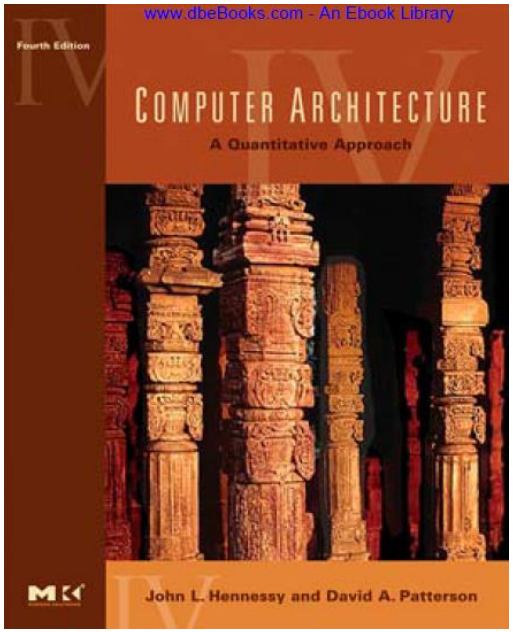
ILP Limits and Multithreading

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University



ILP Limitations

Pipelining, multiple issue, dynamic scheduling and speculation have limits.

Since late 90's focus turned to clock speedup, without increasing issue rates. 15 years later clock speedup ended too.

ILP limitations are subsequently studied. We compare a set of processors in performance and in efficiency measures per transistor and per watt.

Thread-level parallelism is proposed, as an alternative or addition to ILP.

Hardware Model

Assume first an ideal processor. The only limits on ILP are imposed by the actual data flows (RAW hazards) through either registers or memory.

1. **Register renaming**. There are an infinite number of virtual registers available. Hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

2. **Branch prediction** is perfect.

3. **Jump prediction** are perfect (computed jumps, return addresses). With perfect BP, it is equivalent to unbounded instruction buffer available for execution.

4. **All memory addresses are known exactly**, and a load can be moved before a store provided that the addresses are not identical.

5. **Perfect caches**. All memory accesses take 1 clock cycle.

Assumptions 1 and 4 eliminate **all but the true** data dependences, while assumptions 2 and 3 eliminate **all** control dependences.

The above assumptions enable **any** instruction be scheduled on the cycle immediately following the execution of the predecessor on which it depends.

A processor that can issue an unlimited number of instructions at once and looking arbitrarily far ahead in the computation is first examined.

There will be no restrictions on what types of instructions can be executed in a cycle.

For the unlimited-issue case, it means there may be an unlimited number of loads or stores issued in parallel.

All functional unit latencies are assumed to be 1 cycle, so that any sequence of dependent instructions can issue on successive cycles.

Such processor is unrealizable.

The **IBM Power5** was one of the most advanced superscalar processors announced on mid 2000.

It issued up to four instructions per clock and initiates execution of up to six (with restrictions on the instruction type, e.g., at most two load-stores).

It supported 88 renaming integer registers and 88 renaming FP registers.

It allowed over 200 instructions **in flight** (instructions in execution at any point), of which up to 32 could be loads and 32 could be stores).

It used aggressive BP.

After looking at the perfect processor, the impact of restricting various features is examined.

To measure the available parallelism, a set of programs is executed to produce a trace of the instruction and data references.

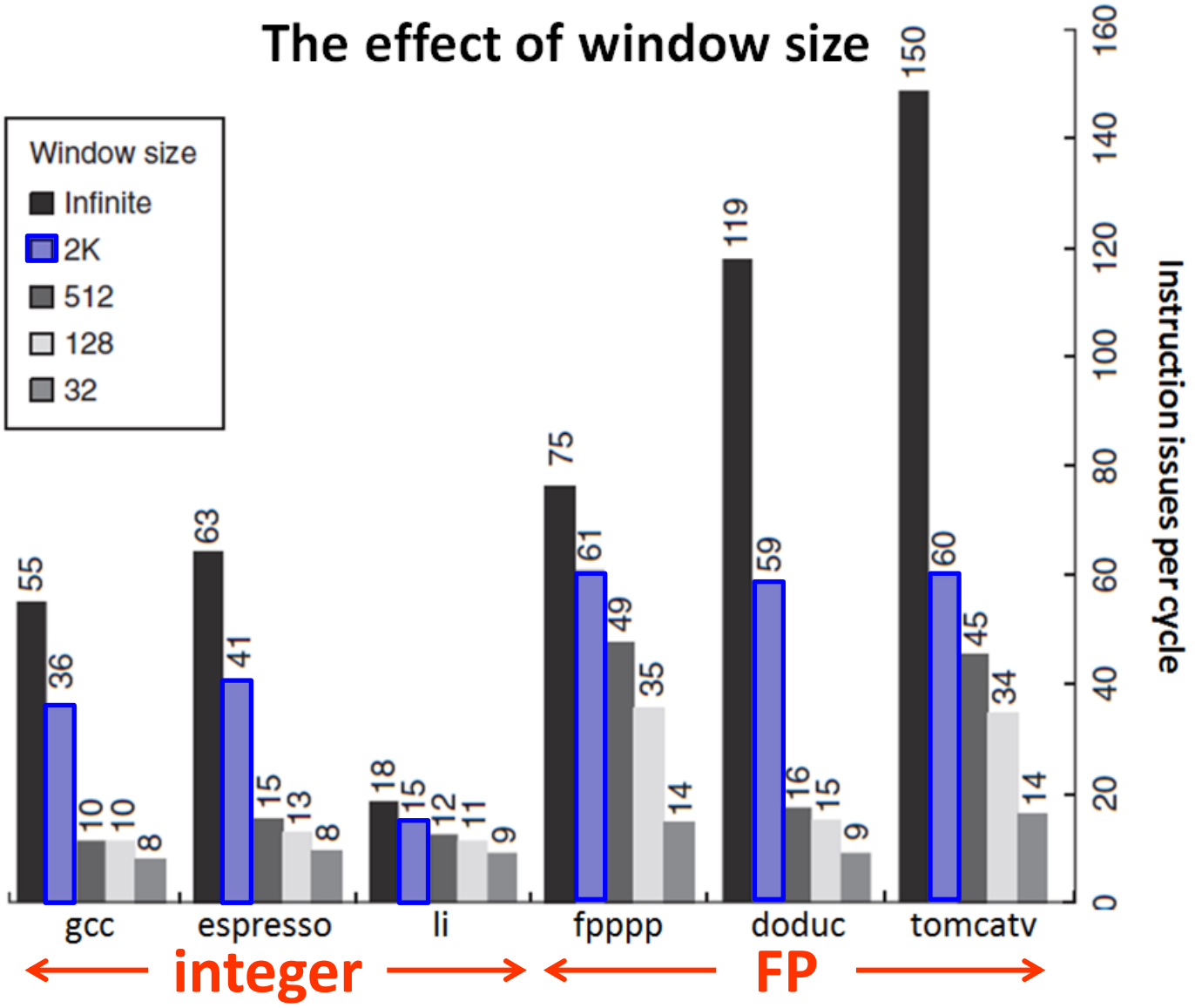
Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences. Since a trace is used, perfect BP is easy to do.

