Performance And Utilization

- Performance (IPC) important
- Utilization (actual IPC / peak IPC) important too
- Even moderate superscalars (e.g., 4-way) not fully utilized
  - Average sustained IPC: 1.5–2 → <50% utilization
  - Mis-predicted branches
  - Cache misses, especially L2
  - Data dependences
- Multi-threading (MT)
  - Improve utilization by multiplexing multiple threads on single CPU
  - One thread cannot fully utilize CPU? Maybe 2, 4 (or 100) can

Latency vs Throughput

- MT trades (single-thread) latency for throughput
  - Sharing processor degrades latency of individual threads
  - But improves aggregate latency of both threads
  - Improves utilization
  - Example
    - Thread A: individual latency=10s, latency with thread B=15s
    - Thread B: individual latency=20s, latency with thread A=25s
    - Sequential latency (first A then B or vice versa): 30s
    - Parallel latency (A and B simultaneously): 25s
    - MT slows each thread by 5s
    - But improves total latency by 5s
- Different workloads have different parallelism
  - SpecFP has lots of ILP (can use an 8-wide machine)
  - Server workloads have TLP (can use multiple threads)
Thread Level Parallelism (TLP)

- Can exploit thread-level parallelism (TLP)
- Collection of asynchronous tasks: not started and stopped together
- Data shared loosely, dynamically
- Dynamically allocate tasks to processors
- Example: database server (each query is a thread)
  - `accts` is shared, can't register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`

```c
struct acct_t { int bal; }
shared struct acct_t acct[21];
int id, amt;
if (acct[id].bal >= amt)
  { dispense_cash();
  }
```

MT Implementations: Similarities

- How do multiple threads share a single processor?
  - Different sharing mechanisms for different kinds of structures
  - Depend on what kind of state structure stores
- **No state**: ALUs
  - Dynamically shared
- **Persistent hard state (aka "context"):** PC, registers
  - Replicated
- **Persistent soft state**: caches, bpred
  - Dynamically partitioned (like on a multi-programmed uni-processor)
  - TLBs need ASIDs, caches/bpred tables don't
  - Exception: ordered "soft" state (BHR, RAS) is replicated
- **Transient state**: pipeline latches, ROB, RS
  - Partitioned ... somehow

MT Implementations: Differences

- Main question: thread scheduling policy
  - When to switch from one thread to another?
- Related question: pipeline partitioning
  - How exactly do threads share the pipeline itself?
- Choice depends on
  - What kind of latencies (specifically, length) you want to tolerate
  - How much single thread performance you are willing to sacrifice
- Three designs
  - Coarse-grain multithreading (CGMT)
  - Fine-grain multithreading (FGMT)
  - Simultaneous multithreading (SMT)

The Standard Multithreading Picture

- Time evolution of issue slots
  - Color = thread (white is idle)

  ```
  CGMT | FGMT | SMT
  ```
Coarse-Grain Multithreading (CGMT)

- **Coarse-Grain Multi-Threading (CGMT)**
  - Sacrifices very little single thread performance (of one thread)
  - Tolerates only long latencies (e.g., L2 misses)
  - Thread scheduling policy
    - Designate a "preferred" thread (e.g., thread A)
    - Switch to thread B on thread A L2 miss
    - Switch back to A when A L2 miss returns
  - Pipeline partitioning
    - None, flush on switch
    - Can’t tolerate latencies shorter than twice pipeline depth
    - Need short in-order pipeline for good performance
  - Example: IBM Northstar/Pulsar
    - Switches on L1 cache miss

Fine-Grain Multithreading (FGMT)

- **Fine-Grain Multithreading (FGMT)**
  - Sacrifices significant single thread performance
  - Tolerates all latencies (e.g., L2 misses, mispredicted branches, etc.)
  - Thread scheduling policy
    - Switch threads every cycle (round-robin), L2 miss or no
  - Pipeline partitioning
    - Dynamic, no flushing
    - Length of pipeline doesn’t matter
  - Need a lot of threads
  - Extreme example: Denelkor HEP
    - So many threads (100+), it didn’t even need caches
    - Failed commercially
  - Current example: Sun Niagara (aka UltraSparc T4)
    - Eight threads x Register windows → lots of registers

Fine-Grain Multithreading
Simultaneous Multithreading (SMT)

- Can we multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of D$ (L1) miss latency

- Simultaneous multithreading (SMT)
  + Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  ± Sacrifices some single thread performance
  + Thread scheduling policy
    - Round-robin (just like FGMT)
  - Pipeline partitioning
    - Dynamic, hmmm...
  - Example: Pentium4 (hyper-threading): 5-way issue, 2 threads
  - Another example: Alpha 21464: 8-way issue, 4 threads (canceled)

Issues for SMT

- Cache interference
  - General concern for all MT variants
  - Can the working sets of multiple threads fit in the caches?
    + Shared memory SPMD threads help here
    + Same insns → share I$   
      + Shared data → less D$ contention
  - Does working set of one thread fit in the caches?
    - If not, cache interference doesn't hurt much
    - MT increases memory-level parallelism (MLP)
  - Helps most for big "server" workloads

