Memory consistency models

Contents of Lecture 5

A simple cache

- What is needed is a small memory on the same chip as the processor.
- If we describe our **hardware** in C, then a CPU could look like:

```
typedef struct {
        int                 reg[32];
        int                 pc;
        struct {
                bool        valid;
                int         data;
                int         address;
        }                   cache_array[8];
} cpu_t;
```

Data and Address

- We have now a cache which can store eight popular words.
- The cache array contains eight pairs of **data** and **address**.
- There is also a boolean called **valid** which tells us whether the data and address are valid for that row.
- Suppose the compiler has decided that a global variable X should be put at the address 293, or 0x125, or 0001 0010 0101.

Using our Cache

- When the program (or CPU) wants to read variable X, it should check whether any of the eight rows has **valid** $=$ **true** and **address** $=$ **293**
- If the CPU found one such row (or, let's call it **line**), then the CPU can take the data from that line and avoid waiting for the slow memory! Great!
- We must call this event something: **a cache hit**
- It can save us 100 clock cycles.

Load Instruction

- ***In hardware all iterations are executed concurrently!!***
- The openmp directive is here to make you alert on that this is ***not*** a sequential loop.

```
case LD:    found = false;
            address = source1 + constant;
            #pragma omp parallel for
            for (i = 0; i < 8; i++) {
                    if (cache_array[i].valid &&
                            cache_array[i].address == address)
                            data = cache_array[i].data;
                            found = true;
                            break;
                    }
            }
```

Cache replacement

- Since the cache by definition is smaller than RAM memory it cannot contain everything
- Cache replacement refers to putting something else somewhere in the cache
- So the old data at that row will be replaced
- If the old data was modified, it needs to be written to RAM memory
- For simplicity in the next slide we always write to RAM memory

Load Continued

```
if (!found) {
    i = select_row();

    if (cache_array[i].valid) // save old data to memory
        memory[cache_array[i].address] = cache_array[i].data;

    // read our data from memory
    data = memory[address];

    // save our data in the cache
    cache_array[i].data = data;
    cache_array[i].address = address;
    cache_array[i].valid = true;
}
```

Similar for a Store

```
case ST:
    found = false;
    address = source2 + constant;
    data = source1;
    #pragma omp parallel for
    for (i = 0; i < 8; i++) {
        if (cache_array[i].valid &&
            cache_array[i].address == address) {
            cache_array[i].data = data;
            found = true;
            break;
        }
    }
    if (found)
        break;
```

Store Continued

```
i = select_row();
if (cache_array[i].valid)
    memory[cache_array[i].address] = cache_array[i].data;
cache_array[i].data = data;
cache_array[i].address = address;
cache_array[i].valid = true;
```

- Next time we want to read or write that variable it is likely that it will be found in the cache.

The Loop — isn't it slow?

- No, it doesn't exist!
- It only exists in the software model of the hardware.
- Recall: ***in hardware the loop is run in parallel.***
- In our case, there are eight so called **comparators** which compare the address requested with the address in its row and says **"here!"** if the addresses are equal and the valid bit is true.

Cache block

- Our cache only fetches one `int` from memory at a miss
- It is almost always better to fetch e.g. 8 or 16 ints at a time
- The number of bytes to transfer is called the cache block size
- As we will see later in the course, if the cache block size is too big, there is a risk that different threads accidently use the same cache block and disturb each other

Sequential consistency

- Sequential consistency (SC) was published by Leslie Lamport in 1979 and is the simplest consistency model.
- Lamport is also known for the $\LaTeX$ macros for $\TeX$
- In a later lecture we will look at his loop parallelization method
- Neither Java, Pthreads, nor C11/C++ require it. They work on relaxed memory models.
- Sequential consistency can be seen from the programmer ***as if*** the multiprocessor has no cache memories and all memory accesses go to memory, which serves one memory request at a time.
- This means that
    - program order for each processor is maintained, and
    - that all memory accesses made by all processors can be thought of as atomic (i.e. not overlapping).
- C11 first intended to use SC but switched to the C++ memory model

Definition of SC

- Lamport's definition: ***A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.***
- Consider the program execution by three threads:

```
int A = B = C = 0;

T1:                T2:                T3:
A = 1;
                   if (A)
                       B = 1;
                                      if (B)
                                          C = A;
```

- Since all memory accesses are atomic, writes are seen in the same order so T3 must read the value 1 when reading A.

