CS3350B
Computer Architecture
Winter 2015

Lecture 7.2: Multicore TLP (1)

Marc Moreno Maza

www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on
Computer Organization and Design,
Patterson & Hennessy, 4th or 5th edition, 2011]
Multiprocessor (Multiple Instruction Multiple Data): a computer system with at least 2 processors

- Deliver high throughput for independent jobs via job-level parallelism on top of ILP
- Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Now Use term core for processor (“Multicore”) because “Multiprocessor Microprocessor” too redundant
Review

- Sequential software is slow software
  - **SIMD** and **MIMD** only path to higher performance

- Multiprocessor (Multicore) uses Shared Memory (single address space) (SMP)

- Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern

- MESI Protocol ensures cache consistency and has optimizations for common cases.
Multiprocessors and You

- Only path to performance is **parallelism**
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b 2011, 512b in 2014?, 1024b in 2018?
    - **Advanced Vector Extensions** are 256-bits wide!
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...

- A **key challenge** is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that **scale**
  - Scheduling, load balancing, time for synchronization, overhead for communication
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - **Phase I:**
    - Partition 1000 numbers per processor;
    - Initial summation on each processor
      
      \[
      \text{sum}[Pn] = 0; \quad // \quad 0 \leq Pn \leq 99
      \text{for} \quad (i = 1000*Pn; \quad
      i < 1000*(Pn+1); \quad i = i + 1) \quad
      \text{sum}[Pn] = \text{sum}[Pn] + A[i];
      \]

- **Phase II:** Add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to **synchronize** between reduction steps
Example: Sum Reduction

Second Phase:
After each processor has computed its “local” sum

This code runs simultaneously on each core

```
half = 100;
repeat
    synch();
    /*Proc 0 sums extra element if there is one */
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];

    half = half/2; /* dividing line on who sums */
    if (Pn < half)
        sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
An Example with 10 Processors


half = 10
half = 5
half = 2
half = 1
Threads

- **thread of execution**: smallest unit of processing scheduled by operating system

- Threads have their own *state* or *context*:
  - Program counter, Register file, Stack pointer,

- Threads share a memory address space

- Note: A “process” is a heavier-weight construct, which has its own address space. A process typically contains one or more threads.
  - Not to be confused with a *processor*, which is a physical device (i.e., a core)
CAN BE SPECIFIED IN A LANGUAGE WITH MIMD SUPPORT – such as OpenMP and CilkPlus
Multithreading

- On a single processor, multithreading occurs by time-division multiplexing:
  - Processor switched between different threads
    - may be “pre-emptive” or “non pre-emptive”
  - Context switching happens frequently enough that user perceives threads as running at the same time

- On a multiprocessor, threads run at the same time, with each processor running a thread
Multithreading vs. Multicore

- Basic idea: Processor resources are expensive and should not be left idle

- For example: Long latency to memory on cache miss?
  - Hardware switches threads to bring in other useful work while waiting for cache miss
    - Cost of thread context switch must be much less than cache miss latency

- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers, …

- Attractive for applications with abundant TLP
Data Races and Synchronization

- Two memory accesses form a **data race** if from different threads, to same location, and *at least one is a write*, and they occur one after another.

- If there is a data race, result of program can vary depending on chance (which thread ran first?)

- Avoid data races by **synchronizing** writing and reading to get **deterministic** behavior.

- Synchronization done by user-level routines that rely on **hardware synchronization** instructions.
Question: Consider the following code when executed *concurrently* by two threads.

What possible values can result in \(*($s0)*\)?

```assembly
# *(s0) = 100
lw $t0,0($s0)
addi $t0,$t0,1
sw $t0,0($s0)
```

☐ 101 or 102
☐ 100, 101, or 102
☐ 100 or 101
☐ 102
Lock and Unlock Synchronization

- Lock used to create region (critical section) where only one thread can operate

- Given shared memory, use memory location as synchronization point: lock, semaphore or mutex

- Thread reads lock to see if it must wait, or OK to go into critical section (and set to locked)
  
  \[
  0 \Rightarrow \text{lock is free / open / unlocked / lock off}
  \]
  
  \[
  1 \Rightarrow \text{lock is set / closed / locked / lock on}
  \]

---

**Set the lock**

Critical section
(only one thread gets to execute this section of code at a time)

e.g., change shared variables

**Unset the lock**
Possible Lock Implementation

- **Lock (a.k.a. busy wait)**

  Get_lock:
  ```
  addiu $t1,$zero,1  # t1 = Locked value
  ```

  Loop:
  ```
  lw $t0,0($s0)  # load lock
  bne $t0,$zero,Loop  # loop if locked
  ```

  Lock:
  ```
  sw $t1,0($s0)  # Unlocked, so lock
  ```

- **Unlock**

  Unlock:
  ```
  sw $zero,0($s0)
  ```

- **Any problems with this?**
Possible Lock Problem

- **Thread 1**
  ```assembly
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock: sw $t1,0($s0)
  ```

- **Thread 2**
  ```assembly
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock: sw $t1,0($s0)
  ```

Both threads think they have set the lock! Exclusive access not guaranteed!
Hardware-supported Synchronization

- Hardware support required to prevent interloper (either thread on other core or thread on same core) from changing the value
  - **Atomic read/write** memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - e.g., **atomic swap** of register ↔ memory
  - or an atomic pair of instructions
Synchronization in MIPS

- **Load linked**: `ll rt, off(rs)`
  
  Load `rt` with the contents at `Mem(off+rs)` and reserves the memory address `off+rs` by storing it in a special link register (`R_{link}`).

- **Store conditional**: `sc rt, off(rs)`
  
  Check if the reservation of the memory address is valid in the link register. If so, the contents of `rt` is written to `Mem(off+rs)` and `rt` is set to 1; otherwise no memory store is performed and 0 is written into `rt`.

  - Returns 1 (success) if location has not changed since the `ll`
  - Returns 0 (failure) if location has changed

- **Note that `sc` clobbers the register value being stored (`rt`) !**
  
  - Need to have a copy elsewhere if you plan on repeating on failure or using value later
Synchronization in MIPS Example

- **Atomic swap** (to test/set lock variable)

  Exchange contents of register and memory:  
  \( \$s4 \leftrightarrow \text{Mem}(\$s1) \)

  ```
  try: add $t0,$zero,$s4  #copy value
  li $t1,0($s1)  #load linked
  sc $t0,0($s1)  #store conditional
  beq $t0,$zero,try  #loop if sc fails
  add $s4,$zero,$t1  #load value in $s4
  ```

  **sc** would fail if another thread executes **sc** here
Test-and-Set

- In a single atomic operation:
  - **Test** to see if a memory location is set (contains a 1)
  - **Set** it (to 1) if it isn’t (it contained a zero when tested)
  - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions

- Useful for implementing lock operations
Test-and-Set in MIPS

- Single atomic operation
- Example: MIPS sequence for implementing a T&S at ($s1)

Try:

```
addiu $t0,$zero,1
ll $t1,0($s1)
bne $t1,$zero,Try
sc $t0,0($s1)
beq $t0,$zero,Try
```

Locked:

```
critical section
sw $zero,0($s1)
```
Summary

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance

- Multiprocessor (Multicore) uses Shared Memory (single address space)

- Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern

- Synchronization via hardware primitives:
  - MIPS does it with Load Linked + Store Conditional