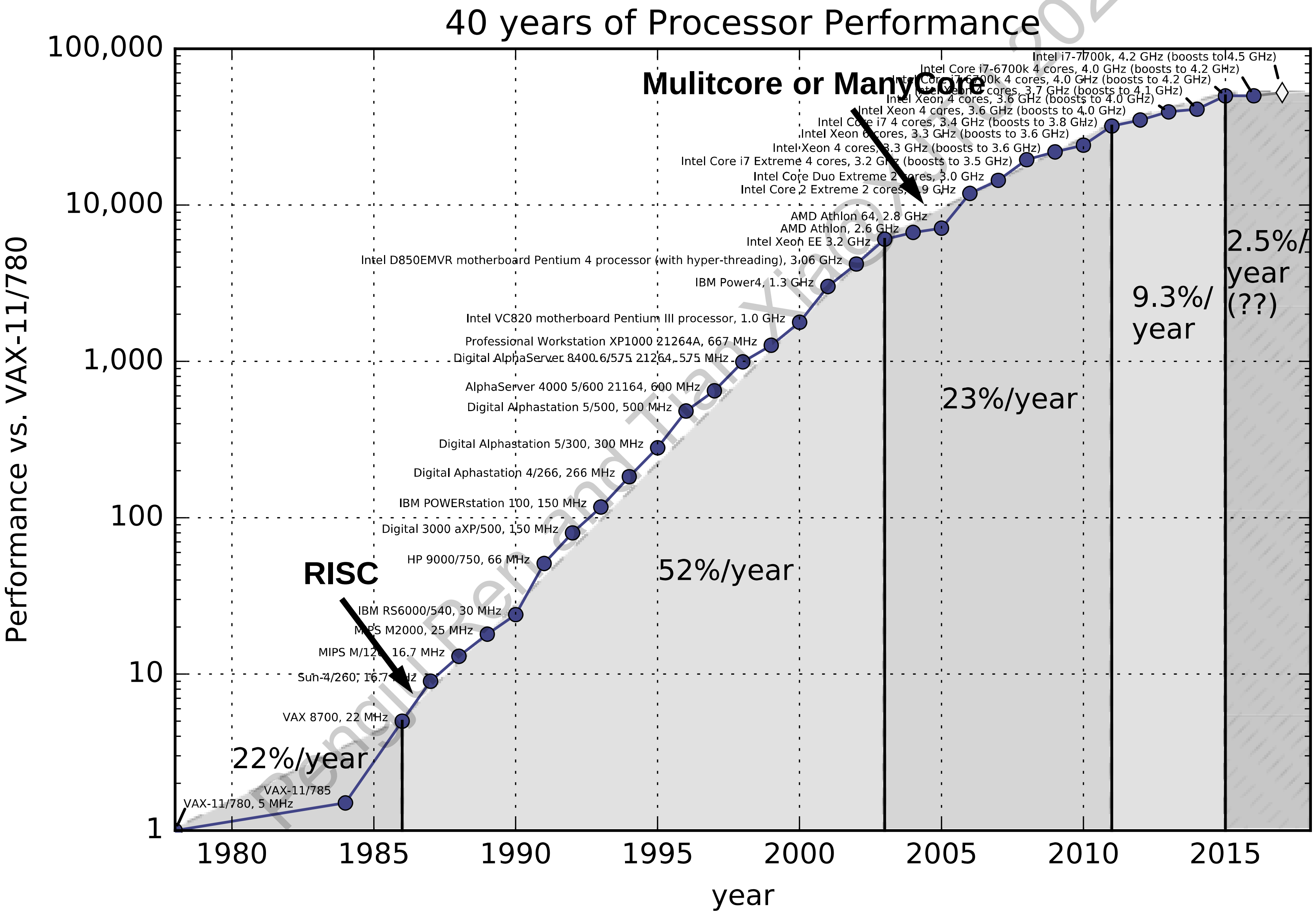
Computer Architecture

Lecture 11 – Multithreading and Multicore (Thread-level Parallel)

Tian XIA
Institute of Artificial Intelligence and Robotics
Xi’an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

