Multithreading

- Introduction
- Fine-grained Multithreading
- Coarse-grained Multithreading
- Simultaneous Multithreading
- Multithreading in Intel processors

Multi-Threading

- A multithreaded CPU is not a parallel architecture, strictly speaking; multithreading is obtained through a single CPU, but it allows a programmer to design and develop applications as a set of programs that can virtually execute in parallel: namely, threads.
- If these programs run on a “multithreaded” CPU, they will best exploit its architectural features.
- What about their execution on a CPU that does not support multithreading?

Multi-Threading

- Multithreading addresses a basic problem of any pipelined CPU: a cache miss causes a “long” wait, necessary to fetch from RAM the missing information. If no other independent instruction is available to be executed, the pipeline stalls.
- Multithreading is solution to avoid waisting clock cycles as the missing data is fetched: making the CPU manage more peer-threads concurrently; if a thread gets blocked, the CPU can execute instructions of another thread, thus keeping functional units busy.
- So, why cannot be threads form different tasks be issued as well?

Multi-Threading

- To realize multithreading, the CPU must manage the computation state of each single thread.
- Each thread must have a private Program Counter and a set of private registers, separate from other threads.
- Furthermore, thread switch must be much more efficient than process switch, that requires usually hundreds or thousands of clock cycles (process switch is a software procedure, mostly)
- There are two basic techniques for multithreading:

1. fine-grained multithreading
2. coarse-grained multithreading

- NB: in the following, we cover initially "single-issue" processors

Fine-grained Multi-Threading

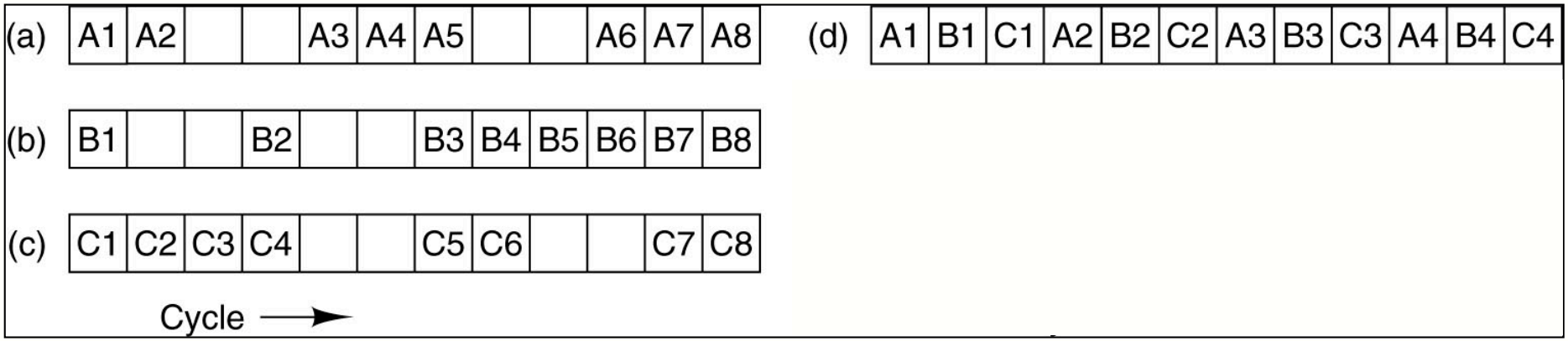
1. **Fine-grained Multithreading**: switching among threads happens at each instruction, independently from the the fact that the thread instruction has caused a cache miss.

- Instructions "scheduling" among threads obeys a round robin policy, and the CPU must carry out the switch with *no* overhead, since overhead cannot be tolerated

- If there is a sufficient number of threads, it is likely that at least one is active (not stalled), and the CPU can be kept running.

Fine-grained Multi-Threading

- (a)-(c) three threads and associated stalls (empty slots).
  (d) Fine-grained multithreading. Each slot is a clock cycle, and we assume for simplicity that each instruction can be completed in a clock cycle, unless a stall happens.
  (Tanenbaum, Fig. 8.7)



- In this example, 3 threads keep the CPU running, but what if A2 stall lasts 3 or more clock cycles?

