

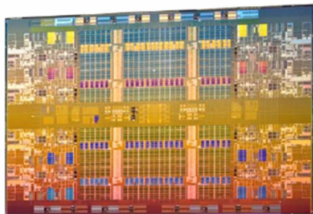
Verification of Concurrent Programs Targeting the x86-TSO Weak Memory Model

Arthur Charguéraud

INRIA

Toccata seminar, 2013/04/19

The multicore revolution



- ▶ Multicore is everywhere
- ▶ Performance matters
- ▶ Programming on a weak memory model is tricky

Multicore programming

Parallelizing an algorithm typically involves:

- ▶ decomposing the work in subtasks,
- ▶ a dynamic load balancing scheduler,
- ▶ a few concurrent data structures.

Good speedups can be achieved, if:

- ▶ work-efficient parallel algorithm,
- ▶ many but not too small subtasks,
- ▶ efficient load balancing algorithm,
- ▶ no lock, contention, or false sharing.

PASL benchmarks, using 30 cores	Speedup
matrix-multiply	21.7
exponential-fibonacci	26.2
maximal-matching (eggrid2d)	19.6
maximal-matching (egr1g)	20.0
maximal-matching (egrmat)	20.1
max-independent-set (grid2d)	17.5
max-independent-set (rlg)	17.9
max-independent-set (rmat)	18.5
quick-hull (plummer2d)	18.0
quick-hull (uniform2d)	19.1
merge-sort (exptintseq)	18.6
merge-sort (randintseq)	21.7
sample-sort (exptseq)	23.2
sample-sort (randdblseq)	23.5

Intel Xeon X7550 2GHz, 4x8 cores, 1Tb

RAM.

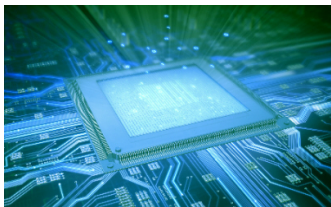
Correctness challenges

The scheduler and the concurrent data structures involve:

- ▶ relatively small pieces of code (a few hundred LOC),
- ▶ code that is very difficult to get right (weak memory model),
- ▶ code that is very hard to test properly (too many interleavings),
- ▶ bugs that can be extremely hard to reproduce.

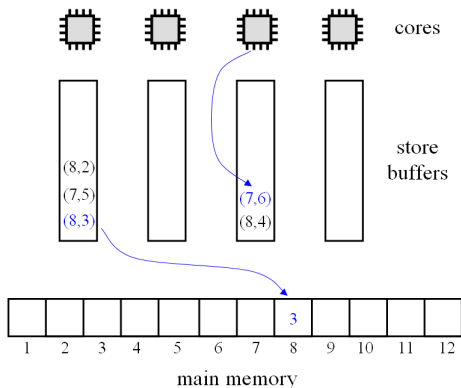
... the ideal scenario for program verification!

x86-TSO: the manufacturer's view



- ▶ Loads may be reordered with older stores to different locations.
- ▶ Other pairs of memory accesses are never reordered.
- ▶ Stores to a same location have a total order.
- ▶ Memory ordering respects transitive visibility.
- ▶ Locked instructions have a total order (e.g., compare-and-swap).

x86-TSO: the researcher's view



- ▶ Memory accesses are atomic operations in the store buffer model.
- ▶ Each core pushes its store operations into its FIFO store buffer.
- ▶ At any time, the oldest store of a buffer can be pushed to memory.
- ▶ Each core reads values from its buffer if possible, else from memory.

x86-TSO: equivalence of the models

The equivalence of the manufacturer's view and the researcher's views is described in the following paper.

A Better x86 Memory Model: x86-TSO. (TPHOLs 2009)

A line of work due to Jade Alglave, Thomas Braibant, Luc Maranget, Magnus Myreen, Scott Owens, Tom Ridge, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli.

