Vers une mémoire transactionnelle temps réel

Toufik Sarni

UNIVERSITÉ DE NANTES

FACULTÉ DES SCIENCES ET DES TECHNIQUES

———

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DE MATHÉMATIQUES

Année 2012

Vers une Mémoire Transactionnelle Temps Réel

———

THÈSE DE DOCTORAT
Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Toufik SARNI

Le 16 Octobre 2012, devant le jury ci-dessous

Président		
Rapporteurs	Luc BOUGANIM	Directeur de recherche - INRIA Rocquencourt
	Michel BANATRE	Directeur de recherche - INRIA Rennes
Examinateurs	Gaël THOMAS	Maître de conférences - Université Pierre et Marie Currie
	Maryline CHETTO	Prof. des universités - Université de Nantes

Directeur de thèse :	Patrick VALDURIEZ	Directeur de recherche - INRIA, LIRMM, Montpellier
Co-encadrant de thèse :	Audrey QUEUDET	Maître de conférences - Université de Nantes

Table des matières

Liste des figures

Liste des tableaux

Liste des Algorithmes

Introduction

Cette décennie est marquée par le bouleversement technologique qu'a connu l'industrie des processeurs. Face au problème de la dissipation de la chaleur dans le monocœur due à la croissance continuelle du nombre de transistors, et aux contraintes d'espace notamment dans le milieu embarqué, le multicœur est apparu comme étant la nouvelle tendance pour pallier ces difficultés. Cette technique de fabrication consiste à mettre plusieurs processeurs sur un même support doté d'une ou plusieurs mémoires communes. Le but est d'égaliser en puissance de calcul un monocœur qui aurait la capacité de tout l'ensemble.

Cependant, la transition vers le multicœur n'est pas si aisée car le domaine logiciel associé au multicœur reste très rattaché à la culture de la programmation séquentielle. La notion de parallélisme implique une logique relativement plus complexe. Dans la littérature, peu de modèles et d'outils sont proposés pour réduire cette complexité. Cette dernière se manifeste notamment dans la gestion de la concurrence aux données partagées pour lesquelles des solutions à base de verrouillage sont communément utilisées pour assurer l'intégrité du système. Bien que les mécanismes à base de verrous soient simples, ils peuvent cependant engendrer des problèmes sévères comme les interblocages et l'inversion de priorité. En outre, lorsqu'un processeur accède à une ressource partagée, les verrous bloquent l'accès à tous les autres processeurs. En multicœur, ce blocage systématique peut engendrer une baisse significative des performances dans la mesure où le parallélisme du multicœur se retrouve faiblement exploité.

Par ailleurs, on peut s'interroger sur le fait que les politiques d'ordonnancement des processus conçues pour les systèmes multiprocesseurs soient directement utilisables ou non pour les systèmes multicœurs. Etant donnés les changements d'architectures apportés aux processeurs multicœurs, notamment au niveau des mémoires communes, il convient d'examiner si toutes les politiques d'ordonnancement sont aussi performantes les unes que les autres.

Plus particulièrement, dans les systèmes temps réel, les politiques d'ordonnancement des tâches doivent prendre en compte le facteur temps, en respectant des contraintes appelées *échéances*. Ces ordonnanceurs sont élaborés à partir d'un modèle de tâches dont les caractéristiques sont bien définies (Liu et Layland 1973). Les tâches temps réel accèdent aux ressources partagées à travers des protocoles de gestion de concurrence. La plupart de ces protocoles sont basés sur les verrous et permettent d'éviter les problèmes tels que les interblocages et l'inversion de priorité. Toutefois, le problème de l'exploitation du parallélisme reste posé pour ces systèmes.

Les systèmes de gestion de bases de données (SGBD) ont su, quant à

eux, contrer le problème de concurrence d'accès aux données et ce, aussi bien dans un environnement séquentiel que parallèle. Les SGBDs s'appuient sur la notion de *transaction* dotée des propriétés ACID (Atomicité, Consistance, Isolation, et Durabilité). Ces propriétés garantissent l'intégrité en lecture et écriture pour les données partagées. Du point de vue d'un programmeur, l'atomicité impose que l'ensemble des instructions appartenant à une transaction donnée soit exécuté ou rejeté en bloc. Avec la propriété d'isolation, les résultats produits par une transaction ne sont pas influencés par d'autres transactions s'exécutant en parallèle.

Le transactionnel est repris dans les systèmes multicœurs sous le nom de *Mémoire Transactionnelle* (Herlihy et Moss 1993, Shavit et Touitou 1995) avec uniquement les propriétés ACI. Il en ressort que la transaction est un concept "plus fort" que les verrous pour d'une part réduire la complexité vis-à-vis du programmeur et d'autre part, pour s'affranchir des problèmes liés aux verrous et ce, grâce aux propriétés ACI.
Des conflits d'accès aux ressources partagées peuvent survenir entre les transactions concurrentes. Les mémoires transactionnelles intègrent des ordonnanceurs qui appliquent une politique de gestion de conflits entre transactions (Scherer III et Scott 2004). La décision de l'ordonnanceur est soit de réexécuter (*retry*) la transaction, soit de la valider (*commit*).

A la fin des années 80, les SGBDs temps réel (SGBDTR) sont apparus pour reprendre les fonctionnalités des SGBDs classiques, avec en plus, le but de garantir des contraintes de temps imposées aux transactions (Abbott et Garcia-Molina 1988). Les travaux de recherches menés sur le transactionnel temps réel sont généralement basés sur des plateformes monocœurs (centralisées ou distribuées), et l'aspect multiprocesseur est rarement considéré.

Dans ce travail de thèse, nous proposons d'étudier l'adaptation des mémoires transactionnelles aux systèmes temps réel multicœurs. Jusqu'à maintenant, l'ordonnancement de transactions temps réel au sein d'une mémoire transactionnelle n'a pas été étudié. Les résultats principaux de cette thèse ont conduit à plusieurs publications (Sarni et al. 2009a;b;c, Queudet et al. 2012).

Le manuscrit est organisé en trois parties (état de l'art, STM pour les systèmes temps réel *soft*, et extension aux contraintes *firm*), et termine par une conclusion générale. Chaque partie contient un ensemble de chapitres décrits ci-après.

Le premier chapitre présente brièvement les systèmes temps réel multiprocesseurs. Il aborde les différents principes des ordonnanceurs temps réel multiprocesseurs ainsi que les types de synchronisation utilisés dans de tels systèmes.

Le deuxième chapitre est une introduction aux SGBDs temps réel. Il rappelle les notions de base du transactionnel et établit une taxonomie des protocoles de gestion de concurrence temps réel.

Le troisième chapitre présente la notion de mémoire transactionnelle. Sans être exhaustif, ce chapitre introduit la terminologie utilisée pour les

mémoires transactionnelles ainsi que quelques critères de classification. Le quatrième chapitre présente nos contributions au travers d'une étude expérimentale et comparative visant à statuer sur l'adéquation des mémoires transactionnelles aux systèmes temps réel *soft*.

Le cinquième chapitre introduit un nouveau modèle transactionnel temps réel pour les mémoires transactionnelles. Il décrit la conception et l'implémentation d'une mémoire transactionnelle logicielle temps réel nommée RT-STM.

Le sixième chapitre présente de nouveaux protocoles de synchronisation qui permettent d'affiner et d'améliorer la RT-STM du point de vue temporel. Plus particulièrement, ces protocoles permettent de prioriser les accès aux ressources partagées en fonction de l'urgence des transactions.

Le septième chapitre montre comment étendre notre RT-STM à un environnement temps réel *firm* en proposant quelques pistes d'adaptation permettant de garantir un certain niveau de qualité de service (QoS) vis-à-vis des accès aux ressources partagées.

En conclusion, nous présentons quelques perspectives de travail, en proposant notamment des pistes pour les principaux défis à relever dans de futurs travaux de recherche s'inscrivant dans la continuité de la thèse.

Première partie

État de l'art

Les systèmes temps réel multiprocesseur

1

Sommaire

1.1 Introduction aux systèmes temps réel

Pour introduire la notion de temps réel, on se réfère généralement à la définition intuitive suivante de Stankovic (1988) :*"la correction du système ne dépend pas seulement des résultats logiques des traitements, mais dépend aussi de la date à laquelle ces résultats sont produits"*. Ces dates à respecter appelées *échéances* sont fondamentales pour un système temps réel, et leur dépassement peut induire dans certains cas une défaillance du système pouvant être grave à la fois pour le système et l'environnement contrôlé. Le respect des échéances ne sous-entend pas la rapidité des traitements, puisque ces dates peuvent être, selon le système, de l'ordre de la milliseconde ou du mois, voire des années.
Les systèmes temps réel sont en général constitués de deux parties : le système contrôlé souvent appelé *procédé* ou encore *partie opérative*, sur lequel on exerce le contrôle, et la *partie commande* qui interagit avec le premier. Le système de contrôle prend connaissance de l'état du procédé par le biais des capteurs, lui permettant de réaliser un traitement spécifique et d'agir par conséquent sur le procédé par le biais des actionneurs.
Le système de contrôle est composé d'une partie logicielle et d'une partie matérielle. Le support d'exécution de cette dernière peut être :

- *monocœur* [1] : tous les programmes de la partie logicielle s'exécutent en parallélisme apparent.
- *multicœur* : les programmes de la partie logicielle sont répartis sur plusieurs cœurs, partageant au moins une mémoire commune.
- *réparti* : les programmes sont répartis sur plusieurs processeurs dépourvus de mémoire commune et d'horloge commune. Ils sont reliés par un médium de communication via lequel ils peuvent communiquer par échanges de messages.

1.1.1 Classification

Les systèmes temps réel sont classés selon le niveau de criticité de leurs contraintes de temps. On parle alors de systèmes (Liu 2000) :

- *temps réel dur ou à contraintes strictes* (*hard* en anglais) pour lesquels le dépassement des échéances n'est pas toléré, sous peine de conséquences catastrophiques sur le système lui-même ou son environnement (ex : le système de contrôle du train d'atterrissage d'un avion),
- *temps réel souple ou à contraintes relatives* (*soft* en anglais) pour lesquels le dépassement des échéances peut provoquer des dégradations acceptables (ex : le système informatique d'une borne interactive multiservice),
- *temps réel ferme* (*firm* en anglais). Ce type de système se situe à la frontière entre les deux contraintes précédentes. Un système temps réel ferme tolère le dépassement des échéances mais à la différence des systèmes à contraintes *soft*, ces dépassements sont quantifiés (ex : le système de lecture d'une vidéo, normalement cadencé à 25 images par seconde peut tolérer une dégradation passagère à 20 images par seconde). De plus, un résultat produit après échéance n'est pas pris en compte.

1. Dans l'ensemble du document, nous confondrons les deux mots : *processeur* et *cœur*

1.1.2 Les tâches temps réel

Notion de tâche temps réel

Dans la littérature, le terme *tâche* se réfère à une suite cohérente d'opérations afin d'exécuter un programme. Ce terme est souvent préféré au terme *processus* pour signifier la périodicité ou la cyclicité d'exécution. Le terme processus est attaché quant à lui, plutôt au sens de l'unicité de la demande d'exécution. Au moment de son exécution, l'entité tâche est composée d'un ensemble de *travaux*. Dans un environnement multicœur, ces travaux peuvent être attribués aux différents cœurs en parallèle, mais en règle générale, le parallélisme au sein même d'un travail n'est pas admis. Outre les contraintes de temps que nous allons définir par la suite, les tâches peuvent avoir d'autres contraintes à respecter parmi lesquelles (Silly-Chetto 1993) :

- *les contraintes de ressources*. Les ressources partagées autres que les processeurs ne sont pas forcément disponibles au moment de l'activation des tâches, et leur accès doit être protégé de manière à assurer leur cohérence (ex : variables partagées en mémoire).
- *les contraintes de synchronisation* qui imposent un certain ordre d'exécution formulé par des relations de précédence. Lorsqu'aucune relation de précédence n'est appliquée à une tâche, celle-ci est dite *indépendante*.
- *les contraintes d'exécution*. Deux modes sont considérés : *préemptif* et *non-préemptif*. Une tâche est dite préemptible, si elle peut être interrompue à un instant quelconque et être reprise ultérieurement. Dans le cas non-préemptif, la tâche s'exécute complètement de manière ininterrompue à partir du moment où elle obtient le processeur.
- *les contraintes de placement* qui portent sur l'identité du (ou des) processeur(s) d'un système multicœur sur lequel une tâche est autorisée à s'exécuter.

Dans la suite de ce chapitre, nous discuterons plus précisément de ces contraintes de placement.

Modélisation des tâches temps réel

Dans ce qui suit nous considérons une tâche τ_i appartenant au système de tâches τ auquel l'on associe l'ensemble de travaux J^i s'exécutant sur l'ensemble de processeurs M.

Définition 1.1 **La tâche périodique**

D'un point de vue temporel, une tâche périodique τ_i est caractérisée par le tuple (r_i, C_i, P_i, D_i) où :

- *r_i représente la date de réveil de la tâche, c'est-à-dire l'instant du premier travail de τ_i.*
- *C_i dénote la durée d'exécution au pire cas, à savoir une limite supérieure sur le temps d'exécution de chaque travail de la tâche τ_i.*
- *P_i correspond à la période d'activation, c'est-à-dire à la durée qui sépare deux arrivées successives de travaux pour τ_i.*

– *un délai critique D_i qui représente l'échéance relative de la tâche, c'est-à-dire l'intervalle de temps dont un travail dispose pour s'exécuter.*

La tâche périodique τ_i définit un nombre infini de travaux ayant tous le même temps d'exécution. Chaque travail $j_n \in J^i$ qui se réveille à l'instant $r_i + (n-1)P_i$, doit se terminer avant sa date d'échéance absolue $d_i = r_i + (n-1)P_i + D_i$, $\forall n \in \mathbb{N}^*$ (voir Figure 1.1).

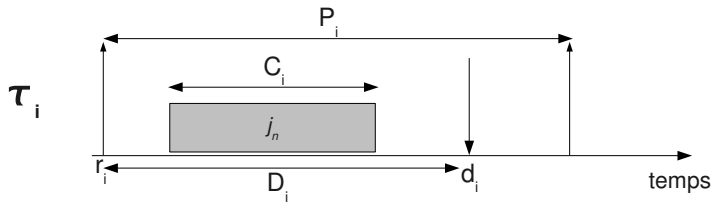


FIGURE 1.1 – *Modélisation d'une tâche temps réel périodique*

Cas particuliers de systèmes périodiques

Dans le cas où $P_i = D_i$ alors τ_i est dite *à échéance sur requête*. De plus, si toutes les tâches périodiques de τ ont la même date de réveil initiale ($\forall i, j \in \mathbb{N}\ r_j = r_i$) alors cette configuration de tâches est dite *synchrone*, sinon elle est dite *asynchrone*.

Définition 1.2 **La tâche sporadique**

Une tâche sporadique τ_i est caractérisée par le triplet (r_i, C_i, P_i, D_i) où :
- *r_i représente l'instant du premier travail de τ_i.*
- *C_i dénote la durée d'exécution au pire cas, à savoir une limite supérieure sur le temps d'exécution de chaque travail de la tâche τ_i.*
- *P_i correspond à la durée minimale qui sépare deux arrivées successives de travaux pour τ_i.*
- *un délai critique D_i qui représente l'échéance relative de la tâche, c'est-à-dire l'intervalle de temps dont un travail dispose pour s'exécuter.*

Le paramètre P_i représente le délai minimum entre deux travaux $j_n \wedge j_{n+1} \in J^i$. De ce fait, la tâche sporadique représente un cas plus général que la tâche périodique.

Définition 1.3 **La tâche apériodique**

Une tâche apériodique est caractérisée par le couple (r_i, C_i, D_i) où :
- *r_i représente l'instant du premier travail de τ_i.*

– *C_i dénote la durée d'exécution au pire cas, à savoir une limite supérieure sur le temps d'exécution de chaque travail de la tâche τ_i.*
– *un délai critique D_i qui représente l'échéance relative de la tâche, c'est-à-dire l'intervalle de temps dont un travail dispose pour s'exécuter.*

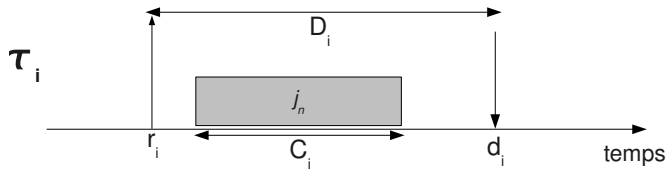


FIGURE 1.2 – *Modélisation d'une tâche temps réel apériodique*

La date d'arrivée a_i dans le système d'une tâche apériodique n'est pas connue *a priori*. Une fois prise en compte par le système, la tâche apériodique commence son exécution à une date r_i tel que $r_i \geq a_i$ et doit avoir terminé son exécution avant sa date d'échéance $d_i = r_i + D_i$ (voir Figure 1.2).

Facteur d'utilisation du processeur

Pour une tâche τ_i, le rapport $u_i^k = \frac{C_i}{P_i}$ est défini comme étant son facteur d'utilisation du cœur $k \in M$. Un cœur k, pouvant être utilisé par l'ensemble du système de tâches τ, a donc un facteur d'utilisation $U = \sum_i u_i^k$

1.1.3 Taxonomie des plateformes multiprocesseurs

Dans Funk et al. (2001), on distingue trois plateformes multiprocesseurs, de la plus générale à la plus spécifique :

– *Processeurs identiques* : il s'agit d'une plateforme multiprocesseur, composée de processeurs ayant les mêmes capacités de calcul,
– *Processeurs uniformes* : cette plateforme est en revanche composée de processeurs ayant des capacités de calcul différentes. Autrement dit, un travail qui s'exécute sur un processeur de capacité de calcul s pour une durée d'exécution de t unités de temps, s'effectuera en $s \times t$ unités de temps,
– *Processeurs indépendants* : cette plateforme est définie par un taux d'exécution u_i^k associé à chaque couple donné ($j_n \in J^i$, $k \in M$). Le travail j_n réalise $u_i^k \times t$ unités de travail lorsqu'il est exécuté sur k pendant t unités de temps. D'un travail à un autre, la vitesse de progression peut alors varier.

1.2 Ordonnancement temps réel multiprocesseur

L'ordonnanceur des tâches représente une partie essentielle du système de contrôle. Il considère l'entité tâche τ_i ainsi que tous les travaux J^i de τ_i, mais sans pour autant considérer les traitements effectués au sein de ces travaux. Il arrive toutefois que l'ordonnanceur prenne en charge certains traitements effectués par les travaux, et ce, à des fins de synchronisation entre tâches. Les objectifs d'un ordonnanceur peuvent différer (ex : optimisation de l'utilisation des ressources, minimisation du temps de réponse, etc.) mais dans tous les cas, l'ordonnanceur doit veiller au respect des échéances des tâches.

1.2.1 Définitions

Nous présentons ci-dessous quelques définitions communément utilisées dans la littérature aussi bien pour les environnements mono et multiprocesseurs.

Définition 1.4 Faisabilité

Un système de tâches τ est dit faisable, s'il existe une configuration d'ordonnancement pour laquelle toutes les tâches s'exécutant sur une plateforme donnée, respectent bien leurs échéances.

Définition 1.5 Ordonnançabilité

Un système de tâches τ est dit ordonnançable par A, s'il existe un algorithme A capable d'ordonnancer toutes les tâches τ_i dans le respect de leurs échéances.

Définition 1.6 Optimalité

L'algorithme d'ordonnancement A est dit optimal, s'il est capable d'ordonnancer tout système de tâches faisable.

Borne supérieure d'ordonnançabilité

C'est une métrique souvent utilisée, pour comparer les algorithmes d'ordonnancement en termes de facteur d'utilisation des processeurs. Si $U_A(M, \alpha)$ est la borne d'utilisation du processeur d'un système de tâches τ_i tel que $u_i^{max,M} \leq \alpha$ pour un algorithme d'ordonnancement A, alors sur l'ensemble M des processeurs, A peut ordonnancer τ avec un taux maximum d'utilisation $\sum_i u_i^k \leq U_A(M, \alpha)$. Dans le cas où au moins un système de tâches a une utilisation processeur $\sum_i u_i^k = U_A(M, \alpha)$ et que $u_i^{max,k} = \alpha$ alors $U_A(M, \alpha)$ est une *borne minimax d'utilisation* de A pour M et $u_i^{max,k}$. Si en revanche, aucun système de tâches n'excède $U_A(M, \alpha)$ quand $u_i^{max,k} = \alpha$, $U_A(M, \alpha)$ est dite alors *borne optimale d'utilisation*. Notons que le minimax est aussi utilisé pour exprimer le pire cas d'utilisation du processeur (Oh et Baker 1998). À titre illustratif, la borne supérieure

d'ordonnançabilité de l'algorithme du Rate Monotonic pour N tâches, est $U_{RM} = N(2^{\frac{1}{N}} - 1)$ et converge vers $\ln 2$ quand $N \to +\infty$.

1.2.2 Types d'ordonnancements

Ordonnancement hors-ligne / en-ligne

Quelle que soit la plateforme considérée (monocœur/multicœur), lorsque l'ordonnancement est établi avant le lancement des tâches et que leur comportement est bien connu pour l'ensemble de leurs actions, alors l'ordonnancement est dit *hors-ligne*. L'ordonnancement est en revanche dit *en-ligne* si la décision d'ordonnancement est prise pendant l'exécution.

Ordonnancement préemptif / non-préemptif

Dans un ordonnancement préemptif, l'exécution d'un travail peut être interrompue à un instant arbitraire puis reprise à un instant ultérieur. L'ordonnancement non-préemptif quant à lui, garantit à tout travail l'exclusivité du processeur durant toute sa durée d'exécution.

Ordonnancement à priorités fixes

Un algorithme d'ordonnancement à priorité fixe au niveau des tâches, attribue des priorités statiques aux tâches. Ces priorités sont directement héritées par les travaux de ces tâches. Tous les travaux d'une même tâche auront toujours la même priorité. Parmi les algorithmes d'ordonnancement à priorité fixe, citons RM (*Rate Monotonic*) (Liu et Layland 1973) et DM (*Deadline Monotonic*) (Audsley 1990).

Ordonnancement à dynamicité restreinte

Ces algorithmes d'ordonnancement sont à priorité fixe au niveau des travaux. Ils permettent, lors de l'exécution des tâches, d'avoir des priorités différentes entre les différents travaux d'une tâche. Pendant la vie d'un travail donné, sa priorité reste cependant identique. L'algorithme d'ordonnancement EDF (*Earliest Deadline First*) (Dertouzos 1974) fait partie de cette catégorie.

Ordonnancement à priorités dynamiques

Les algorithmes à priorité dynamique ne mettent aucune restriction sur les priorités assignées aux travaux. À chaque instant, la priorité d'un travail est recalculée. L'algorithme LLF (*Least Laxity First*) (Mok et Dertouzos 1978) est un exemple d'ordonnancement à priorités dynamiques.

1.2.3 Types d'approches

Approches par partitionnement

Les tâches sont affectées d'une manière statique sur les différents processeurs. Ainsi, tous les travaux d'une tâche donnée, sont toujours exécutés

sur le même cœur durant toute la vie de la tâche. Chaque processeur dispose de sa propre file pour ordonnancer son propre système de tâches, d'une manière totalement indépendante des autres processeurs (voir Figure 1.3). Sur chaque processeur, une stratégie d'ordonnancement locale est implémentée.
Pour veiller au respect de non-dépassement des taux d'utilisation des différents processeurs, l'algorithme de partitionnement de tâches s'assure que $\sum_i u_i^k$ ne dépasse pas la borne supérieure d'ordonnançabilité associée à chacun des processeurs.
L'algorithme de répartition des tâches s'apparente à un problème NP-difficile nommé *bin-packing* (Johnson 1974), dans lequel il faut arriver à placer k objets différents (taux d'utilisation processeur dans ce cas) dans m boîtes de tailles identiques (borne supérieure d'ordonnançabilité). Pour résoudre ce type de problème, des heuristiques de complexité polynomiale comme *First-Fit* et *Best-Fit* (Liu et Layland 1973) sont utilisées au sein d'un algorithme de partitionnement. Avec *First-Fit*, chaque tâche est assignée au premier processeur qui peut l'accepter. Le *Best-Fit* quant à lui, choisit parmi les processeurs qui peuvent accepter la tâche, celui dont le facteur d'utilisation du processeur est le plus petit.

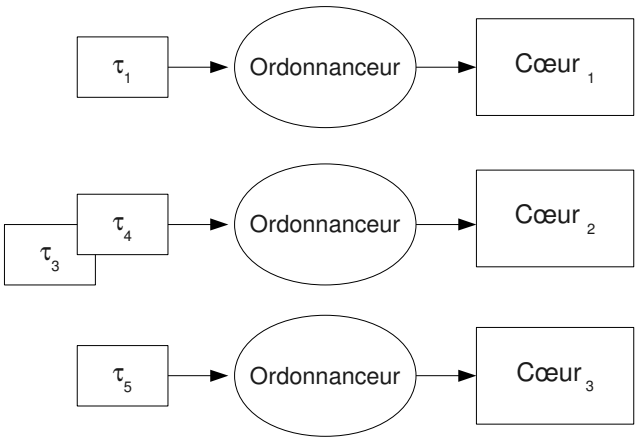


FIGURE 1.3 – *Approche partitionnée*

Approches globales

Contrairement à l'approche par partitionnement, l'approche globale dispose d'une seule file d'attente pour l'ensemble des processeurs. Une seule politique d'ordonnancement est appliquée pour l'ensemble des tâches. En outre, à la différence de l'approche par partitionnement, l'approche globale ne fait aucune restriction quant à l'affectation des travaux d'une même tâche aux différents cœurs. De plus, pour un travail donné, celui-ci peut à chaque instant être réaffecté à un autre cœur, même si le travail est en cours d'exécution (voir Figure 1.4).

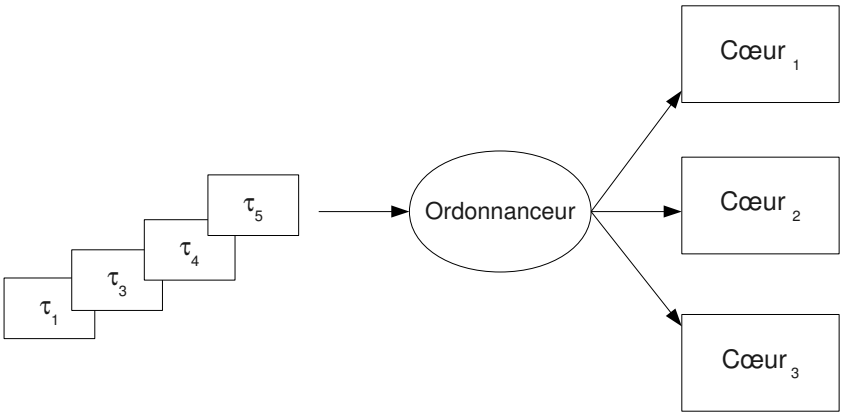


FIGURE 1.4 – *Approche globale*

Approches hybrides

Cette approche combine les deux approches précédentes, en utilisant par exemple l'approche globale au premier niveau, puis l'approche par partitionnement en second niveau. Dans ce cas, les tâches sont d'abord globalement réparties suivant l'approche globale précédemment décrite. Dans un deuxième temps, les travaux de ces tâches passent d'abord par une file de tâches associée à chaque processeur, pour laquelle une politique d'ordonnancement locale est appliquée, typiquement l'approche par partitionnement (voir Figure 1.5). A noter qu'il existe plusieurs variantes qui diffèrent selon la manière de combiner les deux approches, à savoir globales et par partitionnement. Pour le lecteur intéressé par ces variantes, nous l'invitons à consulter Davis et Burns (2011).

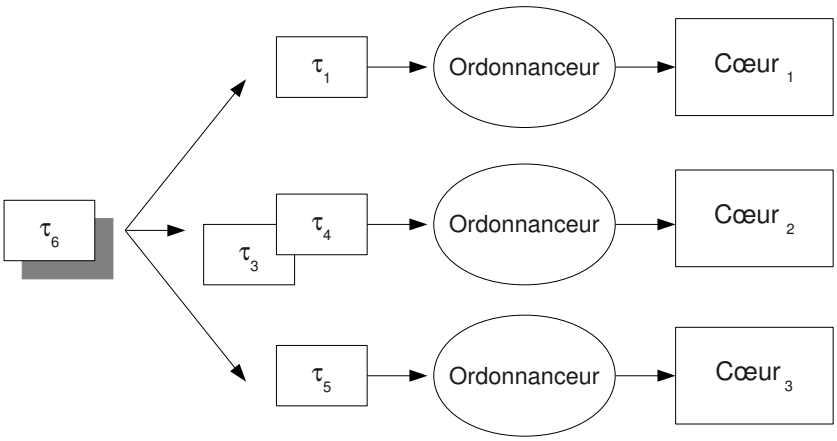


FIGURE 1.5 – *Exemple d'approche hybride*

1.2.4 Types de migrations

La notion de migration dans les systèmes multiprocesseurs temps-réel est souvent non prise en compte. Ceci est dû au fait que d'une part, déplacer un travail assigné à un processeur donné vers un autre processeur est coûteux en temps. D'autre part, la théorie de l'ordonnancement temps réel multiprocesseur ainsi que les outils utilisés, ont été jusqu'à présent développés pour des approches par partitionnement où la migration n'est pas tolérée.
En plus de leur classification par priorités, Carpenter et al. (2004) classent également les ordonnancements temps réel multiprocesseurs par migration :

Sans migration

Cette classe n'autorise aucune migration. Est ici classée l'approche par partitionnement vue précédemment.

Migration restreinte

La migration dans ce type de classe n'est autorisée qu'au niveau travail. Celui-ci doit donc s'exécuter entièrement sur un seul processeur. Dans cette classe, l'approche hybride est utilisée en affectant d'abord les travaux aux différentes files, et au niveau de chacune des files, les travaux sont ordonnancés localement.

Pleine migration

Cette classe n'appose aucune restriction quant à la migration des travaux.

1.2.5 Classification des tests d'ordonnançabilité

Carpenter et al. (2004) ont établi une classification par ordonnançabilité. Cette classification considère à la fois le type de priorité et la classe de migration auxquels appartient l'algorithme d'ordonnancement. Le taux d'utilisation $U = \sum_i u_i^k$ est donné pour chacun des cas dans le tableau 1.1, avec $|M|$ le nombre total de processeurs et $\alpha = u_i^{max,k}$, le facteur d'utilisation maximal par tâche.

1.2.6 Algorithmes d'ordonnancement

P-EDF

Cet algorithme correspond à la classe (partitionné, dynamicité restreinte) du tableau 1.1. Pour illustrer le principe de l'ordonnancement EDF en mode partitionné, nous reprenons l'exemple présenté dans Carpenter et al. (2004). Le système de tâches $\tau = \{\tau_i(r_i, C_i, P_i, D_i), i = 1..4\}$ de la Figure 1.6 a pour taux d'utilisation $U = 2$ et peut donc être ordonnancé en supposant une plateforme à deux cœurs.

	Statique	Dynamicité restreinte	Dynamique
1	$U = \frac{\lfloor M \rfloor + 1}{2}$ (Andersson et Jonsson 2003)	$U = \frac{\beta \lfloor M \rfloor + 1}{\beta + 1}$ où $\beta = \lfloor \frac{1}{\alpha} \rfloor$ (López et al. 2004)	$U = \frac{\beta \lfloor M \rfloor + 1}{\beta + 1}$ où $\beta = \lfloor \frac{1}{\alpha} \rfloor$
2	$U \leq \frac{\lfloor M \rfloor + 1}{2}$	$U \geq \lfloor M \rfloor - \alpha(\lfloor M \rfloor - 1)$ si $\alpha \leq \frac{1}{2}$ (Baruah et Carpenter 2005) $U = \frac{\lfloor M \rfloor + 1}{2}$ sinon	$U \geq \lfloor M \rfloor - \alpha(\lfloor M \rfloor - 1)$ $U = \frac{\lfloor M \rfloor + 1}{2}$ si $\alpha > \frac{1}{2}$
3	$\frac{\lfloor M \rfloor^2}{3\lfloor M \rfloor - 2} \leq U \leq \frac{\lfloor M \rfloor + 1}{2}$ (Andersson et al. 2001)	$U = M - \alpha(\lfloor M \rfloor - 1)$ si $\alpha \leq \frac{1}{2}$ (Goossens et al. 2003) $U = \frac{\lfloor M \rfloor + 1}{2}$ sinon (Baruah 2004)	$U = \lfloor M \rfloor$ (Baruah et al. 1996)

TABLE 1.1 – *Borne supérieure d'ordonnançabilité des classes d'algorithmes multiprocesseurs*

1 : partitionné, 2 : migration restreinte, 3 : pleine migration.

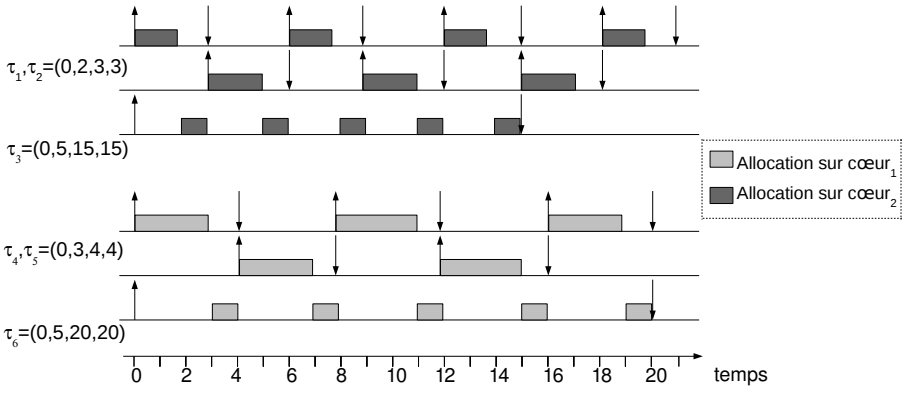


FIGURE 1.6 – *Ordonnancement sous P-EDF (Carpenter et al. 2004)*

G-EDF

A la différence de P-EDF, l'exemple de la Figure 1.7 montre que pour la même plateforme et le même système de tâches considéré, G-EDF permet la pleine migration. En effet, chaque tâche du système, a eu au moins un changement d'allocation processeur lors de son exécution. En outre, entre les instants $t = 11$ et $t = 12$, le cœur 2 reste inactif, ce qui conduira plus tard à une violation d'échéance, puisque le taux d'utilisation est supposé être de 100% pour les deux cœurs.

Comparaison : P-EDF et G-EDF

En terme d'ordonnancement, G-EDF tend à chaque instant à minimiser voire éliminer les intervalles de temps durant lesquels le nombre de processeurs est supérieur à celui des tâches en attente, évitant ainsi l'inactivité des processeurs. Cependant, cette inactivité ne peut être satisfaite dans tous les cas, pour un ordonnancement à dynamicité restreinte des tâches ou travaux.
Sous P-EDF et comme le montre l'exemple de la Figure 1.6, τ_2 et τ_4 sont affectés à des cœurs différents. Aucune règle commune n'est alors prise pour ordonnancer τ_2 et τ_4. Ceci permet un chevauchement des travaux dans le temps (typiquement à l'instant $t = 11$), évitant ainsi l'inactivité

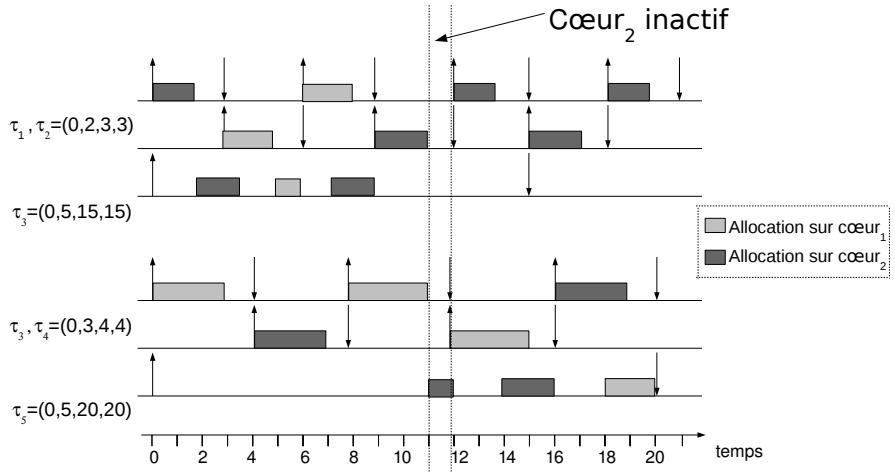


FIGURE 1.7 – *Ordonnancement sous G-EDF (Carpenter et al. 2004)*

des processeurs. Ce chevauchement ne peut jamais se produire avec une politique d'ordonnancement à pleine migration et à dynamicité restreinte. En effet, la dynamicité restreinte sous-entend que l'affectation d'une priorité à un travail reste constante durant toute son exécution. Mais puisque l'espace de priorité pour la pleine migration est le même pour tous les travaux, ceci implique qu'un travail ne peut se chevaucher dans le temps avec un autre que si l'un des travaux a une priorité supérieure à l'autre. Carpenter et al. (2004) ont montré qu'il n'existe pas d'algorithmes d'ordonnancement à pleine migration et à dynamicité restreinte, capables d'ordonnancer le système de tâches considéré dans l'exemple de la Figure 1.7.