Fine-grained Multi-Threading

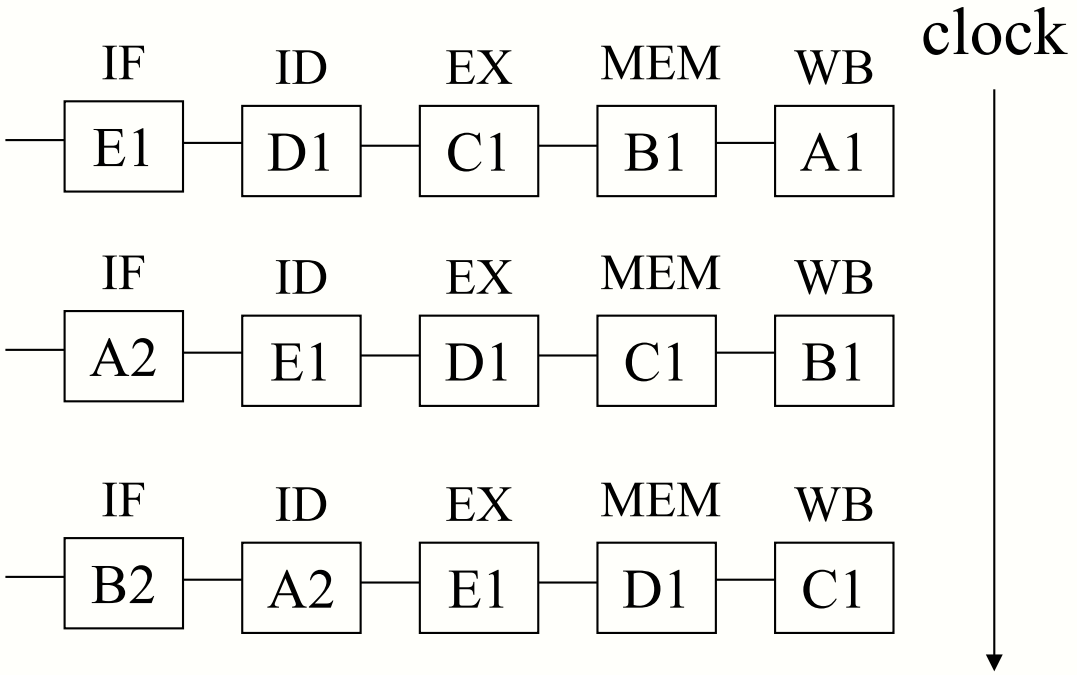
- CPU stalls can be due to a cache miss, but also to a true data dependence, or to a branch: dynamic ILP techniques do not always guarantee that a pipeline stall is avoided.
- With fine-grained multithreading in a pipelined Architecture, if:
  - the pipeline has *k* stages,
  - there are at least *k* threads to be executed,
  - and the CPU can execute a thread switch at each clock cycle
- then there can never be more than a single instruction per thread in the pipeline at any instant, so there cannot be hazards due to dependencies, and the pipeline never stalls (… another assumption is required …).

Fine-grained Multi-Threading

- Fine-grained multithreading in a CPU with a 5-stage pipeline: there are never two instructions of the same thread concurrently active in the pipeline. If instructions can be executed out of order, then it is possible to keep the CPU fully busy even in case of a cache miss.

5 threads in execution:

A1	A2	A3	A4	A5	A6	...
B1	B2	B3	B4	B5	B6	...
C1	C2	C3	C4	C5	C6	...
D1	D2	D3	D4	D5	D6	...
E1	E2	E3	E4	E5	E6	...



Fine-grained Multi-Threading

- Besides requiring an efficient context switch among threads, threads fine-grained scheduling at each instruction slows down a thread even when the thread could go on since it is not causing a stall.
- Furthermore, there might be fewer threads than stages in the pipeline (actually, this is the usual case), so keeping the CPU busy is no easy matter.
- Keeping into account these problems, a different approach is followed in *coarse-grained* multithreading.

