Computer Architecture

计 算 机 体 系 结 构

第20讲：TLP（6）

张献伟

xianweiz.github.io

DCS3013, 12/12/2022

Review Questions

- Coherence vs. consistency?

  Same vs different location, eventually vs when, cache vs. mem, …
- Consistency model?

  Memory ordering, W→R, W→W, R→W, R→R
- Sequential consistency?

  Maintain all four orderings, i.e., the program order.
- Pros and cons of SC?

  Simple paradigm, limited performance.
- Total Store Ordering?

  Relax W→R. Use store buffer, i.e. R can start before W done.
- Partial Store Ordering?

  Atop of TSO, further relax W→W.

Aggressive Memory Ordering???

- SC maintains all four memory operation orderings
- Certain orderings can be violated ???
  - W→R: store buffer to allow read execute earlier
  - W→W: reorder writes in the store buffer
    - Earlier write is a cache miss, later is a hit
  - R→W, R→R: processor may reorder independent instructions
    - Out-of-order execution
  - Note that all are valid optimizations if a program consists of a single instruction stream[对单线程都有效]

- What if we discard all four memory orderings?
  - Still a memory consistency model (**Release Consistency**)

Release Consistency[RC一致性]

- Release Consistency (RC)
  - Processors support special synchronization operations
  - Memory accesses before memory fence instruction must complete before the fence issues
  - Memory accesses after fence cannot begin until fence instruction is complete
  - 硬件不再对一致性做过多保证，需要软件介入以控制执行行为

```
reorderable reads and writes here
...
MEMORY FENCE
...
reorderable reads and writes here
...
MEMORY FENCE
```

Express Synchronization[同步]

- ’00’ is not allowed in SC (the example)
  - But suppose architecture is of RC model, how to get the same effect with SC (i.e., no ‘00’)?
- All modern architectures include **synchronization** operations to bring their relaxed memory models under control when necessary
  - Most common operation: barrier (or fence)

- A barrier inst forces all memory operations before it to complete before any memory operation after it can begin
  - I.e., a barrier inst effectively reinstates SC at a particular point in program execution

Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

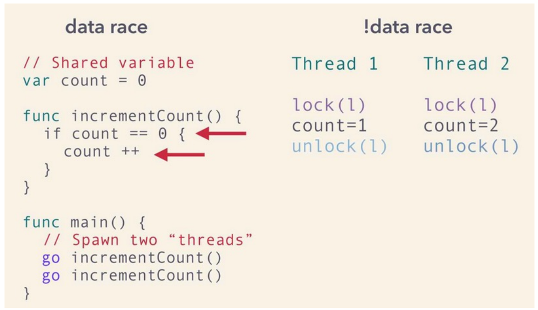
```
(3) B = 1
(4) print(A)
```

```
S1: Store x = NEW;  | S2: Store y = NEW;
FENCE               | FENCE
L1: Load r1 = y;    | L2: Load r2 = x;
```

FENCE: S1/S2 must be completely done before L1/L2

Synchronized Programs[同步程序]

- Two memory accesses by different processors conflict if
  - They access the same memory location
  - At least one is a write



data race

```
// Shared variable
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}

func main() {
  // Spawn two "threads"
  go incrementCount()
  go incrementCount()
}
```

!data race

```
Thread 1     Thread 2

lock(l)      lock(l)
count=1      count=2
unlock(l)    unlock(l)
```

- Unsynchronized program
  - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  - Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)
- In practice, most programs are synchronized (via locks, barriers, etc. implemented in synchronization libraries)
- Synchronized programs <u>yield SC results on non-SC systems</u>[程序在Relaxed Consistency上跑和在SC上跑结果一样]
  - Synchronized programs are data-race-free (DRF)

Summary: Relaxed Consistency

- Motivation: obtain higher performance by allowing reordering of memory operations (reordering is not allowed by SC)
  - Relaxed consistency models differ in which memory ordering constraints they ignore (e.g., TSO, PSO, RC)
- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed
  - Optimize for the common case: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are
  - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)

Compiler Reordering

- Besides hardware, compilers can also reorder memory operations
  - Example: the program prints a string of 100 ‘1’s (always)
  - Possible to optimize the code?
    - Loop-invariant code motion: move the write outside the loop
    - Dead store elimination: remove X = 0
  - These two programs are totally equivalent
    - Produce the same output

```
X = 0
for i in range(100):
  X = 1
  print X
```

```
~~X = 0~~
X = 1
for i in range(100):
  print X
```

```
X = 1
for i in range(100):
  print X
```

Compiler Reordering (cont.)

