Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Transactional Memory

Συστήματα Παράλληλης Επεξεργασίας

9ο Εξάμηνο

National Technical University of Athens
CSLab

Τακτικές Συγχρονισμού

- Κλειδώματα (locks)
  - Coarse-grain
  - Fine-grain
- Non-blocking
  - Lock-free
  - Wait-free
- **Transactional Memory**

Συγχρονισμός

Γιατί δε μας αρέσουν τα locks ...

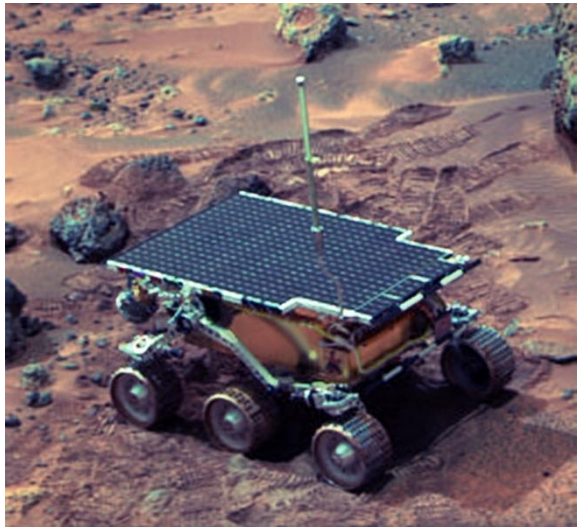
- Δυσκολία προγραμματισμού
  - εύκολο να κάνουμε coarse-grain υλοποιήσεις
  - για να εκμεταλλευτούμε όμως τον παραλληλισμό πρέπει να κάνουμε fine-grain υλοποιήσεις ...
  - ... και εκεί ξεκινάει ο εφιάλτης

- Convoying
  - ένα νήμα που κρατάει ένα lock γίνεται schedule out
  - τα υπόλοιπα νήματα περιμένουν την απελευθέρωση του lock
  - ακόμα και μετά την απελευθέρωση του lock υπάρχει μία ουρά από νήματα που περιμένουν να πάρουν το lock

- Αντιστροφή προτεραιοτήτων (Priority inversion)
  - ένα νήμα Τ1 υψηλής προτεραιότητας περιμένει να ελευθερωθεί κάποιο lock
  - ένα άλλο νήμα Τ2 χαμηλότερης προτεραιότητας εκτελείται πριν το Τ1

Mars Pathfinder (1997)

- αποστολή της NASA για αποστολή του εξερευνητικού οχήματος *Sojourner* στον Άρη
- λίγες μέρες μετά την προσεδάφιση στον Άρη το λογισμικού του συστήματος κάνει επανεκκινήσεις
- **Γιατί;** Priority inversion
- **Concurrent programming is hard:**



Πηγή: wikipedia

"After the failure, **JPL engineers spent hours and hours** running the system on the exact spacecraft replica in their lab with tracing turned on, **attempting to replicate the precise conditions** under which they believed that the reset occurred. Early in the morning, after all but one engineer had gone home, the engineer finally reproduced a system reset on the replica. Analysis of the trace revealed the priority inversion."

"In less that 18 hours we were able to cause the problem to occur." - Glenn E Reeves, Leader of the Pathfinder software team

Πηγή:http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

- Composability

```
void transfer (account A, account B,
               int amount)
{
        lock(A);
        lock(B);
        withdraw(A, amount);
        deposit(B, amount);
        unlock(B);
        unlock(A);
}
```

Νήμα 1: `transfer(A, B, 10);`

Νήμα 2: `transfer(B, A, 20);`

- Η σύνθεση κλειδωμάτων είναι δύσκολη.
  - Χρειάζεται καθολική πολιτική για το κλείδωμα
    - δεν μπορεί πάντα να αποφασιστεί εκ των προτέρων
- **Fine-grain locking:** εκμετάλλευση παραλληλισμού, καλή επίδοση και κλιμακώσιμο, αλλά δυσκολία προγραμματισμού.

- Composability

```
void transfer (account A, account B,
               int amount)
{
        lock(bank);
        withdraw(A, amount);
        deposit(B, amount);
        unlock(bank);
}
```

Νήμα 1:
`transfer(A, B, 10);`

`lock(bank);`
`... critical section...`

Νήμα 2:
`transfer(C, D, 20);`

`lock(bank);`
`... stall ...`

No Concurrency!

- Η σύνθεση κλειδωμάτων είναι δύσκολη.
  - Χρειάζεται καθολική πολιτική για το κλείδωμα
    - δεν μπορεί πάντα να αποφασιστεί εκ των προτέρων
- **Fine-grain locking:** εκμετάλλευση παραλληλισμού, καλή επίδοση και κλιμακώσιμο, αλλά δυσκολία προγραμματισμού.
- **Coarse-grain locking:** δεν υπάρχει παραλληλία, χαμηλή επίδοση

Παράδειγμα: Java 1.4 HashMap

- Map: key → value

```
public Object get(Object key)  {
   int idx = hash(key);              // Compute hash
   HashEntry e = buckets[idx];       // to find bucket
   while (e != null)  {              // Find element in bucket
      if (equals(key, e.key))
            return e.value;
      e = e.next;
    }
   return null;
 }
```

(-) δεν είναι thread-safe

Synchronized HashMap

- Η λύση της Java 1.4: synchronized layer
  - ρητό, coarse-grain locking από τον προγραμματιστή

```
public Object get(Object key) {
    synchronized (mutex) {  // mutex guards all accesses to map m
        return m.get(key);
    }
}
```

- Coarse-grain synchronized HashMap

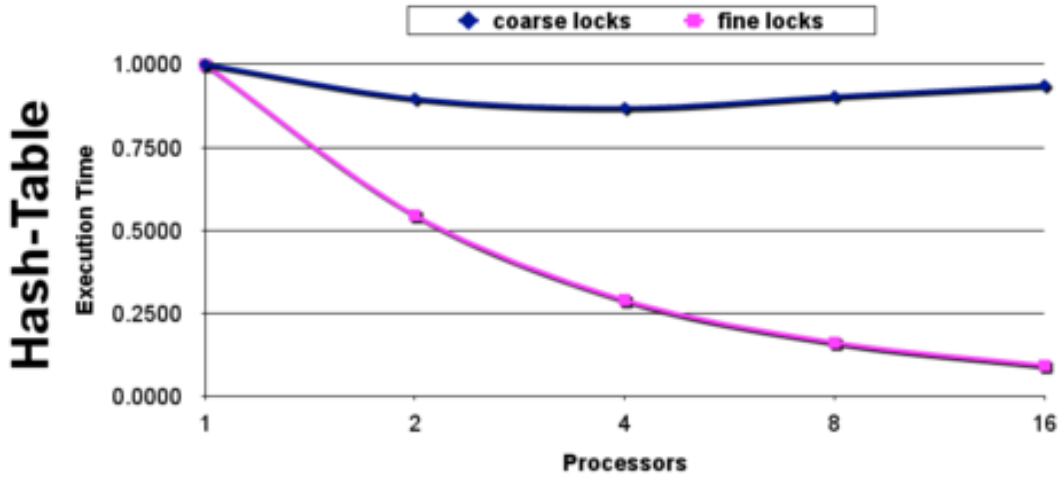
  (+) thread-safe, εύκολο στον προγραμματισμό