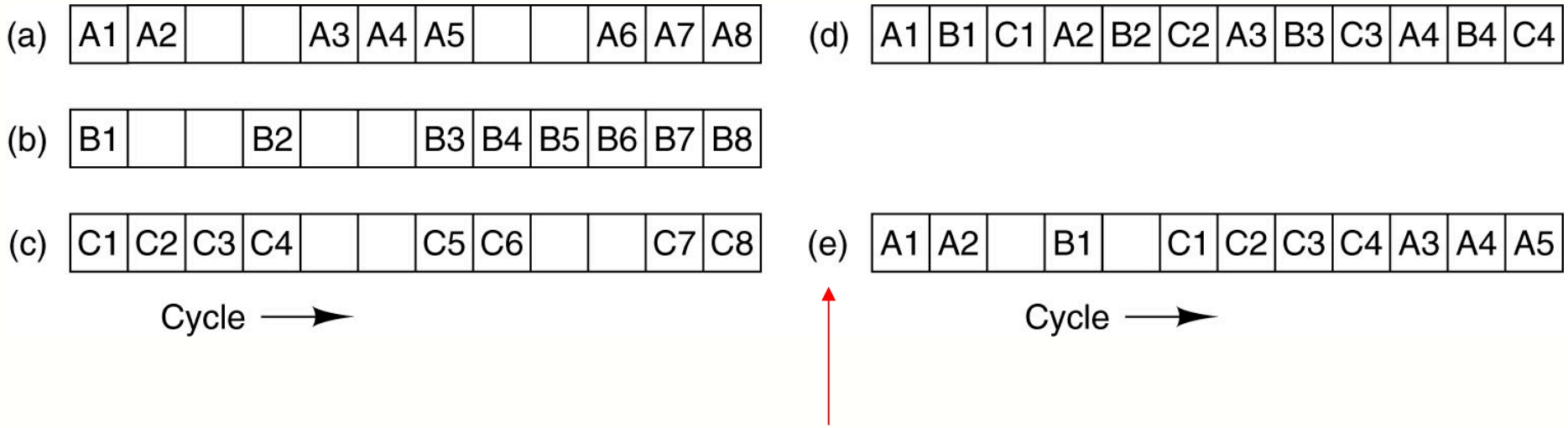
Coarse-grained Multi-Threading

2. **Coarse-grained Multithreading**: a switch only happens when the thread in execution causes a stall, thus wasting a clock cycle.

- At this point, a switch is made to another thread. When this thread in turn causes a stall, a third thread is scheduled (or possibly the first one is re-scheduled) and so on.
- This approach potentially wastes more clock cycles than the fine-grained one, because the switch happens only when a stall happens.
- but if there are few active threads (even just two), they can be enough to keep the CPU busy.

Coarse vs Fine-grained Multi-Threading

- (a)-(c) three threads with associated stalls (empty slots).
  (d) Fine-grained multithreading.
  (e) Coarse-grained multi-threading (Tanenbaum, Fig. 8.7)



Coarse vs Fine-grained Multi-Threading

- In the preceeding drawing, fine-grained multithreading seems to work better, but this is not always the case.
- Specifically, a switch among threads cannot be carried out without any waste in clock cycles.
- So, if the instructions of the threads do not cause stalls frequently, a coarse-grained scheduling can be more convenient than a fine-grained one, where the context switch overhead is paid at each clock cycle (this overhead is very small, but never null).

Coarse e Fine-grained Multi-Threading

- Are Coarse and Fine grained multi-threading similar to concepts discussed in a standard Operating Systems course ?

Medium-grained Multi-Threading

3. **Medium-grained multithreading**: an intermediate approach between fine and coarse - grained multithreading consists of switching among threads only when the running one is about to issue an instruction ***that might cause a long-lasting stall***, such as a load (requesting non-cached data), or a branch.

- The instruction is issued, but the processor carries out the switch to another thread. With this approach, one spares even the small, one-cycle waste due to the stall by the executing load (unavoidable in multithreading coarse-grained).

Multi-Threading

- How can the pipeline know which thread an instruction belongs to?
- In fine-grained MT, the only way is to associate each instruction with a *thread identifier*, e.g. the unique ID attached to the thread within the thread set it belongs to.
- In coarse-grained MT, besides the solution above, the pipeline can be emptied at each thread switch: in this way, the pipeline only contains instructions from a single thread.
- The last option is affordable only if the switch happens at intervals that are much longer than the time required to empty the pipeline.

Multi-Threading

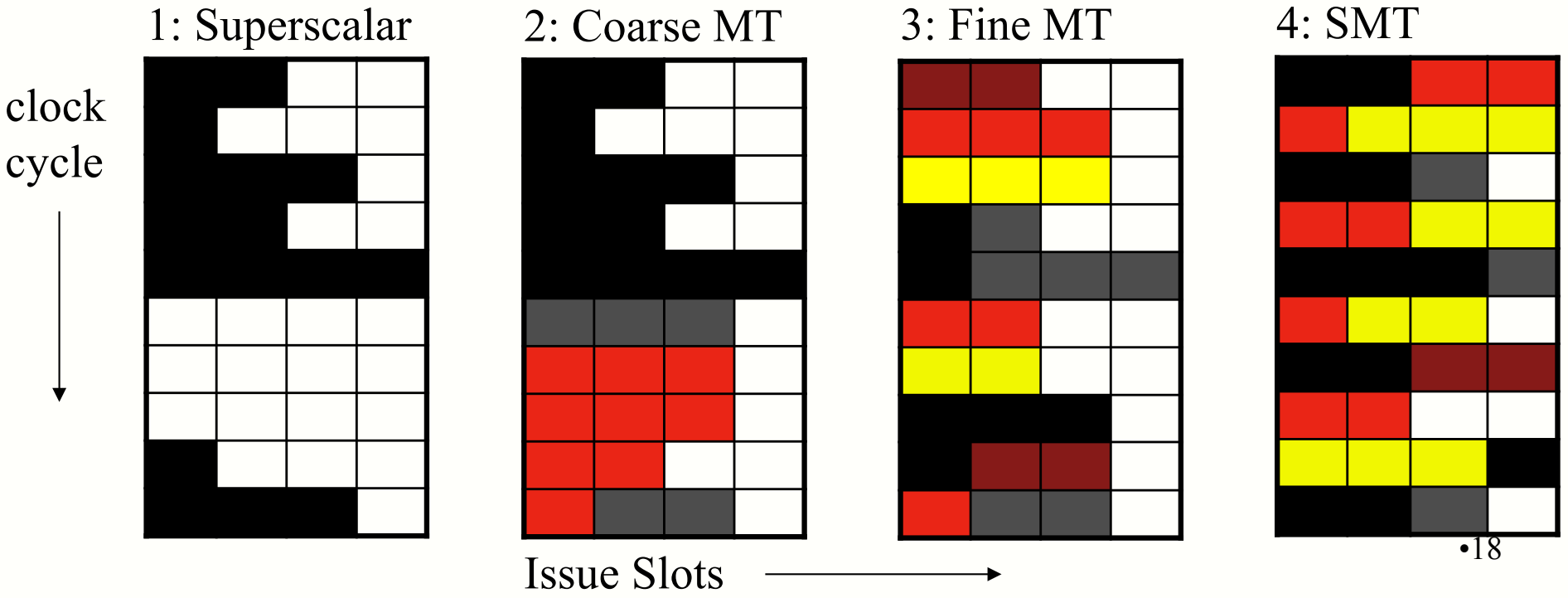
- Finally, all instructions from the executing threads are (as much as possible) in the instruction cache; otherwise, each context switch causes a cache miss, and all advantages from threading are lost.