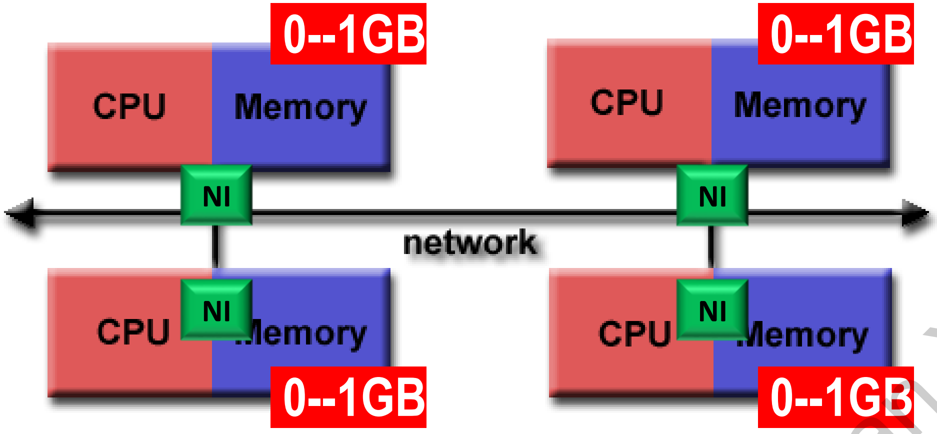
Recap: Processor Performance

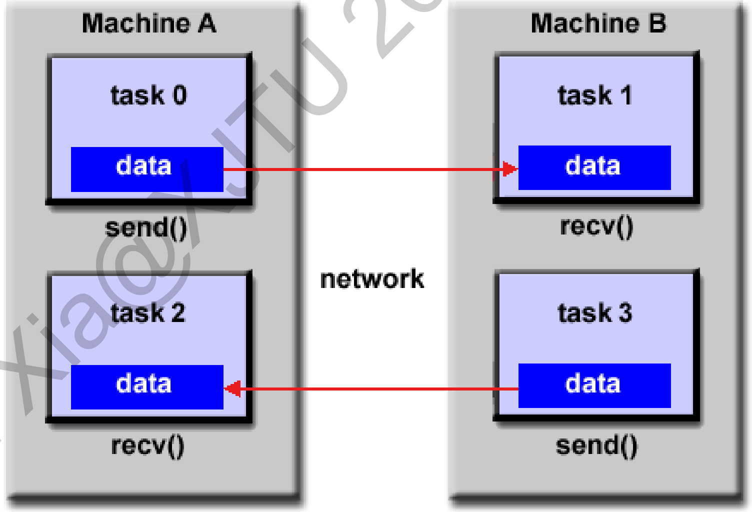


[Hennessy & Patterson, 2017]

Distributed Memory (Message Passing)

(Loosely coupled multiprocessors)