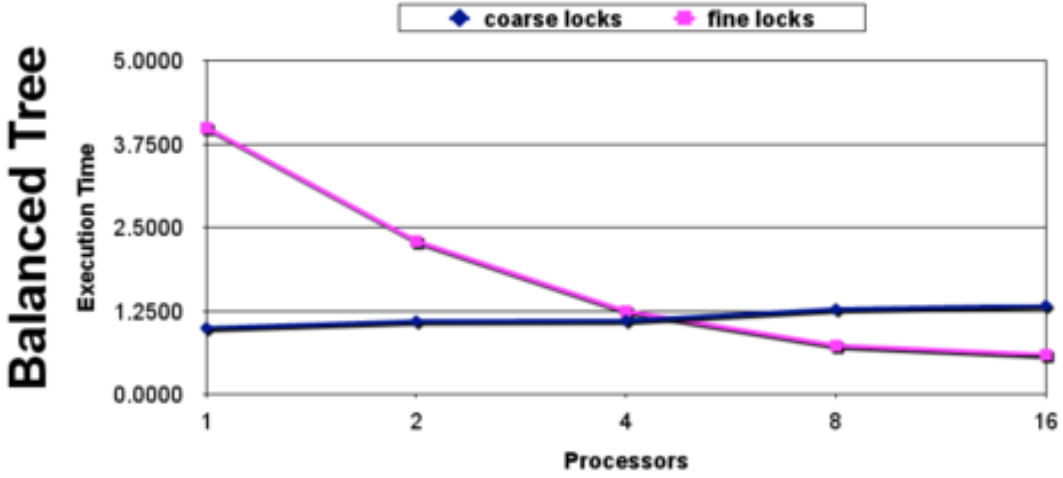
  (-) περιορίζει τον (όποιον) ταυτοχρονισμό, χαμηλή κλιμακωσιμότητα
    - μόνο ένα thread μπορεί να επεξεργάζεται το HashMap κάθε φορά
- Fine-grain locking HashMap (Java 5)
  - ένα lock ανά bucket

  (+) καλή επίδοση και κλιμακωσιμότητα

  (-) δυσκολία προγραμματισμού

Επίδοση: Locks





Transactional Memory (TM)

- Ιδανικά θα θέλαμε:
  - προγραμματιστική ευκολία αντίστοιχη με coarse-grain locking
  - απόδοση συγκρίσιμη με fine-grain locking

- Transactional memory
  - ο χρήστης σημειώνει κομμάτια κώδικα τα οποία πρέπει να εκτελεστούν ατομικά (transactions)
  - το TM σύστημα είναι υπεύθυνο για την σωστή εκτέλεση

- Μεταφέρουμε την πολυπλοκότητα του fine-grain συγχρονισμού στο TM σύστημα.

Προγραμματισμός με TM

```
void transfer (account A, account B,
               int amount)
{
        lock(A);
        lock(B);
        withdraw(A, amount);
        deposit(B, amount);
        unlock(B);
        unlock(A);
}
```

```
void transfer (account A, account B,
               int amount)
{
        tx_begin();
        withdraw(A, amount);
        deposit(B, amount);
        tx_end();
}
```

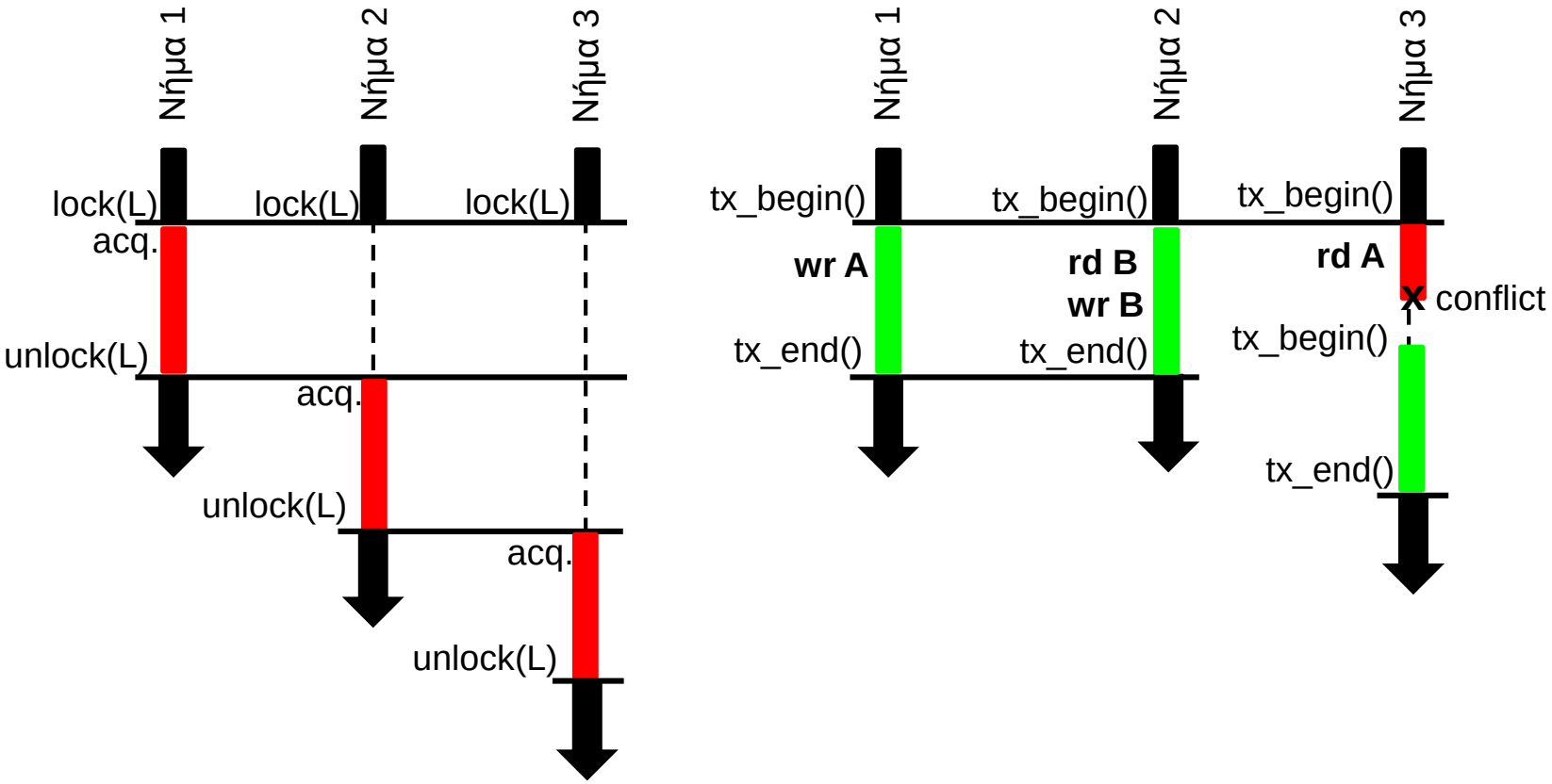
- Προστακτικός vs. Δηλωτικός προγραμματισμός
- Περιγραφή συγχρονισμού σε υψηλό επίπεδο
  - ο προγραμματιστής λέει τι, όχι πώς
- Το υποκείμενο σύστημα υλοποιεί το συγχρονισμό
  - εξασφαλίζει ατομικότητα, απομόνωση & σειριοποίηση
- Επίδοση;
  - εξαρτάται από το κρίσιμο τμήμα και την υλοποίηση του TM

