Mémoires Transactionnelles

Gaël Thomas
gael.thomas@lip6.fr
Basé sur le cours de Herlihy & Shavit

Université Pierre et Marie Curie
Master Informatique
M2 – Spécialité SAR

Limite à l'accélération

Limite : la loi d'Amdahl

p : pourcentage du code exécutable en parallèle
⇒ (1 – p) : pourcentage du code exécuté en séquentiel
⇒ p/n : exécution du code parallèle sur n cœurs
⇒ **a = 1/(1 – p + p/n)** : accélération maximale théorique
⇒ limite pour n → ∞ : a → 1/(1 – p)

Application numérique :
p = 0,25 ⇒ a → 1/0,75 = 4 quand n → ∞ (3,7 à 32 cœurs)
p = 0,95 ⇒ a → 1/0,95 = 20 quand n → ∞ (12,55 à 32 cœurs, 17,42 à 128 cœurs)

⇒ ça vaut le coup de se battre pour paralléliser les quelques pourcents restants!

Solution naturelle

Prendre les verrous les plus fins possibles

```
lock(x)
…
…          0% en //
…
…
unlock(x)
```

```
lock(x)
…
unlock(x);
…
…          50% en //
lock(y);
…
unlock(y)
```

```
CAS(x)
…
CAS(z)     95% en //
…
…
TAS(y)
```


Limites du verrouillage fin

Verrouillage fin ⇒ complexifie le code
⇒ Difficile à maintenir, à faire évoluer
⇒ Bug (très!) difficile à trouver
⇒ Preuve de programme quasi-impossible à faire
(pas de famine, pas de deadlock?)
⇒ **Difficile à composer**,
⇒ Effet de bord important
(protocole d'accès à une variable n'existe souvent que dans la tête du programmeur…)

Verrouillage fin : reste très pessimiste!
Nombre de cœurs augmente ⇒ probabilité de conflit augmente
Conflit ⇒ cœurs oisifs

Les mémoires transactionnelles

Le verrouillage pessimiste n'est plus adéquat à l'heure du multicœurs

Pourquoi prendre des verrous :

Pour exécuter de façon atomique un ensemble d'instructions

Proposition des mémoires transactionnelles :

Définir des blocs atomiques

Les exécuter de façon optimiste, si conflit, on recommence

Avantages attendus :

→ simplifier le code : plus besoin de connaître l'ordre de prise des verrous

→ non-deadlock, non-famine, composabilité assuré par la plateforme

→ ne bloquer un cœur que si c'est strictement nécessaire (i.e. car conflit)

Les mémoires transactionnelles

Définition d'un bloc atomique

Un bloc atomique est exécuté de façon atomique ☺

Plus formellement, si B est un bloc atomique, alors B semble s'exécuter instantanément entre son début et se fin

- ✓ Les variables lues par B correspondent aux variables à un instant dans le temps
- ✓ Les variables écrites par B son modifiées exactement à cet instant là

Initialement : x = 21

```
a. atomic {          f. atomic {
b.   tmp = x;        g.   tmp = x;
c.   tmp = tmp + 1;  h.   tmp = tmp * 2;
d.   x = tmp;        i.   x = tmp;
e. }                 j. }
```

Deux ordonnancements possibles :

*[b,c,e] **puis** [g,h,i] (⇒ 44) **ou** [g,h,i] **puis** [b,c,e] (⇒ 42)*

Les mémoires transactionnelles

Passer du verrouillage optimiste aux transactions

```
synchronized(o)
 if(!x) {
   x = true;
   doSomething();
 }
}
```

0% en //

```
atomic {
 if(!x) {
   x = true;
   doSomething();
 }
}
```

Parallélisme maximal?

x modifié en // pendant la transaction
⇒ transaction **avortée et recommence**

Si x est accédé en lecture en //, soit la valeurs initiale de x est lue, soit **l'autre transaction** est avortée (et recommence)