- Now suppose there's another thread running in parallel with the program, and it performs a single write to X
  - The first program
    - It can print strings like 11101111 …, so long as there's only one single zero (because it will reset X = 1 on the next iteration)
  - The second program
    - It can print strings like 1110000 …, where once it starts printing 0s it never goes back to 1s
  - The first can never print 1110000…; the second cannot print 11011111…

With parallelism, the compiler optimization no longer produces an equivalent program.

```
X = 0
for i in range(100):
  X = 1
  print X
```

```
X = 1
for i in range(100):
  print X
```

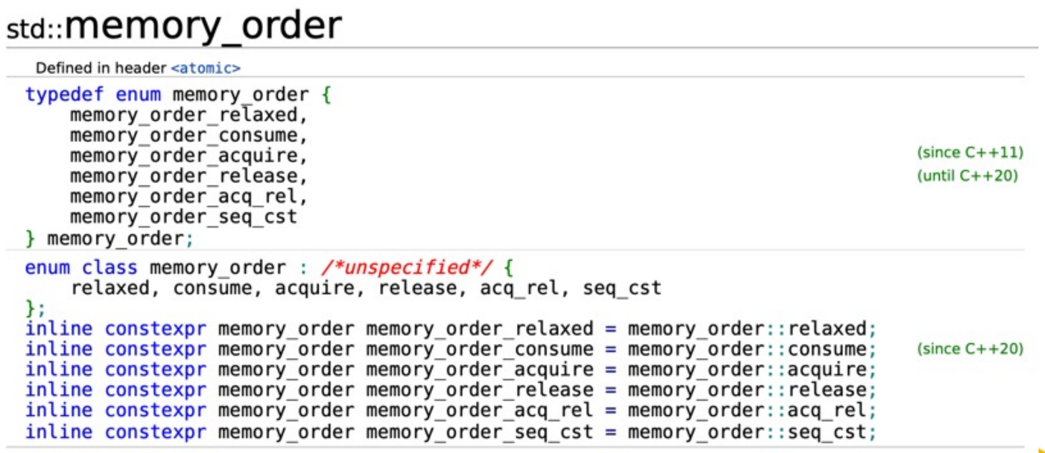
```
X = 0
```

```
X = 0
```

Languages' Memory Models[语言内存模型]

- The compiler optimization is effectively reordering
  - It's rearranging (and removing some) memory accesses in ways that may or may not be visible to programmers
- To preserve intuitive behavior, programming languages need memory models of their own,
  - To provide a contract to programmers about how their memory operations will be reordered
- Memory consistency at the program level

The memory model means that C++ code now has a **standardized library** to call <u>regardless of who made the compiler and on what platform it's running</u>. There's a standard way to control how different threads talk to the processor's memory.


std::memory_order

Defined in header <atomic>

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,                                                   (since C++11)
    memory_order_release,                                                   (until C++20)
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
enum class memory_order : /*unspecified*/ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;   (since C++20)
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

C++ Atomic[原子操作]

- Multithreading: concurrency, data race, thread sync
- Synchronization primitives: synchronize code to avoid race conditions in multithreading
  - std::**mutex**: very annoying, be cautious of deadlock
  - std::**atomic**: lock-free, efficient, usually for variables

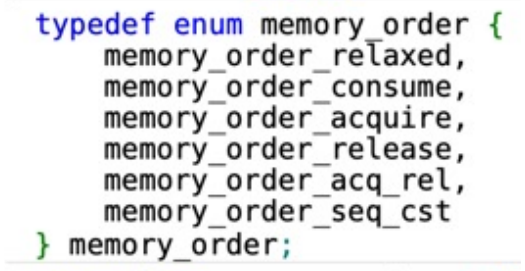
```
std::mutex mtx;
int num = 0;

void inc() {
  std::lock_guard<std::mutex> lock(mtx);
  num++;
}

int main() {
  std::thread t1(inc), t2(inc);
  return 0;
}
```

```
std::atomic<int> num = 0;

void inc() {
  num++;
}

int main() {
  std::thread t1(inc), t2(inc);
  return 0;
}
```

https://www.internalpointers.com/post/lock-free-multithreading-atomic-operations
http://www.max-sperling.bplaced.net/?p=1759