First example: a concurrent counter

Global state

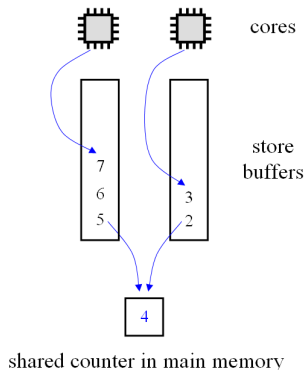
```
int x = 0
```

Code for each core

```
for (int k = 0; k < N; k++)  
    x++ // {int a=x; x=a+1}
```

Specification

$$\min(N, 2) \leq x \leq 2 * N$$



Verification strategy

General strategy to verify a program w.r.t. x86-TSO semantics:

1. **State the invariant** in terms of:
 - ▶ the value in shared memory,
 - ▶ the content of the buffers,
 - ▶ the program point of each core.
2. **Prove stability**: show that pushing the oldest value from a non-empty buffer to the shared memory preserves the invariant.
3. **Prove correctness**: show that the execution of each atomic instruction preserves the invariant.

Verification of the concurrent counter

Notation: B_1 and B_2 denote the buffers, X the value in shared memory, let n_1 and n_2 the number of writes remaining for each core.

Invariant: Let $A_i = \{a_i\}$ if a_i is in scope, else \emptyset . The invariant is:

$$\forall v \in (\{X\} \cup B_1 \cup B_2 \cup A_1 \cup A_2). v \leq n_1 + n_2$$

Stability: Assume B_i is of the form $B'_i \uparrow\uparrow [m]$. We check that the affectations $B_i \leftarrow B'_i$ and $X \leftarrow m$ preserve the invariant.

Correctness: By symmetry, consider only transitions of core 1.

- ▶ $a = x$. The value a_1 is less than $n_1 + n_2$, because it is read from either buffer B_1 or cell X .
- ▶ $x = a+1$. We check that the affectations $B_1 \leftarrow a_1 :: B_1$ and $n_1 \leftarrow n_1 + 1$ preserve the invariant.

Towards modular proofs

`x++`

`y++`

- ▶ Consider a program that uses two independent concurrent counters.
- ▶ We expect to be able to refer twice to the verification of a single counter.
- ▶ To achieve such modularity, we need to allow for independent views on the stores buffered for independent memory cells.



Notation for per-cell store buffers

Given a memory location x , we write:

- ▶ X the value of x in shared memory,
- ▶ \bar{X}^i the buffer containing the stores at location x performed by core i .

By splitting the buffers, we gain modularity, but ...

What have we lost?

By separating store buffers, we lost the ability to reason about the relative ordering between two store operations performed by a same core.

Illustration of the problem

```
// Shared variables
int x = 0
int y = 0

// Code for core 1
void main()
  while (true)
    x++
    y++

// Code for core 2
void main()
  while (true)
    assert (x >= y)
```

Idea of the solution

We allow the invariant to express inequalities between the date at which store operations stored in two buffers \bar{X}^i and \bar{Y}^i associated with a same core i are performed.

Program logic for x86-TSO: data structures

For a memory location x :

- ▶ X denotes the value of x in shared memory.
- ▶ \bar{X}^i now denotes a list of pairs made of a value and a timestamp, representing the buffered stores made by i at location x .

Moreover, for convenience, we let:

- ▶ \bar{X}^i denote directly a list of values when timestamps are not relevant.
- ▶ \vec{X}^i denote the list $\bar{X}^i \uparrow\uparrow [X]$,
- ▶ X^i denote the value of x as seen by core i , i.e., the head of \vec{X}^i .

Program logic for x86-TSO: invariant

The invariant is expressed in terms of:

- ▶ the program point (line number and values of the local variables) of each core,
- ▶ the shared memory values of each memory location,
- ▶ the values contained in the buffers associated with each core and each memory location,
- ▶ comparisons between the timestamps associated with entries in the buffers, only for pairs of buffers that belong to the same core.

