This Unit: Multithreading (MT)

- Why multithreading (MT)?
  - Utilization vs. performance
- Three implementations
  - Coarse-grained MT
  - Fine-grained MT
  - Simultaneous MT (SMT)

Performance And Utilization

- Even moderate superscalar (e.g., 4-way) not fully utilized
  - Average sustained IPC: 1.5–2 → < 50% "utilization".
    - Utilization is (actual IPC / peak IPC)
  - Why so low? Many "dead" cycles, due too:
    - Mis-predicted branches
    - Cache misses, especially misses to off-chip memory
    - Data dependences
  - Some workloads worse than others, for example, databases
    - Big data, lots of instructions, hard-to-predict branches
- Have resource idle is wasteful...
  - ... How can we better utilize the core?
- Multi-threading (MT)
  - Improve utilization by multiplexing multiple threads on single core
  - If one thread cannot fully utilize core? Maybe 2 or 4 can

Readings

- Textbook (MA:FSPTCM)
  - Section 8.1
- Paper
  - Tullsen et al., “Exploiting Choice...”
Superscalar Under-utilization

- Time evolution of issue slot
  - 4-issue processor

Superscalar

Simple Multithreading

- Time evolution of issue slot
  - 4-issue processor

Fill in with instructions from another thread

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)

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, branch predictor
  - Dynamically shared (like on a multi-programmed uni-processor)
  - TLBs need thread ids, caches/bpred tables don’t
  - Exception: ordered "soft" state (BHR, RAS) is replicated

- **Transient state**: pipeline latches, ROB, Instruction Queue
  - Partitioned ... somehow
MT Implementations: Differences

- Key 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

Coarse-Grain Multithreading (CGMT)

- **Coarse-Grain Multi-Threading (CGMT)**
  - **Pros:** Sacrifices very little single thread performance (of one thread)
    - Tolerates only long latencies (e.g., L2 misses)
  - **Cons:**
    - Priority thread scheduling policy
      - Designate a "preferred" thread (e.g., thread A)
      - Switch to thread B on thread A cache miss
      - Switch back to A when A cache miss returns
    - Can also give threads equal priority
    - Pipeline partitioning
      - None, flush the pipe on thread switch
      - Can't tolerate short latencies (<2x pipeline depth)
      - Best with short in-order pipelines
    - Example: IBM Northstar/Pulsar

CGMT

- **CGMT**
  - \[\text{Diagram for CGMT}\]

L2 miss?
Fine-Grain Multithreading (FGMT)

- **(Extremely) Fine-Grain Multithreading (FGMT)**
  - Key idea: have so many threads that never stall on cache misses
    - If miss latency is $N$ cycles, have $N$ threads!
  - Thread scheduling policy
    - Switch threads every cycle (round-robin)
  - Need a lot of threads
    - Many threads $\rightarrow$ many register files
  - Sacrifices significant single thread performance
    + Tolerates latencies (e.g., cache misses)
    + No bypassing needed!
  - Pipeline partitioning
    - Dynamic, no flushing
  - Extreme example: Denelcor HEP, Tera MTA
    - So many threads (100+), it didn’t even need caches
    - Failed commercially, but good for certain niche apps

- **Thread scheduling policy**
  - Switch threads every cycle (round-robin)
  - Need a lot of threads
    - Many threads $\rightarrow$ many register files
  - Sacrifices significant single thread performance
    + Tolerates latencies (e.g., cache misses)
    + No bypassing needed!
  - Pipeline partitioning
    - Dynamic, no flushing
  - Extreme example: Denelcor HEP, Tera MTA
    - So many threads (100+), it didn’t even need caches
    - Failed commercially, but good for certain niche apps

**Vertical and Horizontal Under-Utilization**

- FGMT and CGMT reduce **vertical under-utilization**
  - Loss of all slots in an issue cycle
- Do not help with **horizontal under-utilization**
  - Loss of some slots in an issue cycle (in a superscalar processor)
Simultaneous Multithreading (SMT)

- What can issue insns from multiple threads in one cycle?
  - Same thing that issues insns from multiple parts of same program...
  - ...out-of-order execution

