High-level C++ Implementation of the Read-Copy-Update Pattern

Gábor Márton[1] Imre Szekeres[2] Zoltán Porkoláb[1]

[1]Faculty of Informatics
Eotvos Lorand University

[2]Budapest University of Technology and Economics

2017

Agenda

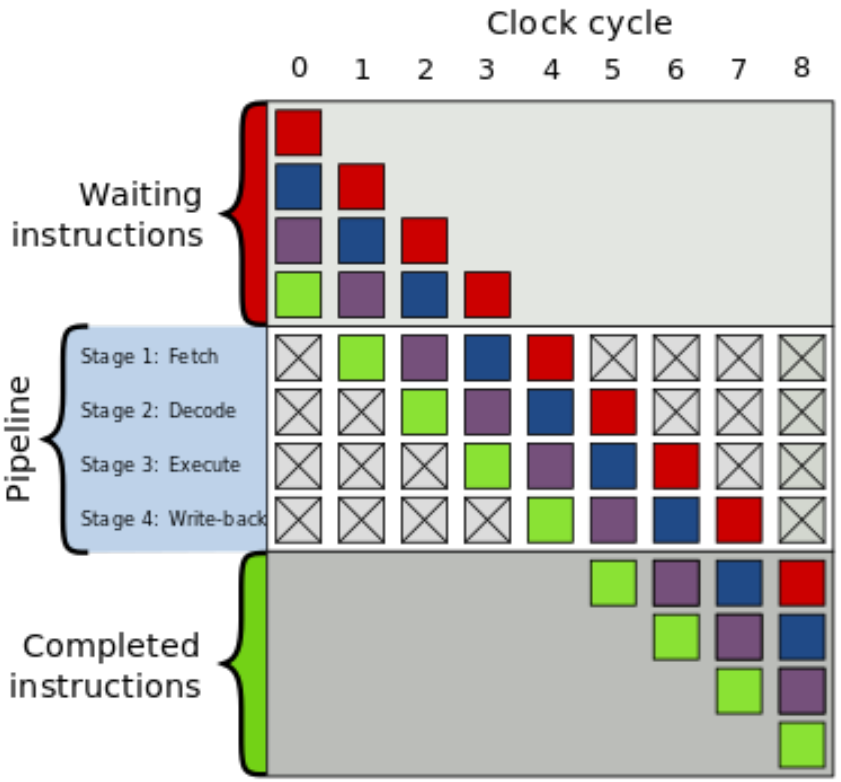
- Problem definition
- Hardware memory model
- C++ memory model
- Mutex
- Readers-writers Lock
- High level RCU

```
class X {
  std::vector<int> v;
  mutable std::mutex m;

public:
  int sum() const { // read operation
    std::lock_guard<std::mutex> lock{m};
    return std::accumulate(v.begin(), v.end(), 0);
  }
  void add(int i) { // write operation
    std::lock_guard<std::mutex> lock{m};
    v.push_back(i);
  }
};
```