With these mechanisms, instructions may be scheduled much earlier than they would otherwise.

The **window size** directly limits the number of instructions that **begin execution** in a given cycle.

Real processors have a limited number of functional units, limited numbers of buses and register ports, limiting the number of instructions initiated per clock.

Thus, the maximum number of instructions that may issue, begin execution, or commit in the same clock cycle is usually **much smaller** than the **window size**.


The effect of window size
Window size
Infinite
2K
512
128
32
Instruction issues per cycle
0
20
40
60
80
100
120
140
160
55
36
10
10
8
63
41
15
13
8
18
15
12
11
9
75
61
49
35
14
119
59
16
15
9
150
60
45
34
14
gcc
espresso
li
fpppp
doduc
tomcatv
integer
FP

The amount of parallelism uncovered falls sharply with decreasing window size.

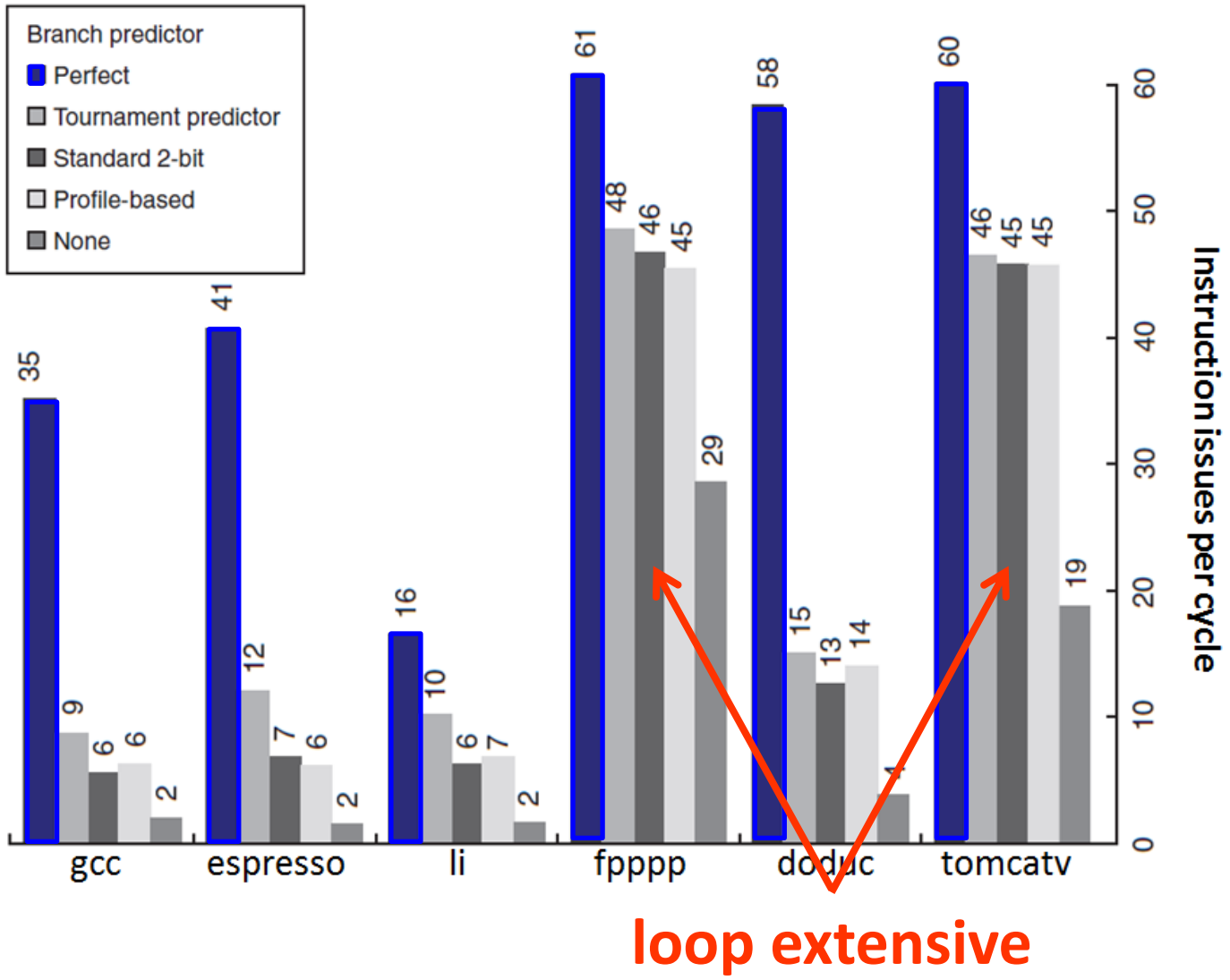
In 2005, the most advanced processors had window sizes in the range of 64–200.

Assume for the rest of this analysis:

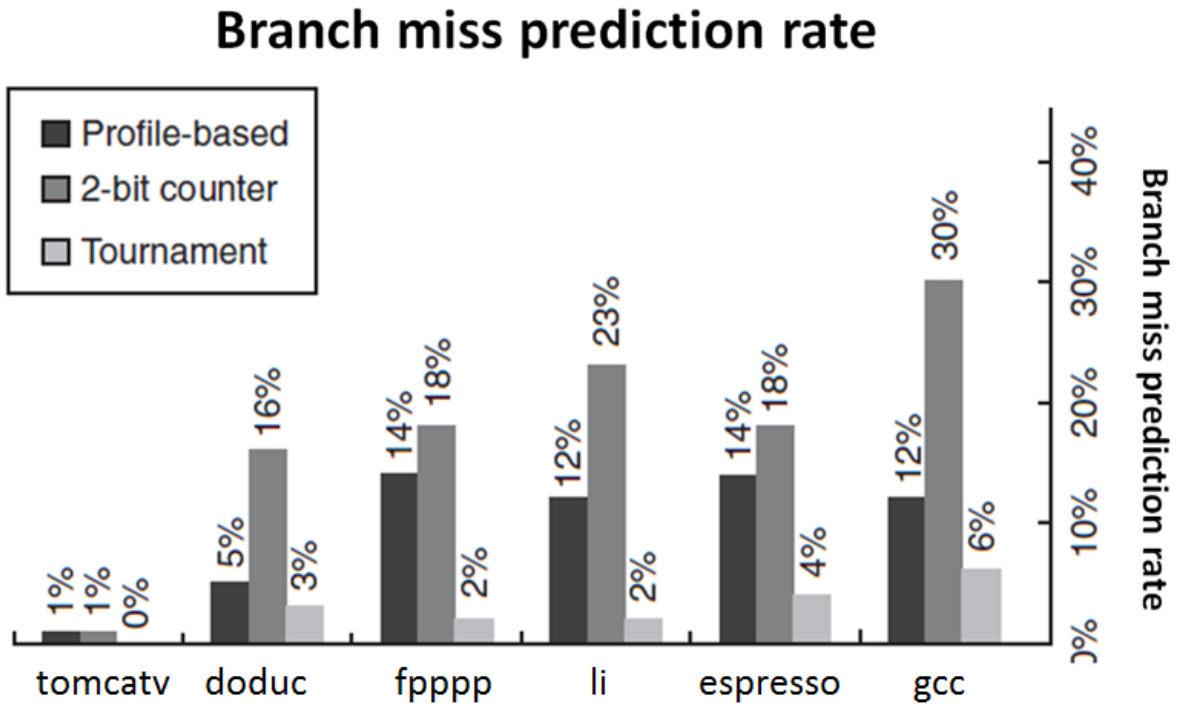
1) window of 2K entries, X10 than largest implementation in 2005, and

