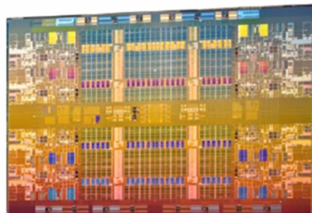
Verification of Concurrent Programs Targeting the x86-TSO Weak Memory Model

Arthur Charguéraud

INRIA

Parkas seminar, 2013/05/27

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,
- avoiding locks, atomic read-write ops, memory fences, contention.

Multicore programming

Parallelizing an algorithm typically involves:

- decomposing the work in subtasks,
- a dynamic load balancing scheduler,
- a few concurrent data structures,
- avoiding locks, atomic read-write ops, memory fences, contention.

PASL benchmarks, using 30 cores	Speedup
matrix-multiply	21.7
exponential-fibonnacci	26.2
maximal-matching (eggrid2d)	19.6
maximal-matching (egrlg)	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 dozens LOC),
- code that is very difficult to get right (weak memory model),
- code that is very hard to test (too many interleavings),
- bugs that can be extremely hard to reproduce.

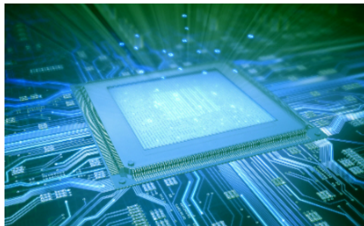
Correctness challenges

The scheduler and the concurrent data structures involve:

- relatively small pieces of code (a few dozens LOC),
- code that is very difficult to get right (weak memory model),
- code that is very hard to test (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 occur atomically and have a total order.

Locked instructions

- `compare_and_swap(&x, 2, 3)`
  - Let `v` be the content of `x`
  - If `v` is 2, then write 3 into `x`, and return true
  - If `v` is not 2, then do nothing, and return false
- `fetch_and_add(&x, 3)`
  - Let `v` be the content of `x`
  - Write `v+3` into `x`
  - Return `v`
- `x = 3; memory_fence(); a = y`
  - Ensures that the read of `y` is not reordered prior to the write of `x`.

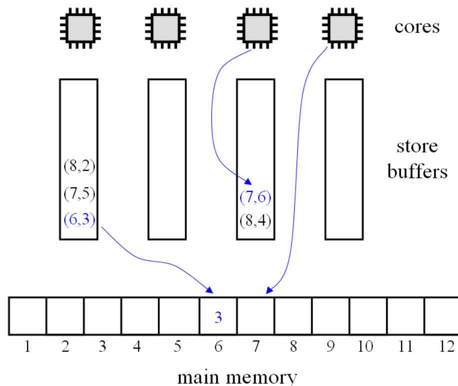
x86-TSO: equivalence of the models

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.

Establishes the equivalence between the manufacturer's view and a relatively simple model.

x86-TSO: the researcher's view



- Cores write values into their buffer.
- Cores read values from their buffer if possible, else from main memory,
- At any time, the oldest store of a buffer can be pushed to memory.
- Locked instructions are only executed with an empty buffer.

First example: a concurrent counter

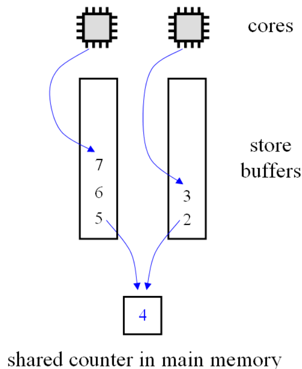
Global state

```
int x = 0
```

Code for each of the two cores

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

Specification?



