Advanced Multicore Systems

- Memory consistency
- Multithreaded cores
- Data-level parallelism: vector extensions and GPUs
Coherence vs Consistency

• Cache coherence makes private caches invisible to software
  – Concerns reads/writes to a single memory location

• Memory consistency models precisely specify how memory behaves with respect to read and write operations from multiple processors
  – Concerns reads/writes to multiple memory locations
Why Consistency Matters

Initial memory contents

\begin{align*}
\text{a: } & 0 \\
\text{flag: } & 0
\end{align*}

Processor 1  
\begin{align*}
\text{Store (a), 10; } \\
\text{Store (flag), 1; }
\end{align*}

Processor 2  
\begin{align*}
\text{L: Load r1, (flag); } \\
\text{if } r_1 == 0 \text{ goto L; } \\
\text{Load r2, (a); }
\end{align*}

- What value does r2 hold after both processors finish running this code?

It depends on the order in which processor 2 observes processor 1’s stores!

10 if Store (flag) > Store (a); 0 or 10 otherwise
Sequential Consistency

A Straightforward Memory Model

“A 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 the order specified by the program”

Leslie Lamport

Sequential Consistency = arbitrary order-preserving interleaving of memory references of sequential programs
Sequential Consistency

- In-order instruction execution
- Atomic loads and stores

SC is easy to understand, but architects and compiler writers want to violate it for performance
Memory Model Issues

Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors.
Consistency Models

• Sequential Consistency
  – All reads and writes in order

• Relaxed Consistency (one or more of the following)
  – Loads may be reordered after loads
    • e.g., PA-RISC, Power, Alpha
  – Loads may be reordered after stores
    • e.g., PA-RISC, Power, Alpha
  – Stores may be reordered after stores
    • e.g., PA-RISC, Power, Alpha, PSO
  – Stores may be reordered after loads
    • e.g., PA-RISC, Power, Alpha, PSO, TSO, x86
Example: Store Buffers

- CPU can continue execution while earlier committed stores are still propagating through memory system
  - Processor can commit other instructions (including loads and stores) while first store is committing to memory

- Local loads can bypass values from buffered stores to same address
Example: Store Buffers

Initially, $\text{flag}_1 = \text{flag}_2 = 0$

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store (\text{flag}_1), 1;</td>
<td>Store (\text{flag}_2), 1;</td>
</tr>
<tr>
<td>Load $r_1$, (\text{flag}_2);</td>
<td>Load $r_2$, (\text{flag}_1);</td>
</tr>
</tbody>
</table>

**Question:** Is it possible that $r_1=0$ and $r_2=0$?

- **Sequential consistency:** No
- **Suppose Loads can go ahead of Stores waiting in the store buffer:** Yes!

Total Store Order (TSO): Sun SPARC, IBM 370, x86
Memory Fence Instructions

- Architectures with relaxed memory models provide memory fence instructions to prevent otherwise permitted reorderings of loads and stores.

\[
\text{Store } (a_1), \text{ r2}; \\
\text{Fence}_{wr} \\
\text{Load } r1, (a_2);
\]

The Load and Store can be reordered if \( a_1 \neq a_2 \). Insertion of \( \text{Fence}_{wr} \) will disallow this reordering.

- Similarly: \( \text{Fence}_{rr}; \) \( \text{Fence}_{rw}; \) \( \text{Fence}_{ww}; \)
Enforcing Ordering using Fences

\textbf{Processor 1}
\begin{itemize}
    \item Store (a), 10;
    \item Store (flag), 1;
    \item Fence_{ww};
\end{itemize}
\begin{itemize}
    \item L: Load \( r_1 \), (flag);
    \item if \( r_1 == 0 \) goto L;
    \item Load \( r_2 \), (a);
\end{itemize}
\begin{itemize}
    \item Store (flag), 1;
\end{itemize}

\textbf{Processor 2}
\begin{itemize}
    \item L: Load \( r_1 \), (flag);
    \item if \( r_1 == 0 \) goto L;
    \item Load \( r_2 \), (a);
\end{itemize}

\textit{Fences required to preserve SC behavior if memory model allows Store-Store and Load-Load reorderings:}

\textbf{Processor 1}
\begin{itemize}
    \item Store (a), 10;
    \item Fence_{ww};
    \item Store (flag), 1;
\end{itemize}

\textbf{Processor 2}
\begin{itemize}
    \item L: Load \( r_1 \), (flag);
    \item if \( r_1 == 0 \) goto L;
    \item Fence_{rr};
    \item Load \( r_2 \), (a);
\end{itemize}
Memory Consistency Takeaways

• SC is too low level a programming model. High-level programming should be based on semaphores, critical sections, atomic transactions, monitors, ...