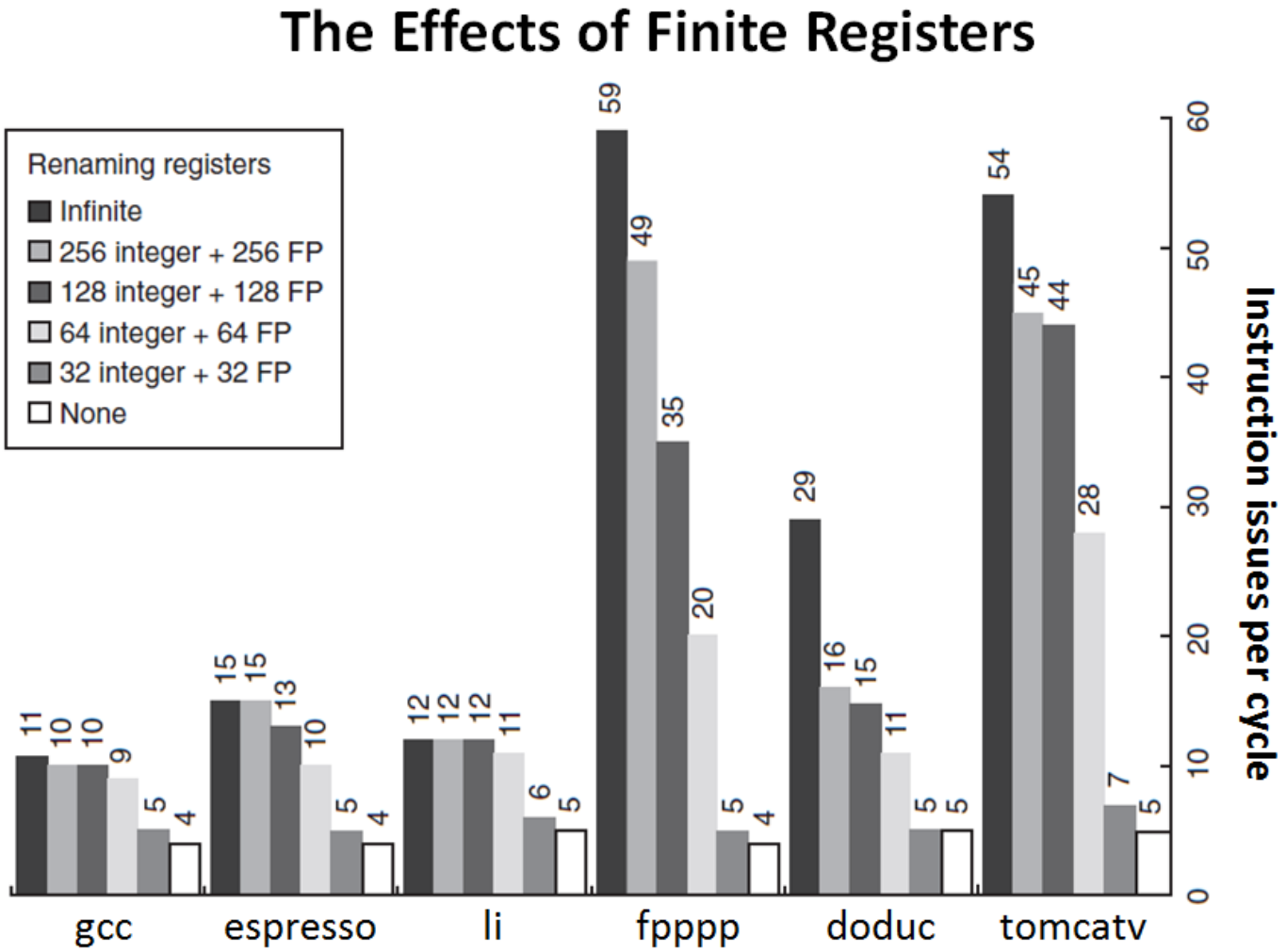
2) max issue capability of 64 instructions per clock, X10 times the widest issue processor in 2005.

The effect of branch-prediction schemes

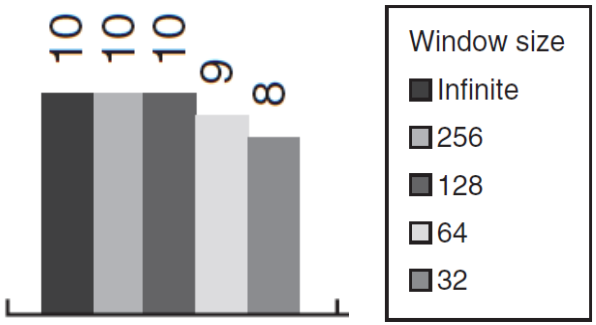


The branch behavior of the two FP programs is much simpler than the others because they have many fewer and more predictable branches, allowing exploitation of significant amounts of parallelism.




The Effects of Finite Registers
Renaming registers
Infinite
256 integer + 256 FP
128 integer + 128 FP
64 integer + 64 FP
32 integer + 32 FP
None
11 10 10 9 5 4
15 15 13 10 5 4
12 12 12 11 6 5
59 49 35 20 5 4
29 16 15 11 5 5
54 45 44 28 7 5
gcc
espresso
li
fpppp
doduc
tomcatv
0 10 20 30 40 50 60
Instruction issues per cycle

Example. Consider three processors running on benchmark achieving ideal issue rate as shown.



1. A 4.0 GHz simple MIPS of two-issue static pipe, achieving 0.8 CPI. Its cache yields 0.5% misses per instruction.

2. A deeply pipelined 5.0 GHz MIPS of two-issue static pipe, achieving 1.0 CPI. It has smaller cache yielding 0.55% misses per instruction.

3. A 2.5 GHz speculative superscalar MIPS with a 64-entry window. It achieves one-half of the ideal issue rate shown above.

It has the smallest caches, yielding 1% misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling.

Cache miss penalty is 50 nSec in all processors. Determine the relative **MIPS** ($10^6\ instructions/sec$) performance of these three processors.