**Simultaneous multithreading (SMT):** OOO + FGTM

- Aka "hyper-threading"
- Observation: once insns are renamed to physical registers...
  - Scheduler need to know which thread they came from
- Some examples
  - IBM Power5: 4-way issue, 2 threads
  - Intel Pentium4: 3-way issue, 2 threads
  - Intel Core i7: 4-way issue, 2 threads
  - Alpha 21464: 8-way issue, 4 threads (canceled)
- Notice a pattern? \#threads (T) * 2 = \#issue width (N)

SMT Resource Partitioning

- Physical regfile and insn buffer entries shared at fine-grain
  - Physically unordered and so fine-grain sharing is possible
- How are ordered structures (ROB/LSQ) shared?
  - Fine-grain sharing (below) would entangle commit (and squash)
  - Allowing threads to commit independently is important
  - Thus, typically replicated or statically partitioned
    - ROB is cheap (just a FIFO, so it can be big)
    - LSQ is more of an issue

Static & Dynamic Resource Partitioning

- **Static partitioning** (below)
  - T equal-sized contiguous partitions
  - \( \pm \) No starvation, sub-optimal utilization (fragmentation)
- **Dynamic partitioning**
  - \( P > T \) partitions, available partitions assigned on need basis
  - Better utilization, possible starvation
  - ICOUNT: fetch policy prefers thread with fewest in-flight insns
- Couple both with larger ROBs/LSQs
Multithreading Issues

- Shared soft state (caches, branch predictors, TLBs, etc.)
- Key example: cache interference
  - General concern for all multithreading variants
  - Can the working sets of multiple threads fit in the caches?
  - Shared memory threads help here (versus multiple programs)
    - Same insns → share instruction cache
    - Shared data → share data cache
  - To keep miss rates low...
    - SMT might need more associative & larger caches (which is OK)
    - Out-of-order tolerates L1 misses

- Large physical register file (and map table)
  - physical registers = (#threads * #arch-regs) + #in-flight insns
  - map table entries = (#threads * #arch-regs)

Notes About Sharing Soft State

- Caches can be shared naturally...
  - Physically-tagged: address translation distinguishes different threads
  - but TLBs need explicit thread IDs to be shared
  - Virtually-tagged: entries of different threads indistinguishable
  - Thread IDs are only a few bits: enough to identify on-chip contexts

- Thread IDs make sense on BTB (branch target buffer)
  - BTB entries are already large, a few extra bits / entry won’t matter
  - Different thread’s target prediction → automatic mis-prediction

- ...but not on a BHT (branch history table)
  - BHT entries are small, a few extra bits / entry is huge overhead
  - Different thread’s direction prediction → mis-prediction not automatic

- Ordered soft-state should be replicated
  - Examples: Branch History Register (BHR), Return Address Stack (RAS)
  - Otherwise it becomes meaningless... Fortunately, it is typically small

Multithreading vs. Multicore

- If you wanted to run multiple threads would you build a...
  - A multicore: multiple separate pipelines?
  - A multithreaded processor: a single larger pipeline?

- Both will get you throughput on multiple threads
  - Multicore core will be simpler, possibly faster clock
  - SMT will get you better performance (IPC) on a single thread
    - SMT is basically an ILP engine that converts TLP to ILP
    - Multicore is mainly a TLP (thread-level parallelism) engine

- Do both
  - Sun’s Niagara (UltraSPARC T1)
  - 8 processors, each with 4-threads (non-SMT threading)
  - 1Ghz clock, in-order, short pipeline (6 stages or so)
  - Designed for power-efficient “throughput computing”

Research: Speculative Multithreading

- Speculative multithreading
  - Use multiple threads/processors for single-thread performance
  - Speculatively parallelize sequential loops, that might not be parallel
    - 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
    - Detect reconvergence points (function calls, conditional code)
  - Effectively chains ROBs of different processors into one big ROB
    - Global commit “head” travels from one PE to the next
    - Mis-parallelization flushes one PEs, but not all PEs
  - Also known as split-window or “Multiscalar”
  - Not commercially available yet...
    - But it is the “biggest idea” from academia not yet adopted
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