Computer Architecture
Lecture 7: Limits on ILP & Multithreading (Chapter 3)

Chih-Wei Liu 劉志尉
National Chiao Tung University
cwliu@twins.ee.nctu.edu.tw

HW vs. SW to Increase ILP

- Increasing performance by using ILP has the great advantage
- Memory disambiguation: HW best
- Speculation: Both
  - HW best when dynamic branch prediction better than compile time prediction
  - Exceptions easier for HW
  - HW doesn't need bookkeeping code or compensation code
  - Very complicated to get right
- (Re-)Scheduling: SW best
  - SW is easily to look ahead to schedule better than HW does
- Compiler independence: HW only
  - HW does not require new compiler (or recompilation) to run well

Limits to ILP

- *How much ILP is available* using existing mechanisms with increasing HW budgets?
- ILP can be quite limited or difficult to exploit in some applications.
  - In particular, with reasonable instruction issue rates, cache misses that go to memory or off-chip caches are unlikely to be hidden by available ILP.

- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Advances in compiler technology + significantly new and different hardware techniques *may* be able to overcome limitations assumed in studies
  - However, unlikely such advances when coupled *with realistic hardware* will overcome these limits in near future

Ideal/Perfect Machine

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. *Register renaming* – infinite virtual registers
=> all register WAW & WAR hazards are avoided

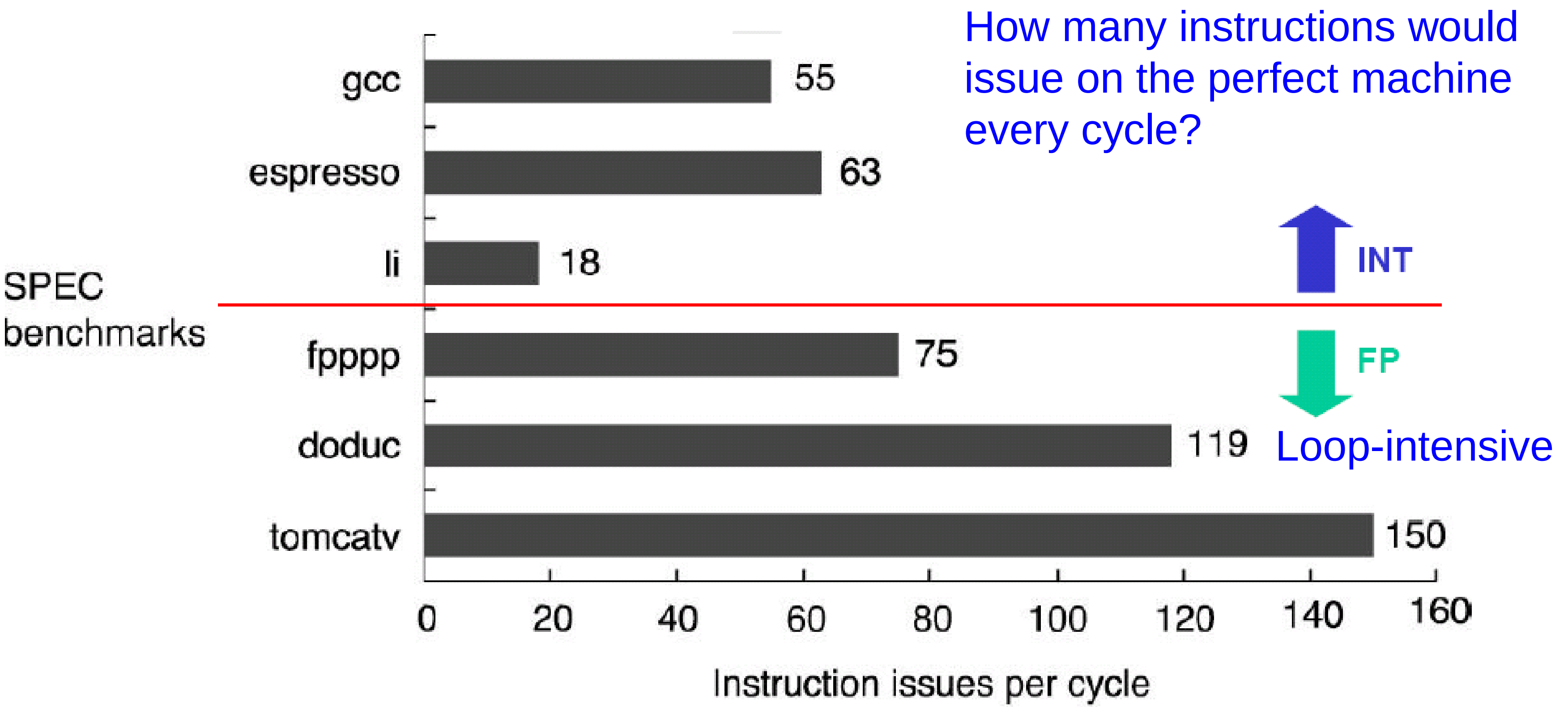
2. *Branch prediction* – perfect; no mispredictions