Answer. Miss rate is first used to compute the contribution to CPI from cache misses.

$$\text{Cache Miss CPI} = \text{Misses per instruction} \times \text{Miss penaly}$$

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

$$\text{Miss penalty}_1 = 50 \text{ nSec}/250 \text{ pSec} = 200 \text{ cycles}$$

$$\text{Miss penalty}_2 = 50 \text{ nSec}/200 \text{ pSec} = 250 \text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.75 \times 50 \text{ nSec}}{400 \text{ pSec}} = 94 \text{ cycles}$$

The miss penalties are combined with the miss rates to yield miss penalties in CPI.

$$\text{Cache Miss CPI}_1 = 0.005 \times 200 = 1.0$$

$$\text{Cache Miss CPI}_2 = 0.0055 \times 250 = 1.4$$

$$\text{Cache Miss CPI}_3 = 0.01 \times 94 = 0.94$$

The pipeline CPI is known for the first two processors. To obtain the pipeline CPI of the 3rd, there is

$$\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = 0.22$$

The CPI is the sum of the pipeline and cache CPIs

$$\text{CPI}_1 = 0.8 + 1.0 = 1.8$$

$$\text{CPI}_2 = 1.0 + 1.4 = 2.4$$

$$\mathbf{CPI_3 = 0.22 + 0.94 = 1.16}$$

All the three MIPS processors can be compared in terms of their MIPS.

$$\mathbf{Instruction\ execution\ rate = \frac{Clock\ rate}{CPI}}$$

$$\mathbf{Instruction\ execution\ rate_1 = \frac{4000MHz}{1.8} = 2222\ MIPS}$$

$$\mathbf{Instruction\ execution\ rate_2 = \frac{5000MHz}{2.4} = 2083\ MIPS}$$

$$\mathbf{Instruction\ execution\ rate_3 = \frac{2500MHz}{1.16} = 2155\ MIPS}$$

Multithreading

Data access latency occurs by cache misses (L1, L2), memory latency, often unpredictable, data hazards.

ILP is advantageous since it is transparent to the programmer, but it can be limited.

Multithreading (MT) tolerates or masks long and often unpredictable latency operations by **switching to another context**, which is able to do useful work.

There may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the ILP techniques.

The higher-level parallelism is called **Thread-Level Parallelism** (**TLP**) because it is logically structured as separate threads of execution.

A **thread** is a separate process with its own instructions and data.

It may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own.

Each thread has all the state (instructions, data, PC, register state, etc.) necessary to allow it to execute.

TLP is an important **alternative** to ILP because it could be more cost-effective to exploit. **TLP** and **ILP** exploit two **different** kinds of parallel structure in a program.

It happens that the functional units of a data-path designed to exploit ILP are often idle because of either stalls or dependences.

A natural question is therefore whether it is possible for an ILP oriented processor to exploit TLP.

Question: Could the parallelism among threads be used as a source of independent instructions that might keep the processor busy during stalls?

Question: Could TLP be used to employ the functional units that would otherwise be idle when insufficient ILP exists?

Multithreading (**MT**) allows multiple threads to share the functional units of a single processor.

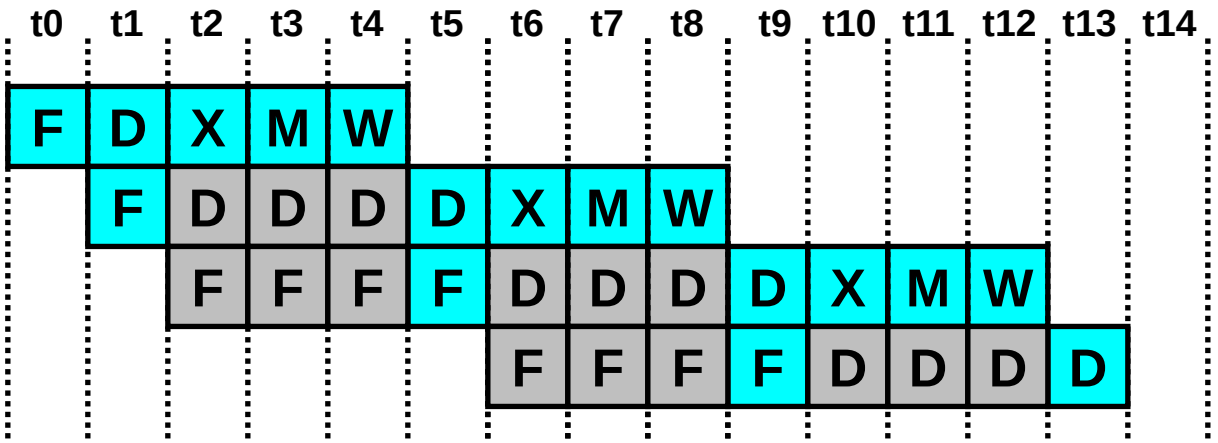
To permit this sharing, the processor must duplicate the independent state of each thread, including RF, PC and page table.

The memory itself can be shared through the VM mechanisms, already supporting multiprogramming.

The HW must support quick thread switching.

A thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

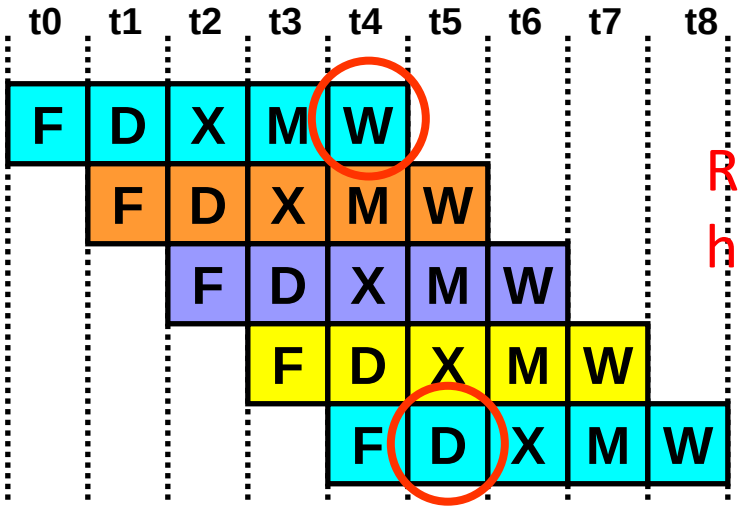
LW r1, 0(r2)
LW r5, 12(r1)
ADDI r5, r5, #12
SW r5, 12(r1)



