LightTM : un Système de Mémoire Transactionnelle pour les MPSoCs

Quentin Meunier, Frédéric Pétrot

Laboratoire TIMA
Groupe SLS
46, Av Félix Viallet, 38031 Grenoble, France

9 Septembre 2009

Contexte

Evolution Matérielle vers les Architectures Multi-coeurs

- Plusieurs processeurs indépendants sur une puce
- Communication à travers de la mémoire partagée
- Puces à deux coeurs communes ($core^2 duo$), puces à quatre coeurs de plus en plus répandues
- Tendance future : massivement multi-processeur intégré
- Nécessité de pouvoir développer des programmes parallèles facilement

Programmation Parallèle

- Problématique, difficulté : exploiter le parallélisme disponible
- Peu de programmes exploitant ce parallélisme
- Souvent, restriction à de larges charges de travail parallèles

Contexte

Besoin d'Abstractions

- Solutions Actuelles
  - Plus utilisée : Threads
- Synchronisation des Exécutions :
  - Verrous
  - Sémaphores
  - Moniteurs

Limites

- L'expérience montre que ces programmes sont difficiles à :
  - Concevoir
  - Implementer
  - Déboguer
  - Maintenir
- ...et souvent n'ont pas de bonnes performances ou ne passent pas bien à l'échelle
- $\rightarrow$ Besoin d'autres constructions de programmation

Outline

1. Introduction
2. Généralités sur les Mémoires Transactionnelles
3. Implémentation Spécifique pour les MPSoCs : LightTM
4. Expérimentations et Résultats
5. Conclusion et Perspectives

Outline

1 Introduction

2 Généralités sur les Mémoires Transactionnelles

3 Implémentation Spécifique pour les MPSoCs : LightTM

4 Expérimentations et Résultats

5 Conclusion et Perspectives

Mémoire Transactionnelle (TM)

Notion de Transaction

- Plusieurs requêtes qui s'exécutent de manière atomique (en apparence)
- Terme emprunté aux bases de données (propriétés ACID)
- Notion restreinte de transaction pour les mémoires transactionnelles (Atomicité et Isolation)

Buts du Modèle de Programmation TM

- Fournir une construction qui permet d'exécuter une portion de code comme une transaction, i.e. de manière atomique du point de vue des autres processeurs
- Fournir un mécanisme qui permet de ne pas se soucier des interactions avec les autres parties du programme, sans limiter le parallélisme
- Être une base pour la construction des programmes parallèles

Avantages et Limitations des TMs

Avantages

- Évite les deadlocks et live-locks du point de vue du programmeur
- Meilleur passage à l'échelle qu'avec des locks
- Déplace le problème de synchronisation entre threads au système TM

Limitations

- Les transactions ne peuvent pas remplacer tous les mécanismes de synchronisation de manière efficace (ex : rendez-vous)
- Besoins élevés. Ex : Besoin de suivre les ensembles d'adresses lues et écrites durant les transactions
- Problèmes relatifs au matériel et au logiciel (vus dans la suite)

Types de Mémoires Transactionnelles

Mémoires Transactionnelles Logicielles (STM)

- Pas de matériel spécifique requis (autre que le CAS)
- Surcoûts importants
- API complexe, souvent plus difficile à utiliser que les locks conventionnels (en particulier si l'on veut une exécution efficace)
- STM objet de critiques importantes [CBM+08]

Mémoires Transactionnelles Matérielles (HTM)

- Support matériel spécifique aux transactions
- Meilleures performances et meilleur passage à l'échelle que les STM

Systèmes HTM

Caractéristiques générales & Problèmes relatifs aux systèmes HTM

- Nécessité de stocker à la fois les anciennes et les nouvelles valeurs (aborts)
  - Beaucoup de systèmes HTM utilisent les caches pour les données spéculées
  - $\Rightarrow$ Problèmes de débordement (capacité finie)
  - Dépendant du protocole de cohérence de cache
- Requièrent un support architectural important
- Coûteux en terme de logique
- Support nécessaire de la part de l'OS [TCUV07, SVG+08]
- Interactions fortes entre le logiciel (langage, compilateur), l'OS et le matériel
- De nombreuses pathologies peuvent apparaître, conséquences sur les performances (ou pire, sur les fonctionnalités)
- Problème de la gestion des commutations de threads

Classification des Systèmes HTM

Trois Principaux Critères

- Détection des conflits (CD) : Quand détecter que deux transactions sont en conflit ?
  - Avide ou Paresseuse
- Gestion de versions (VM) : Où mettre les anciennes et les nouvelles valeurs ?
  - Avide ou Paresseuse
- Résolution des Conflits : Que faire lorsqu'un conflit est détecté ?
  - Geler le processeur/cache ayant fait la requête (CD avide)
  - Aborter celui ayant fait la requête ou les autres en conflits (CD avide)
  - Aborter le processeur/cache qui committe/les autres (CD paresseuse)
  - $\longrightarrow$ peut ne pas suffire

Choix de Conception Répandus

- CD paresseuse, VM paresseuse, le processeur qui commite gagne (LL)
- CD avide, VM paresseuse, celui qui fait la requête gagne (EL)
- CD avide, VM avide, celui qui a fait la requête gêle ou aborte et recommence (EE)

Problèmes Ouverts & Conclusion

Problèmes Ouverts

- Actions ayant un effet de bord : opérations I/O, appels système (allocation), etc.
- Objets sujet à des accès transactionnels et non transactionnels
- Interaction avec des librairies existantes

En Résumé

- Systèmes TM seuls insuffisants, modifications requises dans
  - Les languages de programmation
  - Les systèmes de runtime
  - Les outils (ex : chaine de compilation)
  - ...
- Evolution Future Possible : HTM avec support STM quand les limitations matérielles sont dépassées

Outline

1 Introduction

2 Généralités sur les Mémoires Transactionnelles

3 Implémentation Spécifique pour les MPSoCs : LightTM

4 Expérimentations et Résultats

5 Conclusion et Perspectives

Choix de Conception & Restrictions

Choix de Conception Architecturaux

- Système Multiprocesseur à mémoire partagée à base de processeurs Sparc V8
- MPSoC $\Rightarrow$ Réseau-sur-Puce (NoC) : protocole de cohérence de cache basé sur un répertoire
- Caches L1 seulement, à correspondance directe
- Adresses physiques (pas de MMU)
- Basé sur un protocole de cohérence write-through

Choix de Conception Relatifs au Système TM

- CD avide, VM Paresseuse, résolution basée sur le gel du processeur ayant fait la requête
- Sémantique d'entrelacement à plat, granularité : ligne de cache

Restrictions

- Un thread par processeur, chaque thread fixé à un processeur
- Pas d'opérations I/O ni d'appels systèmes au sein d'une transaction

Modifications Architecturales et de l'OS

Modifications de l'interface de l'OS

- Ajout de trois "fonctions"
  - `begin_transaction()` et `end_transaction()` : implementées directement en utilisant les registres libres du sparc (asr17)
  - `store_log_address(address)` : permet la configuration d'un registre spécifique contenant l'adresse de départ du log
- Exemple d'utilisation :

```
begin_transaction() ; // Garantit que x est
   x++ ;              // incrementé de 1, même si
end_transaction() ;   // c'est une variable partagée
```

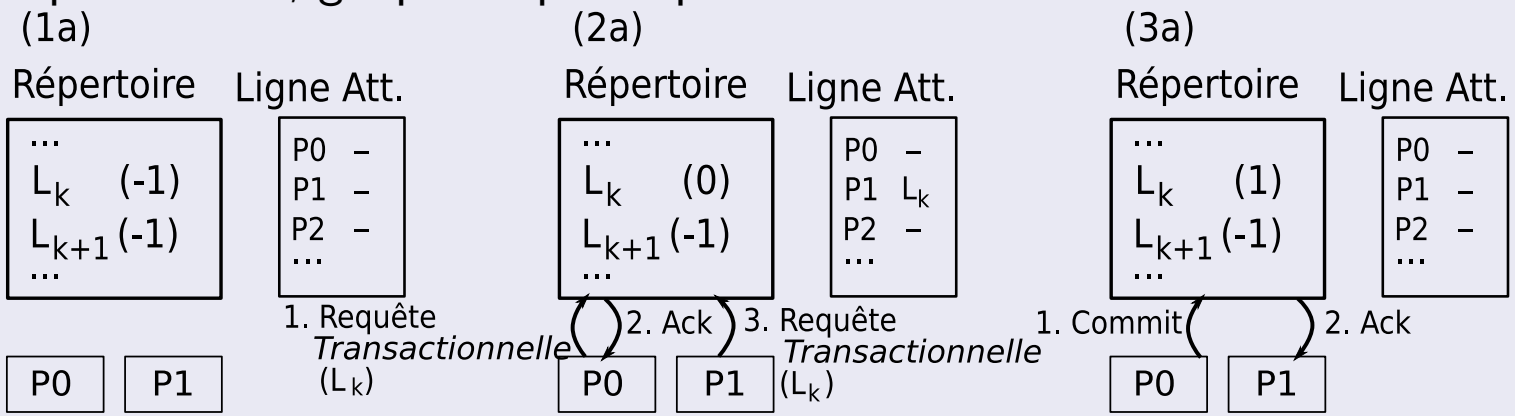
Modifications Architecturales

- Sur les processeurs
  - Ajout d'un banc de registre supplémentaire dans le processeur
  - Besoin de décoder de nouvelles instructions
- Sur les caches et la mémoire
  - Suivi des adresses Lues/Ecrites : bits R/W associés à chaque ligne de cache
  - Enrichissement du protocole cache/processeur et du protocole VCI
  - Modification des différentes machines d'état afin d'implémenter les protocoles étendus avec des transactions