Transactions

- Memory transaction
  - μία **ατομική** και **απομονωμένη** ακολουθία λειτουργιών μνήμης
  - εμπνευσμένη από τις δοσοληψίες στις Βάσεις Δεδομένων
- Ατομικότητα (Atomicity)
  - θα γίνουν είτε όλες οι εγγραφές στη μνήμη ή καμία (all or nothing)
  - **commit**, όλες οι εγγραφές στη μνήμη
  - **abort**, καμία εγγραφή στη μνήμη
- Απομόνωση (Isolation)
  - κανένα άλλο τμήμα κώδικα δεν μπορεί να δεί τις εγγραφές ενός transaction πριν το commit
- Σειριοποίηση (Serializability)
  - τα αποτελέσματα των transactions είναι συνεπή (ίδια με αυτά της σειριακής/σειριοποιημένης εκτέλεσης)
  - τα transactions φαίνονται ότι κάνουν commit σειριακά
  - η συγκεκριμένη σειρά ωστόσο δεν είναι εγγυημένη

Locks vs. TM

3 νήματα εκτελούν ένα κρίσιμο τμήμα

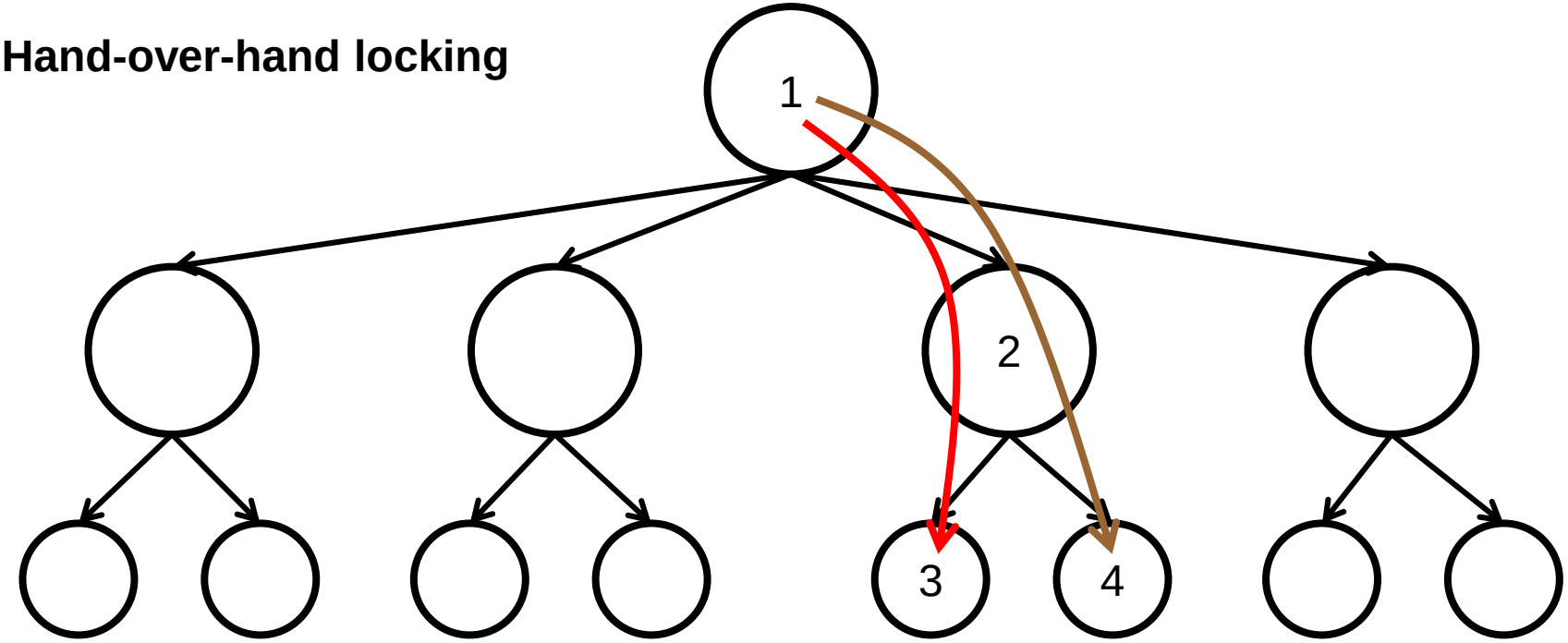


- Αναμενόμενο κέρδος σε περίπτωση μη-conflict
- Επιβράδυνση σε conflicts (R-W ή W-W) λόγω χαμένης δουλείας