Simultaneous Multi-Threading and Multiple Issue

- Modern superscalar, multiple issue and dynamic scheduling pipeline architectures allow to exploit both ILP (instruction level) and TLP (thread level) parallelism.
- **ILP + TLP = Simultaneous Multi-Threading (SMT)**
- SMT is convenient since modern multiple-issue CPUs have a number of functional units that cannot be kept busy with instructions from a single thread.
- By applying register renaming and dynamic scheduling, instructions belonging to different threads can be executed concurrently.

Simultaneous Multi-Threading

- In SMT, multiple instructions are issued at each clock cycle, possibly belonging to different threads; this increases the utilization of the various CPU resources (Hennessy-Patterson, Fig. 6.44: each slot/colour couple represents a single instruction in a thread).



Simultaneous Multi-Threading

- In superscalar CPUs with no multithreading, multiple issue can be useless if there is not enough ILP in each thread, and if a long lasting stall (a L3 cache miss) freezes the whole processor.
- In coarse-grained MT, long-lasting stalls are hidden by thread switching, but a poor ILP level in each thread limits CPU resource exploitation (e.g., not all issue slots available can effectively be used)
- Even in fine-grained MT, a poor ILP level in each thread limits CPU resource exploitation.
- SMT: instructions belonging to different threads are (almost certainly) independent, and by issuing them concurrently, CPU resources utilization raises.

Simultaneous Multi-Threading

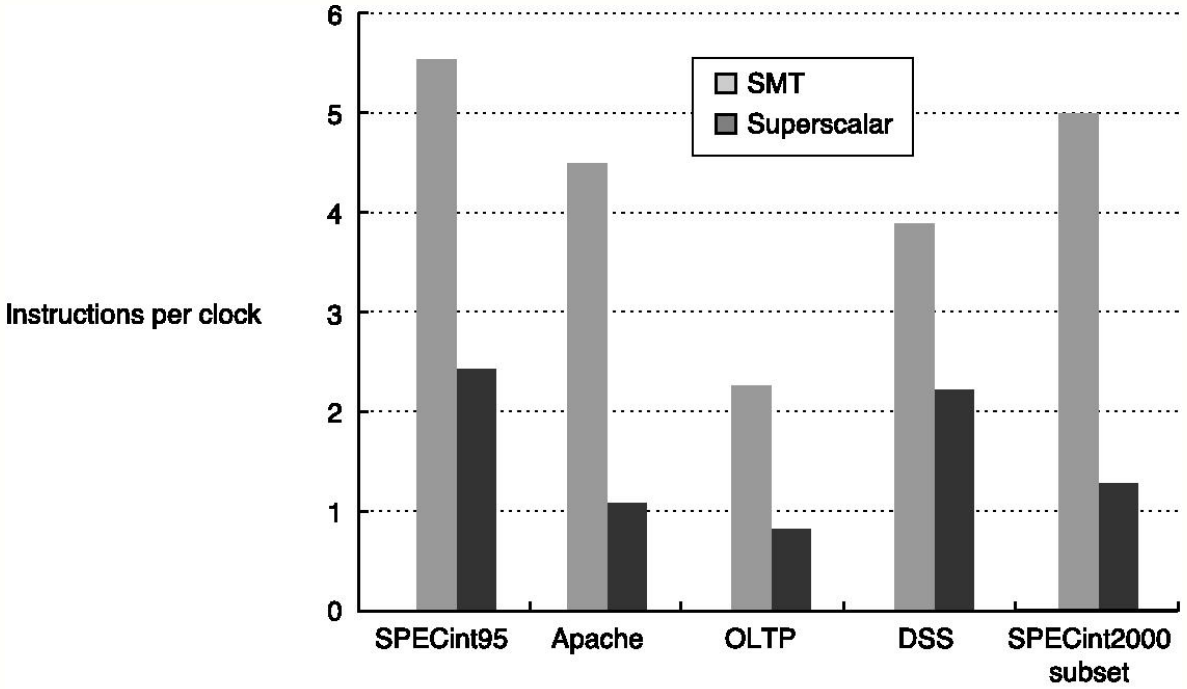
- Even with SMT, it is not always guaranteed to issue the maximum number of instructions per clock cycle, because of limited number of available functional units, reservation stations, I-Cache capability to feed threads with instructions, and shear number of threads.
- Clearly, SMT is viable only if there is a wealth of registers available for renaming.
- Furthermore, in a CPU supportig speculation, there must be a ROB (at least logically) distinct for each thread, so that retirement (instructions commit) is carried out independently by each thread.