Program logic for x86-TSO: stability

To prove stability:

- ▶ we assume that the invariant holds,
- ▶ we consider an arbitrary buffer \bar{X}^i of the form $(v, t) :: B$,
- ▶ we assume that t is smaller than the timestamp contained in any other buffer \bar{Y}^i associated with the same core i ,
- ▶ we prove that the updates $X \leftarrow v$ and $\bar{X}^i \leftarrow B$ preserve the invariant.

Program logic for x86-TSO: correctness

To prove correctness,

- ▶ we assume that the invariant holds,
- ▶ we consider an atomic operation from the code executed by a core i ,
 - ▶ If the read is a write $x \leftarrow v$, then, for an arbitrary timestamp t greater than any timestamp stored in a buffer of the form \bar{Y}^i , we show that the update $\bar{X}^i \leftarrow (v, t) :: \bar{X}^i$ preserves the invariant.
 - ▶ If the operation consists of a read of the content of x into a register a , we show that the updates $a \leftarrow \text{head}(\bar{X}^i \uparrow [X])$ preserves the invariant.

Program logic for x86-TSO: correctness, cont.

- ▶ If the operation is a memory fence, we assume all buffers of i to be empty and we show that stepping to the next program point preserves the invariant.
- ▶ If the operation is $a = \text{fetch-and-add}(\&x, v)$, we assume all buffers of i to be empty, and we show that the updates $a \leftarrow v$ and $X \leftarrow X + v$ preserve the invariant.
- ▶ If the operation is $a = \text{compare-and-swap}(\&x, v, w)$, we assume all buffers of i to be empty. We show that under the assumption $X = v$ the updates $a \leftarrow \text{true}$ and $X \leftarrow w$ preserve the invariant, and that under the assumption $X \neq v$ the update $a \leftarrow \text{false}$ preserves the invariant.

Program logic for x86-TSO: summary

Summary of program verification w.r.t. x86-TSO:

- ▶ statement of the invariant,
- ▶ stability proof,
- ▶ correctness proof.

Completeness: by introducing sufficiently many constraints between buffers, we can describe any state as accurately as if we were reasoning directly on the model with a single buffer per core.

Applications

1. Lock implemented using compare-and-swap
2. Reachability in a graph
3. Single-consumer single-producer FIFO
4. Multi-consumer single-producer FIFO
5. Work stealing using compare-and-swap
6. Work stealing without compare-and-swap

Implementation of locks using CAS

Code pattern

```
...  
if (compare_and_swap(&x, 0, 1))  
{  
    // critical section begins  
    ...  
    x--  
    // critical section ends  
}  
...
```

Invariant

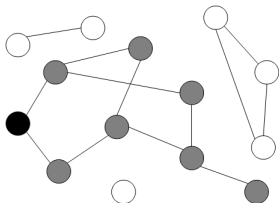
If core i is in the critical section, then $\vec{X}^i = 1 :: \text{nil}$, and, for any $j \neq i$, the core j is not in the critical section and $\vec{X}^j = \text{nil}$.

Otherwise, if i is not in the critical section, then $\vec{X}^i = 0 :: 1 :: \text{nil}$ or $\vec{X}^i = 0 :: \text{nil}$.

Stability: easy. **Correctness:** if core i succeeds on the CAS, we have $\vec{X}^j = \text{nil}$ and $X = 0$, so no core may already be in the critical section.

Reachability in graphs

Goal: compute in parallel the set of nodes accessible from a given source.



- ▶ **Traditional version:** a compare-and-swap operation is used to ensure that each node is processed at most once.
- ▶ **Idempotent version:** visited nodes are simply marked using a write operation, saving the cost of the compare-and-swap operation at the expense of potentially processing nodes and edges more than once.

Reachability in graphs, traditional version

Source code

```
int visited[N]
stack<node_id> to_visit

while true
  if ! to_visit.empty()
    node_id n = to_visit.pop()
    foreach m in edges[n]
      if CAS(&visited[m], 0, 1)
        to_visit.push(m)
  else
    perform_load_balancing()
```

Invariant

Let x denote the cell `visited[n]`.