Mémoire transactionnelle et événements

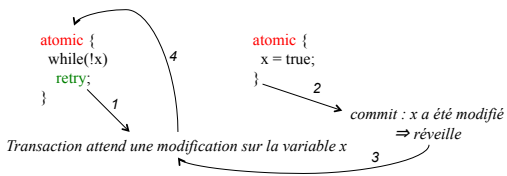
La notion de retry [Harris05] : construction d'une attente (notion proche de la variable condition)

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

```
atomic {
 while(!x)
  retry;
}
```

```
atomic {
 x = true;
}
```

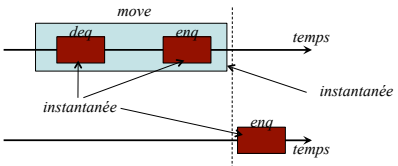


Composition et mémoires transactionnelles

Si A et B sont deux blocs atomiques, on peut les composer dans un autre bloc atomique

Exemple : la queue

```
atomic move(Queue dst, Queue src) {
  Elmt e = src.deq();   ──→ atomic Elmt deq()
  dst.enq(e);           ──→ atomic void enq(Elmt e);
}
```



Composition et mémoires transactionnelles

Composition et attente sur événement : exemple complet

```
class Queue {
                    LinkedList<Elmt> queue;

  void enq(Elmt e) {
    atomic {
      if(queue.size() == MAX_SIZE)
        retry;
      queue.addLast()
    }
  }
}
```

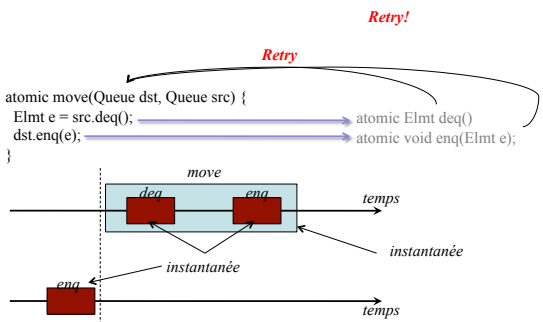
Fait redémarrer la transaction

```
Elmt deq() {
  atomic {
    if(queue.empty())
      retry;
    return queue.removeLast()
  }
}
```

Attend qu'une des variables lues soit modifiée et recommence la transaction

Composition et mémoires transactionnelles

Composition et attente sur événement



Composition et mémoires transactionnelles

Composition par alternative :

Problème : ne pas attendre que la file soit pleine à partir de deq()

```
Elmt deqNoWait() {
 atomic {
   return deq();      Exécute premier bloc
 } orElse {           Si retry
   return null;       Exécute seconde bloc
 }                    Si retry encore (impossible ici),
}                           attend un changement
                      et recommence la transaction
```

Mise en œuvre des mémoires transactionnelles

Mémoire transactionnelle retardée versus immédiate

Mémoire transactionnelle retardée (deferred-update) :

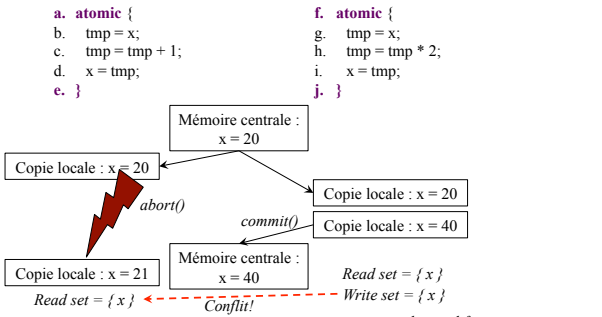
- ✓ Écriture dans une copie locale, propagée en mémoire centrale au commit
- ✓ Du travail lors du commit
- ✓ Cohérence facile à maintenir

Mémoire transactionnelle immédiate (direct-update) :

- ✓ Écriture directement dans la mémoire centrale, annulation si transaction avorte
- ✓ Très peu de travail lors du commit
- ✓ Cohérence plus difficile à maintenir (doit garder les valeurs originales si la transaction avorte)