Simultaneous Multi-Threading

- Realizing a processor fully exploiting SMT is definitely a complex task; is it worth doing?
- A simple simulation: a superscalar multithreaded CPU with the following features:
  - a 9-stage pipeline
  - 4 general purpose floating point units
  - 2 ALUs
  - up to 4 load or store per cycle
  - 100 integer + 100 FP rename registers
  - up to 12 commit per cycle
  - 128k + 128k L1 caches
  - 16MB L2 cache
  - 1k BTB
  - 8 contexts for SMT (8 PC, 8 architectural registers)
  - up to 8 instruction issued per clock cycle, from 2 different contexts

Simultaneous Multi-Threading

- This hypothetical processor has slightly more resources than a modern real processor; notably, it handles up to 8 concurrent threads. Would SMT really be beneficial? Look at figures from benchmarks of concurrent applications: number of retired instructions per clock cycle (Hennessy-Patterson, Fig. 6.46):



Simultaneous Multi-Threading

- An intriguing question: with multithreading, one usually refers to a set of "*peer threads*" whose instructions are concurrently executed in a multithreaded CPU.
- What about the concurrent execution of instructions from different *processes* ?
- Would some specific additional resource be necessary?

Intel Multi-Threading

- Multithreading was first introduced by Intel in Xeon processor in 2002, later in the 3,06 GHz Pentium 4, with code name hyperthreading. The name is attractive, actually hyperthreading supports only two threads in SMT mode.
- According to Intel, designers had speculated that multithreading was the simplest way to increase performance: an increase by 5% of CPU area would allow to run a second thread, thus effectively using CPU resources otherwise wasted.
- Intel benchmark suggested an increase of CPU performance by 25% -- 30%.

Intel Multi-Threading

- To the Operating system, a multithreaded processor is indeed a double processor, with two CPUs sharing caches and RAM: if two applications can run independently and share the same address space, they can be executed in parallel in two threads.
- A movie editing code can use different filters to be applied in each frame. The code can be structured as two threads, that process odd/even frames, and that execute in parallel.

Intel Multi-Threading

SMT Implementation Details

Replicated – Duplicate state for SMT

- Register state
- Renamed RSB
- Large page ITLB

Partitioned – Statically allocated between threads

- Key buffers: Load, Store, Reorder
- Small page ITLB

Competitively shared – Depends on thread's dynamic behavior

- Reservation station
- Caches
- Data TLBs, 2nd level TLB

Unaware

- Execution units

Applications that will benefit

Complex memory access (memory access stalls)
Mix of instruction types (e.g integer and FP computation)

Intel Multi-Threading

- Since two threads can use the CPU concurrently, it is necessary to design a strategy that allows both threads to effectively use CPU resources.
- Intel uses 4 different strategies to share resources between the two threads.
- **Replication.** Obviously, some resources *have* to be replicated, in order to manage the two threads: two *program counters* and *registers mapping tables* (ISA registers vs rename registers) so that each thread has an independent set of registers. This replications accounts for the 5% increase in processor area.

Intel Multi-Threading

- **Partitioning.** Some hardware resources are rigidly partitioned between the two threads. Each thread can use exactly half of each resource. This applies to all buffers (for LOAD, STORE instructions) and to the ROB ("retirement queue" in Intel terminology).
- Partitioning can of course reduce the utilization of the partitioned resources, when a thread does not use its part of the resource, which could be used by another thread.