Détection des Conflits

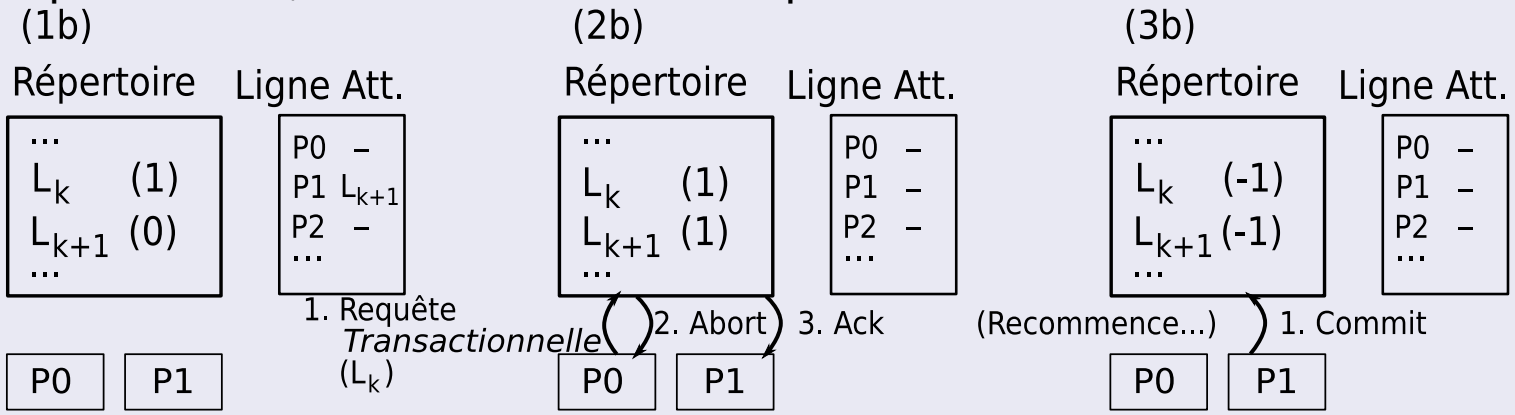
Cas sans Abort

- 2 processeurs, gel puis réponse positive



Cas avec Abort

- 2 processeurs, attente croisée des requêtes



Log et Gestion de Version

Problème

- Classification classique basée sur l'emplacement des anciennes données pas assez précise
- En particulier, ne s'adapte pas au protocole de cohérence write-through
- Besoin de redéfinir l'espace de conception avec le contenu des différentes mémoires

Log et Gestion de Version

Nouvelle Classification

- AD : Anciennes Données, ND : Nouvelles Données, OV : Données qui ont débordé du cache dans la transaction

Cohérence	Mémoire	Cache	Buffer	AD Log
WB, EL	??	AD	ND	-
WB, EE	??	ND	-	AD
WT, EL	??	AD	ND	-
WT, EE (strict)	ND	ND	-	AD
WT, EE (mod)	AD or OV	ND	-	Addr

- Idée : puisqu'une requête par ligne est nécessaire au moment du commit, faire les écriture transactionnelles locales (en cache), et ne mettre à jour la mémoire qu'au commit
- $\rightarrow$ Plus besoin de conserver les anciennes valeurs en log (log néanmoins utilisé pour les adresses)

Politique de Débordement

- Ecrire les valeurs intermédiaires en mémoire
- Requiert qu'au plus un processeur puisse faire un débordement de cache dans une transaction à un instant donné

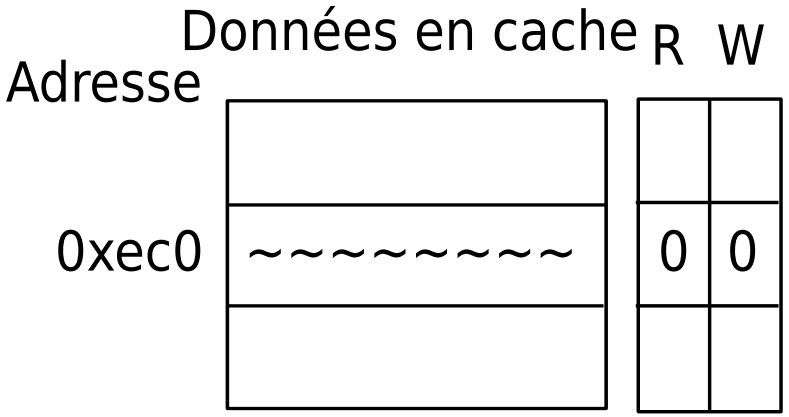
Actions de Haut-Niveau entre le Cache et la Mémoire

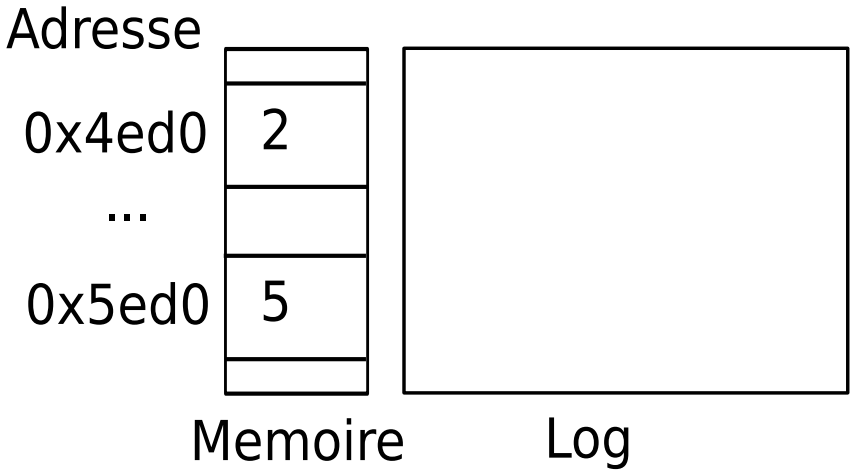
Description des Actions Transactionnelles

Actions	*Quand*	*Description*
MISS	Quand une donnée n'est pas en cache	Va chercher la ligne correspondante en mémoire
T_REQ	Lors du premier accès à une ligne au sein d'une transaction	Demande l'accès exclusif pour cette ligne par le contrôleur de cache
WRITE_LOG	Lors du premier accès à une ligne au sein d'une transaction	Copie l'adresse de la ligne dans le log
WT_UPDATE	Lors d'une écriture au sein d'une transaction	Ecrit la nouvelle valeur dans le cache
WRITE_OV	Quand un débordement de cache se produit au sein d'une transaction	Recopie la ligne de cache en mémoire
MISS_LOG	Quand le log a besoin d'être parcouru (et n'est pas en cache)	Va chercher la ligne de log en mémoire, et la cache si ligne remplacée non-trans.
COMMIT	Quand une transaction commite (données à mettre à jour)	Relâche la ligne et écrit les valeurs
BCOMMIT	Quand une transaction commite (données périmées)	Relâche la ligne
ABORT	Quand un cycle est détecté	Réinitialise le log et recommence la transaction

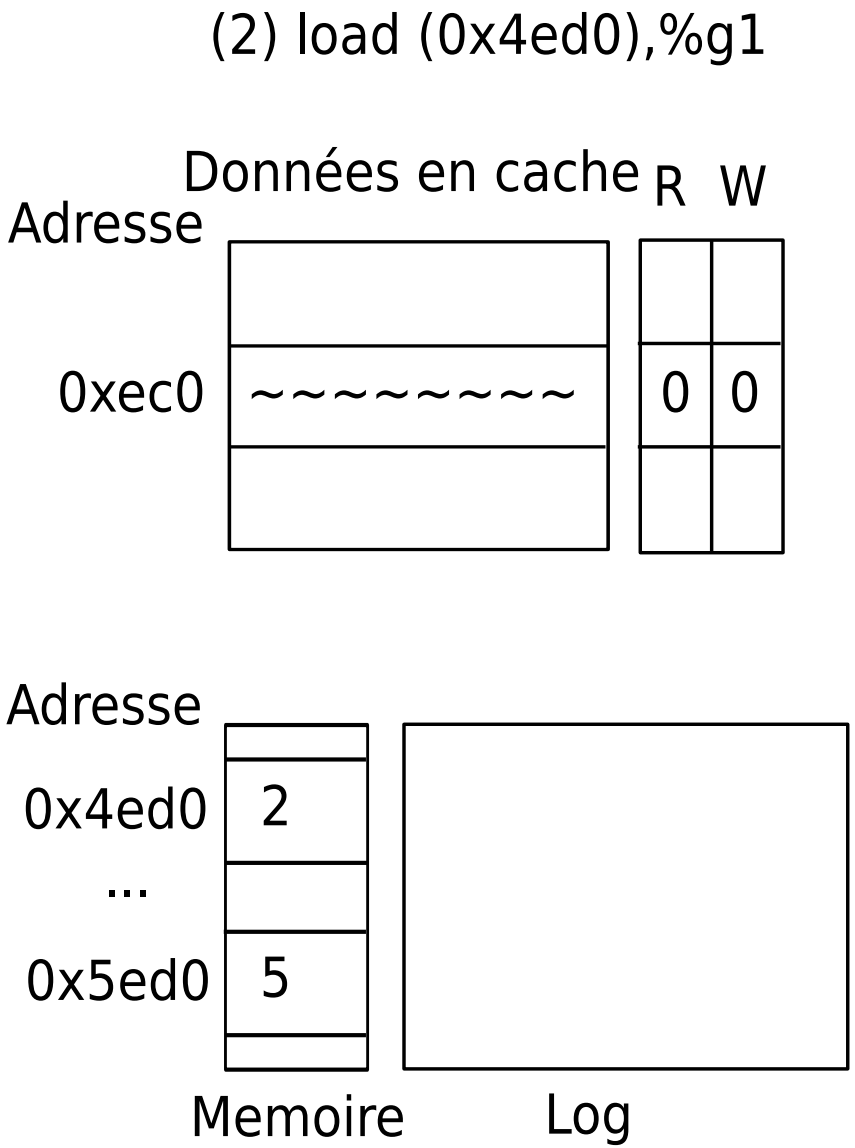
Gestion de Débordements : Exemple Animé

(1) begin_transaction();