Tree update από 2 νήματα

- Στόχος: τροποποίηση κόμβων 3 και 4 με thread-safe τρόπο

Hand-over-hand locking



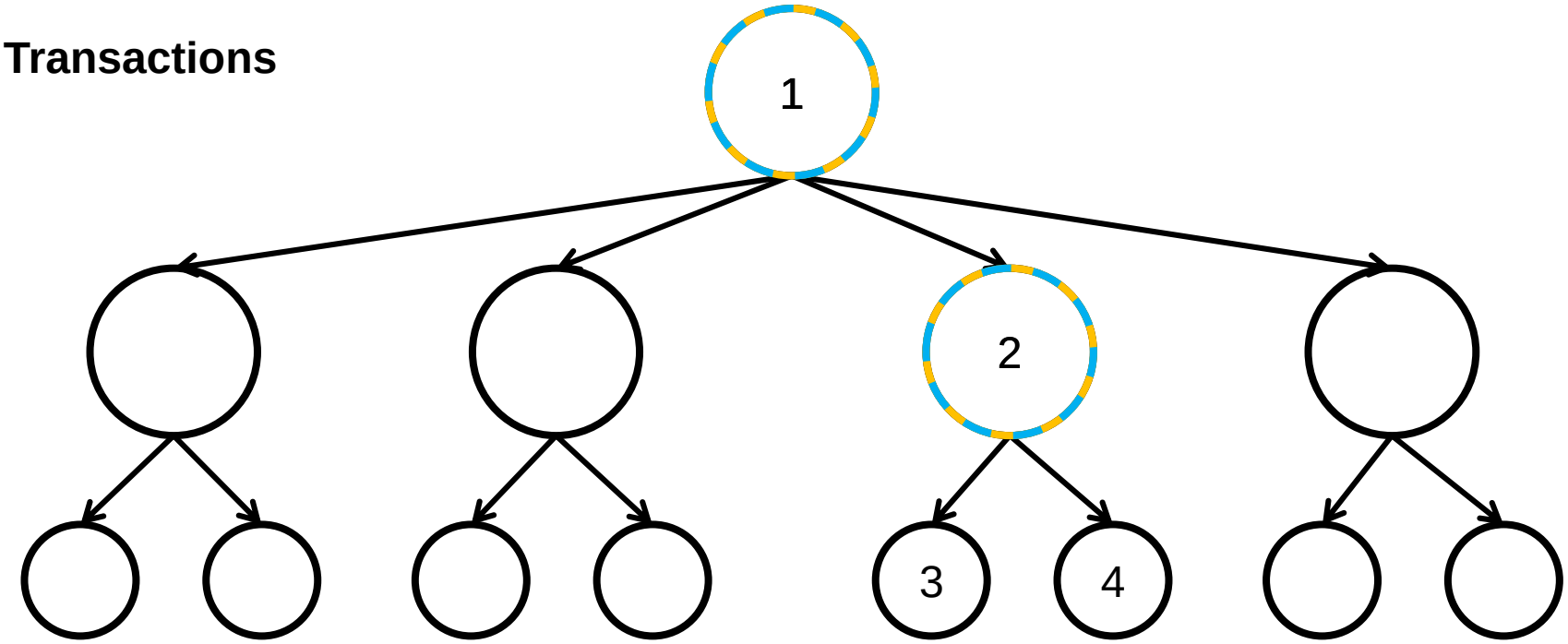
<u>Ακόμα και fine-grain υλοποιήσεις μπορεί να μην εκμεταλλεύονται πλήρως τον παραλληλισμό</u>

Παράδειγμα: ένα lock στον κόμβο 1 μπλοκάρει νήματα που θα μπορούσαν να εκτελεστούν ταυτόχρονα

Tree update από 2 νήματα

- Στόχος: τροποποίηση κόμβων 3 και 4 με thread-safe τρόπο

Transactions



Transaction A
READ 1,2,3
WRITE 3

Transaction B
READ 1,2,4
WRITE 4

Τα δύο updates μπορούν να γίνουν ταυτόχρονα: δεν υπάρχει conflict

Transactional HashMap

- Απλά εσωκλείουμε τη λειτουργία σε ένα atomic block
  - το σύστημα εξασφαλίζει την ατομικότητα

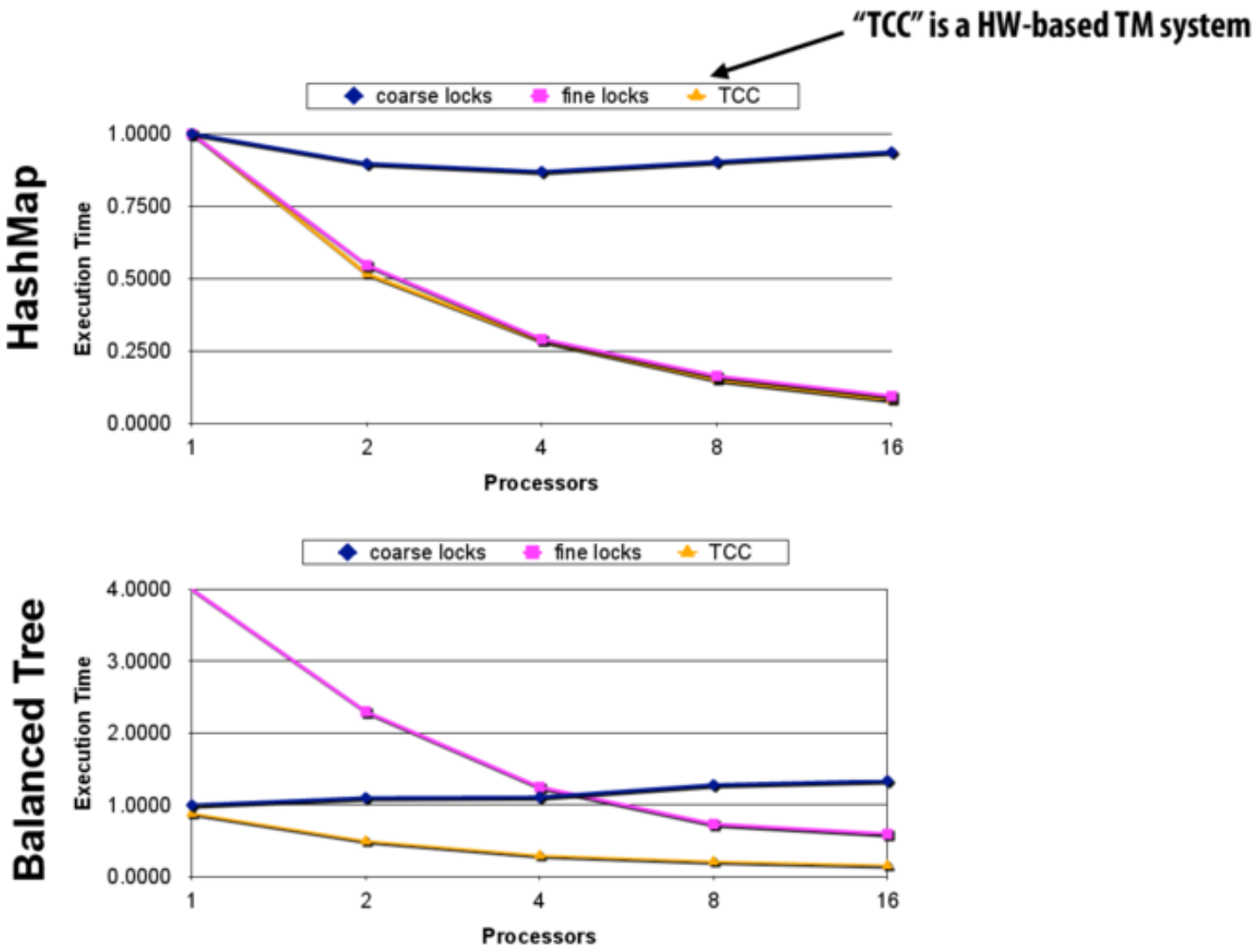
```
public Object get(Object key) {
    atomic {                          // System guarantees atomicity
        return m.get(key);
    }
}
```