- Large map table and physical register file
  - $\text{#mt-entries} = (#\text{threads} \times \text{#arch-regs})$
  - $\text{#phys-regs} = (#\text{threads} \times \text{#arch-regs}) + \text{#in-flight insns}$

SMT Resource Partitioning

- How are ROB/LSQ, RS partitioned in SMT?
  - Depends on what you want to achieve

  - Static partitioning
    + Divide ROB/LSQ, RS into T static equal-sized partitions
    + Ensures that low-IPC threads don't starve high-IPC ones
      - Low-IPC threads stall and occupy ROB/LSQ, RS slots
      - Low utilization

  - Dynamic partitioning
    + Divide ROB/LSQ, RS into dynamically resizing partitions
    + Let threads fight amongst themselves
      + High utilization
      - Possible starvation
    + ICOUNT: fetch policy prefers thread with fewest in-flight insns
Dynamic Partitioning Policies

- Round robin: Easiest, but can lead to IQ clog
- BRCOUNT: highest priority to threads that are least likely to be on a wrong path:
  - count the branch instrs in the decode, rename, and queue stages
  - favor threads with fewer unresolved branches
- MISSCOUNT: Attack IQ clog: give priority to threads that have the fewest outstanding D cache misses
- ICOUNT: priority to threads with fewest instrs in decode, rename, and queue:
  - prevents any one thread from filling IQ
  - gives highest priority to threads that move I efficiently – even mix of instructions from all threads
- IQPOSN: lowest priority to threads with I closest to the head of either the I or FP instruction queues
  - Schedule oldest instructions first

Power Implications of MT

- Is MT (of any kind) power efficient?
  - Static power? Yes
    - Dissipated regardless of utilization
  - Dynamic power? Less clear, but probably no
    - Highly utilization dependent
    - Major factor is additional cache activity
    - Some debate here
  - Overall? Yes
    - Static power relatively increasing

MT for Reliability

- Can multithreading help with reliability?
  - Design bugs/manufacturing defects? No
  - Gradual defects, e.g., thermal wear? No
  - Transient errors? Yes

- Background: lock-step execution
  - Two processors run same program and same time
  - Compare cycle-by-cycle; flush both and restart on mismatch

- Staggered redundant multithreading (SRT)
  - Run two copies of program at a slight stagger
  - Compare results, difference? Flush both copies and restart
    - Significant performance overhead
  - Other ways of doing this (e.g., DIVA)

SMT vs. CMP

- If you wanted to run multiple threads would you build a...
  - Chip multiprocessor (CMP): multiple separate pipelines?
  - A multithreaded processor (SMT): a single larger pipeline?

- Both will get you throughput on multiple threads
  - SMT is basically an ILP engine that converts TLP to ILP
  - CMP is mainly a TLP engine

- Again, do both
  - Sun’s Niagara (UltraSPARC T1)
    - 8 processors, each with 4-threads (fine-grained threading)
    - 1GHz clock, in-order, short pipeline (6 stages)
    - Designed for power-efficient “throughput computing”
Research: Speculative Multithreading

- **Speculative multithreading**
  - Use multiple threads/processors for ILP
  - Speculatively parallelize sequential loops
    - CMP processing elements (called PE) arranged in logical ring
    - Compiler or hardware assigns iterations to consecutive PEs
    - Hardware tracks logical order to detect mis-parallelization
  - Techniques for doing this on non-loop code too

- Effectively chains ROBs of different processors into one big ROB
  - Global commit "head" travels from one PE to the next
  - Mis-speculation flushes entire PEs
  - Also known as split-window or "Multiscalar"

Multithreading Summary

- Latency vs. throughput
- Partitioning different processor resources
- Three multithreading variants
  - Coarse-grain: no single-thread degradation, but long latencies only
  - Fine-grain: other end of the trade-off
  - Simultaneous: fine-grain with out-of-order
- Multithreading vs. chip multiprocessing

Backup slides

Course Redux

- Remember this from lecture 1?
  - Intel Pentium4
- At a high level
  - You know how this works now!
### Course Redux

- Pentium 4 specifications: what do each of these mean?
  - **Technology**
    - 55M transistors, 0.90 µm CMOS, 101 mm², 3.4 GHz, 1.2 V
  - **Performance**
    - 1705 SPECint, 2200 SPECfp
  - **ISA**
    - x86+MMX/SSE/SSE2/SSE3 (x86 translated to RISC uops inside)
  - **Memory hierarchy**
    - 64KB 2-way insn trace cache, 16KB D, 512KB–2MB L2
    - MESI-protocol coherence controller, processor consistency
  - **Pipeline**
    - 22-stages, dynamic scheduling/load speculation, MIPS renaming
    - 1K-entry BTB, 8Kb hybrid direction predictor, 16-entry RAS
    - 2-way hyper-threading