CSCE 4610: Computer Architecture

Final Exam: May 9 10:30-12:30
Term projects Due: May 4, 2017.

Help Session Friday May 5 1:30pm F 223

Two Surveys to Complete:
1. UNT SPOT – evaluate the class
2. CSE exit survey
   – you need to let us know how well you learned the course objectives

Review
- Vector processors
- SIMD processor
- GPUs
- Loop parallelism
  - loop carried dependency

 Dependencies from iteration to iteration
for (i=99; i>=0; i=i-1) {x[i] = x[i-k]+s;}
Such dependencies are called loop carried dependencies

Consider a loop like:
for (i=1; i<= 100; i++) {
    A[i+1] = A[i] + C[i];  /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}

S1: contains a loop-carried dependency. The value of A[i] on right hand side depends on the previous iteration.
    may prevent us from executing loop iterations in parallel (we can unroll for ILP)
S2: uses the value computed in S1 in the same iteration -- RAW data dependency.
    may prevent us from reordering instructions

Together we have a circular dependency and such loop iterations cannot be executed in parallel
If there is NO circular dependency we MAY be able to rewrite loops to eliminate loop-carried dependency.

```c
for (i=1; i<=100; i++) {
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

How can we parallelize this loop?

There is no circular dependency – we can reorder the statements inside the loop

```c
for (i=1; i<=100; i++) {
    B[i+1] = C[i] + D[i]; /* S2 */
    A[i] = A[i] + B[i]; /* S1 */
}
```

Now we still have a loop-carried dependency

Can we rewrite the loop to eliminate this?

```c
for (i=1; i<=99; i++) {
    B[i+1] = C[i] + D[i]; /* S2 */
    A[i+1] = A[i+1] + B[i+1]; /* S1 */
}
B[101] = C[101] + D[101];
```

Now we have no loop-carried dependency and the loop iterations can be executed in parallel.

How do we find if there are loop carried dependencies?

If we can express the indexes of array elements accessed in a loop as an expression, we can try to find if there is a potential loop carried dependency.

Sometimes it is more complex to see the dependency

Say `a[i] = a[i-k]...` or `a[i] = a[f(i)];`
Consider simple relationship for array elements accessed as a function of loop iteration.
Two array elements accessed in a loop may be expressed as
\[ A[a*i+b] \] and \[ A[c*i+d] \]
Or the array indexes are a linear (affine) function of loop iteration.
Given such relationships between iteration number and array elements
can we determine if there will be a “loop carried” dependency between them?

What we are trying find is:
doing the index \([a*i+b]\) in iteration \(i\) equals the index \([c*j+d]\) in iteration \(j\)

If this happens, we have a dependency between iterations \(i\) and \(j\)

There is a theoretical proof that if the
\[ GCD(c, a) \text{ does not divide } (d-b) \]

than the two array references cannot be the same for any \(i\) and \(j\) – different loop iterations.

Consider the example on page 319
\[
\text{for } (i=0; i<100; i=i+1) \{ \\
\quad \text{X}[2*i+3] = \text{X}[2*i+3]*5.0; \\
\}
\]
So we have \(a=2, b=3, c=2\) and \(d=0\)
\[ GCD(a,c) = 2 \] and \(d-b=-3\)
Since 2 does not divide -3, there is no loop carried dependency

If the relationship is not linear then it will not be possible to determine dependencies at compile time
say \(A[f(i)] = A[g(i)]\) where \(f\) and \(g\) are functions

Most compilers assume that the loop is not parallelizable
Or we may use speculation and execute loop iterations in parallel
There are many more compiler techniques to extract parallelism from loops either for Vector processors or array processors. Or even for thread parallelism on MIMD systems — particularly shared memory architectures.

Amdahl’s law for parallel programs:

\[
\text{Speedup} = \frac{\text{Execution time on one processor}}{\text{Execution time on n processors}}
\]

Let \( f \) be a fraction of the program that cannot be executed in parallel:

\[
\text{Speedup} = \frac{1}{\frac{f}{n} + \frac{1-f}{n}}
\]

In addition, serial fraction, parallel execution incurs overheads

Communication waiting for locks and barriers

\( T_{\text{overhead}} = (\text{Serial-fraction}) + (\text{other-overheads}) \)

Amdahl’s law of parallelism example on page 349

We want a speed up of 80 with 100 processors

How much serialization can we tolerate?

80 = \( 1/(\text{serial fraction}) + (1-\text{serial fraction})/100 \)

\( \text{serial fraction} = 0.0025 \) or 0.25%

Another example 5% serial fraction:

<table>
<thead>
<tr>
<th>N</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>16.8</td>
</tr>
<tr>
<td>32</td>
<td>12.55</td>
</tr>
</tbody>
</table>

Measuring performance of parallel processors

Communication overhead -- bandwidth and latency

Shared memory -- delays due to synchronization.
Implicit in Monitors and Mutual Exclusions and barriers is some mechanism for locking a variable (or lock).

Traditionally we do this using an atomic action called **Test and Set**