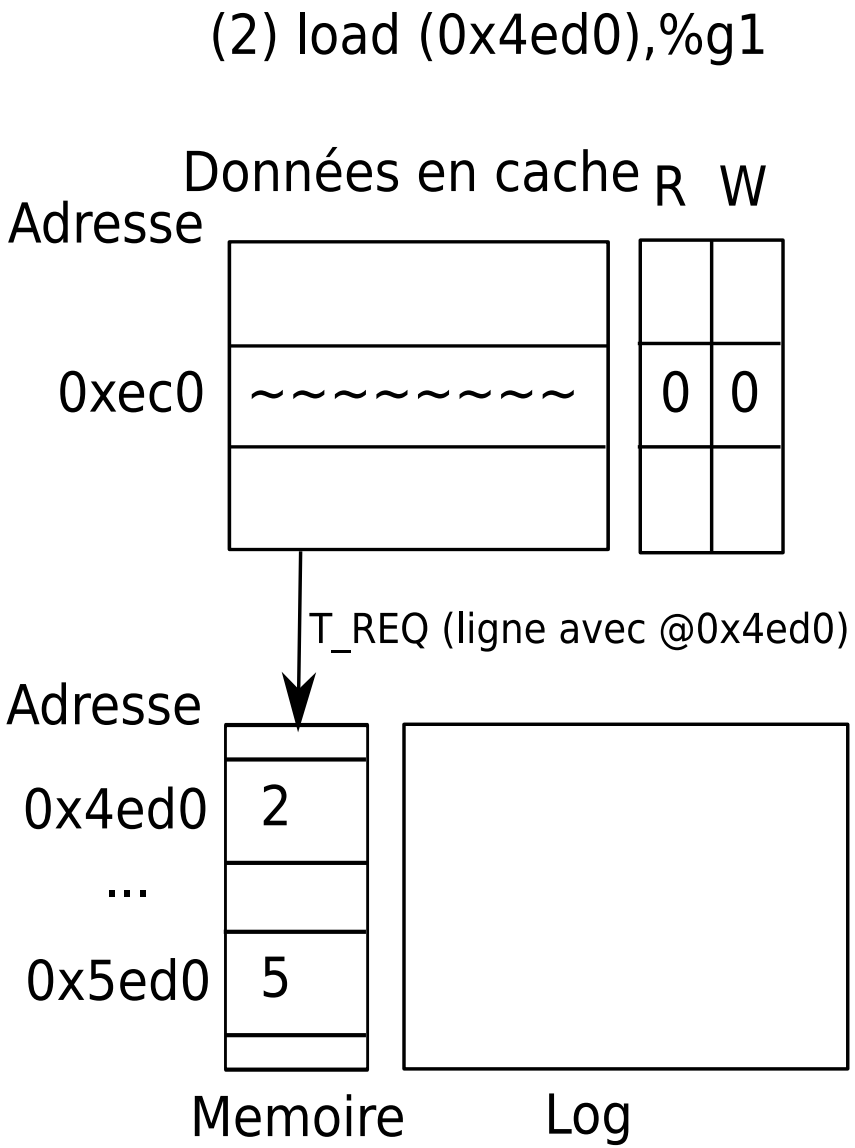




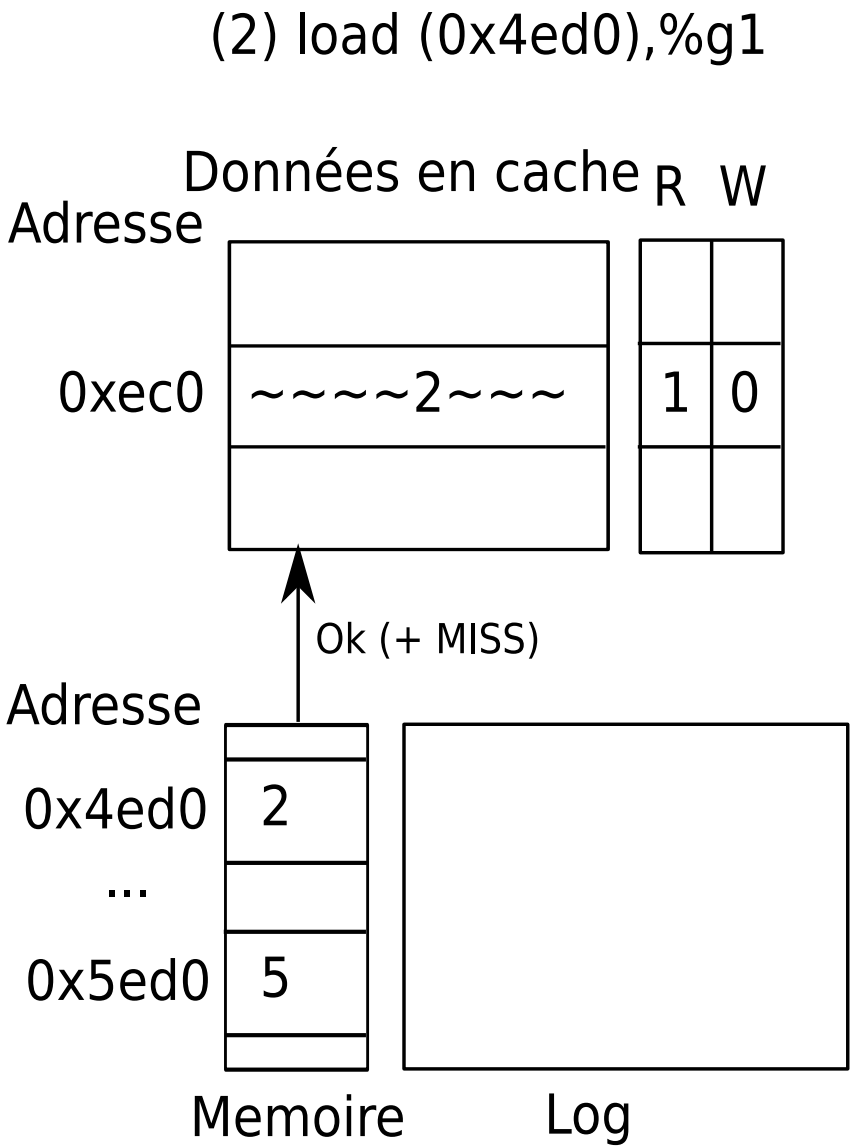
Gestion de Débordements : Exemple Animé



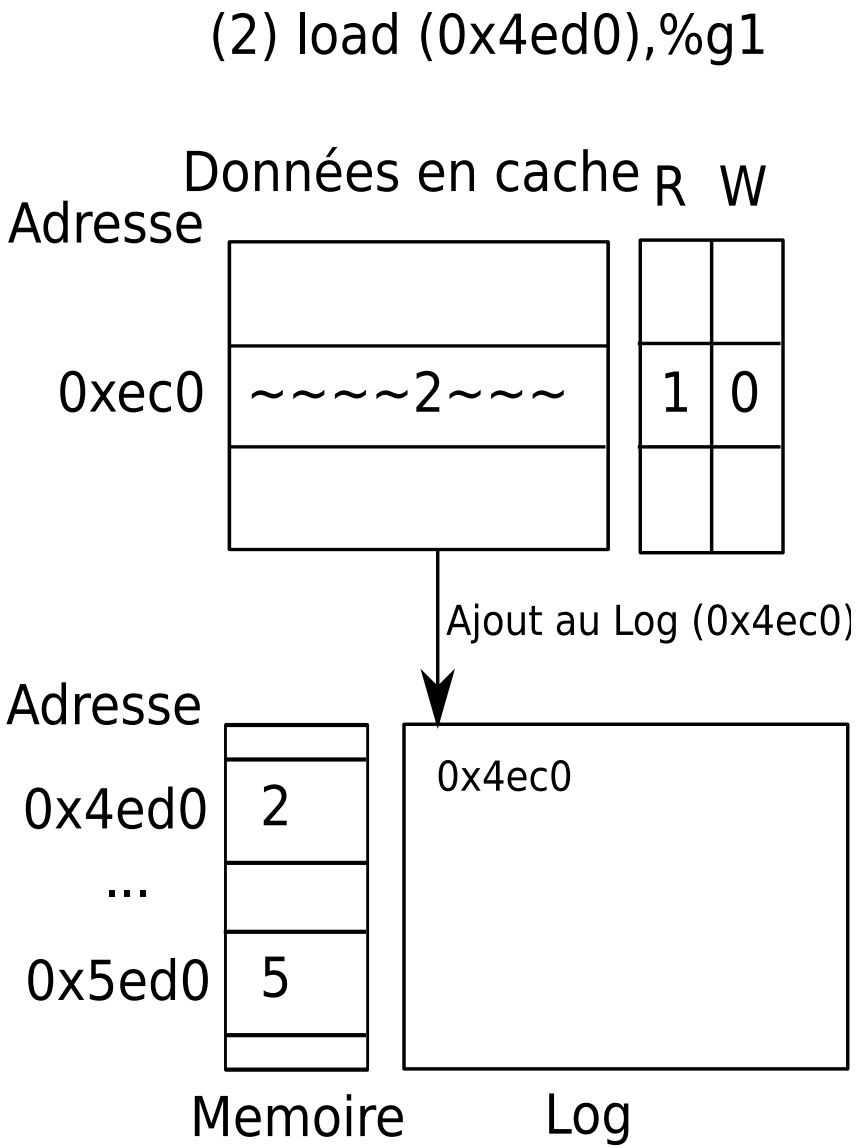
Gestion de Débordements : Exemple Animé



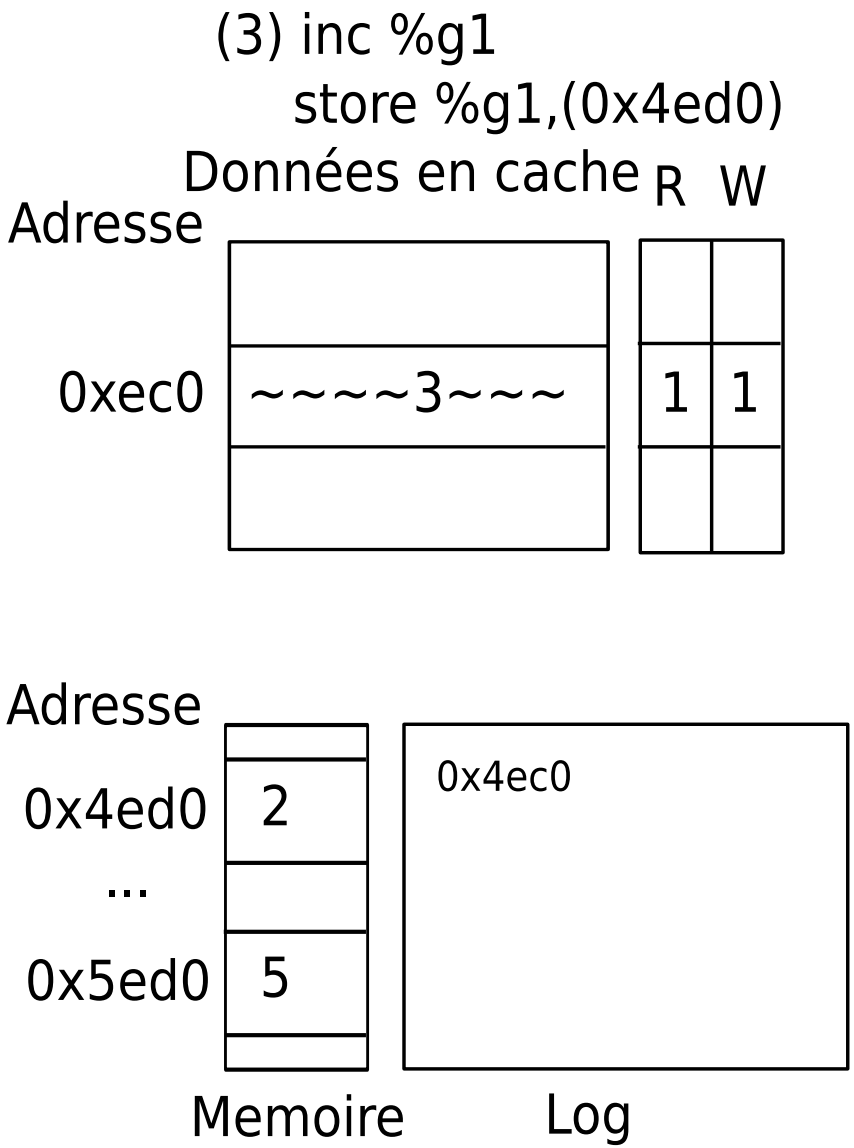
Gestion de Débordements : Exemple Animé



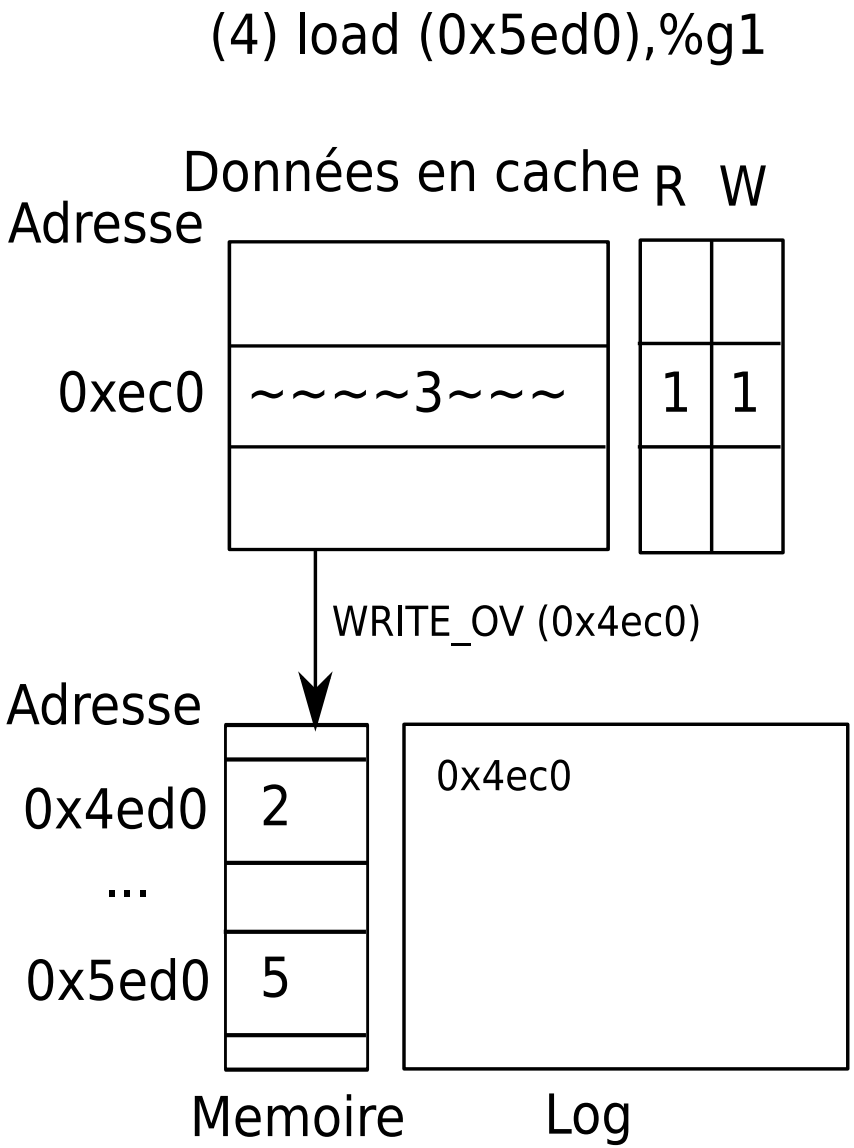
Gestion de Débordements : Exemple Animé



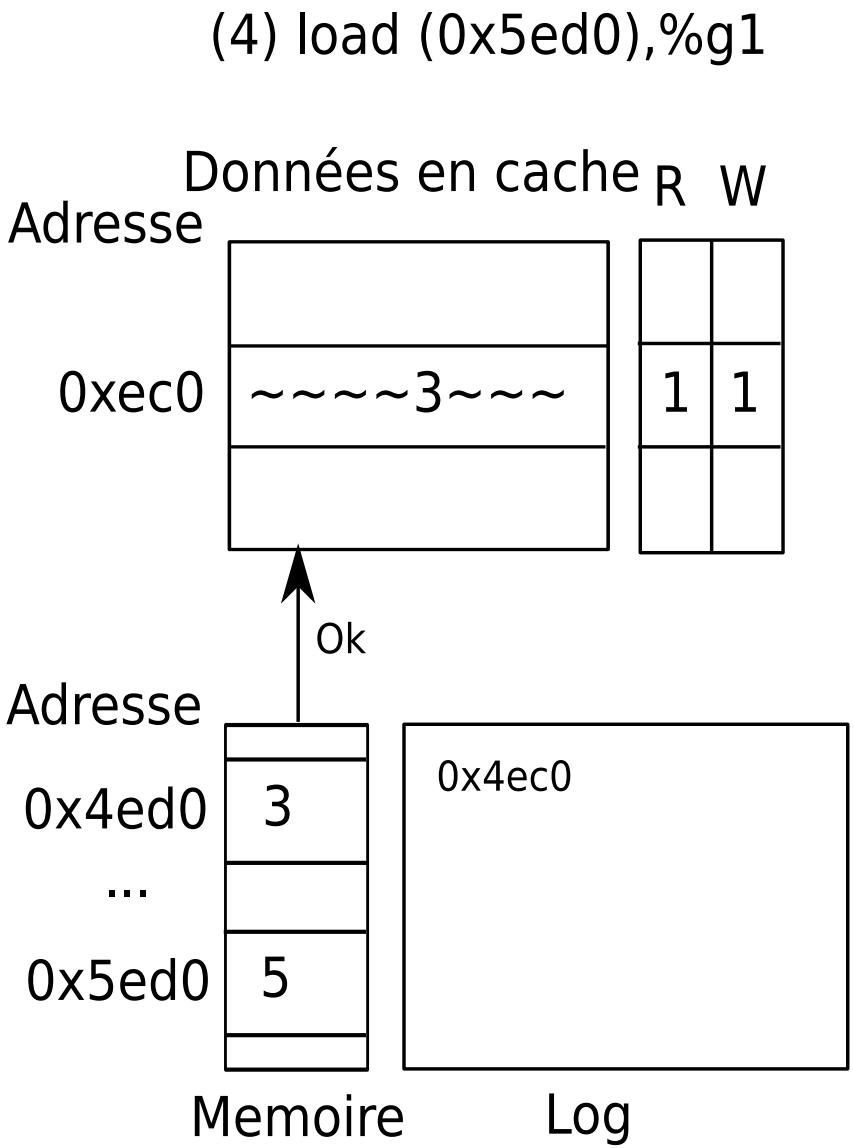
