ECE 587 – Hardware/Software Co-Design
Lecture 08 Concurrency in Practice II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

February 10, 2025

Reading Assignment

- This lecture: Memory Ordering
- Next lectures: RISC-V and Chipyard

Outline

Memory Ordering

Memory Barrier

Observable State Transitions

- We assume each process in a process-based model is specified by a state-based model, e.g. a sequential program.
- For shared memory communication, processes react by observing changes in the shared states.
  - It is possible to implement shared memory communication on top of message passing and vice versa.
  - So here we refer to the shared memory communication as is.
- Are the observed state transitions the same as the state-based model specify in practice?
  - State bits that are not observable (from the shared memory).
  - Missing transitions.
  - Reordered transitions.
- Let's look at multi-threaded programs on multi-core processors.

Single Core Performance

- Each processor core can issue an instruction per clock cycle: fraction of a nano-second (ns).
- Memory access latency: about 10 ns
- There must be optimizations or the core has to wait for memory.
  - Compiler
  - Micro-architecture
  - Cache

Compiler Optimization

- Registers
  - Once a variable is available from a register, there is no need to read it from the memory.
  - A variable in a register need to be written to the memory only if the register need to be used for another variable.
  - Registers are not observable from other processes.
- Operations can be reordered as long as there is no data or control dependency.
  - Think of DFG.
- If the above are true for any single process, what about observed state transitions?
  - Is it possible/*correct* for a variable to stay only in the register?
  - Is it possible/*correct* for a variable to have two different values, one in the register and one in the memory?

Micro-Architecture

- While a core may issue an instruction per cycle, the instruction may take several cycles to complete.
  - Lengthy computations.
  - Wait for data to be available from memory.
- Out-Of-Order (OOO) execution
  - Allow later instructions to execute as long as there is no data/control dependencies.
  - A load/store buffer is usually necessary to buffer memory writes and to forward memory reads for better performance.
  - Re-order buffer (ROB) helps to commit intructions in order, giving the illusion that they are executed in order.
- Out-Of-Order memory updates
  - After the instruction is committed, for correctness on a single core, it doesn't matter when the load/store buffer actually updates the memory.
  - Similar effects as compiler optimization.

Cache

- Cache brings data closer to the processor core.
  - No need to go to (main) memory if the data is available from the cache.
- There are multiple copies of the same data in the memory hierarchy.
  - Main memory.
  - Multiple levels of cache.
  - If there are many cores, each core may have its own cache.
  - The load/store buffer on each core.

Cache Coherence

- Ensure cores to have consensus on the value of a particular memory location.
- MESI protocol
  - State-based, each core maintain a FSM per cache line.
  - 4 states: Modified, Exclusive, Shared, Invalid
  - Shared read or exclusive write per cache line.
- If the variable fits into a cache line, then writes/reads to it are ordered sequentially across cores, though not deterministic.
- Contention happens when many cores attempt to update the same cache line simultaneously.

When Intuition Fails ...

Core 1

```
data = 100;
ready = 1;
```

Core 2

```
while (ready == 0)
  wait_a_while();
print(data);
```

- Assume initially `ready=0` and `data=0`.
- What could happen?
  - Assume a cache coherence protocol is used.
  - Does it matter if you turn on/off compiler optimizations?

Outline

Memory Ordering

Memory Barrier

Memory Barrier

- Cache coherence ensures ordering of memory access to the *same* memory location (cache line).
    - But not to *multiple* memory locations.
- Memory barriers provide guarantees of ordering among accesses to *multiple* memory locations.
- A joint effort of compiler and core micro-architecture.
    - Need language support to guide compiler optimization.
    - Compiler further translates them into corresponding special instructions that force updates in the memory hierarchy.

Example: Use a Channel

```
int main() {
  channel<std::string> chan;

  std::thread p([&] {
    chan.put("hello");
    chan.put("world");
  });

  std::thread c([&] {
    printf("%s\n", chan.get().c_str());
    printf("%s\n", chan.get().c_str());
  });

  p.join(); c.join();
  return 0;
}
```

- Message passing via channel.
- How to implement the channel via shared memory?

A Simple Channel Implementation

```
template <class T>
class channel {
  std::atomic<bool> empty_;
  T val_;

public:
  channel() {
    empty_.store(true, std::memory_order_relaxed);
  }
  ...
}; // class channel<T>
```

- `std::memory_order_relaxed` prevents any compiler optimization and ensures atomic updates.
- Thread creation works as full memory barrier so both threads will see `empty_` as `true` initially.

A Simple Channel Implementation (Cont.)

```
template <class T>
class channel {
  ...
  T get() {
    while (empty_.load(std::memory_order_acquire))
      ;
    T ret = val_;
    empty_.store(true, std::memory_order_release);
    return ret;
  }
  void put(const T &val) {
    while (!empty_.load(std::memory_order_acquire))
      ;
    val_ = val;
    empty_.store(false, std::memory_order_release);
  }
}; // class channel<T>
```

- `store` with `std::memory_order_release` pairs with `load` with `memory_order_acquire`.
  - Prevent compiler optimization.
  - Preceding stores will be seen by loads afterwards.

Discussions

- Memory barriers usually work in pairs to establish the so-called *happens-before* relations to ensure visibility of memory updates.
- There are memory barriers other than the release-acquire pair, depending on processor architectures.
- OS synchronizations (thread creation, lock, etc.) work as full memory barrier where all ordering with reference to them are preserved across all threads.
- When contention is low, since our channel doesn't use any OS synchronization that may potentially suspend the thread, the communication latency could be very small.
- How to implement a channel that can hold more than one token, or can support multiple producers?

Further Readings

- Java Concurrency in Practice, Goetz et al., 2006.
- C++ Concurrency in Action, Williams, 2012.
- Is Parallel Programming Hard, And, If So, What Can You Do About It? `https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`