Parallelism on single core

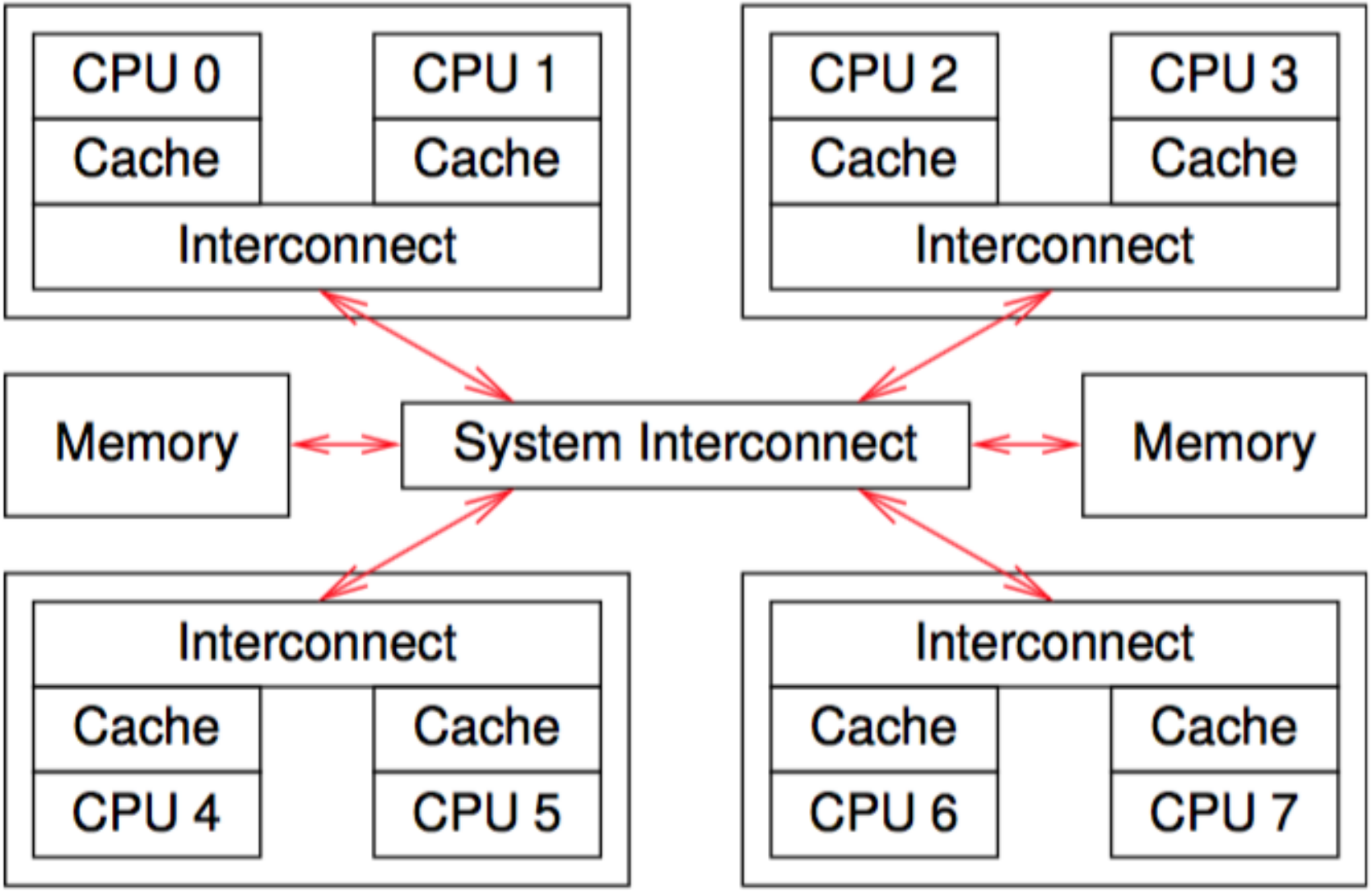
- Pipeline



source: wikipedia

- Out-of-order execution

Parallelism in SMP



source: Paul McKenny, Perfbook

Parallelism in SMP

- Loads reordered after loads
- Loads reordered after stores
- Stores reordered after stores
- Stores reordered after loads
- Atomic reordered with loads
- Atomic reordered with stores
- Dependent loads reordered (DEC alpha)

Compiler reordering (optimizing compilers)

- More available registers
- Better data locality

Locking (spin lock)

```
class spinlock {
private:
  typedef enum {Locked, Unlocked} LockState;
  boost::atomic<LockState> state_;

public:
  spinlock() : state_(Unlocked) {}

  void lock()
  {
    while (state_.exchange(Locked, boost::memory_order_acquire) == Locked) {
      /* busy-wait */
    }
  }
  void unlock()
  {
    state_.store(Unlocked, boost::memory_order_release);
  }
};
```

source: boost

C++ memory model

- What is `memory_order_acquire` and `memory_order_release`?

Hardware barriers

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

source: preshing.com

Acquire-release

source: preshing.com

Sequential consistency

- What is sequential consistency?
- Happens before relation?
  Let A and B represent operations performed by a multithreaded process. If A **happens-before** *B, then the memory effects of A effectively become visible to the thread performing B before B is performed.*
- Synchronizes with relation?

