag Berlin Heidelberg 2010 Symposium on Principles and practice of parallel programming, pages 175–184, 2008.

[48] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, 1991.

[49] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst., 12(3):463–492, 1990

[50] C. A. R. Hoare. Monitors: an operating system structuring concept. Commun. ACM, 17(10):549–557, 1974.

[51] Hoare C. A. R. Towards a theory of parallel programming. In Operating Systems Techniques, vol. 9 of A.P.I.C. Studies in Data Processing, Academic Press, pages 61–71, 1972.

[52] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

[53] Kumar .S, Chu .M, Hughes .C. J, Kundu .P and Nguyen .A. 2006. Hybrid transactional memory. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, New York, USA. PPoPP '06. ACM, New York, NY, March 2006.

[54] Leslie Lamport. Concurrent reading and writing. Communications of the ACM, 20(11):806–811, November 1977.

[55] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, 28(9):690–691, September 1979.

[56] Leslie Lamport. Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, 3(2):125–143, March 1977.

[58] Leslie Lamport. The mutual exclusion problem: partII—statement and solutions. J. ACM, 33(2):327–348, 1986

[59] James R. Larus and Ravi Rajwar. Transactional Memory, ISBN-13: 9781598291254, pages, 2007

[60] Lev Y, Moir M and Nussbaum D. PhTM: Phased transactional memory. In Workshop on Transactional Computing (Transact), 2007.

[61] Lomet D. B. Process structuring, synchronization and recovery using atomic actions. Proceedings of an ACM Conference on Language Design for Reliable Software, Mar 1977, pages, 128–137.

[62] Marathe V. J and Scott M. L. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report Nr. TR 839, University of Rochester Computer Science Dept., 2004.

[63] Marathe V. J, Spear M. F, Heriot C, Acharya A, Eisenstat D, Scherer III, W. N and Scott, M. L. Lowering the Overhead of Software Transactional Memory. Dept. of Computer Science, Univ. of Rochester, March 2006.

[64] V. J. Marathe and M. Moir. Efficient Nonblocking Software Transactional Memory. Technical report, Forthcoming Technical Report, Sun Microsystems Laboratories.

[65] V.J. Marathe, W.N. Scherer III, and M.L. Scott. Adaptive Software Transactional Memory. Proc. Of the 19th Intl. Symp. on Distributed Computing, Cracow, Poland, Sept, 2005.

[66] Minh .C.C, Chung .J, Kozyrakis .C and Olukotun .K. Stamp: Stanford transactional applications for multi-processing. In IISWC ’08, Sep 2008.

[67] Moravan .M. J, Bobba .J, Moore .K.E, Yen. L, Hill .M. D., Liblit .B, Swift. M. M., and Wood .D. A. Supporting nested transactional memory in logTM. SIGPLAN Not., Nov 2006, 359-370.

[68] Victor Pankratius, Ali-RezaAdl-Tabatabai and FrankOtto. Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, IPD, University of Karlsruhe, Germany, September 2009.

[69] Perfumo .C, S¨onmez .N, Stipic .S, Unsal .O, Cristal .A, Harris. T, and Valero .M. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In CF ’08, 2008.

[70] Rajwar R. and Goodman J. R. Speculative lock elision: enabling highly concurrent multithreaded execution. In Proceedings of the 34th Annual ACM/IEEE international Symposium on Microarchitecture, Austin, Texas. International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, December 2001.

[71] R. Ramakrishnan and J. Gehrke, Database Management Systems, Second Edition, McGraw-Hill (2000).

[72] T. Reigel, P. Felber and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In PPoPP, 2008.

[73] Riegel T, Felber P, Fetzer C. A Lazy Snapshot Algorithm with Eager Validation. In: 20th International Symposium on Distributed Computing (DISC), 2006.

[74] Riegel T, Fetzer C and Felber P. Time-based transactional memory with scalable time bases. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures San Diego, California, USA,. SPAA '07. ACM, New York, NY, June 2007.

[75] Christopher Rossbach,Owen Hofmann, and EmmettWitchel. Is transactional memory programming actually easier? In PPoPP ’10: Proc. 15th ACMSIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 47–56, January 2010. Earlier version presented at TRANSACT ’09.

[76] Saha B, Adl-Tabatabai A, Hudson R. L, Minh C, and Hertzberg B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '06. ACM, New York, NY. New York, New York, USA, March 2006.

[77] Scherer W. N and Scott M. L. Advanced contention management for dynamic software transactional memory. In Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (Las Vegas, NV, USA, July 17 - 20, 2005). PODC '05. ACM, New York, NY, July 2005.

[78] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems, 10(2):282–312, April 1988.

[79] Nir Shavit and Dan Touitou. Software transactional memory. In PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pages 204–213, New York, NY, USA, 1995. ACM.

[80] M. F. Spear, V. J.Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In DISC, 2006.

[81] Spear M. F, Michael M. M and Scott M. L. Inevitability Mechanisms for Software Transactional Memory. In Proc. of the 3rd ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, UT, February 2008.

[82] Tabba F, Wang C, Goodman J. R and Moir M. NZTM: Non-blocking Zero-Indirection Transactional Memory. In the 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007.

[83] Tabba F, Wang C, Goodman J. R and Moir M. NZTM: Non-blocking Zero-Indirection Transactional Memory. In the 21st ACM Symposium on Parallelism in Algorithms and Architectures, 2009, Canada.

[84] Thomasian A. Concurrency control: methods, performance, and analysis. ACM, Comput. Surv. pages 70–119, 1998.

[85] Virendra J. Marathe, Mark Moir. Toward High Performance Nonblocking Software Transactional Memory. In PPoPP'08 February 20–23, 2008, Salt Lake City, Utah, USA.

[86] Weikum, G., and Vossen, G. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. Morgan Kaufmann, 2001.

[87] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pages 285–296, New York, NY, USA, 2008. ACM.

[88] Woo, C., S., Ohara, M., Torrie, E., Singh, P.J., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In ISCA '95, 1995.

[89] Zyulkyarov, F., Cvijic, S., Unsal, O., Cristal, A., Ayguade, E., Harris, T., and Valero, M. Wormbench - a configurable workload for evaluating transactional memory systems. In MEDEA '08, 2008.