Concurrent programming: From theory to practice

Concurrent Algorithms 2016
Tudor David
From theory to practice

Theoretical (design)  Practical (design)  Practical (implementation)
From theory to practice

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs
- Correctness

Design (pseudo-code)
From theory to practice

Theoretical (design)
- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs
- Correctness

Practical (design)
- System models
  - shared memory
  - message passing
- Finite memory
- Practicality issues
  - re-usable objects
- Performance

Practical (implementation)

Design (pseudo-code)

Design (pseudo-code, prototype)
## From theory to practice

<table>
<thead>
<tr>
<th>Theoretical (design)</th>
<th>Practical (design)</th>
<th>Practical (implementation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Impossibilities</td>
<td>- System models</td>
<td>- Hardware</td>
</tr>
<tr>
<td>- Upper/Lower bounds</td>
<td>- shared memory</td>
<td>- Which atomic ops</td>
</tr>
<tr>
<td>- Techniques</td>
<td>- message passing</td>
<td>- Memory consistency</td>
</tr>
<tr>
<td>- System models</td>
<td>- Finite memory</td>
<td>- Cache coherence</td>
</tr>
<tr>
<td>- Correctness proofs</td>
<td>- Practicality issues</td>
<td>- Locality</td>
</tr>
<tr>
<td>- Correctness</td>
<td>- re-usable objects</td>
<td>- Performance</td>
</tr>
<tr>
<td></td>
<td>- Performance</td>
<td>- Scalability</td>
</tr>
</tbody>
</table>

- Design (pseudo-code)
- Design (pseudo-code, prototype)
- Implementation (code)
Example: linked list implementations
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- Performance: Programming techniques
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- Performance: Programming techniques
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns

- Cache
  - Large = slow
  - Medium = medium
  - Small = fast
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Core → L3 = ~20ns
  - Core → L2 = ~7ns
  - Core → L1 = ~1ns
Typical server configurations

- **Intel Xeon**
  - 12 cores @ 2.4GHz
  - L1: 32KB
  - L2: 256KB
  - L3: 24MB
  - Memory: 256GB

- **AMD Opteron**
  - 8 cores @ 2.4GHz
  - L1: 64KB
  - L2: 512KB
  - L3: 12MB
  - Memory: 256GB
Experiment
Throughput of accessing some memory, depending on the memory size
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- Performance: Programming techniques
Until ~2004: Single-cores

- Core freq: 3+GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → L3
  - Core → L2
  - Core → L1
After ~2004: Multi-cores

- Core freq: ~2GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → shared L3
  - Core → L2
  - Core → L1
Multi-cores with private caches

Core 0

L1

L2

Core 1

L1

L2

Private

L3

Memory

Disk

= multiple copies
Cache coherence for consistency

Core 0 has X and Core 1

- wants to write on X
- wants to read X
- did Core 0 write or read X?
Cache-coherence principles

- To perform a **write**
  - invalidate all readers, or
  - previous writer
- To perform a **read**
  - find the latest copy
Cache coherence with MESI

- A state diagram
- State (per cache line)
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data
The ultimate goal for scalability

- **Possible states**
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data

- **Which state is our “favorite”?**
The ultimate goal for scalability

- Possible states
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data

= threads can keep the data close (L1 cache)
= faster
Experiment
The effects of false sharing
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- Performance: Programming techniques
Uniformity vs. non-uniformity

- Typical **desktop** machine

  ![Diagram of a typical desktop machine showing uniform memory access]

  = Uniform

- Typical **server** machine

  ![Diagram of a typical server machine showing non-uniform memory access (NUMA)]

  = non-Uniform (NUMA)
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data

Memory

C

L1

1

L2

7

L3

20

Memory

C

L1

L2

C

L1

L2

L3

30
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data

Conclusion: we need to take care of locality
Experiment
The effects of locality
Outline

- CPU caches
- Cache coherence
- Placement of data
- **Hardware synchronization instructions**
- Correctness: Memory model & compiler
- Performance: Programming techniques
The Programmer’s Toolbox: Hardware synchronization instructions