Pfair

L'approche par partitionnement qui est largement étudiée et utilisée sur les systèmes multiprocesseurs, montre ses limites face à des systèmes de tâches à pleine charge ($U = |M|$). L'exemple simple avancé dans la littérature pour montrer cette limite, est un système à trois tâches périodiques avec les mêmes paramètres temporels (r_i, C_i, P_i, D_i) = (0, 2, 3, 3). Ce système n'est en effet pas ordonnançable sur une plateforme à $|M| = 2$ cœurs sauf si la migration (pleine ou restrictive) est permise.
Une des approches globales la plus répandue dans la littérature est celle basée sur l'équité d'exécution (*Proportionate-fairness* en anglais). Au sein de cette catégorie, l'algorithme *Pfair* proposé par Baruah et al. (1996) est optimal pour les systèmes multiprocesseurs. Cet algorithme associe pour chaque tâche un poids qui spécifie le ratio de charge d'un seul processeur pour lequel la tâche est affectée. La décision d'ordonnancement est alors prise de manière à ce que chaque tâche puisse recevoir approximativement son ratio temps processeur.
Cependant, cet algorithme souffre d'un important taux de migrations, et ne met aucune restriction sur les migrations, l'exposant ainsi à de nombreux problèmes pratiques Moir et Ramamurthy (1999).

Dans Pfair, les tâches sont divisées en *sous-tâches* de quantum égaux, et le temps est également divisé en intervalles de tailles égales appelés *fenêtres*. Chaque sous-tâche doit s'exécuter au sein de sa fenêtre de temps associée. Formellement, un ordonnancement S en Pfair est défini comme suit : $S : \tau \times \mathbb{Z} \mapsto \{0,1\}$. Si $S(\tau_i, t) = 1$ alors τ_i est ordonnancée au sein de $[t, t+1)$. Dans un ordonnancement idéal, chaque tâche τ_i reçoit exactement $u_i^k \times t$ unités de temps processeur dans l'intervalle $[0, t)$ et toutes les échéances sont respectées. L'algorithme Pfair essaye alors d'atteindre les meilleures performances en comparant sa propre allocation à celle d'un algorithme idéal. Cette comparaison est basée sur une fonction différence dite *lag* qui représente les erreurs d'allocation associées à chaque tâche :

$$lag(\tau_i, t) = u_i^k \times t - \sum_{j=0}^{t-1} S(\tau_i, j) \tag{1.1}$$

L'ordonnancement S est dit Pfair si et seulement si :

$$-1 < lag(\tau_i, t) < 1 \tag{1.2}$$

D'une manière informelle, l'équation précédente impose que chaque tâche ait toujours une erreur d'allocation processeur inférieure à une unité de temps. Il en résulte qu'à chaque instant, τ_i doit recevoir soit $\lfloor u_i^k \times t \rfloor$ ou bien $\lceil u_i^k \times t \rceil$ unités de temps processeur.
Un test de faisabilité a été proposé dans Baruah et al. (1996) pour un système de tâches périodiques synchrones sur un ensemble de M processeurs. Ce test est une condition nécessaire et suffisante pour un algorithme Pfair :

$$\sum_{\tau_i \in \tau} \frac{C_i}{P_i} \leq |M| \tag{1.3}$$

Comme souligné précédemment, l'algorithme Pfair souffre d'un important nombre de migrations. Pour y pallier, une multitude de variantes que nous n'allons pas détailler ici, sont proposées pour réduire ces coûts de migrations incluant PF (Baruah et al. 1996), PD (Baruah et al. 1995), PD^2 (Anderson 2000) et EPDF (Anderson et Srinivasan 2000). Ces algorithmes sont tous optimaux, dans la mesure où chacun d'eux satisfait la condition d'ordonnançabilité (1.3) de Pfair. Notons que l'algorithme EPDF a été montré optimal seulement pour des systèmes ayant au moins deux processeurs. Cet algorithme offre néanmoins certains avantages pratiques (tels que les coûts d'ordonnancement) par rapport aux autres algorithmes optimaux.

1.3 SYNCHRONISATION EN TEMPS RÉEL MULTIPROCESSEUR

Dans la littérature, la synchronisation est un aspect qui suscite un grand intérêt. L'un des principaux défis lié à cette partie, est de pouvoir assurer l'intégrité des ressources partagées lors d'accès concurrents effectués par les tâches, tout en veillant à ce que ces dernières respectent leurs échéances. Pour cela, des protocoles ont été proposés selon deux approches (protocoles bloquants et non-bloquants).

1.3.1 Protocoles bloquants

Ces protocoles reposent sur la technique des verrous. Les protocoles que nous présentons ici supposent une réentrance binaire, autrement dit, une seule tâche peut accéder en même temps à une ressource partagée. Les sections de code en exclusion mutuelle sont qualifiées de *sections critiques*.

PIP (Priority Inheritance Protocol)

Ce protocole a été proposé (Sha et al. 1990) pour résoudre le problème d'interblocage entre tâches ainsi que le phénomène d'inversion de priorité.
Le principe consiste à rehausser la priorité de la tâche obtenant la ressource critique, à celle de la tâche qu'elle bloque en attente de la même ressource. Il s'agit donc d'attribuer à la tâche τ_i entrant en section critique, une priorité $Pr(\tau_i) = Max\{Pr(\tau_1), Pr(\tau_2), Pr(\tau_3), ..., Pr(\tau_n)\}$, sous condition que les tâches $\tau_1, \tau_2, \tau_3, ...\tau_n$ aient une priorité supérieure à celle de τ_i et attendent toutes la ressource détenue par τ_i. La priorité de τ_i est rétablie au moment de la libération de la ressource qu'elle a occupée. De cette manière, aucune tâche de priorité inférieure ne pourra préempter la tâche τ_i durant son exécution en section critique.

PCP (Priority Ceiling Protocol)

Ce protocole (Sha et al. 1990) prévient les situations d'interblocages. Il définit une nouvelle notion : la *priorité plafond* (PP) d'une ressource, qui est une valeur maximum des priorités des tâches qui l'utilisent, en posant la condition suivante : une tâche τ_i ne peut entrer en section critique que si la priorité de la ressource qu'elle demande, est strictement supérieure à la priorité plafond des ressources en cours d'utilisation (hormis celles de τ_i). En section critique, la tâche devra hériter de la priorité de la tâche la plus prioritaire qu'elle bloque. Ceci sous-entend que le protocole à priorité plafond résout également le problème d'inversion de priorité. De plus, contrairement au protocole PIP, le temps de blocage est réduit à une seule section critique.
Des versions multiprocesseurs (M-PCP) et distribuées (D-PCP) sont proposées par Rajkumar (1991). Pour le M-PCP, l'auteur a considéré une plateforme multiprocesseur avec un ordonnancement à priorités statiques et partitionné. L'adaptation du M-PCP à l'ordonnancement P-EDF est quant à lui proposé par López et al. (2004).

SRP (Stack Resources Protocol)

Tout comme le protocole PCP, SRP se base également sur la notion de priorité plafond (Baker 1991). Cependant, à la différence du PCP, le protocole SRP satisfait immédiatement chaque requête du travail j_i demandant une ressource. Sous SRP, le blocage est effectué uniquement lors du réveil du travail j_i, et son exécution n'est entamée que lorsque sa priorité devient plus grande que la priorité plafond du système. Ainsi, les travaux sont bloqués au plus une seule fois, et il n'y a aucun besoin d'utiliser l'héritage de priorité.

La version multiprocesseur M-SRP pour la politique P-EDF a été proposée par López et al. (2004) et Gai et al. (2003). Ces derniers auteurs ont discuté son implémentation étant donnée la nature expérimentale de leur travail. Les résultats de ce travail ont montré que M-SRP offre de meilleures performances que M-PCP.

FMLP (Flexible Multiprocessor Locking Protocol)

Ce protocole a été spécialement conçu pour les systèmes multiprocesseurs (Block et al. 2007). Il prend en effet en compte les approches partitionnée et globale. L'unité de blocage dans FMLP est un groupe de ressources. Chaque groupe contient soit seulement de longues ressources ou seulement de courtes ressources, et chacun de ces groupes est protégé par un verrou qui est soit un *spinlock* (pour les courtes ressources), soit un sémaphore (pour les longues ressources). Deux ressources R_1 et R_2 sont du même groupe si et seulement s'il existe un travail j_i qui lance une requête pour détenir R_1 et R_2, et que ces deux ressources sont toutes les deux de même type (longues/courtes).
Le protocole FMLP permet ainsi l'utilisation des ressources d'une manière imbriquée. En outre, il a été démontré dans (Block et al. 2007) que FMLP évite l'interblocage et offre de meilleures performances que le protocole M-SRP (Gai et al. 2003).

Stratégie par Supertâches

Moir et Ramamurthy (1999) ont observé que la migration sous Pfair peut poser problème lors de la gestion de ressources partagées. En effet, une tâche qui communique par exemple avec un périphérique externe pourrait avoir besoin de s'exécuter sur un processeur particulier et ne pas migrer. Pour supporter la non-migration des tâches, les auteurs ont proposé l'utilisation de la notion de supertâche (*supertasking*). Dans leur approche, une supertâche S_k remplace l'ensemble des tâches qui sont affectées au processeur $k \in M$ et partageant les mêmes ressources. Avec ce concept, pour $\tau_i \notin S_k$, le pire temps requis pour obtenir la ressource détenue par le groupe S_k est $Max_{\tau_j \in S_\tau}\{B_j\}$ (au lieu de $\sum_{\tau_j \in S_\tau} B_j$), où B_j est le temps de blocage causé par la tâche τ_j.
Selon Pfair, chaque surpertâche est dotée d'un ratio d'utilisation du processeur défini comme suit :

$$u(S_k) = \sum_{\tau \in S_k} u^k(\tau) \tag{1.4}$$

Les tâches partageant les mêmes ressources sont donc groupées puis ordonnancées comme une seule tâche. Le principal problème qui apparaît lors de l'ordonnancement avec cette approche est celui de la faisabilité du système. Deux approches sont alors données dans la littérature. La première consiste à gérer les supertâches comme étant des classes spéciales lors de l'utilisation d'une approche globale. Cependant, la réservation du traitement particulier aux supertâches réduit considérablement les performances du système. Une approche alternative a été proposée dans Holman et Anderson (2001) pour le Pfair. Cette approche consiste à affecter

un ratio d'utilisation du processeur suffisamment grand à une supertâche de manière à ce qu'elle ne manque pas son échéance. Au sein du groupe d'une supertâche, une autre politique d'ordonnancement (que le Pfair) peut être appliquée, et on parle alors d'un ordonnancement hiérarchique. Pour les tâches appartenant au même groupe, l'algorithme EDF est recommandé pour son optimalité à ordonnancer des sous-ensembles de tâches. Sous Pfair, une règle a été donnée par Holman et Anderson (2001) pour que les supertâches ne manquent pas leur échéance :

$$u(S_k) = min(1, \sum_{\tau \in S} u^k(\tau) + \frac{1}{L}) \tag{1.5}$$

où L correspond à la taille minimale de la fenêtre sur tout le système de tâches (incluant les supertâches). Cette règle est valable en ordonnançant les sous-ensemble des supertâches avec l'algorithme EDF ou EPDF.

1.3.2 Protocoles non-bloquants

Ces protocoles permettent la pleine réentrance au niveau des sections critiques. Pour la gestion de conflits entre les tâches accédant aux ressources, des modes de résolution existent et leur stratégie diffèrent selon les contraintes de temps imposées aux tâches. En outre, ces protocoles se distinguent généralement des protocoles bloquants par la non-utilisation de verrouillage à gros grain pour assurer la synchronisation des tâches.
Afin d'assurer une synchronisation à grain fin, les protocoles non-bloquants se basent généralement sur des instructions machine de type CAS (*Compare And Swap*) (IBM 1970) pour les primitives modifiant atomiquement un seul mot mémoire (voir l'Algorithme 1), ou un double mot mémoire contigü pour les DCAS (*Double Compare And Swap*), voire plusieurs mots mémoire par MCAS (*Multi-word Compare And Swap*) pour mono/multiprocesseurs. D'autres instructions machine comme LL/SC (*Load-Link/Store-Conditional*) sont également utilisées (Anderson et Moir 1999) mais étant donnée la difficulté à les émuler ainsi que le fait qu'elles opèrent en général sur des données de tailles identiques à celle du registre d'instructions, les instructions de type CAS ont par conséquent attiré plus d'attention dans la littérature.

Algorithme 1 : Algorithme de l'instruction atomique CAS

Require: &*mot_partage*, *ancienne_valeur*, *nouvelle_valeur*
1: Init : *valeur_temp* ← ∗*mot_partage*
2: **if** *valeur_temp* = *ancienne_valeur* **then**
3: ∗*mot_partage* ← *nouvelle_valeur*
4: **end if**
5: **return** *valeur_temp*

Wait-free

Pour les tâches temps réel à contraintes strictes, le *wait-free* est un mode d'accès aux ressources d'une manière non-bloquante, qui assure à chaque instant que l'exécution de chaque tâche progresse, et ce, malgré les conflits

qui peuvent survenir avec d'autres tâches. Pour cela, les tâches qui arrivent à progresser doivent surveiller périodiquement l'état des autres tâches et les aider à progresser. Cette stratégie est toutefois difficile à mettre en œuvre sur une plateforme matérielle standard, puisque l'équité pour l'accès à la mémoire n'est pas toujours garantie. Il existe de rares tentatives d'implémentation adaptées au monoprocesseur, proposées avec des instructions de type DCAS (Anderson et Moir 1995), ou MCAS (Anderson et al. 1997b) voire LL/SC (Anderson et Moir 1999).

Lock-free

Cette stratégie de gestion des ressources, contrairement au wait-free, est plutôt destinée à établir des contraintes de temps de type souples lors de l'accès aux sections critiques. En effet, le lock-free garantit qu'à chaque instant, l'exécution d'au moins une tâche peut progresser. Autrement dit, il existe des tâches sous cette stratégie qui peuvent manquer leurs échéances faute d'un grand nombre de réitèrements non-bornés lors des conflits d'accès aux sections critiques.
Le principal défi auquel est confrontée cette stratégie est la transformation du code séquentiel en lock-free. Cette transformation dite *construction universelle* (Lamport 1977) représente une classe d'implémentation du lock-free. Il s'agit en général de prendre en compte de nombreuses constructions de structures de données, afin de pouvoir automatiser leur transformation en lock-free par des compilateurs ou environnements dédiés (Turek et al. 1992, Barnes 1993).
Un autre défi consiste à trouver des stratégies d'ordonnancement afin de borner le nombre *d'interférences* entre travaux. Dans (Anderson et al. 1997c) l'interférence concerne un travail manipulant un objet en lock-free et préempté par un autre travail de plus haute priorité mais que ce dernier ne manipulera pas. Ces mêmes auteurs posent une condition simple quant au nombre d'itérations nécessaires pour l'accès aux objets lock-free en le considérant constant. Ceci a permis de déduire des tests d'ordonnançabilité pour les politiques DM et EDF en incluant ces temps de rejoues. Cependant, les conditions posées sont pessimistes, et de plus, elles ne sont applicables que pour des environnements monoprocesseurs.
Il existe peu de travaux utilisant à la fois le lock-free comme synchronisation tout en considérant l'environnement multiprocesseur. Dans Holman et Anderson (2006) les auteurs proposent la prise en compte du lock-free dans un environnement multiprocesseur pour l'algorithme d'ordonnancement Pfair pouvant être cependant généralisé, à tout algorithme d'ordonnancement basé sur des quantums de temps. Les auteurs utilisent une structure de données simple avec insertion dans une file. L'accès à cette file utilise une primitive de type CAS. Le temps d'accès est d'abord borné en tenant compte du nombre de tentatives pour accéder à un objet dans la file. A partir de cette borne, la pire durée d'accès à l'objet est ensuite déduite. Cette borne est réduite par la suite en utilisant le concept de supertâche.

Sur des architectures multiprocesseurs, la synchronisation par lock-free est préférable pour des structures de données simples comme les files, les piles, les buffers et les listes. Cependant, pour des structures de

données plus complexes, les protocoles bloquants offrent de meilleures performances. Cette affirmation est répandue dans la littérature puis a récemment été étayée expérimentalement sur une plateforme réelle multiprocesseur par Brandenburg et al. (2008).

Obstruction-free

En comparaison avec le wait-free et le lock-free présentés précédemment, l'obstruction-free (Herlihy et al. 2003a) représente la plus faible garantie de progression d'une tâche. Ce type de synchronisation est aussi souvent appelé synchronisation *optimiste*. Cette faible capacité à assurer la progression rend ce type de synchronisation non adapté aux systèmes temps réel.

Conclusion

Loin d'être exhaustif, ce chapitre a introduit les principales problématiques dans les systèmes temps réel multiprocesseurs. L'accent a été mis sur les différentes approches et politiques d'ordonnancement mais aussi et surtout sur les différents types de synchronisation qui serviront de discussion dans les prochains chapitres. Parmi ceux-ci, le lock-free a été mis en avant pour montrer l'intérêt d'adopter un mécanisme non-bloquant pour la gestion de la concurrence entre tâches temps réel.
La plupart des travaux se sont intéressés à la difficulté d'implémenter des structures de données pour des synchronisations sans verrous. D'autres travaux traitent du problème de la concurrence non-bloquante en termes d'ordonnançabilité mais supposent des conditions pessimistes ou se restreignent à un environnement monoprocesseur. La synchronisation non-bloquante en multiprocesseur est donc très peu considérée étant donné que la théorie de l'ordonnancement liée à l'environnement parallèle est relativement jeune. De plus, décrire rigoureusement l'accès concurrent aux ressources partagées d'une manière optimiste, reste un problème ouvert et difficile.

Les systèmes transactionnels 2

Sommaire

2.1 Introduction aux systèmes transactionnels

Dans la littérature les systèmes transactionnels sont généralement classés en deux grandes familles, dites *classiques* et *temps réel*. Les systèmes transactionnels classiques ne prennent pas en charge les contraintes de temps contrairement aux systèmes temps réel.

2.1.1 Les transactions

Une *transaction* est définie comme une séquence d'actions qui apparaissent invisibles et instantanées à un observateur extérieur. Pour garantir la cohérence dans un système de gestion de base de données, les transactions doivent vérifier les propriétés d'Atomicité, de Cohérence, d'Isolation et de Durabilité (ACID).

Atomicité

Une transaction doit effectuer toutes ses mises à jour ou ne rien faire du tout. En cas d'échec, le système doit annuler toutes les modifications qui ont été faites par la transaction. Les problèmes d'atomicité interviennent généralement lorsque survient un évènement susceptible d'interrompre la transaction.

Cohérence

La transaction doit faire passer la base de données d'un état cohérent à un autre état cohérent. En cas d'échec, l'état initial doit être restauré. Les problèmes de cohérence interviennent généralement lors d'un conflit d'accès à une ou plusieurs donnée(s) manipulée(s) par des transactions.

Isolation

Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée pour éviter les interférences entre les transactions. Des accès concurrents non contrôlés aux données peuvent violer cette propriété.

Durabilité

Dès qu'une transaction est validée, le système doit garantir que ses modifications (mises à jour) seront conservées quoi qu'il arrive.

Actions d'une transaction

Les opérations que peut exécuter une transaction sont la lecture, l'écriture, l'abandon (*abort* en anglais) et la validation qualifiée aussi de *commit* en anglais.
Après avoir exécuté toutes ses opérations de lecture et d'écriture éventuelles, la transaction tente de les valider. Si cette dernière action échoue, la transaction fait un *abort* et tente de recommencer depuis le début (*rollback* en anglais). Sinon, la transaction est dite *validée* ou *commitée*.

Gestion des transactions

Un des principaux problème dans la gestion des transactions est le contrôle des transactions accédant d'une manière simultanée aux données. Des conflits peuvent alors apparaître. Ces conflits sont classés en trois catégories : Écriture/Écriture, Lecture/Écriture et Écriture/Lecture (Gardarin 2000). La solution est donc de verrouiller les objets jusqu'à la terminaison de la transaction. Cependant, il est nécessaire de laisser s'exécuter un maximum de transactions, et ce, pour des raisons de performances mais tout en veillant à respecter l'intégrité de la base. Cela est assuré par des protocoles de *contrôle de concurrence* (Eswaran et al. 1976).
Les protocoles de contrôle de concurrence ont pour principal rôle de veiller à ce que l'ordre dans lequel les actions sont exécutées par les transactions dans le temps, n'ait pas d'influence sur le résultat final. Une telle caractéristique de cohérence est appelée *sérialisabilité*.

2.1.2 Les contrôleurs de concurrence

Dans les Systèmes de Gestion de Base de Données (SGBDs) classiques (non temps réel), il existe différents types de protocoles de contrôle de concurrence que l'on peut regrouper en deux grandes familles : les protocoles pessimistes et les protocoles optimistes.

Les protocoles pessimistes

Ils partent de l'hypothèse que les conflits apparaissent fréquemment lors de l'exécution des transactions. Ils se basent donc sur les verrous pour effectuer la synchronisation.

Les protocoles optimistes

Ces protocoles quant à eux, considèrent les conflits peu nombreux et la synchronisation des transactions peut être alors différée jusqu'au moment de leur terminaison. Cette section se focalise sur l'environnement monosite. Pour le lecteur intéressé par les protocoles de contrôle de concurrence dans les environnements distribués, nous l'invitons à consulter Özsu et Valduriez (2011).

2.2 Le transactionnel temps réel

Les applications temps réel accédant à des bases de données, affichent des contraintes de temps que les protocoles de contrôle de concurrence utilisés dans les SGBDs classiques, ne sont pas en mesure de satisfaire. En effet, il s'agit pour le transactionnel temps réel non seulement de prendre en compte les contraintes d'intégrité mais aussi et surtout de veiller lors de la gestion des transactions à ce que celles-ci respectent leurs échéances (Abbott et Garcia-Molina 1988). Tout comme les tâches temps réel, les transactions sont classées selon le niveau de criticité attaché à leurs contraintes de temps. Trois classes de contraintes de temps sont alors considérées dans la littérature : *dures*, *souples*, et *fermes*. Dans les

SGBDs temps réel, la première classe (temps réel dur) est rarement utilisée et à plus forte raison pour les environnements multiprocesseurs. Pour ces derniers, l'exemple le plus cité est MDARTS (*Multiprocessor Database Architecture for Real-Time Systems*) qui est une implémentation d'un SGBD temps réel à contraintes strictes, et dans lequel les transactions peuvent accèder aussi bien à une mémoire partagée que distribuée par le biais du RPC (*Remote Procedure Call*) (Lortz 1994). Les sections critiques dans MDARTS sont cependant implémentées d'une manière non-préemptive. Dans (Anderson et al. 1997a), les auteurs ont proposé une approche préemptive pour la gestion de transactions à contraintes strictes. Dans cette approche, les transactions sont dotées d'une priorité et sont synchronisées en *wait-free*.
Les transactions à contraintes fermes sont les plus utilisées dans les SGBDs temps réel. La contrainte ferme combine à la fois les avantages des systèmes à contraintes strictes, tout en gardant la souplesse des systèmes à contraintes souples. C'est ainsi qu'une transaction ferme peut être redémarrée, ou non, par le contrôleur de concurrence, dans le cas où elle ne respecte pas son échéance.

2.2.1 Les protocoles pessimistes

Le protocole 2-PL-HP

Le protocole *2 Phase Locking - High Priority* résout les conflits en tenant compte des priorités des transactions, ces priorités étant définies à partir des échéances des transactions. Le protocole 2PL-HP permet de garantir qu'une résolution de conflit entre deux transactions ne se fait pas au profit de la transaction de plus basse priorité. Autrement dit, les conflits sont résolus en faveur des transactions les plus proches de leur échéance. Le protocole 2PL-HP est pessimiste puisque les conflits sont résolus dès leur apparition en utilisant un mécanisme de verrous. Le principe de ce protocole est le suivant : (Abbott et Garcia-Molina 1988) :

- Si une transaction T_j requiert un verrou sur une donnée déjà verrouillée par d'autres transactions de plus faibles priorités, alors ces dernières sont abandonnées et T_j obtient le verrou et continue à s'exécuter.
- Si la priorité de T_j est inférieure aux priorités des autres transactions, alors T_j attend que la donnée soit libérée.

A noter que pour prévenir les interblocages, les priorités des transactions doivent être statiques. L'inconvénient de cette méthode est que des transactions peuvent être abandonnées à cause de l'exécution d'une transaction de plus haute priorité qui, plus tard, peut manquer son échéance. De plus, une transaction qui est abandonnée pour résoudre le conflit, peut être sur le point de se terminer conduisant ainsi à une baisse de performance du système.

Le protocole 2-PL-WP

Le protocle 2 *Phase Locking - Wait Promote* se base sur le concept d'héritage de priorité répandu dans les systèmes temps réel (Sha et al. (1990)). Lorsqu'une transaction est bloquée par une autre transaction de plus faible priorité, cette dernière hérite de la priorité la plus forte. La transaction qui détient le verrou continue à s'exécuter avec cette nouvelle priorité. Sa priorité étant plus forte, elle a davantage de chances de se terminer plus tôt et permettre ainsi à la seconde transaction de s'exécuter tout en respectant son échéance. Autrement dit, l'héritage de priorité tente de minimiser le temps de blocage des transactions les plus prioritaires. Cependant, ce protocole souffre du problème d'inversion de priorité dans la mesure où une transaction de haute priorité peut être bloquée plusieurs fois durant son exécution en attente des transactions de plus basses priorités. En outre, les transactions de faible priorité ayant hérité de la priorité la plus élevée pour s'exécuter, risquent de concurrencer les transactions de hautes priorités pour l'accès aux autres ressources du système et peuvent les conduire ainsi à manquer leur échéance.
Une variante de ce protocole a été proposée par Huang et Stankovic (1990). Elle permet l'héritage conditionnel de priorité. Il s'agit d'un protocole où une transaction de faible priorité n'hérite d'une haute priorité que dans le cas où elle est proche de sa terminaison. Sinon, elle est abandonnée. L'avantage est qu'il y a moins de travail perdu si la transaction est abandonnée loin de sa terminaison, et les délais d'attente sont réduits pour les transactions de haute priorité. En effet, ces dernières ne sont plus bloquées par des transactions de plus faibles priorités qui nécessitent encore beaucoup de temps pour se terminer.

Le protocole CCA

Le protocole *Cost Conscious Approach* a été proposé par Hong et al. (1993) en vue de la prise en compte d'informations statiques comme l'échéance et d'informations dynamiques, telles que le temps d'exécution restant ou le nombre de redémarrages de la transaction. Il présente de meilleures performances par rapport aux protocoles 2-PL-HP et 2-PL-WP. Le protocole CCA dans sa version initiale ne prend pas en charge l'aspect de facteur de charge du système. La variante CCA-ALF (*Average Load Factor*) qui améliore la première version, a été proposée par Chakravarthy et al. (1994) et prend en compte la charge du système au moment de l'exécution des transactions et permet donc de recalculer dynamiquement les priorités des transactions en fonction des ressources système disponibles.

2.2.2 Les protocoles optimistes

Phases d'un protocole optimiste

Lorsque les transactions s'exécutent de manière optimiste, elles doivent passer un test de certification pour vérifier qu'elles ne sont pas en conflit avec d'autres transactions (Kung et Robinson 1981). Les transactions s'exécutent alors en trois phases : lecture, validation, et écriture. Pendant la phase de lecture, une transaction s'exécute comme si elle était seule dans

le système et ses mises à jour sont effectuées dans son espace mémoire privé. Ensuite, un test de certification permet de vérifier si la transaction peut être validée. Ce n'est que qu'après la phase de validation que les données sont écrites dans la base.

Stratégies de validation

Il existe deux stratégies de validation. La première vérifie que, lors de la phase de certification, la transaction courante n'est pas en conflit avec les transactions déjà validées : c'est la *certification en arrière*. La seconde qui vérifie l'apparition de conflits avec les transactions en cours d'exécution, est dite de *certification en avant*. Si des conflits sont détectés, alors soit la transaction en cours de validation est abandonnée et redémarrée, soit des transactions en cours d'exécution doivent être abandonnées puis redémarrées. Le protocole OCC (*Optimistic Concurrency Control*) est un exemple de protocole par certification en avant.

Définition 2.1 *Soit une transaction T_j en phase de validation, et un ensemble actif de transactions T_k pour $k \in [1, n]$ et $k \neq j$, et soient les ensembles $RS(T)$ et $WS(T)$ contenant respectivement les objets ouverts par la transaction T en Lecture et en Écriture. Trois cas de conflits peuvent causer l'ordre de sérialisation des transactions (Lee et Son 1993) :*

1. *Si $RS(T_j) \cap WS(T_k) \neq \emptyset$*
   Le conflit lecture/écriture entre T_j et T_k peut se résoudre en ajustant la sérialisation entre T_j et T_k de manière à ce que la lecture de T_j ne puisse pas être affectée par l'écriture de T_k. Ce type d'ajustement est appelé : ajustement en avant ou bien ordre en avant.
2. *Si $WS(T_j) \cap RS(T_k) \neq \emptyset$*
   Le conflit écriture/lecture peut être résolu entre T_j et T_k en faisant précéder d'abord la lecture de T_k avant l'écriture de T_j. Cet ajustement d'ordre de sérialisation est appelé dans ce cas : ajustement en arrière.
3. *Si $WS(T_j) \cap WS(T_k) \neq \emptyset$*
   Le conflit écriture/écriture peut être résolu entre T_j et T_k en ajustant l'ordre de sérialisation entre T_j et T_k tel que l'écriture de T_j ne se superpose pas avec l'écriture de T_k. Il est également appelé ajustement en avant.

Le protocole OCC-BC

Haritsa et al. (1992) ont proposé une variante de l'algorithme OCC-BC qui utilise la certification en avant et qui tient compte des priorités des transactions. Ces priorités sont déterminées à partir des échéances des transactions. Considérons une transaction T_j en phase de validation. Le déroulement de la validation de T_j est décrit par l'Algorithme 2.
L'intérêt du protocole OCC-BC est que la transaction T_j se met en attente et garde les ressources qu'elle avait acquises. Ceci peut améliorer les performances du système, puisque la transaction n'aura pas à redemander ces ressources. L'inconvénient cependant est que les transactions abandonnées (ligne 3) sont redémarrées plus tard, ce qui peut impliquer un gaspillage

de ressources si finalement T_j ne respecte pas son échéance. Un autre inconvénient provient du fait que T_j détient des ressources alors qu'elle est bloquée (ligne 6), ce qui peut induire une augmentation des risques de conflits entre les transactions.

Algorithme 2 : Principe du protocole OCC-BC

```
1: if T_j en validation And T_j en conflit avec T_1, T_2, ...T_n then
2:     if priorité(T_j) > priorité(T_k) pour k ∈ [1, n] et k ≠ j then
3:         Abandonner T_k pour k ∈ [1, n]
4:     end if
5:     if ∃T_k pour k ∈ [1, n] And priorité(T_j) < priorité(T_k) then
6:         T_j attend la terminaison de T_k
7:     end if
8: end if
```

Le protocole OCC-WAIT

Il s'agit d'une variante du protocole OCC-BC proposée par Haritsa et al. (1990). Une transaction qui atteint sa phase de validation et qui trouve des transactions de plus hautes priorités dans son ensemble de conflits est bloquée. Cette période d'attente doit permettre aux transactions de priorités plus élevées de se terminer avant leur échéance. Contrairement à OCC-BC, le problème du gaspillage de ressources (cf. Algorithme 2, ligne 3) ne peut pas survenir car la transaction en attente n'est redémarrée par aucune autre transaction de plus haute priorité dont l'échéance n'a pas été respectée. Ainsi, une transaction T_j ne pourrait valider que lorsque toutes les transactions de plus haute priorité manquent leur échéance. Cependant, le protocole OCC-Wait ne résout pas les problèmes du protocole OCC-BC concernant le blocage des transactions pendant la phase de validation et la détention des ressources (cf. Algorithme 2, ligne 6).

Le protocole WAIT-50

Haritsa et al. (1992) ont amélioré le protocole OCC-Wait, en ajoutant un contrôle de l'attente des transactions. Une transaction T_j en validation doit se mettre en attente jusqu'à ce que moins de 50% des transactions actives T_k ($k \in [1, n]$ et $k \neq j$) aient des priorités supérieures à la sienne. Une fois ce pourcentage atteint, les transactions T_k restantes sont abandonnées sans tenir compte de leur priorité, et la transaction T_j réalise sa validation.
Cette méthode peut être généralisée en acceptant que T_k attende jusqu'à ce que X% des transactions ayant des priorités supérieures se terminent. Cette méthode est dite de WAIT-X. Cependant, il a été prouvé par les mêmes auteurs, que la classe 50% offre les meilleures performances.

Le protocole spéculatif SCC

Le protocole SCC est un protocole hybride spéculatif (*Speculative Concurrency Control*) proposé par Bestavros (1992). Il combine à la fois les stratégies des protocoles pessimistes et celles des protocoles optimistes. Lorsqu'un conflit est détecté, une transaction fantôme est créée. Le rôle de la

transaction fantôme est de sauvegarder le point de conflit pendant que la transaction principale continue de s'exécuter. Si la transaction principale doit être abandonnée pour résoudre le conflit, alors la transaction fantôme est libérée. La transaction est alors redémarrée depuis le point de conflit mémorisé par la transaction fantôme et non depuis le début de la transaction. Les chances des transactions de respecter leur échéance sont donc plus importantes.
Le problème posé dans ce protocole est celui des performances globales du système. En effet, le nombre de transactions dupliquées est proportionnel au nombre de conflits, conduisant à des situations de surcharge. Une solution alternative à ce problème consiste à considérer un nombre limité de transactions fantômes par transaction principale. On obtient alors différentes classes de protocoles appelées SCC-kS (*SCC-k Shadow*). Parmi ces classes, la plus utilisée est SCC-2S (SCC-*Two Shadow*) où les transactions ont au plus une transaction principale et une transaction fantôme.

Le protocole OCC-RTDATI

OCC-DATI

La version classique (non temps réel) du protocole DATI (*Dynamic Adjustement using Timestamp Intervals*) (Lindström et Raatikainen 1999) améliore les protocoles OCC-BC et OCC-Wait, en minimisant le nombre de redémarrages inutiles des transactions. Pour cela, le protocole DATI se base sur l'ajustement de l'ordre de la sérialisation en faisant doter chaque transaction d'un intervalle d'estampilles. Cet ajustement est seulement effectué en phase de validation, et le fait d'utiliser un intervalle d'estampilles permet l'application à la fois un ajustement en avant et en arrière. Pour les transactions en conflit, l'ajustement de la sérialisation n'est appliqué que sur les transactions pour lesquelles il y a certitude qu'elles vont être validées.
Au premier lancement d'une transaction T_j, celle-ci obtient un intervalle d'estampilles $TI(T_j) = [0, \infty[$ (tout l'espace d'estampille). En phase de validation, $TI(T_j)$ est réévalué selon l'opération (lecture ou écriture) appliquée sur l'objet. C'est ainsi que pour un objet $D_j \in RS(T_j)$ l'intervalle $TI(T_j) = TI(T_j) \cap [WTS(D_j), \infty[$, tel que $WTS(D_j)$ est l'instant de la dernière écriture de l'objet D_j. Si par contre $D_j \in WTS(T_j)$ alors l'intervalle $TI(T_j) = TI(T_j) \cap [WTS(D_j), \infty[\cap [RTS(D_j), \infty[$ tel que $RTS(D_j)$ est l'instant de la dernière lecture de l'objet D_j. À la fin des réévaluations de $TI(T_j)$, si celui-ci s'avère nul, la transaction T_j est alors redémarrée.
Suivant les cas donnés dans la définition 2.1, l'intervalle $TI(T_k)$ est ajusté. Pour l'ajustement en avant, l'intervalle d'estampille de la transaction en conflit T_k devient $TI(T_k) = TI(T_k) \cap [TS(T_j) + 1, \infty[$ tel que $TS(T_j)$ est la valeur minimum de l'intervalle $TI(T_j)$. Dans le cas d'un ajustement en arrière, l'intervalle $TI(T_k) = TI(T_k) \cap [0, TS(T_j) - 1]$. Si à la fin des réajustements la valeur de $TI(T_k)$ devient nulle, la transaction T_k est alors redémarrée.
Une fois ces opérations effectuées, les dates $RTS(D_j)$ et $WTS(D_J)$ sont mises à jour, et la transaction T_j est validée.

OCC-RTDATI

Lindström et Raatikainen (2000) ont proposé le protocole OCC-RTDATI, en se basant sur leur précédent protocole DATI. Le protocole OCC-RTDATI est une adaptation du protocole DATI au contexte temps réel en faisant doter les transactions d'une échéance. Cette échéance est utilisée pour la prise de décision lors de l'application de l'ajustement en avant/arrière de la transaction en phase de validation. Dans les deux cas d'ajustement, la transaction T_j est redémarrée si et seulement si la transaction active T_k a son échéance plus proche que celle de T_j. Si T_k est loin de son échéance - donc moins prioritaire - elle n'est alors redémarrée que dans le cas d'un ajustement en arrière. Le reste du traitement de la validation, reste identique à celui décrit dans DATI.
Les mêmes auteurs ont proposé une autre variante nommée OCC-PDATI et qui se base sur la priorité des transactions en la définissant selon la criticité de la transaction. Le protocole PDATI offre une souplesse par rapport à RTDATI car à la différence de RTDATI, OCC-PDATI ne redémarre jamais une transaction active même si elle est plus prioritaire.

Conclusion

Les protocoles de contrôle de concurrence ont été abordés dans ce chapitre selon leur nature et dans l'ordre chronologique (améliorations apportées). Le tableau 2.1 synthétise ces protocoles.
Une notion importante a été abordée par le protocole optimiste OCC-RTDATI qui est l'ordre de la sérialisation, en intégrant la priorité des transactions. Le protocole OCC-RTDATI améliore ses prédécesseurs en réduisant le nombre inutile d'abandons de transactions. Ce protocole conçu pour des SGBDs temps réel reste strict quand il s'agit d'annuler des transactions actives de faible priorité. Ceci peut conduire à un gaspillage de ressources si finalement la transaction de plus haute priorité non encore validée manque son échéance. Le protocole OCC-PDATI procède d'une manière plus souple en favorisant la transaction la plus prioritaire tout en en respectant des règles de sérialisation.
Dans les deux cas, le parallélisme est cependant peu exploité dans la mesure où la sérialisation des transactions part de l'hypothèse que la phase de validation n'est pas réentrante (c'est-à-dire qu'une et une seule transaction peut être présente en phase de validation.) Par ailleurs, les autres protocoles abordés sont soit coûteux en ressources mémoire, soit ils offrent de faibles performances globales, ce qui ne favorise pas leur implémentation dans le cas des transactions résidant en mémoire. Ces dernières feront l'objet du prochain chapitre.