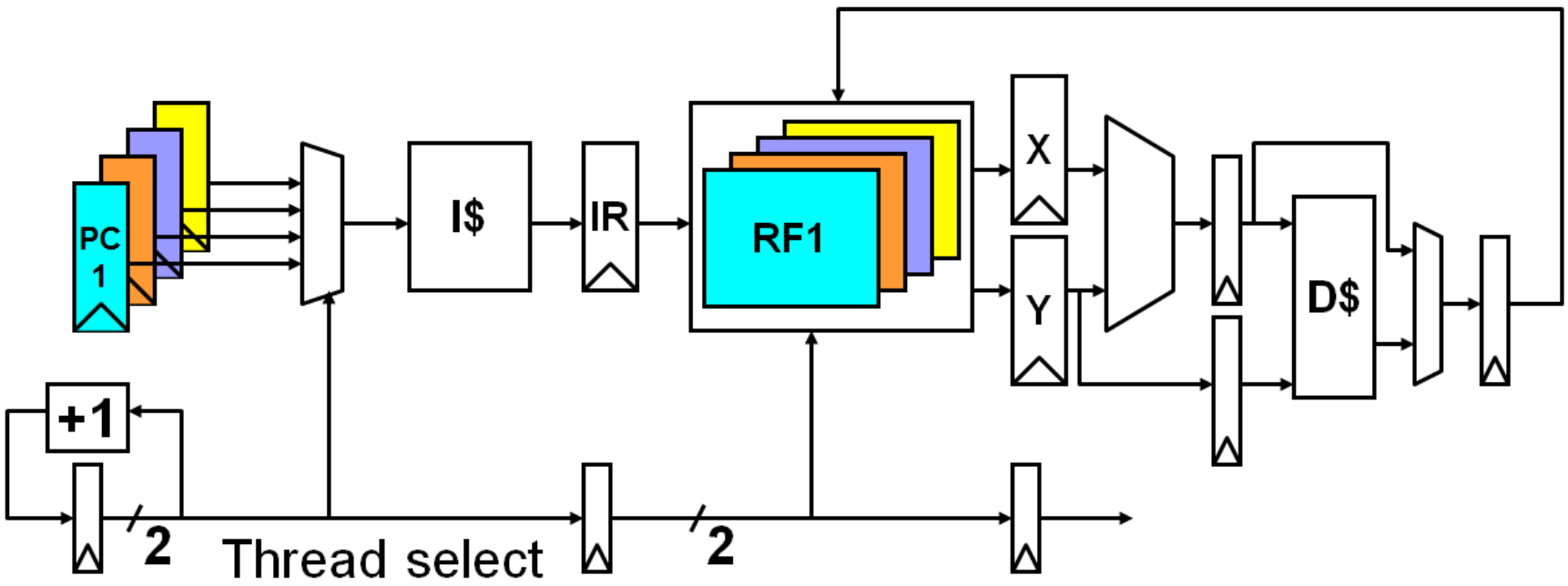
Each instruction may depend on the prevoius.

Interleave 4 threads on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW r5, 0(r7) ???
T1: LW r5, 12(r1)



Resolve RAW hazards

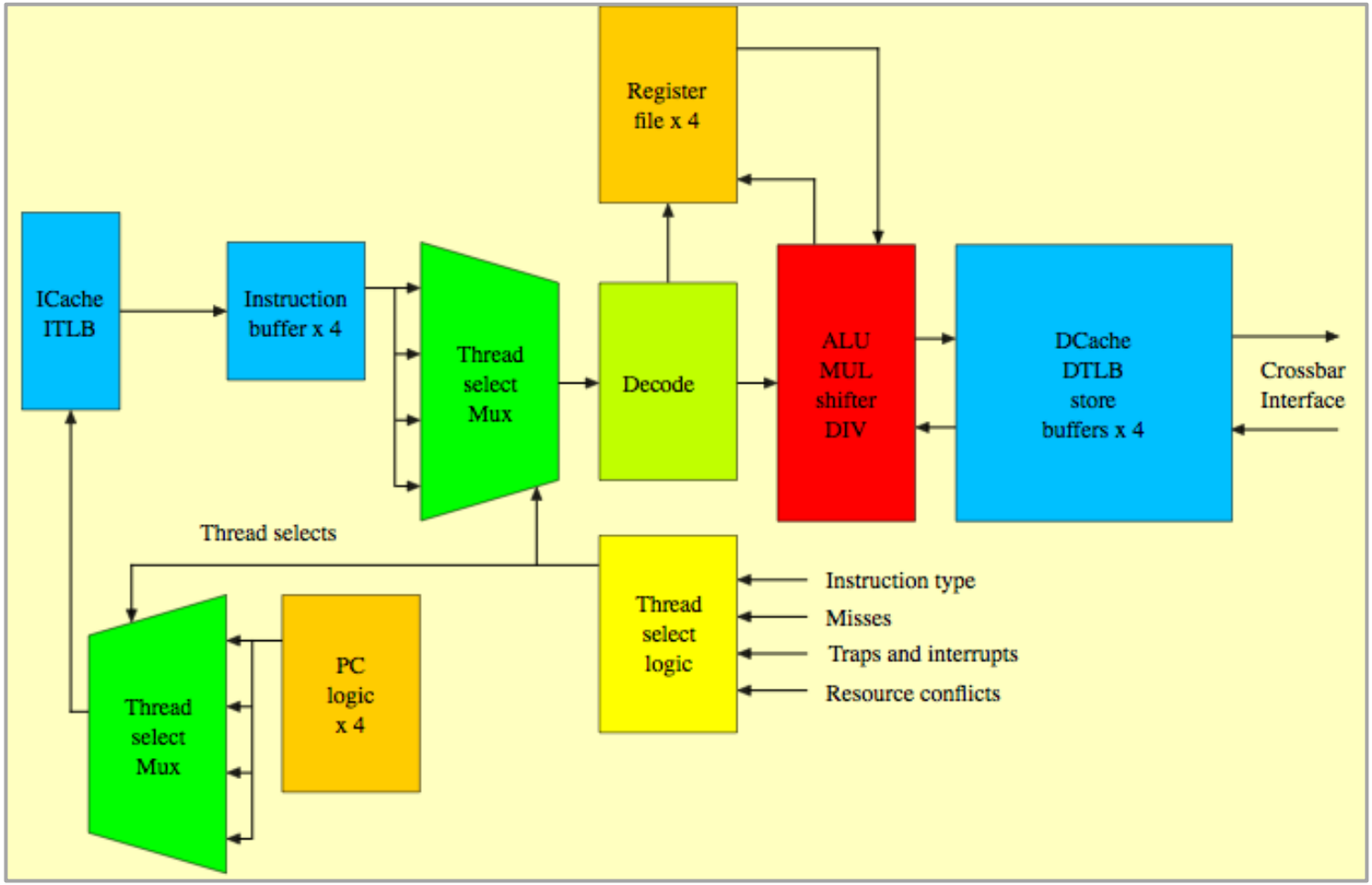


Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage.

Appears to SW (OS) as multiple, albeit slower, CPUs.

Also, needs its own system state VM page table base register, exception handling registers, and more.

Sun Niagara MT Pipeline



MT Challenges

I-Cache: Instruction bandwidth, instruction locality is degraded and the I-cache miss rate rises.

Register file access time: Increases due to the fact that RF had to significantly increase in size to accommodate many separate contexts.

May use SRAM to implement the RF, which means longer access times.

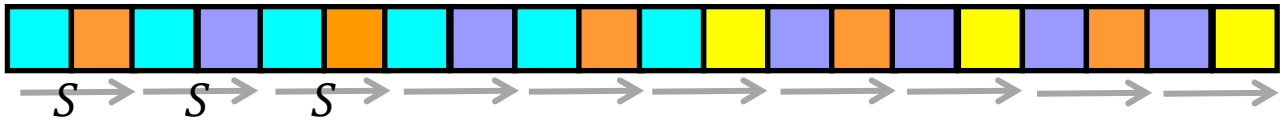
Single thread performance: Significantly degraded since the context is forced to switch to a new thread even if none are available.

Very high bandwidth network, which is fast and wide.

Thread Scheduling Policies

Fixed interleave: Each of N threads executes one instruction every N cycles. If thread not ready to go in its slot, insert pipeline bubble.