Intel Multi-Threading

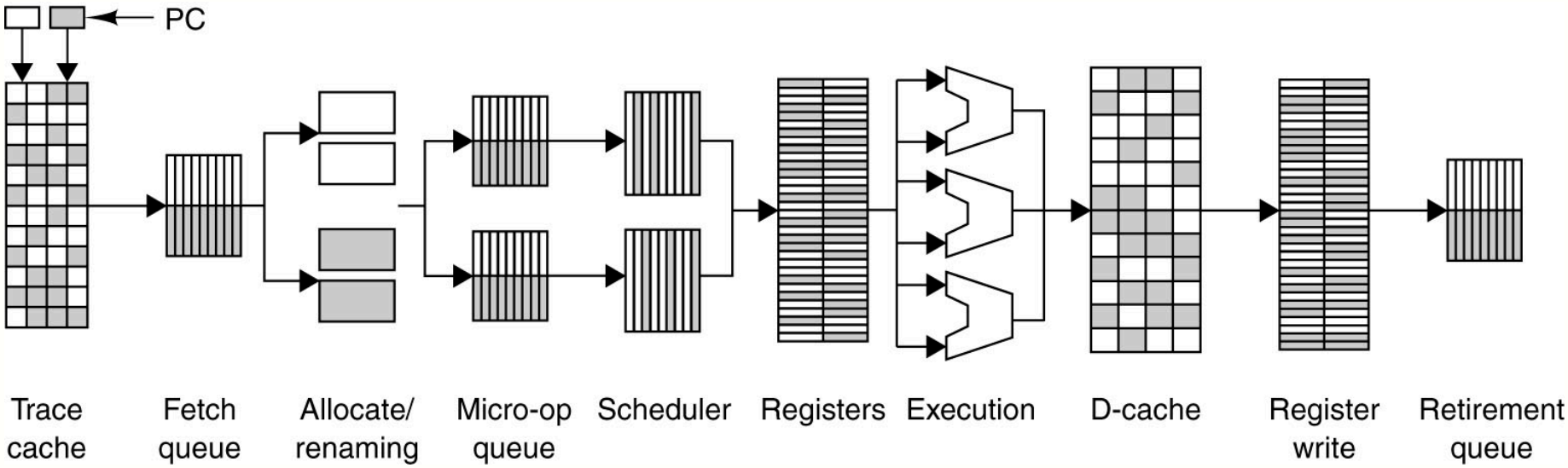
- **Sharing.** The hardware resource is completely shared. The first thread that gets hold of the resources uses it, and the other thread waits.
- This type of resource management solves the problem due to an unused resource (if the thread does not need it), since it can be allocated to the second one. Obviously, the reverse problem arises: a thread can be slowed down if the required resource is completely allocated to the other one.
- For this reason, in Intel processor the only resources completely shared are those available in a great quantity: for them, it is unlikely that a "starvation" problems arises, e.g. cache lines.

Intel Multi-Threading

- **Threshold sharing.** A thread can use dynamically the resource, up to a given percentage; so, a part remains available for the other task (possibly less than half).
- The scheduler that dispatches uops to the reservation stations uses this policy.