	Pessimistes	Optimistes
SGBDs classiques	2PL	OCC-BC OCC-DATI
SGBDs temps réel	2PL-HP 2PL-WP	OCC-BC temps réel OCC-WAIT-X OCC-RTDATI

Table 2.1 – *Protocoles de contrôle de concurrence*

Les Mémoires Transactionnelles

3

Sommaire

3.1 Introduction aux mémoires transactionnelles

Le mot *mémoire transactionnelle* (TM) a été introduit par Herlihy et Moss (1993) et désigne une manipulation des transactions en mémoire. Autrement dit, à la différence des SGBDs, une transaction est dépourvue du paramètre de durabilité dans la mesure où les données issues du traitement des transactions ne sont pas enregistrées sur support pour une durée indéterminée dans le temps. Cependant, l'idée de mettre une abstraction similaire à celle des SGBDs dans un langage de programmation permettant d'assurer la consistance a été étudiée par Lomet (1977) sans pour autant qu'il fournisse une implémentation pratique. Cette dernière a donc été proposée par Herlihy et Moss (1993). L'intérêt de cette implémentation a été renouvelée par l'apparition de la technologie du multicœur et la volonté d'industrialiser les systèmes transactionnels (Tremblay et Chaudhry Feb. 2008). Du point de vue du programmeur d'applications, l'introduction de cette technologie permet de lui faciliter la programmation en lui évitant de se soucier des problèmes liés à la synchronisation par verrous. Ces derniers entraînent en effet des problèmes comme les interblocages et l'inversion de priorités, ainsi que la difficulté à composer du code de programmation modulaire.

3.2 Taxonomie des mémoires transactionnelles

3.2.1 Transactions imbriquées

Une transaction imbriquée est celle dont l'exécution est contenue dans une autre transaction appelée principale. Les données d'une transaction principale sont accessibles par sa transaction imbriquée. Une transaction est dite *aplatie* si l'annulation de la transaction imbriquée cause également l'annulation de la transaction principale. Par contre, une terminaison avec succès de la transaction imbriquée ne prend effet que si la transaction principale se termine aussi avec succès.
Les transactions non aplaties, sont classées en deux grandes catégories. La première est dite *fermée* et diffère du mode aplatie uniquement dans la situation dans laquelle la transaction imbriquée est annulée. En effet, l'annulation d'une transaction imbriquée en mode fermé n'influe pas sur la transaction principale. La deuxième catégorie est dite *ouverte* quand les données de la transaction imbriquée sont visibles pour l'ensemble des transactions, dès lors qu'elle se termine avec succès, et ce, même si la transaction principale est toujours en cours d'exécution.
Moss et Hosking (2006) ont proposé un modèle de transactions imbriquées dans lequel ils introduisent une définition précise des transactions ouvertes et fermées.

3.2.2 Granularité des transactions

La granularité des transactions désigne l'unité de stockage à travers laquelle le système de mémoire transactionnelle détecte les conflits. Les TMs font généralement une extension du concept de l'orienté-objet pour implémenter la granularité objet servant d'unité de base pour l'accès aux

données, mais aussi pour permettre la détection de conflits lors d'accès multiples par plusieurs transactions. D'autres TMs considèrent une granularité plus fine comme le mot mémoire (*word*).

3.2.3 Faible et forte isolation

Blundell et al. (2005) ont introduit les termes d'atomicité faible et forte. La forte atomicité garantit une séparation d'actions et de sémantique entre le code transactionnel et non-transactionnel. Cependant, ces termes sont utilisés dans Larus et Rajwar (2007) sous d'autres appellations, car considérés non-appropriés à la véritable signification de l'isolation. Ainsi, le terme isolation faible signifie qu'un conflit de référence mémoire se trouvant en dehors de la transaction ne doit pas suivre les protocoles du système de TM.

3.3 Gestion des conflits

Les transactions s'exécutant d'une manière concurrente, ont besoin de se synchroniser à la fois pour les types de mises à jour d'objets directe et différée. Dans une mise à jour *directe* les objets peuvent être modifiés dès que les transactions y apportent des changements, ce qui n'est pas le cas dans le mode différé, où il faut attendre la phase de validation (commit) de la transaction manipulatrice pour une éventuelle mise à jour.

Détection des conflits

Un conflit entre deux transactions apparaît sur un objet donné quand celui-ci est manipulé par des opérations de lecture/écriture. Au moins une des transactions doit modifier l'objet en écriture dans la mesure où les opérations de lecture ne peuvent pas provoquer de conflit entre elles. Lorsqu'un un conflit apparaît, le système de gestion de conflit d'une mémoire transactionnelle doit être capable de déterminer quelles sont les opérations qui sont en conflit. La détection suivant les deux cas de mise à jour présentés plus haut, se fait soit d'une manière directe ou différée.

Types de conflits

Une fois le conflit détecté, la résolution se fait suivant la nature de la détection (directe/différée). Scott (2006) a classé les différents types de résolution de conflits et il distingue quatre types :

- *Invalidation lazy.* Les transactions T_A et T_B sont en conflit si T_A écrit un objet, T_B lit le même objet, et T_A est validée avant T_B.
- *Eager Écriture-Lecture E-L.* Les transactions T_A et T_B sont en conflit si T_A et T_B ont un conflit de type lazy, ou bien, T_A écrit un objet et T_B lit par la suite le même objet, et qu'aucune de ces transactions n'est entrée en validation.
- *Invalidation mixée.* Les transactions T_A et T_B sont en conflit si T_A et T_B ont un conflit de type lazy, ou bien, T_A écrit un objet et T_B lit et écrit par la suite le même objet, et qu'aucune de ces transactions n'est entrée en validation.

– *Invalidation Eager.* Les transactions T_A et T_B sont en conflit si T_A et T_B ont un conflit de type Eager *E-L*, ou bien, T_B lit un objet et T_A écrit par la suite le même objet, et qu'aucune de ces transactions n'est entrée en validation.

3.3.1 Gestionnaire de contention

Dans les mémoires transactionnelles, le conflit entre deux transactions peut se résoudre en annulant l'une d'elles. Le *gestionnaire de contention* est typiquement sollicité par certaines TM lors de l'apparition de conflits. Plusieurs politiques sont implémentées au sein du gestionnaire de contention. Le choix d'une de ces politiques peut avoir une influence directe sur les performances du système. Malgré le fait que ces politiques ne garantissent pas toutes l'équité du système, leur stratégie commune tend toutefois à maximiser le facteur de progression des transactions par unité de temps. Scherer et Scott (2005) ont montré qu'il n'existe pas de politique pour un gestionnaire de contention plus performante que d'autres. Une hiérarchisation du gestionnaire de contention a été proposée par Guerraoui et al. (2005) sous le nom de *polymorphe* et classent les gestionnaires de contention par facteur de leur charge dans le système.
Les politiques utilisées par le gestionnaire de contention annulent des transactions en vue de régler le conflit, mais ne garantissent pas *quand* ces transactions vont être relancées. Ce problème a été traité par Yoo et Lee (2008) en proposant un ordonnanceur des transactions basé sur une approche adaptative afin de déterminer d'une manière précise les instants de relance des transactions qui, auparavant avaient été abandonnées. Cette politique d'ordonnancement diffère donc des différentes stratégies utilisées par le gestionnaire de contention présentées ci-après (Scherer et Scott 2005).

Polite

Une transaction essayant de détenir un objet se met en attente pendant un laps de temps limité et prédéterminé. Ces laps de temps appelés fenêtres de contention, vont d'abord croître exponentiellement avant d'annuler la transaction détentrice de l'objet. La vérification quant à la disponibilité de l'objet s'effectue avant chaque régénération de la fenêtre de contention.

Karma

La plus haute priorité est attribuée à une transaction ayant en possession le plus d'objets. Une transaction de plus grande priorité peut directement annuler une autre transaction de moindre priorité si cette dernière détient des objets en conflit avec la première. Pour une transaction ayant une priorité N fois plus faible, elle doit réitérer N fois sur les objets avant de pouvoir annuler la transaction de plus haute priorité.

Eruption

Cette politique est similaire à Karma, sauf que la transaction de plus haute priorité est encore rehaussée en y ajoutant la priorité de la transaction de

plus faible priorité en concurrence sur les mêmes objets. Ce rehaussement permet l'évitement de la situation dans laquelle une troisième transaction peut annuler la transaction de plus haute priorité.

Kindergartin

Cette stratégie se base sur une liste de transactions qui est mise à jour pendant leur exécution. Cette liste contient les transactions qui ont auparavant causé l'annulation d'autres transactions en conflit. Si une transaction détenant des objets en conflits figure dans cette liste, elle est immédiatement annulée. Dans le cas où elle ne figure pas dans la liste, elle est d'abord rajoutée à la liste, et les autres transactions en conflit s'annulent après une attente générée par une fenêtre de contention exponentielle. De cette manière deux transactions partageant les mêmes objets, peuvent tour à tour être annulées.

Timestamp

Une estampille est associée à chaque transaction. Une transaction qui rentre en conflit avec une autre dont la date d'arrivée est plus récente que la première est directement annulée.

Published Timestamp

Idem à la stratégie Timestamp, avec en plus l'annulation des transactions qui apparaissent le plus souvent inactives.

Polka

Combinant les politiques Polite et Karma, une transaction tentant d'accéder à un objet, réitère N fois sur cet objet comme dans Karma, sauf qu'il y a utilisation de N intervalles de temps d'attente générés par une fenêtre de contention exponentielle.

3.3.2 Le Helping

Lors de la résolution de conflit, une transaction dans ce cas ne sollicite pas le gestionnaire de contention, mais "aide" la transaction avec laquelle elle est en conflit à pouvoir s'accomplir. Cette technique est utilisée non pas pour un mode de synchronisation en obstruction-free mais plutôt pour le lock-free et le wait-free afin d'assurer la progression des transactions. L'aide entre transactions est généralement mise en place d'une manière récursive en exécutant les routines de validation de la transaction à aider (Shavit et Touitou 1995, Fraser et Harris 2007).

3.4 Implémentations des mémoires transactionnelles

L'objectif des mémoires transactionnelles est double. Il s'agit d'une part d'arriver à obtenir un système de gestion de concurrence des transactions capable de garder un fort degré de parallélisme entre elles, pour pouvoir utiliser pleinement les ressources matérielles des technologies multicœurs.

D'autre part, il s'agit de permettre aux programmeurs d'applications à travers des langages et compilateurs dédiés (Felber et al. 2007), de pouvoir exploiter et manipuler aisément le parallélisme par une spécification sémantique transactionnelle simple (Harris et Fraser 2003).
Dans la littérature, trois types d'implémentations se distinguent pour atteindre ce double objectif. Ces implémentations que nous détaillerons dans les prochains paragraphes, sont soit matérielles, soit logicielles, ou hybrides.

3.4.1 Implémentations matérielles (HTM)

La première implémentation matérielle proposée par Herlihy et Moss (1993) vise à développer une structure de données lock-free afin d'augmenter les performances et d'éviter les problèmes liés aux verrous comme les situations d'interblocage et d'inversion de priorités. Pour ce faire, le processeur proposé possède de nouvelles instructions et un cache transactionnel afin de permettre la gestion et la mise en buffer de données transactionnelles.
La HTM de Herlihy et Moss dispose de trois primitives. La première concerne la lecture notée *LT reads* qui permet de lire une zone de mémoire partagée dans un registre privé. La seconde est notée *LTX* et fait la même chose que la première primitive, mais tolère en plus que la zone mémoire soit mise à jour. La dernière est la *ST* pouvant écrire directement en mémoire, mais les changements ne sont visibles que lorsque la transaction fait un commit.
L'avantage des implémentations matérielles est l'augmentation des performances. Cependant, les différentes architectures proposées dans la littérature (Ananian et al. 2005, Hammond et al. 2004, Rajwar et al. 2005, Moore et al. 2006, Chuang et al. 2006) se posent plus le problème de l'implémentation que celui des performances.

3.4.2 Implémentations logicielles (STM)

Afin d'offrir une meilleure flexibilité, une première implémentation logicielle a été proposée par Shavit et Touitou (1995) ayant une granularité fine et étant synchronisée par lock-free avec une résolution de conflit par aide. Les implémentations qui ont suivi proposent des optimisations afin d'améliorer les problèmes de performances dont souffrent les STMs. D'autres implémentations traitent de l'adaptation des STMs à des environnements distribués (Herlihy et Sun 2007) ou abordent des problèmes ouverts tels que l'intégration d'opérations d'entrées/sorties au sein des transactions.
Nous présentons ici les différentes implémentations logicielles les plus connues à ce jour dans la littérature. Chaque STM représente un type de résolution de conflits et de synchronisation particulier. En outre, ces implémentations supportent une architecture multiprocesseur mono-site. Pour les lecteurs intéressés par les STMs distribuées, ils peuvent se référer aux articles Manassiev et al. (2006), Herlihy et Sun (2007), Bocchino et al. (2008).

DSTM

Herlihy et al. (2003b) ont proposé la DSTM qui est une mémoire transactionnelle logicielle dynamique de type Eager qui résout l'allocation statique d'objets proposée par Shavit et Touitou (1995). La DSTM est basée sur une synchronisation obstruction-free. Les transactions dans la DSTM référencent un objet à travers une structure de données appelée *TMObject* et qui pointe sur une zone mémoire contenant un pointeur appelé *Locator*. Ce dernier contient la référence au descripteur de la transaction qui l'a créé. L'ancienne et la nouvelle version de l'objet sont également contenues dans un Locator. Le descripteur de la transaction possède une liste de tous les objets accédés en mode lecture par la transaction. Pour acquérir un objet accédé, un nouveau Locator privé à la transaction est créé avec un objet cloné. Si la phase du commit s'est accomplie avec succès, l'ensemble de lecture est validé et la structure TMObject est alors automatiquement mise à jour pour pointer vers le nouveau Locator. Lors d'un conflit, la DSTM n'établit pas de décision sur l'annulation d'une quelconque transaction, mais une interface générale est fournie permettant au gestionnaire de contention d'implémenter une variété de politiques.

OSTM (Object-based STM)

La OSTM est le nom donné à la STM proposée par Fraser et Harris (2007). Elle est basée sur le lock-free avec une résolution de conflit par l'aide de type Lazy. Elle a été implémentée par Fraser sous la forme d'une librairie écrite en langage C (Fraser 2003). L'objet est l'unité de base pour la concurrence. Chaque objet est pointé par un objet-entête qui contient la version courante de l'objet. Cet entête est pointé par un *gestionnaire d'objets* qui sauvegarde l'ancienne et la nouvelle référence de l'objet. Dans le cas où la transaction se termine avec succès, l'entête est mis à jour avec le nouveau bloc de données de l'objet. Le descripteur de la transaction contient à la fois la liste pour les objets manipulés en lecture seule et en lecture-écriture.

STM Ennals

Ennal (2006) a argumenté le fait que les STMs ne doivent pas être basées sur l'obstruction-free et il a proposé une STM permettant de sauver les objets en place. Ce type de sauvegarde évite par conséquent les indirections de pointeurs, lui permettant ainsi d'avoir de meilleures performances par rapport à la OSTM et la DSTM en termes de bande passante. Chaque transaction maintient des descripteurs de lecture et écriture séparément.
Pour écrire un objet, la transaction doit d'abord verrouiller l'objet de manière à ce qu'elle puisse être capable de créer une copie de travail de l'objet. Ce verrou n'est relâché que lors de la terminaison (avec succès ou abandon) de la transaction. Une transaction qui tente de lire un objet doit attendre jusqu'à ce que le gestionnaire d'objets ait un numéro de version. Par conséquent, la STM d'Ennal utilise un verrouillage à deux phases (2-PL) pour l'écriture et un contrôle de concurrence optimiste pour les lectures.

3.4.3 Implémentations hybrides (HyTM)

Il existe des implémentations hybrides qui permettent de combiner les avantages des deux implémentations précédentes (Kumar et al. 2006, Damron et al. 2006). D'une manière générale, dans une HTM, chaque processeur dispose d'un jeu d'instructions lui permettant de lancer des transactions, et de pouvoir les rejouer en cas de conflit. Selon l'implémentation, le contrôle est donné ou non au processeur en cas de conflit détecté sur sa transaction. Ce choix est lié au fait d'utiliser un mécanisme câblé ou logiciel lors de la résolution de conflits. Une implémentation HyTM intègre l'API d'une HTM mais à la différence de la HTM, elle permet de solliciter un gestionnaire de conflits logiciel (typiquement un gestionnaire de contention) voire une extension d'API logicielle.

3.4.4 Implémentations temps réel

La problématique pour les implémentations temps réel se rapproche de celle des SGBDs temps réel, qui consiste à garantir la sérialisation des transactions mais aussi le respect des contraintes de temps.
Comme vu précédemment, de nombreuses implémentations se basent sur des instructions de type CAS ou encore LL/SC, en considérant des plateformes monoprocesseur (Anderson et Moir 1995, Anderson et al. 1997b, Afek et al. 1997, Moir 1997). Sous certaines conditions souvent pessimistes, ces implémentations garantissent des contraintes temps réel strictes en se basant sur une synchronisation de type lock-free ou wait-free. Cependant, la détection de conflits et l'annulation des transactions par un gestionnaire dédié qui représente un aspect important dans les mémoires transactionnelles n'est pas traité.

Implémentations matérielles

Manson et al. (2005) ont proposé un mécanisme appelé *Preemptible Atomic Regions* (PAR) qui constitue une forme restrictive de mémoire transactionnelle pour les systèmes temps réel à contraintes strictes. PAR consiste en une séquence d'instructions s'exécutant de manière atomique. Si une tâche de plus grande priorité se réveille, PAR n'est pas exécuté et la tâche de plus grande priorité s'exécute sans aucune influence de la tâche de plus faible priorité qu'elle a préemptée. PAR est réexécuté lorsque la tâche de faible priorité reprend la main. De plus, l'annulation de PAR est aussi effectuée de la même manière lors d'une interruption. Par conséquent, le temps de blocage d'une tâche est dans le pire des cas égal à la plus longue section critique de la tâche de plus faible priorité. PAR est toutefois implémenté sur une architecture monoprocesseur.
Pour gérer la mémoire, PAR fait une pré-allocation d'objets en mémoire *immortelle* et le ramasse-miettes de java temps réel n'est donc pas utilisé. A noter qu'en général, ces ramasses-miettes engendrent d'importants overheads. Pour pallier ce problème, Higuera-Toledano et al. (2004) ont proposé une solution efficace à base d'instructions matérielles, en améliorant les mécanismes de barrières mémoires en lecture et en écriture.

La première mémoire transactionnelle temps réel à contraintes strictes supportant des systèmes multiprocesseurs a été proposée par Schoeberl et al. (2009). Cette proposition s'appuie également sur une supposition pessimiste dans l'évaluation du pire temps d'exécution (WCET). Chaque transaction possède un temps d'exécution borné au pire cas par un temps proportionnel au nombre de tâches présentes dans le système, dans la mesure où chaque travail d'une tâche a au plus une transaction à effectuer.

Implémentations logicielles

La seule implémentation logicielle temps réel connue pour les systèmes multiprocesseurs a été proposée par Holman et Anderson (2006). Cependant, et comme présenté précédemment, l'ordonnancement est restreint aux algorithmes basés sur des quantums de temps tel que l'algorithme Pfair.
Fahmy et al. (2009) ont montré qu'il est possible de garantir des contraintes temps réel dur pour des systèmes distribués avec tolérance aux fautes, où le contrôle de concurrence est géré par mémoire transactionnelle logicielle. Pour ce faire, les auteurs ont d'abord déterminé le temps de réponse au pire cas en faisant l'extension de l'analyse faite par Spuri (1996). Cette analyse permet de borner le temps de réponse des tâches en déterminant "l'instant critique" et la date de réveil des tâches qui sont susceptibles de provoquer le maximum d'interférences. Autrement dit, déterminer la période durant laquelle toutes les tâches sont simultanément réveillées. L'extension de cette analyse reste pessimiste puisqu'une tâche se réveillant au même instant que les autres, n'exécutera pas forcément une transaction. Ceci induit un faible taux de conflits entre transactions, mais aussi une diminution significative en terme de parallélisme dû à ce pessimisme.

Conclusion

Les concepts présentés dans ce chapitre possèdent plusieurs points communs avec ceux exposés dans les deux précédents chapitres. Sur la gestion de conflits, d'autres terminologies sont utilisées dans les SGBDs mais renvoient au même sens quand il s'agit de signifier par exemple la validation Lazy et Eager. Par ailleurs, les types de synchronisation présentés dans ce chapitre à savoir le lock-free et le wait-free, traitent de la même problématique exposée précédemment dans la partie temps réel. S'agissant de l'environnement temps réel, l'accent est porté dans la littérature sur des TMs temps réel à contraintes strictes, dont les tentatives consistent à chaque fois à borner le temps des rejoues des transactions suite à l'utilisation d'approches optimistes. Par conséquent, l'étude de contrôles de concurrence spécialement dédiés aux mémoires transactionnelles temps réel n'est pas traitée, car considérés comme étant destinés pour le contexte temps réel souple. Or les systèmes multimédia et de communication par exemple ont besoin à la fois d'exploiter le plein parallélisme et de respecter des contraintes temps réel souples.

Deuxième partie

STM pour les systèmes temps réel *soft*

4 Adéquation des STMs existantes aux systèmes temps réel soft

Sommaire

4.1 Introduction

Nous nous proposons dans ce chapitre d'étudier expérimentalement et comparativement, l'adéquation des mémoires transactionnelles aux systèmes temps réel multiprocesseurs. Il s'agit en particulier d'évaluer si la variabilité du temps d'exécution des transactions est prohibitif à une utilisation dans un contexte temps réel lors de l'accès aux ressources partagées.
Pour cela, nous utilisons les trois grandes catégories de synchronisation selon laquelle sont classées les mémoires transactionnelles, à savoir le lock-free représenté par la OSTM de Fraser et Harris (2007), l'obstruction-free typiquement désigné par la DSTM de Herlihy et al. (2003b), et enfin les verrous de type 2-PL représentés par la STM d'Ennal (2006). Ces mémoires transactionnelles sont évaluées conjointement avec les approches d'ordonnancement par partitionnement (P-EDF) et globales (G-EDF et PD^2).
Les principaux résultats de ce chapitre ont conduit aux publications suivantes : Sarni et al. (2009a;b;c).

4.2 Contexte d'évaluation

Intuitivement, pour évaluer les STMs en termes de variation du temps d'exécution des transactions, le système d'exploitation sous-jacent (OS) doit aussi être considéré puisque le temps d'exécution au sein d'un OS classique n'est pas prédictible. L'OS doit donc opérer en temps réel et être déterministe. Nous discutons du choix dans ce chapitre.
En outre, les STMs modernes gèrent dynamiquement l'accès à la mémoire lors de l'allocation et de la libération d'objets. Nous présentons également dans ce chapitre le choix qui a été fait sur l'allocateur mémoire pour mettre en évidence son degré d'influence sur la variation du temps d'exécution des transactions.

4.2.1 Plateforme matérielle

La plateforme matérielle utilisée dans nos expérimentations, repose sur un processeur multicœur Intel Core (TM) 2 Duo T7500 cadencé à 2.20Ghz, muni d'un cache L2 de 4Mo et de 3.5Go de mémoire RAM.

4.2.2 Configuration logicielle

Système d'exploitation

Nous avons sélectionné le système Linux (une distribution Ubuntu 8.04 et une version du noyau 2.6.24) pour représenter l'environnement classique dans nos expérimentations. Pour l'environnement temps réel, nous avons sélectionné l'OS temps réel (RTOS) nommé $LITMUS^{RT}$ (Linux Testbed for Multiprocessor Scheduling in Real-Time systems) (Calandrino et al. 2006). Ce dernier a été spécialement conçu pour fonctionner sur des architectures multiprocesseurs symétriques (SMP) et implémente toutes les politiques d'ordonnancement temps réel multiprocesseur décrites dans le premier

chapitre.
$LITMUS^{RT}$ est basé sur un noyau de système d'exploitation Linux classique (la même configuration que celle citée plus haut). Les ordonnanceurs proposés sont implémentés comme des composants qui peuvent être sélectionnés depuis l'espace utilisateur de Linux. Afin de manipuler à la fois les tâches et les mécanismes de synchronisation depuis l'espace utilisateur de Linux, des appels système sont rassemblés dans une libraire écrite en langage C. Pour toutes ces raisons, $LITMUS^{RT}$ devient un excellent (peut être le seul) candidat pour étudier le comportement des STMs, sur un système multicœur sous un panel de politiques avancées d'ordonnancement temps réel.
La librairie LITMUS^{RT}. Nous avons enrichi la librairie LITMUS^{RT} par des STMs (la OSTM, la DSTM et la STM d'Ennal), et ce, de manière à ce que l'on puisse lancer des transactions au niveau de chaque tâche temps réel.

Benchmarks transactionnels

Des micro-benchmarks synthétiques tels que des ensembles d'entiers sont souvent utilisés pour évaluer les performances des STMs. Certains reposent sur l'accès à des structures de données plus ou moins complexes : listes à enjambements (*skip lists*), arbres rouge et noir (*red-black trees*), conteneurs (*hash sets*), ou listes chaînées (*linked lists*). Tous manipulent un ensemble d'entiers.
L'exécution de tels micro-benchmarks consiste à la fois en des transactions de lecture qui déterminent si un élément est présent dans l'ensemble (i.e. *lookup*), et des transactions d'écriture (ou mise à jour) qui, soit ajoutent (i.e. *update*), soit suppriment (i.e. *remove*) un élément. La taille de l'ensemble est maintenue constante grâce à une alternance des insertions et suppressions. Pour notre évaluation, nous avons retenu les arbres rouge et noir. Une opération de rééquilibrage de l'arbre peut modifier de nombreux éléments et causer des conflits vis-à-vis de nombreuses opérations concurrentes. Ces arbres utilisent des structures de données conçues de telle manière qu'il est possible d'accéder à n'importe quel élément en traversant seulement un petit nombre d'éléments. Les mises à jour étant donc rapides, les transactions considérées sont courtes comparés à d'autres structures de données.

Profil de l'application temps réel

Dans $LITMUS^{RT}$, avant d'être lancée, une tâche temps réel sporadique est initialement créée comme étant un *thread* standard Linux (en utilisant la librairie standard *pthread*). L'environnement temps réel est par la suite initialisé, et les paramètres de la tâche temps réel, à savoir C_i et P_i sont spécifiés (par défaut, $D_i = P_i$). Ainsi, à chaque P_i unités de temps, le thread représentant la tâche, fait démarrer ses travaux en appelant la fonction associée au travail. Les travaux de chaque tâche exécutent des transactions d'une manière continue afin de maximiser le nombre de conflits entre transactions. Ceci dans le but de tenir compte de l'influence du contrôleur de concurrence des STMs sur le temps d'exécution des transactions opérant sur l'arbre rouge et noir.

Les transactions de chaque tâche réalisent 25% d'opérations d'écriture (i.e. *upadate* et *remove*) et 75% d'opérations de lecture (i.e. *lookup*). Fraser (2003) a montré qu'une telle répartition des opérations était observée dans la grande majorité des applications. Le nombre de feuilles maximum pour l'arbre rouge et noir a été fixé à 2^4, ce qui implique une forte contention pour l'accès aux ressources partagées.
Les tâches temps réel ont chacune une durée d'exécution $C_i = 20ms$ et des périodes telles que $\sum \frac{C_i}{P_i} = M$. Un système fortement chargé (i.e. $U = 100\%$) est considéré pour engendrer une forte contention entre transactions. Le nombre de tâches compris entre 2 et 32, est un paramètre d'entrée de l'évaluation.
La durée de l'application a été fixée à $10s$. Dans le cadre de l'évaluation des STMs effectuée par Fraser (2003), cette durée est considérée comme suffisante pour stabiliser les données dans le cache. Le nombre de transactions générées durant le test est de l'ordre de 7×10^6.

Allocateur mémoire

L'allocateur *malloc* est par défaut utilisé dans toutes les STMs testées. Parallèlement, nous avons sélectionné l'allocateur mémoire TLSF (Two-Level Segregate Fit) de Masmano et al. (2004) pour l'utiliser lorsque nous évaluons l'impact de l'allocation d'objets sur l'analyse de la variabilité du pire temps d'exécution observé au niveau des transactions. TLSF est basé sur un algorithme qui a un coût constant $\Theta(1)$. Ainsi, il résout le problème de la borne du pire cas d'exécution, maintenant alors l'efficacité des opérations d'allocation et de libération de la mémoire. De ce fait, TLSF permet une utilisation raisonnable de la mémoire dynamique dans les applications temps réel.
TLSF est une librairie écrite en C. Nous l'avons intégrée au sein de la librairie de Fraser (2003) en remplaçant toutes les fonctions d'allocation et de libération mémoire par celles fournies par TLSF. L'espace mémoire (en anglais, *pool*) utilisé par TLSF est créé au moment de l'initialisation par la fonction classique d'allocation mémoire, à savoir *malloc()*. A noter que l'initialisation de TLSF est faite avant la création des *threads* temps réel.

4.2.3 Métriques étudiées

Dans la littérature, on retrouve trois types de paramètres pour effectuer l'analyse du temps d'exécution de tâches ou de transactions temps réel, à savoir $WCET$, $BCET$ et $ACET$.
Le $WCET$ (Worst-Case Execution Time) est défini comme étant la durée au pire-cas. Le $BCET$ (Best-Case Execution Time) correspond à la durée d'exécution la plus courte. Enfin, le ACET (Average-Case Execution Time) est une valeur comprise entre le $WCET$ et le $BCET$, et dépend du contexte d'exécution.

Gigue sur le temps d'exécution des transactions

Nous avons considéré pour chaque transaction la mesure de la gigue d'exécution, notée $ACET_{jitter}$, et définie de la manière suivante :

$$ACET_{jitter} = WCET_{mesure} - BCET_{mesure} \tag{4.1}$$

Les résultats reportés figurent le relevé de la gigue maximale observée parmi les N transactions exécutées. Elle se définit par :

$$WCET_{jitter} = Max_N\{ACET_{jitter}\} \tag{4.2}$$

Nous avons également considéré pour l'ensemble des tâches τ un facteur de variation du $WCET$ associé aux trois opérations effectuées par toutes les transactions sur l'arbre rouge et noir. Ce facteur est noté V, et il est défini de la manière suivante :

$$V = \frac{\overline{x}}{\sigma} \tag{4.3}$$

où $\overline{x}$ et σ représentent respectivement la moyenne et l'écart type des durées d'exécution sur l'ensemble des opérations effectuées sur l'arbre rouge et noir.

Technique de mesure de la durée d'une transaction

Pour mesurer la durée totale d'une transaction T_k, nous avons récupéré les tops horloge du processeur à l'aide de l'instruction assembleur *rdtsc*.
La durée d'exécution de T_k est obtenue en soustrayant sa date de lancement de la valeur renvoyée par l'instruction *rdtsc* au moment de la terminaison avec succès de T_k.
Cependant, cette technique de mesure en mode utilisateur est techniquement aberrante. En effet, si la transaction T_k démarre sur un cœur et migre lors de son exécution sur un autre cœur sur lequel elle fait son *commit*, la mesure devient alors invalide puisque les tops horloge des cœurs ne sont pas synchronisés.
Nous avons proposé une alternative (voir Algorithme 3) qui consiste à tenir compte de l'identifiant du cœur sur lequel la transaction a démarré son exécution. Cet identifiant est renvoyé par l'instruction en assembleur *cpuid* puis rajouté au sein du contexte de la transaction. Avant d'enregistrer le CPUID, il faut être toutefois sûr qu'il correspond à la valeur prélevée par *rdtsc* (voir ligne 6) car les instructions ne sont pas exécutées atomiquement.
Si la migration de la transaction survient plus de 2 fois durant le test, la boucle est arrêtée (ligne 7). Selon l'état dans lequel le test est effectué, soit le programme est annulé au démarrage de la transaction (ligne 9), soit le test est considéré comme *incorrect* au moment du commit (ligne 11). A la fin de l'expérience, si le nombre de tests *incorrects* est supérieur à 1% du nombre total de transactions, l'expérience est alors manuellement redémarrée.
A noter que nous avons mesuré le temps d'exécution moyen de l'Algorithme 3 qui est de 0.5μs. Le pire cas d'exécution de cet algorithme est de 2μs, c'est-à-dire, 2 x 0,5 au démarrage de la transaction, s'ajoutant 2 x 0.5 au moment du commit). Par conséquent, la précision de la durée d'exécution de T_k est dans l'intervalle $[1,2]\mu$s.

Algorithme 3 : Mesure de la durée d'exécution de T_k

```
1: init RetryCPU ⇐ 2
2: T_k.coreID ⇐ CPUID()
3: repeat
4:     RetryCPU ⇐ RetryCPU − 1
5:     T_k.Start ⇐ ReadProcessorTicks()
6: until T_k.coreID = CPUID() Or RetryCPU = 0
7: if RetryCPU = 0 then
8:     if state = TransactionStarting then
9:         Abort()
10:    else
11:        BadTest ⇐ BadTest + 1
12:    end if
13: end if
```

4.3 Évaluations expérimentales

Dans les sous-paragraphes qui suivent, nous reportons dans un premier temps les résultats comparatifs entre la OSTM, la DTSM et la STM d'Ennals, en termes de variabilité des temps d'exécution des transactions, et ce, sous des environnements classique (Linux) et temps réel ($LITMUS^{RT}$). Pour le cas temps réel, le choix de la politique d'ordonnancement est également étudiée. Enfin, pour la STM offrant les meilleures performances, nous montrons le degré d'influence de l'allocateur mémoire sur la variabilité des temps d'exécution des transactions.

4.3.1 Influence de l'OS

L'objectif de cette expérience est de montrer comment le système d'exploitation sous-jacent peut faire varier les durées d'exécution des transactions. Pour couvrir les trois principales catégories d'STMs, nous avons comparé la OSTM de Fraser *et al.*, avec la STM basée sur les verrous d'Ennals et avec la STM de type obstruction-free de Herlihy *et al.*

Résultats sous Linux. La Figure 4.1 montre que la STM d'Ennals ne passe pas à l'échelle. Ceci est dû aux fréquentes collisions de transactions, mais aussi au fait qu'une transaction peut attendre pour une longue durée avant d'avoir accès aux ressources concurrentes. Cette longue attente peut être expliquée, d'une part, par le fait qu'une transaction est bloquée durant sa phase du commit quand ses ressources sont détenues par une autre transaction. D'autre part, la STM d'Ennals place une restriction sur le nombre de transactions qui ne peuvent pas dépasser le nombre de cœurs $|M|$ à n'importe quel moment. Cette restriction est faite afin d'utiliser pleinement tous les cœurs. D'ailleurs, ce résultat confirme ceux obtenus par Dice et al. (2006) dans lesquels l'algorithme d'Ennals présente de mauvaises performances. Par conséquent, la STM d'Ennals n'a pas été considérée dans le reste des expérimentations.

Résultats sous LITMUS^{RT}. La Figure 4.2 montre que la OSTM et la

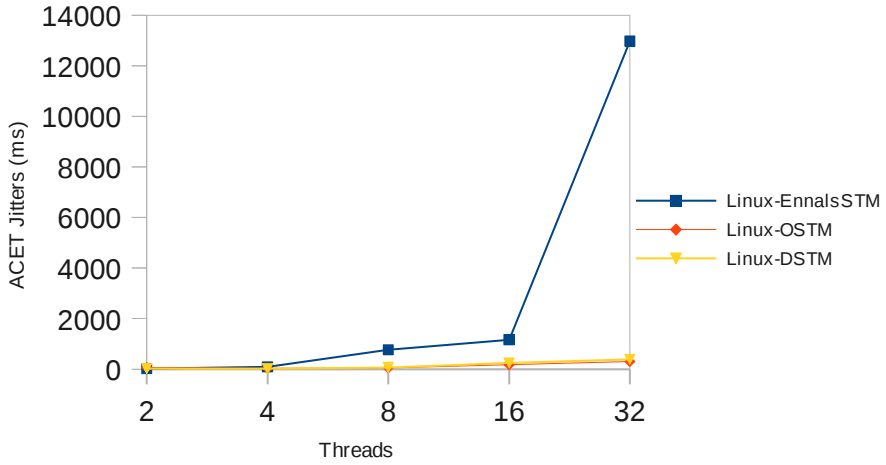


FIGURE 4.1 – *Scalabilité sous Linux*

DSTM se comportent mieux sous un système temps réel. Les $WCET_{jitters}$ sont plus importants sous Linux, et sont dus aux temps de préemption causés par l'interférence avec d'autres applications. Mais sous $LITMUS^{RT}$, les threads qui exécutent le test ont la plus grande priorité, et ne peuvent donc pas être préemptés par les processus Linux.