- ▶ For all i , we have $\bar{X}^i = \text{nil}$.
- ▶ $X = 1$ iff there exists a path from the source to the node n made of processed edges.

An edge (a, b) is *processed* if `visited[a] = 1` and, for all i , we have $a \notin \text{to_visit}[i]$ and if $n_i = a$ then (a, b) has already been handled by the for loop.

Reachability in graphs, idempotent version

Source code

```
int visited[N]
stack<node_id> to_visit

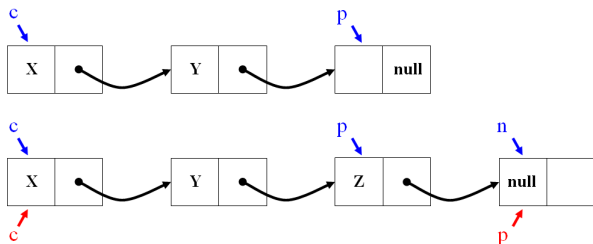
while true
  if ! to_visit.empty()
    node_id n = to_visit.pop()
    for each m in edges_from[n]
      if visited[m] == 0
        visited[m] = 1
        to_visit.push(m)
  else
    perform_load_balancing()
```

Invariant

Let x denote the cell `visited[n]`.
For all i , $1 \in \vec{X}^i$ iff there exists a path from the source to the node n made of edges processed in the eye of i .

An edge (a, b) is *processed in the eye of i* if [????]
 $visited[a]^i = 1$ and, for all i , we have $a \notin to_visit[i]$ and if $n_i = a$ then (a, b) has already been handled by the for loop.

Single-producer single-consumer FIFO



Consumer code

```
while true
  if c.next != null
    received.push(c.data)
    c = c.next
```

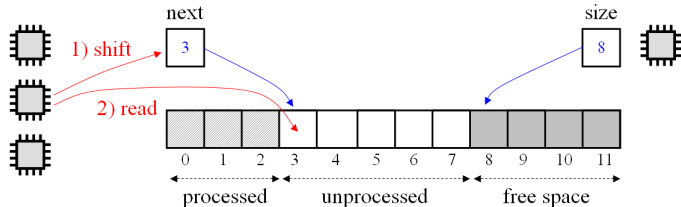
Producer code

```
while ! tosend.empty()
  p.data = tosend.pop()
  n = new cell()
  n.next = null
  p.next = n
  p = n
```

Multi-consumer single-producer FIFO

Data structure

```
array<item> data;  int capacity;  int size = 0;  int next = 0;
```



Producer code

```
// assumes(size < capacity)
void push(item v)
    data[size] = v
    size++
```

Consumer code

```
while true
    int n = next
    if n < size &&
        CAS(&next, n, n+1)
        process(data[n])
```

Multi-consumer single-producer FIFO

For simplicity, let core 0 be the producer and $1 \dots (P - 1)$ be the consumers.

Invariants

- ▶ $\forall i \in [1, P). \overline{\text{size}}^i = \text{nil} \wedge \forall k. \overline{\text{data}[k]}^i = \text{nil}$
- ▶ $\forall i \in [0, P). \overline{\text{next}}^i = \text{nil}$
- ▶ $\forall k \in [0, \text{size}^0). \text{data}[k] = \text{produced}[k]$
- ▶ $\overrightarrow{\text{size}}^0$ is a decreasing list of consecutive integers
- ▶ $0 \leq \text{next} \leq \text{size} \leq \text{size}^0 < \text{capacity}$
- ▶ If $\text{Line}(i) > 2$ for some $i \in [1, P)$, then $n_i \in [0, \text{size}]$
- ▶ If $\text{Line}(i) > 3$ for some $i \in [1, P)$, then $n_i \in [0, \text{size}]$
- ▶