- Transactional HashMap

  (+) thread-safe, εύκολο στον προγραμματισμό

  - Καλή επίδοση & κλιμακωσιμότητα?
    - Εξαρτάται από την υλοποίηση του TM και το σενάριο εκτέλεσης, αλλά τυπικά ναι

Επίδοση: Locks vs. TM



Πλεονεκτήματα TM

- Ευκολία προγραμματισμού
  - παραπλήσια με αυτή των coarse-grain locks
  - ο προγραμματιστής δηλώνει, το σύστημα υλοποιεί
- Επίδοση παραπλήσια με αυτή των fine-grain locks
  - εκμεταλλεύεται αυτόματα τον fine-grain ταυτοχρονισμό
  - δεν υπάρχει tradeoff ανάμεσα στην απόδοση & την ορθότητα
- Failure atomicity & recovery
  - δεν «χάνονται» locks όταν ένα νήμα αποτυγχάνει
  - failure recovery = transaction abort + restart
- Composability
  - σύνθεση επιμέρους ατομικών λειτουργιών / software modules σε μία ενιαία ατομική λειτουργία
  - ασφαλής & κλιμακώσιμη

Υλοποίηση TM

- Τα TM συστήματα πρέπει να παρέχουν ατομικότητα και απομόνωση
  - χωρίς να θυσιάζεται ο ταυτοχρονισμός
- Ζητήματα υλοποίησης
  - Data versioning (ώστε να μπορεί να γίνει abort/rollback)
  - Conflict detection & resolution (για να ανιχνεύουμε πότε πρέπει να γίνει abort)
- Επιλογές
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
    - Hardware accelerated STMs
    - Dual-mode systems

Data Versioning

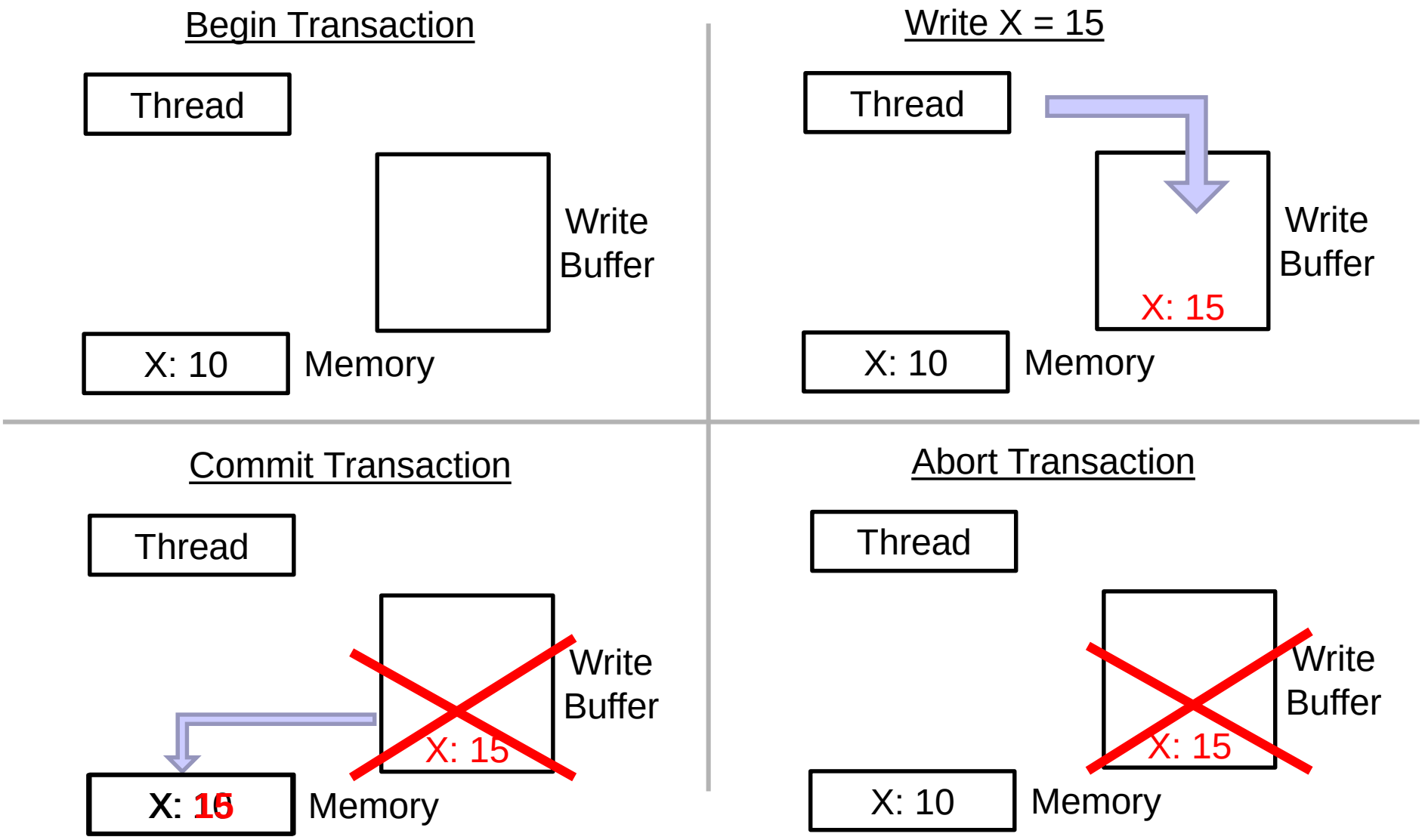
- Διαχείριση uncommitted (νέων) και committed (παλιών) εκδόσεων δεδομένων για ταυτόχρονα transactions