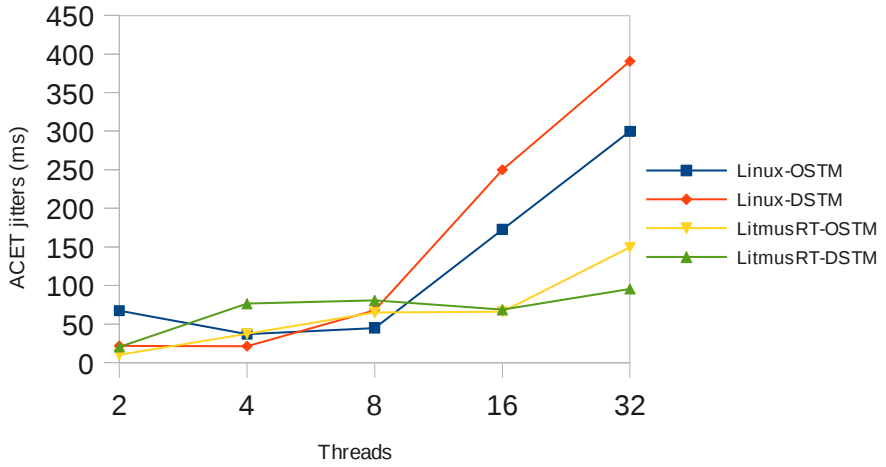


FIGURE 4.2 – *Scalabilité sous $LITMUS^{RT}$*

4.3.2 Influence des politiques d'ordonnancement

Comme la OSTM et la DSTM passent mieux à l'échelle sous un environnement temps réel, le reste des expérimentions a donc été conduit uniquement avec des politiques d'ordonnancement de tâches temps réel, et ce, dans le but de déterminer la politique d'ordonnancement la mieux adaptée à ces STMs.

Sous la politique Pfair. Les Figures 4.3 et 4.4 montrent que sous Pfair, la OSTM et la DSTM offrent tous les deux, des performances médiocres. Ce résultat peut être expliqué par le fait que la politique d'ordonnancement

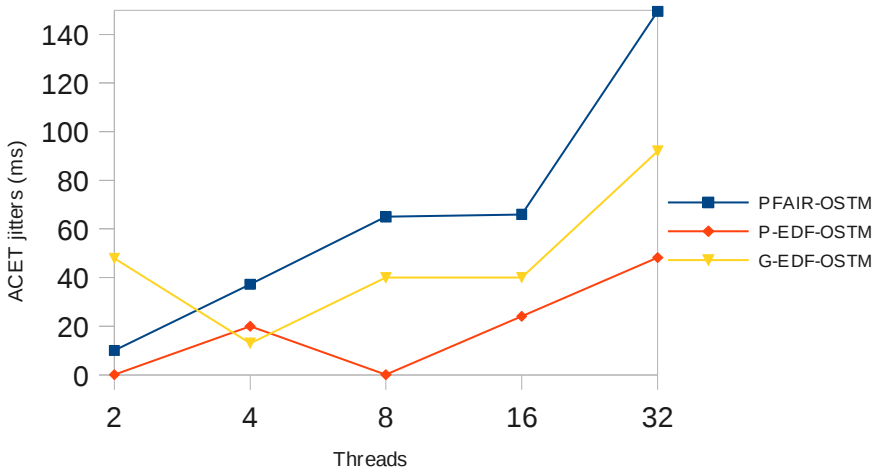


FIGURE 4.3 – *Scalabilité de la OSTM sous les politiques RT*

Pfair est plus complexe que celles basée sur les approches EDF, impliquant ainsi des *overheads* plus importants.

Sous la politique G-EDF. Contrairement à la OSTM, la DSTM passe

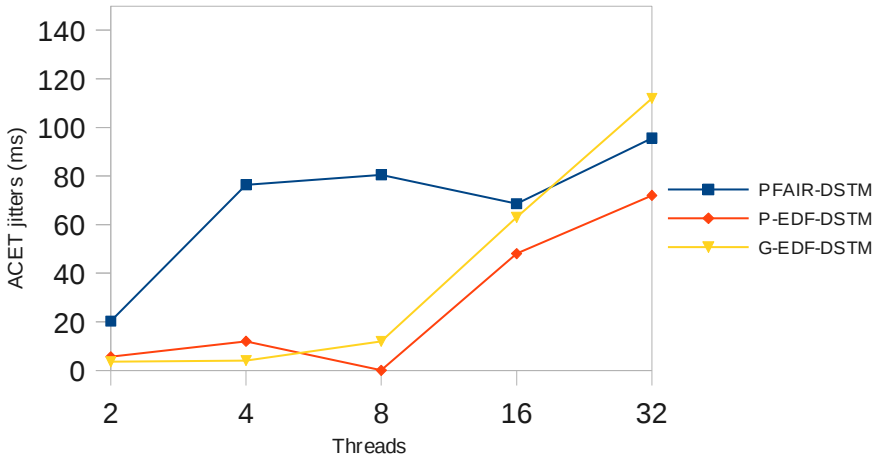


FIGURE 4.4 – *Scalabilité de la DSTM sous les politiques RT*

assez bien à l'échelle sous cette politique, quand le nombre de *threads* ne dépasse pas 8. Dans ce cas, la forte gigue observée sous la OSTM est causée par un surplus d'informations stockée en interne. Les données dans la pile sont donc plus importantes dans la OSTM. En conséquence, le coût de migration avec G-EDF est plus important et cause plus d'*overheads* dans la OSTM. Néanmoins, au delà de 8 threads, à la différence de la DSTM, la OSTM passe à l'échelle et le $ACET_{jitter}$ est relativement amorti.

Sous la politique P-EDF. La OSTM offre de meilleures performances que la DSTM, et P-EDF se révèle être la meilleure politique. En effet, il n'y a pas de coûts de migrations et les *overheads* sont réduits. En outre, le résultat obtenu ici présente certaines similarités avec les expérimentations conduites par Brandenburg et al. (2008), qui démontrent également que

les algorithmes lock-free passent mieux à l'échelle sous P-EDF. Par conséquent, les expérimentations suivantes ont été seulement conduites avec la OSTM sous la politique P-EDF.

4.3.3 Influence de l'allocateur mémoire

Il est souvent affirmé dans la littérature que les rejoues (*rollbacks* en anglais) des transactions sont les principales causes de l'indéterminisme au niveau des temps d'exécution des STMs. Cependant, les STMs récentes sont souvent basées sur des allocations mémoire dynamiques, ce qui peut donc rajouter de la variabilité aux temps d'exécution des transactions.
Nous démontrons dans cette section que l'allocation mémoire requiert plus de considération pour approximer la variabilité du temps d'exécution des transactions. Pour ce faire, nous avons utilisé la OSTM que nous avons retenue à l'issue des expérimentations précédentes. En outre, nous avons mesuré sous P-EDF, le temps d'exécution pour les trois opérations qu'effectue une transaction sur l'arbre rouge et noir, à savoir *lookup*, *update* et *remove*. Le facteur de variation du pire temps d'exécution V (voir 4.3), est obtenu sur 10 expérimentations. Nous avons préféré ici exprimer la variabilité du temps d'exécution par le facteur V au lieu du $ACETjitter$. Nous jugeons en effet plus commode d'exprimer l'influence de l'allocateur mémoire en pourcentage, plutôt que de donner l'ordre de grandeur du $ACETjitter$.

Résultats d'expérimentations

Nous rapportons ici les résultats obtenus en utilisant soit l'allocateur de mémoire classique *malloc*, soit l'allocateur mémoire TLSF.
La Figure 4.5 montre que la durée des transactions a une importante gigue pour les trois opérations. Bien que P-EDF soit utilisé, la OSTM a un important facteur de variation V qui caractérise l'environnement d'exécution à chaque lancement de programme. La OSTM utilise un ramasse-miettes que nous avons configuré de sorte à ce qu'il fonctionne en mode *minimal*. En effet, le mode *normal* fait que le programme s'arrête à cause des grands fragments (en anglais, *chunks*) imposés non seulement pour amortir le coût des allocations, mais aussi pour augmenter la densité de pointeurs par ligne de cache. Cependant, nous avons remarqué que ce mode de configuration du ramasse-miettes impacte sur la quantité totale de mémoire utilisée par la OSTM, et non pas sur le facteur V.
La véritable raison de cette variation est démontrée dans les Figures 4.6 et 4.7. Quand TLSF est utilisé à la place de l'allocateur classique *malloc*, le facteur de variation V reste relativement plus stable à partir de 4 *threads*. Plus important encore, la facteur de variation a significativement diminué en utilisant TLSF. En effet, la variation maximum qui est atteinte en utilisant *malloc* est d'environ 160% contre 8% dans le cas de TLSF.
Ceci montre que la OSTM pourrait satisfaire des contraintes temps réel molles, sous condition d'utiliser un temps constant pour l'allocation dynamique de la mémoire, comme c'est le cas avec TLSF.

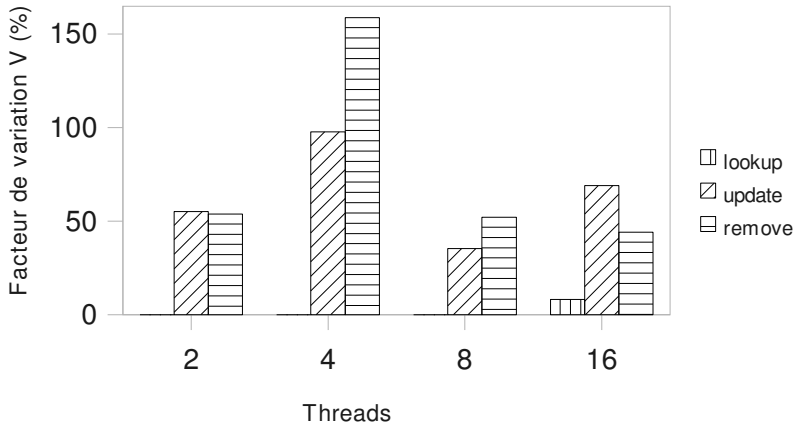


FIGURE 4.5 – *Variabilité du temps d'exécution des transactions avec malloc*

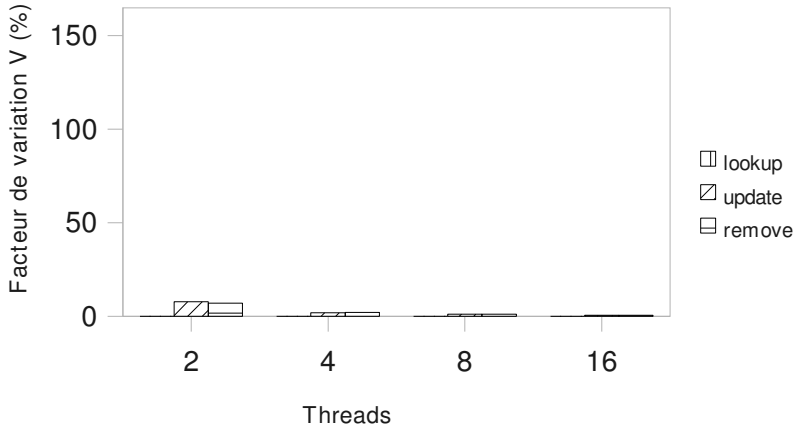


FIGURE 4.6 – *Variabilité du temps d'exécution des transactions avec TLSF*

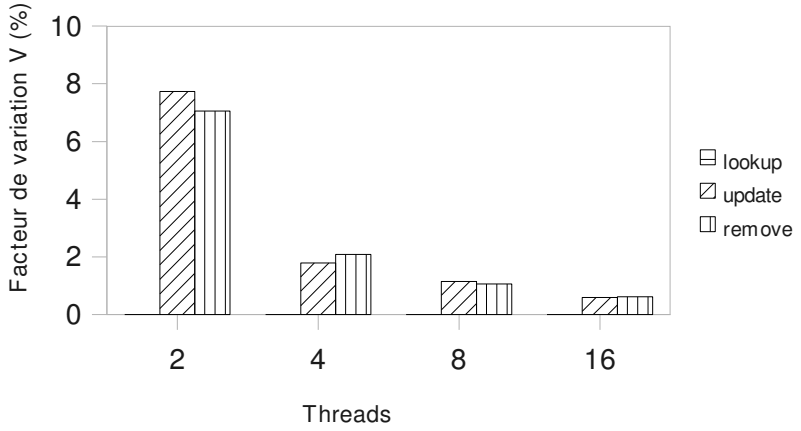


FIGURE 4.7 – *Zoom sur la Figure 4.6*

Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'adéquation des mémoires transactionnelles aux systèmes temps réel multiprocesseurs. Il s'agissait en particulier d'évaluer la variabilité du temps d'exécution des transactions lors de l'accès aux ressources partagées.
Dans un premier temps, nous avons effectué une étude comparative entre la OSTM, la DTSM et la STM d'Ennal sur des environnements classique et temps réel. Cette comparaison nous a conduit à sélectionner la OSTM comme étant la plus performante d'entre elles. En outre, nous avons montré que l'algorithme P-EDF est la politique l'ordonnancement temps réel qui présente expérimentalement les meilleures performances parmi G-EDF et PD^2.
Dans un second temps, nous avons montré qu'au sein de la OSTM, l'allocation mémoire requiert plus de considération pour approximer la variabilité du temps d'exécution des transactions. C'est pourquoi nous avons conforté les performances de l'allocateur mémoire classique *malloc* à celle d'une allocateur à temps d'allocation constant appelé TLSF.
Enfin, cette étude nous permet de conclure sur le fait que la synchronisation lock-free est plus adaptée au contexte temps réel soft, que le sont l'obtruction-free ou les verrous de type 2-PL.

Conception d'une STM temps réel (RT-STM)

5

Sommaire

5.1 Motivations et objectifs

L'un des premiers constats que nous avons dressés dans l'état de l'art, est que la synchronisation non-bloquante en multiprocesseur, bien qu'utile pour faire augmenter le parallélisme et éviter les problèmes liés aux verrous, reste particulièrement peu exploitée dans les systèmes temps réel. La difficulté réside dans le fait d'établir un test d'ordonnançabilité dans un environnement multiprocesseur en intégrant des transactions sous un optimisme total.
Les solutions existantes dans la littérature, pour le peu qu'il en existe, sont répertoriées en deux catégories indépendantes. La première catégorie s'appuie sur les ordonnanceurs de tâches temps réel mono ou multiprocesseurs. Quant à la deuxième, elle se présente sous forme de protocoles de gestion de concurrence dans les SGBDs temps réel.
Cependant, l'indépendance entre ces deux grandes catégories de solutions, rend de plus en plus difficile la programmation d'applications temps réel dont la complexité est grandissante.
Dans la première catégorie, la complexité due à la programmation concurrente est directement gérée par le programmeur. En effet, le concept de transaction est rarement utilisé, et lorsque celui-ci l'est, le contrôleur de concurrence reste dans une forme rudimentaire et la gestion des transactions est alors léguée en grande partie au programmeur.
Dans la seconde catégorie, les contraintes temps réel sont peu satisfaites dans la mesure où les contrôleurs de concurrence assurent la sérialisation des transactions mais sans pour autant intégrer les spécificités de l'environnement, à savoir l'ordonnanceur des tâches et la plateforme matérielle.

Dans les systèmes classiques, et comme nous l'avons vu précédemment, le concept de mémoire transactionnelle a l'avantage de faciliter la programmation en offrant au programmeur un plus haut niveau d'abstraction lui faisant ainsi éviter de se soucier de la programmation concurrente sous-jacente. Une mémoire transactionnelle intègre également le concept de transaction avec des contrôleurs de concurrence qui sont proches de leur environnement à la fois matériel et logiciel. Mais contrairement aux SGBDs temps réel, les mémoires transactionnelles sont pour la plupart non adaptées aux environnements temps réel puisqu'elles n'offrent pas la possibilité de spécifier des contraintes de temps.

Pour pallier ce problème, nous présentons dans ce chapitre une nouvelle solution pour les systèmes temps réel multicœurs à contraintes molles, à base de mémoire transactionnelle. Cette nouvelle solution introduit un modèle transactionnel temps réel pour les mémoires transactionnelles en partant à la fois de celui des tâches et des SGBDs temps réel. En se basant sur ce nouveau modèle, nous présentons la conception d'une mémoire transactionnelle logicielle temps réel nommée RT-STM qui est une amélioration de la OSTM. Le choix porté à cette dernière est en partie justifié dans le précédent chapitre quant à son adéquation aux systèmes temps réel soft. Nous détaillons son architecture dans le prochain paragraphe.
Par ailleurs, étant donné que le défi essentiel dans les STMs est de démontrer la possibilité de leur mise en pratique, le choix d'améliorer la OSTM,

représente donc un argument de plus. En effet, le premier objectif derrière la conception de la OSTM, est de démontrer la faisabilité pratique de son algorithme lock-free, et ce, sur diverses plateformes matérielles (Fraser 2003). Plus particulièrement pour la RT-STM, il s'agit non seulement de démontrer sa faisabilité pratique, mais aussi de prendre en compte les contraintes de temps de son environnement.
Les principaux résultats de ce chapitre ont été publiés dans Sarni et al. (2009b).

Architecture détaillée de la OSTM

La OSTM dispose de deux procédures principales pour ouvrir un objet en lecture et en écriture. Avant l'ouverture de l'objet en lecture, la transaction vérifie d'abord si elle ne l'a pas déjà ouvert auparavant. S'il s'agit d'une première ouverture, un gestionnaire d'objets est créé et inséré dans la liste de lecture-seule. Dans le cas d'une première ouverture en écriture, une copie *fantôme* de l'objet est créée et reste privée jusqu'à ce que la transaction se termine (voir Figure 5.1). La copie *fantôme* est directement retournée à la procédure appelante dans le cas où la transaction dispose déjà d'une copie de l'objet préalablement ouvert. Par contre, si l'objet est déjà ouvert en lecture, au moment de son ouverture en écriture, son gestionnaire est retiré de la file de lecture-seule, puis inséré dans la file de lecture-écriture. Ces deux principales procédures de lecture et d'écriture, utilisent toutefois la même sous-procédure de bas niveau afin de récupérer la version courante de l'objet. Si l'entête de l'objet n'est pas marqué comme étant *acquis* (techniquement effectuée par le mise à 1 de son bit de poids faible *LSB*), le contenu de cet entête est directement renvoyé à la procédure appelante. Ceci n'est pas le cas si le LSB est mis à 1 où le contenu de l'entête de l'objet est plutôt considéré en tant qu'adresse mémoire du descripteur de la transaction acquérant l'objet. Noter que cette acquisition d'objet signifie implicitement le fait que la transaction acquérante soit en première phase de terminaison (*commit*). Dans ce cas-ci, toute transaction qui tente d'ouvrir en parallèle l'objet acquis, va systématiquement procéder à l'aide de la transaction acquérante. Cette aide consiste à exécuter en parallèle la procédure du *commit* en prenant l'identité de la transaction acquérante. Une fois cette aide effectuée, le *commit* conduit au relâchement du LSB (mise à zéro).

Contrôleur de concurrence

Dans la OSTM, la phase de terminaison ou du phase du *commit* de la transaction est divisée en trois parties.
La première phase. Cette phase est celle de l'acquisition. La transaction essaye d'acquérir tous les objets qui sont dans sa file de lecture-écriture dans un ordre canonique (ordre croissant des adresses mémoire des objets, attribuées par l'allocateur mémoire). Pour ce faire, l'opération atomique de type CAS est effectuée sur l'entête de l'objet afin de remplacer le pointeur vers l'objet par l'adresse mémoire de son descripteur comme décrit précédemment. Si le contenu de l'entête de l'objet pointe sur une version plus récente de l'objet, la transaction échoue. Cependant, si l'objet

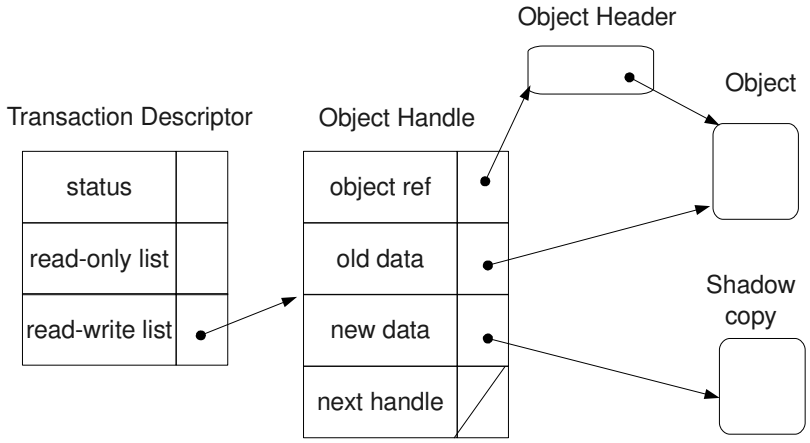


FIGURE 5.1 – *Structures de données de la OSTM*

est déjà acquis par une autre transaction qui est simultanément dans sa première phase du commit, cette dernière transaction est alors aidée par la première en exécutant sa procédure de commit.

La seconde phase. Elle représente la phase de lecture dans la mesure où le test s'effectue uniquement sur la file des objets manipulés en lecture-seule, en s'assurant que tous les objets n'ont pas été modifiés par d'autres transactions depuis leur ouverture.

La troisième phase. Dans cette phase dite phase d'écriture, tous les objets acquis sont relâchés, et si de plus tous les tests préalables sont passés sans échec, la transaction procède au remplacement de l'ancienne copie de chaque objet par sa nouvelle copie fantôme correspondante.

Dans la OSTM, l'aide récursive entre transactions n'est pas systématiquement effectuée lorsqu'elles opèrent en *commit*. En effet, l'aide récursive n'est permise que pour les transactions qui sont dans la première ou la seconde phase du *commit*, à savoir la phase d'écriture et de lecture respectivement. De plus, les conditions suivantes doivent être satisfaites :

- Soit une transaction T_1 en phase d'écriture et tentant d'acquérir l'objet O. Si l'objet O est déjà acquis par une autre transaction T_2 alors T_1 aide T_2.
- Soit deux transactions T_1 et T_2 en phase de lecture, et soit une relation d'ordre totale $\prec$ entre transactions incomplètes. T_1 annule T_2 si et seulement si $T_1 \prec T_2$. Autrement, T_1 aide T_2.

A noter qu'une transaction dans sa phase de lecture n'aide jamais une transaction en phase d'écriture. En outre, en imposant la condition $T_1 \prec T_2$, ceci garantit la non-formation de cycle d'aide. Cette situation se pré-

sente quand les transactions dans leur phase de lecture essayent de lire des objets qui sont mutuellement détenus lors de leur phase d'écriture.

5.2 La RT-STM

5.2.1 Le modèle transactionnel

Objectifs. Si l'on se place uniquement dans la problématique des SGBDs temps réel, il n'est pas utile de proposer un modèle faisant apparaître à la fois celui des tâches et des transactions.
Cependant, la solution que nous proposons préconise un modèle transactionnel temps réel enrichi par le modèle de tâches tout en gardant la possibilité d'être indépendant de ce dernier.
Pour cela, le modèle transactionnel temps réel devrait être enrichi par les spécificités de son environnement à la fois matériel et logiciel, en se basant pour ce faire sur le concept de mémoire transactionnelle. Par ailleurs, étant donné que l'objectif ultime d'une STM est d'être transformée en HTM ou en HyTM, nous pensons alors qu'il faudrait un modèle qui permette de décrire des transactions temps réel s'exécutant sur des processeurs transactionnels dédiés. Ces processeurs, exclusivement (ou en partie) dédiés aux traitements transactionnels, sont supposés être en mesure de prendre en charge les échéances des transactions.

Le modèle. Soit un système multicœur composé d'un ensemble de processeurs M et d'un ensemble de processeurs exclusivement transactionnels M_T. Soit Ω l'ensemble fini des transactions temps réel T_k (sans Entrées/Sorties) s'exécutant sur $M \cup M_T$. $\forall T_k \in \Omega$, alors

$$T_k = (s_k, D_k) \tag{5.1}$$

Où s_k représente la date d'arrivée de la transaction et D_k son délai critique. On supposera que les paramètres de T_k restent constants durant toute son exécution. La valeur $d_k = s_k + D_k$ représente l'échéance absolue de T_k. Lors de l'exécution de T_k, celle-ci possède un nombre de rejoues $n_k \geq 0$, et se termine en cas de succès à une date f_k. Ainsi, le temps de réponse de T_k est symbolisé par $R_k = f_k - s_k$.
La transaction T_k s'exécutant sur un processeur $m \in M \cup M_T$ en phase de rejoue n_k a une durée d'activité $w^m_{k,n}$. La durée totale d'activité de T_k sur l'intervalle de temps $[s_k, f_k)$ est symbolisée par $W_k = \sum_{m,n} w^m_{k,n}$.
La transaction T_k peut avoir des temps d'inactivité (pouvant être dus à des temps d'exécution non transactionnels) dont leur somme est représentée par I_k tel que $R_k = W_k + I_k$.
On dira qu'une transaction T_k respecte son échéance si et seulement si $R_k \leq D_k$ (voir Figure 5.2)

Définition 5.1 *Une transaction T_k s'exécutant sur un processeur $m \in M_T$, en phase de rejoue n_k, possède un taux d'utilisation du processeur transactionnel :*

$$X_k = \frac{\sum_{m,n} w^m_{k,n}}{D_k}$$

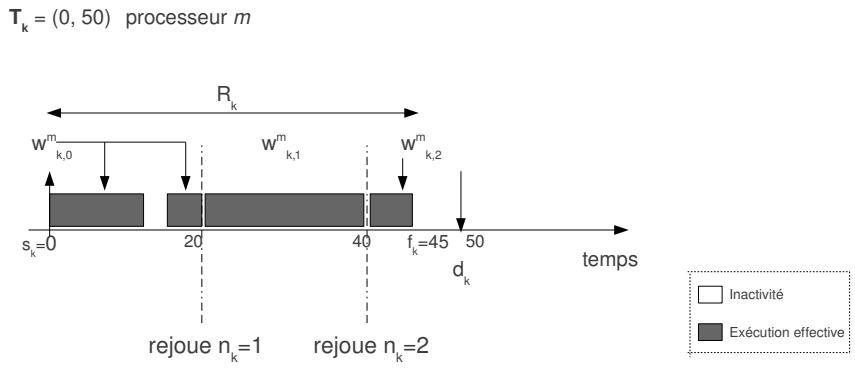


FIGURE 5.2 – *Modélisation d'une transaction temps réel multicœur*

5.2.2 La gestion de concurrence

Dans la littérature, quand le concept de transaction est utilisé dans les solutions basées sur les ordonnanceurs temps réel multiprocesseurs, les fonctionnalités du contrôleur de concurrence se voient réduites, et léguées aux programmeurs. D'une part, ceci est dû au fait qu'un contrôleur de concurrence implique des surcoûts de charge processeur non négligeables (ce problème concerne également les systèmes non temps réel.) D'autre part, inclure un contrôleur de concurrence complexifie davantage l'élaboration d'un test d'ordonnançabilité pour des systèmes à contraintes strictes, étant donné l'indéterminisme en temps lors de la gestion des transactions.
Les systèmes multimédia et de communication ont toutefois besoin à la fois d'exploiter le plein parallélisme, mais aussi de garantir des contraintes temps réel molles. En outre, malgré l'importance actuelle des surcoûts dus au contrôleur de concurrence, nous pensons qu'ils seront relativisés aux fortes puissances de calcul des futures technologies multicœurs, et *a fortiori* quand des processeurs transactionnels leur seront dédiés.
Le défi essentiel qui en découle, est de démontrer la faisabilité d'une intégration d'un contrôleur de concurrence dans un environnement temps réel muticœur, pour une synchronisation non-bloquante.
Plus formellement, le contrôleur de concurrence de la RT-STM, doit permettre à chaque instant de satisfaire la condition suivante :

$$\forall T_k \in T : F_k \leq D_k \tag{5.2}$$

5.2.3 La synchronisation

La progression des transactions dépend de la synchronisation adoptée par le contrôleur de concurrence. Si on choisit d'adopter le wait-free pour la RT-STM, la condition (5.2) sera satisfaite à chaque instant pour toutes les transactions du système, puisque le wait-free assure simultanément la progression de chaque transaction. Cependant, le wait-free est difficile à mettre en œuvre, car les temps d'accès à la mémoire dans un système multicœur est généralement variable.
Le lock-free quant à lui, assure moins de progression que le wait-free, mais

il a l'avantage d'être supporté par la plupart des architectures, comme le démontre la libraire de la OSTM (Fraser 2003). La OSTM peut donc assurer, à chaque instant, la progression d'au moins une transaction. Nous nous basons sur cette garantie de progression, en modifiant les heuristiques de résolutions de conflits de la OSTM. La condition (5.2) est ainsi relaxée dans la RT-STM :

$$\exists T_k \in T : F_k \leq D_k \tag{5.3}$$

5.2.4 Règles de résolution de conflits

Dans la RT-STM, la gestion des conflits entre les transactions est basée sur des règles modifiées de la OSTM. Il s'agit d'introduire au sein des heuristiques de résolution de conflits, les échéances des transactions, pour gérer les conflits sur la base de contraintes de temps. Les modifications apportées sont les suivantes :

- Soit une transaction T_1 en phase d'écriture, et tentant d'acquérir l'objet O. Si l'objet O est déjà acquis par une autre transaction T_2 et

  $$d_1 > d_2 \tag{5.4}$$

  alors T_1 aide T_2. Autrement, T_1 annule T_2.
- Soit deux transactions T_1 et T_2 en phase de lecture, T_1 annule T_2 si et seulement si

  $$d_1 \leq d_2 \tag{5.5}$$

  Autrement, T_1 aide T_2

La relation d'ordre totale $\prec$ qui caractérise la résolution de conflits dans la OSTM, est remplacée par l'ordre basé sur les échéances des transactions. Autrement dit, le contrôleur de concurrence de la RT-STM, prend une décision qui tient compte des échéances des transactions, au lieu de l'ordre FIFO d'arrivée des transactions en mémoire, comme tel est le cas dans la OSTM.
En partant de la OSTM qui est de type lock-free, on est sûr qu'à tout moment, il existe au moins une transaction T_k qui progresse. Avec ces nouvelles règles de résolution basées sur les échéances, la transaction T_k qui progresse au moment du conflit, est celle dont l'échéance est la plus proche. En d'autres termes, la condition (7.1) est satisfaite dans la RT-STM.

5.3 Implémentation et évaluation

5.3.1 Implémentation au sein de la OSTM

Nous avons intégré un ensemble d'informations relatives à l'ordonnancement au sein du contexte de la transaction, afin que des contraintes temps réel molles puissent être supportées par les transactions. Ces informations sont groupées dans une structure appelée *RTSched* et qui résulte de notre modèle transactionnel temps réel.

Au moment de l'initialisation du contexte de la transaction T_k, la valeur D_k est donnée comme paramètre d'entrée pour T_k, et *RTsched* est initialisée à la fois avec la valeur courante du top d'horloge processeur s_k, et l'échéance d_k (voir l'Algorithme 4).

Algorithme 4 : Initialisation de la transaction temps réel T_k

Require: D_k
$T_k.RTSched.s_k \Leftarrow ReadProcessorTicks()$
$T_k.RTSched.d_k \Leftarrow RTSched.s_k + D_k$

D'autre part, nous avons implémenté les règles de résolution de conflits, que nous avons présentées en section 5.2.4, au sein de la fonction récursive du commit (nous ne la présentons pas ici pour des raisons de lisibilité).

5.3.2 Évaluation de la RT-STM

Nous présentons ici la comparaison entre la RT-STM et la OSTM sous P-EDF. Nous avons étudié l'impact à la fois du nombre de *threads* et la variation de la longueur de la fenêtre des échéances molles des transactions temps réel.

Success ratio des transactions temps réel. Ce ratio mesure le nombre de transactions qui respectent leur échéances sur le nombre total de transactions. Le nouveau paramètre temporel (*i.e*, les échéances) a été intégré aussi bien dans la OSTM que dans la RT-STM afin d'effectuer la comparaison.
Après avoir vérifié que la transaction T_k s'exécute bien sur le cœur m où elle a été lancée (voir l'Algorithme 3), la valeur du compteur des tops horloge de m, est alors prélevée en cas de succès de T_k, puis comparée à d_k. Cette comparaison est faite afin de savoir si la transaction T_k qui est sur le point de terminer avec succès, a respecté son échéance. Si tel est le cas, le nombre de transactions qui respectent leurs échéances est incrémenté atomiquement en utilisant l'instruction CAS. D'une manière similaire, le nombre total de transactions est incrémenté au lancement de chaque transaction.

Résultats d'expérimentations

OSTM vs RT-STM. La Figure 5.3 montre le taux de respect des échéances sous la OSTM et la RT-STM. Nous observons que la RT-STM a de meilleures performances que la OSTM. Le ratio de performance est constant (environ 55.6%) et reste indépendant du nombre de *threads* lancés.
Par ailleurs, contrairement aux bases de données temps réel dans lesquelles les transactions sont souvent longues, dans les STMs, le nombre

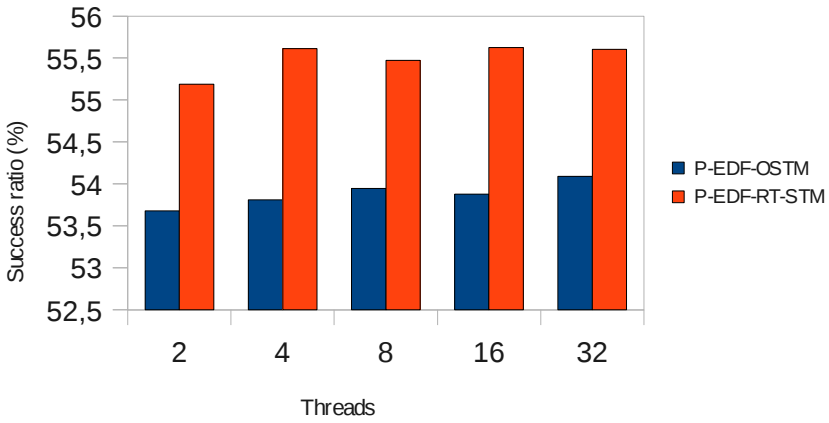


FIGURE 5.3 – *OSTM vs RT-STM*

de transactions est relativement plus important, et les valeurs R_k sont plus petites puisque les transactions résident uniquement en mémoire. De ce fait, une petite différence de performance implique un grand nombre de transactions. A titre illustratif, sur les 10 secondes de test, le nombre de transactions qui respectent leurs échéances au sein de la RT-STM, dépasse celui de la OSTM de plus de 10^5 transactions. Ce nombre est toutefois variable, et sa variabilité est étudiée dans le point suivant.

Gain de la RT-STM. Dans ce qui suit, nous utilisons la définition du *facteur de la fenêtre des échéances* comme décrit dans (Huang et Stankovic 1990). Pour chaque transaction T_k, une échéance spécifique D_k est générée aléatoirement par le biais de la fonction *rnd* comme suivant :

$$D_k = rnd[0, \lambda.base) \tag{5.6}$$

Où λ représente le facteur de la fenêtre d'échéance. La valeur de *base* est arbitrairement fixée à 548 et est un multiple de la fréquence d'horloge des cœurs.
La Figure 5.4 montre le taux de respect des échéances de la RT-STM relativement à celui de la OSTM, en fonction de la longueur de la fenêtre des échéances. Quand les échéances générées sont plus petites que 548 tops d'horloges (c'est-à-dire, $\lambda \in [0, 1)$), les transactions manquent toujours leurs échéances aussi bien dans la OSTM que dans la RT-STM, puisque l'intervalle des échéances est trop petit par rapport à la durée des transactions (*i.e*, $D_k < R_k \forall n_k \geq 0$). A l'opposé, quand $\lambda \in [64, 128]$, les échéances sont toujours respectées dans les deux STMs. En effet, les contraintes temps réel sont plus faciles à satisfaire dans ce cas (*i.e*, $D_k >> R_k$) et ce qui réduit également à nul le gain relatif. Cependant, pour $\lambda \in [2, 4]$ la performance de le RT-STM est maximale, et ce ratio maximum correspond à la situation dans laquelle le taux de respect des échéances est égal à 50% à la fois dans la RT-STM et la OSTM. En outre, la performance maximale obtenue pour la RT-STM est essentiellement due à la première règle de résolution de conflits (voir le paragraphe 5.2.4). En effet, la résolution par la seconde règle de la RT-STM, survient rarement, et elle est définie pour

prévenir la cyclicité.

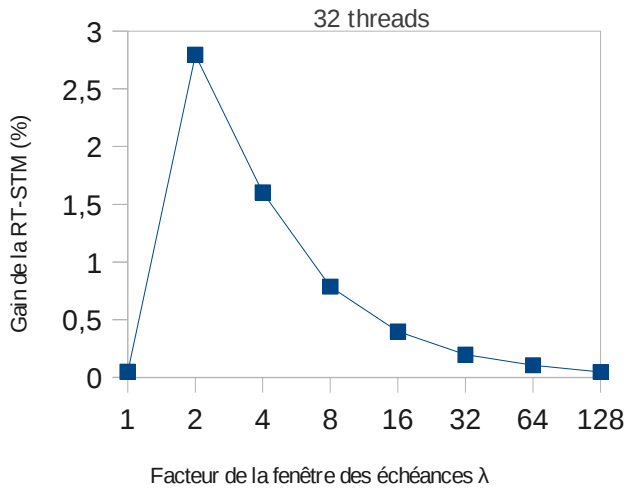


FIGURE 5.4 – *Gain de la RT-STM*

CONCLUSION

Dans ce chapitre, nous avons démontré la faisabilité d'une mémoire STM temps réel (RT-STM). La RT-STM est une adaptation de la OSTM au contexte temps réel *soft*. Nous avons pour cela muni les transactions d'échéances, pouvant ainsi permettre aux transactions de respecter des contraintes de temps. Plus particulièrement, nous avons fait évoluer les heuristiques du contrôleur de concurrence de la OSTM, pour qu'elles se basent sur des contraintes de temps lors de la résolution de conflits. Les résultats expérimentaux ont montré qu'une telle démarche est possible, et nous avons pu obtenir un meilleur gain par rapport à la OSTM en termes de nombre de transactions qui respectent leur échéances.
Par ailleurs, nous avons proposé un modèle transactionnel temps réel pour les STMs temps réel pouvant caractériser aussi bien des transactions à contraintes *hard* que *soft*. De plus, il fait abstraction du type de la plate-forme matérielle utilisée (selon qu'il soit dédié aux processeurs transactionnels ou non) par les transactions temps réel. Ce modèle est mis en avant afin de servir comme base aux prochains chapitres.