First example: a concurrent counter

Global state

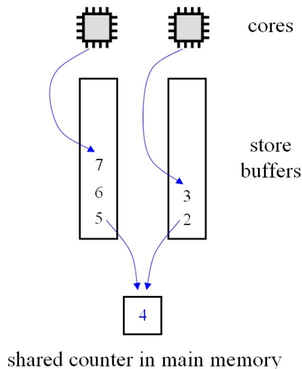
```
int x = 0
```

Code for each of the two cores

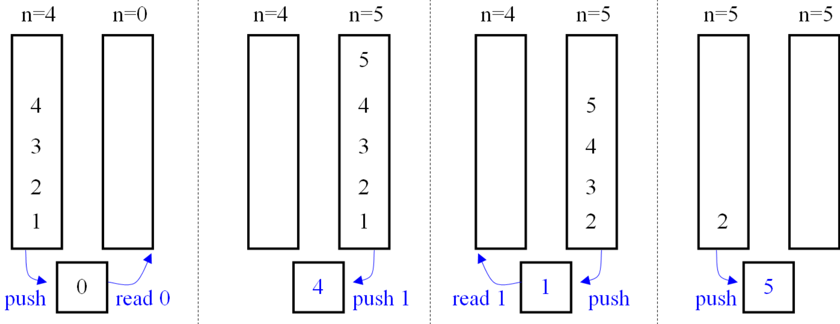
```
for (int k = 0; k < N; k++)
  x++   // {int a=x; x=a+1}
```

Specification?

$$\min(\mathtt{N}, 2) \leq \mathtt{x} \leq 2 * \mathtt{N}$$



First example: a concurrent counter



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

Verification of the concurrent counter

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

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

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

Verification of the concurrent counter

Notation: let B_1 and B_2 denote the buffers, X the value in shared memory, and n_1 and n_2 the number of writes performed by 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 $B_i' +\!\!+ [m]$. We check that the affectations $B_i \leftarrow B_i'$ and $X \leftarrow m$ preserve the invariant.

Verification of the concurrent counter

Notation: let B_1 and B_2 denote the buffers, X the value in shared memory, and n_1 and n_2 the number of writes performed by 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 $B'_i +\!\!+ [m]$. 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 $B_1 \leftarrow (a_1 + 1) :: B_1$ 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.
- Need independent views on the stores buffered for distinct 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.

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, however ...

The lost ordering

We loose the ability to reason about the relative ordering between store operations performed by a same core.

Illustration of the problem

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

// Code for core 1
while (true) {
  x++
  y++
}

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

The lost ordering

We loose the ability to reason about the relative ordering between store operations performed by a same core.

Illustration of the problem

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

// Code for core 1
while (true) {
  x++
  y++
}

// Code for core 2
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.

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.

Additional notation:

- $\vec{X}^i$ denotes the list $\bar{X}^i ++ [X]$,
- X^i denotes the value of x as seen by core i, that is, the head of $\vec{X}^i$.

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.

Additional notation:

- $\vec{X}^i$ denotes the list $\bar{X}^i +\!\!+ [X]$,
- X^i denotes the value of x as seen by core i, that is, the head of $\vec{X}^i$.

Remark: when convenient, $\bar{X}^i$ and $\vec{X}^i$ may be interpreted as list of values.

Program logic for x86-TSO: invariant

The invariant is expressed in terms of:

- the program point of each core (line number and local variables),
- the shared memory values of every memory location,
- the values contained in the buffers of each core and each location,
- comparisons between timestamps from buffers of 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 $B +\!\!+ [(v, t)]$,
- 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 operation 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 is a read of the content of x into a local variable a, we show that the updates $a \leftarrow \text{head}(\bar{X}^i +\!\!+ [X])$ preserves the invariant.

Program logic for x86-TSO: correctness, cont.

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

Program logic for x86-TSO: summary

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

- state the invariant,
- prove stability,
- prove correctness.

Program logic for x86-TSO: summary

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

- state the invariant,
- prove stability,
- prove correctness.

Completeness (conjecture): any x86-TSO program behavior can be verified using this logic of per-cell store buffers with time constraints.

