CIS501: Multicore

Shreyas S. Shivakumar

December 10, 2019

1 Lecture Notes

1. Why *Multicore*?

Consider the following *example* (double precision a × x + y):

```
void daxpy(): {
  for(i=0;i<SIZE;i++) {
    z[i] = a*x[i] + y[i]
  }
}
```

- Every iteration in the loop is *independent*.
- Each iteration is about a dozen instructions (daxpy).
- With our best methods so far (supserscalar, out-of-order, pipeline) we can maybe exploit 4× (4 wide) or 8× (8 wide) parallelism at best.
- But if we run this loop 10,000 times, we technically have 10,000 way parallelism which we cannot support with the above methods.

Q: How would you do this? What is the individual ***workload*** *for each core?*

The simple way to think about it is to: break up the entire loop into **N** chunks and allocate them to **N** cores. These cores will have both *private* as well as *public* variables and in the simplest assumption $SIZE$ is a multiple of N.

N threads are then spawned which will handle $\frac{SIZE}{N}$ of the overall workload.

Q: What is an easy way to do this in C++?

OpenMP allows programmers to define structures in their code that can be *parallelized*. However, these pieces of code must actually be inherently parallel otherwise unpredicted behavior may result.

2. Energy Efficiency

Another reason why *multicore* design is popular is because it is more *energy efficient*.

Relationship: Energy consumption does not scale *linearly* with clock frequency. It is almost *cubic*.

For *example*, going from a single 1 GHz core to a single 200 MHz core *reduces* energy utilization by 30×. And if you had 5 × 200 MHz cores, you could still save $\frac{5}{30} = \frac{1}{6}$ of the energy when compared to a single 1 GHz core.

3. Amdahl's Law

As we've seen before, this only applies to programs that have portions that can be parallelized. And these portions need to be significant enough to invest the time and effort into optimizing them.

$$Speedup \ = \ \frac{S + P}{S + \frac{P}{N}}$$

Where S is the serial portion, P is the parallel portion and N is the number of cores.

4. Threading

With a single processor (*uni-processor design*), there are independent flows of execution. The different threads have *shared* state such as global variables, heaps etc. They also share the same *memory space*. These threads are managed by the *operating system*.

Programmers will usually explicitly create multiple threads, and all **loads** and **stores** go to a single *Shared Memory* space.

This can create non-deterministic behavior such as if **Thread 1** executes (a) store 1 $\rightarrow$ y and (b) load x and **Thread 2** executes (a) store 1 $\rightarrow$ x and (b) load y.

5. A Simple Design

A simple *Multiprocessor Design* is to replicate the entire pipeline except **share the caches**.

The main invariant here is:

LOADs must return the value written by the most recent STORE

6. What are the *problems* that can arise with *Shared Memory* designs?

- Cache Coherence - Each *core* will have it's own *private cache*, so how do we make one core's *write* to one cache show up in another's cache?
- Parallel Programming - How is the parallelism expressed?
- Synchronization - How do we *regulate* access to shared data and implement *locks*?
- Memory Consistency Models - What kind of *contract* can we design for the *Shared Memory* structure to keep the programmer sane?

7. Cache Coherence - *Introduction*

In our simple multiprocessor design, what happens if we decide not to share the L1 cache across different processors?

Let's have a per-processor **private cache** instead. If left unchecked, naively this solution will cause inconsistencies across elements in all the different private caches so a Cache Coherence Protocol must also be introduced that will define how they co-ordinate with each other.

The *Shared Memory Invariant* of **Loads read the value written by the most recent Store** must still hold.