RT-STM : Protocoles de gestion de concurrence

6

Sommaire

6.1 Motivations et objectifs

Dans ce chapitre, l'objectif est d'améliorer la mémoire transactionnelle temps réel *soft* (RT-STM) que nous avons définie au chapitre précédent. Il s'agit non seulement d'apposer des contraintes de temps aux transactions, mais aussi d'assurer une bande passante (nombre de succès de transactions par unité de temps) acceptable pour le système de tâches. Ce dernier objectif est motivé par le fait que l'idée d'utiliser des mémoires transactionnelles dans un environnement temps réel *soft*, est avant tout de profiter des avantages qu'elles offrent aussi bien en termes de facilité de programmation que d'amélioration de bande passante par rapport aux verrous.
Dans une mémoire transactionnelle, spécifier des contraintes de temps aux transactions tout en ayant une bande passante acceptable au niveau du système, revient en définitive à répartir le taux de progression des transactions en fonction de leurs priorités respectives. Autrement dit, le contrôleur de concurrence doit être en mesure d'assurer une meilleure progression à la transaction la plus prioritaire, permettant par là-même, la progression de la tâche qui l'exécute. Ainsi, le modèle transactionnel induit, est objectivement similaire à celui présenté dans le chapitre précédent. A celui-ci s'ajoute en plus le fait qu'une contrainte de temps attachée à une tâche donnée, doit être héritée par chaque transaction lancée par la tâche.
Pour cela, nous nous proposons d'analyser dans un premier temps, le facteur influant sur les performances de la RT-STM. Puis, à la lumière de cette analyse, nous présentons de nouveaux protocoles qui permettent d'affiner et d'améliorer notre solution à base de mémoire transactionnelle logicielle pour le temps réel *soft*.
Les principaux résultats de ce chapitre ont conduit à la publication suivante : Queudet et al. (2012).

6.2 Formulation du problème

Dans le précédent chapitre, nous avons démontré que la conception d'une mémoire transactionnelle temps réel (RT-STM) est possible, en considérant à la fois le modèle de tâches temps réel et celui des transactions. Pour cela, nous avons muni le contrôleur de concurrence de la OSTM de nouvelles heuristiques permettant à une transaction en cas de conflit, d'aider uniquement une autre de plus grande priorité. Les résultats de nos expérimentations ont montré que dans la RT-STM, le taux de transactions respectant leurs échéances est supérieur à celui de la OSTM. Cependant, le gain reste relativement faible comparativement à celui de la OTSM, et ne dépasse pas les 3% dans le meilleur des cas.

Afin d'analyser les causes de ce faible gain de la RT-STM par rapport à la OSTM, nous avons porté notre plateforme de tests présentée dans le chapitre précédent (architecture double cœurs) sur une architecture à huit cœurs de type Intel-Xeon. Cette dernière en effet, nous permet de voir le comportement de la RT-STM et plus particulièrement celui du contrôleur de concurrence lors du passage à l'échelle en termes de nombre de processeurs.
La Figure 6.1 montre que l'impact exercé par le contrôleur de concur-

rence est proportionnel au nombre de processeurs. Plus particulièrement, nous remarquons que dans le cas de deux cœurs, la majorité (*i.e* environ $100 - 13,5 = 86,5\%$) des transactions ne sont soumises qu'aux règles de sérialisation qui ne sont pas temps réel. En d'autres termes, les heuristiques caractérisant la RT-STM ne sont sollicitées par le contrôleur de concurrence qu'avec un taux de $13,5\%$, ce qui rend l'effet de ces heuristiques peu influent, et explique donc le faible gain de la RT-STM exposé dans le chapitre précédent.
En cas de conflit, les heuristiques de la RT-STM, rappelons-le, effectuent au moment du commit une aide entre transactions en se basant sur leurs échéances. Les heuristiques ne sont donc sollicitées par le contrôleur de concurrence que pour des transactions ayant un conflit sur le même objet, et surtout pour des transactions étant toutes en même temps en phase du commit. Ce cas précis survient rarement par rapport à la situation où les transactions sont abandonnées d'office par les règles de sérialisation non temps réel.
Ces règles non temps réel dans la RT-STM sont héritées de la conception de l'ordonnanceur des transactions de la OSTM. En effet, tout comme la OSTM, la RT-STM est aussi basée sur une détection de conflits de type *lazy*, puisque tous les conflits ne sont détectés et résolus qu'en phase du commit. A titre d'exemple, s'il y a une seule transaction T_1 dans la phase du commit, alors la RT-STM fait systématiquement *commiter* T_1. Quand une autre transaction T_2 arrive en phase du commit et trouve ses objets mis à jour par T_1, alors T_2 va faire un *rollback*, et ce, même si T_2 a une plus grande priorité que T_1. C'est ainsi que sur un espace de conflits important (correspondant au complémentaire de la courbe de la Figure 6.1) des transactions moins prioritaires arrivent à progresser au détriment des plus prioritaires en échappant aux heuristiques temps réel.

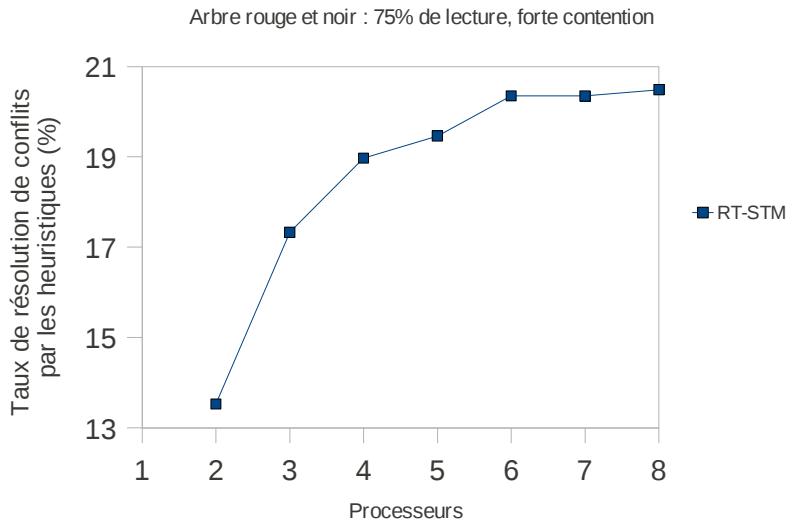


FIGURE 6.1 – *Effet des heuristiques de la RT-STM*

Critique de l'existant

Le problème énoncé au précédent paragraphe se synthétise par le fait qu'une transaction puisse mettre à jour des objets qui sont en cours de manipulation par une autre transaction de plus grande priorité, forçant ultérieurement cette dernière à faire un *rollback*.
Les solutions existantes permettent des résolutions aussi bien d'une manière pessimiste comme le 2-PL-HP de Abbott et Garcia-Molina (1988), qu'optimiste comme la résolution par sérialisation de Lindström et Raatikainen (2000).
Cependant, le 2-PL-HP est basé sur des verrous dont l'utilisation est remise en cause par le concept-même de mémoire transactionnelle, puisque les sections de code verrouillées ont une granularité élevée, et cela réduit la bande passante du système.
La bande passante n'est pas pour autant un problème résolu dans les protocoles optimistes tels que RT-DATI (Lindström et Raatikainen 2000). En effet, le parallélisme dans ce dernier est peu exploité dans la mesure où la sérialisation des transactions part de l'hypothèse que la phase de validation n'est pas réentrante.

6.3 Modèle du système

Le modèle transactionnel que nous avons présenté au chapitre précédent (cf. paragraphe 5.2.1) est suffisant pour décrire les contraintes de temps des transactions temps réel. Cependant, ce modèle à lui seul ne permet pas de répondre au nouvel objectif fixé dans ce chapitre. En effet, il s'agit de répartir le taux de progression des transactions en fonction de la priorité des tâches. Or le modèle des tâches décrit au premier chapitre (cf. paragraphe 1.1.2) ne permet pas de décrire un tel objectif. Pour cela, nous mettons en relation le modèle de tâches avec le modèle transactionnel temps réel.
Modèle de tâches étendu. Une tâche sporadique $\tau_i = (r_i, C_i, P_i, D_i, \Omega_i)$ partageant des ressources accédées via des transactions, est caractérisée par une date de réveil r_i, une durée d'exécution au pire cas C_i, un délai critique D_i, un délai minimum P_i d'inter-arrivée des travaux, et un ensemble Ω_i de k transactions : $\Omega_i = \{T_1, T_2, ...T_k\}$. La tâche doit se terminer avant son échéance absolue $d_i = r_i + D_i$.
Modèle de transaction. Une transaction $T_j = (s_j, D_j)$ est caractérisé par une date d'arrivée s_j, et un délai critique D_j. La transaction doit se terminer avant sa date d'échéance $d_j = s_j + D_j$. Nous supposons que l'ensemble de transactions $\Omega_i = \{T_1, T_2, ...T_k\}$ d'une tâche $\tau_i = (r_i, C_i, P_i, D_i, \Omega_i)$ est tel que $d_j = d_i \forall j \in [1, k]$.

6.4 Protocoles pour la RT-STM

Nous présentons dans ce paragraphe deux nouveaux protocoles, qui sont pour l'un pessimiste et l'autre optimiste. Le premier protocole est mono-écrivain et de type *eager*. Nous l'avons appelé *1W* (*one-Writer* en anglais).

Le deuxième protocole est multi-écrivains de type *lazy* dans sa détection de conflits. Nous l'avons appelé *MW* (*Multi-Writer* en anglais).
Dans le but d'atteindre les objectifs fixés dans ce chapitre, ces deux protocoles se basent d'une part sur le principe de marquage d'objets afin qu'une transaction active et de priorité élevée ne voie pas ses objets mis à jour par une transaction de plus faible priorité. D'autre part, ils utilisent un verrouillage à grain fin, moyennant l'instruction machine CAS, afin d'améliorer la bande passante par rapport aux solutions de synchronisation existantes pour les systèmes temps réel basées sur les verrous.
Par ailleurs, nous partons de la constatation que l'un des principaux problèmes qui conduit à avoir des résultats de comparaison différents lors de l'évaluation des protocoles, est l'hétérogénéité à la fois des plateformes de test et des structures de données composant les protocoles. Pour cette raison, nous gardons les structures de données de la OSTM, et ce, même dans le cas où leur complexité s'avère non nécessaire au protocole qui les manipule.

6.4.1 Structures de données

Pour les deux protocoles de synchronisation temps réel que nous proposons (*i.e*, 1W *eager* et MW *lazy*), chaque transaction en phase du commit devrait être en mesure de "voir" les objets qui sont en cours de manipulation en écriture par d'autres transactions, ceci, dans le but de prendre une décision d'ordonnancement basée sur les échéances des transactions. Dans cette perspective, nous introduisons au sein de chaque entête d'objet, un pointeur de marquage noté *Mark* (cf. Figure 6.2).

Marquage d'objets (*Mark*)

L'opération de marquage est seulement effectuée sur un objet auquel une transaction souhaite accéder en écriture. Avant cette ouverture, la transaction essaye d'abord de le marquer. L'opération de marquage n'est pas effectuée si l'objet a déjà été marqué par une autre transaction de plus grande priorité (*i.e*, une transaction ayant une plus petite échéance absolue). Dans le cas contraire, la valeur de marquage est mise à jour avec l'adresse du descripteur de la transaction.
Comme nous l'avons déjà énoncé, nos deux protocoles utilisent un verrouillage à grain fin de type CAS. Les instructions existantes de type CAS Conditionné (CCAS) (Fraser et Harris 2007) sur certains processeurs ne fournissent pas la fonction atomique qui permet de mettre à jour un mot mémoire avec le troisième paramètre de l'instruction si le second paramètre a une plus grande ou petite valeur que le premier. Même si cette instruction est fournie sur certaines machines, elle reste toutefois non répandue sur la majorité des processeurs.

Nous avons donc implémenté une nouvelle procédure légère que nous avons nommée en anglais *CAS if Greater priority* (CASG) (cf. Algorithme 5) qui est basée sur l'instruction atomique CAS. La procédure CASG est effectuée uniquement si le contenu du pointeur de marquage *Mark* est vide (ligne 3) ou bien si la transaction T_{new} qui tente d'effectuer le marquage,

a une plus grande priorité (ligne 5 et ligne 9). Pour les transactions ayant les mêmes échéances, l'adresse mémoire du descripteur de la transaction est utilisé pour prévenir les situations d'interblocage (ligne 6).
La procédure CASG retourne soit un marquage échoué (*MARK FAILED*), soit le résultat de l'instruction CAS sous-jacente. Pour ce dernier cas, même si le CASG n'a pas échoué, l'appelant de la procédure CASG doit tout de même tester si la sous-procédure atomique CAS a réussi.

Algorithme 5 : *CASG*

Require: mot : *Mark, Transaction : T_{old}, T_{new}
Retourne : mot
1: **Init** mot :*p $\Leftarrow$ MARK FAILED
2: **if** (T_{old} = nil) **then**
3: **return** CAS(&Mark, nil, T_{new})
4: **end if**
5: **if** (T_{new}.Deadline $\leq$ T_{old}.Deadline) **then**
6: **if** ((T_{new}.Deadline = T_{old}.Deadline) **And** ($T_{old} < T_{new}$)) **then**
7: **return** p
8: **end if**
9: p $\Leftarrow$ CAS(&Mark, T_{old}, T_{new})
10: **end if**
11: **return** p

Tableau de lecteurs (*Readers Array*)

Pour gérer les conflits de type lecture/écriture selon la priorité des transactions, le schéma idéal serait d'appliquer le même principe de marquage que celui vu au paragraphe précédent, en prenant en charge autant de pointeurs de marquage qu'il y a de lecteurs dans le système. Cependant, un MCAS (Multi-Word CAS) est nécessaire dans ce cas pour maintenir la consistance de l'ensemble des mots mémoire dédiés au marquage, à cause de l'indéterminisme caractérisant la date de lancement et de terminaison de la transaction. Par ailleurs, nous pensons qu'implémenter un MCAS au sein de de la RT-STM est coûteux, et pourrait générer d'importants overheads.
Nous avons donc fait un compromis et intégré au sein de chaque entête d'objet un tableau statique de pointeurs pour gérer, selon la priorité des transactions, les conflits de type lecture/écriture. La transaction qui ouvre un objet en lecture, mettra (ou retirera) juste son identifiant à la case du tableau correspondant à son propre index au moment respectivement de son lancement ou de sa terminaison (voir Figure 6.2).
Il est important de noter que la faible inconsistance du tableau *Readers Array* (RA) influe sur les contraintes temps réel et non pas sur la sérialisation des transactions. Le RA est alors juste utile pour aider le contrôleur de concurrence durant la phase du commit à prendre une décision basée sur les échéances des transactions. En outre, le RA est indexé par l'ID de la transaction qui est également le même que celui de la tâche puisque la RT-STM ne supporte pas les transactions imbriquées. De plus, nous considérons comme coûteux le tri en ligne par priorité à cause de la dyna-

micité des échéances des transactions. Nous mentionnons à cet effet que l'échéance d'une transaction reste constante durant toute l'exécution de la transaction même si cette dernière fait plusieurs fois des *rollbacks*. En conséquence de quoi, nous supposons juste que les tâches partageant un objet en lecture, ont des IDs triés par ordre croissant de leur période, dans le but de minimiser la recherche de la plus petite valeur d'échéance dans le RA.

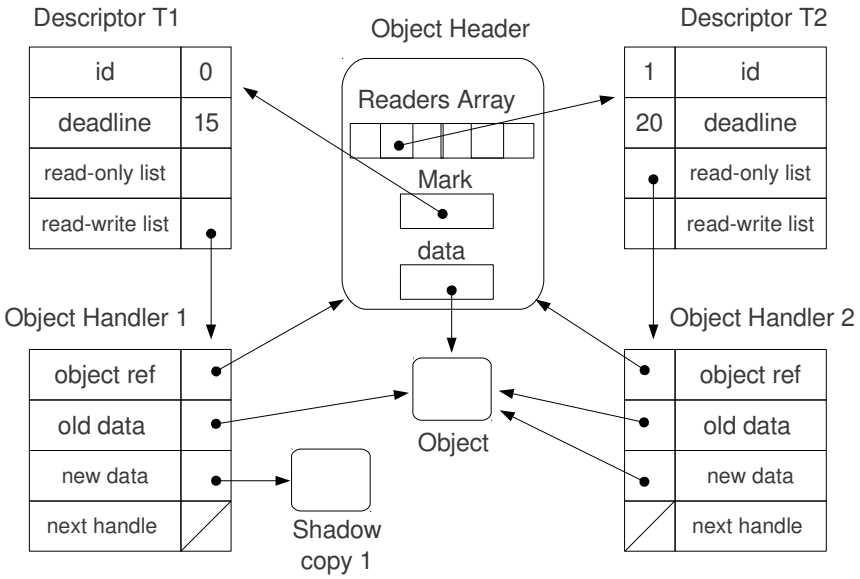


FIGURE 6.2 – *Structures de données de la RT-STM : exemple de deux transactions T_1 et T_2 accédant à un objet respectivement en écriture et en lecture.*

6.4.2 Le protocole mono-écrivain (1W)

Ce protocole est de type *eager* dans sa détection de conflits. Nous l'avons appelé mono-écrivain car chaque transaction qui tente d'ouvrir un objet en écriture, réitère jusqu'à ce qu'elle marque l'objet. Ainsi, seule la transaction qui est la plus proche de son échéance peut progresser parmi toutes les transactions tentant d'ouvrir le même objet en écriture.
Le 1W partage ainsi le même principe de résolution de conflits que le protocole 2-PL-HP (cf. 2.2.1 Les protocoles pessimistes). Mais contrairement à la version de base du 2-PL-HP, le 1W permet d'une part, un verrouillage à grain fin de type CAS et d'autre part intègre le mécanisme d'aide (*helping*) afin de maximiser la bande passante du système.
Par ailleurs, l'utilisation du CAS comme instruction de synchronisation rend le 1W proche du protocole de Ennal (2006) qui, rappelons-le, est une implémentation du 2-PL non temps réel.
Nous avons modifié les deux procédures principales d'ouverture d'un objet en lecture et/ou en écriture décrites dans la OSTM. En particulier, les modifications apportées ciblent la sous-procédure de bas niveau chargée de renvoyer la version courante de l'objet, initialement commune à ces deux procédures. Celle-ci se retrouve adaptée différemment, en fonction du mode d'ouverture de l'objet (lecture ou écriture).

Adaptation de la sous-procédure d'écriture

Cette sous-procédure renvoie la version courante de l'objet à la procédure principale d'ouverture d'un objet en écriture (cf. Algorithme 6). Si la transaction a déjà marqué l'objet, la version courante est retournée (lignes $2-4$). Dans le cas contraire, on réitère jusqu'à ce que l'opération de marquage soit effectuée (lignes $5-22$). Pour cela, une seconde boucle (lignes $6-21$) est nécessaire pour garantir la bonne condition du test de sortie de la boucle principale (ligne 22) car le contenu du pointeur *mark* peut être changé en parallèle par d'autres transactions.
On commence d'abord par récupérer la valeur de marquage (ligne 7) et on teste l'état de son bit de poids faible (LSB) (ligne 8). Si le LSB est mis à un, cela veut dire que le pointeur de marquage contient le descripteur de la transaction (ligne 9) qui est en train de *commiter*. On l'aide donc à finir (ligne 10), et ce, même si cette transaction aidée est de plus faible priorité. En effet, nous estimons qu'abandonner une transaction au moment où elle est entrain de *commiter*, peut conduire à une baisse significative de la bande passante du système. On récupère de nouveau le pointeur de marquage (ligne 11) puisqu'il pourrait être modifié après l'appel du commit. Cette dernière opération est répétée tant que le LSB est à un (ligne 12).
Si le LSB du champ de marquage est à zéro, on essaye de le marquer, en invoquant la procédure CASG décrite précédemment. Dans le cas où le marquage échoue, cela veut dire que le champ de marquage a déjà été marqué par une autre transaction de plus haute priorité, et auquel cas, on réitère l'opération de marquage (ligne 17). Dans le cas contraire, on sort de la seconde boucle (ligne 19). Mais avant de retourner l'objet marqué (ligne 23), on teste d'abord le résultat du CASG afin de vérifier si la valeur du champ de marquage est restée inchangée (ligne 22).

Adaptation de la sous-procédure de lecture

Cette sous-procédure renvoie la version courante de l'objet à la procédure principale d'ouverture d'un objet en lecture (cf. Algorithme 7). Pour ce faire, une boucle est mise en place pour récupérer à la fin, une copie valide de l'objet (lignes $2-13$). La copie de l'objet est d'abord sauvegardée dans la variable *data* (ligne 3). Si l'objet est acquis par une autre transaction, celle-ci est aidée dans le but d'obtenir après cette aide, une éventuelle nouvelle copie de l'objet (lignes $5-9$). Puis, on vérifie si la copie de l'objet courante, est différente de celle sauvée dans la variable *data*, et auquel cas, on reboucle. Dans le cas contraire, on sort de la boucle (ligne 11), et avant de retourner la variable *data*, la transaction s'inscrit dans le tableau RA (ligne 14).

Adaptation de la procédure du commit

Comme dans la OSTM, la procédure du commit est divisée en trois phases, à savoir l'acquisition, la lecture et la validation.

Dans la première phase (cf. Algorithme 8), on tente d'acquérir tous les objets marqués. Pour réaliser cette opération sur l'entête de l'objet d'une manière atomique, l'instruction CAS est utilisée pour mettre à la valeur 1

Algorithme 6 : *1W : Sous-procédure d'écriture*

Require: Transaction : T, ObjectHeader : OH
Retourne : Adresse de la version courante de l'objet marqué.

```
1: Init mot : *mark ⇐ OH.Mark, Transaction : T_other
2: if (mark = T) then
3:     return OH.Data
4: end if
5: repeat
6:     loop
7:         T_other ⇐ mark
8:         if (LSB(mark) ⇐ 1) then
9:             T_other ⇐ (mark & LSB(mark) ⇐ 0)
10:            Aider en commitant (T_other)
11:            mark ⇐ OH.Mark
12:            continue loop
13:        end if
14:        mark ⇐ CASG(&OH.Mark, T_other, T)
15:        if (mark = MARK FAILED) then
16:            mark ⇐ OH.Mark
17:            continue
18:        else
19:            break
20:        end if
21:    end loop
22: until T_other ≠ mark
23: return OH.Data
```

Algorithme 7 : *1W : Sous-procédure de lecture*

```
Require: Transaction : T, ObjectHeader : OH
    Retourne : Adresse de la version courante de l'objet.
 1: Init mot : *data, mot : *mark, Transaction : T_other
 2: loop
 3:     data ⇐ OH.Data
 4:     mark ⇐ OH.Mark
 5:     while (LSB(mark) = 1) do
 6:         T_other ⇐ (mark & LSB(mark) ⇐ 0)
 7:         Aider en commitant (T_other)
 8:         mark ⇐ OH.Mark
 9:     end while
10:     if (data = OH.Data) then
11:         break
12:     end if
13: end loop
14: OH.ReadersArray[T.Id] ⇐ T
15: return data
```

le LSB du pointeur de marquage (ligne 4).
Dans le cas où le CAS échoue, cela veut dire qu'une transaction avec une plus grande priorité a déjà marqué l'objet (ligne 6). Le traitement continue alors que si le CAS renvoie le descripteur de la transaction courante T ou bien le descripteur T déjà acquis (ligne 5). Puis, un parcours du tableau RA est effectué afin de vérifier si une transaction de plus grande priorité a ouvert l'objet en lecture (ligne 8). Si tel est le cas, on prévient un conflit de type lecture-écriture entre la transaction de plus grande priorité et celle de plus petite priorité (ligne 9).

Algorithme 8 : *1W : Commit : première phase (acquisition)*

```
Require: Transaction : T Retourne : SUCCESS ou FAILED.
 1: Init mot : status ⇐ FAILED, mot : *data, mot : *mark,
    Transaction : T_other, ObjectHandler : Handler, ObjectHeader : OH
 2: for each Handler in T.read-write-list do
 3:     OH ⇐ Handler.objectRef
 4:     mark ⇐ CAS(&OH.Mark, T, (T & LSB(T)⇐ 1))
 5:     if ((mark ≠ T) And (mark ≠ (T & LSB(T)⇐ 1))) then
 6:         goto Fail
 7:     end if
 8:     if ((∃ Transaction non commitée : R in OH.ReadersArray) And
        (R.Deadline < T.Deadline)) then
 9:         goto Fail
10:     end if
11: end for
```

Dans la seconde phase (cf. Algorithme 9), on vérifie les objets ouverts en lecture. Si un objet ouvert en lecture, a été modifié à l'insu de la

transaction en cours T, celle-ci est alors rejouée (ligne 16). Mais avant que cette vérification ne soit effectuée, on commence d'abord par vérifier l'état d'autres éventuelles transactions qui ont ouvert également l'objet en écriture (ligne 4). Et si une transaction de plus grande priorité est trouvée dans le RA de l'objet, celle-ci est aidée à accomplir son commit le cas échéant (ligne 12). Cependant, une précaution doit être prise pour éviter la cyclicité d'aide entre transactions, dans le cas où les échéances des transactions seraient égales. Ceci est effectué en annulant arbitrairement la transaction qui est arrivée la dernière, ou en d'autres termes, la transaction dont l'adresse du descripteur est la plus grande (ligne 9). Autrement, l'aide n'est pas effectuée (ligne 7).
Avant la dernière phase, si la transaction n'a manipulé que des objets en lecture seule, on met juste à *nil* le tableau RA à la position d'index Id de la transaction, et le statut final de la transaction pour cette phase de lecture est par la suite renvoyé (ligne 26). A ce point d'avancement, la transaction a effectué les deux phases en réussissant le commit (ligne 19).

Algorithme 9 : *1W : Commit : seconde phase (lecture)*

```
1:  for each Handler in T.read-only-list do
2:      OH ⇐ Handler.ObjectRef
3:      mark ⇐ OH.Mark
4:      while (LSB(mark) = 1) do
5:          T_other ⇐ (mark & LSB(mark) ⇐ 0)
6:          if (T.Deadline < T_other.Deadline) then
7:              break
8:          end if
9:          if ((T.Deadline = T_other.Deadline) And
            (T_other < T)) then
10:             goto Fail
11:         end if
12:         Aider en commitant (T_other)
13:         mark ⇐ OH.Mark
14:     end while
15:     if (OH.Data ≠ Handler.OldData) then
16:         goto Fail
17:     end if
18: end for
19: status ⇐ SUCCESS
20: Fail :
21: if (T.read-write-list = 0) then
22:     for each Handler in T.read-only-list do
23:         OH ⇐ Handler.ObjectRef
24:         if (OH.ReadersArray[T.Id]=T) then
            OH.ReadearsArray[T.Id] = nil end if
25:     end for
26:     return status
27: end if
```

Dans la troisième phase (cf. Algorithme 10), l'état de la transaction

est atomiquement changé à l'état *SUCCESS* (ligne 1), puisque les phases précédentes sont supposées réussies à ce niveau d'exécution. Le CAS est utilisé pour ce faire, afin d'éviter un changement multiple du statut de la transaction quand celle-ci est aidée par d'autres transactions en parallèle. Ainsi, une seule transaction effectue le remplacement des copies d'objets (*i.e*, faire pointer atomiquement la zone *Data* de l'entête de l'objet, vers la copie locale de la transaction) parmi toutes celles qui s'exécuteraient en parallèle (lignes 4-7).
Une fois ceci fait, on met à *nil* le RA aussi bien pour les entêtes contenus dans la file de lecture-seule que dans la file de lecture-écriture (lignes 9-12).
Au final, les pointeurs de marquage sont remis à *nil* (lignes 13-18). Pour cette dernière opération, deux cas sont à prendre en compte pour la transaction en cours d'exécution T. Dans le premier cas, T a réussi au préalable à la fois à marquer et à acquérir l'objet (ligne 15). Le deuxième cas correspond uniquement à la situation où T a marqué l'objet sans avoir pu l'acquérir en première phase (ligne 16). Ce dernier cas survient typiquement lorsqu'une transaction en mode *helping* (*i.e*, une transaction aidante) trouve en première phase ses objets déjà mis à jour par la transaction principale, et que cette dernière est toujours en commit.

Algorithme 10 : *1W : Commit : troisième phase (validation)*

```
 1: CAS(&T.Status, PROGRESS, status)
 2: status ⇐ T.Status
 3: if (T.Status = SUCCESS) then
 4: for each Handler in T.read-write-list do
 5:     OH ⇐ Handler.ObjectRef
 6:     CAS(&OH.Data, Handler.OldData, Handler.NewData)
 7: end for
 8: end if
 9: for each OH in T.read-only-list ∪ T.read-write-list do
10:     OH ⇐ Handler.ObjectRef
11:     if OH.ReadersArray[T.Id]=T then mise à nil end if
12: end for
13: for each Handler in T.read-only-list do
14:     OH ⇐ Handler.ObjectRef
15:     if (T = CAS(&OH.Mark, (T & LSB(T) ⇐ 1), nil)) then
16:         CAS(&OH.Mark, T, nil)
17:     end if
18: end for
19: return status
```

6.4.3 Le protocole multi-écrivain (MW)

Ce protocole est de type *lazy* dans sa détection de conflits. Les conflits sont détectés au plus tard, ce qui conduit à avoir des écrivains concurrents dans le système. Ceci rend donc le MW plus optimiste que le 1W.
Contrairement au protocole 1W, le fait de ne pas pouvoir marquer un ob-

jet lors d'une ouverture en écriture ne pénalise pas la transaction. Bien au contraire, le MW laisse s'exécuter les transactions sans que l'exécution de l'une n'influe sur l'exécution de l'autre. Le MW est un protocole lock-free utilisant à la fois le CAS et le Double-CAS (DCAS) comme instructions de base pour la synchronisation.
Nous rappelons que face aux algorithmes lock-free, le premier défi à relever est de démontrer la possibilité de leur implémentation. Le MW s'inscrit alors dans ce challenge avec en plus le fait de prendre en compte les contraintes de temps des transactions. Pour cela, nous avons utilisé les deux points de synchronisation contigus, à savoir la zone de marquage *Mark* et la zone du pointeur *Data* contenant les données de l'objet. Ces deux zones mémoires sont mises à jour séparément (ou en même temps) à l'aide des instructions atomiques CAS et CASG (ou à l'aide du Double-CASG respectivement). Le but du Double-CASG, comme nous le verrons en détails dans le prochain paragraphe, est de pouvoir prendre une décision au plus tard (*i.e*, lazy) devant une situation d'objet non marqué.

Adaptation de la sous-procédure d'écriture

Cette procédure ressemble à celle du protocole 1W (cf. Algorithme 6). A la différence près que le MW ne réitère pas sur un objet si celui-ci est marqué par une autre transaction. Ainsi, lorsque la procédure d'écriture du MW tente de marquer l'objet avec la procédure CASG, les valeurs de retour de cette dernière ne sont pas prises en compte. Ce qui ne met donc aucune contrainte quant à l'ouverture des objets en écriture.

Adaptation de la sous-procédure de lecture

Le 1W et le MW ont les procédures de lecture identiques (cf. Algorithme 7).

Adaptation de la procédure du commit

Tout comme la procédure du 1W, celle du MW est aussi décomposée en trois phases. Hormis la première phase, le 1W et le MW présentent tous les deux des phases identiques.
Pour le protocole MW, dans cette nouvelle adaptation de la phase d'acquisition du MW (cf. Algorithme 11), si la transaction a marqué l'objet, celui-ci est acquis (ligne 5) de la même manière que dans le 1W. De plus, quand une transaction tente d'acquérir un objet et qu'elle le trouve déjà acquis par une autre transaction, cette dernière est aidée à accomplir son commit (ligne 21). La vérification faite au niveau du tableau de lecteurs (RA) (lignes 22-24) est également la même que dans le 1W.
Dans le cas où l'objet n'est pas marqué (ligne 9), la situation devient atomiquement indécidable. En effet, l'objet est soit mis à jour auparavant par une autre transaction, soit libéré après un échec de la transaction qui l'a parallèlement manipulée. Il est alors nécessaire d'avoir deux points de synchronisation, à savoir la zone de marquage *Mark* et le pointeur vers les données de l'objet *Data*, contenus dans l'entête de l'objet. Le but de disposer de ces deux points de synchronisation, permet à la fois de tenter de marquer (ou marquer de nouveau) l'objet (lignes 10) tout en s'assurant

que le pointeur vers l'objet n'a pas changé depuis son ouverture (ligne 11). En outre, l'opération de marquage à réaliser fait également l'acquisition de l'objet en mettant à la valeur 1 le LSB du champ de marquage (ligne 13). Pour ce faire, nous procédons par le biais de la procédure Double-CASG dont nous détaillons le fonctionnement ci-après.

Algorithme 11 : *MW : Commit : première phase (acquisition)*

Require: Transaction : T Retourne : SUCCESS ou FAILED.
1: **Init** mot : status $\Leftarrow$ FAILED, mot : *data, mot : *mark,
Transaction : T_{other}, ObjectHandler : Handler, ObjectHeader : OH
double mot : D_{word}, $Double_{old}$, $Double_{new}$
2: **for each** Handler **in** T.read-write-list **do**
3: OH $\Leftarrow$ Handler.ObjectRef
4: **loop**
5: mark $\Leftarrow$ CAS(&OH.Mark, T, (T & LSB(T)$\Leftarrow$ 1))
6: **if** ((mark = T) **Or** (LSB(mark) = 1)) **then**
7: **break**
8: **end if**
9: **if** (LSB(mark) = 0) **then**
10: $Double_{old}$.Mark $\Leftarrow$ mark
11: $Double_{old}$.Data $\Leftarrow$ T.read-write-list.OldData
12: $Double_{new}$.Data $\Leftarrow$ $Double_{old}$.Data
13: $Double_{new}$.Mark $\Leftarrow$ (T & LSB(T)$\Leftarrow$ 1)
14: D_{word} $\Leftarrow$ DCASG(&OH.Mark, $Double_{old}$, $Double_{new}$)
15: **if** ($D_{word} \neq Double_{old}$) **then**
16: **goto** Fail
17: **end if**
18: **break**
19: **end if**
20: T_{other} $\Leftarrow$ mark
21: Aider en *committant* (T_{other})
22: **if** (($\exists$ Transaction non *commitée* : R **in** OH.ReadersArray)
And (R.Deadline $<$ T.Deadline)) **then**
23: **goto** Fail
24: **end if**
25: **end loop**
26: **end for**

La procédure Double-CASG (DCASG)

Cette procédure (cf. Algorithme 12) opère sur deux mots mémoire, et ressemble au fonctionnement du CASG.
Si le champ de marquage est vide, la procédure DCASG tente directement de le marquer (lignes 2-4). Le DCAS sous-jacent (ligne 3) peut échouer si les deux mots mémoire ont été modifiés en parallèle par d'autres transactions.
Dans un deuxième temps, la procédure DCASG reconstruit les descripteurs de transactions à partir des paramètres de la fonction (lignes 5-6) afin d'évaluer leurs échéances (ligne 7). D'une manière similaire au CASG,

l'évitement d'un éventuel interblocage est pris en compte (ligne 8) avant de tenter le remplacement atomique des deux mots mémoire (ligne 11).

Algorithme 12 : *Double CASG*

Require: double mot : *pointer, double mot : $Double_{old}$, double mot : $Double_{new}$
Retourne : double mot.

```
1:  Init *p ⇐ MARK FAILED (double mot), Transactions : T_old, T_new
2:  if (Double_old.Mark = nil) then
3:      return DCAS(&pointer, Double_old, Double_new)
4:  end if
5:  T_new ⇐ (Double_new.Data & LSB(Double_new.Data) ⇐ 0)
6:  T_old ⇐ (Double_old.Data)
7:  if (T_new.Deadline ≤ T_old.Deadline) then
8:      if ((T_old.Deadline = T_new.Deadline) And (Double_old < Double_new))
        then
9:          return p
10:     end if
11:     p ⇐ DCAS(&pointer, Double_old, Double_new)
12: end if
13: return p
```

6.5 Évaluation des protocoles

Dans les systèmes classiques non temps réel, les protocoles de contrôle de concurrence des STMs sont souvent évalués avec des métriques caractérisant à la fois la bande passante et la progression globale des transactions. Cependant, pour les systèmes temps réel, les temps de rejoues des transactions influent négativement sur le respect des contraintes temporelles des tâches. Par conséquent, l'objectif de ce paragraphe est de montrer si nos protocoles peuvent réduire l'effet de rejoues des transactions tout en ayant une bande passante acceptable. Il s'agit plus particulièrement de comparer nos protocoles 1W et MW avec deux protocoles respectivement dans le domaine classique et temps réel, à savoir le protocole non bloquant de la OSTM et le protocole bloquant FMLP.
Pour ce faire, nous rapportons d'abord les résultats comparatifs nous permettant de sélectionner les paramètres qui fournissent la meilleure bande passante pour nos protocoles. Puis, nous fixons ces paramètres pour évaluer l'effet de rejoues des transactions.

6.5.1 Contexte d'évaluation

La plateforme de test que nous avons présentée au chapitre précédent est reprise ici, mais avec des changements notables notamment en termes d'architecture et de *benchmarks*.