Applications

Locks and atomicity

- Locks implemented using compare-and-swap
- Reachability in graphs

Concurrent FIFO

- Single-consumer single-producer FIFO
- Multi-consumer single-producer FIFO

Work stealing

- Work stealing using concurrent deques
- Work stealing using private deques, using compare-and-swap
- Work stealing using private deques, 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
}
...
```

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 $X = 1$, and $\forall j.\ \bar{X}^j = \text{nil}$, and cores other than i are not in the critical section.

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

Implementation of locks using CAS

Code pattern

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

Invariant

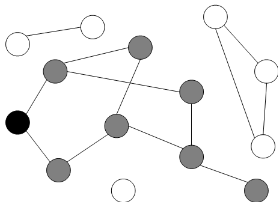
If core i is in the critical section, then $X = 1$, and $\forall j.\ \bar{X}^j =$ nil, and cores other than i are not in the critical section.

Else, if i is not in the critical section, then either $\bar{X}^i = 0 ::$ nil and $X = 1$, or $\bar{X}^i =$ nil and $X = 0$.

Stability: easy.
Correctness: if core i succeeds on the CAS, we have $\bar{X}^j =$ 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 node are simply marked using a write operation, saving the cost of the compare-and-swap operation.

Reachability in graphs, traditional version

Source code

```
// global
int visited[N]

// per core
stack<node_id> to_visit

// code of each core
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()
```

Reachability in graphs, traditional version

Source code

```
// global
int visited[N]

// per core
stack<node_id> to_visit

// code of each core
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 $\bar{X}^i = \text{nil}$.
- $X = 0$ or $X = 1$
- $X = 1$ iff there exists a path from the source to the node n made of *processed* edges.

Reachability in graphs, traditional version

Source code

```
// global
int visited[N]

// per core
stack<node_id> to_visit

// code of each core
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 $\bar{X}^i = \text{nil}$.
- $X = 0$ or $X = 1$
- $X = 1$ 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 $a \notin \overrightarrow{\texttt{to_visit[i]}}^i$ and if $\mathtt{n}_i = a$ then (a, b) has already been handled by the for loop.

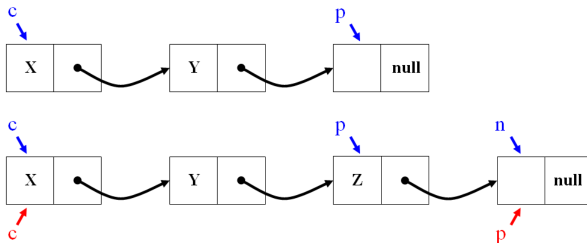
Reachability in graphs, idempotent version

Source code

```
// global
int visited[N]

// per core
stack<node_id> to_visit

// code of each core
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()
```

Single-producer single-consumer FIFO