Gestion de Débordements : Exemple Animé



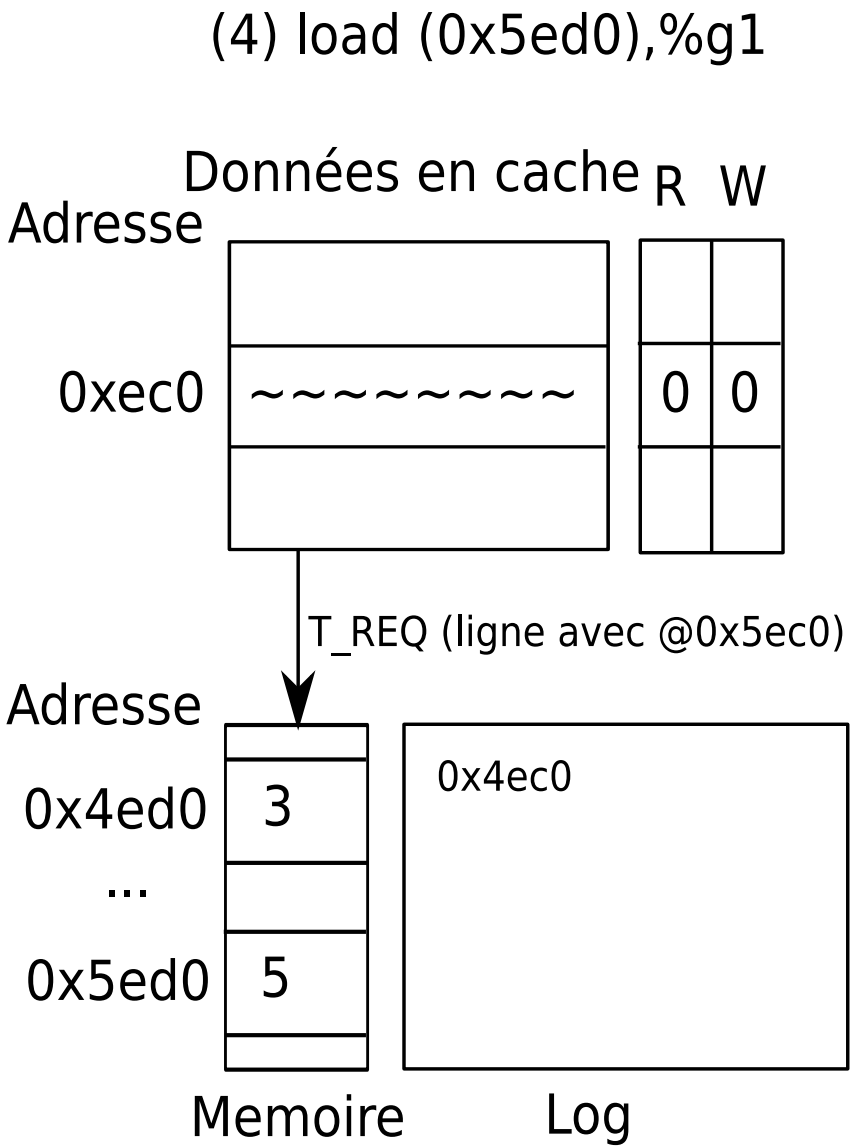
Gestion de Débordements : Exemple Animé



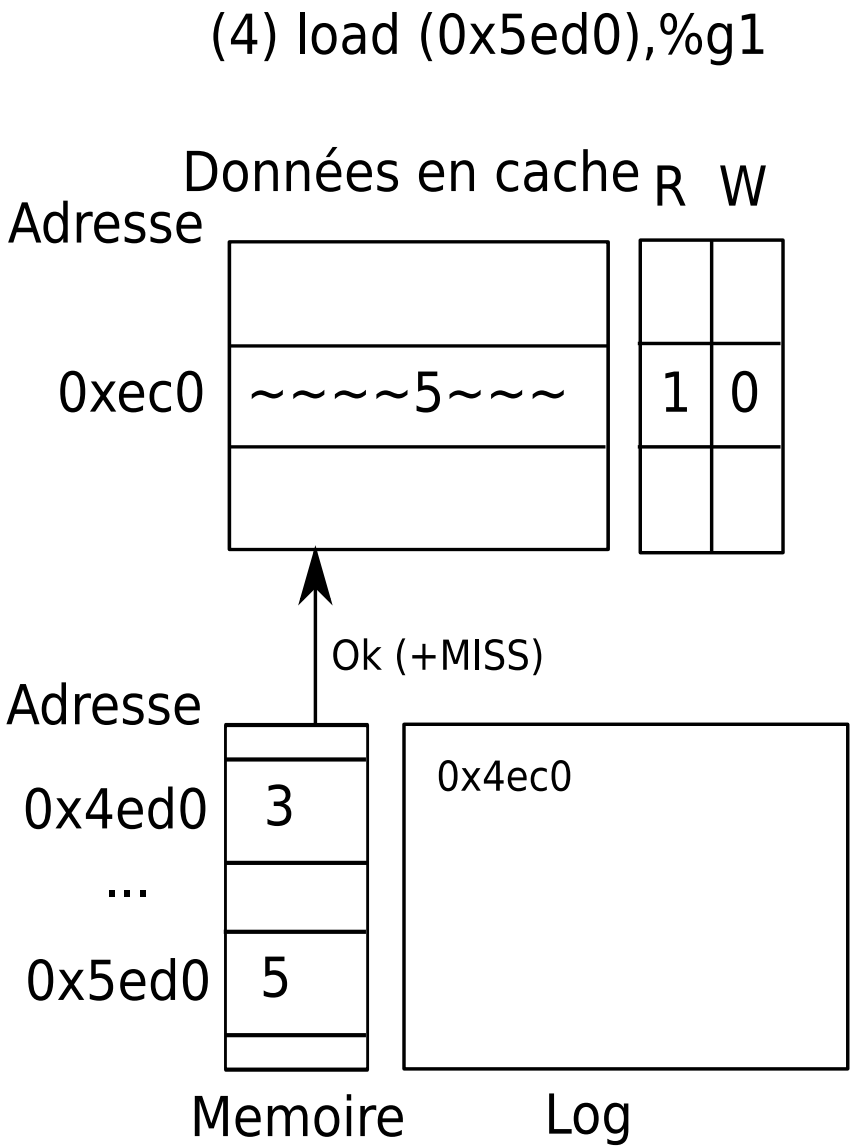
Gestion de Débordements : Exemple Animé



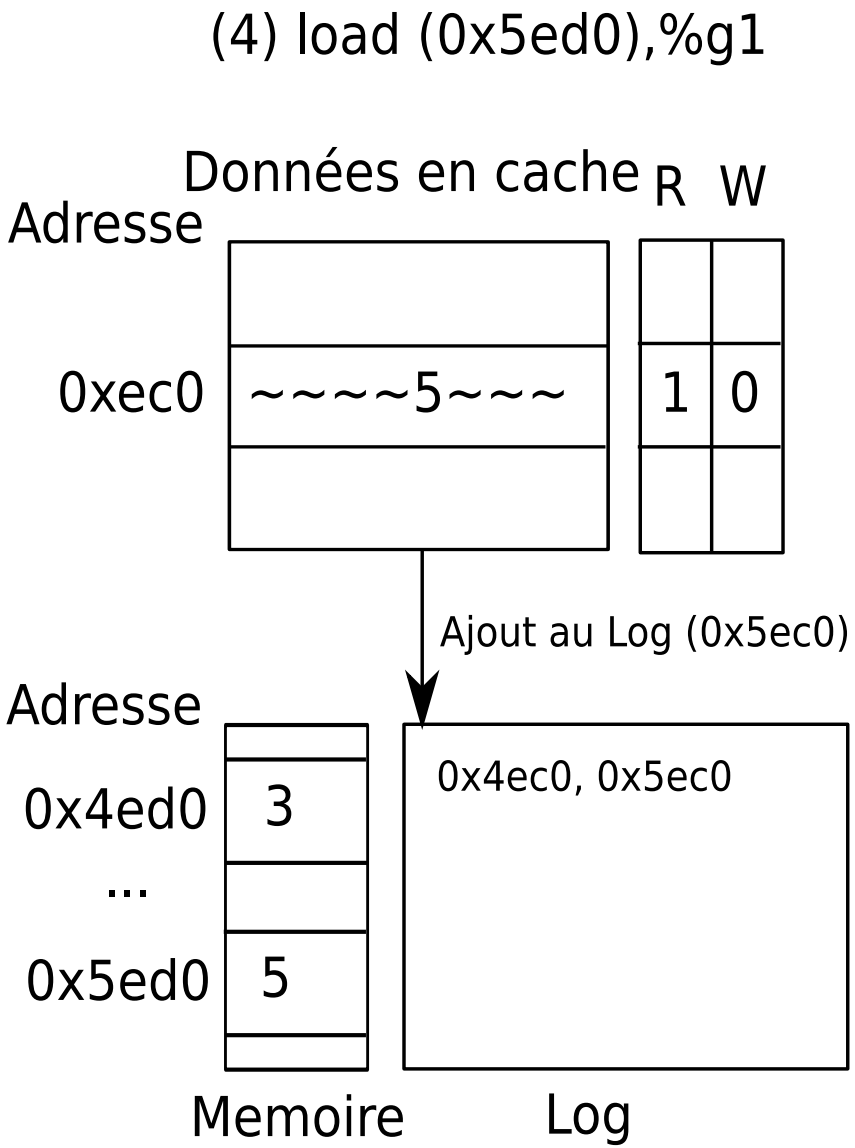
Gestion de Débordements : Exemple Animé



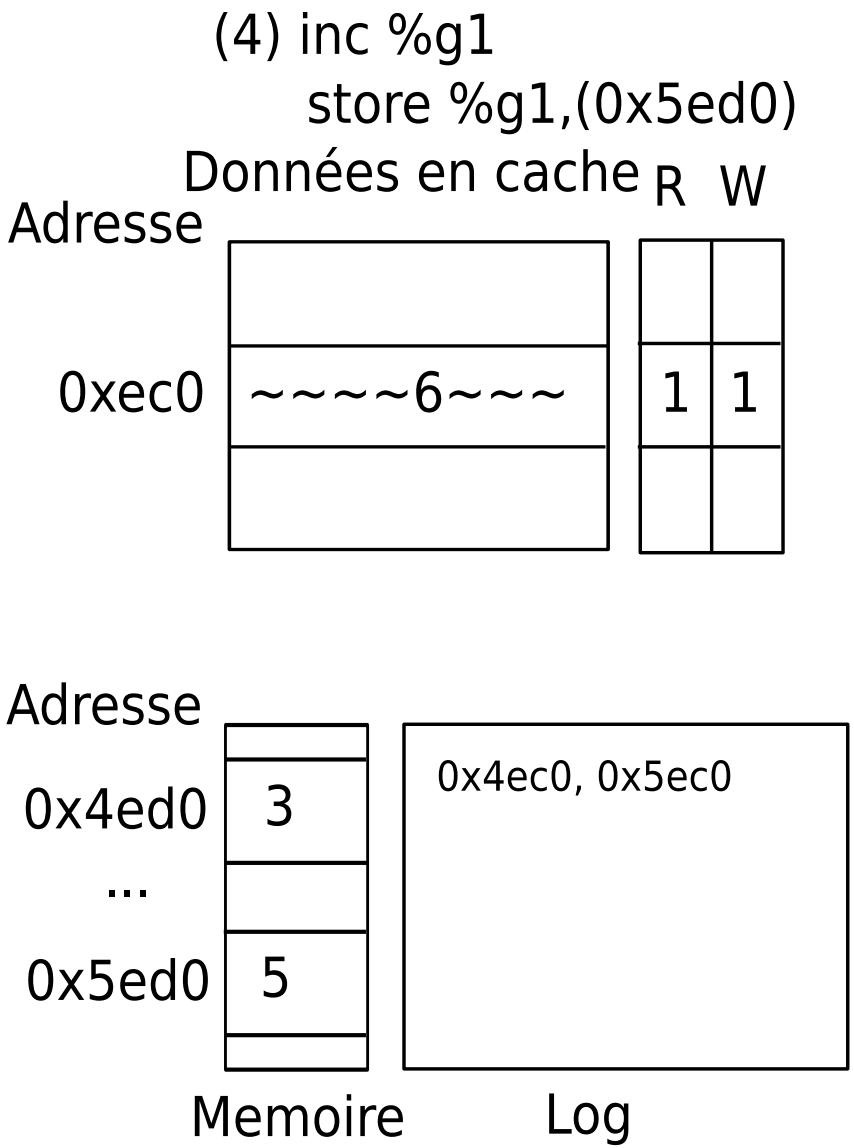
Gestion de Débordements : Exemple Animé



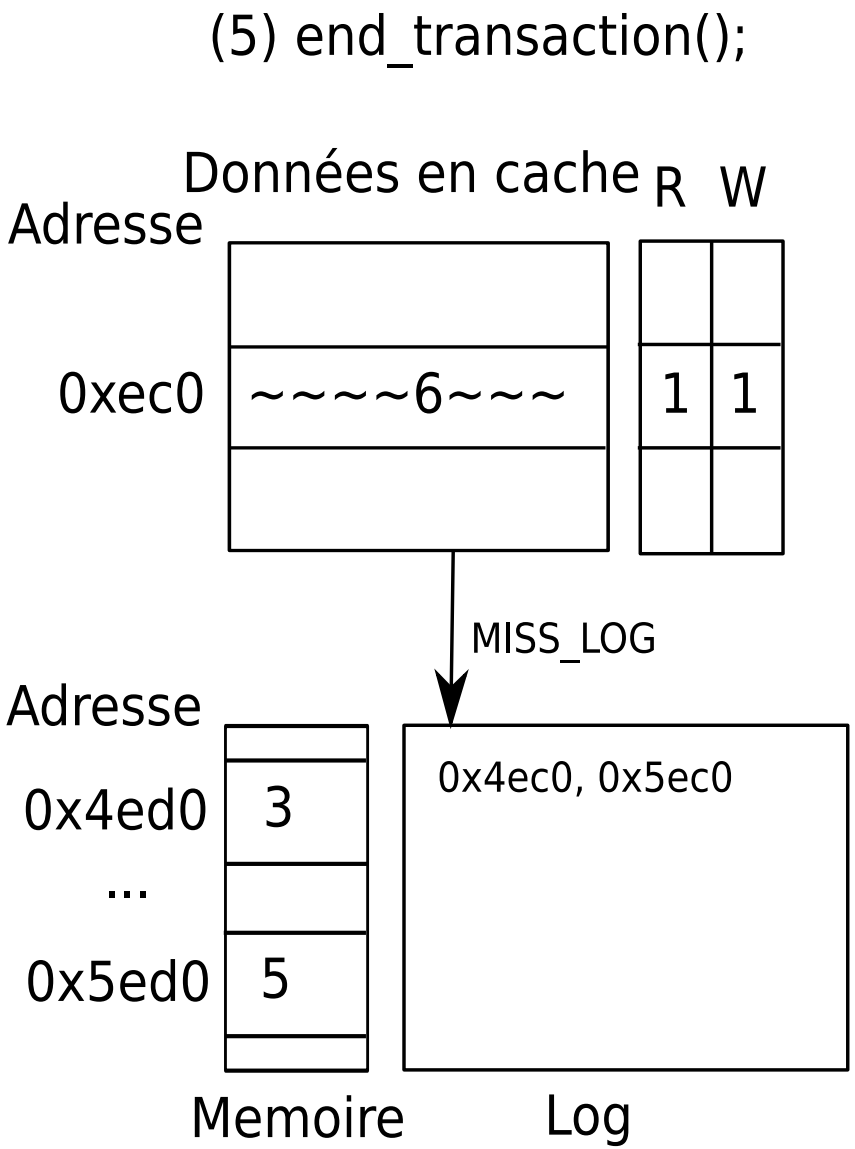
