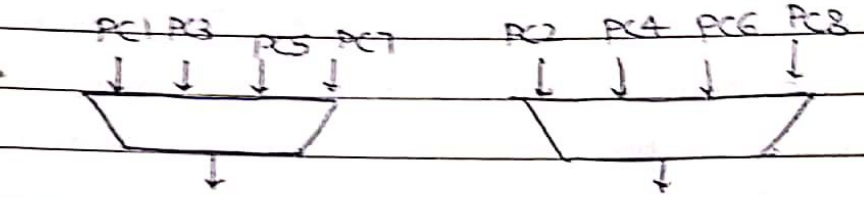
- 'Round Robin'
  Fetch 4 instructions each of T1, T2, then T3, T4, ... per cycle

- This has poorer identification of independent instructions, but is still fine.



ii - Because each thread has PC at different locations that may map to same location in L1/L2/L3 cache, we have a huge problem of cache misses
  * All threads are sharing the same cache hierarchy
  ∴ L1 hit-rate ↓
  ∴ Need to increase associativity of L1 cache
    - Because increasing associativity can increase access time of cache, we need to 'pipeline' L1 cache
      - Stage 1 :- Tag Matching
        Stage 2 :- Reading

11/3

'Out-of-order execution Engine'

- These multiple threads can share the entire back-end (decoder, scheduler, ~~out-of-order~~ reservation station, etc.)

  - If nothing else, out-of-order engine must be shared

Page No.
Date

- ~~Say fetch width~~

- Consider fetch width 4
  2 memory ports ⇒ 2 instructions per thread.

- ARF must be different for different threads
  RRF can be shared.
  - Need to fetch operand from appropriate ARF.
    ∴ Instructions need to have a tag containing the thread to which they belong.
    - This tag must also be written in RoB entry, to know which ARF to write back to.
    - Execution does not need thread ID tag.

- RoB

~~We don't retire~~	T1	I70	Not yet complete
	T2	I50	Complete ✓
Because RoB is	T1	I71	NYC
a queue, we	T3	I20	C ✓
can only write	T2	I51	NYC

back in FIFO order ☹
- If we retire I50 of T2, we will have to fill the hole by shifting all subsequent instructions one place ahead
  - Too much power dissipated ☹

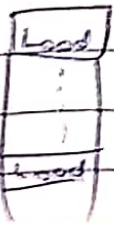
- Private RoB for each thread?
  → Solves FIFO issue ☺
  - If self-RoB is full, a thread cannot use other's RoB
    - Space wasted.

P.T.O.

- Like RoB, SB can be private or shared.
- For single-threaded program, single RoB is much better than private RoB.

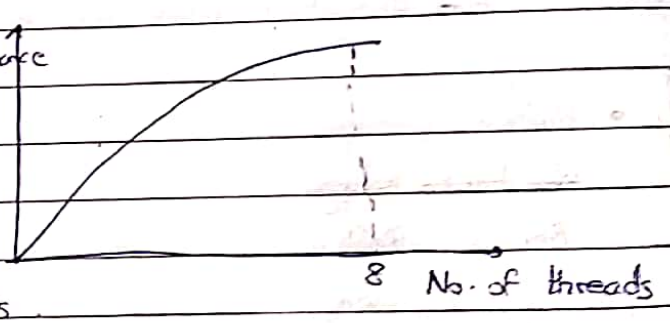
Page No.
Date

- Larger RoB also has following benefit.
  - If there are two loads in RoB, first is fetching from memory (200 cycles) second L3 cache (40 cycles)
  - Loading can be done simultaneously if caches are non-blocking (total 200 cycles).
  - Would have taken 240 cycles if both Loads were not accommodated in the small RoB.



→ Policies better than Round Robin for which thread to fetch :-

- RR Performance



Beyond 5-6 threads, very less marginal benefit to increase no. of threads.

* Fine-grain multithreading - Fetch instructions only from one thread at a time.

- Idea :- Fetch instructions from the thread that is progressing well (is not stuck at one instruction for long)

- Idea 2 :- Fetch from threads that do not have many outstanding branches.