```
struct cell = { void* data; cell* next }
```



Consumer code

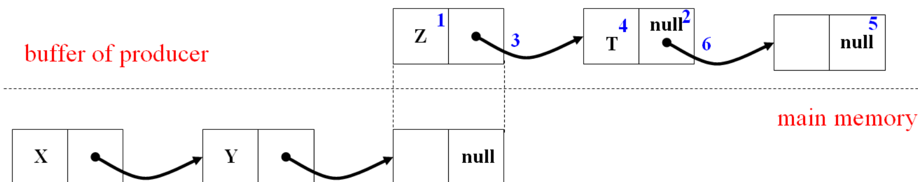
```
while true
  if c.next != null
    received.push(c.data)
    c = c.next
```

Producer code

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

Single-producer single-consumer FIFO

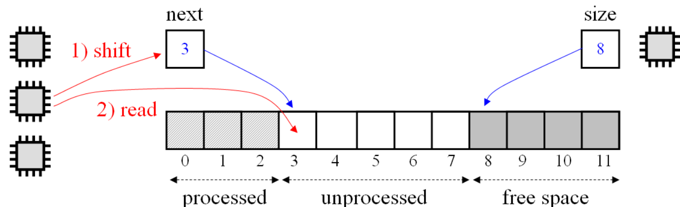
Invariant: there exists a non-empty list L of pairs of locations and items, with all locations in L disjoint, describing, e.g., the picture below.



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 $1 ... (P-1)$ 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 $\text{n}_i \in [0, \text{size}]$
- If $\text{Line}(i) > 3$ for some $i \in [1, P)$, then $\text{n}_i \in [0, \text{size})$
- $$\bigcup_{k \in [0, \text{next})} \{\text{data}[k]\} = \bigcup_{i \in [1, P)} \text{consumedBy}[i] \quad \cup \bigcup_{\substack{i \in [1, P) \\ \text{Line}(i) = 5}} \text{data}[\text{n}_i]$$

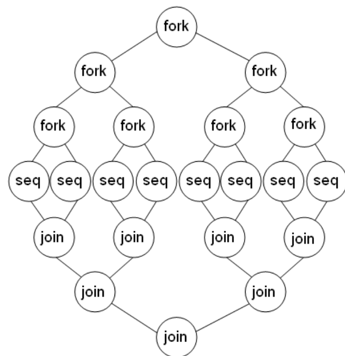
Work stealing

Introduction to the fork-join model

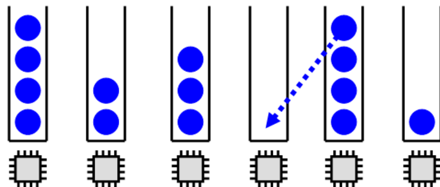
Quick-hull pseudo-code

```
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

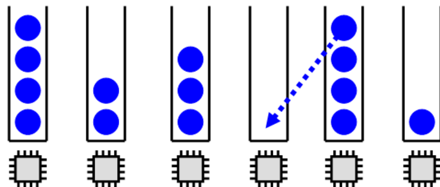


Introduction to work stealing



- 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 picks a task from the top of another core's deque, selected at random.

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 picks a task from the top of another core's deque, selected at random.

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

Work stealing with concurrent deques

```
struct Deque = { int size;
                 item* data }
// per core
Deque* deque
int bottom
int top
// global
Deque* deques[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
  deques[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 = deques[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.

Instead, associate to each core a private deque:

- load balancing through explicit communication,
- faster, more flexible local operations,
- requires frequent communication.

Work stealing with private deques

Abandon the idea of using a concurrent data structure.

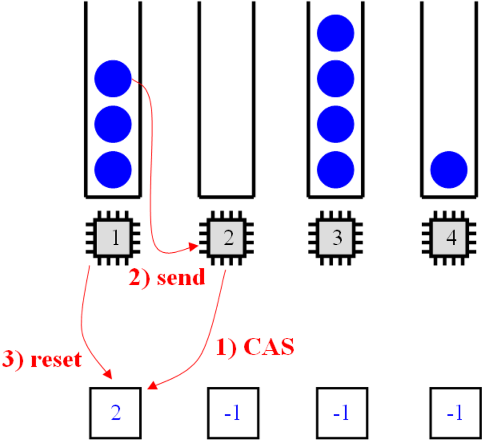
Instead, associate to each core a private deque:

- load balancing through explicit communication,
- faster, more flexible local operations,
- requires frequent communication.

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

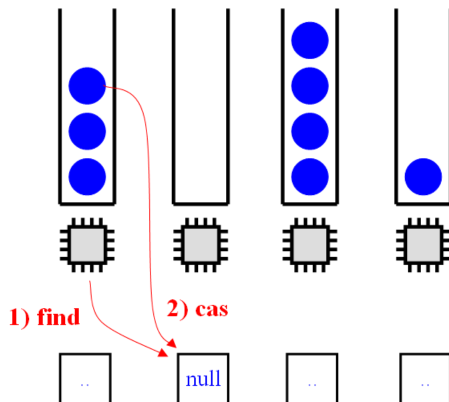


Receiver-initiated work stealing