```
......
Test_and_set R1, 0(R2)  /* the lock is in memory at 0(R2)
......
```

May work in a single processor organization
But more difficult to implement in a multiprocessor system

Consider two instructions: Load-Linked (LL) and Store Conditional (SC) which are related

LL remembers the memory address of the load.
If some other access to the same address is made (either on the same processor or another processor), the remembered address is lost

On a SC, if the remembered address is the same as that of SC, store will be successful

```
Try:
Move R3, R4        ; Move value to be exchanged (say 1)
LL R2, 0(R1)       ; load linked to memory (lock location)
SC R3, 0(R1)       ; Store conditional to the same memory location
BEQZ R3, Try       ; if unsuccessful, try again
BNEQZ R2, Try      ; if a nonzero value was read (that means lock is
                   ; not available) on LL try again
```

The BEQZ R3, Try indicates that SC was not successful - no value was written to the lock.

BNEQZ R2, Try checks the value read by LL to see if it is zero (indicates a successful lock acquisition) or not (lock not acquired)

*This is covered in section 5.5 pages 386-391*
Other ways of implementing locks
Transactional Memories
  Acquire shared data without first getting lock
  But check for any changes to the data used before committing results

Atomicity of actions
  When two concurrent computations interact
  we need to assure atomicity of the interactions – all changes are committed or deleted
  Note that we are not talking about data directly but implicitly

Concurrency in database is based on atomic transactions

Consider booking airline tickets
  Each “agent” works on a copy of data
  and either commits or retries the results

Implementing Atomic Transactions
1. Need to be able to rollback -- changes must be buffered
   To implement Atomicity and Consistency property
2. Need to recognize failures
   Maintain read and write sets with each transaction
   Check to make sure the “read” set is not modified since last read
3. Determine successful commit
   Read current values for all “write” set and make sure these values are
   the one you wrote.

Hardware, Software and Hybrid Implementations

Project Idea: Find an implementation of Transactional Memories, write some programs and test them
Barrier Synchronization.

Fork and Join semantics

How would we implement the join (barrier) using the atomic instructions we have seen so far. We need two locks: first acquire the join variable -- and increment; and one to make sure all processes wait until the last process arrives at the barrier.

```c
Lock (counter_lock); /* to make sure updates are atomic
If (counter ==0) release = 0; /* first reset release
    /* this is to make sure that processes
    /* are forced to wait
    count = count+1;
    unlock(counter_lock);
    If (count ==total) { /* after all arrive, reset the counter
        count = 0;
        release = 1; } /* and release waiting processes
    else spin (release = 1); /* otherwise spin or wait until release
```

There are two locks -- the counter_lock and the release

Why Cache memories cause a problem in Shared Memory Systems

Consider the following memory access by different processors

```
M  m  M  m  M  m
     |
     V
Cache   Cache   Cache
P0: read 120
P1: read 120
P1: write 120 ← 80
P0: write 120 ← 60
P1: read 120
P1: read 110
P0: read 110
P0: write 110 ← 90
```

Let us think of each processor with its own local cache

We will use P#: <op> <address> [ ← <value>]

To indicate that processor # is performing a read or write (as <op>) to memory address given by <address> and if the operation is write, the value stored at the address is given by <value>. 

Diagram:

```
Shared Memory MIMD
```

We will use P#: <op> <address> [ ← <value>]

To indicate that processor # is performing a read or write (as <op>) to memory address given by <address> and if the operation is write, the value stored at the address is given by <value>.
CSCE 4610: Computer Architecture

Data from address 120 is brought to P0 cache
Data from address 120 is brought to P1 cache
(120) is modified to 80 in P1 cache
(120) is modified to 60 in P0 cache
P1 reads local copy (gets 80)
(110) is in P1 cache
(110) is in P0 cache
(110) is changed to 90 in P0 cache

If we use write-back cache, your updates will be made only to local cache--
Main memory may still have old data
Other processors may obtain old copy from memory

Why not use only Write-through cache?

Write-through caches cause congestion on memory buses
Also we still need to update copies of data in local caches
If a local copy is available, how would you know of a change?
And there is a delay in the updates visible due to write buffers

Hardware supported Cache Coherency?

Simple solutions: 1). Shared Cache
congestion in accessing shared cache
2). Do not cache shared data
need different instructions to cache and not cache

More Complex -- Caching shared data is permitted
Need to somehow update or invalidate local cache copies when some
other processor modifies the data
We can think of announcing an interest in modifying a shared variable
All other processors will “invalidate” their data

Consider simple shared memory architecture with a common bus
In principle, each cache can monitor traffic on bus, see if local copies need to be invalidated or updated local values.

Each cache also need to decide if any local changes need to be placed on the shared bus.

Implementation of a Snoopy Protocol (MESI)

Each cache cache line will be associated with one of 4 states. Invalid (I); Modified (M); Exclusive (E) or Shared (S).

Cases:
- Read Miss: That is, the data is not available in local cache or the local data is Invalid.
  - No other cache has the data. Missing data is fetched from main memory. The local copy is set to Exclusive state.
  - One or more caches have copies of the data (in Shared or read only). Main memory supplies the data, the local copy is set to Shared (or read only).
  - Only one other cache has a copy (either in Exclusive or Modified state).
    - If in M, data is written back to memory, data is supplied to requesting cache; all copies are set to Shared.
    - If in E, data is supplied to requesting cache and all copies are set to Shared.

Read hit: No special actions. Read data from local cache.
CSCE 4610: Computer Architecture

Write Hit: The address to which you are planning to write is in local cache (and not in Invalid state).

Cases:

✓ The local copy is in Shared State. An invalidation message is broadcast on the bus; all other cache copies are set to Invalid; local copy is set to Modified.

✓ Local copy is in Exclusive state. Only copy, hence write can proceed; local copy is set to Modified.

✓ Local copy is in Modified state; only copy and write can proceed; local copy remain in Modified state.

Write-Miss
Treat this as Read-Miss followed by Write-hit

The same implementation is described in more detail on pages 357-361

Note, we can combine E and M states.

On page 360, the diagrams show what happens to the state of a local cache data and remote cache data. This diagram uses only 3 states: Invalid, Exclusive and Shared.

Figure on page 361 combines the two state diagrams on page 360

Local actions in bold lines and remote actions on bus in light lines.
CSCE 4610: Computer Architecture

Cache coherency: MESI protocol example

<table>
<thead>
<tr>
<th>P0: read</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 E</td>
<td>120 I</td>
<td>120 I</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1: read</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 S</td>
<td>120 I</td>
<td>120 S</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1: write 120 → 80</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 I</td>
<td>120 M 80</td>
<td>120 I</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0: write 120 → 60</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 M 60</td>
<td>120 I</td>
<td>120 I</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1: read 120</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 S 60</td>
<td>120 I</td>
<td>120 S 60</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1: read 110</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 S 60</td>
<td>120 S 60</td>
<td>120 I</td>
</tr>
<tr>
<td>110 I</td>
<td>110 I</td>
<td>110 I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0: read 110</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 S 60</td>
<td>120 S 60</td>
<td>120 S 60</td>
</tr>
<tr>
<td>110 S</td>
<td>110 S</td>
<td>110 S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0: write 110 → 90</th>
<th>P0:</th>
<th>P1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>120 S 60</td>
<td>120 S 60</td>
<td>120 S 60</td>
</tr>
<tr>
<td>110 M 90</td>
<td>110 M 90</td>
<td>110 I</td>
</tr>
</tbody>
</table>