CS3350B Computer Architecture
Multicore Thread Level Parallelism (Part 2)

Ning Xie

http://www.csd.uwo.ca/courses/CS3350b
Department of Computer Science
University of Western Ontario, Canada

Winter, 2016
Find a way to “hide” true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions

- **Hardware multithreading** - increase the utilization of resources on a chip by allowing multiple processes (threads) to share the functional units of a single processor
  - Processor must duplicate the state hardware for each thread - a separate register file, PC, instruction buffer, and store buffer for each thread
  - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
  - The memory can be shared through virtual memory mechanisms
  - Hardware must support efficient thread context switching
Types of Hardware Multithreading

- **Fine-grain** - switch threads on every instruction issue
  - Round-robin thread interleaving (skipping stalled threads)
  - Processor must be able to switch threads on every clock cycle
  - **Advantage** - can hide throughput losses that come from both short and long stalls
  - **Disadvantage** - slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads

- **Coarse-grain** - switches threads only on costly stalls (e.g., L2 cache misses)
  - **Advantage** - thread switching doesn’t have to be essentially free and much less likely to slow down the execution of an individual thread
  - **Disadvantage** - limited, due to pipeline start-up costs, in its ability to overcome throughput loss
    - Pipeline must be flushed and refilled on thread switches
Multithreaded Example: Sun’s Niagara (UltraSparc T2)

- **Eight fine grain** multithreaded single-issue, in-order cores (no speculation, no dynamic branch prediction)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Niagara 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>64-b</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1.4 Ghz</td>
</tr>
<tr>
<td>Cache (I/D/L2)</td>
<td>16K/8K/4M</td>
</tr>
<tr>
<td>Issue rate</td>
<td>1 issue</td>
</tr>
<tr>
<td>Pipe stages</td>
<td>6 stages</td>
</tr>
<tr>
<td>BHT entries</td>
<td>None</td>
</tr>
<tr>
<td>TLB entries</td>
<td>64I/64D</td>
</tr>
<tr>
<td>Memory BW</td>
<td>60+ GB/s</td>
</tr>
<tr>
<td>Transistors</td>
<td>??? million</td>
</tr>
<tr>
<td>Power (max)</td>
<td>&lt; 95 W</td>
</tr>
</tbody>
</table>
Cores are simple (single-issue, 6 stages, no branch prediction), small, and power-efficient
Simultaneous Multithreading (SMT)

- A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both ILP and TLP
  - Most SS processors have more functional unit parallelism than a single thread can effectively use
  - With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
    - Need separate rename tables (RUUs) for each thread or need to be able to indicate which thread the entry belongs to
    - Need the capability to commit from multiple threads in one cycle
- Intel’s Pentium 4 SMT is called hyperthreading
  - Supports just two threads (doubles the architecture state)
Threading on a 4-way SS Processor Example

Thread A
1 2
3
4 5 6
7 8
9 10 11 12

Thread B
1 2 3
4 5
6
7 8
9 10 11 12

Thread C
1 2 3
4 5
6
7 8
9 10 11 12

Thread D
1
2 3 4
5 6
7 8
9 10 11 12

Coarse-grain
1 2
3
4 5 6
7 8
9 10 11 12

Fine Grain
1 2
1 2 3
1 2 3
4 5
1
3
4 5
2 3 4
4 5 6

SMT
1 2 1 2
3 1 2 3
1 3 4 5
1 3 4 5
4 5 6 6
2 3 4 7
2 3 4 7
7 8 3 5
7 8 3 5
5 6 8 6
5 6 8 6
9 10 11 12
9 10 11 12
7 8 7
7 8 7
9 10 11 12
9 10 11 12
## Microprocessor Comparison

