



T e c h n o l o g y

There is a better way

Sponsors
Platinum



Sponsors
Gold



Les secrets de la JVM pour les algorithmes à haute fréquence

Version : 1.2
ParisJUG – 8 Décembre 2015



- > Philippe PRADOS
- > Manager équipe « Architecture Réactive »



> +PhilippePrados



> @pprados



> in/pprados/fr



> BrownBagLunch.fr



> Conférences:

- + Solution Linux
- + MISC
- + Devoxx
- + PAUG
- + Scala IO
- + ...

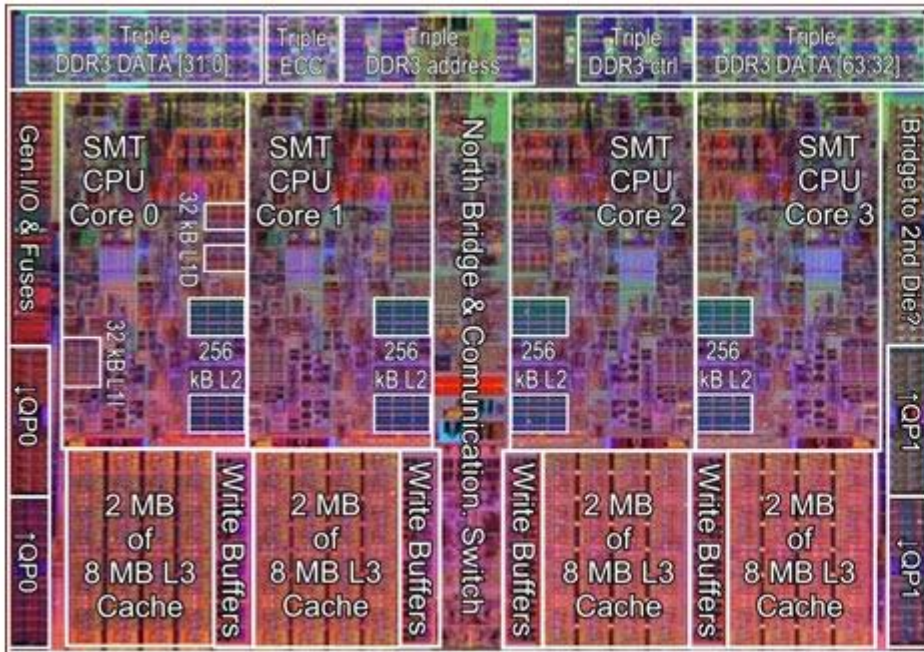


Attention, pour Barbus !!!

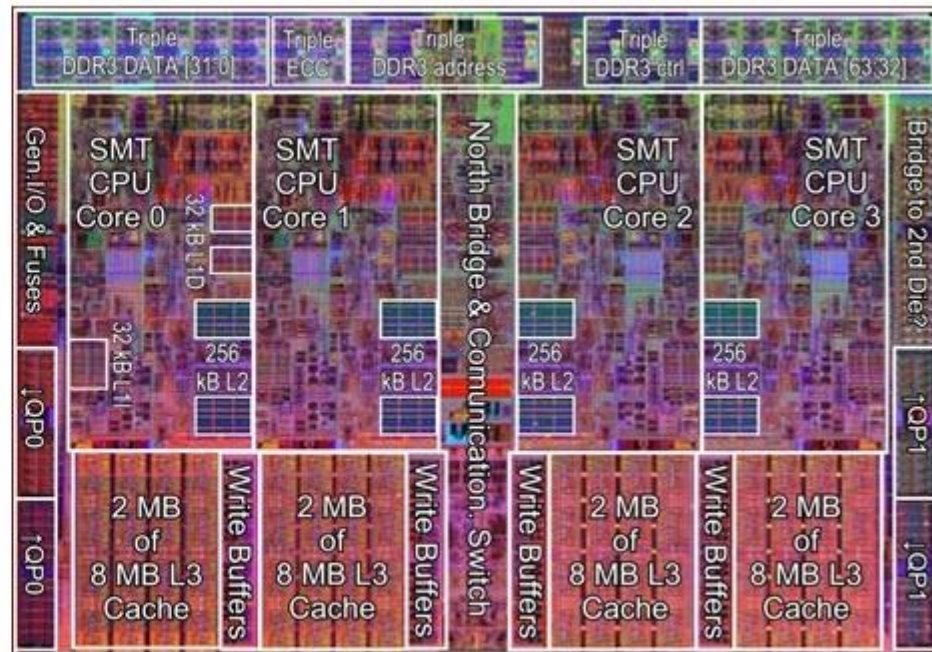


- > Un algorithme à haute fréquence doit **exploiter au mieux les architectures des processeurs** pour obtenir un maximum de performance

- > L'objectif est triple :
 - + Exploiter au maximum la mémoire
 - + Éviter les verrous. Il faut qu'à tout moment, l'algorithme puisse poursuivre son exécution
 - + Exploiter au maximum les caches des processeurs pour l'accès à la mémoire



Socket 1



Socket 2

- Un *node* est une machine physique possédant de la RAM.
- Un *socket* est un support où installer un processeur.
- Un *core* est un sous-ensemble du processeur physique, capable d'exécuter des traitements indépendamment des autres *cores* du *socket*.
- Les *virtuals cores* sont des traitements exécutés en parallèle par un seul *core*

- > **Softs-threads** : threads **simulant le multi-tâches** par un partage du temps d'un *core* par l'OS (Context switch périodique)
- > **Hard-threads** : threads **réellement multi-tâches**, distribué dans les *core* et *virtual core* des différents sockets

- > De nombreuses astuces sont utilisées dans la JVM
- > Des API spécifiques permettent de les utiliser
- > Nous avons identifié **14 secrets** dans 6 familles que nous souhaitons vous révéler

- 1 L'ASSEMBLEUR
- 2 EXPLOITATIONS DES CACHES
- 3 COMPARE AND SET
- 4 FRAMEWORK
- 5 AUTRES
- 6 CONTENEURS LOCK-FREE



L'assembleur

Le constat

- Les registres des processeurs sont les zones mémoire les plus rapides

1. Utiliser autant que possible les registres du processeur

Pseudo Assembleur

```
1 Cell[] as=cells;
```

```
Ldr r1, cells
Sto as,r1
```

```
2 if (as != null ) // Moins efficace
```

```
3 {
```

```
4 ...
```

```
5 }
```

```
Ldr r1,as
Cmp r1,0
```

- > Une écriture comme celle-ci est plus efficace et permet un meilleur contrôle des caches (on est certain que le code généré est optimisé)

```
1 Cell[] as;
```

```
2 if ((as = cells) != null ) // Tous reste dans les registres
```

```
3 {
```

```
4 ...
```

```
5 }
```

```
Ldr r1, cells
Sto as,r1
Cmp r1,0
```

- > Exemple JVM : La méthode **add()** de la classe **LongAdder**.

```
// LongAdder
86 if ((as = cells) != null || !casBase(b = base, b + x)) {
87   boolean uncontended = true;
88   if (as == null || (m = as.length - 1) < 0 ||
89     (a = as[getProbe() & m]) == null ||
90     !(uncontended = a.cas(v = a.value, v + x)))
91     longAccumulate(x, null, uncontended);
92 }
```

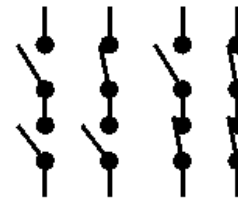
<http://goo.gl/QdihyC>

Le constat

- > Les instructions câblées sont plus efficaces que les micro-codes

Division binaire de 1010001b (81) par 11b (3) = 11011b (27)
reste 0b

$$\begin{array}{r} \overbrace{1010001} \\ - \quad 11 \\ \hline 100 \\ - \quad 11 \\ \hline 10 \\ 100 \\ - \quad 11 \\ \hline 11 \\ - \quad 11 \\ \hline 0 \end{array} \quad \begin{array}{r} 11 \\ \hline 11011 \end{array}$$



AND

$$\begin{array}{r} 0101 \\ \underline{0011} \\ 0001 \end{array}$$

2. Sélectionner les instructions assembleurs les plus rapides

- > La taille des tableaux de hashtable est un multiple de 2 (et non un nb premier)

```
// ConcurrentHashMap.java
692 private static final int tableSizeFor(int c) {
693     int n = c - 1;
694     n |= n >>> 1;
695     n |= n >>> 2;
696     n |= n >>> 4;
697     n |= n >>> 8;
698     n |= n >>> 16;
699     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY :
n + 1;
700 }
```