3. *Jump prediction* – all jumps perfectly predicted (returns, case statements)
2 & 3 $\Rightarrow$ no control dependencies; perfect speculation & an unbounded buffer of instructions available

4. *Memory-address alias analysis* – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Upper Limit to ILP: Ideal Machine



- Limited only by the ILP inherent in the benchmarks
  - Benchmarks are small codes
  - More ILP tends to surface as the codes get bigger

How to Exceed ILP Limits?

- Performance beyond single *thread* ILP
  - ILP exploits implicit parallel instructions/operations within a loop or straight-line code segment
  - There can be much higher natural parallelism (or explicit parallelism) in some applications
- Thread: instruction stream with its own PC and data
  - Thread may be a process part of a parallel program of multiple processes, or just an entire program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- *Thread Level Parallelism* (*TLP*) *and/*or *Data Level Parallelism* (*DLP*)
  - TLP: Perform executions of multiple threads that are inherently parallel
  - DLP: Perform identical operations on data, and lots of data
  - TLP/DLP could be more cost-effective to exploit than ILP

Multithreading: Exploiting TLP to Improve Uniprocessor Throughput

- Multiple threads share the functional units of in a processor via overlapping
  - Hardware duplicates only private state of each thread e.g., a separate register file, a separate PC, and a separate page table
  - HW supports the ability of (fast) thread/context switch
- When switch?
  - *Fine-grain multithreading*: Alternate instruction per thread
  - *Coarse-grain multithreading*: When costly stalls occurred, e.g. a cache miss, another thread can be executed
  - *Simultaneous multithreading* (SMT): Fine-grain multithreading + multiple-issue, dynamically scheduled processor

Fine-Grained Multithreading aka *Interleaved Multithreading* (*IMT*)

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in *a round-robin fashion*, skipping any stalled threads
- HW must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara, SPARC T1 through T5, and NVIDIA GPUs

Coarse-Grained Multithreading aka *Block Multithreading* (*BMT*)

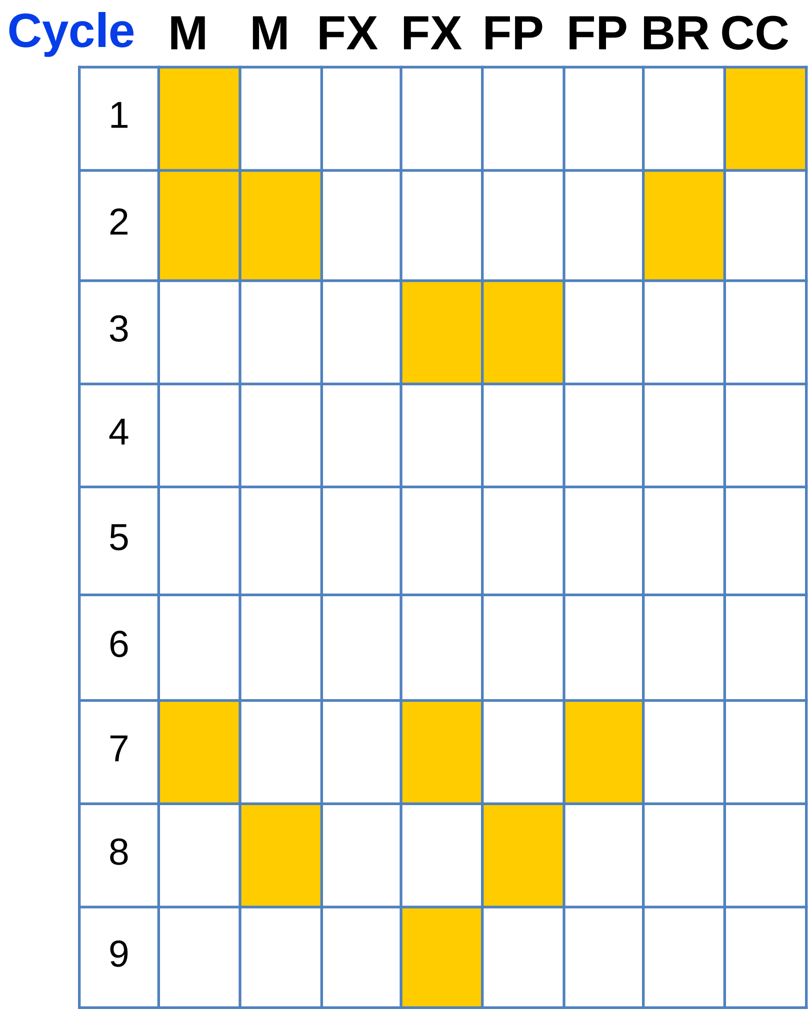
- HW switches threads only on costly stalls (e.g. L2/L3 misses), where pipeline refill << stall time
- Advantages
  - Relieve HW cost (fast thread-switching is not necessary).
  - Much less likely to slow down the execution of any one thread.
- Disadvantages
  - Hard to overcome throughput losses esepecially from shorter stalls, due to pipeline start-up costs
  - New thread must fill pipeline before instructions can complete
- No major current processors use this technique