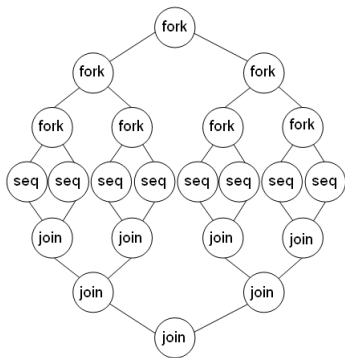
$$\bigcup_{k \in [0, \text{next})} \{\text{data}[k]\} = \bigcup_{i \in [1, P)} \text{consumedBy}[i] \cup \bigcup_{\substack{i \in [1, P) \\ \text{Line}(i) = 5}} \text{data}[n_i]$$

Introduction to the fork-join model

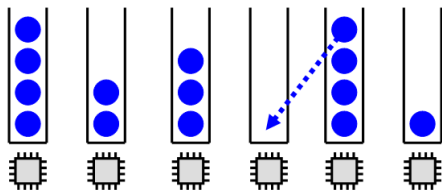
Quick-hull pseudo-code

```
let rec quick_hull points =  
  if points.size < cutoff  
    graham_hull(points)  
  else  
    let (p1,p2) = split points  
    let (h1,h2) = (| quick_hull p1,  
                   quick_hull p2 |)  
    merge_hulls h1 h2
```

Computation DAG



Introduction to work stealing



- ▶ Each core owns a deque of tasks, and work by popping and pushing tasks from the bottom of its deque.
- ▶ When a core runs out of work, it pops a task from the top of another core's deque, selected at random.

→ Work stealing performs very well both in theory and in practice.
The expected number of task migration is $O(\text{nbCores} * \text{depthOfTheDAG})$.

Work stealing with concurrent dequeues

```
struct Deque = { int size;
                item* data };

// per core
Deque* deque
int bottom
int top
// global
Deque* dequees[P]

void push_bottom(item v)
    int b = bottom
    int t = top
    Deque* q = deque
    if b-t >= q.size-1
        expand()
        q = deque
    q.data[b % q.size] = v
    bottom = b+1

void expand()
    int nb = deque.size
    Deque* q = new Deque(2*nb)
    for (i = top; i < bottom; i++)
        q.data[i % (2*nb)] =
            deque.data[i % nb]
    deque = q
    dequees[myid] = q

item pop_bottom()
    int b = bottom-1
    Deque* q = deque
    bottom = b
    store_load_fence
    int t = top
    if b < t
        bottom = t
        return EMPTY
    item v = q.data[b % q.size]
    if b > t
        return v
    if ! CAS(&top, t, t+1)
        return EMPTY
    bottom = t+1
    return v

item pop_top_from(int id)
    int t = top
    int b = bottom
    Deque* q = dequees[id]
    if t >= b
        return EMPTY
    item v = q.data[t % q.size]
    if ! CAS(&top, t, t+1)
        return ABORT
    return v
```

Work stealing with private deques

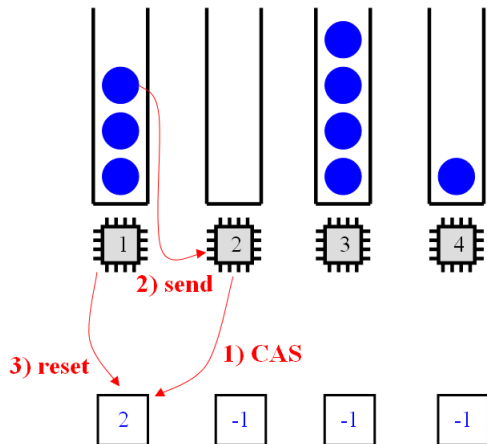
Abandon the idea of using a concurrent data structure:

- ▶ a private deque associated with each core,
- ▶ load balancing is performed using explicit communication,
- ▶ local accesses to the deques are faster and more flexible,
- ▶ performance is maintained if the communication period is properly set.

Two strategies:

- ▶ **Receiver-initiated:**
steal requests by idle cores, periodic polling by busy cores.
- ▶ **Sender-initiated:**
idle cores raise their flag, periodic deal attempts by busy cores.