Plateforme matérielle

Comme nous l'avons vu au paragraphe 6.2, plus le nombre de cœurs sera important sur l'architecture matérielle plus il sera aisé de montrer l'effet du contrôleur de concurrence. Pour cette raison, la plateforme matérielle utilisée dans nos expérimentations repose sur un processeur à 8 cœurs de type Intel Xeon 5552 cadencé à 2.26Ghz, muni d'un cache L3 de 8Mo et de 3Go de mémoire RAM.
Plus précisément, dans toutes nos expérimentations, nous utilisons 7 cœurs sur 8, le cœur restant étant utilisé pour exécuter le script *bash* de lancement de nos tests.

Configuration logicielle

Benchmarks transactionnels. Les *benchmarks* synthétiques usuels comme les arbres rouges et noir et les *skips lists*, permettent d'évaluer le comportement des STMs face à la complexité que peuvent fournir de telles structures de données. Cependant, pour le contexte temps réel, nous avons besoin de contrôler cette complexité et les overheads temporels qui en découlent.
Pour cela, nous utilisons plutôt un tableau d'objets. Tous les objets sont identiques et contiennent un entier. Pour des raisons de compatibilité, chaque STM que nous testons, effectue les trois opérations supportées par la OSTM à savoir, *update*, *remove*, et *lookup*. L'opération *update* ouvre tous les objets du tableau et incrémente leur entier, puis tente de faire un commit. D'une manière similaire, l'opération *remove* décrémente l'entier. Enfin, l'opération *lookup* ouvre en lecture tous les objets. Dans le reste de ce chapitre, nous utiliserons l'opération *write* pour se référer aux deux opérations *update* et *remove*, et l'opération *read* pour se référer à celle du *lookup*.

Profil de l'application temps réel. Le paramètre P_i des tâches, est généré d'une manière aléatoire dans l'intervalle $[20ms, 100ms]$. Les valeurs générées sont triées d'une manière croissante. Ce tri n'est pas obligatoire mais utile lors de l'accès au tableau RA pour nos protocoles. Le paramètre C_i est quant à lui variable et dépend du facteur d'utilisation des cœurs qui est un paramètre d'entrée. Rappelons que les tâches temps réel sont sporadiques avec $D_i = P_i$.

Ordonnancement des tâches. Pour toutes nos expérimentations, nous utilisons la politique P-EDF. Ce choix a été justifié au chapitre précédent par le fait que les autres politiques comme G-EDF et Pfair souffrent d'importants *overheads*.
Une tâche ayant l'identifiant ID est assignée d'une manière bijective au cœur $ID + 1$. Chaque test d'expérimentation dure 10 secondes, car comme nous l'avons évoqué au chapitre précédent, cette durée permet de stabiliser les données dans le cache. Durant ces 10 secondes de test, les STMs que nous évaluons effectuent les trois opérations citées auparavant.

Métriques étudiées

Afin d'évaluer les performances de la RT-STM munie des protocoles de concurrence 1W et MW, nous définissons trois métriques pour l'ensemble de nos tests :

- La bande passante du système (*throughput*) qui définit le nombre de transactions réussies dans le respect de leurs contraintes temporelles par unité de temps.
- La progression globale des transactions (*global progress*) qui définit le ratio du nombre de transactions réussies dans le respect de leurs contraintes de temps par rapport au nombre total de transactions lancées.
- Le ratio de temps de rejoue des transactions par tâche (*rollback times*) qui définit la proportion de temps gaspillé à rejouer des transactions par rapport à la durée d'exécution totale de la tâche.

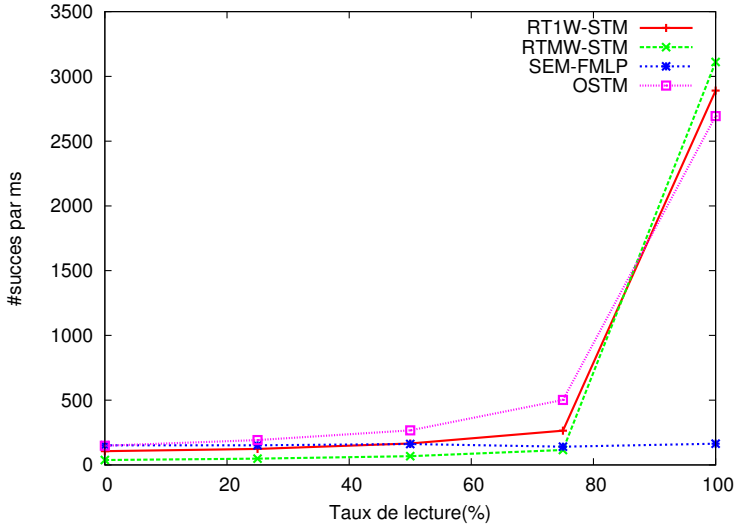
Paramètres du test

Les paramètres de variation en entrée de nos tests sont :

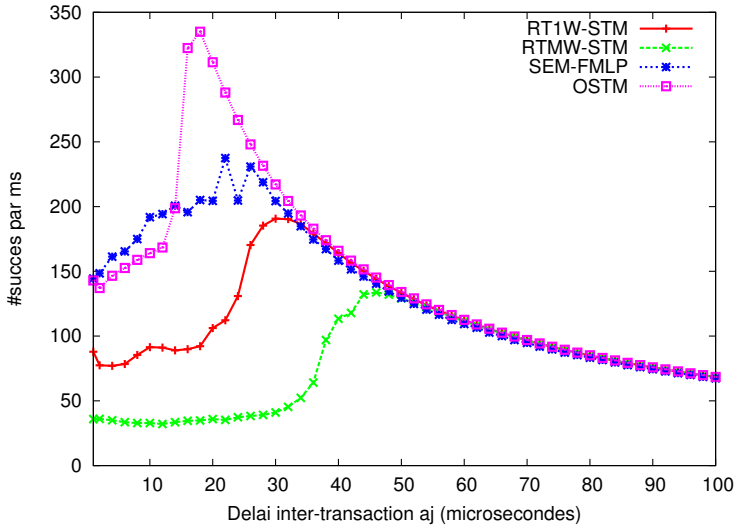
- Le taux de lecture qui définit pour les tâches la proportion des objets accédés en lecture (opérations *read*). Il contrôle donc le degré de contention entre les transactions.
- Le facteur d'utilisation des cœurs U_p.
- Le délai inter-transactions au sein d'une même tâche symbolisé par a_j ($a_j = 0$ correspond à une tâche n'exécutant que du code transactionnel).
- La durée du traitement transactionnel symbolisée par σ_j (*i.e*, la durée de manipulation des objets au sein d'une transaction). Ne sont pas considérés les temps d'exécution des procédures d'ouverture d'objets et du commit de la transaction. $\sigma_j = 0$ correspond à une transaction de longueur minimale.
- Le nombre d'objets qui représente la taille du tableau d'objets.

6.5.2 La bande passante du système

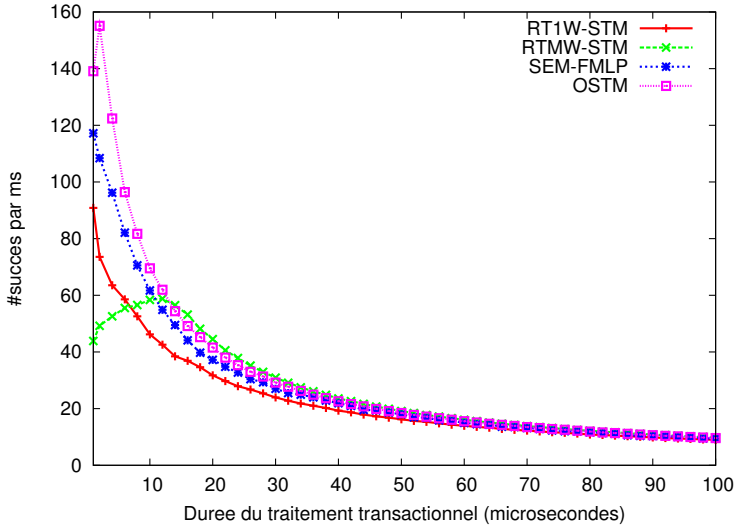
La Figure 6.3 montre l'évaluation de la bande passante de nos protocoles. L'évaluation est faite en comparaison avec le protocole bloquant FMLP basé sur des sémaphores, et le protocole non bloquant de la OSTM. Sous une forte contention (voir Figure 6.3-(a)) tous les protocoles ont de faibles performances. Dans ce cas précis, nos protocoles ont une faible performance comparée à la fois à celle du FMLP et de la OSTM. Mais en augmentant le taux de lecture, les mécanismes des trois protocoles transactionnels à savoir 1W, MW et celui de la OSTM, prennent effet et engendrent une meilleure bande passante que le FMLP. Toutefois, cet effet n'est pas le même entre le 1W et le MW. La bande passante du 1W augmente plus rapidement que le MW, à cause du mécanisme relativement lourd du protocole MW. En effet, le MW exécute fréquemment la procédure DCASG, qui exécute à son tour une double instruction CAS. En outre, au moment du commit, le MW effectue des tests supplémentaires lors de l'appel à la procédure DCASG.
Par ailleurs, nous pouvons citer deux raisons essentielles qui rendent la



(a) $\sigma_j = 0\mu$s et $a_j = 0\mu$s.



(b) 0% de lecture et $\sigma_j = 0\mu$s.



(c) 0% de lecture et $a_j = 0\mu$s.

FIGURE 6.3 – *Bande passante pour* $U_p = 100\%$.

OSTM meilleure que nos deux protocoles en termes de bande passante. La première est l'important coût du CASG et c'est d'ailleurs le compromis que nous faisons dans nos protocoles pour qu'ils intègrent les contraintes de temps.
La seconde raison est directement liée à la gestion de contention du bus par le processeur multicœur que nous utilisons. Cette gestion de contention intervient quand il y a une haute fréquence d'accès au tableau RA par les transactions en mode *helping*. Le fait de mettre à jour une case mémoire du tableau RA par plusieurs threads, provoque le déclenchement du mécanisme de gestion de contention du bus d'Intel Xeon.
Nous avons cherché à comprendre la manière dont agit la gestion matérielle de contention sur le tableau RA, en testant l'accès parallèle en lecture/écriture à des tableaux de tailles différentes, et donc à des contentions différentes. Les résultats sont rapportés sur la Figure 6.4. Ces résultats montrent qu'il y a une sorte d'"écroulement" lorsque le bus est fortement sollicité, mais qu'avec une taille d'un tableau inférieure à 16 éléments, la bande passante reste bien répartie sur tous les cœurs.

Au-delà de 90% de lectures (voir Figure 6.3-(a)) nos protocoles deviennent les plus performants en termes de bande passante, et ce, malgré l'effet négatif de la gestion du bus que peut causer l'enregistrement systématique des transactions dans le tableau RA lors de l'ouverture d'objets en lecture seule. Ceci s'explique par le fait que contrairement à nos protocoles, la OSTM effectue des tests supplémentaires lors de l'ouverture d'objets en lecture, ce qui influe négativement sur sa bande passante.

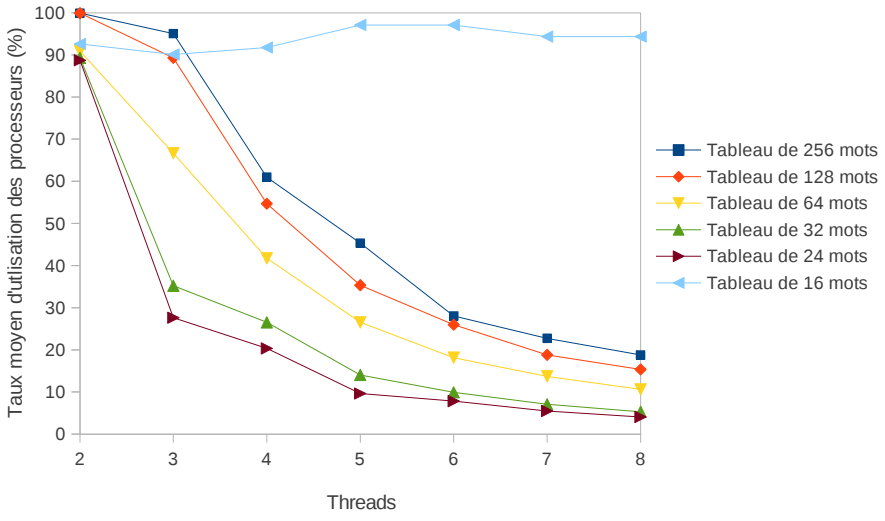
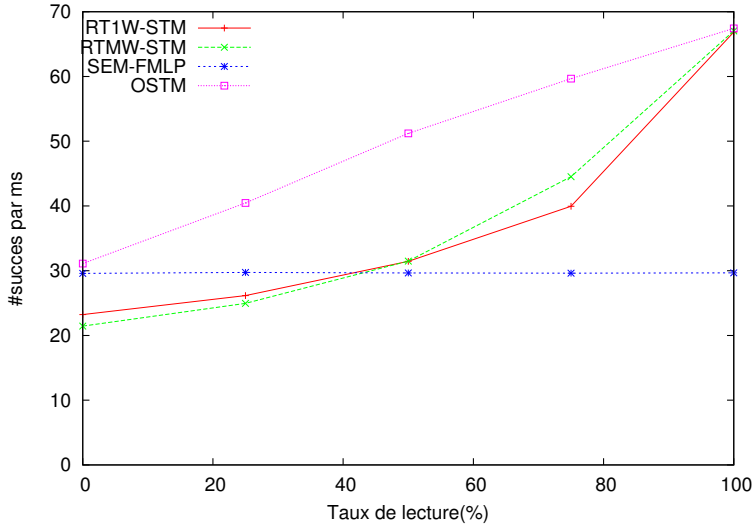
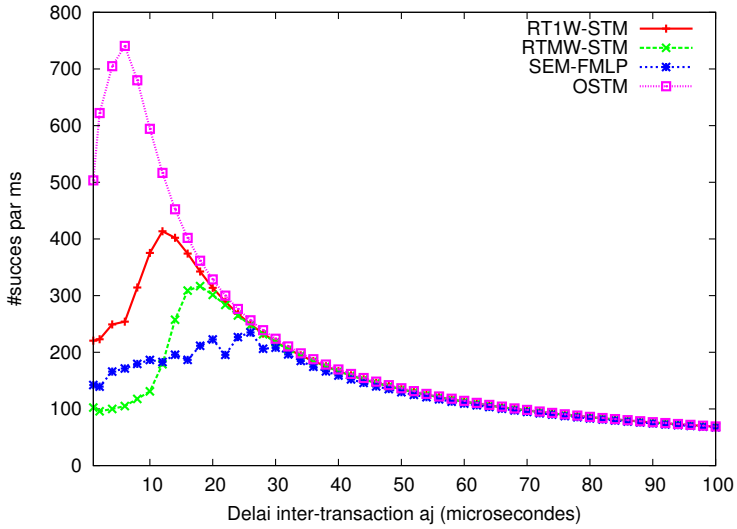


FIGURE 6.4 – *Gestion de la contention du bus Intel Xeon 5552*

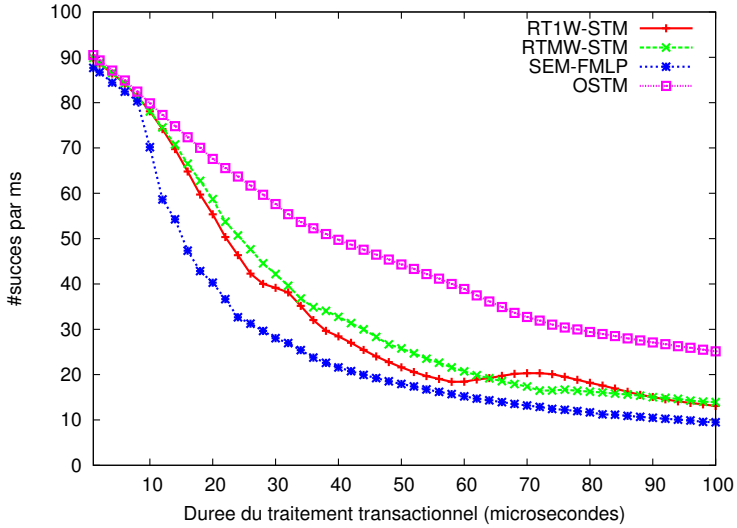
La Figure 6.3-(b) montre le cas d'une forte contention (*i.e*, aucune lecture n'est effectuée) en faisant varier le paramètre a_j. Le même phénomène peut être observé à la Figure 6.3-(a) dans le cas d'une forte contention. Le paramètre a_j influe également sur le degré de contention. Plus les transactions sont temporellement distantes les unes des autres, et moins les conflits sont fréquents. Mais quand la valeur du paramètre a_j dépasse



(a) $\sigma_j = 0\mu$s et $a_j = 75\mu$s.



(b) 75% de lecture et $\sigma_j = 0\mu$s.



(c) 75% de lecture et $a_j = 75\mu$s.

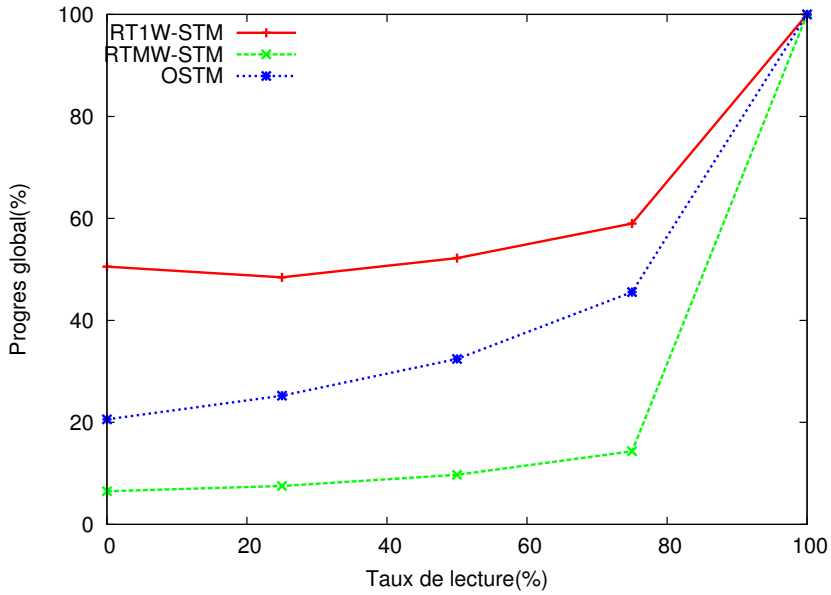
FIGURE 6.5 – *Bande passante pour* $U_p = 50\%$.

le point d'équilibre entre le degré de contention et le nombre relatif de transactions générées par les tâches, alors la bande passante diminue pour tous les protocoles. La rapidité avec laquelle le protocole atteint ce point d'équilibre dépend de sa performance en termes de bande passante.
L'influence de la longueur des transactions σ_j est rapportée sur la Figure 6.3-(c). Pour les transactions longues, notre protocole MW offre les meilleures performances. Ceci confirme par ailleurs que la faible performance obtenue pour le MW dans les autres cas de configurations, est principalement causée par la lourdeur de son code plutôt que par sa politique de contrôle de concurrence.

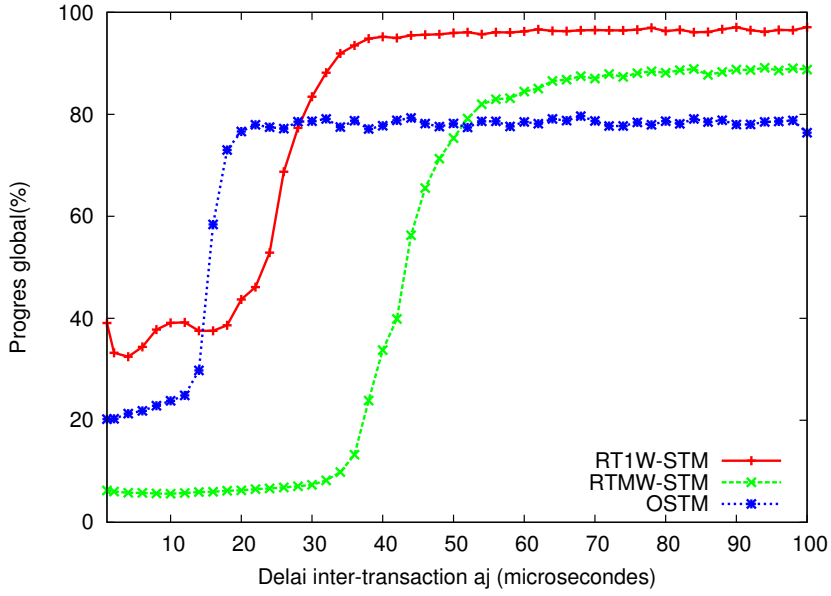
Le cas général observé dans la grande majorité des applications (taux de 75% de lecture) est présenté dans la Figure 6.5. Le degré de contention est diminué, avec $a_j = 75\mu$s. Cette configuration fait que notre protocole MW surpasse le 1W en termes de bande passante comme l'atteste la Figure 6.5-(a). La variation du paramètre a_j dans la Figure 6.5-(b) montre que nos deux protocoles offrent une meilleure bande passante dès que $a_j \geq 14\mu$s par rapport au protocole bloquant FMLP. Cette performance peut aussi être observée dans la Figure 6.5-(c).
A noter que nous avons fixé d'une manière arbitraire $a_j = 75\mu$s et nous considérons que ce temps d'arrivée de transactions est plus réaliste pour nous que la valeur de 14μs préalablement déterminée.
Par ailleurs, nous ne présentons pas ici l'influence du taux d'utilisation du processeur sur la bande passante, car la variation du paramètre U_p d'une manière isolée des autres, maintient les résultats constants. Mais il est important de noter que dans le cas de la Figure 6.5-(c), en appliquant un $U_p = 100\%$, les performances du 1W sont les seules à chuter en dessous de celles du FMLP.

6.5.3 La progression globale

Le protocole FMLP n'est pas concerné par cette métrique telle que nous l'avons définie. Celle-ci en effet, suppose un rejoue des transactions qui intrinsèquement n'a pas de sens pour le protocole FMLP.
La Figure 6.6-(a) montre que notre protocole 1W offre de meilleures performances que la OSTM. Le MW quant à lui, a de de faibles performances sous une forte contention, mais offre une meilleure progression que la OSTM à partir de la valeur $a_j = 50\mu$s (voir Figure 6.6-(b)). Pour le MW, la valeur de $a_j = 50\mu$s correspond également dans ce cas au point d'équilibre entre la contention et le nombre relatif de transactions générées.
Après dépassement de tous les points d'équilibres des protocoles, nos deux protocoles ont toujours une meilleure performance par rapport à la OSTM. Ceci parce que la OSTM n'offre pas la meilleure stratégie pour une telle situation. En effet, puisque dans ce cas, toutes les transactions effectuent uniquement des opérations d'écriture, alors le contrôleur de concurrence devrait servir les transactions à la fréquence à laquelle elles apparaissent afin de maintenir une progression globale.

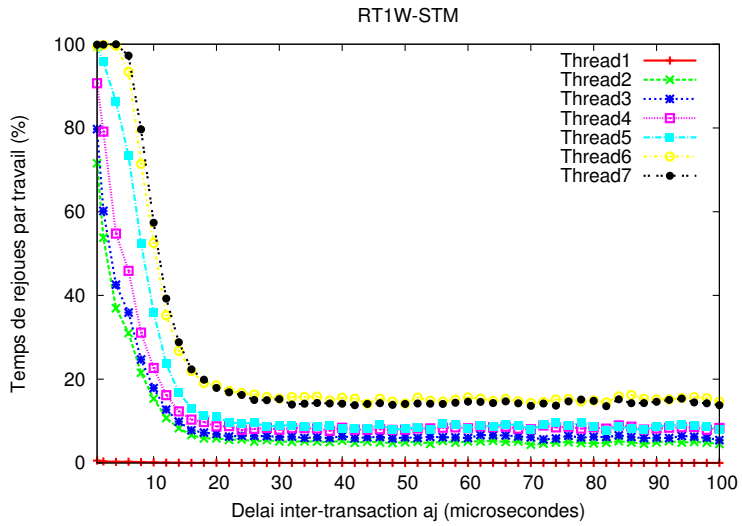


(a) $\sigma_j = 0\mu$s et $a_i = 0\mu$s

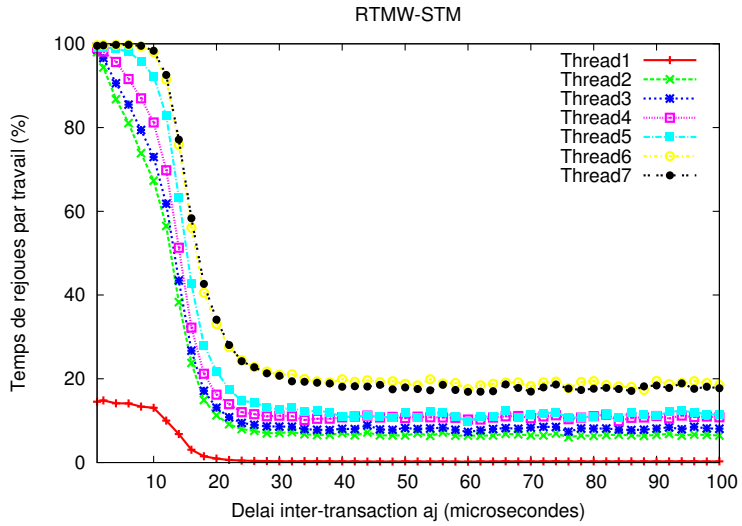


(b) 0% de lecture et $\sigma_j = 0\mu$s.

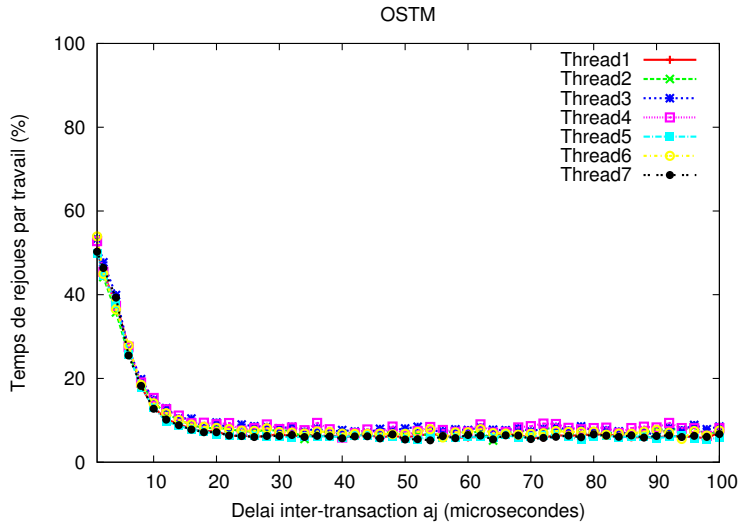
FIGURE 6.6 – *Progression globale pour* $U_p = 100\%$.



(a) Rollbacks distribués selon p_i.



(b) Rollbacks distribués selon p_i.



(c) Rollbacks équidistribués.

FIGURE 6.7 – *Temps de rollbacks en fonction du délai inter-transaction.*

6.5.4 Le ratio de temps de rejoue

Afin d'évaluer le temps de rejoue des transactions, nous fixons ici $U_p = 50\%$, $a_j = 75\mu$s et le taux de lecture à 75%. Comme montré précédemment, avec ces paramètres, nos protocoles offrent de meilleures performances que le FMLP. En outre, un taux de lecture à 75% est la valeur la plus usuelle dans la littérature, et reflète souvent le cas pratique (Fraser 2003). A rappeler que la métrique concernant les *rollbacks* utilisée ici, est donnée comme un taux moyen et décrit le temps de *rollbacks* des transactions observée vis-à-vis de la durée d'exécution c_i des tâches.

Effet du délai inter-transactions

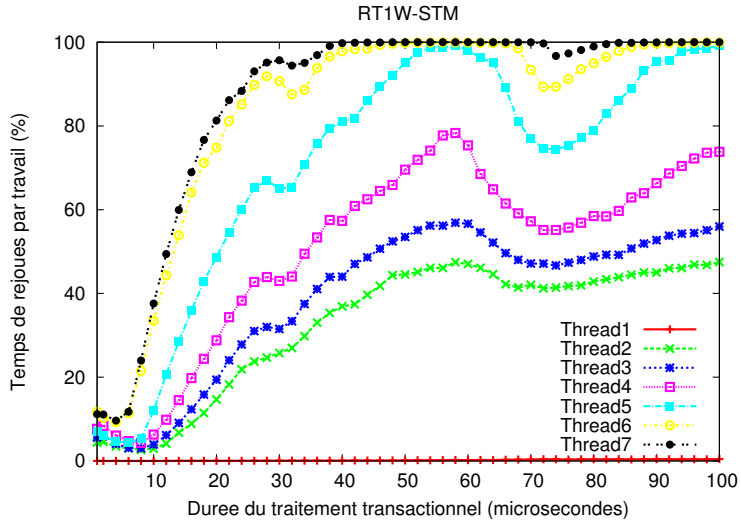
La Figure 6.7 montre la différence entre nos protocoles et le protocole classique de la OSTM. A la différence de la OSTM, les transactions sont rejouées selon la priorité de la tâche qui les exécute (voir Figure 6.7-(a) et (b)). Nous rappelons que cette priorité est basée sur les échéances $D_i = P_i$, d'où la distribution des rejoues des transactions selon les périodes P_i.
En augmentant la valeur de a_j ($a_j \geq 20\mu$s) la contention diminue. Cette faible contention fait diminuer le taux de *rollbacks* dans les trois cas de figure ((a)-(b) et (c)).
Cependant, il y a une légère différence entre le 1W et le MW. Dans le protocole MW, pour des valeurs assez réduites de a_j, toutes les transactions de faibles priorités sont rejouées. Ceci indique que le MW effectue une meilleure répartition de *rollbacks* relativement au 1W, et ce, malgré la relative faible progression dont souffre le MW.

Effet de la longueur des transactions

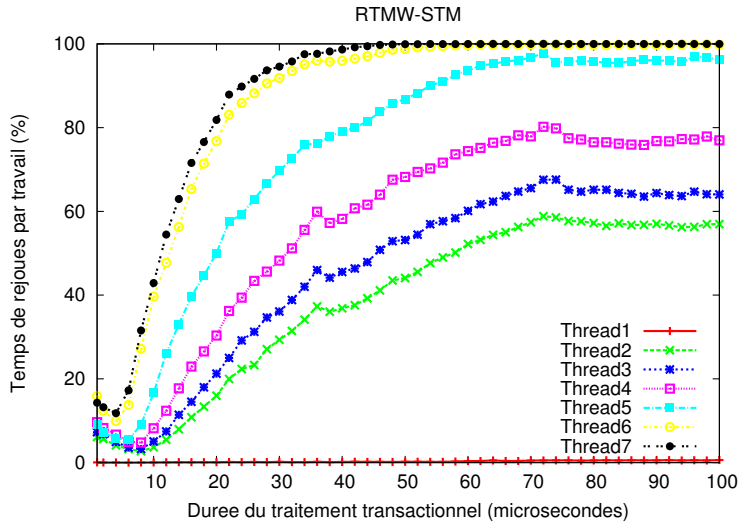
L'effet de σ_j est rapporté dans la Figure 6.8. Ici, l'effet de notre protocole sur la répartition des *rollbacks* est plus important que celui rapporté dans la Figure 6.7.
Plus le σ_j est grand, et plus l'effet de la répartition des *rollbacks* devient important. Quand une transaction de plus grande priorité marque des objets, elle les "garde" pour une longue période, et donc toute transaction de plus faible priorité, soit reboucle davantage sur le CASG (dans le cas du 1W), soit elle fait davantage de *rollbacks* (dans le cas du MW). Dans les deux cas de figure, le progression individuelle est réduite et le ratio de *rollbacks* est par conséquent augmenté.

Effet du taux d'utilisation du processeur

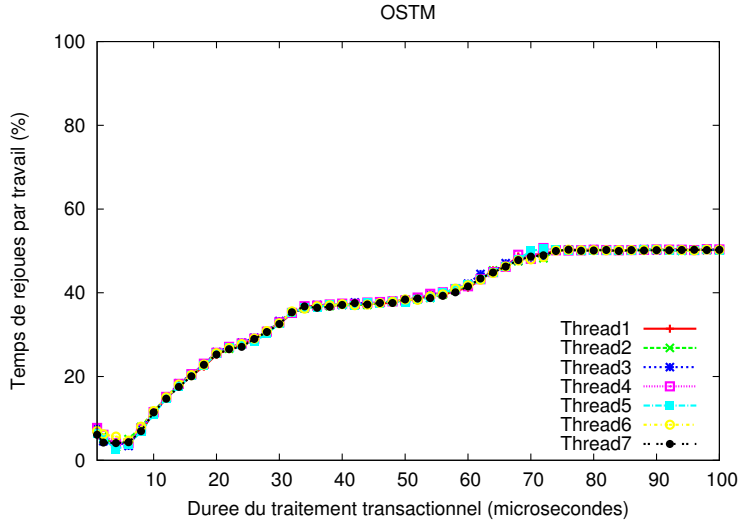
La Figure 6.9 montre qu'à la différence de la OSTM, les rejoues des transactions dans notre protocole sont distribués selon la priorité de la tâche (voir Figure 6.9-(a) et (b))
L'utilisation du processeur a un impact assez faible sur les trois protocoles car l'effet du contrôleur de concurrence est lié en grande partie au degré de contention.
Dans le cas de nos deux protocoles ((a)-(b)), l'effet de répartition des rejoues a tendance à diminuer lorsque le paramètre U_p augmente. En réalité,



(a) Rollbacks distribués selon p_i.

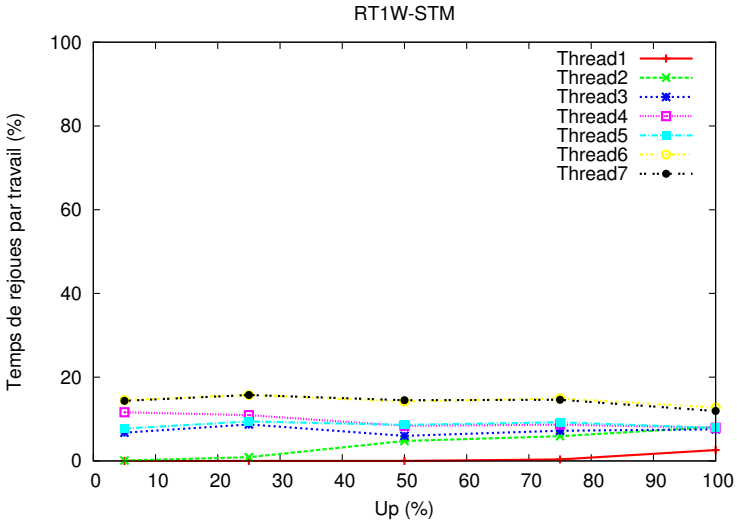


(b) Rollbacks distribués selon p_i.

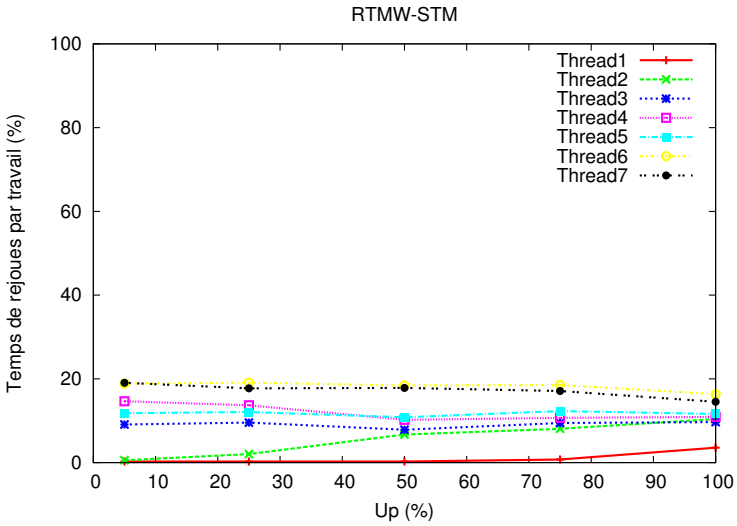


(c) Rollbacks équidistribués

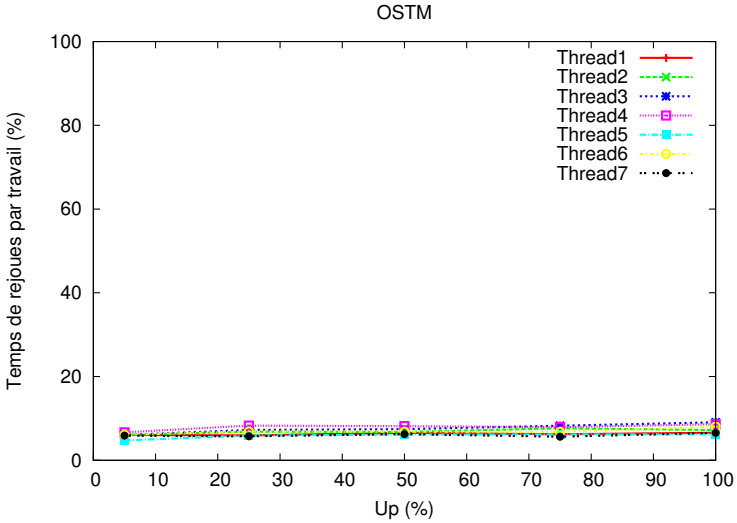
FIGURE 6.8 – *Temps de rollbacks en fonction de la durée du traitement transactionnel.*



(a) Rollbacks distribués selon p_i.