Dekker's algorithm

```
bool    flag[2] = { false, false };
int     turn = 0;

void work(int i)
{
        while (true) {
                flag[i] = true;
                while (flag[!i]) {
                        if (turn != i) {
                                flag[i] = false;
                                while (turn != i)
                                        ;
                                flag[i] = true;
                        }
                }

                /* enter critical section */

                /* ... */

                /* leave critical section */

                turn = !i;
                flag[i] = false;

        }
}
```

- SC ensures that Dekker's algorithm works.

Implementing SC in a system without caches

- We will examine the effects of each of the following hardware optimizations:
    - write buffer with read bypass
    - overlapping writes
    - non-blocking reads
- As we will see, none of these can be used even if there are no caches.

Write buffer with read bypass

- Assume a bus-based multiprocessor.
- Since there are no caches, the write buffer, a FIFO queue, sits between the CPU and the bus interface.
- With read bypass, it is thus meant that a read skips the queue in the buffer and goes first to the bus before any write (to a different address).
- In Dekker's algorithm both CPUs can set their `flag[i]` to true and put that write into it's write buffer.
- Then the reading of the other thread's flag will bypass the write in the write buffer.
- When bypassing the old values of the `flag[!i]` can be returned (e.g. if there were other writes before in the buffers) and both can enter the critical section!

The read bypass destroys the atomicity and hence the sequential order.

Overlapping writes

- In a bus-based system, the FIFO write buffer queue ensures that all writes from the CPU are ordered.
- In a general topology (e.g. nodes in a ”chess board”), however, different nodes typically are located at different distances and writes easily can arrive in an order different from the program order.
- In the example with variables A, B, and C, the new value of B may reach T3 before A does which violates SC.

T2 should not be allowed to start its write to* `b` *before T3 has become aware of the write to* `a`*.

Non-blocking reads

- Consider the following example.
- With speculative execution and non-blocking reads, T2 can read the value of a before it leaves the while-loop, which violates SC.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        f = 1;
}
```

```
// called by T2
void w(void)
{
        while (!f)
                ;
        printf("a = %d\n", a);
}
```

Implementing SC in a system with caches

- Clearly we should not expect caches to help us making a computer support SC
- Instead, the three issues in systems without caches can violate SC also with caches.
- For example a read cache hit must not be allowed to precede a previous read miss.
- In addition, since there can be multiple copies of a variable, there must be a mechanism which controls which CPU is allowed to write to a variable.
- This mechanism is called a ***cache coherence protocol***.
- A cache coherence protocol has three main tasks, as we will see next.

Implementing SC with cache coherence protocols

1. At a write, the cache coherence protocol should either remove all other copies, including the memory copy, or send the newly written data to update each copy.
   - A protocol which uses the former technique is called a **write invalidate protocol** while the latter is called a **write update protocol**.
   - Which is best depends on the sharing behavior of the application but write invalidate is almost always better.
2. Detecting when a write has completed so that the processor can perform the next memory access.
3. Maintaining the illusion of atomicity — with memory in multiple nodes the accesses cannot be atomic but a SC machine must behave as if they are.

- Faster machines without SC also use cache coherence protocols but some rules are relaxed — and others added

Detecting write completion

- Consider a write to a memory location which is replicated in some caches.
- How can the writing processor know when it's safe to proceed?
- The write request is sent to the memory where the data is located.
- The memory knows which caches have a copy (recall from a previous lecture this information is stored in a directory, e.g. as a bit vector).
- The memory then sends either updates or invalidations to the other caches.
- The receiving caches then must acknowledge they have received the invalidation message from memory.
- The acknowledgement is typically sent to the memory and then when all acknowledgements have been received, a message is sent to the writing processor (actually, its cache) to tell it the write has completed.
- After that, the processor can proceed.

Write atomicity 1(2)

- There are two different problems:
  1. (1) Write atomicity for a particular memory location, i.e. ensuring all CPUs see the same sequence of writes to a variable.
  2. (2) Write atomicity and reading the modified value.
- For (1) it is sufficient to ensure that writes to the same memory location are serialized, but not for (2). See next page.
- The memory controller can easily enforce serialization.
- Assume writes from two different caches are sent to it.
- One of them must arrive first. The other can simply be replied to with a negative acknowledgement of ”try again later!”
- When the cache receives that message it will simply try again and after a while it will be its turn.

Write atomicity 2(2)

- Let us now consider (2): reading the modified value.
- A write has completed when all CPUs with a copy have been notified.
- However, if one CPU is allowed to read the written value before the write has completed, SC can be violated.