1. Lazy versioning (write-buffer based)

2. Eager versioning (undo-log based)

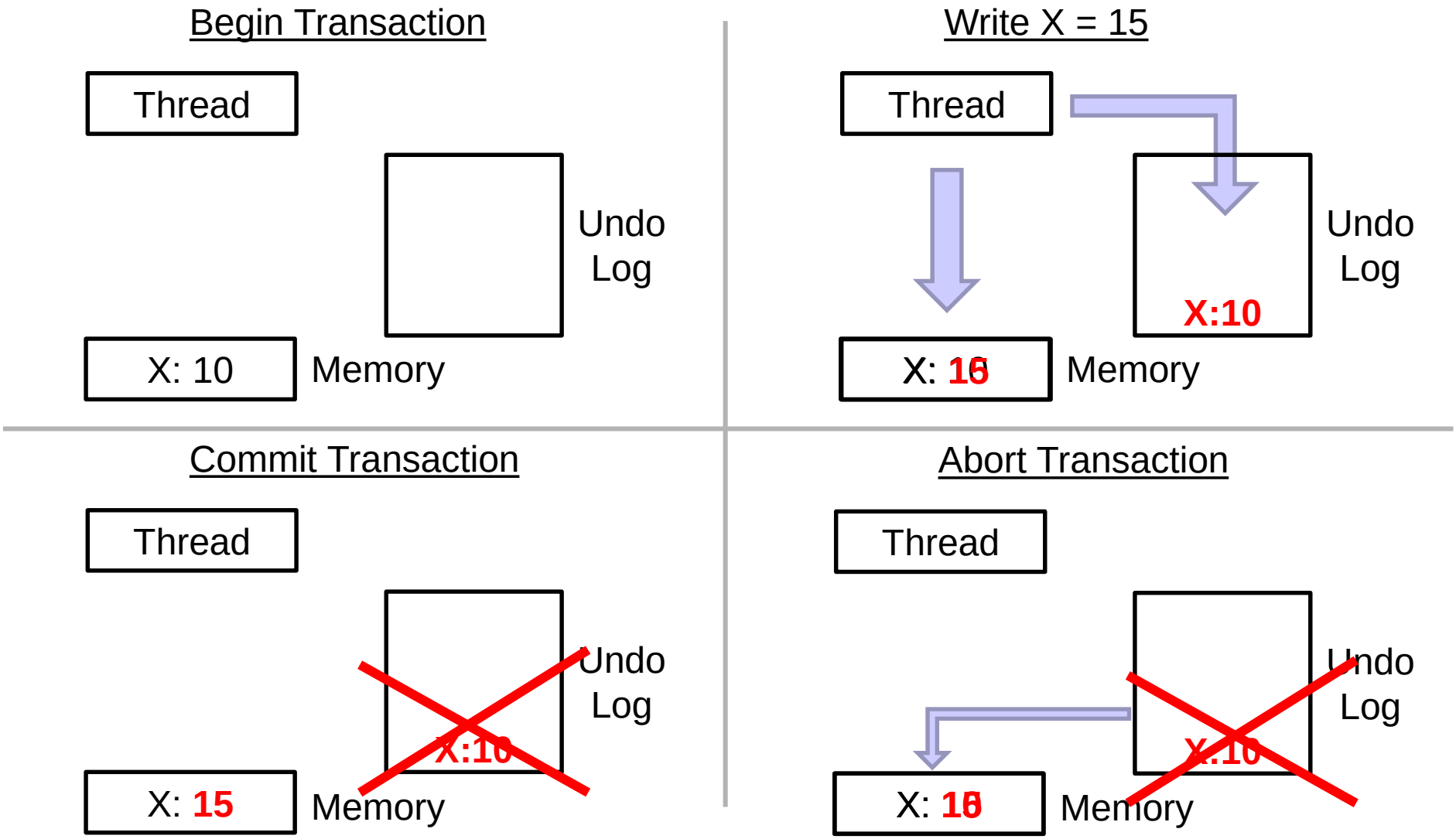
Lazy versioning

Καταγραφή των αλλαγών σε "write buffer", ενημέρωση της μνήμης κατά το commit



Eager versioning

Άμεση ενημέρωση μνήμης, διατήρηση "undo log" για την περίπτωση abort



Data versioning

- Lazy versioning (write-buffer based)
  - καταγραφή νέων δεδομένων σε write-buffer μέχρι το commit
  - πραγματική ενημέρωση μνήμης κατά το commit

  (+) γρήγορο abort, fault tolerant (απλά «πετάμε» τον writer-buffer)

  (-) αργά commits (πρέπει τα δεδομένα να γραφτούν στη μνήμη)

- Eager versioning (undo-log based)
  - απευθείας ενημέρωση μνήμης
  - διατήρηση undo πληροφορίας σε log

  (+) γρήγορο commit (τα δεδομένα είναι ήδη στη μνήμη)

  (-) αργό abort, ζητήματα fault tolerance (τι γίνεται αν ένα thread αποτύχει κατά τη διάρκεια ενός transaction)