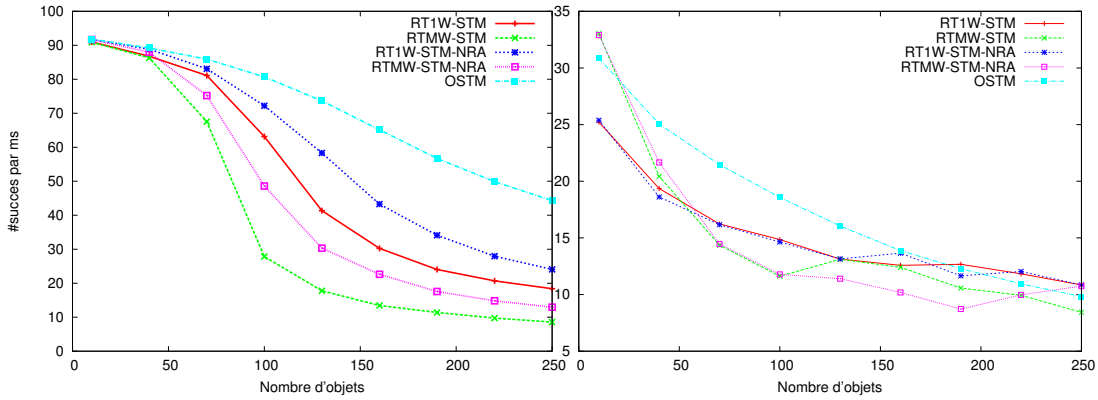


(b) Rollbacks distribués selon p_i.



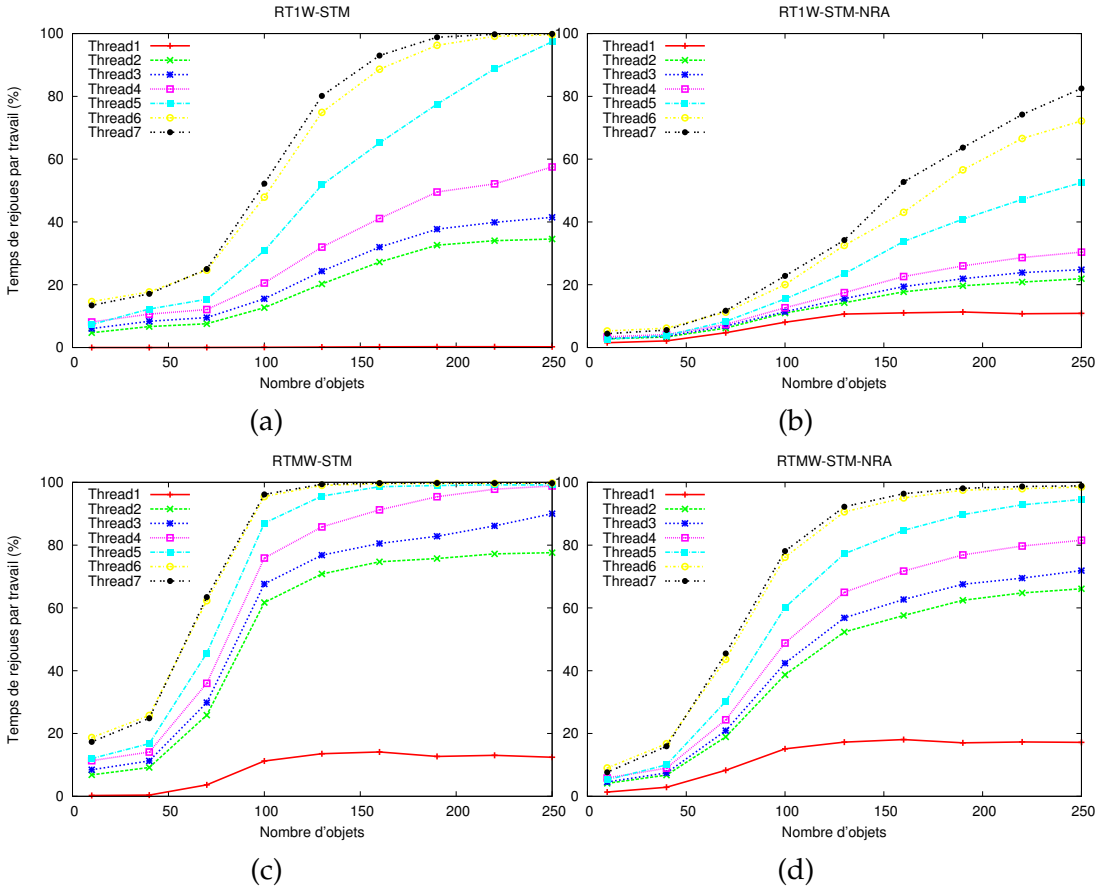
(c) Rollbacks équidistribués.

FIGURE 6.9 – *Temps de rollbacks en fonction du taux d'utilisation du processeur.*



(a) 75% de lecture et $U_p = 50\%$. (b) 0% de lecture et $U_p = 100\%$.

FIGURE 6.10 – *Effet du nombre d'objets sur la bande passante.*



(a) (b)

(c) (d)

FIGURE 6.11 – *Effet du nombre d'objets sur les temps de rollbacks.*

cet effet de répartition reste toujours constant mais la diminution observée est due à la métrique de mesure des *rollbacks*. En effet, l'augmentation de U_p réduit le nombre de travaux par tâche induisant ainsi un taux plus important de *rollbacks* par travail. Ce phénomène s'accentue donc d'une manière proportionnelle avec la priorité des tâches (et à plus forte raison pour la Tâche 1 qui est la plus prioritaire).

Effet du nombre d'objets

Pour montrer l'effet du nombre d'objets, et plus particulièrement l'effet du tableau RA sur les performances de nos protocoles, nous avons créé pour chacun de nos protocoles deux variantes qui n'utilisent pas le tableau RA. Nous avons appelé ces deux variantes 1W-NRA et MW-NRA. Celles-ci désignent respectivement 1W et MW, et n'incluent dans aucune de leurs procédures (ouverture d'objets en lecture/écriture et du commit) la prise en charge du RA.

Vis-à-vis de la bande passante

L'effet du nombre d'objets sur la bande passante de nos protocoles varie selon le degré de contention. Sous faible contention (voir Figure 6.10-(a)) le tableau RA influe négativement sur la bande passante de nos deux protocoles. Cette diminution de bande passante est prévisible à cause du temps que peut induire le parcours du tableau RA, et c'est d'ailleurs le compromis que nous avons fait lors de la conception du RA. En augmentant la contention, le tableau RA est moins sollicité, et la bande passante reste donc quasiment la même pour toutes les variantes (voir Figure 6.10-(b)).

Vis-à-vis des *rollbacks*

Le tableau RA permet d'empêcher une transaction d'écrire un objet ouvert en lecture par une autre transaction de plus grande priorité.
La Figure 6.11 montre l'effet du nombre d'objets sur les rejoues des transactions avec un taux de lecture de 75% et une charge $U_p = 50\%$. L'effet du RA sur la distribution des rollbacks, est plus important pour le 1W (cf. Figure 6.11-(a)(b)) que pour le MW (voir Figure 6.11-(c)(d)). La variante MW-NRA affiche de meilleures performances que la variante 1W-NRA (cf. Figure 6.11-(b)(d)). Cette différence de performance est due à la nature même des deux protocoles. En effet, le MW permet d'avantage de *helping* que le 1W. Le mécanisme du *helping* pénalise le MW en bande passante, mais en contre-partie, offre de meilleures performances temps réel que le 1W.

Conclusion

Le but principal de ce chapitre était de concevoir et proposer des protocoles de concurrence efficaces pour une mémoire transactionnelle logicielle adaptée au temps réel *soft*. Pour cela, nous nous sommes appuyés sur la RT-STM qui munit les transactions de contraintes de temps. L'objectif consistait à non seulement respecter un maximum de contraintes

de temps des transactions, mais aussi à assurer une bande passante acceptable pour le système. Autrement dit, il s'agissait de répartir la bande passante entre les tâches en fonction de leur priorité. Pour ce faire, nous avons établi un modèle transactionnel étendu et proposé de nouveaux protocoles.
Les protocoles que nous avons conçus se basent pour l'un sur un seul écrivain (1W), et pour l'autre, sur l'exécution parallèle des écrivains (MW). Le 1W ressemble au 2PL-HP mais avec intégration du mécanisme du *helping*. Le MW quant à lui, est un protocole lock-free avec prise en compte des contraintes de temps. Les deux protocoles sont à grain fin et utilisent l'instruction de base CAS ainsi que des variantes que nous avons construites à savoir CASG et DCASG.
Sur une plateforme réelle dotée de 8 cœurs, nous avons comparé le 1W et le MW à des protocoles de référence dans la littérature, aussi bien pour le domaine classique (protocole non bloquant de la OSTM) que temps réel (protocole bloquant FMLP). Les principaux résultats obtenus peuvent être synthétisés comme suit :
(**i**) Sous forte contention, et pour des transactions courtes, les protocoles 1W et MW ont une bande passante inférieure au FMLP et au protocole de la OSTM.
(**ii**) Sous une contention usuelle (75% de lecture), la bande passante des deux protocoles 1W et MW se situe entre celles du protocole de la OSTM et du FMLP.
(**iii**) Dans le cas de faible contention et avec des transactions longues, le MW a une bande passante supérieure au protocole 1W. Mais globalement, le 1W a de meilleures performances que le MW en termes de bande passante.
(**iv**) Contrairement à la OSTM, nos deux protocoles 1W et MW répartissent les temps de *rollbacks* des transactions (et donc la bande passante) entre les tâches selon leur priorité.
(**v**) Le MW répartit mieux les temps des *rollbacks* entre les transactions que le 1W.

Troisième partie

Extension aux contraintes *firm*

RT-STM en environnement firm

7

Sommaire

7.1 Motivations et objectif

Dans le précédent chapitre, nous avons présenté la conception des protocoles 1W et MW pour la RT-STM. Ces protocoles permettent de répartir la bande passante du système entre les tâches en fonction de leur priorité. Plus précisément, le contrôleur de concurrence de ces protocoles se base sur les priorités des tâches temps réel pour ordonnancer les conflits entre les transactions. Ainsi, le taux de progression (resp. de rejoues) des transactions d'une tâche, est proportionnel (resp. inversement proportionnel) à la priorité de la tâche.
Cependant, étant donné le contexte temps réel *soft*, la RT-STM assure au mieux (best effort) ce taux de progression. Pour une utilisation dans un environnement plus critique dans lequel des garanties d'accès aux ressources partagées doivent être affichées et respectées, il est nécessaire de garantir en amont la progression de chaque tâche temps réel.
A titre illustratif, soit un système de télécommunication qui effectue une recherche sur un répertoire contenant un très grand nombre de numéros d'abonnés. Ces numéros d'abonnés sont fréquemment insérés et supprimés du répertoire. Si l'on suppose que les requêtes de recherche ne sont pas critiques, alors le système peut lancer une transaction temps réel *soft*. Maintenant, si l'utilisateur définit un délai maximal suite auquel il souhaite récupérer les résultats de sa requête, le système devient relativement plus critique. Le respect du délai spécifié par l'utilisateur, va directement dépendre de la progression de la transaction temps réel de la tâche qui a traité la requête.

L'objectif principal dans ce chapitre est de montrer qu'il est possible d'adapter notre RT-STM de manière à garantir la progression de chaque tâche selon un taux prédéfini par l'utilisateur. Cette progression peut être garantie au sein de chaque tâche, sans pour autant apporter des modifications aux protocoles de synchronisation sous-jacents en apposant simplement des contraintes en amont sur les durées d'exécution au pire-cas des tâches. Autrement dit, pour assurer la progression de chaque tâche, la synchronisation de la RT-STM n'a pas besoin d'être de type wait-free. En effet, nous ne cherchons pas à garantir la progression des transactions à tout instant, mais plutôt sur un intervalle de temps donné. Il s'agit alors de garantir uniquement l'existence d'un ensemble de transactions qui progressent en fonction de leurs priorités. De plus, si cet ensemble de transactions existe dans le cas où toutes les transactions sont en conflits (*i.e*, situation du pire cas d'exécution des transactions) nous pouvons alors être sûr que le taux de progression des tâches peut toujours être garanti sur un intervalle de temps donné.

7.2 Modèles et algorithmes existants

Dans la littérature, les systèmes temps réel *firm* sont définis comme étant des systèmes qui tolèrent de temps à autre le dépassement des échéances. A la différence des systèmes à contraintes *soft*, ces dépassements sont quantifiés. Des modèles de tâche étendus associés à des ordonnanceurs adéquats permettent la prise en compte de cette quantification. Dans ce

qui suit, nous présentons un de ces modèles. Pour le lecteur intéressé par les autres modèles existants, nous l'invitons à consulter (Marchand 2006).

7.2.1 Le modèle de tâche (m,k)-firm

Dans un système temps réel, les contraintes (m,k)-firm introduites par Ramanathan et Hamdaoui (1995) traduisent le fait que des dépassements d'échéance sont tolérés au niveau des tâches. Dans ce modèle, au moins m travaux d'une tâche doivent respecter leurs échéances dans une fenêtre de k travaux consécutifs.
Bernat (1998) a montré la nécessité d'utiliser deux paramètres. En effet, le taux de succès d'une tâche (le ratio du nombre total de travaux respectant leurs échéances sur le nombre total de travaux de la tâche) ne fournit pas suffisamment d'informations sur l'exécution de cette tâche. Par exemple, un taux de succès de 90% est équivalent à un travail manquant son échéance parmi 10 travaux, ou bien à 100 travaux parmi 1000. Si les travaux qui ne respectent pas leurs échéances sont consécutifs, les résultats sont alors beaucoup mois précis dans le cas de 100 travaux parmi 1000.
L'utilisation de deux paramètres permet de pallier ce problème. Le premier paramètre indique le nombre de travaux qui doivent respecter leurs échéances, et le second limite la fréquence à laquelle les travaux peuvent manquer leurs échéances. Ainsi, pour l'exemple précédent, une tâche avec des contraintes (9,10)-firm est considérée comme plus critique qu'une tâche sous contraintes (900,1000)-firm.

Algorithmes d'ordonnancement (m,k)-firm

Les algorithmes qui sont proposés dans la littérature (Ramanathan et Hamdaoui 1995, Ramanathan 1999, Wang et al. 2002, Song 2002) peuvent être regroupés en deux grandes familles : les algorithmes à priorités dynamiques et les algorithmes à priorité statiques.

Algorithmes à priorités dynamiques. Pour ce type d'algorithmes, la priorité de chaque travail est déterminée en fonction de l'état courant du système et peut donc varier tout au long de la vie d'une tâche applicative. Par exemple, l'algorithme *Distance Based Priority* (DBP) de Ramanathan et Hamdaoui (1995) calcule dynamiquement le nombre d'échéances qui peuvent être dépassées en se basant sur l'historique des k derniers travaux de la tâche. Ce nombre représente la distance pour chaque tâche d'un état dit d'*échec* (*i.e*, au moins $k - m + 1$ des travaux de la tâche ont manqué leur échéance).

Algorithmes à priorités statiques. Ces algorithmes affectent une priorité statique aux tâches. Cette priorité est généralement calculée sur la base d'un ratio fonction des paramètres m et k, et est souvent de la forme m/k. L'algorithme détermine hors ligne (off-line) les travaux obligatoires des tâches ainsi que leurs travaux optionnels.
Par exemple, l'algorithme *Enhanced Rate Monotonic* (ERM) associe des identifiants aux travaux afin de déterminer ceux qui sont obligatoires et

optionnels (Ramanathan 1999). Dans ERM, le travail j d'une tâche τ_i est dit obligatoire si l'identifiant a associé à j satisfait l'équation suivante :

$$a = |\lceil \frac{a.m}{k} \rceil . \frac{k}{m}|$$

Les travaux obligatoires sont ensuite ordonnancés par un algorithme statique tel que RM. Les travaux optionnels quant à eux, sont ordonnancés en FIFO.

Pour résumer, les algorithmes dynamiques fournissent une approche probabiliste en tenant compte des éventuels changements du système, alors que les algorithmes statiques fournissent une approche totalement déterministe.

7.2.2 Le modèle transactionnel $\binom{m}{k}$-firm

Les auteurs Haubert et al. (2004) ont adapté le modèle (m,k)-firm au transactionnel temps réel. Leur modèle est noté $\binom{m}{k}$-firm afin de le différencier du (m,k)-firm. A la différence du (m,k)-firm, le modèle de ces auteurs est destiné aux environnements distribués. Cette différence est majeure car le modèle (m,k)-firm suppose des tâches périodiques, alors que les tâches au sein des systèmes distribués ne le sont généralement pas.
Dans le modèle $\binom{m}{k}$-firm, chaque transaction temps réel distribuée T_j est divisée en sous-transactions, et le nombre de sous-transactions de T_j est noté k_j. Chaque sous-transaction se voit attribuer un poids en fonction de la criticité des données auxquelles elle accède. Si $\binom{m_j}{k_j}$ représente les contraintes $\binom{m}{k}$-firm de T_j, les m_j sous-transactions de T_j dont les poids sont les plus forts sont marquées comme étant obligatoires, et les autres sont marquées comme étant optionnelles.

7.3 Modélisation et analyse de la RT-STM *firm*

Dans ce paragraphe nous décrivons l'adaptation de la RT-STM au contexte temps réel *firm*. Nous allons d'abord décrire le nouveau modèle de la RT-STM. Puis, nous allons déterminer les durées d'exécution au pire cas des transactions de manière à ce que la RT-STM puisse respecter les contraintes *firm*.

7.3.1 Nouveau modèle sous contraintes m-firm

Dans les paragraphes précédents, nous avons vu que la plupart des solutions pour les systèmes *firm* se basent sur des modèles à deux paramètres, typiquement m et k. En effet, les modèles (m,k)-firm pour les tâches temps réel périodiques (Ramanathan et Hamdaoui 1995) ainsi que $\binom{m}{k}$-firm pour les transactions distribuées (Haubert et al. 2004), supposent la connaissance de ces deux paramètres pour quantifier d'une manière précise le dépassement d'échéances.
Cependant, le modèle pour les mémoires transactionnelles que nous avons proposé au chapitre 5, suppose que le nombre total de transactions par travail n'est pas connu *a priori*.

Dans ce paragraphe, nous proposons une adaptation de notre modèle au contexte *firm*. Ce modèle nous l'avons nommé *m-firm*. La signification que nous donnons aux paramètre m ressemble à celle donnée dans le modèle $\binom{m}{k}$-firm de (Haubert et al. 2004) mais diffère du modèle (m,k)-firm classique.
Les transactions dites *m-firm* ont comme paramètre additionnel le nombre minimum m de transactions dites *obligatoires* qui doivent respecter leurs échéances. Par contre, le nombre total k de transactions n'est pas disponible explicitement et est déterminé expérimentalement. En effet, étant données les heuristiques sur lesquelles se base la RT-STM, il est difficile de prédire hors-ligne les k transactions qui peuvent être générées. Le nombre de transactions $k - m$ représente les transactions *optionnelles*. Par conséquent, les transactions optionnelles ne peuvent pas être explicitement spécifiées dans notre cas, et ne sont exécutées qu'après terminaison des m transactions obligatoires.
En outre, étant donné que nous ne modifions pas la RT-STM au niveau de ses protocoles, le paramètre m est ainsi pris en compte en amont. Autrement dit, le paramètre m est spécifié au niveau de la tâche.

Modèle de tâches étendu sous contraintes m-firm

Soit un ensemble de tâches τ tel que $|\tau| = N$. Chaque tâche sporadique qui s'exécute sur une plateforme multicœur composée de M processeurs, et partage des ressources accédées via des transactions, est définie comme suit.

$$\tau_i = (r_i, C_i, P_i, D_i, \Omega_i, m_i)$$

Une tâche τ_i est caractérisée par une date de réveil r_i, une durée d'exécution au pire cas C_i, un délai minimum P_i d'inter-arrivée des travaux, un délai critique D_i, un ensemble Ω_i de k transactions ($\Omega_i = \{T_1, T_2, ...T_k\}$), et un nombre minimum m_i de transactions dans Ω_i qui doivent respecter leurs échéances. La tâche τ_i doit se terminer avant sa date d'échéance $d_i = r_i + D_i$.

Modèle transactionnel étendu sous contraintes m-firm

Au vu des résultats obtenus au chapitre 6, de nouveaux paramètres apparaissent influents sur le contrôleur de concurrence et doivent donc être intégrés au modèle transactionnel.
Soit alors un système multicœur composé d'un ensemble de processeurs M et d'un ensemble de processeurs exclusivement transactionnels M_T.
Ω_i est l'ensemble fini des transactions T_j^i, associé à la tâche τ_i. Chaque transaction T_j^i est supposée sans entrées/sorties, non-imbriquée, et s'exécutant sur $M \cup M_T$. $\forall T_j^i \in \Omega_i$, alors

$$T_j^i = (s_j^i, D_j^i, a_j^i, \sigma_j^i)$$

Où s_j^i représente la date d'arrivée de la transaction, D_j son délai critique, a_j^i est le délai minimal inter-transactions séparant T_j^i des transactions T_{j-1}^i et T_{j+1}^i, et σ_j^i la durée du traitement transactionnel. Lors de l'exécution de

T_j^i, celle-ci accède aux objets avec un taux de lecture supposé constant L_j^i. La transaction T_j^i possède un nombre de rejoues $n_j^i \geq 0$, et se termine en cas de succès à une date f_j^i. Ainsi, le temps de réponse de T_j^i est symbolisé par $R_j^i = f_j^i - s_j^i$. La date d'échéance de T_j^i est symbolisée par $d_j^i = s_j^i + D_j^i$.

7.3.2 Nouveau modèle sous contraintes de QoS

L'un des objectifs de ce chapitre, est d'adapter notre RT-STM de manière à faire progresser chaque tâche selon un taux prédéfini par l'utilisateur qui représentera le niveau de qualité de service (QoS) de la tâche vis-à-vis des accès aux ressources partagées. Dans le modèle précédent m-firm, le nombre m_i de transactions spécifié par l'utilisateur ne doit pas dépasser le nombre total de transactions k_i sous peine d'avoir des transactions qui dépassent leur échéance. Ainsi, quand m_i est inférieur à k_i, les m_i transactions respectent leurs échéances. Dans le cas contraire, le résultat n'est pas garanti.
Nous pouvons alors définir le ratio m_i/k_i comme étant le niveau de qualité de service (QoS) associé à la tâche τ_i. Il y a donc autant de niveaux de QoS que de valeurs entre m_i et k_i.
Etant donné que le m_i est fourni par l'utilisateur, celui-ci doit donc se soucier du nombre de transactions qui doivent respecter leur échéance au sein de chaque tâche. Autrement dit, pour exprimer la progression d'une tâche, l'utilisateur doit raisonner sur le nombre de transactions qui doivent progresser au sein des travaux de la tâche, ce qui peut s'avérer complexe lors de la programmation d'applications temps réel. Par conséquent, nous pensons que dans le contexte de mémoires transactionnelles, la spécification de la QoS doit être faite sous forme de ratio au lieu d'un nombre de transactions.

Modèle de tâche sous contraintes de QoS

Nous redéfinissons la tâche sporadique τ_i de la manière suivante :

$$\tau_i = (r_i, C_i, P_i, D_i, \Omega_i, q_i)$$

Où le nouveau paramètre q_i désigne la QoS de la tâche τ_i. Le paramètre q_i indique le taux avec lequel la transaction $T_k \in \Omega_i$ doit progresser sur un intervalle de temps donné. Ainsi, la satisfaction de q_i n'est garantie que si Ω_i est *m-firm* (cf. Corollaire 7.6 ci-après).
Par ailleurs, puisque nous ne faisons pas de modifications au niveau de la RT-STM, le modèle transactionnel de la RT-STM reste inchangé.

7.3.3 Analyse du WCET de transactions m-firm

D'un point de vue de la RT-STM, le temps d'exécution au pire cas (Worst-Case Execution Time ou WCET en anglais) surgit lorsque toutes les transactions sont en conflit entre elles. Le but de l'analyse de ce pire cas, est d'une part de borner le temps d'exécution des transactions et d'autre part, de nous permettre d'en déduire la configuration de tâches la mieux

adaptée à notre modèle m-firm.
Pour des raisons de simplification d'écriture, nous supposons comme dans le chapitre 6 que les identifiants des tâches sont ordonnés dans l'ordre inverse de leur priorité.

Théorème 7.1 *Le temps d'exécution au pire cas Ψ_j^i d'une transaction quelconque dans le système est tel que*

$$\forall T_j^i \in \bigcup_{\tau_i \in \tau} \Omega_i : \Psi_j^i = \sum_{j=1}^{m_i} (R_j^i + a_j^i) + \sum_{i-1} \sum_{j=1}^{m_{i-1}} (R_j^i + a_j^i)$$

Preuve. Dans le chapitre 5 (cf. 5.2.3) nous avons vu que la RT-STM pouvait assurer la progression d'au moins une transaction T_j^i puisqu'elle se base sur la OSTM, qui par sa nature lock-free, assure cette progression. En outre, nous savons que T_j^i est forcément la plus prioritaire d'après les heuristiques des protocoles 1W et MW. Ainsi la condition suivante s'avère toujours vraie au sein de la RT-STM :

$$\exists T_j^i \in \Omega_i : R_j^i \leq D_j \tag{7.1}$$

Pour qu'au pire cas la transaction T_j puisse toujours être exécutée au sein d'un travail de durée C_i, la condition (7.1) peut être trivialement reformulée comme suit :

$$\exists T_j^i \in \Omega_i : R_j^i \leq C_i \tag{7.2}$$

Par ailleurs, à un instant donné t, la transaction T_l^{i+1} ne peut *commiter* qu'après la terminaison de T_j^i, puisque T_l^{i+1} a une priorité supposée inférieure à celle de T_j^i et sous la condition que T_{j+1}^i ne rentre pas en conflit avec T_l^{i+1}. Si tel est le cas, la transaction T_l^{i+1} subit un seul rejoue suite au conflit avec T_j^i.
En outre, pour que la transaction T_l^{i+1}, après son rejoue, puisse à son tour satisfaire les conditions (7.1) et (7.2), il faut que la condition suivante soit satisfaite

$$R_l^{i+1} + R_j^i + a_j^i \leq C_{i+1} \tag{7.3}$$

En généralisant (7.3) pour m transactions, le temps d'exécution de la transaction T_l^{i+1} devient comme suit :

$$\sum_{l=1}^{m_{i+1}} (R_l^{i+1} + a_l^{i+1}) + \sum_{j=1}^{m_i} (R_j^i + a_j^i) \leq C_{i+1} \tag{7.4}$$

Nous avons supposé que les tâches sont ordonnées dans l'ordre inverse de leur priorité. En sommant sur l'ensemble des tâches plus prioritaires que la tâche τ_{i+1}, nous obtenons alors le temps d'exécution de la transaction T_l^{i+1} au pire cas. □

Contrairement aux verrous, la RT-STM n'est pas pessimiste. Par conséquent, le taux de lecture L_j^i qui est commun à toutes les transactions, peut faire réduire le temps Ψ_j^i comme suit

$$\Psi_j^i := \sum_{j=1}^{m_i} (R_j^i + a_j^i) + ((\sum_{i-1} \sum_{j=1}^{m_{i-1}} (R_j^i + a_j^i)).(1 - L_j^i)) \tag{7.5}$$

En considérant l'équivalence entre (7.1) et (7.2), nous posons le corollaire suivant.

Corollaire 7.1 *Un ensemble transactionnel Ω_i est dit m-firm si :*

$$\exists \Gamma_j^i \subset \Omega_i : |\Gamma_j^i| = m_i \wedge \forall T_j^i \in \Gamma_j^i : \Psi_j^i \leq D_j \tag{7.6}$$

L'intervalle de temps nécessaire pour garantir le corollaire 7.6 va dépendre de la tâche ayant le pire cas d'exécution transactionnel. Cette tâche est la moins prioritaire dans le système.
Nous symbolisons par Φ la borne de cet intervalle de temps définie comme suit :

$$\Phi = \max \bigcup_{\tau_i \in \tau} \{\Psi_j^i\} \tag{7.7}$$

Enfin, comme nous avons supposé qu'au pire cas les transactions démarrent toutes à partir de l'instant t, l'intervalle de temps pour garantir le corollaire (7.6) est alors $[t, t + \Phi)$.

Contraintes associées à des tâches m-firm

Il reste maintenant à déterminer les paramètres des tâches pour que celles-ci puissent satisfaire le corollaire 7.6. Rappelons-le, les identifiants des tâches sont ordonnés dans l'ordre inverse de leur priorité.

Corollaire 7.2 *Un ensemble de tâches τ est dit m-firm si :*

$$\forall \tau_i \in \tau : C_i \leq C_{i+1} + \Psi_j^i$$

Preuve. La condition (7.4) montre que chaque tâche τ_{i+1} doit avoir une durée d'exécution additionnelle par rapport à la tâche τ_i. Ce temps additionnel permet à la tâche τ_{i+1} d'exécuter ses m_{i+1} transactions au pire cas. Autrement dit, la tâche τ_{i+1} doit avoir au minimum un temps d'exécution $C_{i+1} \geq \Psi_j^i$. Mais comme la tâche τ_i doit aussi à son tour satisfaire la condition (7.3), nous devons donc nécessairement avoir l'inégalité du corollaire (7.2). □

7.4 Extension et évaluation de la RT-STM

Dans les paragraphes suivants, nous présentons l'implémentation du modèle m-firm et de la QoS au sein de la RT-STM, ainsi que les résultats d'expérimentations associés à leur évaluation.

7.4.1 Implémentation du modèle *m-firm*

Nous rappelons que le paramètre m_i est spécifié au niveau de la tâche et représente les transactions *obligatoires*. La valeur k_i est déterminée après les 10 secondes de test sur la RT-STM. Les $k_i - m_i$ transactions sont considérées comme *optionnelles*.

Afin de différencier les notations, nous désignons par $k_i(t)$ le nombre de transactions à l'instant t qui ont déjà effectué leur *commit* avec succès. L'Algorithme 13 effectue la prise en charge du paramètre m_i. Cet algorithme est exécuté par toutes les tâches avant chaque lancement d'une transaction.
Si le travail de la tâche τ_i a exécuté toutes ses transactions obligatoires (i.e,

Algorithme 13 : Prise en charge du paramètre m_i

Require: m_i
1: Init Entier : index $\Leftarrow \max \bigcup_{\tau_i \in \tau} P_i$
2: $d_j^i \Leftarrow d_i$
3: **if** $(k_i(t) > m_i)$ **then**
4: $\quad d_j^i \Leftarrow (d_i + index + 1)$
5: **end if**
6: $D_j^i \Leftarrow (s_j^i + d_j^i)$
7: Lancer $T_j^i(s_j^i, D_j^i, a_j^i, \sigma_j^i)$

$k_i(t) - m_i > 0$) (ligne 3), alors les transactions restantes sont considérées optionnelles (ligne 4), et ce, jusqu'à la terminaison du travail. La priorité des transactions optionnelles sont indexées (lignes 1 et 4) par rapport à la plus grande période du système de tâches. De cette manière, aucune transaction optionnelle ne peut progresser au détriment d'une autre transaction obligatoire. Par contre, toutes les transactions optionnelles sont concurrentes entre elles, et s'exécutent dans le respect des priorités de leurs tâches.

7.4.2 Évaluation empirique du modèle m-firm

Dans ce paragraphe, nous évaluons notre RT-STM pour des contraintes m-firm. Pour ce faire, nous utilisons la plateforme de test présentée au chapitre précédent. Nous allons d'abord décrire les *benchmarks* utilisés. Puis, nous observons le nombre maximum de transactions *obligatoires* que peut garantir notre RT-STM. Le MW est utilisé durant toutes les expérimentations. Le choix du MW est justifié au chapitre précédent, car il permet une meilleure répartition de *rollbacks* par rapport à celle obtenue avec le 1W.

Paramètres transactionnels

Nous avons vu au chapitre précédent que les meilleures performances du protocole MW en termes de répartition de bande passante, sont obtenues en fixant les paramètres transactionnels comme suit : $a_j^i = 75\mu$s, $\sigma_j^i = 0\mu$s. Nous fixons également à la valeur usuelle le taux de lecture $L_j^i = 75\%$.

Paramètres des tâches

Nous rappelons que dans LITMUS^{RT}, les tâches sporadiques sont générées tel que $P_i = D_i$. Le paramètre C_i doit être quant à lui fixé de manière à satisfaire l'inégalité 7.2. Pour ce faire, nous pouvons réécrire l'inégalité

(7.2) de la manière suivante :

$$2 \times C_i \leq \max\{C_{i+1}, \Psi_i\}$$

Mais pour satisfaire m_i transactions nous devons avoir $\Psi_i \leq C_i$. En outre, nous devons aussi avoir $C_i \leq C_{i+1}$. Ainsi, pour garantir d'une manière minimale l'inégalité 7.2, nous proposons de générer les C_i comme suit :

$$C_{i+1} = 2 \times C_i \tag{7.8}$$

Dans la plateforme de test, le *benchmark* de la RT-STM est un tableau d'objets identiques. Comme $\sigma_j^i = 0\mu$s, la durée d'exécution effective d'une transaction sans rejoue est de l'ordre de la microseconde. Pour cela, nous considérons que la durée d'exécution $C_i = 20$ms de la première tâche (cf. Tableau 7.1) est largement suffisante pour contenir un nombre total de transactions $k_j > 1$.
Par ailleurs, comme nous l'avons vu au chapitre 6 (cf. Figure 6.9-b), le taux d'utilisation du processeur n'a pas une influence significative sur la répartition de la bande passante du MW, nous avons donc arbitrairement choisi de fixer le taux d'utilisation du processeur $U_p = 50\%$ (cf. Tableau 7.1).

Tâches	C_i(ms)	P_i(ms)
τ_1	20	40
τ_2	40	80
τ_3	60	120
τ_4	80	160
τ_5	100	200
τ_6	120	240
τ_7	140	280

TABLE 7.1 – *Paramètres des tâches pour le m-firm.*

Le nombre de transactions m_i est un paramètre d'entrée. Il varie entre 10 et 4000.

Résultats expérimentaux

Les transactions obligatoires. La Figure 7.1 montre le ratio de transactions obligatoires accomplies dans le respect de leur échéance pour chaque tâche en fonction de m_i (cf. Tableau 7.2). Nous observons que ce taux décroît en fonction de m_i. Puisque le taux de lecture $L_j^i = 75\%$ est élevé, le point de décroissance dépend alors de la durée d'exécution C_i. La tâche τ_1 ne peut pas garantir plus de $k_1 = 518$ transactions obligatoires. Autrement dit, la tâche τ_1 peut avoir un ensemble *m-firm* dont la cardinalité est bornée par $k_1 = 518$. En outre, le système de tâches est *m-firm* si $m_i \leq 518$ puisque toutes les tâches arrivent à satisfaire ce nombre de transactions.
Par ailleurs, même si les valeurs de k_i sont dépassées, les courbes ne passent pas immédiatement à zéro comme l'exige le contexte temps réel *firm*. Mais nous avons fait le choix de laisser à l'application *firm* qui va utiliser la RT-STM, d'ignorer/accepter ces transactions qui n'ont pas respecté leur échéance.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
k_i	518	1038	1557	2077	2594	3114	3630

TABLE 7.2 – *Nombre total de transactions par travail*

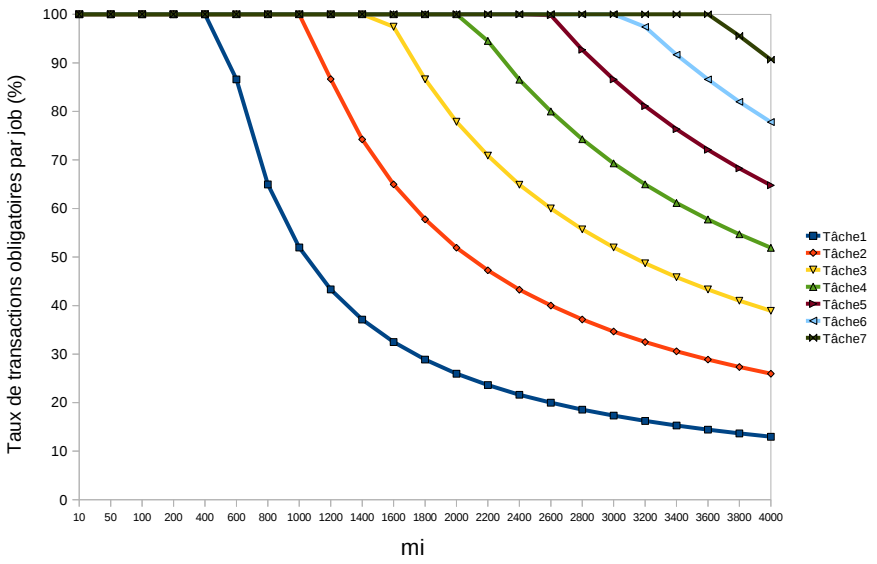


FIGURE 7.1 – *Taux de transactions m_i par travail*

Les transactions optionnelles. La Figure 7.2 montre l'évolution du ratio des transactions optionnelles qui respectent leur échéance sur le nombre total de transactions. Quand m_i est petit, les transactions optionnelles sont majoritaires, ce qui explique dans ce cas leur fort taux de présence. Le nombre de transactions optionnelles se réduit à zéro quand m_i converge vers k_i. Nous pouvons observer également que les transactions optionnelles sont bien ordonnancées par la RT-STM selon la priorité des tâches.