Receiver initiated work stealing



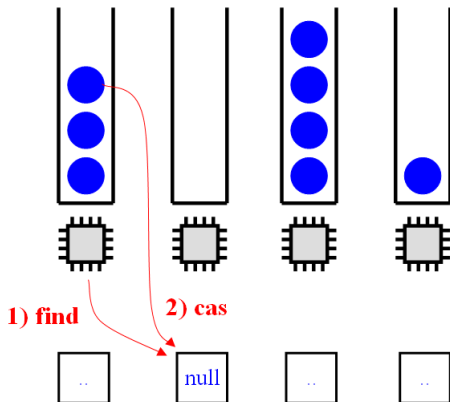
Receiver-initiated work stealing

```
// per core
queue<task>* deque
// global
int query[P]
task* answer[P]

// called when out of work
void acquire()
    while true
        answer[myid] = NONE
        int k = random_other_id()
        if CAS(&query[k], NONE, myid)
            while answer[myid] == NONE
                communicate()
            task* t = answer[myid]
            if t != null
                deque.push_bottom(t)
            return
```

```
// called periodically when busy
void communicate()
    int j = query[myid]
    if j == NONE then return
    if deque.empty()
        answer[j] = null
    else
        answer[j] = deque.pop_top()
    query[myid] = NONE
```

Sender-initiated work stealing



Sender-initiated work stealing

```
// per core
queue<task>* deque
// global communication cells
task* c[P]
// dummy task pointer
const task* SOON = 1

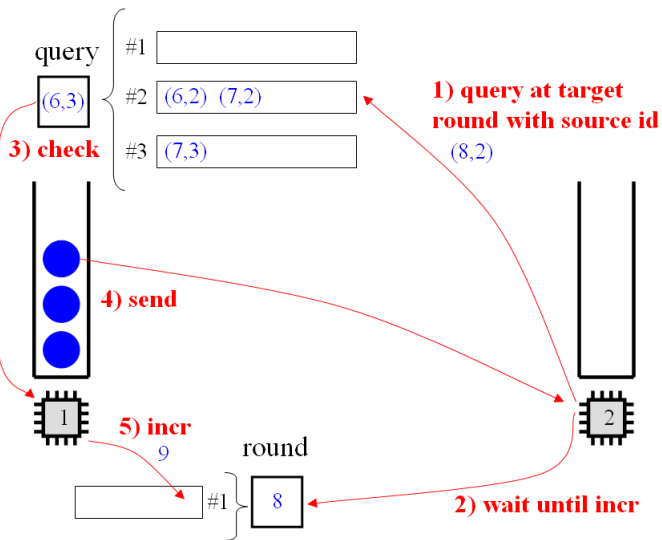
// called when out of work
void acquire()
    c[myid] = null
    while c[myid] == null
        noop
    deque.push_bottom(c[myid])
```

```
// called periodically when busy
void communicate()
    if deque.empty() then return
    int j = random_other_id()
    if c[j] != null then return
    task* t = deque.get_top()
    bool r = CAS(&c[j], null, t)
    if r then deque.pop_top()
```

Fence-free work stealing

Question: can we efficiently implement work stealing on multicore architecture without using any locked operation (compare-and-swap, fetch-and-add, memory fence)?

Fence-free sender-initiated work stealing



Fence-free sender-initiated work stealing

```
// per core
queue<task>* deque
// global
int round[P]
Query query[P]
task* answer[P]

// called periodically when busy
void communicate()
    // check incoming queries
    Query q = query[myid]
    if q.round != round[myid]
        return
    // answer the queries
    if deque.size() > 0
        answer[q.id] = deque.pop_top()
    // starts a new phase
    round[myid]++

// called to reject/prevent queries
void block()
    int r = round[myid]
    if query[myid] != Query(myid,r)
        query[myid] = Query(myid,r+1)
        round[myid] = r+1

// 64bit representation of queries
type Query = { 40bits round;
               24bits id }

// called when out of work
void acquire()
    answer[myid] = null
    while true
        int j = random_other_id()
        int r = round[j]
        if query[j].round < r
            // send a query
            query[j] = Query(i,r)
            while round[j] == r
                // resend if needed
                if query[j].round < r
                    query[j] = Query(i,r)
            block()
            // receive task if any
            task* t = answer[myid]
            if t != null
                deque.push(t)
                round[myid]++
            return
    block()
```