• High-level parallel programming should be oblivious of memory model issues
  – Programmer should not be affected by changes in the memory model

• ISA definition for Load, Store, Memory Fence, synchronization instructions should
  – Be precise
  – Permit maximum flexibility in hardware implementation
  – Permit efficient implementation of high-level parallel constructs
Multithreaded Cores
Reminder: Pipeline Hazards

- Dependent instructions introduce pipeline hazards, cause stalls

LD(R1, 0, R2)
LD(R2, 4, R5)
ADDC(R5, 12, R5)
ST(R5, 8, R1)

- Even forwarding, speculation, and finding something else to do (via out-of-order execution) do not eliminate all stalls

- What else can we do to cope with this?
Resolving Hazards

• Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

• Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

• Strategy 3: Speculate
  – Guess a value and continue executing anyway
  – When actual value is available, two cases
    • Guessed correctly $\rightarrow$ do nothing
    • Guessed incorrectly $\rightarrow$ kill & restart with correct value

• Strategy 4: Find something else to do
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

Take instructions from different threads

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

T1: LD(R1, 0, R2)
T2: ADD(R7, R1, R4)
T3: XOR(R5, R1, R4)
T4: ST(R7, 0, R5)
T1: LD(R2, 4, R5)

Prior instruction in a thread always completes write-back before next instruction in same thread reads register file.
Simple Multithreaded Pipeline

Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
Multithreading Costs

• Each thread needs its own user architectural state
  – PC and general-purpose registers

• Also, needs its own system architectural state
  – Virtual memory page table base register, etc.

• Other costs?

• Appears to software (including OS) as multiple, albeit slower, CPUs
Thread Scheduling Policies

- **Fixed interleave** (*CDC 6600 PPUs, 1965*)
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble

- **Software-controlled interleave** (*TI ASC PPUs, 1971*)
  - OS allocates S pipeline slots among N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- **Hardware-controlled thread scheduling** (*HEP, 1982*)
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute from ready ones based on priority scheme
Multithreading Granularity

A) Conventional Processor

B) Coarse-grained Multithreaded (CMT)

C) Fine-grained Multithreaded (FMT)

D) Simultaneous Multithreaded (SMT)

Execution Units

Time
Multithreading Granularity

• Coarse-grained multithreading (CMT): Context switch among threads every few cycles, on long-latency events (e.g., L2 miss)
  – Threads do not share the pipeline: each switch requires flushing the pipeline (turn all the instructions into NOPs)

• Fine-grained multithreading (FMT): Context switch among threads every cycle
  – Threads coexist in the pipeline: each pipeline stage may have instructions from a different thread

• Simultaneous multithreading (SMT): Interleave multiple threads to multiple issue slots with no restrictions
  – Each pipeline stage may have instructions from different threads
Data-Level Parallelism
Data-Level Parallelism

- Same operation applied to multiple data elements
  \[
  \text{for } (\text{int } i = 0; i < 16; i++) \ x[i] = a*b[i] + c[i];
  \]
- Exploit with \textbf{vector processors} or vector ISA extensions

- Each datapath has its own local storage (register file)
- All datapaths execute the same instruction
- Memory access with vector loads and stores + wide memory port
Vector Code Example

for (i = 0; i < 16; i++) x[i] = a[i] + b[i];

**Beta assembly**

- `CMOVE(16, R0)`
- `loop:`
  - `LD(R1, 0, R4)`
  - `LD(R2, 0, R5)`
  - `ADDC(R1, 4, R1)`
  - `ADDC(R2, 4, R2)`
  - `ADD(R4, R5, R6)`
  - `ST(R6, 0, R3)`
  - `ADDC(R3, 4, R3)`
  - `SUBC(R0, 1, R1)`
  - `BNE(R0, loop)`

# of cycles = 1 + 10*15 + 9 = 160

**Equivalent vector assembly**

- `LD.V(R1, 0, V1)`
- `LD.V(R2, 0, V2)`
- `ADD.V(V1, V2, V3)`
- `ST.V(V3, 0, R3)`

# of cycles = ~5
Vector Processing Implementations

- **Advantages of vector ISAs:**
  - **Compact:** 1 instruction defines N operations
  - **Parallel:** N operations are (data) parallel and independent
  - **Expressive:** Memory operations describe regular patterns

- **Modern CPUs: Vector extensions & wider registers**
  - SSE (1999): 128-bit operands (4x32-bit or 2x64-bit)
  - AVX (2011): 256-bit operands (8x32-bit or 4x64-bit)
  - AVX-512 (2017): 512-bit operands
  - Explicit parallelism, extracted at compile time (vectorization)

- **GPUs: Designed for data parallelism from the ground up**
  - Each core processes ~32 32-bit floating-point elements per cycle
  - Each core is highly multithreaded (~64 threads/core)
    - Hide long delays effectively ➔ design for throughput, not latency
  - Implicit parallelism, scalar binary with multiple instances executed in lockstep (and regrouped dynamically)
Putting It All Together: Intel Core i7 (Nehalem)

- 4 cores/chip, 2 threads/core
- 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Huge overheads vs simple, energy-optimized cores!
Putting It All Together: Knights Landing

- 72 energy-efficient cores
  - 2-wide superscalar
  - Out-of-order execution
  - 1.5 GHz
- 4 threads per core
- Vector extensions (512-bit registers)
- 4-level memory hierarchy
- Peak performance ~6 TFLOPS (32-bit floats)
  - Performance for a sequential program?

Intel, 2016, 14 nm, 7.1B transistors, 683mm²
Putting It All Together: NVIDIA Pascal GP100

- 60 cores at 1.5 GHz:
  - 64 threads per core
  - 32x32-bit elements/thread
  - 2-wide superscalar, in-order

- Memory hierarchy:
  - 256 KB register file/core (!)
  - 64 KB L1/core
  - 4 MB shared L2
  - 8GB memory, 720GB/s

- Lots of specialized logic for graphics (texture units, raster ops, ...)

- Peak performance 
  ~12 TFLOPS (32-bit floats)