Gestion de Débordements : Exemple Animé



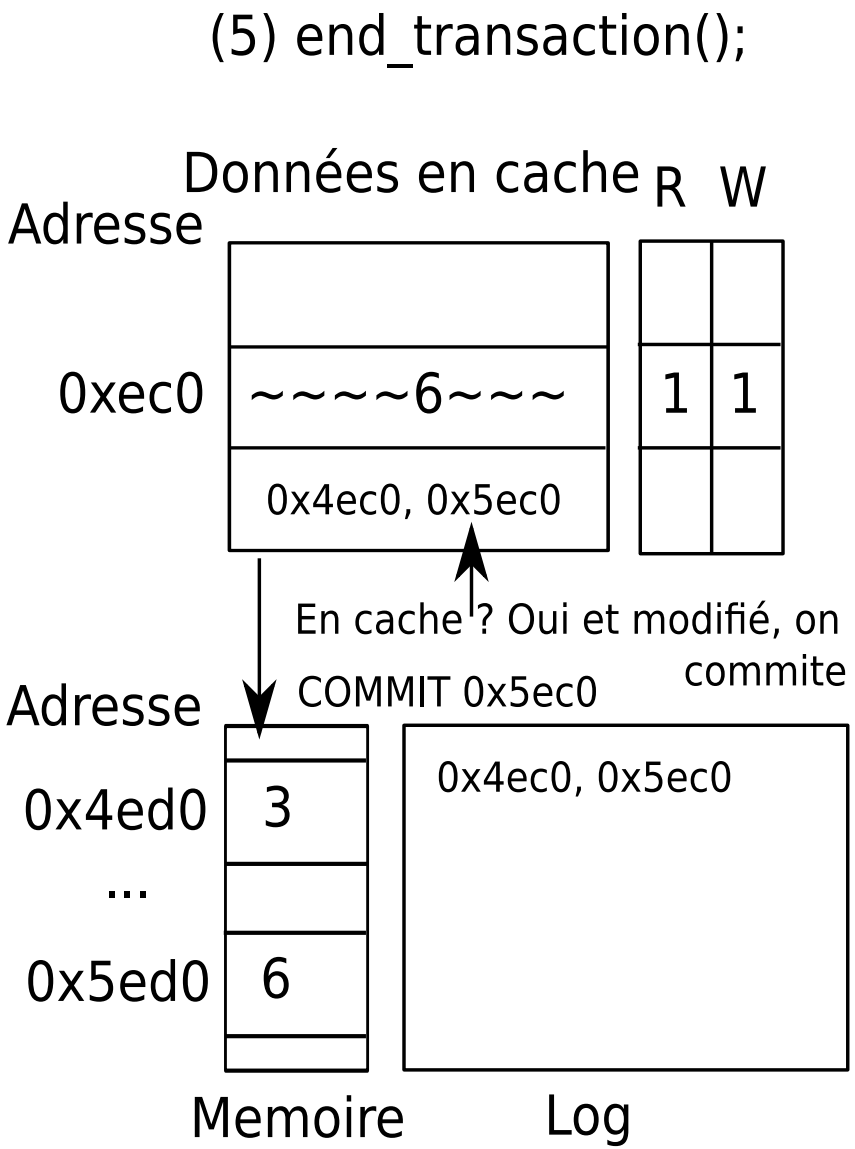
Gestion de Débordements : Exemple Animé



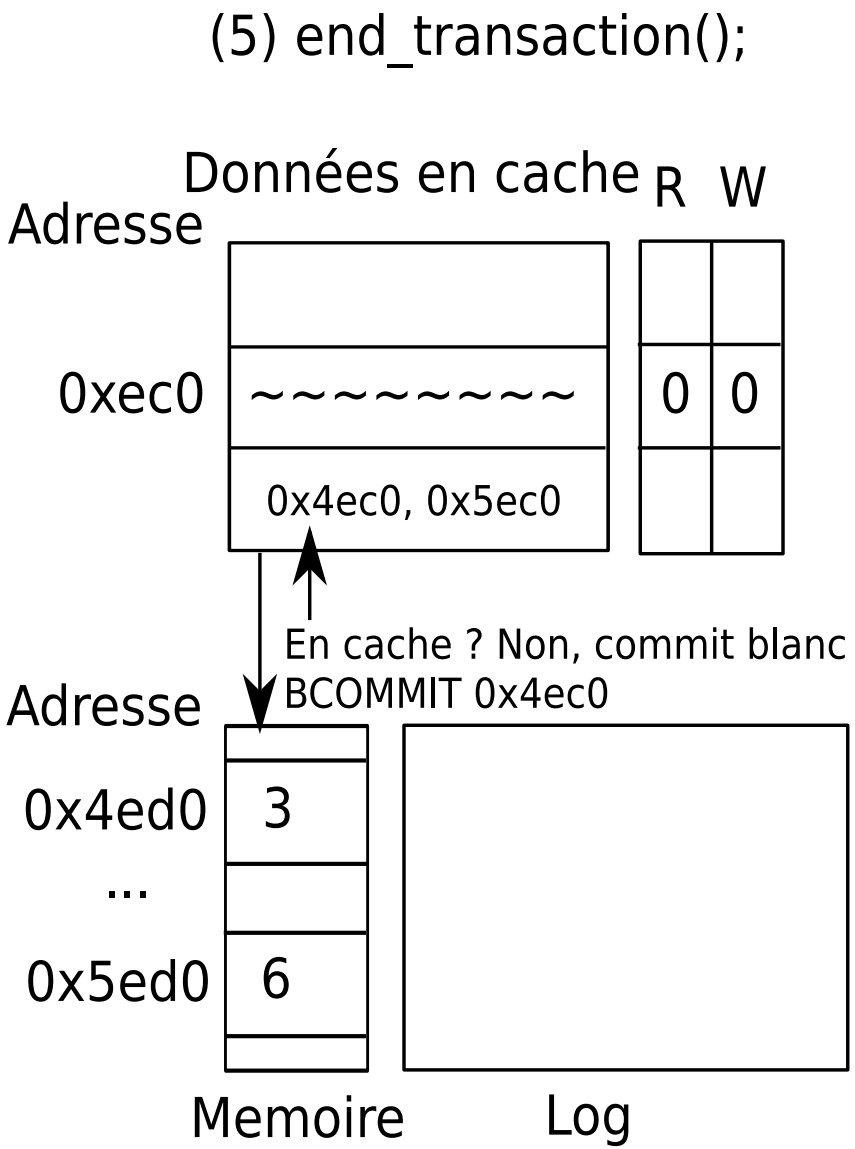
Gestion de Débordements : Exemple Animé



Gestion de Débordements : Exemple Animé



Gestion de Débordements : Exemple Animé



Outline

1 Introduction

2 Généralités sur les Mémoires Transactionnelles

3 Implémentation Spécifique pour les MPSoCs : LightTM