Conflict detection

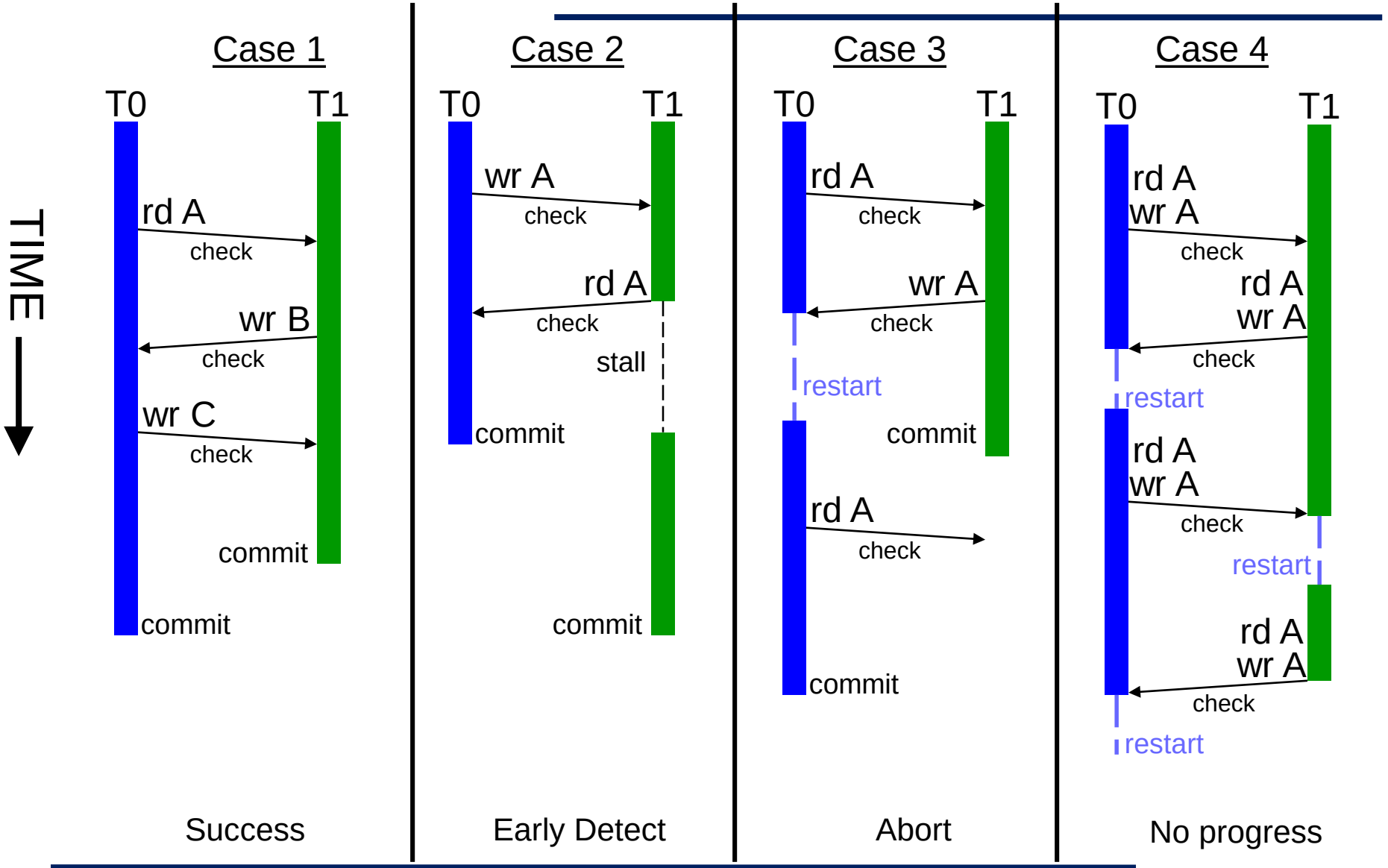
- Ανίχνευση και διαχείριση conflicts ανάμεσα σε transactions
  - **read-write conflict**: ένα transaction A διαβάζει τη διεύθυνση X, η οποία έχει γραφτεί από ένα τρέχων transaction B.
  - **write-write conflict**: δύο transactions γράφουν στην ίδια διεύθυνση μνήμης X
- Πρέπει να παρακολουθούνται το read-set & write-set ενός transaction
  - **read-set**: οι διευθύνσεις που διαβάζονται εντός του transaction
  - **write-set**: οι διευθύνσεις που γράφονται εντός του transaction
- Conflict resolution
  - τι γίνεται όταν ανιχνευθεί ένα conflict
  - διάφορες πολιτικές
    - stall
    - writer wins
    - committer wins
    - ...

Conflict detection

1. Pessimistic (eager) detection
   - Έλεγχος για conflicts σε κάθε load ή store
   - Χρήση contention manager για να αποφασίσει να κάνει stall ή abort
   - Απαισιοδοξία: «Θα έχω conflicts οπότε ας ελέγχω μετά από κάθε πρόσβαση στη μνήμη... Αν χρειαστεί να γίνει abort θα γλιτώσω άσκοπη δουλειά.»

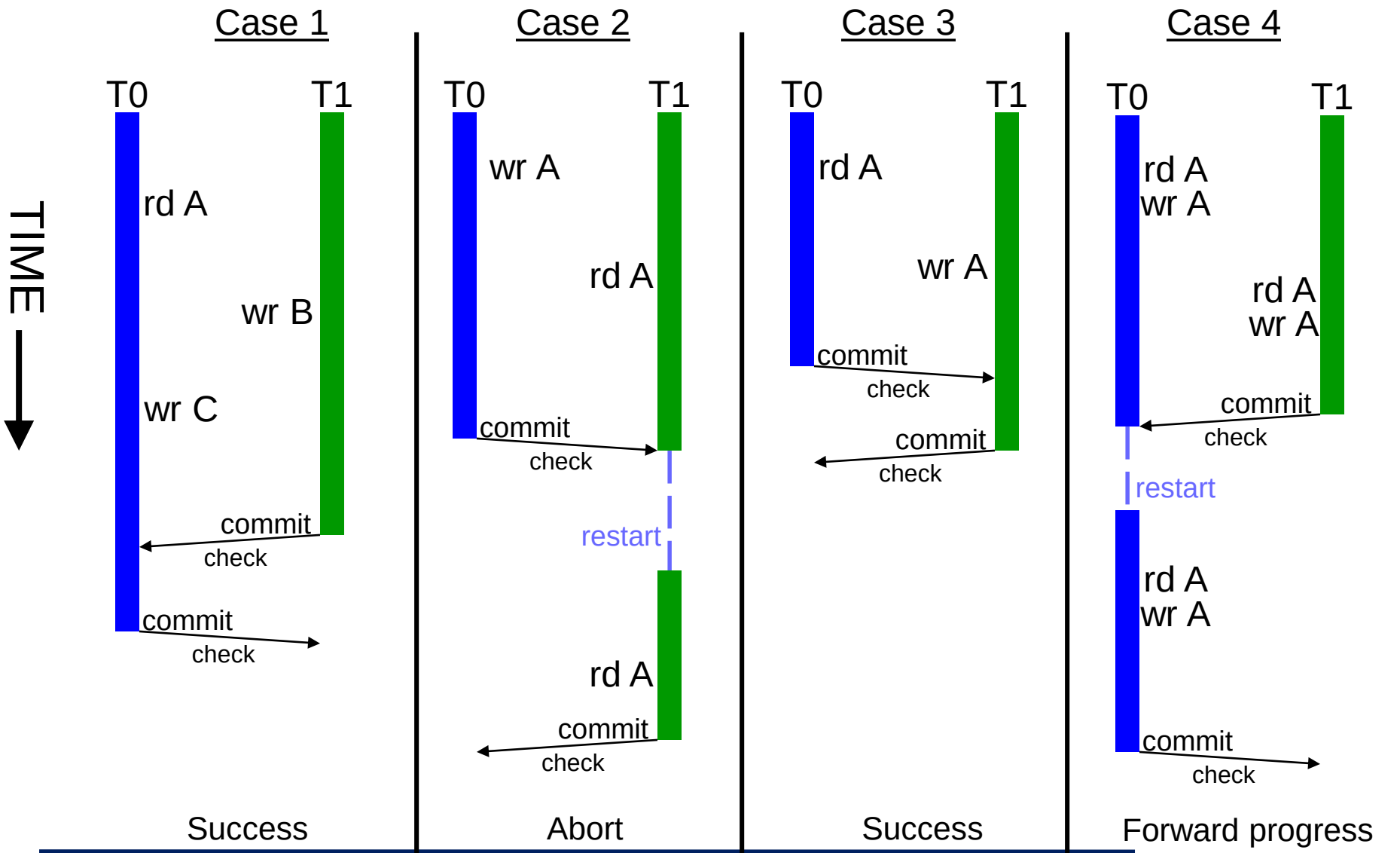
2. Optimistic (lazy) detection
   - Έλεγχος για conflicts κατά το commit
   - Σε περίπτωση conflict, προτεραιότητα στο committing transaction
   - Αισιοδοξία: «Δεν θα έχω conflicts οπότε ελέγχω μόνο κατά την διάρκεια του commit.»