<http://goo.gl/Oon94e>

- > Permet d'utiliser les opérations binaires (&) à la places des modulo (%)

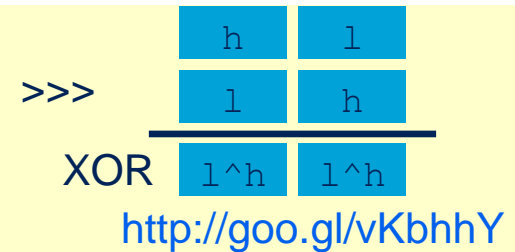
```
// Striped64.java
226 if ((a = as[(n - 1) & h]) == null) {
227 if (cellsBusy == 0) { // Try to attach new Cell
```

<http://goo.gl/WkrRRp>

- > Avec éventuellement une amélioration du hash (généralement pauvre)

```
// ConcurrentHashMap.java
```

```
684 static final int spread(int h) {
685     return (h ^ (h >>> 16)) & HASH_BITS;
686 }
```



- > Ou, plus sérieuse, via l'algo Jenkins qui modifie chaque bit de sortie suivant tous les bits d'entrée

```
// NonBlockingHashMap.java
```

```
112 private static final int hash(final Object key) {
113     int h = key.hashCode(); // The real hashCode call
114     // Spread bits to regularize both segment and index locations,
115     // using variant of single-word Wang/Jenkins hash.
116     h += (h << 15) ^ 0xffffcd7d;
117     h ^= (h >>> 10);
118     h += (h << 3);
119     h ^= (h >>> 6);
120     h += (h << 2) + (h << 14);
121     return h ^ (h >>> 16);
122 }
```

<http://goo.gl/dZaJaV>

Le constat

- Il existe des instructions assembleur spécialisées, plus efficaces qu'un code classique équivalent

3. Injection de code assembleur

- > La classe `Unsafe` est connue du JIT et génère l'équivalent assembleur

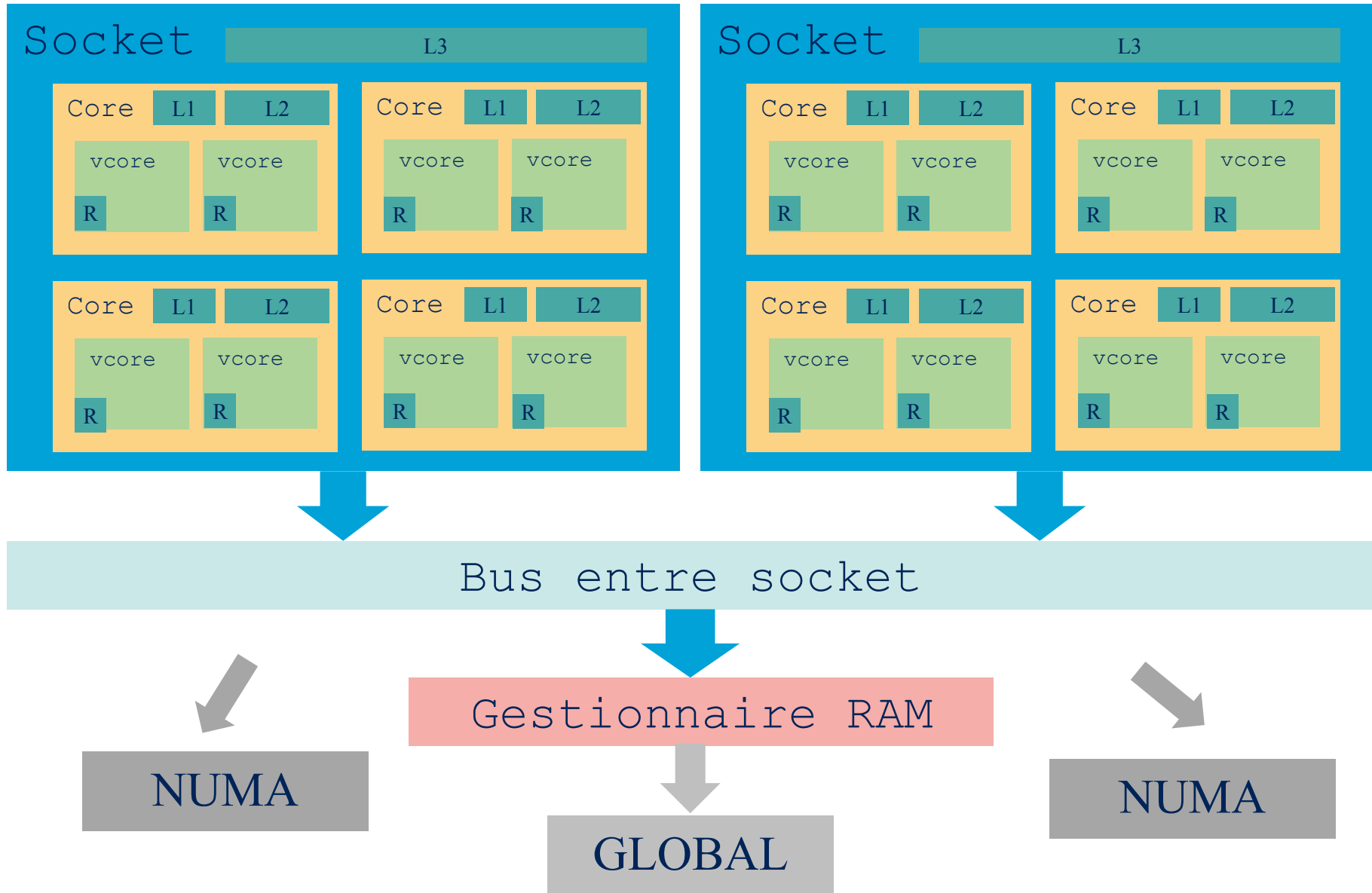
```
// atomic_linux_x86.inline.hpp
86 inline jint Atomic::cmpxchg(jint exchange_value, volatile jint* dest,
    jint compare_value) {
87     int mp = os::is_MP();
88     __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
89         : "=a" (exchange_value)
90         : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
91         : "cc", "memory");
92     return exchange_value;
93 }
```