'Branch Count'

Instructions of threads that have several outstanding branches are likely to be flushed out

Page No.
Date

- Idea 3 - Fetch from ~~inst~~ thread that has fewest data cache misses at that instant
(subsequently fetched instructions will depend on the missed data and clog our system)

'Cache Miss Count'

- Idea 4 - Fetch from threads whose previous instruction is not at top of RoB.
(that instruction ~~will~~ may make subsequent instructions wait).

Instruction Position

- Idea 5 - Fetch from the thread that has least no. of instructions currently in system.

Instruction Count

- RR is still almost as good as all the above ideas.
Intel, IBM use 2 active threads and RR fetch policy ✓

→ Branch Predictors

- 8 different BPs ? - Not optimum

- Some branches are commonly 'taken' for all threads of a program
eg - 'Function call' is always taken.

- We want the learning of each thread to influence other threads.

- But, depending on their data, different threads can behave differently at one branch.

Page No.
Date

- PHTs can be shared. BHSR cannot.
  - BHSR is private because we need to track behaviour of different threads differently.
- Return Address Stack has to be private
  - It tracks sequence of execution.

SYMMETRICAL MULTIPROCESSOR (SMP)

Page No.
Date

- Multiple identical processors - Each processor ~~shares~~ executes a thread.

- Need to communicate amongst them
  - Shared bus or ethernet or token ring, etc
  - Very slow :(

* SMP came before SMT.

- Managed by OS
  - OS considers each thread as being executed on different processors
  eg - Dual core processor - Execute 4 threads at a time
  OS considers this to be 4 processors.

* Performance of SMT also depended on programmer's ability to identify independent threads :)

* Reducing area of chip to ½ makes power density approximately ½.

$P = CV^2f$

Year-on-year (before 2004) :- $C \rightarrow C/\sqrt{2}$

$V \rightarrow V/\sqrt{2}$

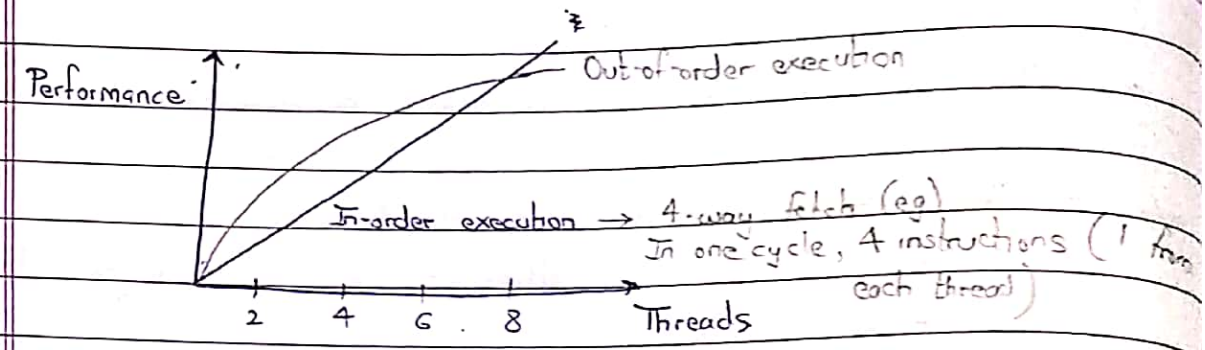
$f \rightarrow 1.4f$

$\therefore P \rightarrow P/2$

After 2004 :- Cannot ↓ V ⇒ Should not ↑ f.

∴ Increase no. of cores.

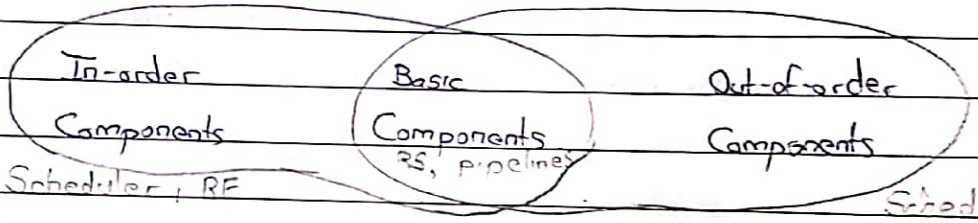
25/3



For threads > 4, we prefer in-order execution because of similar performance and much lower power dissipation

* In-order ~ No ~~RS~~ RRF and RoB required

→ Dynamic switching between in-order and out-of-order :-



Choose I-O when no. of threads is high.