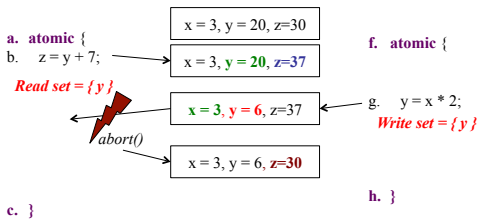
Mise en œuvre des mémoires transactionnelles

Mémoire transactionnelle retardée : gagne si beaucoup de abort



Mise en œuvre des mémoires transactionnelles

Mémoire transactionnelle immédiate : gagne si beaucoup de success



Mise en œuvre des mémoires transactionnelles

Détection des conflits : garder les ensembles de valeurs lues ou écrites

En cas de conflit lecteur/écrivain

Annuler une des transactions

```
a. atomic {
b.   x++;
c. }
```

En cas de conflit écrivain/écrivain

Annuler une des transactions

```
a. atomic {
b.   x = *p;
c.   y = *p + 1;
d. }
```

(doit respecter l'invariant y = x +1)

Mise en œuvre des mémoires transactionnelles

Deux systèmes de détection de conflit

Détection au plus tôt (eager) : dès le conflit observé, transaction annulée
Obligation de barrière pour chaque lecture et chaque écriture

Détection au plus tard (lazy) : lorsque la transaction finie
Peut éviter certaines barrières
Risques d'incohérences car exécution de code sur des valeurs invalides

Mise en œuvre des mémoires transactionnelles

Trois façons de construire une mémoire transactionnelle

- ✓ *Mémoire transactionnelle matérielle (HTM)*
  Utilise le cache du processeur pour construire une stm retardée
  Détection au plus tard lorsque la ligne de cache est propagée
  + très rapide
  - taille très limitées, ne convient pas pour de grandes/longues transactions
- ✓ *Mémoire transactionnelle logicielle (STM)*
  Barrière en lecture et/ou écriture insérée par le compilateur
  - lent + taille quelconque
- ✓ *Mémoire transactionnelle hybride (HyTM)*
  Matérielle tant que le cache suffit, passe en logiciel sinon

Mise en œuvre des mémoires transactionnelles

Exemple de mise en œuvre : une STM délayée au plus tard avec verrou au commit

Principe :
- ✓ Chaque case mémoire possède un compteur
- ✓ Lecture : mémorise le compteur
- ✓ Ecriture : mémorise la case et le compteur
- ✓ Lors d'une mise à jour
  - vérifie que les compteurs n'ont pas changé
  - incrémente les compteurs des cases écrites