<http://goo.gl/7hS4T7>

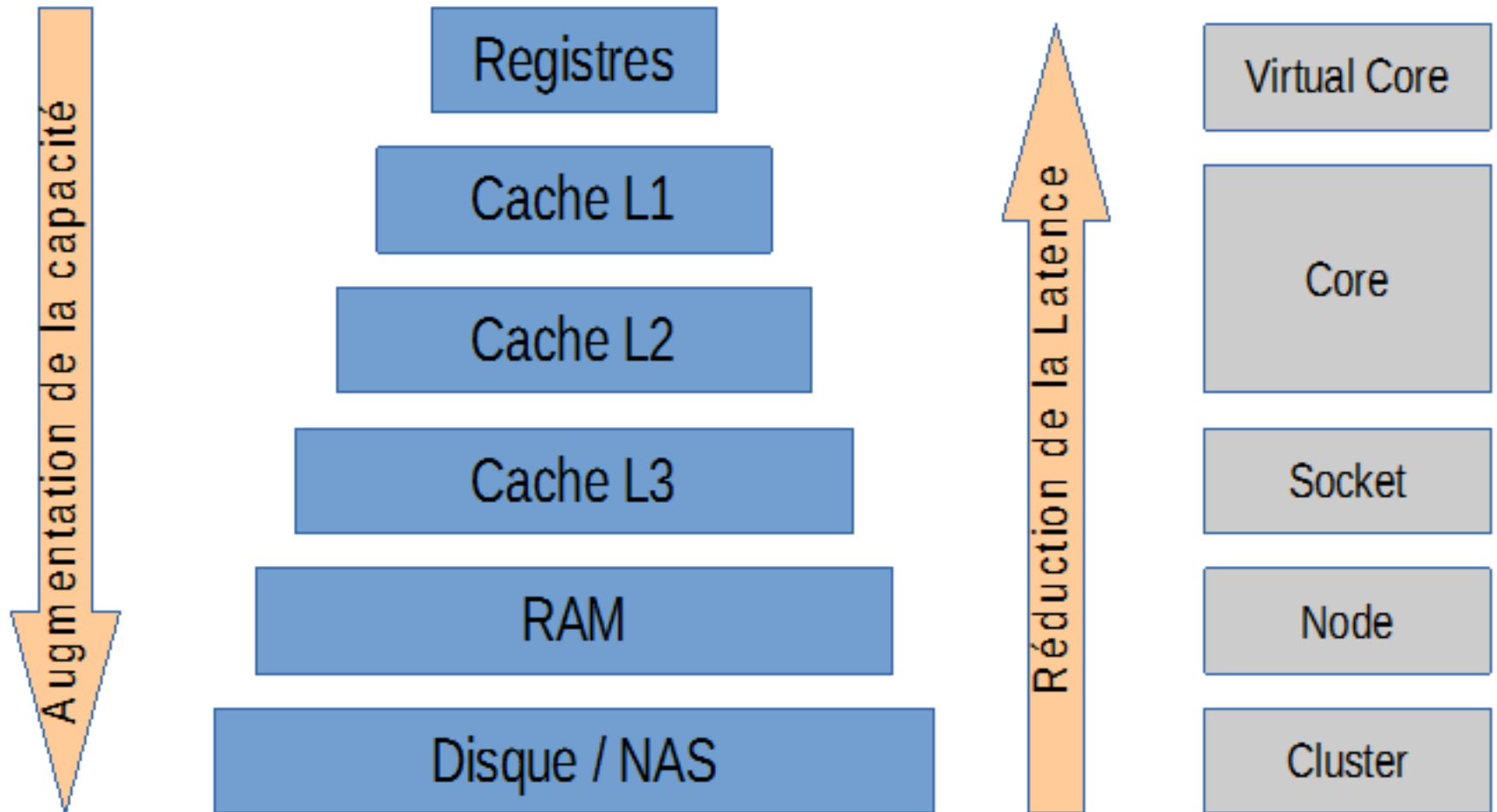
- > En x86, `AtomicLong.incrementAndGet()` est compilé en **LOCK INC**. `AtomicLong.getAndIncrement()` est compilé en **LOCK XADD**, etc.
- > Le préfixe **LOCK** devant une instruction assembleur permet de demander la *flush* des caches mémoire après l'exécution de l'instruction.
- > Les implémentations des classes du package `java.util.concurrent` utilisent abondamment cette approche.

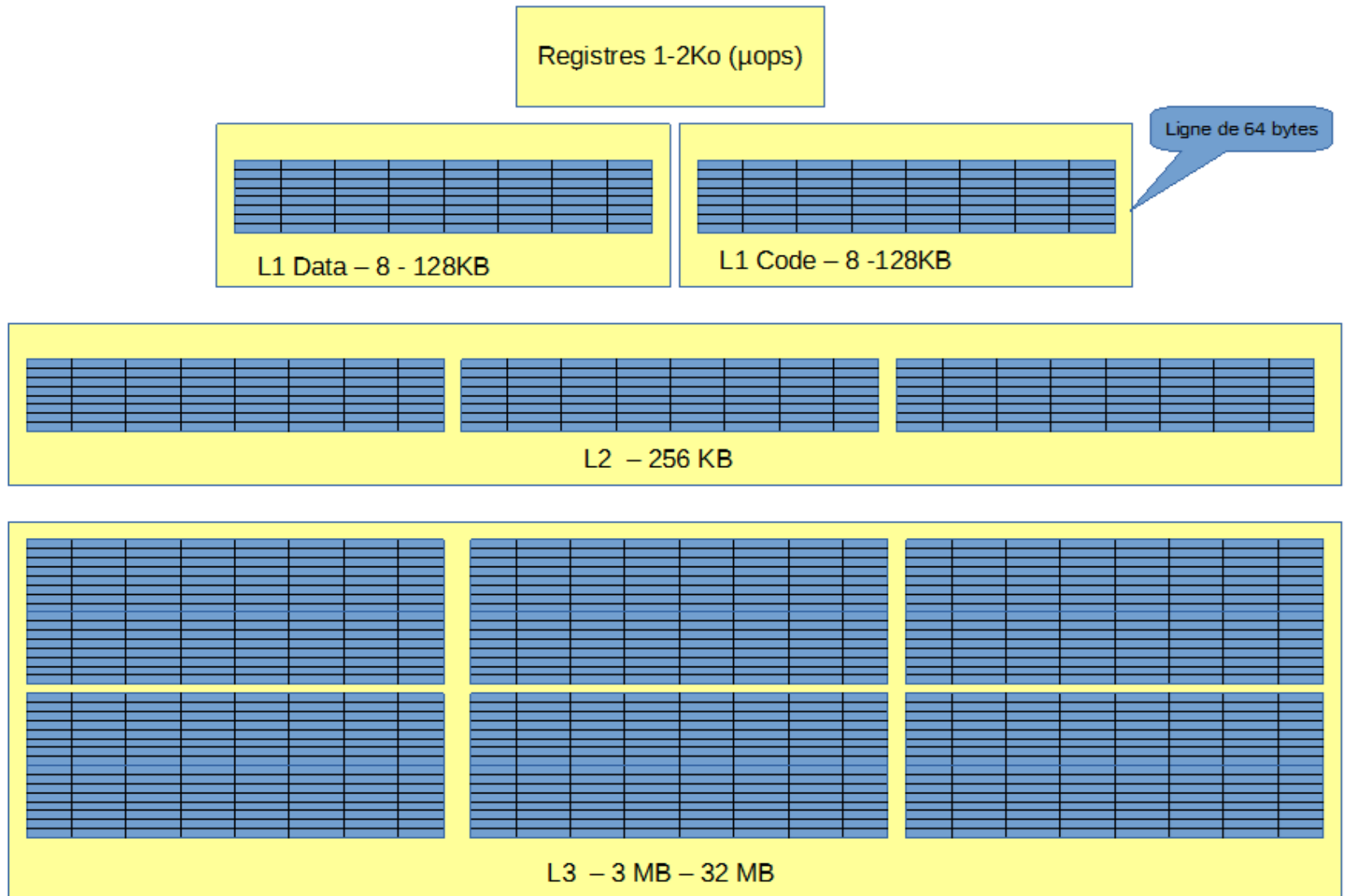


Exploitation des caches



Le constat





Niveau de cache	Taille	Temps d'accès en cycle	Concurrence	Technologie	Géré par
Registre	1/2 KB	μ	virtual core	Custom CMOS	Compilateur
Niveau 1	8KB – 128 KB	1	core	SRAM	Hardware
Niveau 2	256 KB	3	core	SRAM	Hardware
Niveau 3	3 MB – 32 MB	10 – 20	Socket	SRAM	Hardware
RAM	4 MB – 4 TB	200+	Chaque région séparément	Variable	OS

Le constat

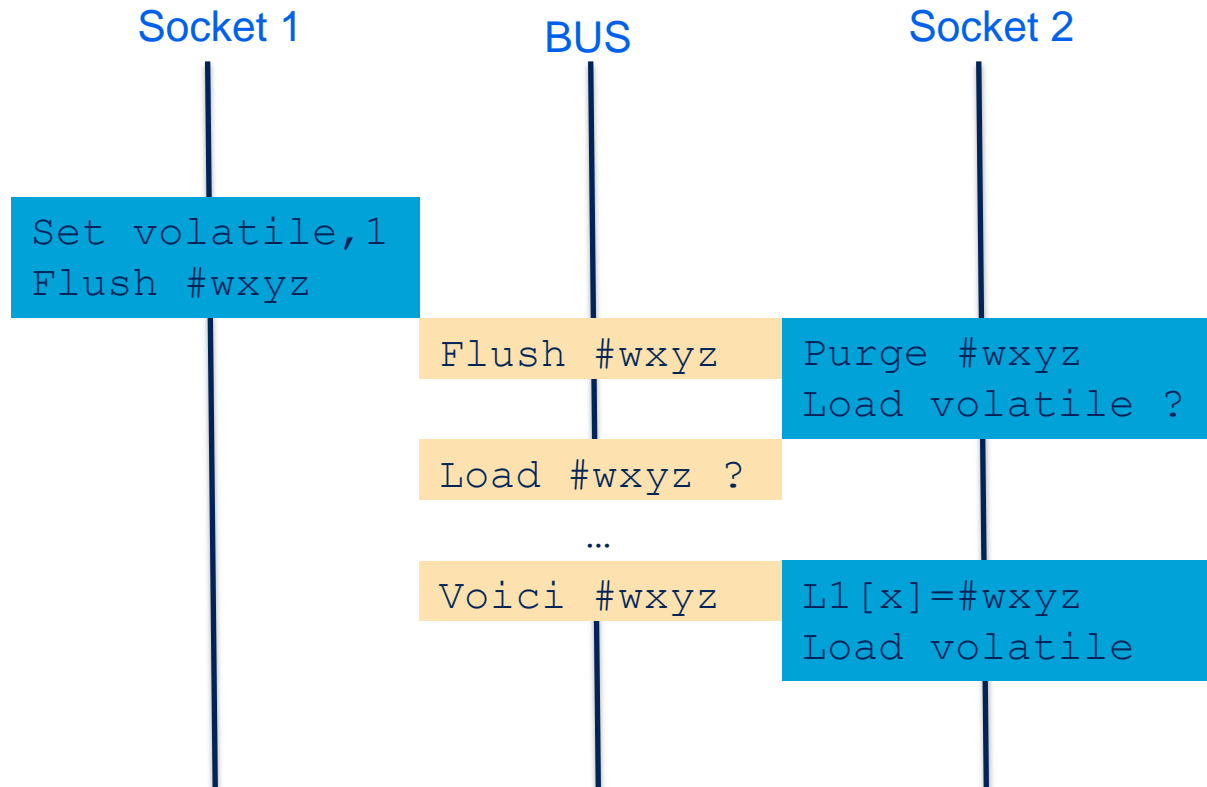
- > Un *flush* excessif des caches est préjudiciable aux performances