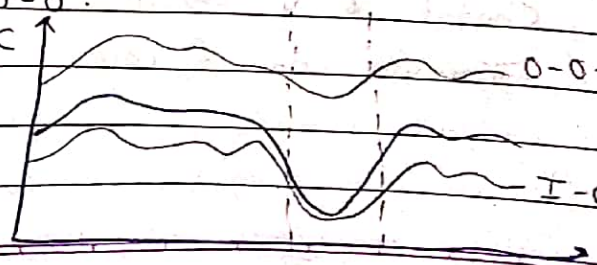
→ big.LITTLE system (ARM)

big ≡ Superscalar Out-of-order } Both single thread
LITTLE ≡ In-order

- LITTLE is used most of the times to save power

LITTLE has $IPC_{max} = 1$

- When IPC is low, this low IPC (~1) can be achieved even by I-O while consuming lesser power than O-O-O.

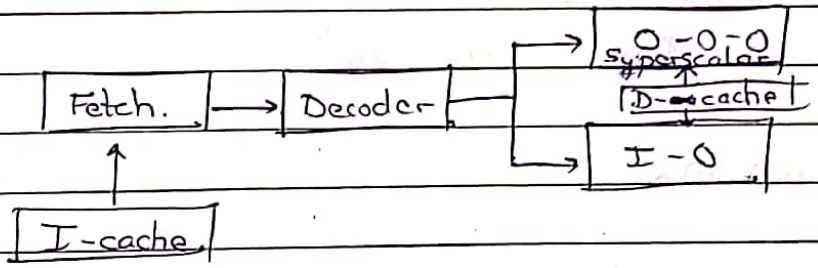


For this fetch, use I-O instead of O-O-O

- LITTLE and big are completely independent processors (not even basic components common)
  - Even RFs, L1 caches are different (I & D cache) because they are tightly coupled to processor.

- When switching between I-O and O-O-O, contents of RF and L1 caches must be copied to other RF and L1 caches
  - Takes ~ kilo cycles ☹
  - ∴ Switch only if IPC is low for several fetches.

→ Better :- Dynamic Core
- Single processor



- Can switch even for shorter 'low-IPC durations
- We have to wait for current execution engine to completely drain out before resuming execution on other engine.

* In multicore systems, it is easy to increase overall throughput but difficult to increase performance of single program.

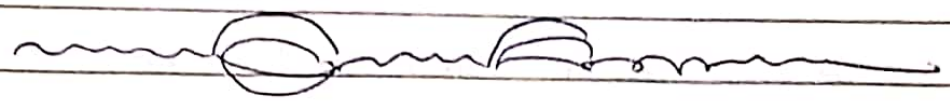
- When single program is running on multiple cores, we need to share data a memory.
  - Need to communicate between processors.
    - Bus can only let two cores to communicate at a time.

- Should all cache hierarchies be private?
  - In spite of one core having modified a value in cache (and written it back to the (common) memory), other core can use stale value from its cache :

'Data Coherency'

→ CONFIGURATION

- Possibilities:-
  - All cores identical - 'Homogeneous Multicore'
  - Few & different - 'Heterogeneous Multicore'
    - Some large cores, some small.
    - Distribution of tasks amongst cores is tricky.
  - 'Dynamic Core'
    - Acts as few big cores when Instruction-level parallelism is needed.
    - several small Thread

* Program :- 

Fraction of
~~Time taken for~~ serial part of program = s

parallelizable , if they are executed serially

= p

$s + p = 1$

- If there are N cores which run the p fraction parallelly

Time $= s + \frac{p}{N} = 1 - p + \frac{p}{N}$

- Speedup $= \frac{s+p}{s+\frac{p}{N}} = \frac{1}{1-p+\frac{p}{N}}$

For $N \to \infty$, $= \frac{1}{1-p}$

$= \frac{1}{s}$ = Maximum Speedup.