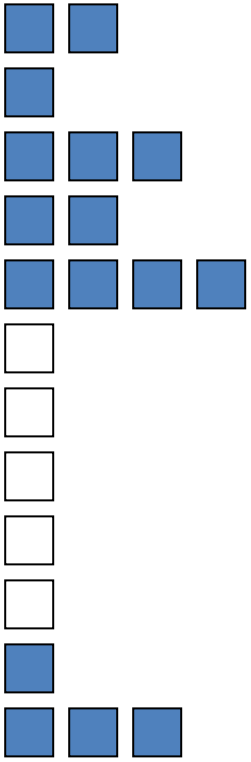
SW-controlled interleave: OS allocates S pipeline slots amongst N threads. HW performs fixed interleave over S slots, executing whichever thread is in that slot.



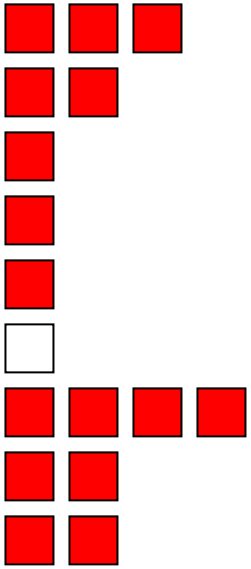
HW-controlled thread scheduling: HW keeps track of which threads are ready to go. Picks next thread to execute based on HW priority scheme.

4-issue machine

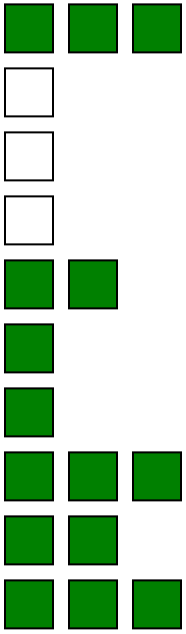
Thread A



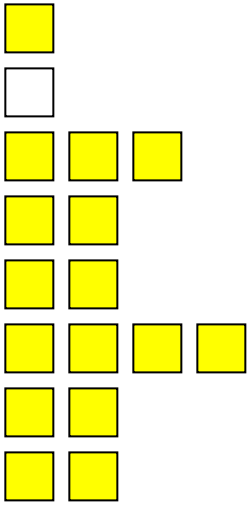
Thread B



Thread C



Thread D



There are two main approaches to MT, **Fine-grained MT** and **Coarse-grained MT**.

Coarse-grained MT switches threads only on costly stalls, such as L2 misses.

The processor is not slowed down (by thread switching), since instructions from other threads will only be issued when a thread encounters a costly stall.

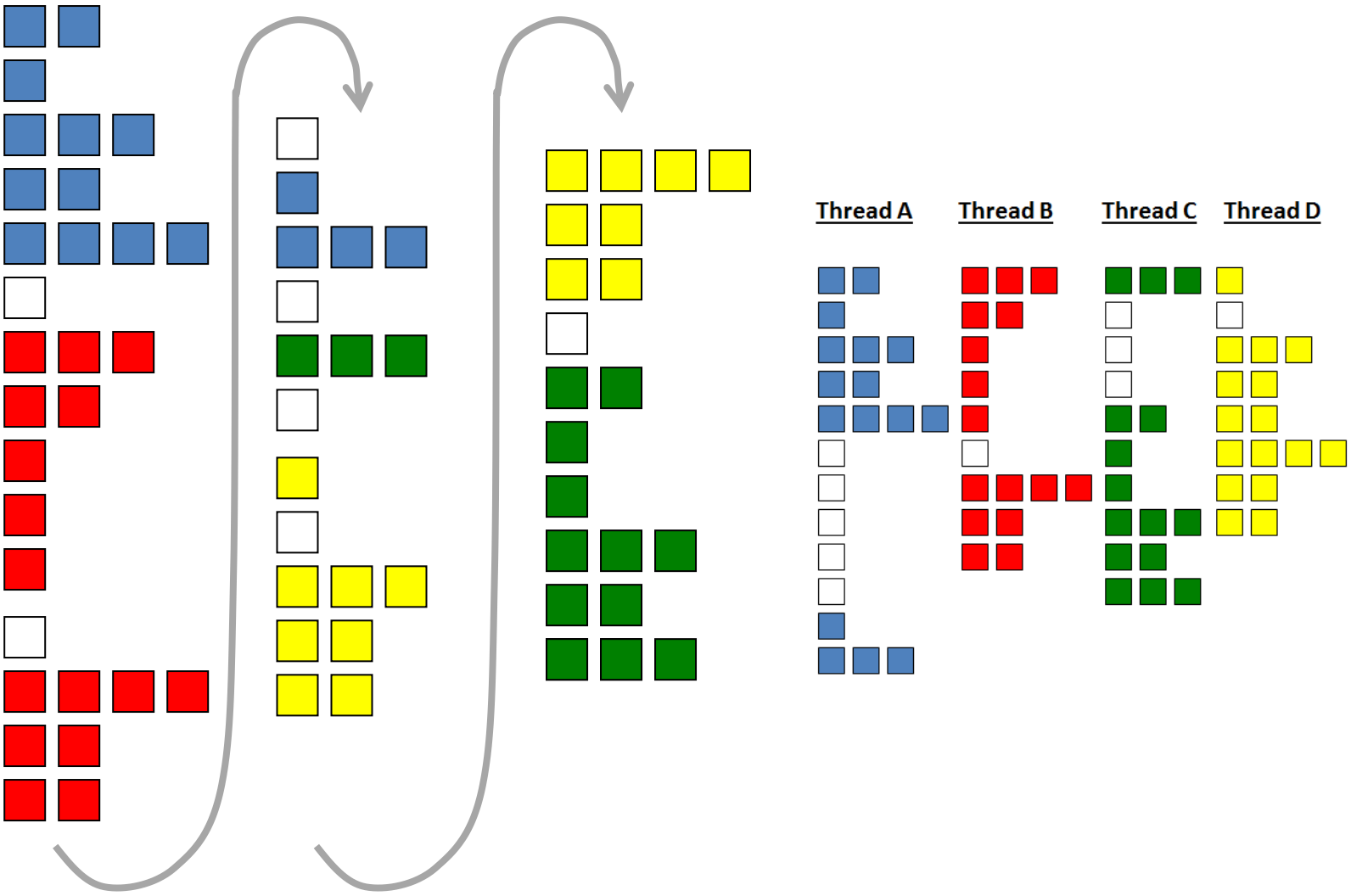
It is however limited in overcoming throughput losses caused by short stalls (employment of FUs).

Since a CPU with coarse-grained MT issues instructions from a single thread, when a stall occurs the pipeline must be emptied.

The new thread must fill the pipeline before instructions will be able to complete.