	Throughput (/ second)
lock cmpxchg	147,304,564
cmpxchg	458,940,006

Intel Core i7-3615QM at 2.30 GHz

4. Contrôler le vidage des caches des processeurs

- > Les variables **volatiles** entraînent une notification sur le bus pour invalider les caches des autres *sockets*
- > Garantie que la donnée sera en mémoire physique (la RAM)
- > Tous les threads de tous les *sockets* en sont informés
- > **Cela a un impact important sur les performances**



- > La classe Unsafe du JDK offre des méthodes pour contrôler les flushs des caches

- > Au début de la classe, des offsets vers les attributs sont obtenus

```
// ConcurrentHashMap.java
```

```
6287 static {
6288     try {
6289         U = sun.misc.Unsafe.getUnsafe();
6290         Class<?> k = ConcurrentHashMap.class;
6291         SIZECTL = U.objectFieldOffset
6292             (k.getDeclaredField("sizeCtl"));
6293         TRANSFERINDEX = U.objectFieldOffset
6294             (k.getDeclaredField("transferIndex"));
```

<http://goo.gl/5TxkdB>

- > Ensuite, suivant les besoins `getObjectVolatile(obj,offset)` et ses variantes sont utilisées
- > Exemple : La classe **ConcurrentLinkedQueue** utilise **Unsafe** pour initialiser un **Node**, sans flusher les caches dans le constructeur

```
// ConcurrentLinkedQueue.java
```

```
188     Node(E item) {
189         UNSAFE.putObject(this, itemOffset, item);
190     }
```

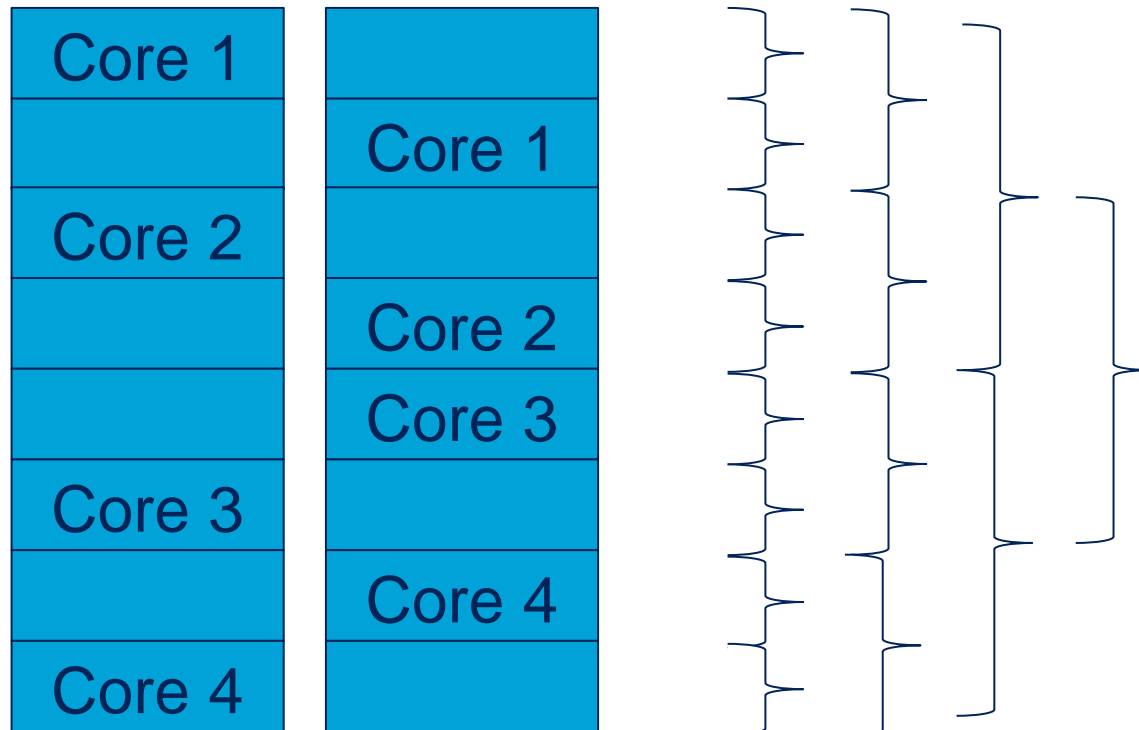
<http://goo.gl/xHdi3p>

Le constat

- > Les algorithmes peuvent être inefficaces vis-à-vis des caches

5. Utiliser des algos copains avec les caches

- > Il existe des algorithmes récursifs pour distribuer les traitements d'un ensemble de telle sorte que l'exploitation des caches soit optimale sans avoir à en connaître la taille



https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

Le constat

- Le cache L1 est mutualisé avec d'autres variables

6. Garder une ligne de cache L1 pour une seule variable

- > Faux-partage : lorsque deux variables atomiques **A** et **B** sont proches en mémoire
 - + L'écriture sur **B** invalide toute la zone mémoire du cache de niveau 1, et donc invalide également **A**
- > Les *cores* pensent à un partage des variables **A** ou **B** par plusieurs *cores*. Les caches sont alors synchronisés
- > Puisque **A** et **B** sont éjectés du cache : impact important sur les performances (de 1 à 10)



Faux
partage



Isolation L1

- > Saturer l'espace pour remplir une zone L1

```
// synchronizer.cpp
452 struct SharedGlobals {
453     // These are highly shared mostly-read variables.
454     // To avoid false-sharing they need to be the sole occupants of a $ line.
455     double padPrefix [8];
456     volatile int stwRandom;
457     volatile int stwCycle;
458
459     // Hot RW variables -- Sequester to avoid false-sharing
460     double padSuffix [16];
461     volatile int hcSequence;
462     double padFinal [8];
463 } ;
```

<http://goo.gl/tdwLqg>

- > Cela permet d'avoir une variable par *core*. Chaque *core* possède une ligne de cache différent des autres. Il n'y a plus de collision.

- > Java propose une annotation spéciale : **@sun.misc.Contended**
- > Dans Java 8, seule cinq classes utilisent cette annotation
 - + **Thread, Striped64, ConcurrentHashMap, Exchanger** et **ForkJoinPool**.

```
// LongAdder
```

```
120  @sun.misc.Contended static final class Cell {  
121      volatile long value;  
122      Cell(long x) { value = x; }  
123      final boolean cas(long cmp, long val) {  
124          return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val);  
125      }  
                                     http://goo.gl/RWfzpt
```

- > Chaque thread possède une variable unique dans une ligne de cache
- > Elle est incrémentée sans devoir invalider les caches des autres *cores*
- > Lors de la lecture, toutes les variables de tous les threads rencontrés sont ajoutées pour produire la somme totale

Le constat

- > Le cache L1 est mutualisé avec d'autres
et c'est une bonne chose !

