Topic 22: Multi-Processor Parallelism

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August
Review: Parallelism

Independent units of work can execute concurrently if sufficient resources exist.

- Dependency: Limited
- Resource: Limited

Diagram showing the execution of units of work over time with arrows indicating dependencies and resource limitations.
Review: Where to Find Parallelism?

Parallelism can be found/exists at different granularities

- **Instruction Level**
  - Ex: add instruction executes with multiply instruction
  - Compiler and hardware good at finding this

- **Thread Level**
  - Ex: screen redraw function executes with recalculate in spreadsheet
  - Programmers OK at finding this

- **Process Level**
  - Ex: Simulation job runs on same machines as spreadsheet
  - Users good at creating this
Thread Level Parallelism

Programmer generally makes TLP explicit
Compilers can extract threads in regular programs

\[
\begin{align*}
\text{for } (i = 0; i < 200; i++) & \quad \text{for } (j = 1; j < 20000; j++) \\
val[i, j] & = val[i, j-1] + 1;
\end{align*}
\]

\[
\begin{align*}
\text{forall}(i = 0; i < 200; i++) & \quad \text{for } (j = 1; j < 20000; j++) \\
val[i, j] & = val[i, j-1] + 1;
\end{align*}
\]
Thread Level Parallelism
Synchronization

• Unlike in ILP, flow of data/dependences must be explicit

```c
while (ptr = ptr->next)
    sum += ptr->val;
```

```c
while (ptr = ptr->next)
    produce(ptr);
produce(NULL);
```

```c
while (ptr = consume(ptr))
    sum += ptr->val;
```

Communication and Synchronization... (order and flow)
Multiple Processor Organization
Message Passing/Private Memory

- Threads communicate directly (send, receive)
- Scales relatively well
- No memory coherence problem (for the hardware at least)
Multiple Processor Organization
May exist on single chip
Thread Level Parallelism

Programmer generally makes TLP explicit
Compilers can extract threads in regular programs

\[
\text{for (i = 0; i < 200; i++)}
\quad \text{for (j = 1; j < 20000; j++)}
\quad \text{val}[i,j] = \text{val}[i,j-1] + 1;
\]

\[
\text{forall(i = 0; i < 200; i++)}
\quad \text{for (j = 1; j < 20000; j++)}
\quad \text{val}[i,j] = \text{val}[i,j-1] + 1;
\]
Multiple Processor Organizations

Shared Memory/Shared Bus

Processor → Cache → Single bus → Memory

Processor → Cache → Single bus

Processor → Cache → Single bus

Processor → Cache → Single bus → I/O
Multiple Processor Organizations
May be on single chip!
Multiple Processor Organizations

Shared Bus

Synchronization and Communication through memory

The cache coherency problem
Multiple Processor Organizations

Shared Bus

Processor 1

Cache

A=5

Processor 2

Cache

A=7

... 

Processor N

Cache

Single bus

P1: M[A] = 5
P2: M[A] = 7
P1: r1 = M[A] ;; value?
PN: r1 = M[A] ;; value?
P1: M[A] = 5
P2: M[A] = 7
P1: r1 = M[A] ;; value?
PN: r1 = M[A] ;; value?
Cache Coherency
Solution: Snoopy Bus
Snooping Protocols

- Variety of protocols minimize traffic for different situations
- Generally many states including: invalid, dirty read/write, clean/read-only

Reads: just work

Writes:
- Write-invalidate - other caches with address invalidate line (block) - only first write generates traffic
- Write-update - other caches with address update the values in the line (block) - like write through
Sample Protocol
Signals From Processor

- **Invalid (not valid cache block)**
  - Processor read miss
  - Processor write miss

- **Read/Write (dirty)**
  - Processor read miss
  - Write back dirty block to memory

- **Read Only (clean)**
  - Processor write (hit or miss)
Sample Protocol
Signals From Bus

Other protocols are MESIer
Spin lock

1. Load lock variable

2. Unlocked? (= 0?)
   - No
   - Yes
     - Try to lock variable using swap: read lock variable and then set variable to locked value (1)

3. Succeed? (= 0?)
   - No
   - Yes
     - Begin update of shared data

4. Finish update of shared data

5. Unlock: set lock variable to 0

Atomic

Exclusive Access
Multiple Processor Organizations
Simultaneous Multithreading (“Hyperthreading”)

- Multiple threads in single core
- Helps when single thread ILP is low

Like ILP processor, but
- Multiple PCs, one per thread
- Instructions are tagged with thread ID
- Architectural register file per thread
- Threads share execution resources

- Cross thread synchronization and communication through memory/cache
CFGs, PCs, and Cross-Iteration Deps

1. r1 = 10

1. r1 = r1 + 1
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r1 < 1000

No register live outs
Loop-Level Parallelization: DOALL

1. r1 = 10
2. r1 = r1 + 1
3. r2 = MEM[r1]
4. r2 = r2 + 1
5. MEM[r1] = r2
6. Branch r1 < 1000

1. r1 = 9
2. r1 = r1 + 2
3. r2 = MEM[r1]
4. r2 = r2 + 1
5. MEM[r1] = r2
6. Branch r1 < 999

1. r1 = 10
2. r1 = r1 + 2
3. r2 = MEM[r1]
4. r2 = r2 + 1
5. MEM[r1] = r2
6. Branch r1 < 1000

No register live outs
Another Example

1. \( r1 = 10 \)

2. \( r2 = \text{MEM}[r1] \)

3. \( r2 = r2 + 1 \)

4. \( \text{MEM}[r1] = r2 \)