4 Expérimentations et Résultats

5 Conclusion et Perspectives

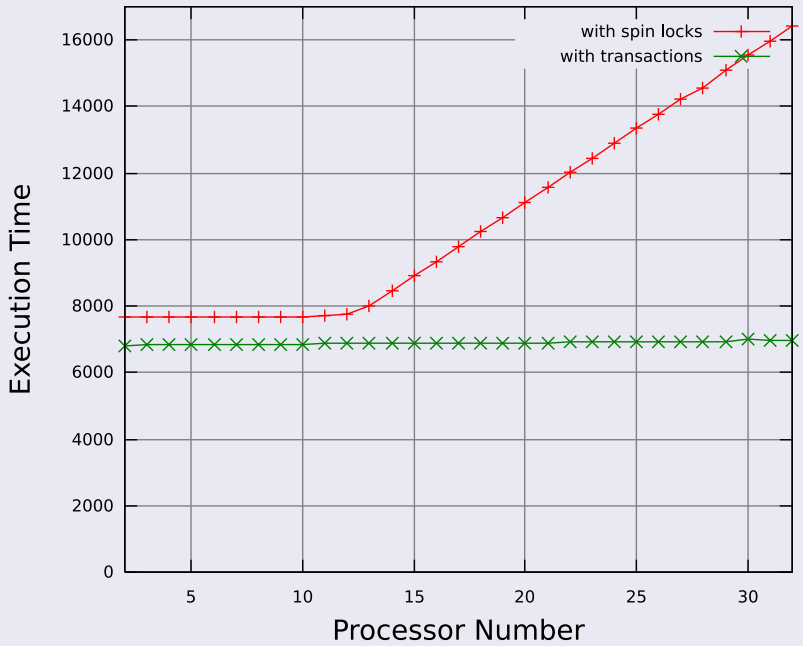
Microkernel : Description et Résultats

Paramètres relatifs à la simulation

- Implémentation "Cycle-Accurate Bit-Accurate" du système LightTM, par dessus des modèles SoCLib existants
- Simulations de 1 à 32 processeurs pour le mircrokernel, et avec 16 processeurs pour les benchmarks
- Utilisation de l'OS Mutek