7. Colocaliser des variables dans une ligne de cache L1

- > Rapprocher des variables dans la même ligne de cache (autant que possible)
- > En dehors des types primitifs, ce n'est pas facile avec Java, car les objets sont réparties dans la mémoire

```
// ConcurrentHashMap.java
```

```
619 static class Node<K,V> implements Map.Entry<K,V> {
```

```
620     final int hash;
```

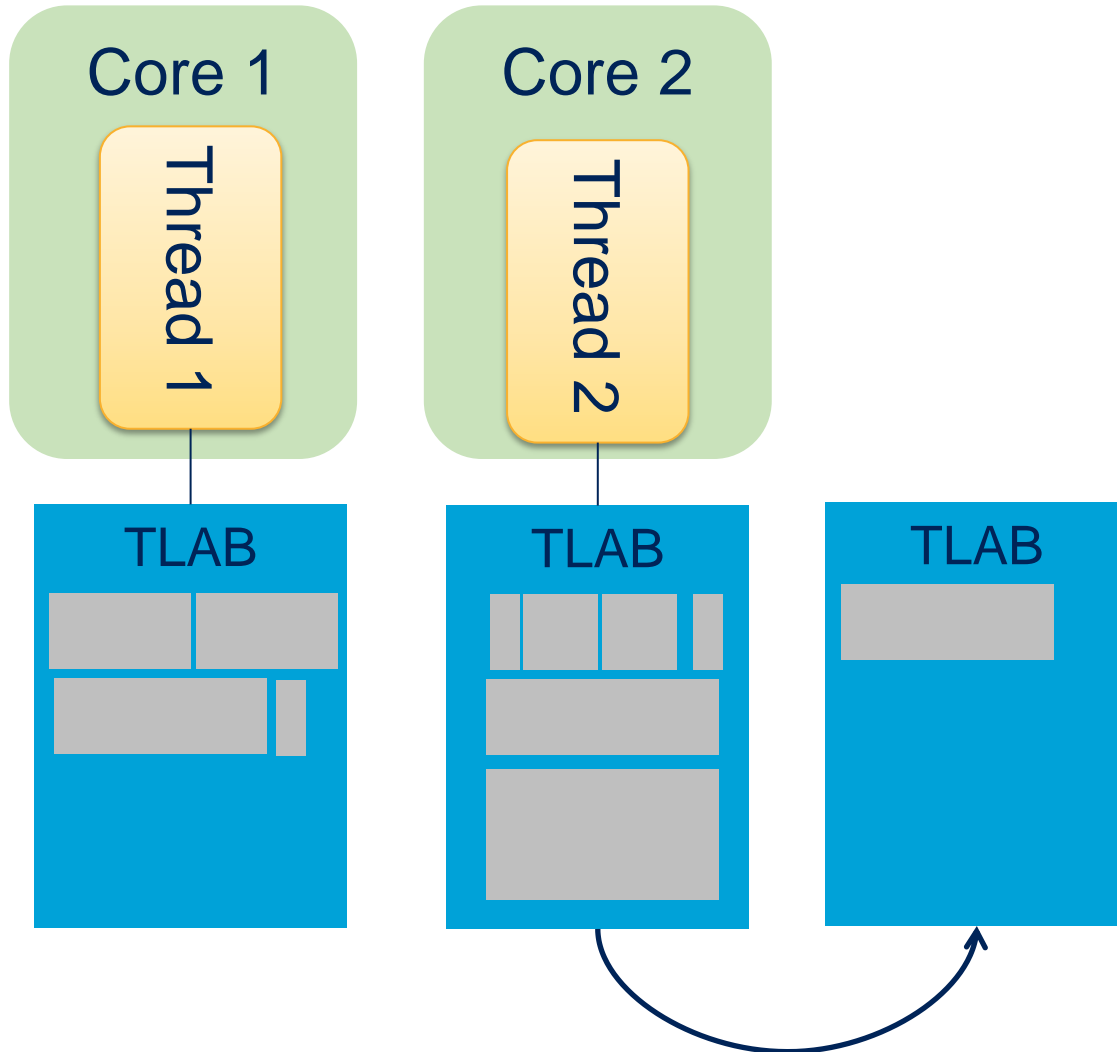
```
621     final K key;
```

```
622     volatile V val;
```

```
623     volatile Node<K,V> next;
```

<http://goo.gl/hqcmKf>

- > La valeur de hash est dupliquée localement dans le Node du HashMap
- > Évite de naviguer dans la mémoire



Allocateur mémoire Eden

- > Un bloc par thread
- > Allocation par incrément d'un offset
- > Ajout d'un bloc si nécessaire

TLAB: Thread Local Allocation Bloc



Compare and set

Le constat

- > La mémoire peut être modifiée par un autre *core* à tout moment

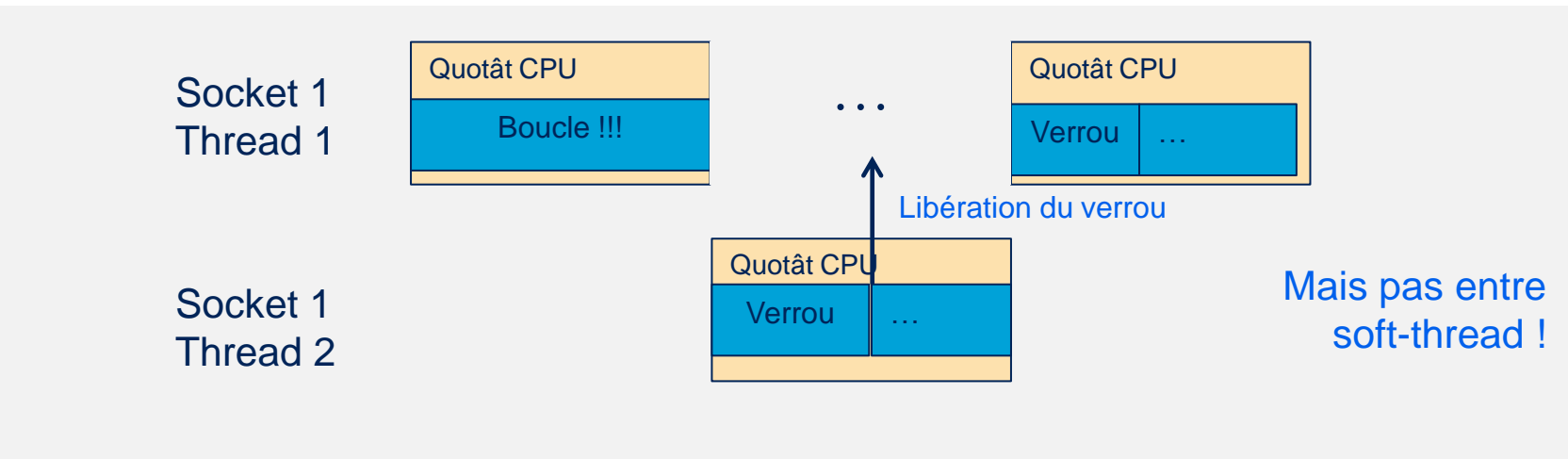
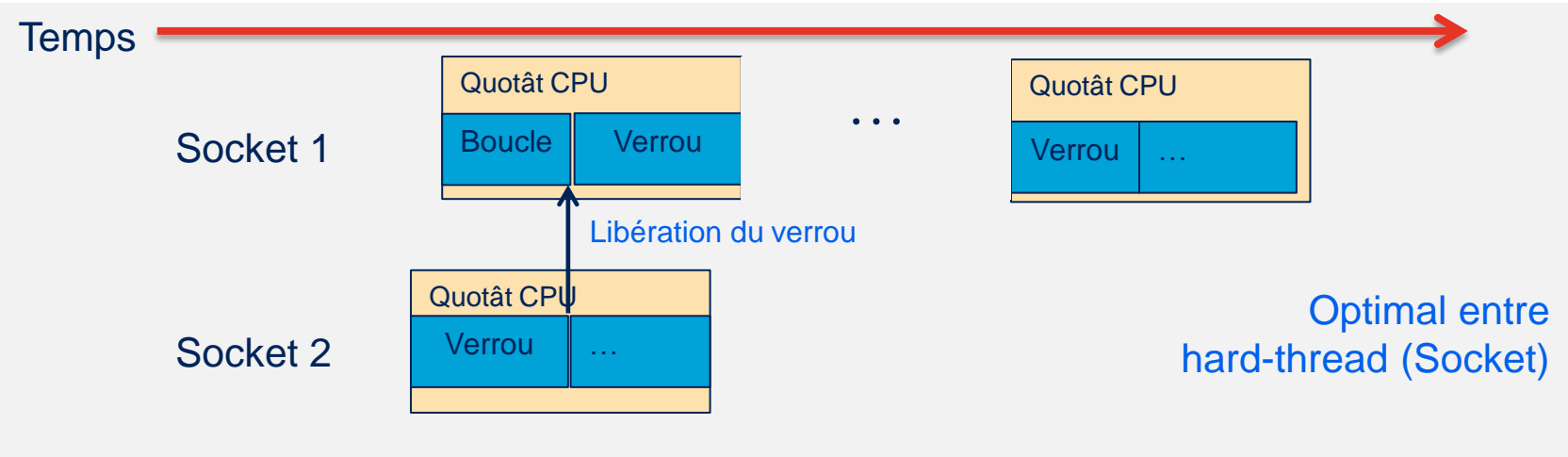
8. Boucler sans fin pour apporter des modifications