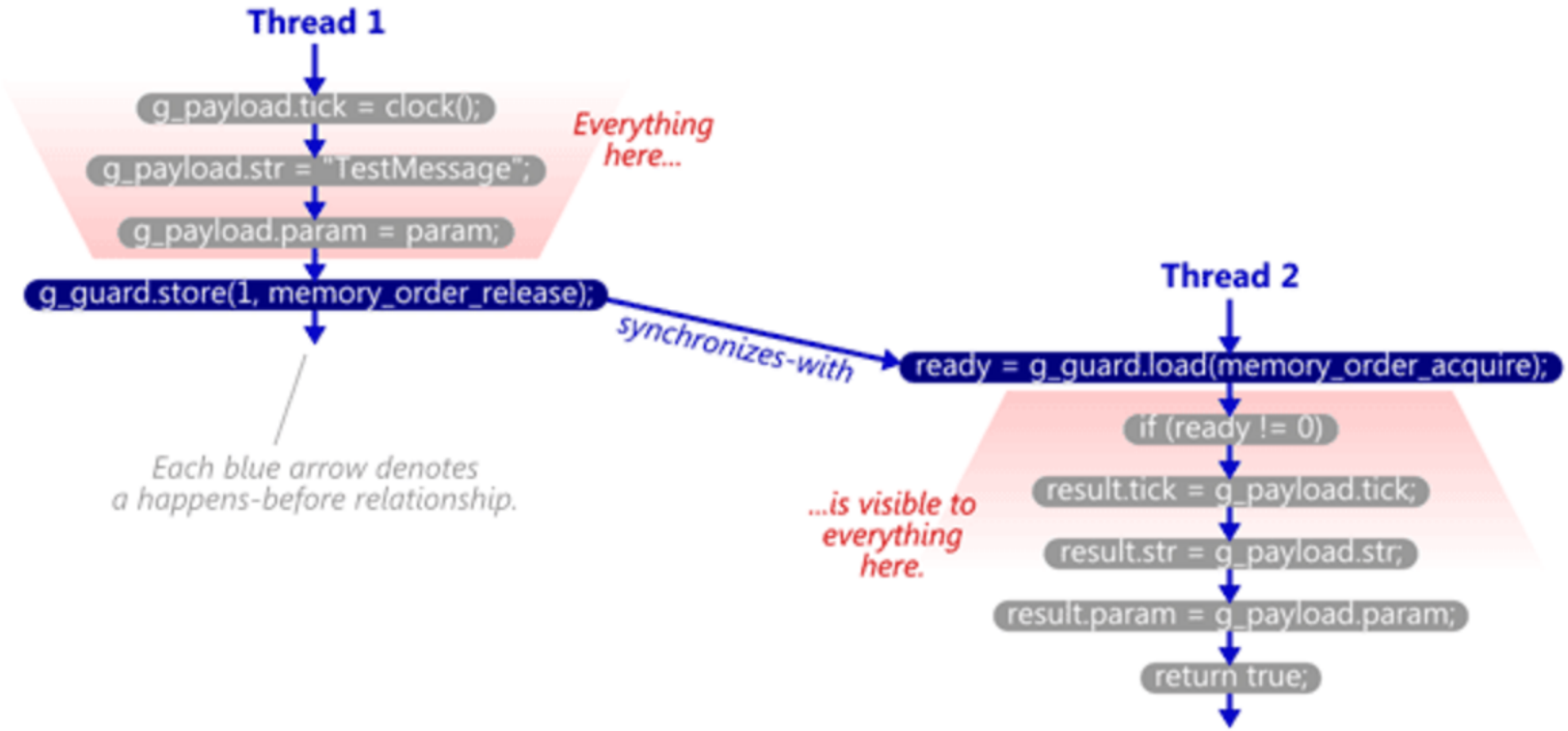
Happens before, really?

```
1   int A = 0;
2   int B = 0;
3
4   void foo()
5   {
6       A = B + 1;              // (1)
7       B = 1;                  // (2)
8   }
```

```
mov     eax,DWORD PTR [rip+0x0]
mov     DWORD PTR [rip+0x0],0x1
add     eax,0x1
mov     DWORD PTR [rip+0x0],eax
ret
```

source: preshing.com

Synchronizes with



source: preshing.com

Mutex

- What is a mutex?
- How is it different than a spin lock?
- When to use a spin lock?