Microkernel : Description et Résultats

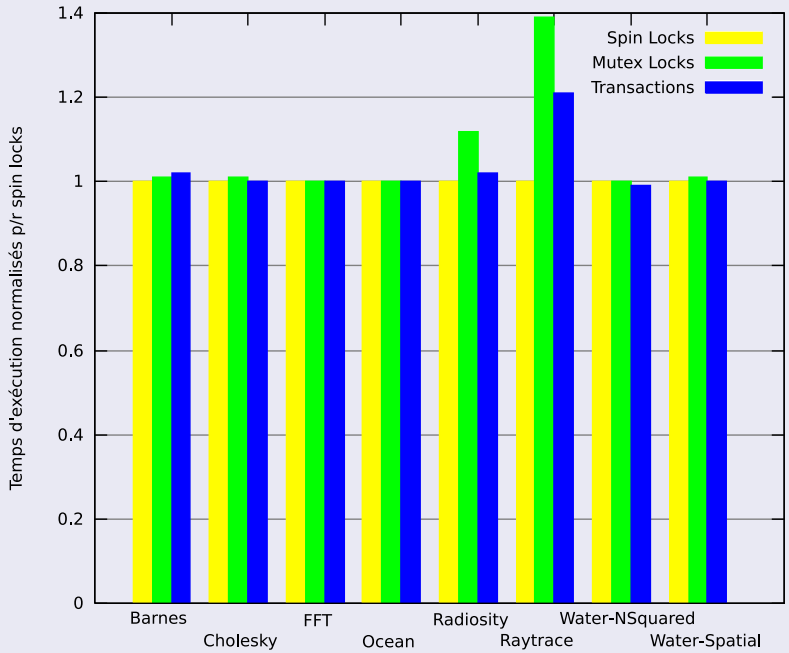
```
int end = 0;
while (end != 1){
    begin_transaction();
    if (shared_var == 10000){
        end = 1;
    }
    else {
        shared_var++;
    }
    end_transaction();
}
```



Benchmarks : Execution des applications Splash-2

Configuration avec les Benchmarks Splash-2 et Résultats

Application	Données en Entrée
Barnes	512 bodies
Cholesky	tk14
FFT	65 536 points complexes
Ocean	contiguous partitions, 258
Radiosity	Room
Raytrace	teapot (256 x 256)
Water N-Squared	512 molécules
Water Spatial	512 molécules



Analyse des résultats

- Temps d'exécution identiques, sauf pour Raytrace (plus lent)
- Loin des résultats de la littérature
- Sections critiques pas assez importantes ?

Outline

1 Introduction

2 Généralités sur les Mémoires Transactionnelles

3 Implémentation Spécifique pour les MPSoCs : LightTM

4 Expérimentations et Résultats

5 Conclusion et Perspectives

Conclusion et Perspectives

Conclusions

- Mémoires transactionnelles
  - Modèle de programmation qui répond au besoin face aux multicoeurs
  - Bonnes performances (?)
- Mais :
  - Coûteux et complexe
  - Pour les MPSoCs, problème dû à l'indéterminisme pour les sytèmes temps réel
  - Réel problème avec les pathologies

Perspectives

- Résultats obtenus moins bons que dans la littérature : quelles hypothèses diffèrent ?
- Implémentation d'un modèle EE plus classique basé sur des caches WB
- Etude des pathologies à bas niveau, influence des choix du système sur les pathologies

LightTM : un Système de Mémoire Transactionnelle pour les MPSoCs

Quentin Meunier, Frédéric Pétrot

Laboratoire TIMA
Groupe SLS
46, Av Félix Viallet, 38031 Grenoble, France

9 Septembre 2009

References I

C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie, *Unbounded transactional memory*, Proc. 11th International Conference on High-Performance Computer Architecture (11th HPCA'05) (San Francisco, CA, USA), IEEE Computer Society, February 2005, pp. 316–327.

Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood, *TokenTM : Efficient execution of large transactions with hardware transactional memory*, Proc. 35th International Symposium on Computer Architecture (35th ISCA'08) (Beijing), ACM SIGARCH, June 2008.

Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee, *Software transactional memory : why is it only a research toy ?*, ACM Queue : Tomorrow's Computing Today **6** (2008), no. 5, 46–58.

Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval, *Bulk disambiguation of speculative threads in multiprocessors*, ISCA, IEEE Computer Society, 2006, pp. 227–238.

References II

Fraser and Harris, *Concurrent programming without locks*, ACMTCS : ACM Transactions on Computer Systems **25** (2007).

Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael K. Chen, Christos Kozyrakis, and Kunle Olukotun, *Transactional coherence and consistency : Simplifying parallel hardware and software*, IEEE Micro **24** (2004), no. 6, 92–103.

Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu Joseph Lu, and Sridhar Narayanan, *TSOtool : A program for verifying memory systems using the memory consistency model*, Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04), ACM Computer Architecture News (CAN) (Munich, Germany), ACM SIGARCH / IEEE Computer Society, June 2004, Published as Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04), ACM Computer Architecture News (CAN), volume 32, number ?, pp. 114–123.

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood, *LogTM : Log-based transactional memory*, Proceedings of the 12th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, February 2006, pp. 254–265.

References III

Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood, *Supporting nested transactional memory in logTM*, Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006 (John Paul Shen and Margaret Martonosi, eds.), ACM, 2006, pp. 359–370.

Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun, *Testing implementations of transactional memory*, Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (15th PACT'06) (Seattle, Washington, USA) (Erik Altman, Kevin Skadron, and Benjamin G. Zorn, eds.), ACM, September 2006, pp. 134–143.

Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai, *Virtualizing transactional memory*, ISCA, IEEE Computer Society, 2005, pp. 494–505.

References IV

Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M. Martin, and David A. Wood, *Specifying and verifying a broadcast and a multicast snooping cache coherence protocol*, Technical Report 1412, Univ. of Wisconsin Computer Sciences, Madison, WI, March 2000.

M.M. Swift, H. Volos, N. Goyal, L. Yen, M.D. Hill, and D.A. Wood, *OS Support for Virtualizing Hardware Transactional Memory*, 2008.

Sasa Tomic, Adrián Cristal, Osman S. Unsal, and Mateo Valero, *Hardware transactional memory with operating system support, HTMOS*, Euro-Par Workshops (Luc Bougé, Martti Forsell, Jesper Larsson Träff, Achim Streit, Wolfgang Ziegler, Michael Alexander, and Stephen Childs, eds.), Lecture Notes in Computer Science, vol. 4854, Springer, 2007, pp. 8–17.

Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood, *LogTM-SE : Decoupling hardware transactional memory from caches*, HPCA, IEEE Computer Society, 2007, pp. 261–272.

Implémentation des Principaux Systèmes HTM existants

Principaux Systèmes HTM existants

Abbr.	Nom Complet	Référence
LogTM	Log Based Transactional Memory[a]	[MBM+06a]
TCC	Transactional Coherence and Consistency	[HCW+04]
VTM	Virtualizing Transactional Memory	[RHL05]
UTM	Unbounded Transactional Memory	[AAK+05]
LTM	Large Transactional Memory	[AAK+05]
Bulk	-	[CTTC06]

[a] and its variants : LogTM-SE [YBM+07], TokenTM [BGH+08] and LogTM-VSE [SVG+08]

Positionnement de ces Systèmes

LogTM	$\longrightarrow$	EE
TCC	$\longrightarrow$	LL
VTM	$\longrightarrow$	EL
UTM	$\longrightarrow$	EE
LTM	$\longrightarrow$	EL
Bulk	$\longrightarrow$	LL

Exemple d'Utilisation d'une API de STM

Insertion dans une liste triée (simplifiée)

```
typedef struct {int key; struct node* next;} node;
typedef struct {node *head;} list;

void insert_in_list(list *l, int k){
   node *n := new node(k);
```

```
      node *prev := l→head;
      node *curr := prev→next;
      while (curr→key < k){
         prev := curr;
         curr := curr→next;
      }
      n→next := curr;
      prev→next := n;
```

```
}
```

Exemple d'Utilisation d'une API de STM

Insertion dans une liste triée (simplifiée)

```
typedef struct {int key; struct node* next;} node;
typedef struct {node *head;} list;

void insert_in_list(list *l, int k){
   node *n := new node(k);
   wstm_transaction *tx;
   do {
      tx := WSTMStartTransaction();
      node *prev := WSTMRead(tx, &(l→head));
      node *curr := WSTMRead(tx, &(prev→next));
      while (curr→key < k){
         prev := curr;
         curr := WSTMRead(tx, &(curr→next));
      }
      n→next := curr;
      WSTMWrite(tx, &(prev→next),n);
   } while (not(WSTMCommitTransaction(tx)));
}
```

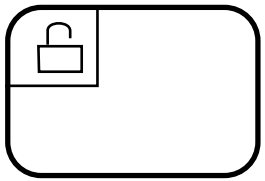
Exemple de Composition avec des Transactions

Description de l'exemple avec des locks

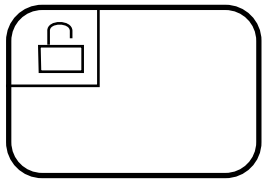
- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

```
    lock(s1.lock);
        // modification des
champs de s1
    unlock(s1.lock);
```

Thread t1 :



s1



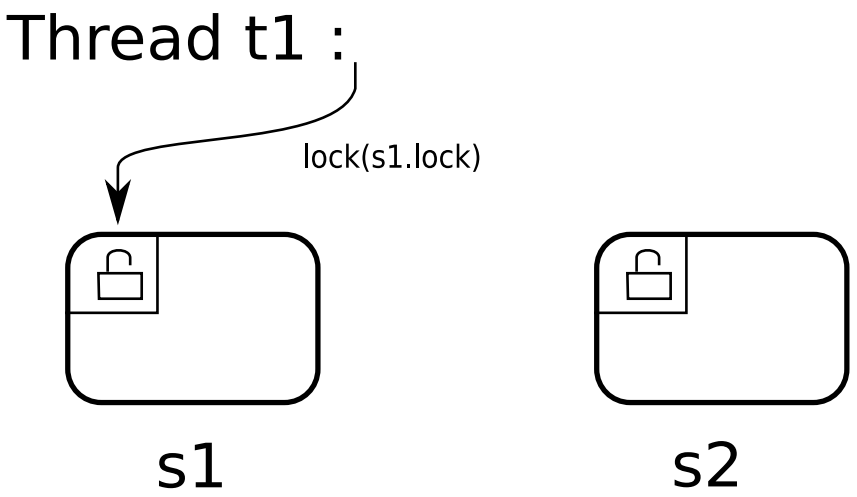
s2

Exemple de Composition avec des Transactions

Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

```
    lock(s1.lock);
        // modification des
champs de s1
    unlock(s1.lock);
```