```
int A = B = C = 0;

T1:             T2:             T3:
A = 1;
                if (A)
                    B = 1;
                                if (B)
                                    C = A;
```

- Assume all variables are cached by all threads, and T2 reads A before T3 knows about the write.
- Then T2 can write to B which might be so close to T3 that T3 can read A from its cache before the invalidation of A reaches T3.
- The solution is to disallow any CPU from reading A before the write is complete, which can be implemented with a ”try again” reply to the cache (using a special so called transient state of that address)

Write atomicity in write update protocols

- Recall that in a write update protocol, instead of invalidating other copies, new values are sent to the caches which replicate the value.
- So A is sent to T2 and to T3.
- While it's tempting to read A for T2 it's not permitted to.
- In write update, the updates are done in two phases. First is the data sent, and all CPUs acknowledge they have received it. Second each CPU is sent a message that it may read the new data.
- There are other problems with write update as well, for instance updates may be sent to CPUs which no longer are interested in the variable, thus wasting network traffic.

Optimizing compilers and explicitly parallel codes

- We have now seen the restrictions on hardware so that it does not reorder memory accesses and thus violate SC.
- The same restrictions must be put on optimizing compilers.
- The compiler must preserve ”source code order” of all memory accesses to variables which may be shared — but not the order of stack accesses or other data known to be private to the thread.
- Examples of optimizations which cannot (in general) be used:
  - Register allocation
  - Code motion out of loops
  - Loop reordering
  - Software pipelining
- It’s easy to imagine that these restrictions will slow down SC.
- Before C11 the solution for C has often been to compile code for uniprocessors and use the `volatile` qualifier for shared data.
- Recall that volatile in C is different from volatile in Java!

Parallelizing compilers

- If the compiler is doing the parallelization, these restrictions don't apply since the compiler writer hopefully knows what she or he is doing!
- After this course, however, you will probably not have too high hopes for automatic parallelization, except for some numerical codes.
- In my view, parallelization of ”normal” programs needs so drastic changes to the source code that automatic tools hardly can do that very well.

Cache coherence protocol states

- Recall a block is e.g. 32 or 64 bytes and not only one variable.
- The cache coherence protocol maintains a state for each cache and memory block.
- The cache state can for instance be:
  - SHARED
  - INVALID
  - EXCLUSIVE — memory and this cache has a copy but it's not yet modified.
  - MODIFIED — only this cache has a copy and it's modified
- There are similar states for a memory block and also the bit-vector with info about which cache has a copy.
- In addition, a memory block can be in the transient state when acknowledgements are being collected, for instance.

Memory access penalty

- The time the processor is stalled due to waiting for the cache is called the memory access penalty.
- Waiting for read cache misses is difficult to avoid in any computer with caches
- Waiting for obtaining exclusive ownership of data at a write access is one of the disadvantages for SC
- How significant it is depends on the application
- Of course, once the CPU owns some data, it can modify it's cached copy without any further write access penalty, until some other CPU also wants to access that data, in which case the state becomes SHARED again.

Optimizing SC in hardware

- Data prefetch can either fetch data in shared or exclusive mode
- By prefetching data in exclusive mode, the long delays of waiting for writes to complete can possibly be reduced or eliminated.
- Since exclusive mode prefetching invalidates other copies it can also increase the cache miss rate.
- Somewhat more complex cache coherence protocols can monitor the sharing behavior and determine that it probably is a good idea to grant exclusive ownership directly instead of only a shared copy which is then likely followed by an ownership request.
- In superscalar processors it can be beneficial to permit speculative execution of memory reads. If the value was invalidated, the speculatively executed instructions (the read and subsequent instructions) are killed and the read is re-executed.

Optimizing SC in the compiler

- Another approach is to have a memory read instruction which requests ownership as well.
- This can be done easily in optimizing compilers but needs new instructions (one for each basic data type).
- It's very useful for data which moves from different caches and is first read and then written:

```
p.a += 1;
p.b += 1;
p.c += 1;
```

- Here the compiler can very easily determine that it's smarter to request ownership while doing the read.

Summary of sequential consistency

- Recall the two requirements of SC:
    1. Program order of memory accesses
    2. Write atomicity
- While SC is ”nice” since it’s easy to think about, it has some serious drawbacks:
    - The above two requirements...
    - ...which limit compiler and hardware optimizations, and...
    - introduce a write access penalty
- The write access penalty is due to the processor cannot perform another memory access before the previous has completed.
- This is most notable for writes, since read misses are more difficult to optimize away by any method
- We will next look at relaxed memory consistency models

Relaxed memory models

- Relaxed memory models do not make programming significantly more complicated.
- You need to follow additional rules about how to synchronize threads.
- C11/C++11, Pthreads and Java are specified for relaxed memory models.
- In relaxed memory models, both the program order of memory references and the write atomicity requirements are removed and are replaced with different rules.
- For compiler writers and computer architects this means that more optimizations are permitted.
- For programmers it means two things:
  1. you must protect data with special system recognized synchronization primitives, e.g. locks, instead of normal variables used as flags.
  2. your code will almost certainly become faster, perhaps by between 10 and 20 percent, due to eliminating the write access penalty.

System recognized locks

- Under SC, you normally must protect your data using locks to avoid data races
- However, there are programs in which data races are acceptable
- Data races are forbidden in C/C++ and result in undefined behaviour.
- Under SC you can write your own locks by spinning on a variable `int flag` as you wish.
- Under relaxed memory models you should use whatever the system provides.

Relaxed memory models

- Recall that relaxed memory models relax the two constraints of memory accesses: program order and write atomicity.
- There are many different relaxed memory models and we will look only at a few.
- We have the following possibilities of relaxing SC.
- Relaxing A to B program order: we permit execution of B before A.
  1. write to read program order to different addresses
  2. write to write program order to different addresses
  3. read to write program order to different addresses
  4. read to read program order to different addresses
  5. read other CPU's write early
  6. read own write early
- Different relaxed memory models permit different subsets of these.

Assumptions

- All writes will eventually be visible to all CPUs.
- All writes are serialized which can be done at the memory by letting one write be handled at a time — other writes must be retried
- Uniprocessor data and control dependences are enforced

Relaxing the write to read program order constraint

- Obviously different addresses are assumed!
- A read may be executed before a preceding write has completed.
- With it, Dekker's Algorithm fails, since both can enter the critical section.

```
bool    flag[2] = { false, false };
int     turn = 0;

void work(int i)
{
        for (;;) {
                flag[i] = true;
                while (flag[!i]) {
                        if (turn != i) {
                                flag[i] = false;
                                while (turn != i)
                                        ;
                                flag[i] = true;
                        }
                }

                /* critical section */

                turn = !i;
                flag[i] = false;

                /* not critical section */
        }
}
```

Some models which permit reordering a (write, read) pair

- Processor consistency, James Goodman (Univ. of Wisconsin)
- Weak Ordering, Michel Dubois (USC, Los Angeles)
- Release Consistency, Kourosh Gharachorloo (Stanford)
- IBM 370, IBM Power
- Sun TSO, total store ordering
- Sun PSO, partial store ordering
- Sun RMO relaxed memory order
- Intel X86

Additionally relaxing the write to write program order constraint

- Recall the program below.

```
int a, f;

// called by T1                // called by T2
void v(void)                   void w(void)
{                              {
        a = u();                       while (!f)
        f = 1;                                 ;
}                                      printf("a = %d\n", a);
                               }
```

- By relaxing the write to write program order constraint, the write to `f` may be executed by T_1 even before the function call to `u`, resulting in somewhat unexpected output.

Some models which permit reordering a (write, write) pair

- Weak Ordering
- Release Consistency
- IBM Power
- Sun PSO, partial store ordering
- Sun RMO relaxed memory order

Relaxing all memory ordering constraints

- The only requirements left is the assumption of uniprocessor data and control dependences.
- Models which impose no reordering constraints for normal shared data include:
  - Weak Ordering
  - Release Consistency
  - IBM Power
  - Sun RMO relaxed memory order
- These consistency models permit very useful compiler and hardware optimizations and both Java and Pthreads (and other platforms) require from the programmer to understand and use them properly!
- In the preceding example, the two reads by T_2 are allowed to be reordered.
- The obvious question then becomes: how can you write a parallel program with these memory models???

What do you say?

Special machine instructions for synchronization

- The short answer is that machines with relaxed memory models have special machine instructions for synchronization.
- Consider a machine with a **sync** instruction with the following semantics:
  - When executed, all memory access instructions issued **before** the sync must complete before the sync may complete.
  - All memory access instructions issued **after** the sync must wait (i.e. not execute) until the sync has completed.
  - Assume both T_1 and T_2 have cached a.

