

CS533: Simultaneous Multithreading

Josep Torrellas

University of Illinois in Urbana-Champaign

February 26, 2015

Instruction Level Parallelism (ILP)

- Between individual, independent instructions
- Hardware can only look for ILP within an instruction-window size
- Compilers can re-organize the instructions so that those that fall within the window are more independent

Types of Parallelism (II)

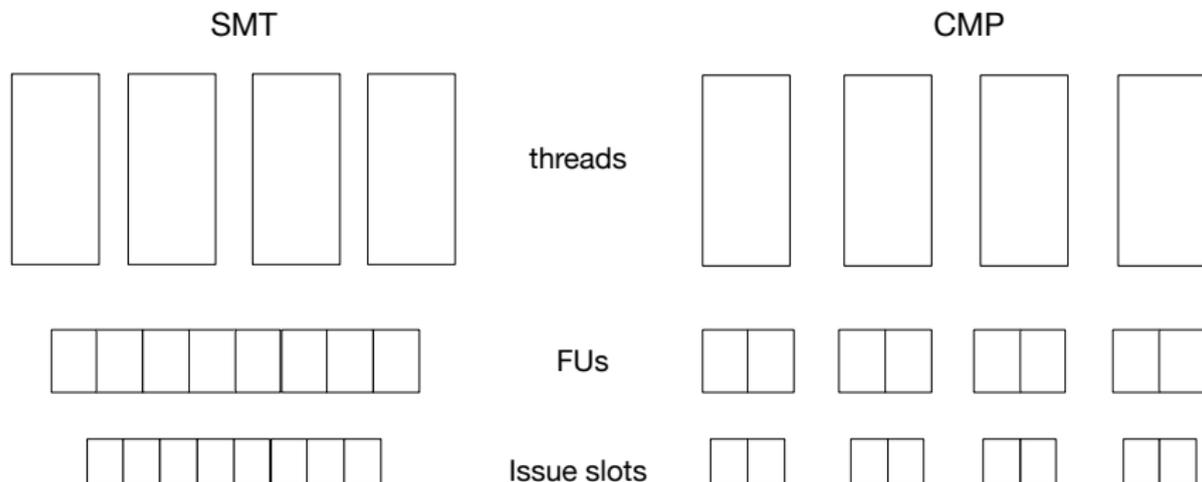
Thread Level Parallelism (TLP)

- Compiler divides the program into multiple threads of control (each executing a set of instructions)
- No need to look at a large window
- Each thread can look at a smaller window

Thread-Level Parallelism

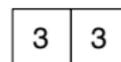
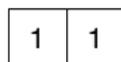
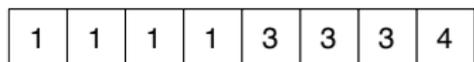
Alternate approaches

- Simultaneous multithreading (SMT)
- Chip multiprocessor (CMP)



Simultaneous Multithreading

- Multiple threads share functional units and issue slots in the same cycle
- Advantages
 - Can utilize all the resources (less waste)
 - Can run single applications faster
- Disadvantages
 - More complicated design: FU, issue slots are shared
 - Wire delays kill: slower frequency

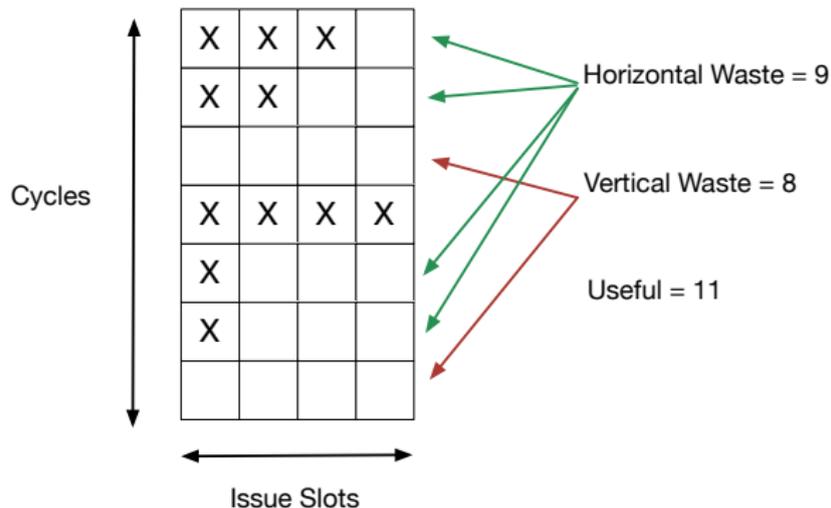


Simultaneous Multithreading

- Objectives
 - Speedup one application (parallel)
 - Speedup a mix of serial applications (throughput)
- How? Increase the use of slots by tolerating
 - memory latency (Cache, TLB miss, . . .)
 - data dependence
 - control dependence
 - structural hazards
- If a thread cannot use an issue slot, another one can

Waste

- Horizontal waste
- Vertical waste



- Superscalars: dependences cause vertical and horizontal waste
- Multithreaded (traditional): eliminates vertical waste; dependences cause horizontal waste
 - fine-grained: HEP, Tera
 - coarse-grained: Alewife

8-Issue Superscalar: Where the Cycles Go?