<table>
<thead>
<tr>
<th>Processor</th>
<th>SUN T1</th>
<th>Opteron</th>
<th>Pentium D</th>
<th>IBM Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Instruction issues /clock/core</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Peak instr. issues /chip</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Multithreading</td>
<td>Fine-grained</td>
<td>No</td>
<td>SMT</td>
<td>SMT</td>
</tr>
<tr>
<td>L1 I/D in KB per core</td>
<td>16/8</td>
<td>64/64</td>
<td>12K uops/16</td>
<td>64/32</td>
</tr>
<tr>
<td>L2 per core/shared</td>
<td>3 MB shared</td>
<td>1 MB / core</td>
<td>1 MB / core</td>
<td>1.9 MB shared</td>
</tr>
<tr>
<td>Clock rate (GHz)</td>
<td>1.2</td>
<td>2.4</td>
<td>3.2</td>
<td>1.9</td>
</tr>
<tr>
<td>Transistor count (M)</td>
<td>300</td>
<td>233</td>
<td>230</td>
<td>276</td>
</tr>
<tr>
<td>Die size(mm²)</td>
<td>379</td>
<td>199</td>
<td>206</td>
<td>389</td>
</tr>
<tr>
<td>Power (W)</td>
<td>79</td>
<td>110</td>
<td>130</td>
<td>125</td>
</tr>
</tbody>
</table>
Summary of Hardware Multithreading

- Benefit:
  - All multithreading techniques improve the utilisation of processor resources and, hence, the performance
  - If the different threads are accessing the same input data they may be using the same regions of memory
    - Cache efficiency improves in these cases

- Disadvantage:
  - The perceived performance may be degraded when comparing with a single-thread CPU
    - Multiple threads interfering with each other
  - The cache has to be shared among several threads so effectively they would use a smaller cache
  - Thread scheduling at hardware level adds high complexity to processor design
    - Thread state, managing priorities, OS-level information, ...
Shared Memory Model with Explicit Thread-based Parallelism

- Shared memory process consists of **multiple threads, explicit programming model** with full programmer control over parallelization

- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about **data placement**
  - Programming model is “serial-like” and thus conceptually simpler than alternatives
  - **Compiler directives** are generally simple and easy to use
  - Legacy serial code does not need to be rewritten

- **Cons:**
  - Codes can only be run in shared memory environments!
  - Compiler must support
    - (e.g., CilkPlus and OpenMP in gcc 4.xx)
Introduction to OpenMP

- API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized
- See
  http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  http://computing.llnl.gov/tutorials/openMP/
Fork - Join Model:

OpenMP programs begin as single process: master thread; Executes sequentially until the first parallel region construct is encountered

- **FORK**: the master thread then creates a team of parallel threads
- **Statements**: in program that are enclosed by the parallel region construct are executed in parallel among the various team threads
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
OpenMP Directives

- **FORK**
  - **DO/for loop**
  - **JOIN**
  - Shares iterations of a loop across the team

- **FORK**
  - **SECTIONS**
  - **JOIN**
  - Each section executed by a separate thread

- **FORK**
  - **SINGLE**
  - **JOIN**
  - Serializes the execution of a thread
OpenMP Specification

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

parallel directive
governs flow of control in the program

do/parallel do and section directives

distributes work among threads

scopes variables

shared and private clauses

coordinates thread execution

critical and atomic directives

barrier directive

runtime environment

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE
OpenMP Extends C with Pragmas

- **Pragmas** are a mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes
- Good mechanism for OpenMP because compilers that don’t recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas
Matrix Multiply in OpenMP

```c
#pragma omp parallel for private(tmp, i, j, k)
for(i = 0; i < Ndim; i++) {
    for(j = 0; j < Mdim; j++) {
        tmp = 0.0;
        for(k = 0; k < Pdim; k++) {
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
        }
        C[i*Ndim+j] = tmp;
    }
}
```

- Note: Outer loop spread across N threads; inner loops inside a thread
Amdahl’s Law

Theoretically how much speed up you can get by parallelization

\[
\text{Speed up} = \frac{S + P}{S + (P/N)}
\]

- \( S \) = Fraction of the code which is serial
- \( P \) = Fraction of the code which can be parallel
- \( S + P = 1 \)
- \( N \) = Number of processor
Amdahl's Law

The graph shows the speedup as a function of the number of cores for different P values:

- Black line: P = 0.5
- Red line: P = 0.75
- Yellow line: P = 0.95
- Green line: P = 0.99
- Purple line: P = 1

The x-axis represents the number of cores, and the y-axis represents the speedup.
Exercise: Parallelize Sum of Squares

\[ S = 0; \]
\[
\text{for } (i = 0; i < 100; ++i) \\
\quad s += X[i]*2; \quad //\text{two instructions per loop}
\]

- Each iteration depends on the result of the iteration before.
- As written, unparallelizable
  \[ P = 0 \]
- How would you create parallelism here?