- > Boucler jusqu'à l'obtention de l'écriture attendue (SpinLock)

```
static AtomicBoolean semaphore=new AtomicBoolean(false) ;  
void lock()  
{ while ( !mutex.compareAndSet(  
    false, // Valeur attendue  
    true  // Valeur valorisée ));  
}
```

(**AtomicBoolean** utilise **Unsafe** en interne, donc les instructions assembleur)

8. Boucler sans fin pour apporter des modifications



- > Nécessite que l'OS répartisse les traitements sur plusieurs *cores*

```
void lock()
{
  int numberOfLoopBeforeTryWithASoftThread=40 ;
  int cnt=0 ;
  while ( !semaphore.compareAndSet(
    false, // Valeur attendue
    true // Valeur valorisée
  ))
    if (++cnt== numberOfLoopBeforeTryWithASoftThread)
      { cnt=0;
        Thread.yield(); // Force a context switch
      }
}
```

- > Hotspot utilise cela pour la gestion des synchronisations

```
// synchronizer.cpp
495 while (obj->mark() == markOopDesc::INFLATING()) {
496 // Beware: NakedYield() is advisory and has almost no effect on some
platforms
497 // so we periodically call Self->_ParkEvent->park(1).
498 // We use a mixed spin/yield/block mechanism.
499 if ((YieldThenBlock++) >= 16) {
500 Thread::current()->_ParkEvent->park(1);
501 } else {
502 os::NakedYield();
503 }
504 }
```

<http://goo.gl/kVDYVN>

- > Ces boucles peuvent entraîner un trafic important sur le bus de communication entre les *sockets*
- > Une succession de **Thread.Yield()** n'est pas efficace vis-à-vis du *context switch*
- > Il existe justement un paramètre à la JVM pour éviter cet excès (-XX:+DontYieldALot).

9. Boucler avec un Compare simple, puis un Compare-And-Set

- > Une première boucle **sans flush** du cache dans le core
- > Une deuxième **avec flush** du cache

```
void lock()
{
  for (;;)
  {
    { // Attend une invalidation du cache L1 par un autre core ou socket
      while (semaphore.get()) {} ;
      // Ici, tous peut arriver
      // Verifie que le sémaphore est toujours libéré, mais en CAS
      if (!semaphore.compareAndSet(false,true)) return ; // Toujours ok ?
    }
  }
}
```

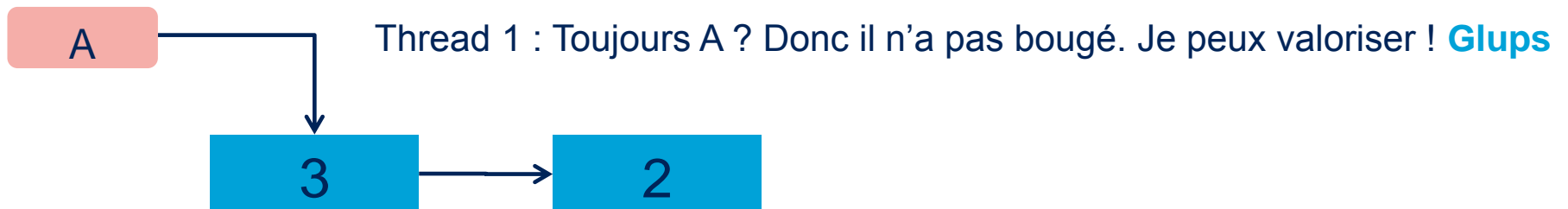
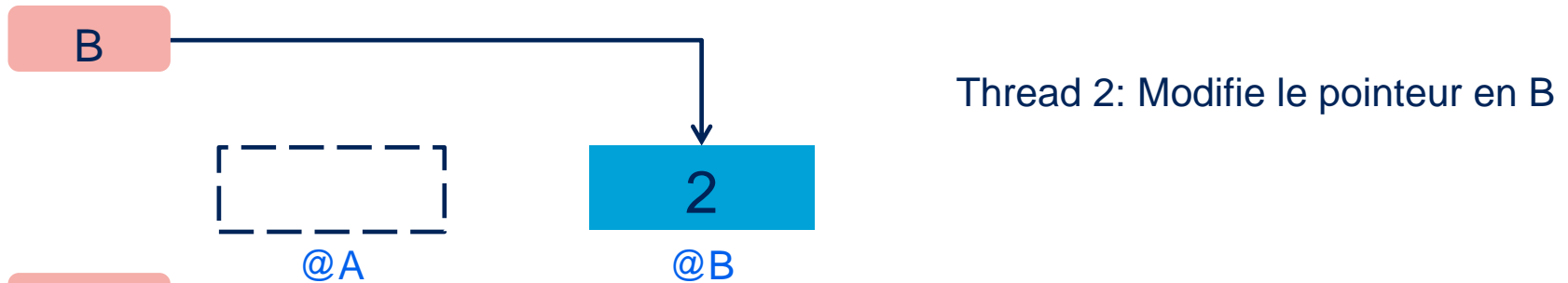
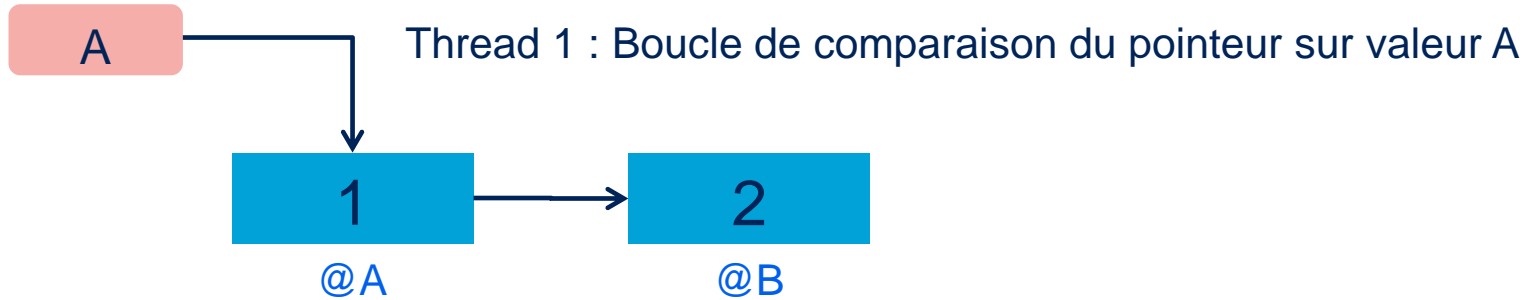
- > La première boucle consulte le cache local, sans envoyer d'invalidation du cache sur le bus
- > Le cache local est rafraîchi ssi un autre *socket* ou un autre *core* le demande
- > Ensuite, un CAS est appliqué
- > Entre temps, le sémaphore peut avoir été pris. Si c'est le cas, on recommence

10. Préparer spéculativement une modification et essayer de l'écrire en mémoire

- > Exemple : valoriser le pointeur **head** d'une liste chaînée

```
AtomicReference <Node> head;
void insert(int x)
{
    Node newNode=new Node(x);
    Node currentHead;
    do {
        newNode.next= (currentHead=head) // Ici
    } while (!currenthead.compareAndSet(currenthead,newNode)) // Là
}
```

- > Si un hard-thread ou un soft-thread intervient entre ici et là le **compareAndSet()** échoue
- > Un autre thread a modifié **head**
- > On recommence
- > Notez la valorisation spéculative de **next** à chaque itération



- > Il arrive lorsque la donnée **A** est modifiée en **B**, puis immédiatement en **A (A-B-A)**

- > Pour résoudre cela, plusieurs approches sont proposées :
 - + Utilisation d'un pool de **Node**
 - + Ajout d'un octet de plus s'il est possible de rester atomique (`long` à la place de `int` par exemple)
 - + Utilisation de quelques bits de l'adresse mémoire pour indiquer un numéro de version à chaque adresse (utilisé pour les threads java)
 - + ...