Consume-release

- Data dependency ordering `p->a`
- DEC Alpha
- RCU (Linux kernel)

```
class X {
  std::vector<int> v;
  mutable std::mutex m;

public:
  int sum() const { // read operation
    std::lock_guard<std::mutex> lock{m};
    return std::accumulate(v.begin(), v.end(), 0);
  }
  void add(int i) { // write operation
    std::lock_guard<std::mutex> lock{m};
    v.push_back(i);
  }
};
```

Readers-writer lock

Begin Read

- Lock r.
- Increment b.
- If $b = 1$, lock g.
- Unlock r.

End Read

- Lock r.
- Decrement b.
- If $b = 0$, unlock g.
- Unlock r.

Begin Write

- Lock g.

End Write

- Unlock g.

source: wikipedia

Readers-writer lock

- Readers exclude writer
- Lock contention (locking frequency)?
- Possible deadlock
- MEASURE!

```
class X {
  std::shared_ptr<std::vector<int>> v;
  mutable std::mutex m;
public:
  X() : v(std::make_shared<std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<std::vector<int>> local_copy;
    {
      std::lock_guard<std::mutex> lock{m};
      local_copy = v;
    }
    // assume processing the data takes longer than copying it
    return std::accumulate(local_copy->begin(), local_copy->end(), 0);
  }
  void add(int i) { // write operation
    std::shared_ptr<std::vector<int>> local_copy;
    {
      std::lock_guard<std::mutex> lock{m};
      local_copy = v;
    }
    local_copy->push_back(i);
    {
      std::lock_guard<std::mutex> lock{m};
      v = local_copy;
    }
  }
};
```

```
void add(int i) { // write operation
  std::shared_ptr<std::vector<int>> local_copy;
  {
    std::lock_guard<std::mutex> lock{m};
    local_copy = v;
  }
  auto local_deep_copy =
      std::make_shared<std::vector<int>>(*local_copy);
  local_deep_copy->push_back(i);
  {
    std::lock_guard<std::mutex> lock{m};
    v = local_deep_copy;
  }
}
```

```
class X {
  std::shared_ptr<std::vector<int>> v;

public:
  X() : v(std::make_shared<std::vector<int>>()) {}
  int sum() const { // read operation
    auto local_copy = std::atomic_load(&v);
    return std::accumulate(local_copy->begin(), local_copy->end(), 0);
  }
  void add(int i) { // write operation
    auto local_copy = std::atomic_load(&v);
    auto exchange_result = false;
    while (!exchange_result) {
      // we need a deep copy
      auto local_deep_copy =
          std::make_shared<std::vector<int>>(*local_copy);
      local_deep_copy->push_back(i);
      exchange_result = std::atomic_compare_exchange_strong(
          &v, &local_copy, local_deep_copy);
    }
  }
};
```

Pointer to const

```
class X {
  std::shared_ptr<const std::vector<int>> v;
  // ...
};
```

Move

```
exchange_result =
  std::atomic_compare_exchange_strong(
    &v, &local_copy, std::move(local_deep_copy));
```

Strong vs weak CAS

```
atomic_compare_exchange_strong
atomic_compare_exchange_weak
```