Do both ILP and TLP?

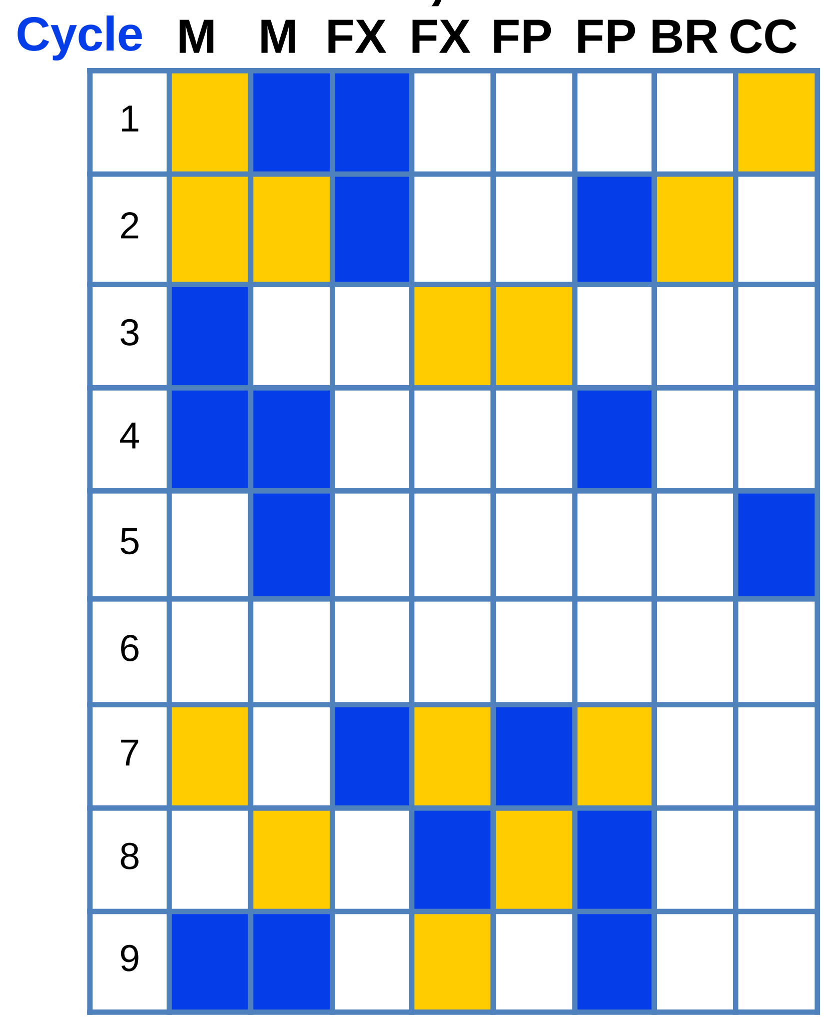
- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Simultaneous Multi-threading ...

One thread, 8 units



Two threads, 8 units



M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Source: Microprocessor Report, "Compaq Chooses SMT for Alpha" December 6, 1999

Multithreaded Categories

Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance is still there
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

And in conclusion …

- Limits to ILP (power efficiency, compilers, dependencies …) seem to limit to 3 to 6 issue for practical options
- Explicitly parallel (Data level parallelism or Thread level parallelism) is next step to performance
- Coarse grain vs. Fine grained multithreading
  - Only on big stall vs. every clock cycle
- Simultaneous Multithreading if fine grained multithreading based on OOO superscalar microarchitecture
  - Instead of replicating registers, reuse rename registers