```
int a, f;

// called by T1            // called by T2
void v(void)               void w(void)
{                          {
        a = u();                   while (!f)
        asm("sync");                       ;
        f = 1;                     asm("sync");
}                                  printf("a = %d\n", a);
                           }
```

- With `asm` we can insert assembler code with gcc and most other compilers.
- The sync instructions are required, as explained next...

Sync instruction example 1(3)

```
int a, f;

// called by T1              // called by T2
void v(void)                 void w(void)
{                            {
        a = u();                     while (!f)
        asm("sync");                         ;
        f = 1;                       asm("sync");
}                                    printf("a = %d\n", a);
                             }
```

- The write by T_1 to a results in an invalidation request being sent to T_2 — via memory since T_1 is unaware of T_2.
- At the sync, T_1 must wait for an acknowledgement from T_2.
- When T_2 receives the invalidation request, it acknowledges it directly and then puts it in a queue of incoming invalidations.
- When T_1 receives the acknowledgement, the write is complete and the sync can also complete, since there are no other pending memory accesses issued before the sync.

Sync instruction example 2(3)

```
int a, f;

// called by T1            // called by T2
void v(void)               void w(void)
{                          {
        a = u();                   while (!f)
        asm("sync");                       ;
        f = 1;                     asm("sync");
}                                  printf("a = %d\n", a);
                           }
```

- The write by T_1 to `f` also results in an invalidation request being sent to T_2.
- When T_2 receives the invalidation request, it acknowledges it directly and then puts it in the queue of incoming invalidations.
- T_2 is spinning on `f` and therefore requests a copy of `f`.
- When that copy arrives, eventually with value one, it must wait at the `sync` until the invalidation to a has been applied
- Applied means invalidated a in the cache.
- Without the sync by T2 its two reads could be reordered:
  - The compiler could have put a in a register before the while-loop.
  - The CPU could speculatively have read a from memory.
  - The incoming transactions may be reordered in a node.

Sync instruction example 3(3)

- Instead of this sync instruction used in order to be concrete, we can use the Linux kernel memory barrier:

```
int a, f;

// called by T1                // called by T2
void v(void)                   void w(void)
{                              {
        a = u();                       while (!f)
        smp_mb();                              ;
        f = 1;                         smp_mb();
}                                      printf("a = %d\n", a);
                               }
```

- The memory barrier is a macro which will expand to a suitable machine instruction.
- Now, however, C/C++ and other languages have standardized support for such operations.

Weak ordering

- The memory consistency model introduced with the sync instruction is called Weak Ordering, WO, and was invented by Michel Dubois.
- The key ideas for why it makes sense are the following:
  - Shared data structures are modified in critical sections.
  - Assume N writes to shared memory are needed in the critical section.
  - In SC the processor must wait for ***each*** of the N writes to complete in sequence.
  - In WO, the processor can pipeline the writes and only wait at the end of the critical section.
  - Sync instructions are then executed as part of both the lock and unlock calls.
- Of course, some or all of the N writes may be to already owned data in which case there is no write penalty.
- Measurements on different machines and applications show different values but 10-20 % percent of the execution time can be due to writes in SC.

Release consistency

- Release Consistency, RC, is an extension to WO, and was invented for the Stanford DASH research project.
- Two different synchronization operations are identified.
- An **acquire** at a lock.
- A **release** at an unlock.
- An acquire orders all subsequent memory accesses, i.e. no read or write is allowed to execute before the acquire. Neither the compiler nor the hardware may move the access to before the acquire, and all acknowledged invalidations that have arrived before the acquire must be applied to the cache before the acquire can complete (i.e. leave the pipeline).
- A release orders all previous memory accesses. The processor must wait for all reads to have been performed (i.e. the write which produced the value read must have been acknowledged by all CPUs) and all writes made by itself must be acknowledged before the release can complete.

WO vs RC

- Recall: by $A \to B$ we mean B may execute before A
- WO relaxation: data $\to$ data
- RC relaxation:
    - data $\to$ data
    - data $\to$ acquire
    - release $\to$ data
    - release $\to$ acquire
- acquire $\to$ data: forbidden
- data $\to$ release: forbidden
- In practice not very significant difference on performance.

Atomic variables in C/C++

- With atomic variables in C/C++ we can specify which memory order we want
- There are two forms for the functions:

```
x = atomic_load_explicit(&a, memory_order_relaxed);
y = atomic_load(&b);
```

- Without `_explicit`, `memory_order_seq_cst` is used.
- More about this in the next lecture!