Design Decisions:

- If there is just a single level of *Shared Cache*, all the different processors will have to interact with the single cache using an interconnect and this can create a *bottleneck* at the *Shared Cache*. Instead, each processor is given a small *Private Cache* in addition to the *Shared Cache*. The *Shared Cache* exists right above the *Memory* layer abstraction.
- The *Shared Cache* contains a **State** bit which is either *DIRTY* or *CLEAN*. For *example*, if the Shared Cache contains a cache line that was updated by some processor, the *State Bit* of that cache line is changed to *DIRTY* from *CLEAN* and depending on the policy the *Shared Cache* state is either **Written-Back** or **Written-Through** to the Memory.
- Similarly, each processor's *Private Caches* will also have **State** bits that do the same at a processor / core level.

Q: What if a processor P_1 update's it's copy of a cache line in it's private cache (It's state bit is now $DIRTY$) and another processor P_0 tries to load that value from Memory?

There are now 3 versions of that cache line element, i.e one version that lives in *Memory*, one version that lives in the *Shared Cache* with state bit $CLEAN$ and one version that lives in P_1's *Private Cache* with state bit $DIRTY$. Which of these values will P_0 read and which of these values **should** P_0 read? If it reads from the *Shared Cache* it will get a stale value. This is the **Cache Coherence** problem.

Coherence *is a property of the* ***Private Caches*** *since it must be enforced only at the private cache level to guarantee consistency.*

8. VI Protocol – *Valid* / *Invalid*

Idea: Track every copy of each block by specifying the owner of that block in the *Shared Cache*. This can be done by adding an **Owner** field to the *Shared Cache* table.

In the above situation, when P_0 wants to $LOAD$ $[A]$ and if there are 3 different *versions* of A, the *Shared Cache Owner* entry will indicate who owns the latest / updated version of A (in our case P_1) and then A is removed from P_1's private cache and transferred to P_0's private cache and the *Shared Cache Owner* field is updated to indicate that P_0 is the new owner of cache line A.

The *Tracking of Ownership* is used to **Validate (V)** or **Invalidate (I)** a specific cache line and hence maintain consistency.

Discussion:

- There can only be at most **one valid** "copy" of the block.
- If a processor / core wants a "copy", it must find it an invalidate it so that it will then have the only valid "copy".
- *Problem:* What if multiple processors / cores want to just read this block and not make any changes to it? In this design that won't be possible.
- *On a cache miss, how is the valid copy found?* One approach is to use **broadcasting / snooping** and the other is to use a **directory** based method such as the one we previously discussed with the ownership field. The directory based method is more intuitive and commonly used.

9. MSI Protocol – *Modified / Shared / Invalid*

Idea: What if we allowed multiple processors access to *READ* but only allowed a single processor access to *WRITE*. This can be done by adding a **State** bit field to each processors *Private Cache* as well as a **Sharer** field to the *Shared Cache*, similar to the ownership field from the VI Protocol.

In this mechanism, there are three possible states:

- **Modified (M):** If a processor has this state, it can read / write to a cache line. However, only one processor can access the cache line, i.e the processor that is in state M.
- **Shared (S):** If a processor has this state, it can only read a cache line. Unlike the previous VI Protocol, multiple processors can be in state S allowing for multiple processors to read from a given cache line.
- **Invalid (I):** Similar to the VI Protocol this is an invalid state and the processor has no permissions over that cache line.
- *Note:* A cache line can only be in **either** modified (M) or shared (S) state and not both. If it is in both, this implies that one processor could be writing to a line while another is reading from that line and hence result in inconsistent values.

Coherence Miss:

If a processor P_0 and P_1 has a cache line A that is currently in state S and if P_0 comes along and attempts to write to cache line A, P_1's cache line A will get **invalidated** and then P_0's state for cache line A will transition to **modified**.

This is unfortunate for P_1 since it's cache line was **externally invalidated** and this could lead to cache misses if P_1 tries to read A again. However this is the price to be paid for this design. **Increasing size** of the cache does not improve this.

These kind of misses are of two types:

- **Upgrade Miss:** If a processor has state S and needs to write to S such as from the point of view of processor P_0 in the above example.
- **Coherence Miss:** If a processor misses to a block that was evicted by another processor's request such as from the point of view of processor P_1 in the above example.