C++ Memory Model

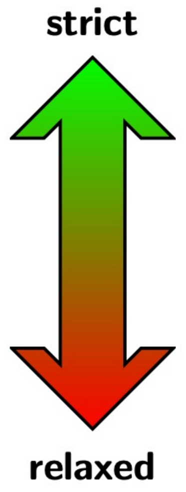
- The six memory orders can be combined with each other to achieve three ordering models
  - **Sequential consistent** ordering: achieves synchronization and guarantees a single total order
    - memory_order_seq_cst
  - **Acquire-release** ordering: implements synchronization, but does not guarantee global order consistency
    - memory_order_acquire / load
    - memory_order_release / store
    - memory_order_acq_rel / load, store, read-modify-write
    - memory_order_consume
  - **Relaxed ordering**: does not implement synchronization, but only guarantees atomicity
    - memory_order_relaxed



```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```

https://www.sobyte.net/post/2022-06/cpp-memory-order/

C++ Memory Model (cont.)



seq_cst	total order (SC for DRF code)
release / acquire	message passing
release / consume	message passing
relaxed	no synchronization

枚举值	定义规则
memory_order_relaxed	不对执行顺序做任何保证
memory_order_consume	本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行
memory_order_acquire	本线程中，所有后续的读操作必须在本条原子操作完成后执行
memory_order_release	本线程中，所有之前的写操作完成后才能执行本条原子操作
memory_order_acq_rel	同时包含memory_order_acquire和memory_order_release标记
memory_order_seq_cst	全部存取都按顺序执行

https://kernelgo.org/memory-model.html

Example

```
std::atomic<bool> x{false}, y{false};

void thread1() {
   x.store(true, std::memory_order_relaxed); // (1)
   y.store(true, std::memory_order_relaxed); // (2)
}

void thread2() {
   while (!y.load(std::memory_order_relaxed)); // (3)
   assert(x.load(std::memory_order_relaxed)); // (4)
}
```

no determined order between (1) and (2)

when loop exits, y has been true. I.e., (2) happened;
but possible that (1) has not been done → (4) may fail

```
std::atomic<bool> x{false}, y{false};

void thread1() {
   x.store(true, std::memory_order_relaxed); // (1)
   y.store(true, std::memory_order_release); // (2)
}