∴ Maximum Speedup is limited by fraction of serial ~~part~~ fraction in program

∴

'Amdahl's Law'

---

* Choosing between Homo / Hetero / Dynamic.

→ Performance

* Performance $\propto \sqrt{\text{Area}}$ (approx) 'Pollack's Law'.

Power $\propto$ Area.

P.T.O.

- If you merge two small cores,

$A \rightarrow 2A$

$Perf^{*} = \sqrt{2}$

Homo - Say we have N 'small' cores (constant based on power constraint)

If we merge R small cores into 1,

→ No. of cores = $\frac{N}{R}$

→ Performance of individual core $\approx \sqrt{R}$

$perf(R)$

→ Speedup = $\frac{1}{\frac{1-p}{perf(R)} + \frac{p}{perf(R) \times \frac{N}{R}}}$ ← single small core; N remains constant

* R and p are variables.

Find range of p based on statistics.

Over range of p, plot speedup as function of R.

Find maximum

Hetero 1 big core made up of R resources

R' cores made up of 1 resources each

→ No. of cores = $N - R + 1$

→ Serial part is executed on big core

→ Speedup = $\frac{1}{\frac{1-p}{perf(R)} + \frac{p}{(N-R)(1) + 1 \cdot perf(R)}}$

$perf(x) \approx \sqrt{x}$

- N small cores, each of IPC1 :- $IPC_{max} = N$
  1 big core made from those N cores :- $IPC_{max} = \sqrt{N}$

Dynamic → Serial part is executed by combining all small cores

→ Speedup = $\dfrac{1}{\dfrac{1-p}{perf(N)} + \dfrac{p}{N}}$

Best speedup of all three possibilities.

28/3

- Homogeneous :-
  - Easier to allocate threads, manage by OS, etc :)
  - No performance benefit for serial code.

→ Power → Static (Leakage in idle cores)
→ Dynamic (& Actual work, logic consumption)

- W = fraction of power core consumes when idle (static)
  ~~Dynamic power~~ $P_{dynamic} + P_{static} = 1$
  1 bce → 1 unit power
  R bces → R unit power

Homo: Energy (serial) = Time × Power = $\dfrac{1-p}{perf(R)} \left(R(1) + \left(\dfrac{N}{R} - 1\right)(WR) \right)$ (active core, idle cores)

Energy (parallel) = $\dfrac{p}{perf(R) \times \frac{N}{R}} \times N$

$$E = E_s + E_p = \frac{(1-p)}{perf(R)} R\left(1 + \left(\frac{N}{R} - 1\right)W\right) + \frac{pR}{Perf(R)}$$

- Power = $\frac{\text{Energy}}{\text{Time}}$

$$= \frac{E_s + E_p}{\frac{1-p}{perf(R)} + \frac{p}{perf(R)\frac{N}{R}}}$$

- Performance per ~~Joule~~ Watt = $\frac{\text{Speedup}}{\text{Power}}$

'Power Efficiency'

$$= \frac{1}{E}$$

- Performance per Joule = $\frac{1}{E} \times \frac{1}{\text{Time}}$

'Energy Efficiency'

$$= \frac{1}{E} \times \left(\frac{1}{\frac{1-p}{perf(R)} + \frac{p}{perf(R)\frac{N}{R}}} \right)$$

Hetero

$$E_s = \frac{1-p}{perf(R)} \left(\underbrace{R(1)}_{\text{active}} + \underbrace{(N-R)W}_{\text{idle}} \right)$$

$$E_p = \frac{p}{N-R+perf(R)} \times N$$

↳ all active

$$E = E_s + E_p$$

- Power = $\frac{\text{Energy}}{\text{Time}} = \frac{E}{\frac{1-p}{perf(R)} + \frac{p}{N-R+perf(R)}}$

- Performance per watt = $\frac{1}{E}$

- Performance per Joule = $\frac{1}{E} \times \frac{1}{Time}$

Dynamic $E_s = \frac{1-p}{perf(N)}$ .... no idle cores

- $E_p = \frac{p}{N} \times N$

- ~~Abac :~~