Because of this start-up overhead, coarse-grained MT is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Coarse-grained MT



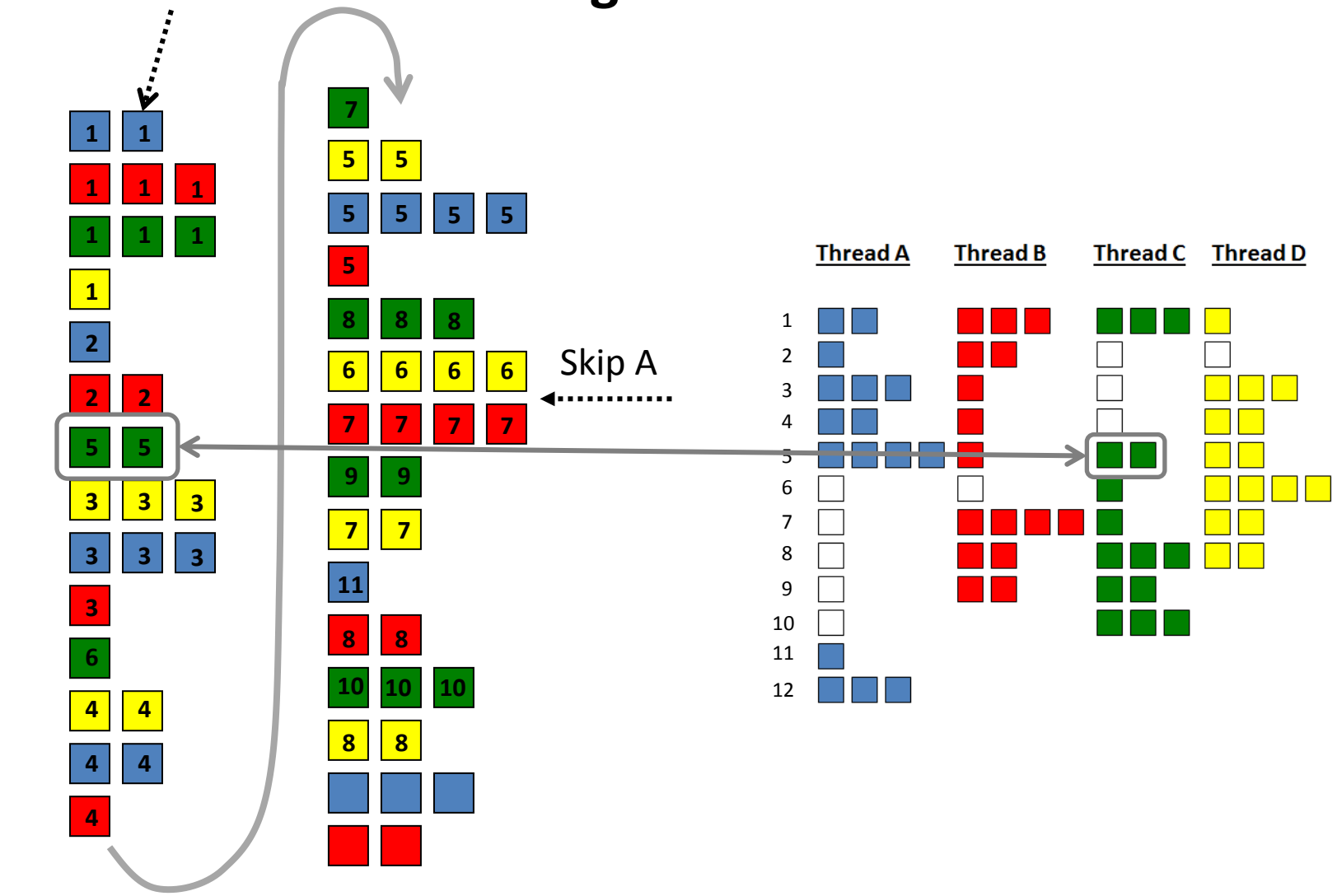
Fine-grained MT switches between threads on each instruction, causing multiple threads to be interleaved.

It is often done in a round-robin fashion, skipping any threads that are stalled at that time. The CPU must switch threads on every clock cycle.

An advantage is that it hides the throughput losses arising from short stalls, since instructions from other threads are executed when one thread stalls.

A disadvantage is the slowdown in executing individual threads, since a thread that is ready to execute without stalls is delayed by other threads.