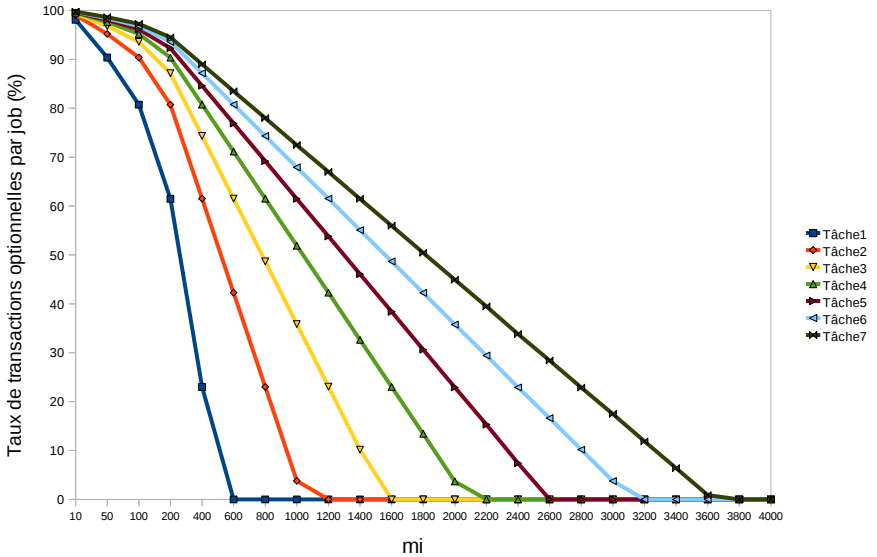


FIGURE 7.2 – *Taux de transactions optionnelles par travail*

7.4.3 Implémentation de la QoS

Lorsque l'utilisateur spécifie une QoS de ratio q_i pour une tâche τ_i, celle-ci doit dynamiquement progresser en fonction d'une part du nombre de transactions accomplies, et d'autre part, du nombre de transactions qu'il lui reste encore à lancer afin d'atteindre q_i. Plus formellement, nous définissions ce ratio dynamique comme suit :

$$RQ_i = \lceil \frac{\lambda \times k_i(t)}{k_i(t) + \sum_{j=1}^{k_i(t)} n_j} \rceil \tag{7.9}$$

Où λ représente un facteur de normalisation, $k_i(t)$ le nombre total de transactions déjà exécutées avec succès à l'instant t, et n_j le nombre de rejoues de la transaction T_j. Le nombre de rejoues est sommé sur la totalité des transactions accomplies à l'instant t.

L'Algorithme 14 garantit à la tâche τ_i une progression à un taux q_i. Avant chaque lancement de la transaction T_j^i (ligne 8), on compare le ratio RQ_i à q_i (ligne 2). La différence entre RQ_i et q_i représente la *distance* restante à τ_i pour satisfaire la QoS q_i. Plus cette *distance* est importante, plus le ratio RQ_i est petit, et donc plus la transaction est prioritaire (ligne 3). Ainsi, le ratio RQ_i peut, après normalisation, être directement utilisé comme échéance pour la transaction T_j^i (ligne 3) (rappelons qu'au sein de la RT-STM, plus une transaction a une échéance proche, plus elle est prioritaire, et plus elle progresse vis-à-vis des autres transactions).
Une fois q_i satisfaite (ligne 4), la priorité de la transaction T_j^i est réduite afin de ne pas empêcher les autres transactions de progresser. Pour cela, la priorité de T_j^i est indexée par rapport à un paramètre de normalisation λ' et par rapport à la période P_i de la tâche à laquelle elle appartient (ligne 5).

Algorithme 14 : Prise en charge de la QoS

Require: q_i
1: Init Paramètre de normalisation : λ'
2: **if** $(RQ_i \leq q_i)$ **then**
3: $\quad d_j^i \Leftarrow RQ_i$
4: **else**
5: $\quad d_j^i \Leftarrow (\tau_i.P_i + \lambda')$
6: **end if**
7: $D_j^i \Leftarrow (s_j^i + d_j^i)$
8: Lancer $T_j^i(s_j^i D_j^i, a_j^i, \sigma_j)$

7.4.4 Evaluation empirique des garanties de QoS

Nous avons effectué deux tests sur notre plateforme MW-RT-STM (cf. Tableau 7.3). Aussi, nous reprenons les paramètres utilisés lors de l'évaluation du modèle m-firm (cf. paragraphe 7.4.2) aussi bien pour les transactions que les tâches.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
Test1 : q_i	100%	100%	100%	100%	100%	100%	100%
Test2 : q_i	14%	57%	28%	61%	6%	77%	32%

TABLE 7.3 – *Paramètres de tests de la QoS*

Le premier test

Dans ce test, nous avons attribué à toutes les tâches le même $q_i = 100\%$, ce qui revient à apposer des contraintes temps réel *hard*. La Figure 7.3 montre les résultats de ce test. Nous remarquons que la QoS obtenue sur les diagrammes, n'est pas celle attendue (i.e, 100%) mais plutôt une valeur moyenne de 82, 9%. Cette valeur n'est rien d'autre que le taux moyen de progression déjà obtenu au chapitre 6 (cf. Figure 6.6).

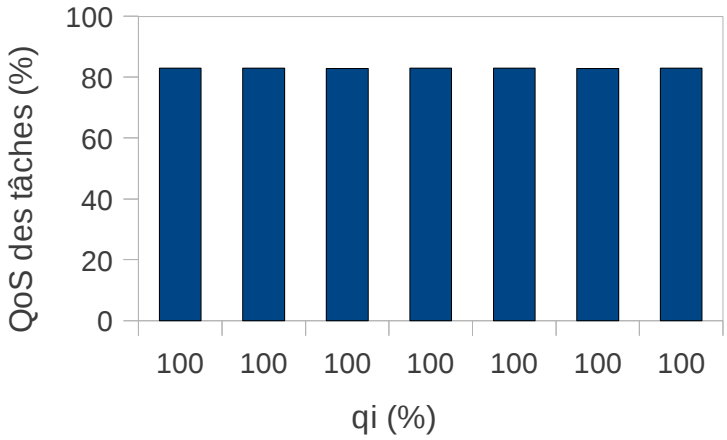


FIGURE 7.3 – *La QoS atteint la progression maximale du MW*

Le deuxième test

Nous avons généré aléatoirement les valeurs des q_i afin d'éviter la coïncidence entre les contraintes temporelles de la tâche et celles relatives aux garanties d'accès aux ressources partagées.
La Figure 7.4 montre que tous les paramètres des q_i sont satisfaits à plus de 99% d'efficacité.

CONCLUSION

Les systèmes temps réel requièrent de plus en plus de performances à la fois en termes de bande passante et de satisfaction de leurs contraintes de temps. Dans le chapitre précédent, nous avons proposé des protocoles de concurrence pouvant répondre à de tels besoins. Avec ces protocoles, l'importante bande passante qu'offre les mémoires transactionnelles par rapport aux verrous est réparties entre les tâches en fonction de leurs priorités.

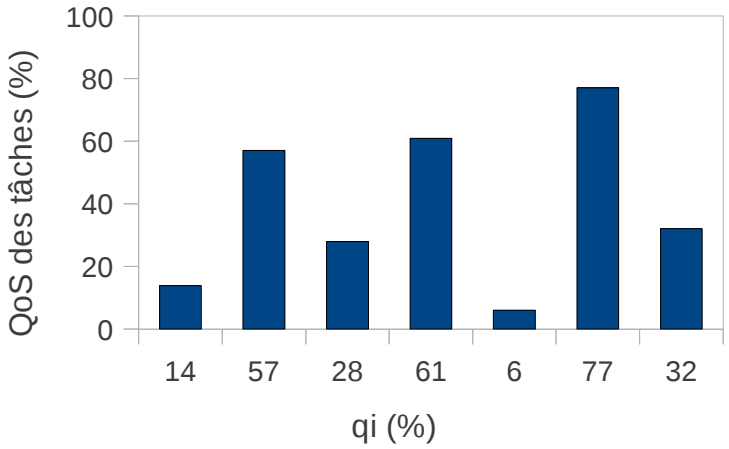


FIGURE 7.4 – *Garantie de la QoS pour toutes les tâches*

Dans ce dernier chapitre, nous avons montré comment spécifier et contrôler d'une manière plus précise cette bande passante.
Pour cela, nous avons établi un nouveau modèle que nous avons nommé *m-firm*, et qui permet entre autres de prendre en compte le nombre de transactions obligatoires qui doivent respecter leurs échéances par travail. L'analyse du pire cas d'exécution des transactions a montré qu'une telle contrainte peut être prise en compte par la RT-STM sous certaines conditions apposées aux tâches applicatives. Puis, nous avons validé cette analyse expérimentalement sur une plateforme réelle.
Enfin, au vu de ces derniers résultats, nous avons introduit la QoS au sein de la RT-STM. Nous avons pour cela adapté le modèle m-firm pour qu'il tienne compte du paramètre de QoS associée à la progression de chaque tâche. Puis, nous avons proposé un algorithme qui permet à chaque tâche de progresser en fonction du niveau de QoS prédéfini par l'utilisateur. Les résultats expérimentaux, montrent que la RT-STM peut être adaptée pour supporter la progression selon un niveau de QoS, avec toutefois une borne supérieure au-delà de laquelle les accès ne peuvent plus être garantis.

Conclusion générale

Ce rapport de thèse a montré dans quelle mesure il était possible d'adapter les mémoires transactionnelles aux systèmes temps réel multicœurs. L'objectif visé par ce travail était de ramener les avantages que procure le concept de mémoire transactionnelle vers les systèmes temps réel.

Dans la première partie de ce rapport, nous avons d'abord mené un état de l'art sur les systèmes temps réel multiprocesseurs. Cette partie nous a permis de mettre l'accent sur les différentes approches et politiques d'ordonnancement existantes mais aussi et surtout sur les différents types de synchronisation. Parmi ceux-ci, le lock-free a été mis en avant pour montrer l'intérêt d'adopter un mécanisme non-bloquant pour la gestion de concurrence entre des tâches temps réel. En outre, cet état de l'art nous a permis par la suite de sélectionner un système d'exploitation temps réel multiprocesseur.

Puis, nous nous sommes intéressés aux systèmes de gestion des bases de données temps réel. Ceci nous a permis de lister les différents protocoles et les techniques qu'ils utilisent. En classifiant ces protocoles, il a alors été possible d'écarter d'emblée ou bien de retenir certains protocoles en vue d'une intégration au sein d'une mémoire transactionnelle. En effet, certains protocoles se sont avérés soit coûteux en ressources mémoire, soit offrant de faibles performances globales, ce qui représentait un frein à leur implémentation dans le cas des transactions résidant en mémoire. Les protocoles 2PL-HP et RT-DATI présentaient des particularités intéressantes. La stratégie de résolution de conflit du 2PL-HP a été reprise par la suite dans l'un des protocoles proposés dans la thèse. Le RT-DATI quant à lui traitait d'une notion importante qui est l'ordre de la sérialisation, en intégrant la priorité des transactions. L'idée de ce mécanisme nous a aussi aidés par la suite à diagnostiquer le manque de performances dont souffrait notre première solution.

Enfin, la présentation des mémoires transactionnelles, nous a permis de constater que l'étude de contrôles de concurrence spécialement dédiés aux mémoires transactionnelles temps réel n'est pas traitée dans la littérature. Aussi, l'étude des différentes mémoires transactionnelles nous a permis de sélectionner par la suite la STM qui était la plus susceptible d'être adaptée aux systèmes temps réel *soft*.
L'étude de l'état de l'art a conduit à la publication suivante : Sarni et al. (2009a).

Dans la seconde partie de ce rapport, nous avons proposé une STM temps réel pour le temps réel *soft*. Dans un premier temps, nous nous sommes intéressés à l'adéquation des mémoires transactionnelles aux systèmes temps réel multiprocesseurs, en évaluant la variabilité du temps

d'exécution des transactions lors de l'accès aux ressources partagées. En utilisant les systèmes d'exploitation Linux et LITMUS^{RT}, nous avons effectué une étude comparative entre diverses mémoires transactionnelles (OSTM, DTSM et STM d'Ennal). Cette comparaison sur des environnements classique et temps réel, nous a conduit (Sarni et al. 2009b) à sélectionner la OSTM comme étant la plus performante d'entre elles. En outre, nous avons conclu que l'algorithme P-EDF est la politique l'ordonnancement temps réel qui présente expérimentalement les meilleurs performances parmi G-EDF et PD^2. Enfin, nous avons montré (Sarni et al. 2009c) qu'au sein de la OSTM, l'allocation mémoire requiert plus de considération pour approximer la variabilité du temps d'exécution des transactions, et qu'il est préférable d'utiliser des allocateurs à temps d'allocation constant, tel que TLSF.

Par la suite, nous avons proposé un modèle transactionnel temps réel pour les STMs temps réel pouvant caractériser aussi bien des transactions à contraintes *hard* que *soft*. En se basant sur ce modèle, nous avons démontré la faisabilité d'une mémoire STM temps réel (RT-STM). Nous avons pour cela muni les transactions d'échéances, imposant par là-même que leur exécution s'effectue dans le respect de contraintes de temps. En particulier, nous avons fait évoluer les heuristiques du contrôleur de concurrence de la OSTM, pour qu'elles se basent sur des contraintes temps réel lors de la résolution des conflits. Les résultats expérimentaux ont montré (Sarni et al. 2009b) qu'une telle démarche est possible, et nous avons pu obtenir un gain de performance par rapport à la OSTM en termes de nombre de transactions respectant leurs échéances.

Dans un deuxième temps, nous avons conçu et réalisé deux nouveaux protocoles de synchronisation pour une mémoire transactionnelle logicielle adaptée au temps réel *soft*. Ces protocoles se basent pour l'un sur un seul écrivain (1W), et pour l'autre, sur l'exécution parallèle des écrivains (MW). Le 1W ressemble au 2PL-HP mais avec intégration du mécanisme du *helping*. Le MW quant à lui, est un protocole lock-free avec prise en compte des contraintes de temps. Les deux protocoles sont à grain fin et utilisent l'instruction de base CAS ainsi que des variantes que nous avons construites à savoir CASG et DCASG. Sur une plateforme réelle dotée de 8 cœurs, nous avons comparé le 1W et le MW à des protocoles de référence de la littérature, aussi bien pour le domaine classique (protocole non bloquant de la OSTM) que temps réel (protocole bloquant FMLP).
Les résultats ont montré (Queudet et al. 2012) que sous une contention usuelle (75% de lecture) la bande passante de nos deux protocoles 1W et MW se situe entre celles de la OSTM et du FMLP. De plus, contrairement à la OSTM, nos deux protocoles 1W et MW répartissent les temps de *rollbacks* des transactions (et donc la bande passante) entre les tâches selon leur priorité.

Dans la troisième partie de ce rapport, nous avons établi un nouveau modèle que nous avons nommé *m-firm*, et qui permet entre autres de prendre en compte le nombre de transactions obligatoires qui doivent respecter leurs échéances par travail. L'analyse du pire cas d'exécution a montré qu'une telle contrainte peut être prise en compte par la RT-STM

sous certaines conditions. Nous avons également proposé une adaptation de la RT-STM afin de garantir aux tâches un certain niveau de qualité de service (QoS) vis-à-vis des accès aux ressources partagées.

Perspectives

Plusieurs perspectives peuvent être formulées pour les travaux de recherche futurs. Nous les classons ici par catégorie, et nous allons nous restreindre aux plus pertinentes d'entre elles.

Sur site local

Les protocoles que nous avons conçus pour la RT-STM se basent sur des heuristiques. Bien que ces protocoles puissent garantir qu'un ensemble de transactions respectent leurs échéances, ils ne peuvent cependant pas garantir des contraintes temps réel *hard*. Ce type de contrainte exige que toutes les transactions respectent de manière stricte leurs échéances. L'une des pistes serait de concevoir un protocole *wait-free* pour la RT-STM. Les procédures de marquage des protocoles doivent être alors améliorées de manière à exiger de la procédure de synchronisation un nombre fixe d'itérations avant le marquage. Pour cela, la RT-STM doit être portée sur une autre architecture matérielle (de type Motorola par exemple, ou bien des futures architectures exclusivement transactionnelles comme celles de SUN) afin de borner plus aisément le temps de chaque procédure transactionnelle.

Sur site distribué

Les travaux de Herlihy et al. (2003a) sur l'utilisation des mémoires transactionnelles sur une DHT (*Distributed Hash Table*) sont tout aussi prometteurs pour le temps réel. Plusieurs bus de terrain assurent un temps de communication borné. Le modèle transactionnel devrait être enrichi pour prendre en compte ces contraintes de communication de bout-en-bout.

Problèmes ouverts

Dans un système classique, l'utilisation d'une mémoire transactionnelle avec entrées/sorties réduirait considérablement leur performance en termes de bande passante et reste donc un problème difficile. Le problème durcit davantage avec les systèmes temps réel, puisque le temps d'accès au périphérique doit être borné.

Bibliographie

Robert K. Abbott et Hector Garcia-Molina. Scheduling real-time transactions : a performance evaluation. Dans *VLDB*, pages 1–12, 1988. (Cité pages 8, 33, 34 et 78.)

Yehuda Afek, Michael Merritt, Gadi Taubenfeld, et Dan Touitou. Disentangling multi-object operations (extended abstract). Dans *PODC '97 : Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 111–120, New York, NY, USA, 1997. ACM. ISBN 0-89791-952-1. (Cité page 48.)

C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, et Sean Lie. Unbounded transactional memory. Dans *HPCA*, pages 316–327. IEEE Computer Society, 2005. ISBN 0-7695-2275-0. (Cité page 46.)

James H. Anderson. Early-release fair scheduling. Dans *proc. the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000. (Cité page 25.)

James H. Anderson, Rohit Jain, et Srikanth Ramamurthy. Implementing hard real-time transactions on multiprocessors. Dans *RTDB*, pages 247–260, 1997a. (Cité page 34.)

James H. Anderson et Mark Moir. Universal constructions for multi-object operations. Dans *PODC '95 : Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 184–193, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. (Cité pages 29 et 48.)

James H. Anderson et Mark Moir. Universal constructions for large objects. *IEEE Trans. Parallel Distrib. Syst.*, 10 :1317–1332, December 1999. ISSN 1045-9219. (Cité pages 28 et 29.)

James H. Anderson, Srikanth Ramamurthy, et Rohit Jain. Implementing wait-free objects on priority-based systems. Dans *PODC '97 : Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 229–238, New York, NY, USA, 1997b. ACM. ISBN 0-89791-952-1. (Cité pages 29 et 48.)

James H. Anderson, Srikanth Ramamurthy, et Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2) : 134–165, 1997c. ISSN 0734-2071. (Cité page 29.)

James H. Anderson et A. Srinivasan. Pfair scheduling : beyond periodic task systems. Dans *RTCSA 'oo : Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, page 297, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0930-4. (Cité page 25.)

Bjorn Andersson, Sanjoy Baruah, et Jan Jonsson. Static-priority scheduling on multiprocessors. Dans *RTSS '01 : Proceedings of the 22nd IEEE Real-Time Systems Symposium*, page 93, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1420-0. (Cité page 23.)

Bjorn Andersson et Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. *Euromicro Conference on Real-Time Systems*, 0 :33, 2003. (Cité page 23.)

Neil C. Audsley. Deadline monotonic scheduling, 1990. (Cité page 19.)

T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3 :67–99, April 1991. ISSN 0922-6443. (Cité page 26.)

Greg Barnes. A method for implementing lock-free shared-data structures. Dans *SPAA '93 : Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2. (Cité page 29.)

S. K. Baruah, N. K. Cohen, C. G. Plaxton, et D. A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15 : 600–625, 1996. (Cité pages 23, 24 et 25.)

Sanjoy K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Trans. Comput.*, 53(6) :781–784, 2004. ISSN 0018-9340. (Cité page 23.)

Sanjoy K. Baruah et John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *J. Embedded Comput.*, 1 (2) :169–178, 2005. ISSN 1740-4460. (Cité page 23.)

Sanjoy K. Baruah, Johannes E. Gehrke, et C. G Plaxton. Fast scheduling of periodic tasks on multiple resources. Rapport technique, Austin, TX, USA, 1995. (Cité page 25.)

G. Bernat. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Departament de Ciències Matemàtiques i Informàtica. Universitat de les Illes Balears. Spain, 1998. (Cité page 109.)

A Bestavros. Speculative concurrency control. Rapport technique, Boston University, Boston, MA, 1992. (Cité page 37.)

Aaron Block, Hennadiy Leontyev, Bjorn B. Brandenburg, et James H. Anderson. A flexible real-time locking protocol for multiprocessors. Dans *RTCSA '07 : Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2975-5. (Cité page 27.)

Colin Blundell, E Christopher Lewis, et Milo M. K. Martin. Deconstructing transactions : The subtleties of atomicity. Dans *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005. (Cité page 43.)

Robert L. Bocchino, Vikram S. Adve, et Bradford L. Chamberlain. Software transactional memory for large scale clusters. Dans *PPoPP '08 : Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. (Cité page 46.)

Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, et James H. Anderson. Real-time synchronization on multiprocessors : To block or not to block, to suspend or spin ? Dans *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society, 2008. (Cité pages 30 et 60.)

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, et James H. Anderson. Litmusˆrt : A testbed for empirically comparing real-time multiprocessor schedulers. Dans *RTSS*, pages 111–126. IEEE Computer Society, 2006. (Cité page 54.)

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, et Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. Dans *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004. (Cité pages 3, 22, 23 et 24.)

S. Chakravarthy, D. Hong, et T. Johnson. Real-time transaction scheduling : a framework for synthesizing static and dynamic factors. Rapport technique, Computer and Information Science and Engineering Department, University of Florida, 1994. (Cité page 35.)

Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, et Osvaldo Colavin. Unbounded page-based transactional memory. Dans *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358. ACM, 2006. (Cité page 46.)

Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, et Daniel Nussbaum. Hybrid transactional memory. Dans John Paul Shen et Margaret Martonosi, éditeurs, *ASPLOS*, pages 336–346. ACM, 2006. ISBN 1-59593-451-0. (Cité page 48.)

Robert I. Davis et Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43 :35 :1–35 :44, Octobre 2011. ISSN 0360-0300. (Cité page 21.)

Michael L. Dertouzos. Control Robotics : The Procedural Control of Physical Processes. Dans *Proceedings of IFIP Congress (IFIP'74)*, pages 807–813, Stockholm, Sweden, 1974. (Cité page 19.)

David Dice, Ori Shalev, et Nir Shavit. Transactional locking ii. Dans Shlomi Dolev, éditeur, *DISC*, volume 4167 de *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006. ISBN 3-540-44624-9. (Cité page 58.)

R. Ennal. Softawre transactional memory should not be obstruction-free. Rapport technique, Intel Research Cambridge, 2006. (Cité pages 47, 54 et 81.)

K. Eswaran, J. Gray, R. Lorie, et I. Traiger. The notions of consistency and predicate locks in a database system. *Communication of the ACM*, 19(11) : 624–630, 1976. (Cité page 33.)

Sherif F. Fahmy, Binoy Ravindran, et E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. Dans *SAC '09 : Proceedings of the 2009 ACM symposium on Applied Computing*, pages 334–338, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. (Cité page 49.)

Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, et Heiko Sturzrehm. Transactifying applications using an open compiler framework. Dans *TRANSACT*, August 2007. (Cité page 46.)

Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579. (Cité pages 47, 56, 67, 71 et 98.)

Keir Fraser et Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007. (Cité pages 45, 47, 54 et 79.)

Shelby Funk, Joël Goossens, et Sanjoy K. Baruah. On-line scheduling on uniform multiprocessors. Dans *IEEE Real-Time Systems Symposium*, pages 183–192, 2001. (Cité page 17.)

Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, et Paolo Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. Dans *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 189–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1956-3. (Cité page 27.)

G. Gardarin. *Bases de données objet & relationnel*. Eyrolles ed, 2000. (Cité page 33.)

Joël Goossens, Shelby Funk, et Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3) : 187–205, 2003. ISSN 0922-6443. (Cité page 23.)

Rachid Guerraoui, Maurice Herlihy, et Bastian Pochon. Polymorphic contention management. Dans *DISC '05 : Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005. (Cité page 44.)

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, et Kunle Olukotun. Transactional memory coherence and consistency. Dans *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004. (Cité page 46.)

Jayant R. Haritsa, Michael J. Carey, et Miron Livny. On being optimistic about real-time constraints. Dans *PODS*, pages 331–343. ACM Press, 1990. (Cité page 37.)

Jayant R. Haritsa, Michael J. Carey, et Miron Livny. Data access scheduling in firm real-time database systems. *Real-Time Systems*, 4(3) :203–241, 1992. (Cité pages 36 et 37.)

Tim Harris et Keir Fraser. Language support for lightweight transactions. Dans *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003. (Cité page 46.)

J. Haubert, B. Sadeg, et L. Amanton. $\binom{m}{k}$-firm real-time distributed transactions. Dans *Proc. of the 16th WIP Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–65, 2004. (Cité pages 110 et 111.)

Maurice Herlihy, Victor Luchangco, et Mark Moir. Obstruction-free synchronization : Double-ended queues as an example. Dans *ICDCS '03 : Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003a. IEEE Computer Society. ISBN 0-7695-1920-2. (Cité pages 30 et 123.)

Maurice Herlihy, Victor Luchangco, Mark Moir, et William N. Scherer III. Software transactional memory for dynamic-sized data structures. Dans *PODC*, pages 92–101, 2003b. (Cité pages 47 et 54.)

Maurice Herlihy et J. Eliot B. Moss. Transactional memory : Architectural support for lock-free data structures. Dans *proc. the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993. (Cité pages 8, 42 et 46.)

Maurice Herlihy et Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3) :195–208, 2007. (Cité page 46.)

M. Teresa Higuera-Toledano, Valérie Issarny, Michel Banâtre, et Frédéric Parain. Memory management for real-time java : An efficient solution using hardware support. *Real-Time Systems*, 26(1) :63–87, 2004. (Cité page 48.)

Philip Holman et James H. Anderson. Guaranteeing pfair supertasks by reweighting. Dans *proc. the 22nd IEEE Real-time Systems Symposium*, pages 203–212, 2001. (Cité pages 27 et 28.)

Philip Holman et James H. Anderson. Supporting lock-free synchronization in pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66 (1) :47–67, 2006. ISSN 0743-7315. (Cité pages 29 et 49.)

D. Hong, T. Johnson, et S. Chakravarthy. Real-time transaction scheduling : a cost conscious approach. *SIGMOD Rec.*, 22(2) :197–206, 1993. ISSN 0163-5808. (Cité page 35.)

J. Huang et J. Stankovic. Concurrency control in real-time database system : Optimistic scheme vs. two-phase locking. Rapport technique, UM-CS-1990-066, University of Massachusetts, 1990. (Cité pages 35 et 73.)

IBM. System/370 principles of operation. 1970. (Cité page 28.)

D. Johnson. Fast algorithms for bin packing. *Journal of Computer ans Systems Science*, 8(3) :272–314, 1974. (Cité page 20.)

Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, et Anthony Nguyen. Hybrid transactional memory. Dans Josep Torrellas et Siddhartha Chatterjee, éditeurs, *PPOPP*, pages 209–220. ACM, 2006. ISBN 1-59593-189-9. (Cité page 48.)

H. Kung et J. Robinson. On optimistic methods for concurrency control. *Communication of the ACM*, 6(2), 1981. (Cité page 35.)

Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11) : 806–811, 1977. ISSN 0001-0782. (Cité page 29.)

James R. Larus et Ravi Rajwar. *Transactional Memory*. Morgan & Claypool publishers, 2007. (Cité page 43.)

Juhnyoung Lee et Sang Hyuk Son. Using dynamic adjustment of serialization order for real-time database systems. Dans *14th IEEE Real-Time Systems Symposium*, pages 66–75, 1993. (Cité page 36.)

Jan Lindström et Kimmo Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. *Real-Time Computing Systems and Applications, International Workshop on*, 0 :13, 1999. ISSN 1530-1427. (Cité page 38.)

Jan Lindström et Kimmo Raatikainen. Extensions to optimistic concurrency control with time intervals. Dans *7th International Workshop on Real-Time Computing and Applications*. IEEE Computer Society, 2000. ISBN 0-7695-0930-4. (Cité pages 39 et 78.)

C. L. Liu et James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973. ISSN 0004-5411. (Cité pages 7, 19 et 20.)

Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st édition, 2000. ISBN 0130996513. (Cité page 14.)

David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. Dans *ACM Conference on Language Design for Reliable Software*, pages 128–137. Mar 1977. (Cité page 42.)

J. M. López, J. L. Díaz, et D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1) :39–68, 2004. ISSN 0922-6443. (Cité pages 23, 26 et 27.)

Victor Bradley Lortz. *An object-oriented real-time database system for multiprocessors*. PhD thesis, Ann Arbor, MI, USA, 1994. (Cité page 34.)

Kaloian Manassiev, Madalin Mihailescu, et Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. Dans *PPoPP '06 : Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. (Cité page 46.)

Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, et Jan Vitek. Preemptible atomic regions for real-time java. Dans *RTSS '05 : Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. (Cité page 48.)

Audrey Marchand. *Ordonnancement temps réel avec contraintes de qualité de service*. PhD thesis, Université de Nantes, France, 2006. (Cité page 109.)

Miguel Masmano, Ismael Ripoll, Alfons Crespo, et Jorge Real. Tlsf : A new dynamic memory allocator for real-time systems. Dans *ECRTS*, pages 79–86, 2004. (Cité page 56.)

Mark Moir. Transparent support for wait-free transactions. Dans *WDAG '97 : Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, London, UK, 1997. Springer-Verlag. ISBN 3-540-63575-0. (Cité page 48.)

Mark Moir et Srikanth Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. Dans *RTSS '99 : Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 294, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0475-2. (Cité pages 24 et 27.)

A.K. Mok et M.L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. Dans *proc. Seventh Texas Conference on Computing System*, 1978. (Cité page 19.)

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, et David A. Wood. Logtm : Log-based transactional memory. Dans *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006. (Cité page 46.)

J. Eliot B. Moss et Antony L. Hosking. Nested transactional memory : model and architecture sketches. *Sci. Comput. Program.*, 63(2) :186–201, 2006. ISSN 0167-6423. (Cité page 42.)

D. Oh et T. P. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems : The International Journal of Time-Critical Computing*, (15) :183–192, 1998. (Cité page 18.)

T. Özsu et P. Valduriez. *Principles of Distributed Data-base Systems*. Springer, 2011. (Cité page 33.)

Audrey Queudet, Toufik Sarni, et Patrick Valduriez. Non-blocking real-time synchronization protocols for multicore systems. *Software : Practice and Experience (en cours de soumission)*, 2012. (Cité pages 8, 76 et 122.)

R. Rajkumar. Synchronization in real-time systems : A priority inheritance approach. Boston, USA, 1991. Kluwer Academic Publishers. (Cité page 26.)

Ravi Rajwar, Maurice Herlihy, et Konrad K. Lai. Virtualizing transactional memory. Dans *ISCA*, pages 494–505. IEEE Computer Society, 2005. (Cité page 46.)

Parameswaran Ramanathan. Overload management in real-time control applications using (m,k)-firm guarantee. *IEEE Trans. Parallel Distrib. Syst.*, 10(6) :549–559, 1999. ISSN 1045-9219. (Cité pages 109 et 110.)

Parameswaran Ramanathan et Moncef Hamdaoui. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Trans. Comput.*, 44(12) :1443–1451, 1995. ISSN 0018-9340. (Cité pages 109 et 110.)

Toufik Sarni, Audrey Queudet, et Patrick Valduriez. Real-time scheduling of transactions in multicore systems. *Workshop on Massively Multiprocessor and Multicore Computers*, 2009a. (Cité pages 8, 54 et 121.)

Toufik Sarni, Audrey Queudet, et Patrick Valduriez. Real-time support for software transactional memory. Dans *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 477–485, Los Alamitos, CA, USA, August 2009b. IEEE Computer Society. (Cité pages 8, 54, 67 et 122.)

Toufik Sarni, Audrey Queudet, et Patrick Valduriez. Software transactional memory : Worst case execution time analysis. Dans M. Silly-Chetto et Mikael Sjodin, éditeurs, *17th International Conference on Real-Time and Network Systems*, pages 107–114, Paris France, October 2009c. INRIA. (Cité pages 8, 54 et 122.)

William N. Scherer et Michael L. Scott. Advanced contention management for dynamic software transactional memory. Dans Marcos Kawazoe Aguilera et James Aspnes, éditeurs, *PODC*, pages 240–248. ACM, 2005. (Cité page 44.)

William N. Scherer III et Michael L. Scott. Contention management in dynamic software transactional memory. Dans *proc. the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004. (Cité page 8.)

Martin Schoeberl, Bent Thomsen, et Lone Leth Tomsen. Towards transactional memory for real-time systems. Research Report 19/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009. (Cité page 49.)

Michael L. Scott. Sequential specification of transactional memory semantics. Dans *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006. (Cité page 43.)

Lui Sha, Ragunathan Rajkumar, et John P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9) :1175–1185, 1990. (Cité pages 26 et 35.)

N. Shavit et D. Touitou. Software transactional memory. Dans *proc. the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995. (Cité pages 8, 45, 46 et 47.)

M. Silly-Chetto. Sur la problématique de l'ordonnancement dans les systèmes informatiques temps réel. Dans *Rapport d'HDR, Université de Nantes*. 1993. (Cité page 15.)

Ye-Qiong Song. Matrix-DBP for (m,k)-firm real time guarantee. Dans *Séminaire du Programme de Recherches Avancées Franco-Chinois PRA SIo1-04*, page 35, Zhejiang University, China, 2002. (Cité page 109.)

M. Spuri. Dans *Analysis of deadline scheduled real-time systems*. 1996. (Cité page 49.)

John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 20(10) :10–19, 1988. (Cité page 14.)

M. Tremblay et S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc r processor. *IEEE International Solid-State Circuits Conference*, Feb. 2008. (Cité page 42.)

John Turek, Dennis Shasha, et Sundeep Prakash. Locking without blocking : making lock based concurrent data structure algorithms nonblocking. Dans *PODS '92 : Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, New York, NY, USA, 1992. ACM. ISBN 0-89791-519-4. (Cité page 29.)

Zhi Wang, Ye qiong Song, Enrico-Maria Poggi, et Youxian Sun. Survey of Weakly-Hard Real Time Schedule Theory and Its Application. Dans *International Symposium on Distributed Computing and Applications to Business. Engineering and Science (DCABES)*, 2002. (Cité page 109.)

Richard M. Yoo et Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. Dans Friedhelm Meyer auf der Heide et Nir Shavit, éditeurs, *SPAA*, pages 169–178. ACM, 2008. (Cité page 44.)

Vers une mémoire transactionnelle temps réel

Résumé. Avec l'émergence des systèmes multicœurs, le concept de mémoire transactionnelle (TM) a été renouvelé à la fois dans le domaine de la recherche et dans le monde industriel. En effet, en supportant les propriétés ACI (Atomicité, Consistance et Isolation) des transactions, le concept de TM facilite la programmation parallèle et évite les problèmes liés aux verrous tels que les interblocages et l'inversion de priorité. De plus, contrairement aux méthodes basées sur les verrous, une TM permet à plusieurs transactions d'accéder en parallèle aux ressources, et augmente ainsi la bande passante du système. Enfin, une TM intègre un ordonnanceur de transactions qui, soit ré-exécute (*retry)* la transaction en cas de détection de conflits, soit valide (*commit)* la transaction en cas de succès.
L'objectif de cette thèse est d'étudier l'adaptation des TMs à des systèmes temps réel *soft* au sein desquels les processus doivent s'exécuter le plus souvent possible dans le respect de contraintes temporelles. Jusqu'à maintenant, l'ordonnancement de transactions temps réel au sein d'une TM n'a pas été étudié. Dans un premier temps, nous proposons une étude expérimentale comparative nous permettant de statuer sur l'adéquation des TMs aux systèmes temps réel multicœurs. Il s'agit en particulier d'évaluer si la variabilité du temps d'exécution des transactions est prohibitif à une utilisation dans un contexte temps réel lors de l'accès aux ressources partagées. Dans un second temps, nous introduisons un modèle transactionnel temps réel pour les TMs et nous décrivons la conception et l'implémentation d'une mémoire transactionnelle logicielle temps réel nommée RT-STM. Celle-ci intègre de nouveaux protocoles de synchronisation qui permettent de prioriser les accès aux ressources partagées en fonction de l'urgence des processus. Enfin, nous montrons comment adapter notre RT-STM à un environnement temps réel *firm* en proposant quelques pistes d'adaptation permettant de garantir aux processus un certain niveau de qualité de service (QoS) vis-à-vis des accès aux ressources partagées.

Mot clés. Ordonnancement temps réel, multicœur, mémoires transactionnelles, contrôleurs de concurrence, transactions temps réel, synchronisation non-bloquante.

Towards a real-time transactional memory

Abstract. With the advent of multicore systems, the transactional memory (TM) concept has attracted much interest from both academy and industry. Indeed, by supporting the ACI (Atomicity, Consistency and Isolation) proprieties, the TM concept eases parallel programming and avoids the severe problems of lock-based methods such as deadlock situations and priority inversion. In addition, unlike lock-based methods, TM allows several transactions to access resources in parallel, and thus increases the system's bandwidth. Moreover, TM embeds a transaction's scheduler which either rollbacks the transaction when the conflict is detected, or commits the transaction on success.
The thesis's objective is to study the TM's adaptation to *soft* real-time systems in which processes must complete within deadlines as far as it is possible. Up to now, the scheduling of real-time transactions within TM has not been studied. To address this issue, we first make an experimental and comparative study to show whether the TM is suitable for real-time multicore systems. In particular, we evaluate the transaction's execution time variation when accessing shared resources. Second, we propose a novel solution which introduces a real-time transactional model within TM and we design and implement a real-time software transactional memory named RT-STM. This one integrates new synchronization protocols which allows to prioritize the shared resources' accesses according to the processes' urgency. Finally, we show how to make RT-STM suitable for *firm* real-time systems, proposing some adaptations allowing to guarantee a certain level of quality of service (QoS) to processes sharing resources.

Keywords. Real-time scheduling, multicore, transactional memory, concurrency control, real-time transactions, synchronization without locks.