False Sharing:

This is a particular pattern of cache miss where two or more processors share parts of the same block. They don't share the same *bytes* within that block but just different parts of it. This will cause a *ping-pong* behavior between the two processors resulting in poor performance. This is very difficult to diagnose but a solution is to pad each block so that no sub-block regions can be shared.

10. MESI Protocol – *Modified / Exclusive / Shared / Invalid*

Consider the situation when one particular core / processor issues a $LOAD$ instruction followed by a $STORE$ instruction. This would result in an **upgrade miss** as well as a **coherence miss** even if the block is not shared such as in a single threaded core. And at the very least, it is not unreasonable to expect that a single threaded core should behave as well as it would on a uni-processor system.

Idea: Add another state **Exclusive (E)** which can be interpreted as *I have the only cached copy and it is* $CLEAN$.

Mechanism:

- If a processor has a $LOAD$ miss and that block currently has no $Sharers$, then that block can be granted $EXCLUSIVE$ (E) state in the processor's private cache.
- If a processor has a $LOAD$ miss but that block currently has other $Sharers$, then that block can only be granted a $SHARED$ (S) state.
- Here is where we see a performance gain: If that processor then issues a $STORE$ instruction to the same block while in the $EXCLUSIVE$ (E) state, it can be **instantaneously** changed to $MODIFIED$ (M) state.
- On *eviction*, if a block is $MODIFIED$ and $DIRTY$, it must be written back to the next level. If it is $EXCLUSIVE$, writing back is not necessary.

11. MOESI Protocol – *Modified / Ownership / Exclusive / Shared / Invalid*

A more popular and recent *cache coherence protocol* is $MOESI$. As before, in the *write* state, only one copy may be accessed at a time and if it is *clean*, it is granted the *Exclusive (E)* state. If it is *dirty* it is granted the *Modified (M)* state.

However, previously we did not consider the implications of the *clean* and *dirty* states for the *read* process, where multiple processors may be given access to a cache line. Previously we assigned both *clean* and *dirty* attributes to the *Shared (S)* state. With $MOESI$, we add an additional state called *Ownership (O)* which is when the cache line is read only but *dirty*. The implications here are that this processor is now responsible to **writeback** this line.

State	Clean	Dirty
One Copy (write)	Exclusive (E)	Modified (M)
> 1 Copy (read)	Shared (S)	Owned (O)

12. Coherence Protocols

- **Update Based** Cache Coherence: On the lines of a *Write Through* update to all caches – too much traffic and not commonly used anymore.
- **Invalidation Based** Cache Coherence: There are two approaches to the *invalidation* based method:
  (a) *Snooping* or *Broadcasting*: No explicit state but generates a lot of traffic since each cache request broadcasts it's requests to all the other levels. We will use a **Shared Bus** to interface the different private caches with the shared cache and memory.
  (b) *Directory Based*: This is what we've seen before where we track the different *sharers* of blocks and invalidate blocks accordingly. These designs can be either inclusive or exclusive. We will use an **End to End Interconnect** to interface the different private caches with the shared cache and memory.
- While the *Snooping* based method uses more bandwidth with *Shared* and *Exclusive* states, we can save 1 entire transaction / hop since processors can *interact* with each other by passing requests.
- The *Directory* based methods have lower bandwidth consumption, thus more scalable but individual latency is worse.

13. Coffman Conditions for Deadlock

- Mutual Exclusion
- Holding and Waiting
- No Pre-emption
- Circular Waiting

Note: If you can break **any one** of these conditions, you can prevent or stop a deadlock.

Note: Always acquire multiple locks in the same order. Acquiring locks in random order increases the risk of arriving at a deadlock condition - *Dining Philosophers*.