- Table 3 from paper by T,E,L
- Figure 2 from T,E,L

Evaluation: Different Machine Models

- In all cases, 8-issue machine
- Fine-grain: only 1 thread/cycle
- Full simultaneous issue: all 8 threads compete for each issue slot
- Single issue/dual issue/four issue: each thread is limited to N slots/cycle
- Limited connection: each thread is limited to 1 of each type of unit (still each FU is shared by at least 2 threads)

Instruction Throughput

Figure 3 of T,E,L:

- fine grain multithreading
- single issue per thread
- full simultaneous issue
- All models

- Fine-grain: with a few threads, all vertical waste gone, but quite a bit of horizontal waste
- Single-issue, Full-simultaneous issue:
 - better performance (higher throughput)
 - priority has effect
- Four issue very close to full issue
- Complexity?

- Throughput gain of SMT without extensive changes to superscalar
- Base SMT: throughput $1.8\times$ superscalar
- Tuned SMT: $2.5\times$ superscalar
- SMT need not compromise single thread performance
- Ability to choose the best instructions \implies favor threads most effectively using the processor

Changes to Support SMT (Fig 1)

- Multiple PCs and a mechanism by which the FU selects one each cycle
- Separate return address stack for each thread to predict subroutine destinations
- Per-thread I-retirement, I-queue flush, and trap
- Thread ID with each BTB to avoid predicting phantom branches
- Large register file (arch regs for all threads + additional for reg renaming). The size of the reg file affects:
 - pipeline: +2 extra states
 - scheduling of load-dependent instructions

- Conventional instruction queue that contains Instructions from all threads:
 - apparent dependences between threads removed w/ reg renaming
 - when an Inst is ready, it is issued
- Fetch from one PC round robin every cycle from those not experiencing an I-miss (refined later)
- Large register file:
 - 2 cycles to read it
 - A “register write” stage (Fig 2)

Implications of Slower Reg Access

- Increases distance between fetch and exec \implies Increase the branch misprediction +1
- Extra cycle to write back \implies extra level of bypass logic
- Increased distance between queue and exec \implies more time that wrong-path instructions remain in the pipeline after misprediction found
- No increase in inter-instruction latency between dependent instructions (except loads) \implies consecutive cycles
- 2 additional stages between rename and commit \implies increase the minimum time that a reg is held \implies increase pressure on regs

- Since Instructions are scheduled a cycle earlier (relative to exec cycle), load hit latency increases +1 (to 2 cycles)
- To handle this case:
 - schedule load-dependent instructions assuming a 1-cycle data latency but squash those instructions in the case of an L1 cache miss or bank conflict \implies Optimistic Issue
- Performance costs of Optimistic Issue:
 - Optimistically issued Inst that get squashed: waste issue cycles
 - Optimistic instructions must still be held in the IQ an extra cycle after they are issued, until it is known that they will not be squashed

Overall Claims

- I-scheduling no more complex than on a dynamically-scheduled superscalar
- Reg file data paths are no more complex than in superscalar, and the performance hit of the large reg file + extended pipe are small
- Required fetch throughput is attainable, even without any increase in fetch bandwidth
- Unmodified cache and branch prediction do not thrash
- Even aggressive superscalar technologies such as dynamic scheduling and spec execution are no match for SMT

- Fetch and decode at most 8 instructions/cycle
- Each cycle, one thread is given control of the fetch unit, chosen among those not stalled due to a l-cache miss
- Study different fetch policies:
 - partition the fetch unit among threads (fetch from multiple threads)
 - improve the quality of the instructions fetched
 - eliminate the conditions that block the fetch unit

Improve the Quality: Use Feedback

- Round robin
- BRCOUNT: highest priority to threads that are least likely to be on a wrong path:
 - count the branch instructions that are in the decode, rename, and queue stages, favoring those 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 Inst in decode, rename, and queue. Goal:
 - prevents any one thread from filling IQ
 - gives highest priority to threads that move Insts efficiently
 - even mix of instructions from all threads

Improve the Quality: Use Feedback

- IQPOSN: lowest priority to threads with Inst closest to the head of either the Int or FP instruction queues \implies oldest instructions
- Fig 5