Fence-free sender-initiated work stealing

- $A_i \equiv L_i \in [71; 90]$
(i is in the main loop of the acquire function)
- $B_i \equiv A_i \wedge (\forall k \neq i. \bar{T}_i^k = \text{nil}) \wedge (\bar{T}_i^i = \text{NULL} :: \text{nil} \vee (\bar{T}_i^i = \text{nil} \wedge T_i = \text{NULL}))$
(i has set its reception field and is in the main loop of acquire)
- $C_{i,j,r} \equiv L_i \in [78; 82] \wedge j = j_i^{72} \wedge r = r_i^{73} \leq R_j^j$
(i has made an answer to j at round r and is waiting for an answer)
- $E_{i,j,t} \equiv L_i \in [78; 87] \wedge i \neq j \wedge j = j_i^{72} \wedge (\forall k \neq j. \bar{T}_i^k = \text{nil}) \wedge ((\bar{T}_i^j = t :: \text{nil} \wedge H) \vee (\bar{T}_i^j = \text{nil} \wedge T_i = t))$
where H asserts that the last write in \bar{T}_i^j occurred before the write of any value greater than r_i^{73} in \bar{R}_j^j
(i has obtained the task t from j but has not yet pushed it in deque)
- $E_i \equiv \exists j t. E_{i,j,t}$

Fence-free sender-initiated work stealing

- $\mathcal{I}_{A_1} \equiv$ If A_i is false then $\forall k. \bar{T}_k^i = \text{nil}$.
- $\mathcal{I}_{A_2} \equiv$ If A_i is false and if \vec{Q}_i^i contains a query with id i , then round number of this query is less than R_i^i .
- $\mathcal{I}_{R_1} \equiv \forall j \neq i. \bar{R}_j^i = \text{nil}$ (meaning that processors update only their own round numbers).
- $\mathcal{I}_{R_2} \equiv \bar{R}_i^i$ is a list of strictly decreasing values (meaning that round numbers only increase through time).
- $\mathcal{I}_{Q_1} \equiv$ All the queries in \vec{Q}_j^i have an id equal to i (meaning that queries always contain the id of their sender).
- $\mathcal{I}_{Q_2} \equiv$ All the queries in \vec{Q}_j^i have a round number no greater than R_j^j ,
with one exception: the case where $L_j = 111$ (block function) and the query has round number $R_j^j + 1$.
- $\mathcal{I}_{Q_3} \equiv$ If a query in \vec{Q}_j^k has a round number equal to R_j^j and an id i with $i \neq j$, then $C_{i,j,r}$ is true for $r = R_j^j$ and this query is the last write in \vec{Q}_j^i and it occurred after the last write in \bar{T}_i^i .
- $\mathcal{I}_{T_1} \equiv$ If A_i is true (i in the main loop of acquire) then either B_i or E_i is true
(in the first case, T_i^i is NULL, whereas in the second case T_i^i is or will soon become not NULL).
- $\mathcal{I}_{T_2} \equiv$ If $L_i \in [84; 85]$ (just about to read the reception field) then $r_i^{73} < R_j$.
- $\mathcal{I}_{T_3} \equiv$ If $L_i \in [86; 85]$ (just read a non-null pointer) then E_i is true and T_j is the task pointer read.

Conclusion

Program verification for x86-TSO

- ▶ Model: one buffer per cell and per core, plus time constraints.
- ▶ Schema: statement of the invariant, stability proof, correctness proof.
- ▶ Applications: concurrent data structures, algorithms exploiting races.

Future work

- ▶ A tool for mechanizing the proofs,
- ▶ Partial inference of the invariants,
- ▶ Integration with separation logic.