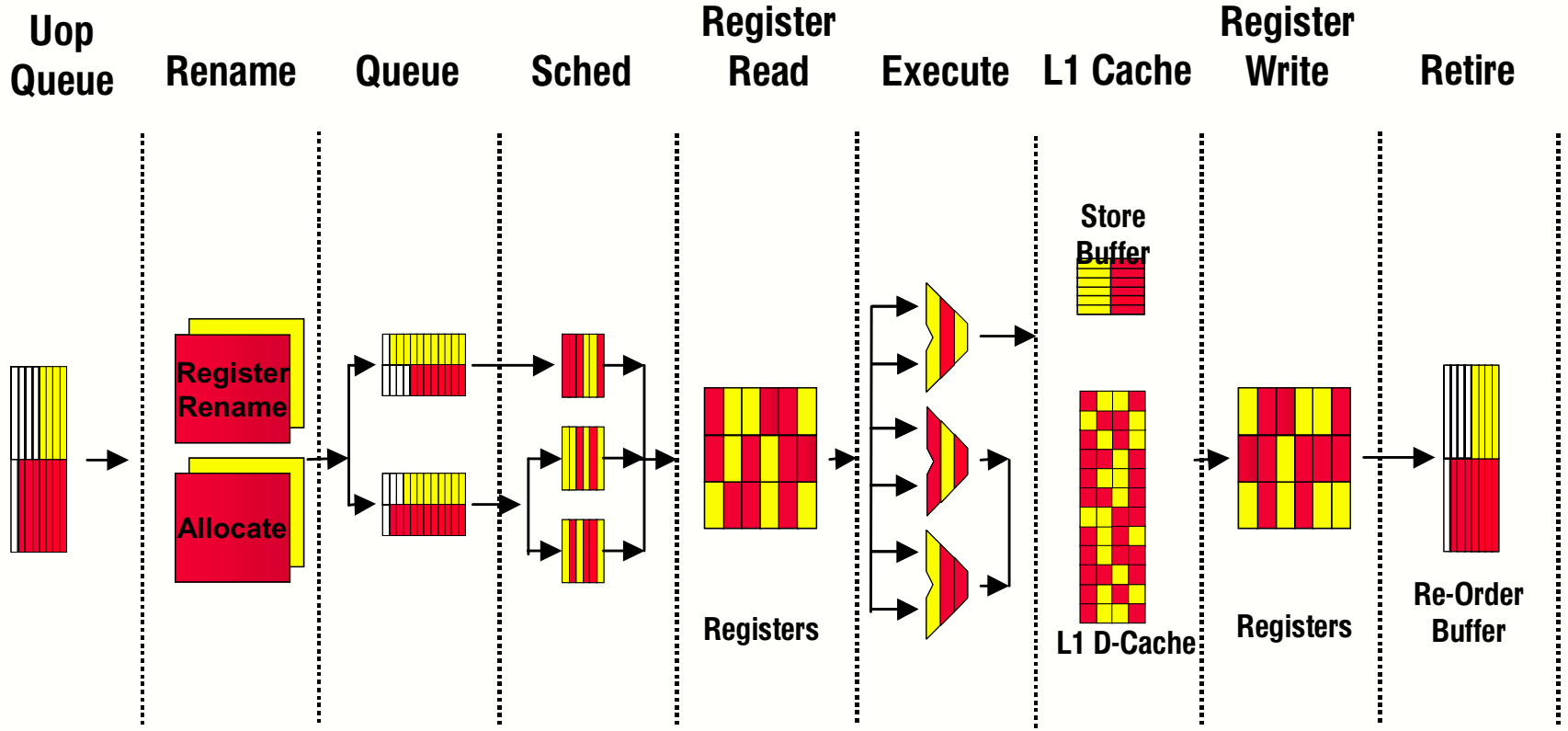
Intel Multi-Threading

- Resourse sharing in Pentium 4 pipeline (Tanenabum, Fig. 8.9).



Intel Multi-Threading

- Resourse sharing in Pentium 4 pipeline (Intel source).



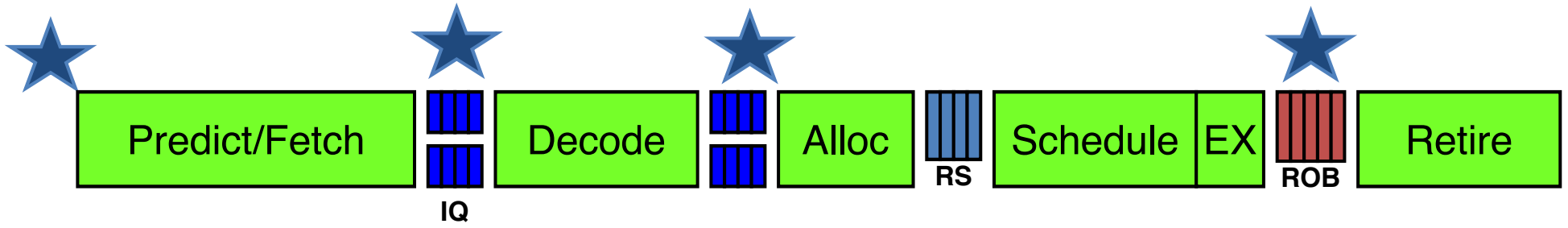
Intel Multi-Threading

- "IP"
-
-
- "Trace Cache"