Cases mémoire	*Compteurs*
a	17
b	13
c	2
d	26
e	83

Mise en œuvre des mémoires transactionnelles

Fonctionnement

Mémoire global

	Valeur	*Compteurs*
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Cases

Copie local

a
b
c
d
e

```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

Mise en œuvre des mémoires transactionnelles

Fonctionnement

Mémoire global

a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Valeur *Cases* *Compteurs*

Copie local

a	31	17
b		13
c		
d		
e		

a. atomic {
b. a = a + b;
c. c = a - e;
d. b = c;
e. }

Mise en œuvre des mémoires transactionnelles

Fonctionnement

Mémoire global

a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Valeur *Cases* *Compteurs*

Copie local

a	31	17
b		13
c	23	2
d		
e		83

a. atomic {
b. a = a + b;
c. c = a - e;
d. b = c;
e. }

Mise en œuvre des mémoires transactionnelles

Fonctionnement

Mémoire global

a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Valeur *Cases* *Compteurs*

Copie local

a	31	17
b	23	13
c	23	2
d		
e		83

a. atomic {
b. a = a + b;
c. c = a - e;
d. b = c;
e. }

Mise en œuvre des mémoires transactionnelles

Fonctionnement

Mémoire global

a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Valeur *Cases* *Compteurs*

Copie local

a	31	17
b	23	13
c	23	2
d		
e		83

Mémoire global au commit

a	**31**	**18**
b	**23**	**14**
c	**23**	**3**
d	83	26
e	8	83

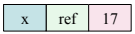
a. atomic {
b. a = a + b;
c. c = a - e;
d. b = c;
e. }

Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```

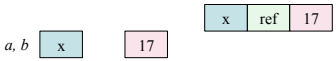


Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```

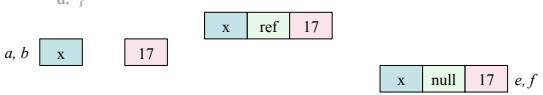


Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```

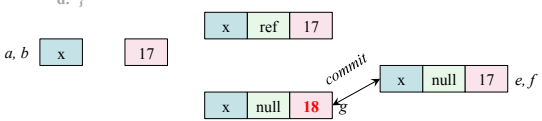


Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies

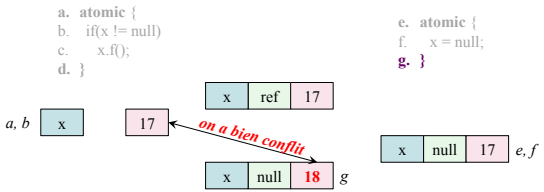
```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```



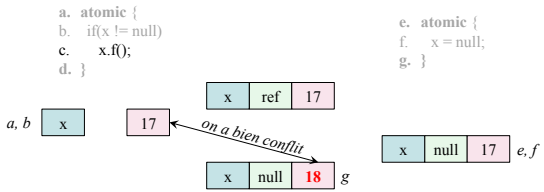
Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies



Mise en œuvre des mémoires transactionnelles

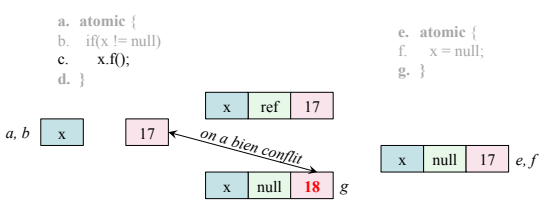
Problème classique : les transactions zombies



c: NullPointerException

Mise en œuvre des mémoires transactionnelles

Problème classique : les transactions zombies



c: NullPointerException

Solution intuitive : vérifier le compteur à chaque lecture

⇒ ne suffit pas!

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

x	4	17
y	5	83

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

a,b : t1 = 4 x → 17

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

a,b : t1 = 4 x → 17

x	217	17
y	4	83

f, g, h

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

a,b : t1 = 4 x → 17

commit

x	217	17
y	4	83

f, g, h

x	**217**	**18**
y	**4**	**84**

i

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

a,b : t1 = 4 x → 17

on a bien conflit

x	217	17
y	4	83

f, g, h

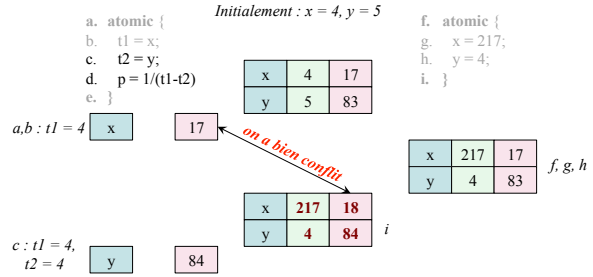
x	**217**	**18**
y	**4**	**84**

i

Mise en œuvre des mémoires transactionnelles

Problème classique les transactions zombies : la lecture du compteur ne suffit pas!

Invariant : x != y



Problème : pas de lecture de x avant le commit

d ⇒ division par zéro!

Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : une horloge globale

À chaque instant, le compteur d'une variable doit être inférieur à l'horloge

⇒ Assure que la valeur lue a été mise à jour avant le début de la transaction

Mise en œuvre des mémoires transactionnelles

Début transaction :

- ✓ Copie l'horloge globale dans horloge locale

A chaque lecture

- ✓ Vérifie que compteur strictement plus petit que horloge locale (sinon annule)
- ✓ Ajoute au readSet et au writeSet

A chaque écriture

- ✓ Ajoute au writeSet

Commit :

- ✓ Si existe un compteur du RS/WS >= horloge **locale**, annule transaction
- ✓ Pour toute variable écrite, met à jour son compteur à horloge **globale** et sa valeur
- ✓ Incrémente horloge globale

Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale

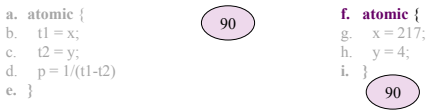
a. atomic {
b. t1 = x;
c. t2 = y;
d. p = 1/(t1-t2)
e. }

90

f. atomic {
g. x = 217;
h. y = 4;
i. }

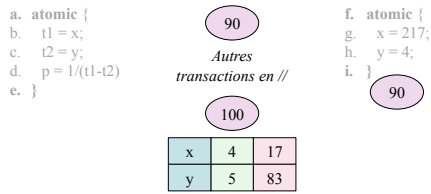
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



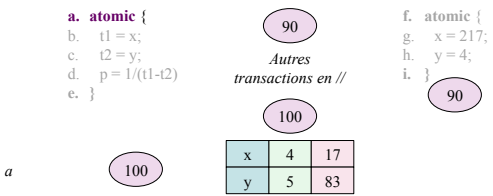
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



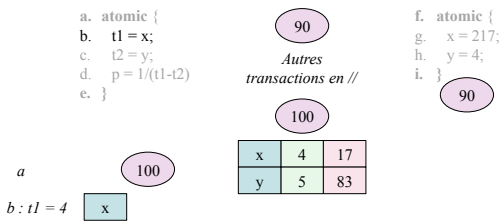
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



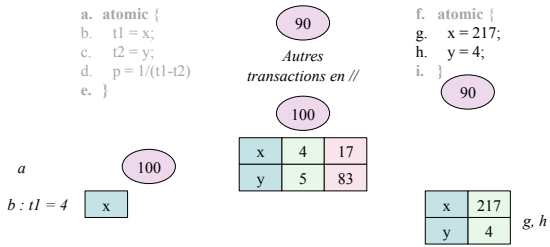
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



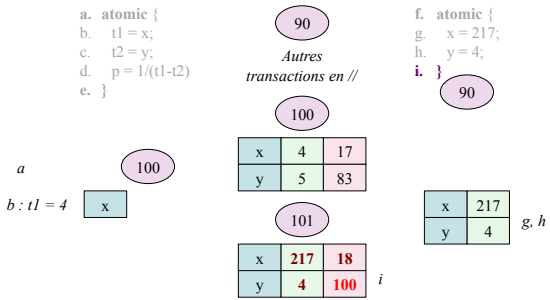
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



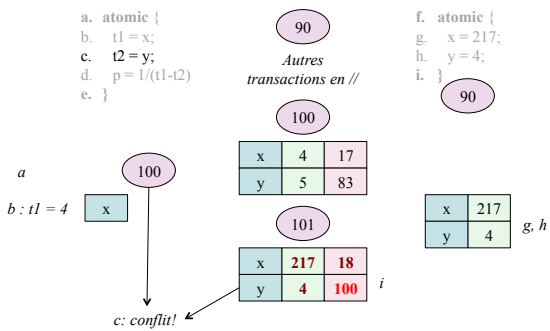
Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



Mise en œuvre des mémoires transactionnelles

Solution au problème des transactions zombies : l'horloge globale



Mise en œuvre des mémoires transactionnelles

Principe :

- ✓ La mémoire est un tableau de pointeurs vers des couples compteur/valeur
- ✓ Mise à jour systématique d'un pointeur complet et non d'un couple
- ✓ Un ramasse-miettes s'occupe de récupérer les vieux couples (jamais de libération explicite)