Fine-grained MT



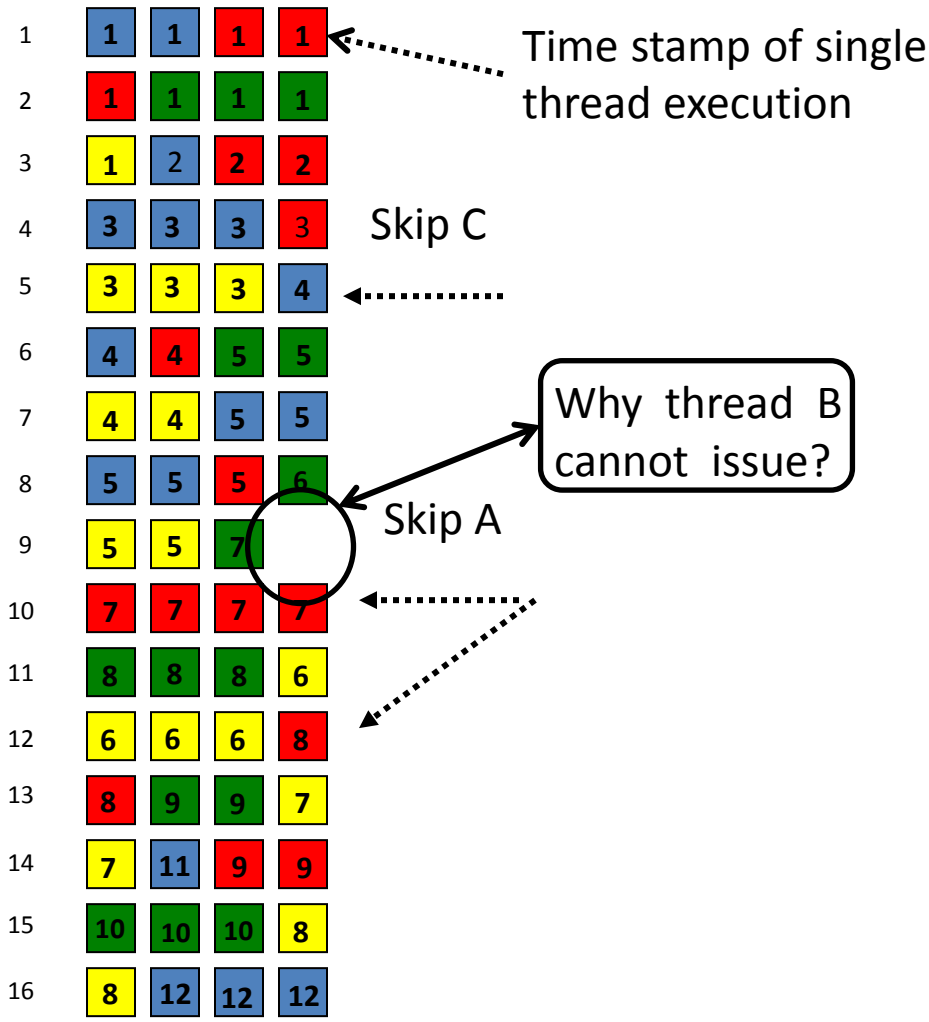
Simultaneous Multithreading

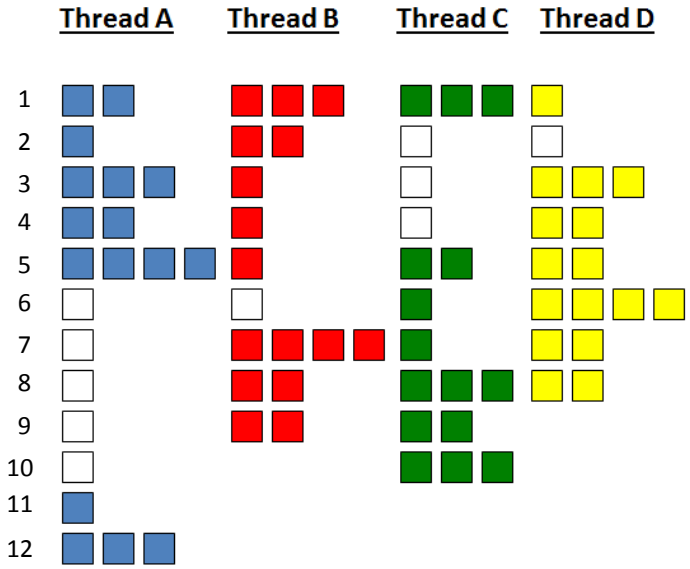
Simultaneous multithreading (**SMT**) is a variation on MT to exploit **TLP** simultaneously with **ILP**.

SMT is motivated by **multiple-issue** processors which have more functional unit parallelism than a single thread can effectively use.

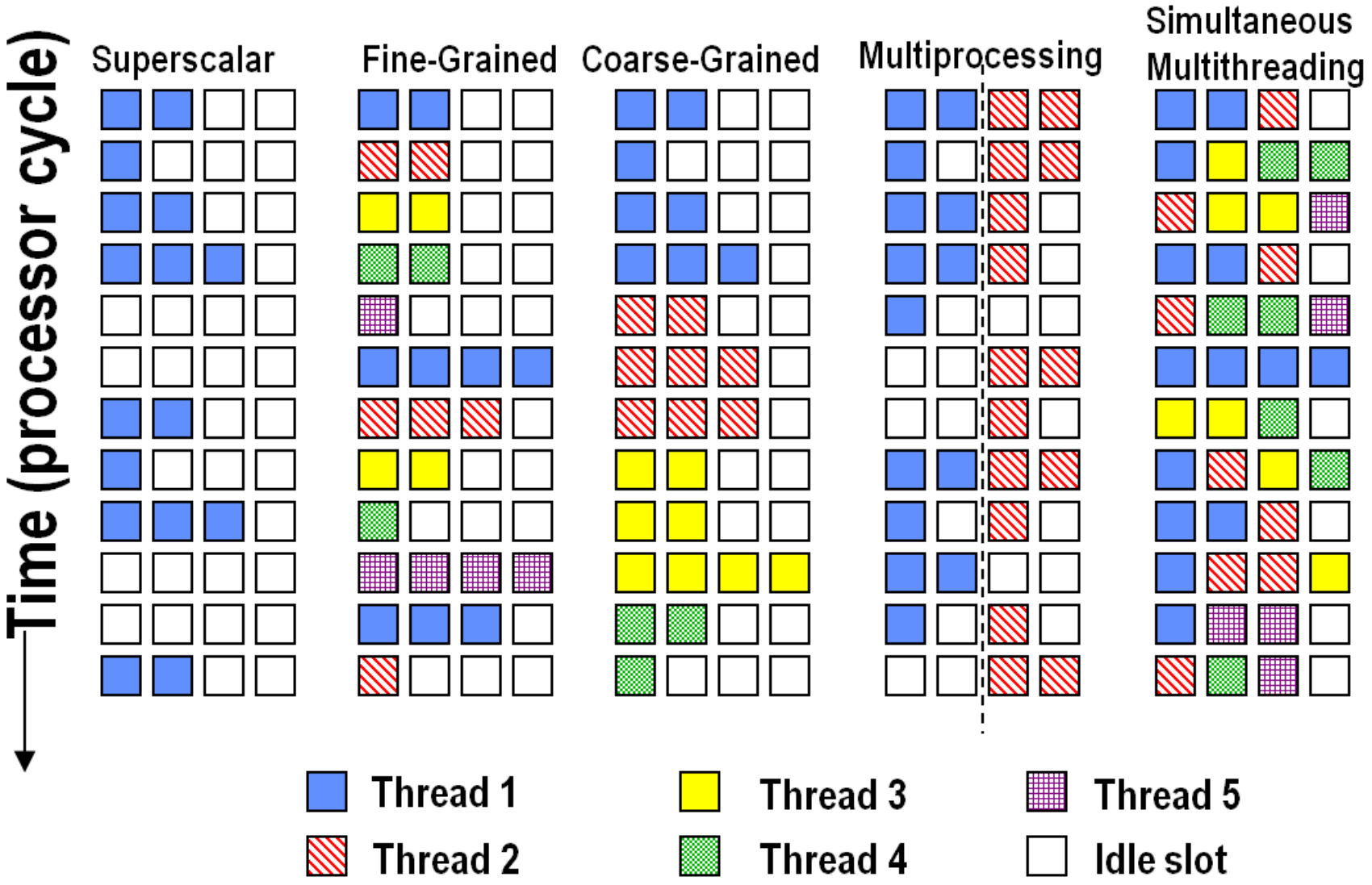
Register renaming and dynamic scheduling enables issuing multiple instructions from **independent threads ASAP** regardless of the **dependences** among them.

Dependences resolution can be handled by the dynamic scheduling capability (**Tomasulo, ROB**).


1 1 1 1
1 1 1 1
1 2 2 2
3 3 3 3
3 3 3 4
4 4 5 5
4 4 5 5
5 5 5 6
5 5 7
7 7 7 7
8 8 8 6
6 6 6 8
8 9 9 7
7 11 9 9
10 10 10 8
8 12 12 12
Time stamp of single thread execution
Skip C
Why thread B cannot issue?
Skip A


Thread A
Thread B
Thread C
Thread D

Approaches to use the issue slots.



In **superscalar** without MT, the use of issue slots is limited by a lack of ILP. A major stall, such as cache miss, can leave the entire processor idle.

In **coarse-grained MT** superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.

Although reducing the number of completely idle clock cycles, the ILP limitations still lead to idle cycles.

Since thread switching only occurs when there is a stall and the **new thread** has a **start-up period**, some fully idle cycles will still remain.

In **fine-grained MT**, the interleaving of threads eliminates fully empty slots.

Because only one thread issues instructions in a given clock cycle, ILP limitations still lead to idle slots within individual clock cycles.

In **SMT**, **TLP and ILP are exploited** simultaneously, with multiple threads using the issue slots in a single clock cycle.

The issue slot usage is still limited by imbalances in the resource needs and resource availability over multiple threads.

Other factors, such as how many active threads are considered, buffers limits, instruction fetch limits, can also restrict how many slots are used.

SMT uses the fact that many of the HW mechanisms needed to support TLP already exist in a **dynamically scheduled** processor.

Dynamically scheduled processor have a large set of virtual registers.

These can be used to hold the register sets of independent threads (assuming separate renaming tables are maintained for each thread).

Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads.

This allows to build MT on top of an OOO processor by adding a **per-thread renaming table**, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

The independent commitment of instructions from separate threads can be supported by keeping a **separate reorder buffer** for each thread.