Pessimistic detection



National Technical University of Athens
CSLab

Optimistic detection



Conflict Detection Tradeoffs

1. Pessimistic conflict detection

   (+) ανιχνεύει τα conflicts νωρίς

   - αναιρεί λιγότερη δουλειά, μετατρέπει μερικά aborts σε stalls

   (-) δεν εγγυάται την πρόοδο προς τα εμπρός, πολλά aborts σε κάποιες περιπτώσεις

2. Optimistic conflict detection

   (+) εγγυάται την πρόοδο προς τα εμπρός

   (-) ανιχνεύει τα conflicts αργά, άσκοπη δουλειά

   (-) starvation

TM Implementation Space (παραδείγματα)

- HTM συστήματα
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM, Sun's Rock
  - Eager + pessimistic: Wisconsin LogTM
- STM συστήματα
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd) / pessimistic (wr): MS OSTM
  - Eager + optimistic (rd) / pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM

Hardware TM

- 2011 – IBM Blue Gene/Q
- 2012 – IBM zEC12 mainframe
- 2013 – Intel TSX (Haswell, Broadwell …)
- 2014 – IBM Power8

- Κοινά χαρακτηριστικά
  - εκμετάλλευση των ήδη υπάρχοντων μηχανισμών για coherency
  - conflict detection σε επίπεδο cache line – false sharing
  - περιορισμένο μέγεθος read/write sets
  - aborts για διάφορους λόγους εκτός από data conflict
    - cache line eviction, interrupt …
  - best-effort: δεν εγγυώνται forward progress
    - ένα transaction μπορεί να μην κάνει commit ποτέ
    - είναι ευθύνη του χρήστη να εγγυηθεί forward progress με ένα non-transactional path

Intel TSX

- Διαθέσιμο στους επεξεργαστές Haswell και μεταγενέστερους.
- Η πρώτη υλοποίηση HTM διαθέσιμη σε εμπορικούς επεξεργαστές.
- Στις πρώτες εκδόσεις επεξεργαστών Haswell και Broadwell το TSX είναι buggy
  - Αύγουστος 2014: «**Errata prompts Intel to disable TSX in Haswell, early Broadwell CPUs»**

    http://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus
  - **HSW136. Software Using Intel TSX May Result in Unpredictable System Behavior**

    http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf
  - **Λύση;** Απενεργοποίηση του TSX από το BIOS. ☺

Intel TSX interfaces

- Παρέχονται δύο interfaces για την χρήση του
  1. Restricted Transactional Memory (RTM)
     - 4 νέες εντολές assembly για τη διαχείριση transactions
     - μεγαλύτερη ευελιξία – ο προγραμματιστής επιλέγει τι θα γίνει σε περίπτωση abort
  2. Hardware Lock Elision (HLE)
     - 2 νέα προθέματα εντολών assembly
     - ένα critical section προστατεύεται από ένα lock, το σύστημα προσπαθεί πρώτα να το εκτελέσει σε transaction, αν αποτύχει εκτελείται με την απόκτηση του lock
     - ο κώδικας που προκύπτει εκτελείται και σε παλιότερους επεξεργαστές χωρίς TSX (τα προθέματα αντιμετωπίζονται σαν nops)

Intel TSX - RTM

- XBEGIN <fallback address>
  - δηλώνει την αρχή ενός transaction
  - <fallback address>: η διεύθυνση του κώδικα που θα εκτελεστεί σε περίπτωση abort
- XEND
  - δηλώνει το τέλος ενός transaction
- XTEST
  - δείχνει αν εκτελείται transaction ή όχι
- XABORT <abort code>
  - αναγκάζει το τρέχων transaction να κάνει abort
  - <abort code>: χρησιμοποιείται για την διάκριση μεταξύ διαφορετικών λόγων για abort

Intel TSX – RTM example

```
int aborts = MAX_TX_RETRIES;
lock_t = fallback_global_lock;

start_tx:
int status = TX_BEGIN();
if (status == TX_BEGIN_STARTED) {
    if (fallback_global_lock is locked)
        TX_ABORT();

    ... Critical Section ...

    TX_END();
} else { /* status != TX_BEGIN_STARTED */
    if (--aborts > 0)
        /* retry transaction */
        goto start_tx;

    acquire_lock(fallback_global_lock);
    ... Critical Section ...
    release_lock(fallback_global_lock);
}
```

Ξεκίνα ένα transaction

Το transaction ξεκίνησε επιτυχώς

Προσθήκη του fallback lock στο read-set

Transactional path – σε πέρίπτωση abort το hardware αναλαμβάνει να κάνει rollback και η ροή του προγράμματος επιστρέφει στην κλήση της TX_BEGIN()

Commit

Transaction aborted

Αν δεν έχει ξεπεραστεί το όριο των aborts κάνουμε retry το transaction

Αλλιώς «κλείδωσε» το lock και εκτέλεσε το κρίσιμο τμήμα – **non-transactional path**

Intel TSX – HLE

- 2 νέα προθέματα εντολών assembly
  - XACQUIRE
  - XRELEASE
- Παράδειγμα: elision ενός TAS lock

```
/* acquire lock */
while (__sync_lock_test_and_set(&lock_var) == 0)
    /* do nothing */;

... Critical section with lock acquired ...

/* release lock */
__sync_lock_release(&lock_var);
```

```
/* elide lock */
while (__hle_acquire_test_and_set(&lock_var) == 0)
    /* do nothing */;

... Critical section with lock acquired ...

/* release lock */
__hle_release_clear(&lock_var);
```

- Το σύστημα εκτελεί αρχικά το κρίσιμο τμήμα με transaction
- Αν γίνει abort παίρνει το lock και επανεκτελεί το κρίσιμο τμήμα