- Depends on the processor
- **CAS generally provided 😊**
- TAS and atomic increment not always provided
- x86 processors (Intel, AMD):
  - Atomic exchange, increment, decrement provided
  - Memory barrier also available
- Intel as of 2014 provides **transactional memory**
Example: Atomic ops in GCC

```c
void __sync_fetch_and_OP(type *ptr, type value);
void __sync_OP_and_fetch(type *ptr, type value);
// OP in \{add, sub, or, and, xor, nand\}

void __sync_val_compare_and_swap(type *ptr, type oldval, type newval);
void __sync_bool_compare_and_swap(type *ptr, type oldval, type newval);

__sync_synchronize(); // memory barrier
```
Intel’s transactional synchronization extensions (TSX)

1. **Hardware lock elision (HLE)**
   - Instruction prefixes:
     
     ```
     XACQUIRE
     XRELEASE
     ```

   Example (GCC):
   ```
   __hle_{acquire,release}_compare_exchange_n{1,2,4,8}
   ```

   - Try to execute critical sections without acquiring/releasing the lock
   - If conflict detected, abort and acquire the lock before re-doing the work
2. **Restricted Transactional Memory (RTM)**

```c
_xbegin();
_xabort();
_xtest();
_xend();
```

**Limitations:**

- Not starvation free
- Transactions can be aborted for various reasons
- Should have a non-transactional back-up
- Limited transaction size
2. Restricted Transactional Memory (RTM)

Example:

```c
if (xbegin() == XBEGIN_STARTED){
    counter = counter + 1;
    xend();
} else {
    __sync_fetch_and_add(&counter,1);
}
```
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- Performance: Programming techniques
Concurrent algorithm correctness

• Designing correct concurrent algorithms:
  1. Theoretical part
  2. Practical part $\rightarrow$ involves implementation

The processor and the compiler optimize assuming no concurrency!

😊
The memory consistency model

//A, B shared variables, initially 0;
//r1, r2 – local variables;

P1
A = 1;
r1 = B;

P2
B = 1;
r2 = A;

What values can r1 and r2 take?
(assume x86 processor)

Answer:
(0,1), (1,0), (1,1) and (0,0)
The memory consistency model

→ The order in which memory instructions appear to execute

What would the programmer like to see?

**Sequential consistency**

All operations executed in some sequential order;
Memory operations of each thread in program order;
Intuitive, but limits performance;
The memory consistency model

How can the processor reorder instructions to different memory addresses?

x86 (Intel, AMD): TSO variant

- Reads not reordered w.r.t. reads
- Writes not reordered w.r.t. writes
- Writes not reordered w.r.t. reads
- Reads may be reordered w.r.t. writes to different memory addresses

```c
//A,B,C
//globals
...
int x,y,z;
x = A;
y = B;
B = 3;
A = 2;
y = A;
C = 4;
z = B;
...
```
The memory consistency model

- **Single thread** – reorderings transparent;
- **Avoid reorderings**: memory barriers
  - x86 – implicit in atomic ops;
  - “volatile” in Java;
  - Expensive - use only when really necessary;
- **Different processors** – different memory models
  - e.g., ARM – relaxed memory model (anything goes!);
  - VMs (e.g. JVM, CLR) have their own memory models;
Beware of the compiler

- The compiler can:
  - reorder instructions
  - remove instructions
  - not write values to memory

```c
volatile int the_lock=0;

void lock(int * some_lock) {
  while (CAS(some_lock,0,1) != 0) {}
  asm volatile("" ::: ""memory"); //compiler barrier
}

void unlock(int * some_lock) {
  asm volatile("" ::: ""memory"); //compiler barrier
  *some_lock = 0;
}
```

C "volatile" !=
Java “volatile”
Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- **Performance: Programming techniques**
What techniques can we use to speed up our concurrent application?