```
// 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

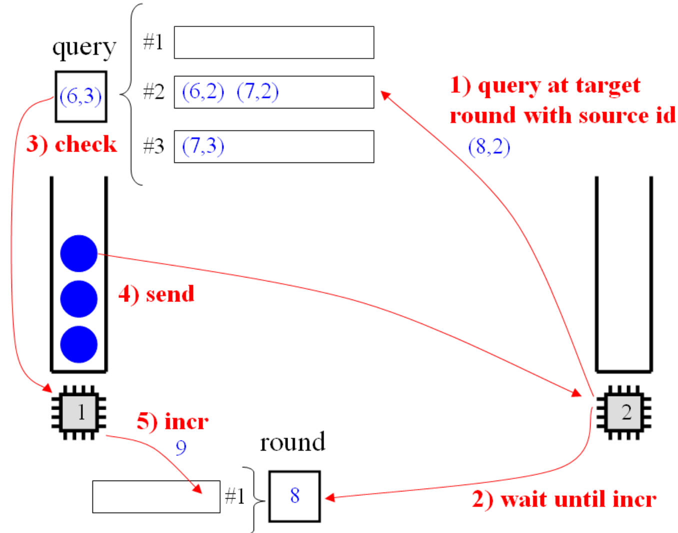
```
// per core
queue<task>* deque

// global
task* c[P]

// 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)

$B_i \equiv A_i \wedge (\forall k \neq i.\ \bar{T}_i^k = \mathsf{nil}) \wedge (\bar{T}_i^i = \mathsf{NULL} :: \mathsf{nil} \vee (\bar{T}_i^i = \mathsf{nil} \wedge T_i = \mathsf{NULL}))$
(i has set its reception field and is in the main loop of acquire)

$C_{i,j,r} \equiv L_i \in [78;82] \wedge j = \mathsf{j}_i^{72} \wedge r = \mathsf{r}_i^{73} \leq R_j^j$
(i has made an answer to j at round r and is waiting for an answer)

$E_{i,j,t} \equiv L_i \in [78;87] \wedge i \neq j \wedge j = \mathsf{j}_i^{72} \wedge (\forall k \neq j.\ \bar{T}_i^k = \mathsf{nil}) \wedge ((\bar{T}_i^j = \mathsf{t} :: \mathsf{nil} \wedge \mathrm{H}) \vee (\bar{T}_i^j = \mathsf{nil} \wedge T_i = t))$
where H asserts that the last write in $\bar{T}_i^j$ occured before the write of any value greater than r_i^{73} in $\vec{R}_j^j$
(i has obtained the task t from j but has not yet pushed it its in deque)

$E_i \equiv \exists jt.\ E_{i,j,t}$

Fence-free sender-initiated work stealing

$\mathcal{I}_{A_1} \equiv$ If A_i is false then $\forall k.\ \bar{T}^i_k = \mathsf{nil}$.

$\mathcal{I}_{A_2} \equiv$ 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.

$\mathcal{I}_{R_1} \equiv$ $\forall j \neq i.\ \ \bar{R}^i_j = \mathsf{nil}$ (meaning that processors update only their own round numbers).

$\mathcal{I}_{R_2} \equiv$ $\vec{R}^i_i$ is a list of stricly decreasing values (meaning that round numbers only increase through time).

$\mathcal{I}_{Q_1} \equiv$ All the queries in $\bar{Q}^i_j$ 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 $\vec{Q}^i_j$ 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 $\vec{Q}^k_j$ 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 $\vec{Q}^i_j$ and it occured 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 $\mathsf{r}^{73}_i < R_j$.

$\mathcal{I}_{T_3} \equiv$ 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 on x86-TSO

- Model: one buffer per cell and per core, plus time constraints.
- Schema: state the invariant, prove stability, prove correctness.
- Applications: concurrent data structures, scheduling algorithms, algorithms exploiting races in nontrivial ways, . . .

Future work

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