Maintenance and productivity

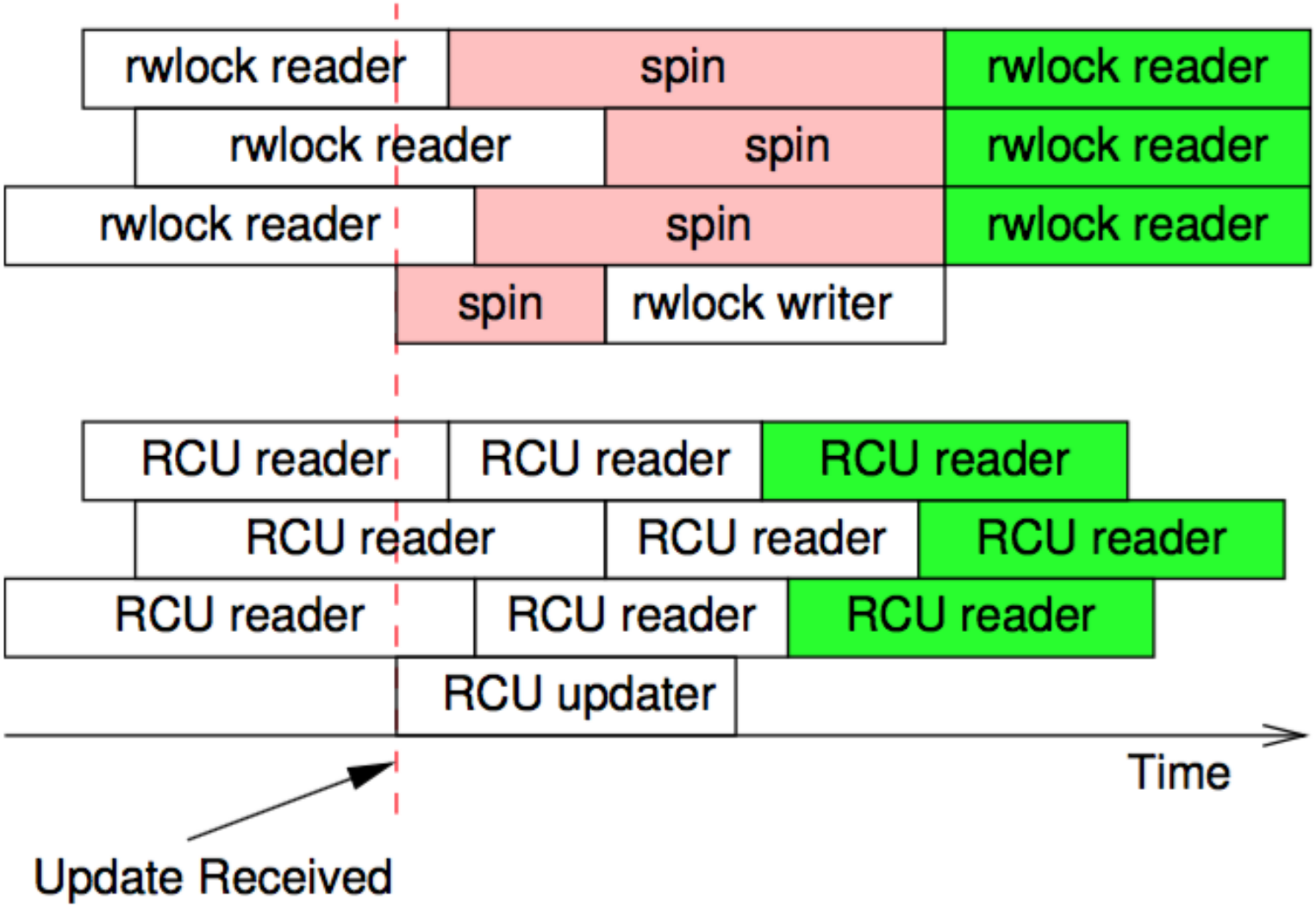
```
int another_sum() const {
    return std::accumulate(v->begin(), v->end(), 0);
}
```

rcu_ptr

```
class X {
  rcu_ptr<std::vector<int>> v;

public:
  X() : v(std::make_shared<std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<const std::vector<int>> local_copy = v.read();
    return std::accumulate(local_copy->begin(), local_copy->end(), 0);
  }
  void add(int i) { // write operation
    v.copy_update(
        [i](std::vector<int> *copy) { copy->push_back(i); });
  }
};
```

RCU vs readers-writer lock



source: Paul McKenny, Perfbook

- Anthony Williams, Concurrency in Action
- Jeff Preshing, preshing.com
- Paul McKenny, Perfbook

- martongabesz@gmail.com
- https://github.com/martong/rcu_ptr