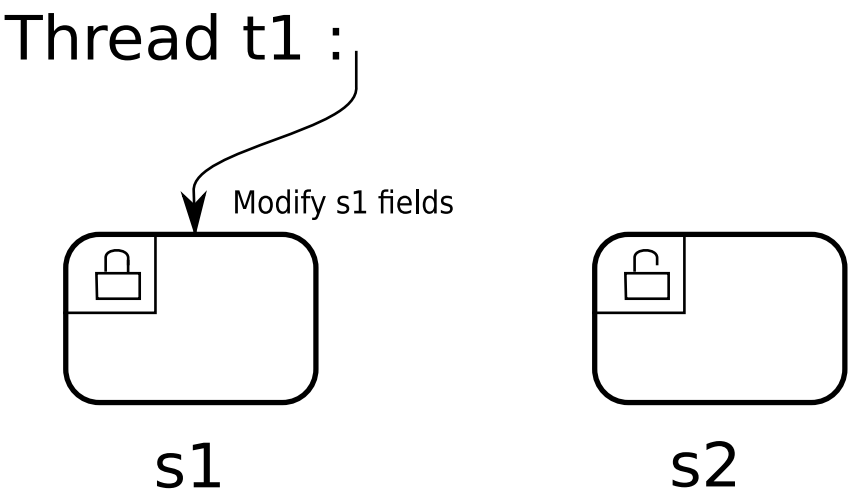


Exemple de Composition avec des Transactions

Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

```
    lock(s1.lock);
        // modification des
champs de s1
    unlock(s1.lock);
```

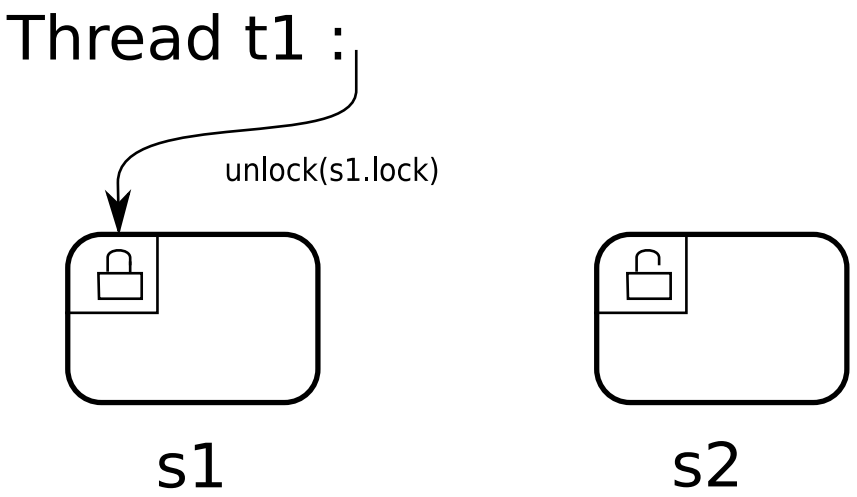


Exemple de Composition avec des Transactions

Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

```
    lock(s1.lock);
        // modification des
champs de s1
    unlock(s1.lock);
```



Exemple de Composition avec des Transactions

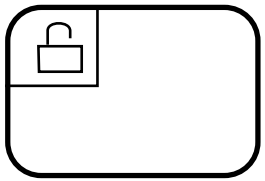
Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

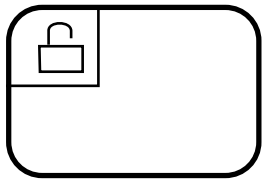
```
    lock(s1.lock);
        // modification des
champs de s1
    unlock(s1.lock);
```

- Faire des opérations atomiques entre deux objets
  - Risque de deadlock
  - Ou impose un ordre total sur les structures
  - Ou requiert un lock supplémentaire par couple de structure
  - $\Rightarrow$ Nouvelle interface

Thread t1 :



s1



s2

Exemple de Composition avec des Transactions

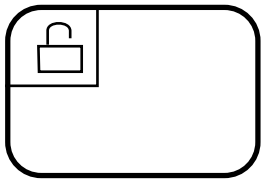
Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

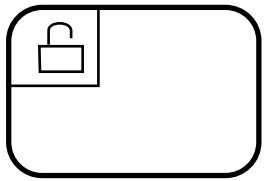
```
lock(s1.lock);
    // modification des champs de s1
unlock(s1.lock);
```

- Faire des opérations atomiques entre deux objets
  - Risque de deadlock
  - Ou impose un ordre total sur les structures
  - Ou requiert un lock supplémentaire par couple de structure
  - ⇒ Nouvelle interface

Thread t1 :



s1



s2

Avec des Transactions

Exemple de Composition avec des Transactions

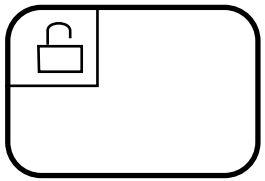
Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

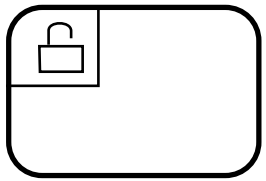
```
lock(s1.lock);
    // modification des champs de s1
unlock(s1.lock);
```

- Faire des opérations atomiques entre deux objets
  - Risque de deadlock
  - Ou impose un ordre total sur les structures
  - Ou requiert un lock supplémentaire par couple de structure
  - ⇒ Nouvelle interface

Thread t1 :



s1



s2

Avec des Transactions

- Modifier une structure :

```
begin_transaction();
    // modification de s1
end_transaction();
```

Exemple de Composition avec des Transactions

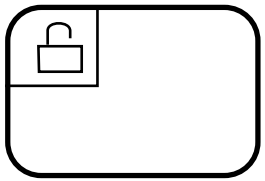
Description de l'exemple avec des locks

- Plusieurs structures partagées, un lock par structure
- Modifier une structure s1 :

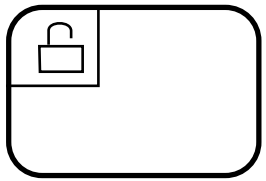
```
lock(s1.lock) ;
    // modification des champs de s1
unlock(s1.lock) ;
```

- Faire des opérations atomiques entre deux objets
  - Risque de deadlock
  - Ou impose un ordre total sur les structures
  - Ou requiert un lock supplémentaire par couple de structure
  - ⇒ Nouvelle interface

Thread t1 :



s1



s2

Avec des Transactions

- Modifier une structure :

```
begin_transaction() ;
    // modification de s1
end_transaction() ;
```

- Modifier deux structures de manière atomique :

```
begin_transaction() ;
    // modification de s1
    // modification de s2
end_transaction() ;
```

Sémantique d'Entrelacement

Solutions proposées/existantes [MBM+06b]

- À plat :
  - La transaction commite complètement lorsque le `end_transaction()` le plus externe est atteint : on ignore les transactions internes
- Fermée :
  - Abort interne : abort partiel (ne recommence que les transactions correspondantes)
  - Commits internes : commits partiels
  - Coûts matériels beaucoup plus importants que pour la sémantique à plat
- Ouverte :
  - Commits internes : "vrais commits", i.e. visibles par les autres processeurs
  - Nécessite des "Opérations d'annulation" associées aux transactions internes pour le cas d'abort
  - Complexe

Sémantique d'Entrelacement

Problème

- Une transaction peut toujours avoir à aborter (quelque soit le choix de conception)
- $\Rightarrow$ Sens du code suivant, en particulier en cas d'abort dans une transaction interne :

```
begin_transaction() ;
    begin_transaction() ;
        x = x + 1 ;
    end_transaction() ;
    ...
    begin_transaction() ;
        a = a + 1 ;
        b = b + 1 ;
    end_transaction() ;
end_transaction() ;
```

- $\rightarrow$ Notion de sémantique d'entrelacement

Log et Gestion de Version

Protocole Transactionnel

- Durant une transaction, quand une requête (R ou W) se produit, une requête est envoyée vers la mémoire pour obtenir un accès "transactionnel" sur la ligne
- Ensuite, la ligne correspondante est chargée dans le cache si elle ne l'est pas déjà (Possibilité de fusionner ces deux requêtes)
- Si accès autorisé
  - Adresse de la ligne ajoutée au log
  - Requêtes sur cette ligne traitées localement
  - Au commit : log parcouru pour avertir la mémoire des lignes relâchées + m-à-j lignes modifiées
  - Lors d'un débordement, requête spéciale envoyée à la mémoire pour s'assurer qu'aucun autre processeur n'a déjà débordé
- Si accès refusé (abort)
  - Le contrôleur de cache invalide les lignes correspondantes, remet à 0 les bits R et W associés, et relâche les lignes possédées
- Aborts détectés pour des requêtes cycliques
  - Pour les cycles de longueur 2, si un cache a débordé, l'autre aborte, sinon celui ayant fait la requête aborte
  - Pour les cycles de longueur > 2, si un cache du cycle a débordé, les autres abortent (sinon, décision faite sur un index) ; puis un protocole est mis en place pour s'assurer qu'une progression est faite

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :
  - ligne 2 : **T_REQ**, **MISS** (seulement si la ligne contenant x n'est pas déjà en cache), **WRITE_LOG**, **WT_UPDATE**

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :
  - ligne 2 : **T_REQ**, **MISS** (seulement si la ligne contenant x n'est pas déjà en cache), **WRITE_LOG**, **WT_UPDATE**
  - ligne 3 : **WT_UPDATE**

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :
  - ligne 2 : **T_REQ**, **MISS** (seulement si la ligne contenant x n'est pas déjà en cache), **WRITE_LOG**, **WT_UPDATE**
  - ligne 3 : **WT_UPDATE**
  - ligne 4 : **WRITE_OV**, **T_REQ**, **MISS**, **WRITE_LOG**, **WT_UPDATE**

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :
  - ligne 2 : **T_REQ**, **MISS** (seulement si la ligne contenant x n'est pas déjà en cache), **WRITE_LOG**, **WT_UPDATE**
  - ligne 3 : **WT_UPDATE**
  - ligne 4 : **WRITE_OV**, **T_REQ**, **MISS**, **WRITE_LOG**, **WT_UPDATE**
  - ligne 5 : **WRITE_OV**, **T_REQ** (il est impossible de détecter s'il s'agit de la première modification de la ligne dans cette transaction sans parcourir le log), **MISS**, **WRITE_LOG**, **WT_UPDATE**

Actions de Haut-Niveau entre le Cache et la Mémoire

Exemples d'Actions prises dans une Transaction

- Hypothèses : x et y sont deux variables situées dans deux lignes mémoires différentes mais allouées dans la même ligne de cache
- Soit le code suivant (non optimizé) :

```
begin_transaction() ;
    x = 1 ;
    x = x + 4 ;
    y = 1 ;
    x = 2 ;
end_transaction() ;
```

- Actions prises :
  - ligne 2 : **T_REQ**, **MISS** (seulement si la ligne contenant x n'est pas déjà en cache), **WRITE_LOG**, **WT_UPDATE**
  - ligne 3 : **WT_UPDATE**
  - ligne 4 : **WRITE_OV**, **T_REQ**, **MISS**, **WRITE_LOG**, **WT_UPDATE**
  - ligne 5 : **WRITE_OV**, **T_REQ** (il est impossible de détecter s'il s'agit de la première modification de la ligne dans cette transaction sans parcourir le log), **MISS**, **WRITE_LOG**, **WT_UPDATE**
  - ligne 6 : **MISS_LOG**, **COMMIT** (x), **BCOMMIT** (y), **BCOMMIT** (x)