Main idea: Minimize contention on cache lines

Use case: Locks
  - acquire() = lock()
  - release() = unlock()
TAS – The simplest lock

Test-and-Set Lock

typedef volatile uint lock_t;

void acquire(lock_t * some_lock) { 
    while (TAS(some_lock) != 0) {} 
    asm volatile("" ::: "memory");
}

void release(lock_t * some_lock) { 
    asm volatile("" ::: "memory"); 
    *some_lock = 0;
}
How good is this lock?

• A simple benchmark
• Have 48 threads continuously acquire a lock, update some shared data, and unlock
• Measure how many operations we can do in a second

Test-and-Set lock: 190K operations/second
How can we improve things?

Avoid cache-line ping-pong:
Test-and-Test-and-Set Lock

```c
void acquire(lock_t * some_lock) {
    while(1) {
        while (*some_lock != 0) {} // Test and Set
        if (TAS(some_lock) == 0) {
            return;
        }
    }
    asm volatile("" ::: "memory");
}

void release(lock_t * some_lock) {
    asm volatile("" ::: "memory");
    *some_lock = 0;
}
```
Performance comparison

Ops/second (thousands)

Test-and-Set

Test-and-Test-and-Set
But we can do even better

Avoid thundering herd:
Test-and-Test-and-Set with Back-off

```c
void acquire(lock_t * some_lock) {
    uint backoff = INITIAL_BACKOFF;
    while(1) {
        while (*some_lock != 0) {}  
        if (TAS(some_lock) == 0) {
            return;
        } else {
            lock_sleep(backoff);
            backoff=min(backoff*2,MAXIMUM_BACKOFF);
        }
    }
    asm volatile("" ::: "memory");
}

void release(lock_t * some_lock) {
    asm volatile("" ::: "memory");
    *some_lock = 0;
}
```
Performance comparison

Ops/second (thousands)

Test-and-Set
Test-and-Test-and-Set
Test-and-Test-and-Set w. backoff
Are these locks fair?

**Processed requests per thread, Test-and-Set lock**

- **Thread number**
- **Number of processed requests**
What if we want fairness?

Use a FIFO mechanism: Ticket Locks

typedef ticket_lock_t {
    volatile uint head;
    volatile uint tail;
} ticket_lock_t;

void acquire(ticket_lock_t * a_lock) {
    uint my_ticket = fetch_and_inc(&(a_lock->tail));
    while (a_lock->head != my_ticket) {}  
    asm volatile("" ::: "memory");
}

void release(ticket_lock_t * a_lock) {
    asm volatile("" ::: "memory");
    a_lock->head++;
}
What if we want fairness?

Processed requests per thread, Ticket Locks

Thread number

Number of processed requests
Performance comparison

Ops/second (thousands)

- Test-and-Set
- Test-and-Test-and-Set
- Test-and-Test-and-Set w. backoff
- Ticket
Can we back-off here as well?

Yes, we can:
Proportional back-off

```c
void acquire(ticket_lock_t * a_lock) {
    uint my_ticket = fetch_and_inc(&(a_lock->tail));
    uint distance, current_ticket;
    while (1) {
        current_ticket = a_lock->head;
        if (current_ticket == my_ticket) break;
        distance = my_ticket - current_ticket;
        if (distance > 1)
            lock_sleep(distance * BASE_SLEEP);
    }
    asm volatile("" ::: "memory");
}

void release(ticket_lock_t * a_lock) {
    asm volatile("" ::: "memory");
    a_lock->head++;
}
```
Performance comparison
Still, everyone is spinning on the same variable….

Use a different address for each thread: Queue Locks

Use with care:
1. storage overheads
2. complexity
Performance comparison

Ops/second (thousands)

Test-and-Set
Test-and-Test-and-Set
Test-and-Test-and-Set w. backoff
Ticket
Ticket w. backoff
Queue lock
To summarize on locks

1. Reading before trying to write
2. Pausing when it’s not our turn
3. Ensuring fairness (does not always bring ++) 
4. Accessing disjoint addresses (cache lines)

More than 10x performance gain!
Conclusion

- Concurrent algorithm design
  - Theoretical design
  - Practical design (may be just as important)
  - Implementation

- You need to know your hardware
  - For correctness
  - For performance