Implémentation d'une STM

```
class Value {
  int value;
  int counter;
}
```

Valeur = (contenu, compteur)

```
class Memory {
  static Value values[];
  static int   clock;
}
```

Mémoire = tableau De valeurs + horloge globale

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

Transaction = readSet + writeSet + horloge au début transaction

Implémentation d'une STM

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

```
void TX.begin() {
  clock    = Memory.clock;
  readSet  = new HashSet();
  writeSet = new HashSet();
}
```

Commencer une transaction : Copier l'horloge

Implémentation d'une STM

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

```
void TX.write(int idx, int value) {
  writeSet.put(idx, value);
}
```

Ecriture dans le tampon local

Implémentation d'une STM

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

```
int TX.read(int idx) {
  if(writeSet.contains(idx))
    return writeSet.get(idx);

  Value value = Memory.values[idx];

  if(value.counter >= clock)
    abort();

  readSet.add(idx);

  return res;
}
```

Si déjà une écriture locale, utilise La version locale

Si lecture valeur trop récente, abort (obligatoire à cause des transactions Zombies)

Implémentation d'une STM

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>       readSet;
  HashMap<int, int> writeSet;
  int                clock;
}
```

```
void TX.commit() {
  synchronized(Memory.values) { // commit serializés!
    for(int idx : readSet)
      if(Memory.values[idx].counter >= clock) abort();
    for(int idx : writeSet.keySet())
      if(Memory.values[idx].counter >= clock) abort();

    // ok, commit!
    for(Map<int, Value> entry : writeSet.entrySet()) {
      Value v = new Value(entry.getValue(), Memory.clock);
      Memory.values[entry.getKey()] = v;
    }

    Memory.clock++;
  }
}
```

Conflit lecteur/écrivain?

Conflit écrivain/écrivain?

Enregistre les write (mise à jour value ET counter atomiquement!)

Pour toute transaction qui commence Après cette ligne, Les écritures du dessus sont cohérentes

Implémentation d'une STM

Problème :

Deux transactions s'annulent l'une l'autre

Recommence ⇒ vont s'annuler de nouveau

Solution :

- ✓ Délayer le redémarrage de la transaction de façon aléatoire
- ✓ Si nouveau conflit, augmenter la plage aléatoire

```
int delay(int n) {
  Thread.sleep(1+(int)(n*Math.random()));
  return n < 512 ? n<<1 : n;
}
void doTransaction() {
  n = 16;
  try {              tx.begin(); …; tx.commit();     }
  catch(TXAbort e) { n = delay(n); doTransaction();  }
}
```


Mémoire Transactionnelle et Entrée/Sortie

```
atomic {
  if(x > 42)
    launchMissile();
}
```

Les entrées/sorties se prêtent très mal aux transactions

Cas spécifiques :

- ✓ Ecriture : difficile voir impossible à mettre dans une transaction
- ✓ Lecture : possibilité de jouer sur les tampons de réceptions

Solution possible :

- ✓ Vérifie que la transaction peut aboutir avant l'E/S
- ✓ Marque la transaction impossible à annuler
- ⇒ Complique beaucoup les algorithmes

  Nécessité de détecter les lecteurs des valeurs écrites (au lieu de l'inverse)

Conclusion

Simplification de la programmation concurrente

- ✓ Concept simple
- ✓ Plus de problème de deadlock, de famine
- ✓ Transactions composables
- ✓ Transactions imbriquées (non étudié ici)

Mise en œuvre difficile : les performances ne sont pas au rendez vous!

- ✓ Mémoire logicielle : moins bon que verrou [Rossback07]
- ✓ Mémoire matérielle : trop restreinte
- ✓ Mémoire hybride : passage matériel → logiciel très couteux

(Sur notre algo, 100 threads incrémentent 10'000 fois un compteur sur un bipro : 3,0s en STM sans delay, **0, 48s en STM** et **0,19s avec verrou**)