Intel Multi-Threading

SMT Thread Selection Points

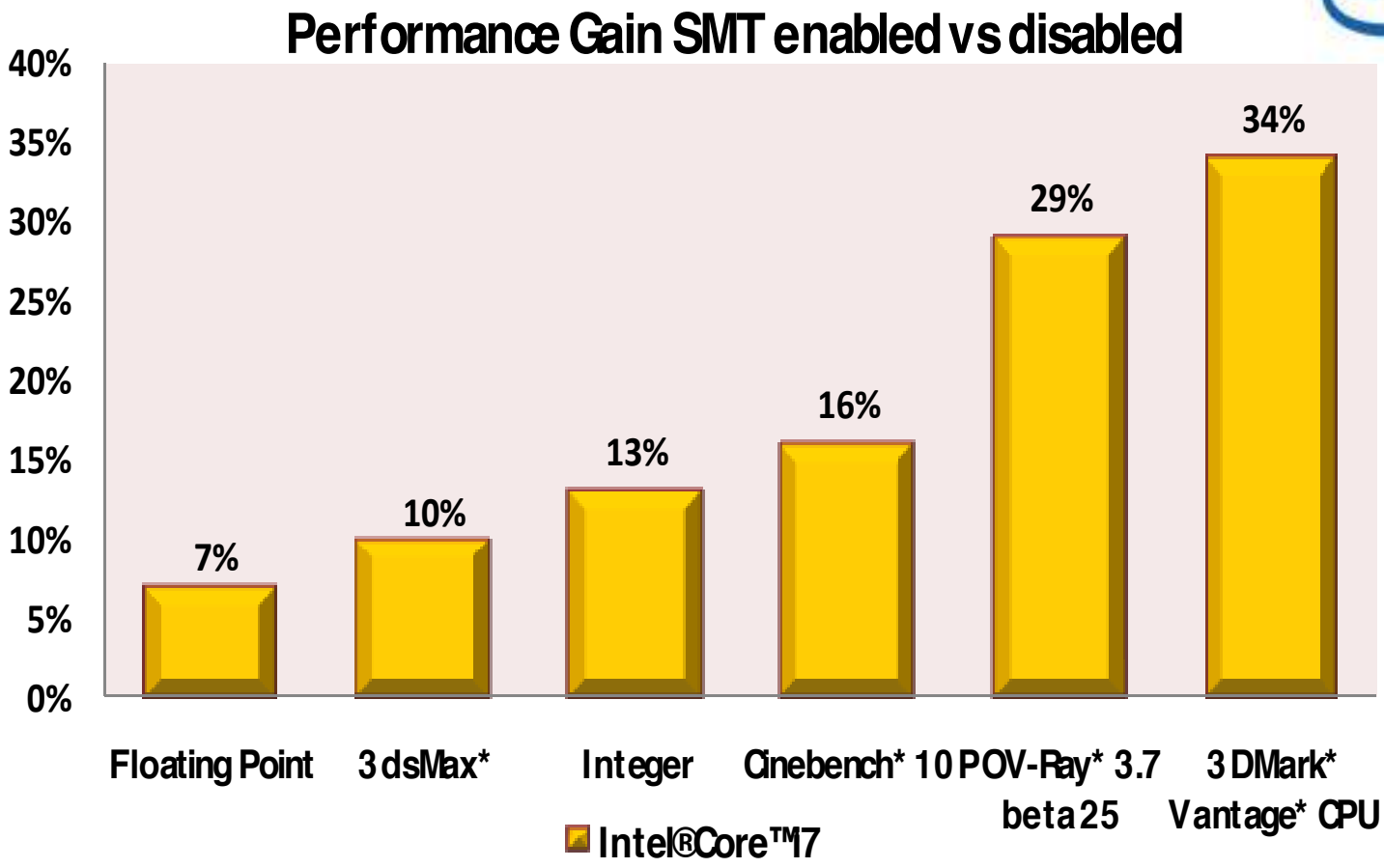
Execution pipeline has multiple thread selection points where the architecture can select to work for one of the 2 logical threads



- Select thread to fetch instructions from
- Select instruction to decode
- Select u-operation to allocate
- Select instruction to retire
- Additional selection points in memory pipeline like scheduling of MOB entries (memory order buffer)

Intel Multi-Threading

SMT Performance Chart



SPEC, SPECint, SPECfp, and SPECrate are trademarks of the Standard Performance Evaluation Corporation.
For more information on SPEC benchmarks, see: http://www.spec.org

Floating Point is based on SPECfp_rate_base2006* estimate
Integer is based on SPECint_rate_base2006* estimate

Intel Multi-Threading

- Threshold sharing applied at run time requires run time monitoring of resources utilization; additional hardware is necessary, and some computational overhead ensues.
- Complete sharing can also cause problems. This is true especially with cache memory. Sharing cache lines makes cache management simple, but what happens if both threads require each ¾ of the cache lines for a speedy execution?
  - A high number of cache miss, that would not arise, if only a single thread were executing (would coarse-grained multithreading be more efficient?)

CPU Multi-Threading

- Intel has temporarely dropped hyper-threading technology in dual core processors (dual-core processors are absed on an updated version of P6 microarchitecture, that does not support multithreading).
- Hyper-threading has been re-introduced since 2008 in Nehalem type processors.
- Other processors using multithreading are:
  - IBM Power 5 (2004) was dual core with 2 SMT;
  - IBM Power 7 (2010) 2-8 core with 4 SMT
  - UltraSPARC T2 (*Niagara*) and T3 (*Rainbow Fall*) have fine-grained multithreading with 8 threads per core