- > Plus radical :
 - + Instructions spécifiques de certains processeurs
 - o Présent dans : Alpha, PowerPC, MIPS, et ARM mais pas x86 !
- > La lecture de la mémoire (LL) signale que la zone de cache correspondante doit être traitée avec attention
- > Si le cache est écrit, l'instruction SC utilisée dans un CAS échoue
- > Ce n'est plus la valeur qui compte, mais la présence de la valeur dans le cache
- > Une chance, le langage Java n'est pas confronté à ce problème car il utilise un ramasse miettes

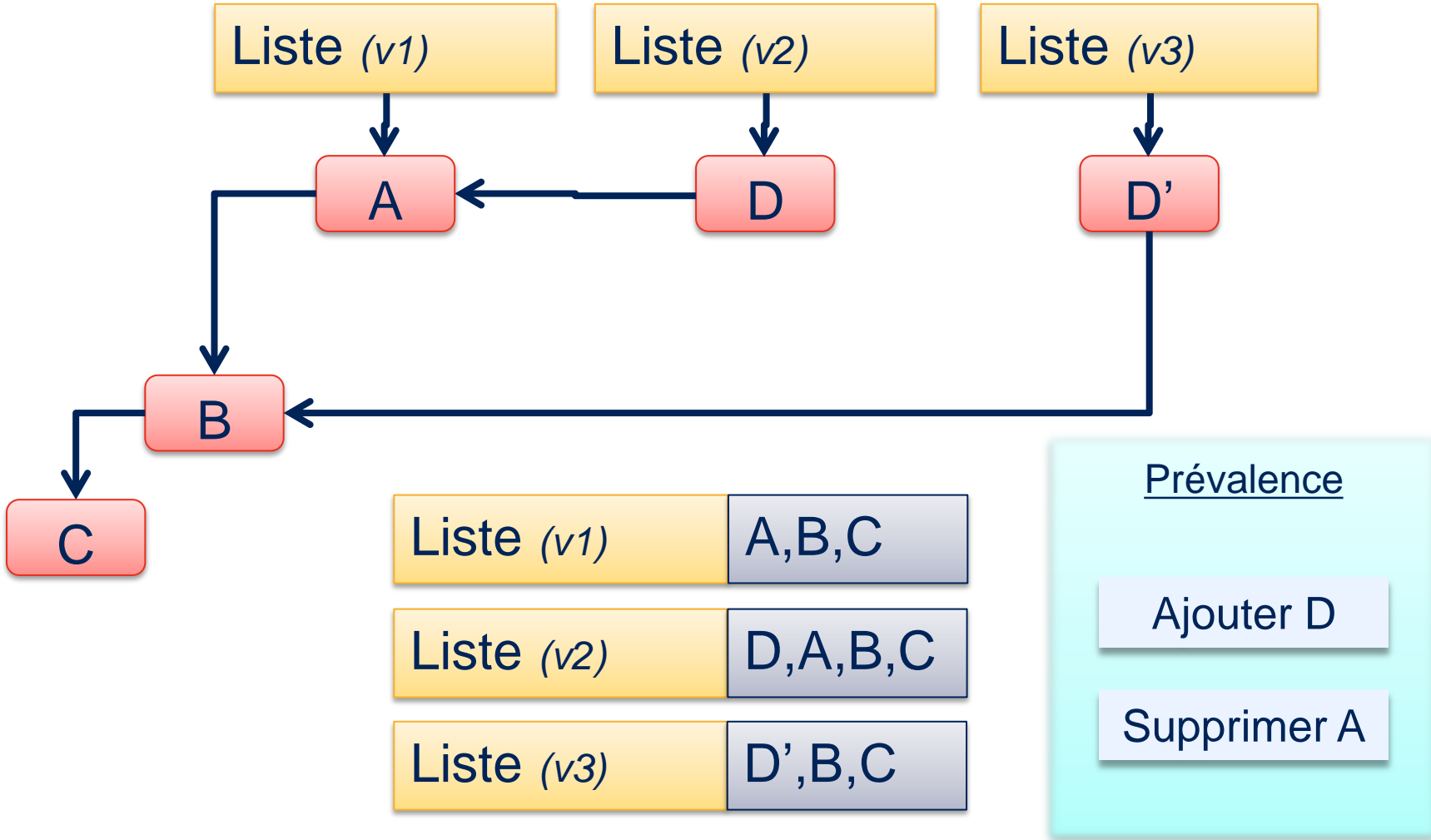


Framework

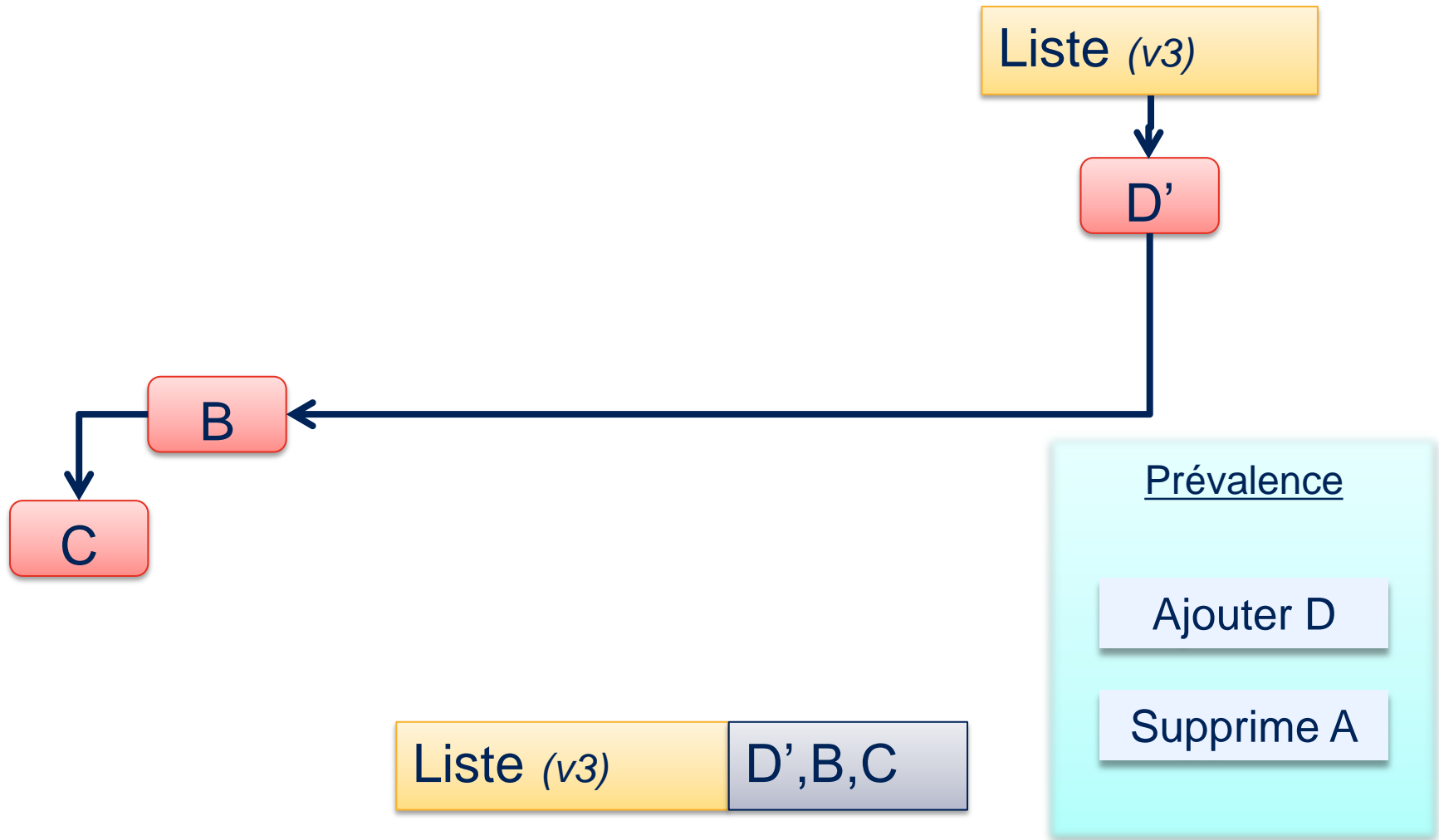
Le constat

- > Ne plus modifier les données évite l'éviction des caches

11. Conteneur immuable

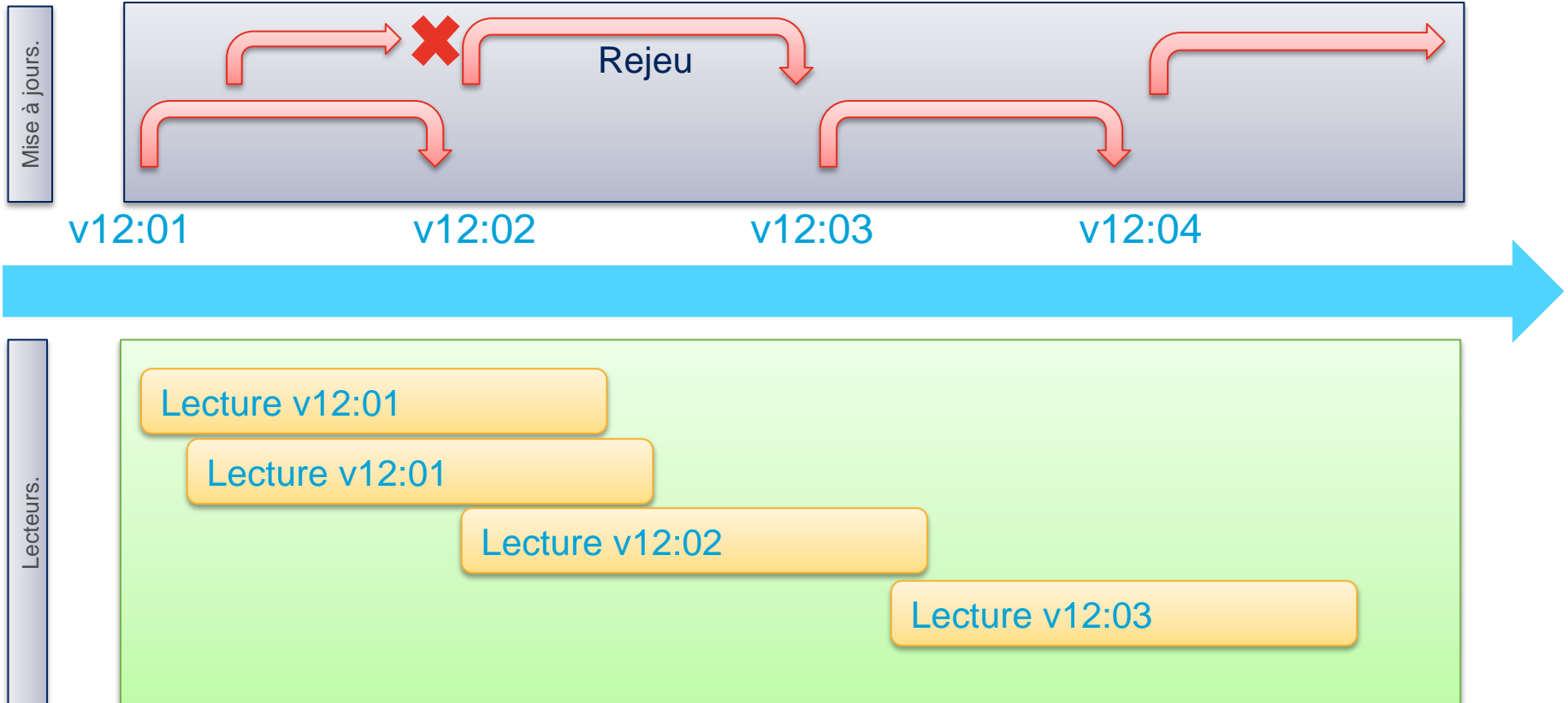


11. Conteneur immuable



12. Transaction en mémoire

- + Les modifications s'effectuent dans des « transactions »
- + En cas de collisions, les modifications sont rejouées
- + Les frameworks utilisent les approches « haute-fréquence »



- > Généralisation du modèle *Read-Modify-Write*
- > En cas d'échec de l'écriture, il faut rollbacker
 - + Via des call-backs
 - + Ou des indirections de pointeur, permettant de reconstruire l'état précédent
- > Librairie en Scala ou Closure
- > En cas de forte contention, un délai aléatoire est ajouté avant de tenter à nouveau



Autres

13. Affinité des threads avec les sockets ou les cores

- > Java **ne permet pas** de jouer avec les affinités des threads
- > Il existe des librairies pour cela (OpenHFT), compatibles Linux et Windows

```
private static final ExecutorService ES =  
    Executors.newFixedThreadPool(  
        Runtime.getRuntime().availableProcessors(),  
        new AffinityThreadFactory("bg",  
            DIFFERENT_SOCKET,DIFFERENT_CORE, ANY));
```

- > Des scénarios ajuste le kernel linux pour le limiter à 1 core et laisser l'intégralité des autres à l'application. L'amélioration des performance est très importante !

- > **LongAdder** de Java8 utilise un identifiant caché par thread dans une ligne de cache L1

```
// Striped64.java
```

```
185 static final int getProbe() {
```

```
186     return UNSAFE.getInt(Thread.currentThread(), PROBE);
```

```
187 }
```

<http://goo.gl/YGEQAD>

- > Donc : Une ligne de cache L1 par thread !
- > Il pourrait bénéficier d'une variable dans un cache L1 par *core* (et non par thread)
- > *Attention à la migration de threads entre cores*