14. Atomics

- Compare And Swap (CAS)
- Test And Set (TAS)
- Test And Test And Set (TATAS)
- Queue Locks - *release in order*

15. Shared Memory Problems

If you have two thread *thread* 1 and *thread* 2; Thread 1 has the following instructions (a) $STORE\ 1 \rightarrow$ y (b) $LOAD$ x and Thread 2 has the following instructions (a) $STORE\ 1 \rightarrow$ x and (b) $LOAD$ y. Is there a possibility that at the end of execution you can read $(x = 0, y = 0)$? Intuitively you would think **No**, but this situation is possible.

16. Why reorder memory instructions?

In the **hardware**, memory instructions are re-ordered to hide write latency and to simplify out of order execution. In **compilers**, memory instructions are re-ordered in order to maximize performance during an optimization step.

17. Memory Consistency Models

A *Memory Consistency Model* specifies the semantics of shared memory operations, i.e they specify **what values a load may return** as a set of rules. This makes the life of a programmer easier as he/she is now aware of *what* the possible values might be.

(a) Sequential Consistency (SC): This model is *typically* what programmers expect. Here everybody agrees on the type of interleaving and it is indistinguishable from multi-programmed uni-processors.

(b) Total Store Order (TSO): Here more behaviors are allowed. It incorporates a *Store Buffer* and stores can be deferred but are put into the cache in order.

(c) Release Consistency (RC): Even more behaviors are allowed with this model. Here even the *Store Buffer* is un-ordered and stores can be put into the cache in any order.

$$SC \subset TSO \subset RC$$

With *Total Store Ordering (TSO)* the re-ordering of the following instruction sequence: *Operation* $1 \rightarrow$ *Operation* 2 is allowed if it is a $STORE \rightarrow LOAD$ instruction. If they both are writing to the same memory location, *bypassing* is required. An additional **memory barrier** or **fence** instruction is added.

Important:

Because of the *Store Buffer* and the re-ordering of the $STORE$ and $LOAD$ instruction sequence, a situation arises when the caches of both threads or processors contains the older value of the LOAD instruction and thus the previously mentioned condition where $(x = 0, y = 0)$ becomes possible. This is possible in $x86$, $SPARC$ and ARM.

With *Relaxed Consistency (RC)*, **ALL** re-orderings are permitted as long as they are **not** writing to the same memory location. While this may seem like it is allowing for more problems to occur, this actually simplifies out-of-order execution. Along with a *Load Queue (LQ)* we can allow these odd re-orderings and detect and recover from bad re-orderings if they do occur.

2 Textbook Notes

1. Introduction

- *Multiprocessor* software must be designed to work with a variable number of processors.
- The current *energy efficiency* problem indicates that future increase in performance will most likely come from *more processors per chip* rather than higher clock rates and improved CPI.
- *Job Level Parallelism* or *Process Level Parallelism* − Utilizing multiple processors by running independent programs simultaneously.
- *Shared Memory Processor (SMP)* − A parallel processor with a single address space, implying implicit communication with loads and stores. A more accurate term would have been *shared-address multiprocessor*.
- *Synchronization* − The process of coordinating the behavior of two or more processes, which may be running on different processors.
- *Lock* − A synchronization device that allows access to data to only one processor at a time.

2. Parallel Programming Challenges

- Scheduling
- Load Balancing
- Synchronization
- Communication Overhead
- Amdahl's Law

3. Hardware Multithreading

Increasing utilization of a processor by switching to another thread when one thread is stalled. *Hardware Multithreading* allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread.

The hardware must support the ability to change to a different thread relatively quickly. A *thread switch* **must** be more efficient than a *process switch*.

Fine Grained Multithreading − switches between threads on each instruction, resulting in interleaved execution of multiple threads. This is usually done in a round-robin fashion, skipping threads that are stalled at that time. This can hide throughput losses for short and long stalls but slows down the execution of individual threads since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-Graned Multithreading − switches threads only on costly stalls, such as second-level cache misses. This change results in being much less likely to slowing down individual threads such as in the previous example. It is limited by throughput losses, specially from shorter stalls.