5. Branch \( r2 == 10 \)

No register live outs
Another Example

<table>
<thead>
<tr>
<th></th>
<th>1. r1 = 10</th>
<th></th>
<th>1. r1 = 9</th>
<th></th>
<th>1. r1 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. r1 = r1 + 1</td>
<td></td>
<td>1. r1 = r1 + 2</td>
<td></td>
<td>1. r1 = r1 + 2</td>
</tr>
<tr>
<td></td>
<td>2. r2 = MEM[r1]</td>
<td></td>
<td>2. r2 = MEM[r1]</td>
<td></td>
<td>2. r2 = MEM[r1]</td>
</tr>
<tr>
<td></td>
<td>3. r2 = r2 + 1</td>
<td></td>
<td>3. r2 = r2 + 1</td>
<td></td>
<td>3. r2 = r2 + 1</td>
</tr>
<tr>
<td></td>
<td>4. MEM[r1] = r2</td>
<td></td>
<td>4. MEM[r1] = r2</td>
<td></td>
<td>4. MEM[r1] = r2</td>
</tr>
<tr>
<td></td>
<td>5. Branch r2 == 10</td>
<td></td>
<td>5. Branch r2 == 10</td>
<td></td>
<td>5. Branch r2 == 10</td>
</tr>
</tbody>
</table>

No register live outs
### Speculation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1.</td>
</tr>
<tr>
<td>r1 = 9</td>
<td>r1 = 10</td>
</tr>
<tr>
<td>1.</td>
<td>1.</td>
</tr>
<tr>
<td>r1 = r1 + 2</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td>2.</td>
<td>2.</td>
</tr>
<tr>
<td>r2 = MEM[r1]</td>
<td>r2 = MEM[r1]</td>
</tr>
<tr>
<td>3.</td>
<td>3.</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
<td>r2 = r2 + 1</td>
</tr>
<tr>
<td>4.</td>
<td>4.</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>5.</td>
<td>5.</td>
</tr>
<tr>
<td>Branch r2 == 10</td>
<td>Branch r2 == 10</td>
</tr>
</tbody>
</table>

No register live outs
Speculation, Commit, and Recovery

1. \( r1 = 9 \)
2. \( r1 = r1 + 2 \)
3. \( r2 = \text{MEM}[r1] \)
4. \( r2 = r2 + 1 \)
5. Send\{1\} \( r2 \)
6. Jump

No register live outs

1. \( r2 = \text{Receive}\{1\} \)
2. Branch \( r2 \neq 10 \)
3. \( \text{MEM}[r1] = r2 \)
4. \( r2 = \text{Receive}\{2\} \)
5. Branch \( r2 \neq 10 \)
6. \( \text{MEM}[r1] = r2 \)
7. Jump

1. \( r1 = 10 \)
2. \( r1 = r1 + 2 \)
3. \( r2 = \text{MEM}[r1] \)
4. \( r2 = r2 + 1 \)
5. \( \text{MEM}[r1] = r2 \)
6. Jump

1. Kill and Continue
Difficult Dependences

1. `r1 = Head`

1. \( r1 = \text{MEM}[r1] \)
2. Branch \( r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. \( \text{Print}(r3) \)
6. Jump

No register live outs
1. \( r1 = \text{Head} \)

1. \( r1 = \text{MEM}[r1] \)
2. Branch \( r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. Print \( (r3) \)
6. Jump

No register live outs
1. \( r1 = \text{Head} \)

1. \( r1 = \text{MEM}[r1] \)
2. Branch \( r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. Print ( \( r3 \) )
6. Jump

No register live outs
Golden era of computer architecture

Era of DIY:
- Multicore
- Reconfigurable
- GPUs
- Clusters

10 Cores!

10-Core Intel Xeon
"Unparalleled Performance"
P6 SUPERSCALAR ARCHITECTURE (CIRCA 1994)

Automatic Speculation

Automatic Pipelining

Commit

Parallel Resources

Automatic Allocation/Scheduling
Multicore Architecture (Circa 2010)

- Automatic Pipelining
- Parallel Resources
- Automatic Speculation
- Commit
- Automatic Allocation/Scheduling
Realizable parallelism

Credit: Jack Dongarra
"Compiler Advances Double Computing Power Every 18 Years!"
– Proebsting’s Law
Example
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Core 1 | Core 2 | Core 3
---|---|---
A1 | B1 | C1 | D1
A2 | B2 | C2 | D2
Example

A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Control Dependence

Data Dependence

Spec-DOALL
Spec-DOALL

Example

A: while (node) {
B: node = node->next;
C: res = work(node);
D: write(res);
}

Program Dependence Graph

<table>
<thead>
<tr>
<th>Control Dependence</th>
<th>Data Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

SpecDOALL

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td></td>
</tr>
</tbody>
</table>
Spec-DOALL

Example

A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Control Dependence
Data Dependence

Slowdown

Number of Threads

Control Dependence
Data Dependence

Core 1
Core 2
Core 3

SpecDOALLPerf
Spec-DOACROSS
Throughput: 1 iter/cycle

Spec-DSWP
Throughput: 1 iter/cycle
Comparison: Spec-DOACROSS and Spec-DSWP

Comm. Latency = 1: 1 iter/cycle
Comm. Latency = 2: 0.5 iter/cycle
Comm. Latency = 2: 1 iter/cycle

Pipeline Fill time
Spec-DOACROSS vs. Spec-DSWP

[MICRO 2010]
Geomean of 11 benchmarks on the same cluster

Performance Speedup (X)

(Number of Total Cores, Number of Nodes)

TLS
Spec-PS-DSWP
Performance relative to Best Sequential
128 Cores in 32 Nodes with Intel Xeon Processors [MICRO 2010]
Restoration of Trend
“Compiler Advances Double Computing Power Every 18 Years!”

– Proebsting’s Law

Era of DIY:
• Multicore
• Reconfigurable
• GPUs
• Clusters

Compiler technology inspired class of architectures?