void thread2() {
   while (!y.load(std::memory_order_acquire)); // (3)
   assert(x.load(std::memory_order_relaxed)); // (4)
}
```

(1) happens before (2):
if (2) is visible, then all before release are visible

when loop exits, y has been true. I.e., (2) happened
(3) before (4) → (1) before (4) → (4) never fails

Summary of TLP

- Multiprocessors with thread-level parallelism
  - Sharing memory, having private caches
- Cache coherence
  - **Snooping**: every cache block is accompanied by the sharing status of that block
    - All cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary
  - **Directory-based**: a single location (directory) keeps track of the sharing status of a block of memory
    - Reduce storage and communication overheads
- Memory consistency
  - **Sequential consistency**: maintains all four memory operation orderings (W→R, R→R, R→W, W→W)
  - **Relaxed consistency**: allows certain orderings to be violated
    - TSO, PSO, RC

Computer Architecture

计 算 机 体 系 结 构

第20讲：WSC & Interconnect

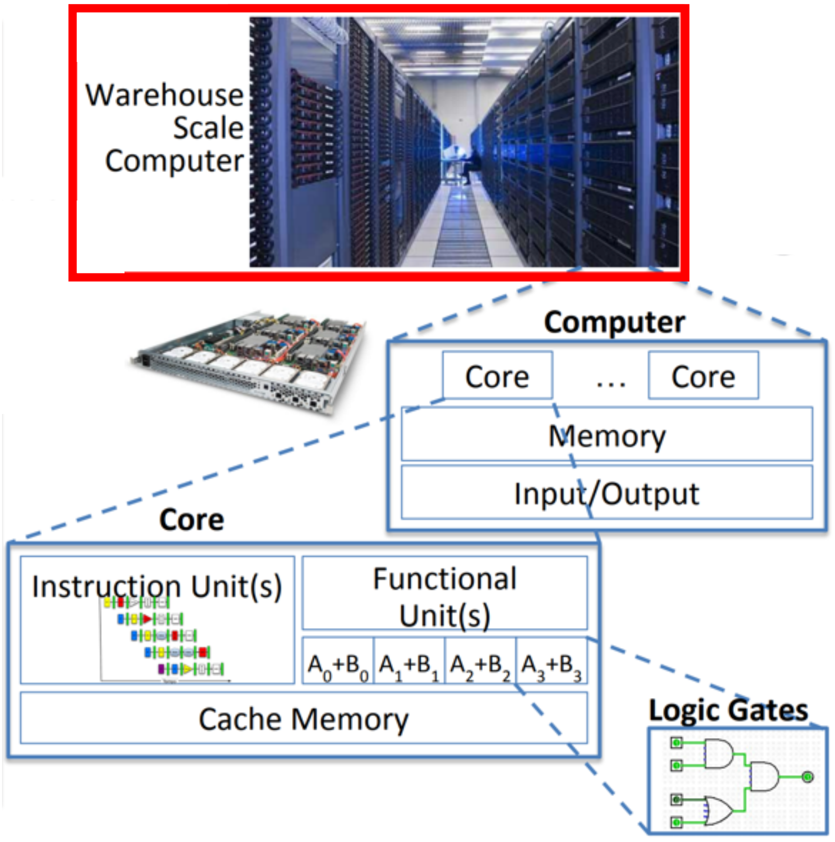
张献伟

xianweiz.github.io

DCS3013, 12/12/2022

Parallelism[并行]

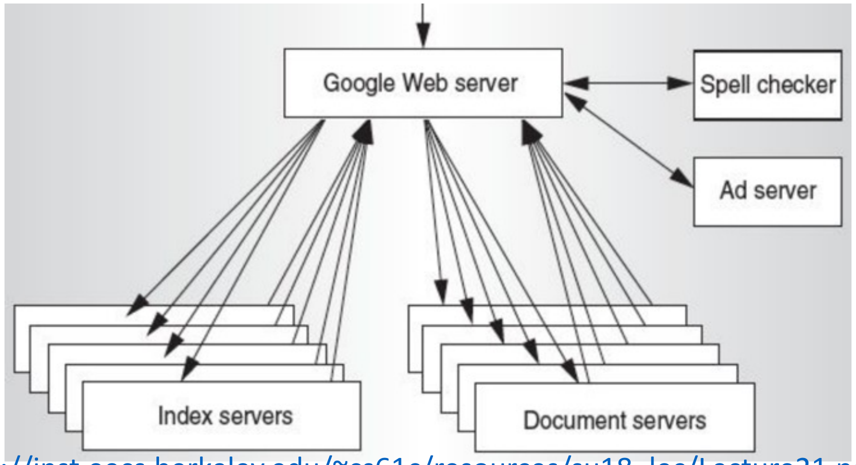
- Instruction-level parallelism (ILP)
  - Pipelining, speculation, OoO, …
- Data-level parallelism (DLP)
  - Vectors, GPU, AVX, …
- Thread-level parallelism (TLP)
  - Multithreading, multi-cores
- Request-level parallelism (RLP)
  - Parallelism among multi decoupled tasks



https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture21.pdf

Request-level Parallelism[请求级并行]

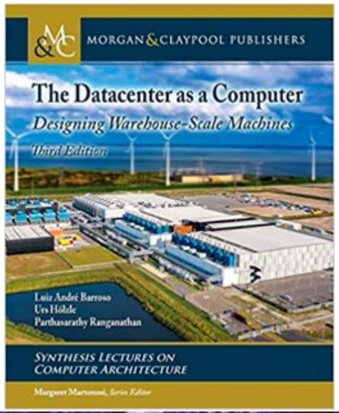
- Hundreds or thousands of requests per second
  - Not your laptop or cell-phone, but popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve strict read–write data sharing or synchronization across requests
- Computation easily partitioned within a request and across different requests



https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture21.pdf

Luiz André Barroso

- Google Fellow & former VP of Engineering
- ACM-IEEE CS Eckert-Mauchly Award
  - For pioneering the design of **warehouse-scale computing** and driving it from concept to industry
- Award lecture:
  - Q: What to focus on while educating the next generation of computer engineers and scientists?
  - A: … **being a good programmer** is really important, doesn't matter what you are where you are in the field of computer science; make sure you are **graduating good programmers** whatever that means for you …



MORGAN & CLAYPOOL PUBLISHERS

The Datacenter as a Computer

Designing Warehouse-Scale Machines

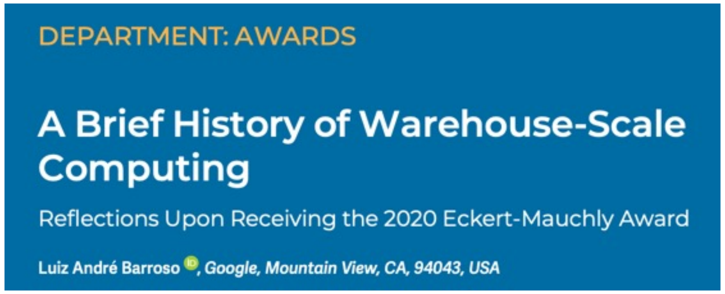
Third Edition

Luiz André Barroso
Urs Hölzle
Parthasarathy Ranganathan

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, Series Editor

of them being a good programmer is
really important doesn't matter what you



DEPARTMENT: AWARDS

A Brief History of Warehouse-Scale Computing

Reflections Upon Receiving the 2020 Eckert-Mauchly Award

Luiz André Barroso, *Google, Mountain View, CA, 94043, USA*

https://www.youtube.com/watch?v=Lv_eZX99lUU