 - + *Le scheduler Unix ne migre pas si un core travaille (pas d'I/O)*

14. Utiliser synchronize

- > C'est maintenant rapide pour un traitement sans IO !
- > Mais le verrouillage doit être très court et n'utiliser que de la CPU

- > Avec Hotspot 8, la JVM utilise des stratégies efficaces pour gérer les verrous
 - + L'en-tête de chaque objet utilise des bits pour associer le threads au verrous
 - + Les threads sont alloués en bas de la mémoire de la JVM
 - + Cela laisse des bits disponibles pour une utilisation d'un CAS (*Compare-and-set*) pour l'état et l'association avec le thread

- > Plusieurs stratégies sont utilisées. De la plus rapide à la plus lente, au fur et à mesure des échecs successifs



Conteneurs lock-free

- > Les conteneurs de Java de type **Blocking*** ou **Concurrent*** sont implémentés avec les techniques que nous avons évoqué
- > L'implémentation de **ConcurrentHashMap** de Java8 est une ré-implémentation complète de la version de Java7
 - + N'a pas besoin de verrous tant qu'il n'y a pas de collision sur la valeur de hash
 - + Sinon bloque l'accès
- > D'autres librairies sont plus efficaces
 - + High-Scale-Java
 - + LMAXCollection
 - + SnapTree pour un arbre balancé sans verrous
 - + Concurrency Freaks avec différentes améliorations de **ReadWriteLock**



Pour conclure

- > À tous les niveaux, nous retrouvons ces algorithmes (futex de Linux, ...)
- > Les nouveautés annoncées pour Java9 vont également dans le même sens
 - + co-localiser les structures et les conteneurs de types primitifs avec les « Value classes » pour mutualiser les données dans la même ligne de cache
 - + Mais Java9 envisage de supprimer Unsafe !!!
 - ◉ JEP 193: Variable Handles
 - ◉ *Oracle's plan is for Unsafe not to be removed, but rather hidden through the new modules system, to be available in Java 9*

- > Ces approches sont très subtiles
mais **oh combien amusantes !!!**
- > C'est un terrain de recherche encore à défricher !
- > Proposez vos implémentation ASAP !



<https://www.linkedin.com/jobs2/view/91749910>

Vous croyez que *les technologies* changent le monde ?

Nous aussi ! Rejoignez-nous !

recrutement@octo.com

Pourquoi choisir OCTO !

1^{er} Cabinet d'Architecture

MÉTHODES DE DÉVELOPPEMENT AGILE

Great place to work®

L'ENGAGEMENT

LE PRAGMATISME **NOS VALEURS** FORFAITAIRE

DE BEAUX SUCCÈS DEPUIS 13 ANS **MULTI TECHNO**

L'EXCELLENCE ET LE PLAISIR

Partage de nos savoirs

Questions ?