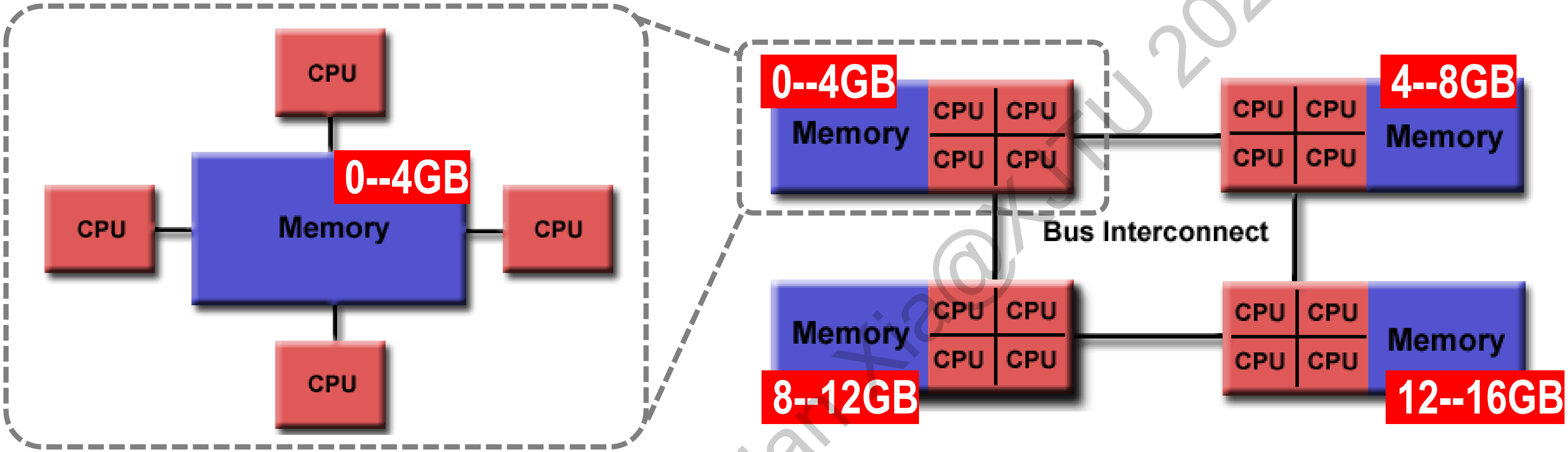


- Each processors have their **own local memory**, and operates **independently**.
- When a processor needs access to data in another processor, it is usually the task of the **programmer** to **explicitly** define how and when data is communicated.
- ***Message Passing Interface (MPI)*** is the "de facto" industry standard for message passing

MPI tutorial: hpc-tutorials.llnl.gov/mpi/

Shared Memory (Symmetric & unsymmetric)

(Tightly coupled multiprocessors)



Uniform Memory Access (UMA):

- Most commonly represented today by **Symmetric Multiprocessor (SMP)** machines
- Identical processors
- **Equal latency** when access memory
- Sometimes called **CC-UMA (Cache Coherent UMA)**. Cache coherency is accomplished at by hardware.

Non-Uniform Memory Access (NUMA):

- Often made by **physically linking** two or more SMPs
- One SMP can **directly access** memory of another SMP
- Processors have **non-equal access time** to different memories
- Memory access **across link** is slower
- If cache coherency is maintained, then may also be called **CC-NUMA (Cache Coherent NUMA)**

Distributed vs. Shared (A Concept View)

- **Appearance of memory to *software (Programmer's view point)***

  Q: Can processors communicate directly via memory?

  – **Distributed (message passing):** no, communicate via messages

  – **Shared (shared memory):** yes, communicate via load/store

- **Appearance of shared memory to *hardware (Architect's view point)***

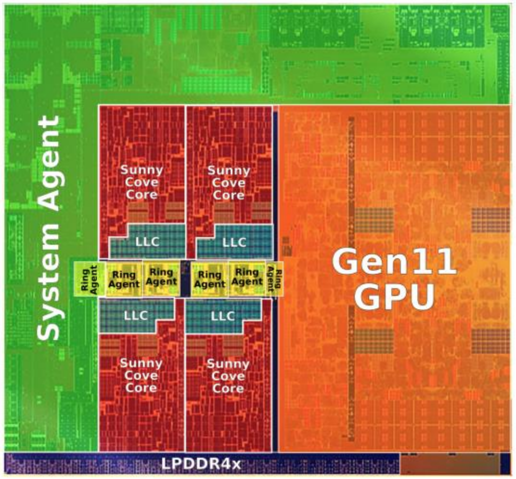
  Q: Memory access latency uniform for Shared Memory?

  – **Uniform Memory Access(UMA):** yes, doesn't matter where data goes

  – **Non-Uniform Memory Access(NUMA):** no, makes a big difference

Multi-/Many-Core @ Intel, AMD, ARM, …

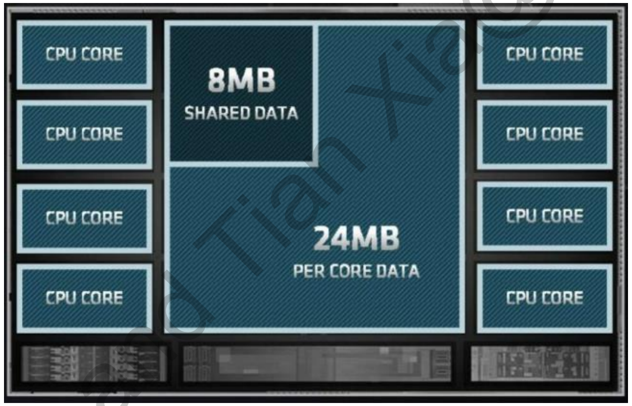
Today general-purpose "multicore" processors implement **NUMA (not SMP)** on a single chip is everywhere, Server, Laptop and Mobile Phone.



Intel Xeon

Ice Laker (Gen10)@10nm

#Core: 8~40 #Threads: 16~80



AMD Zen3

EPYC@7nm

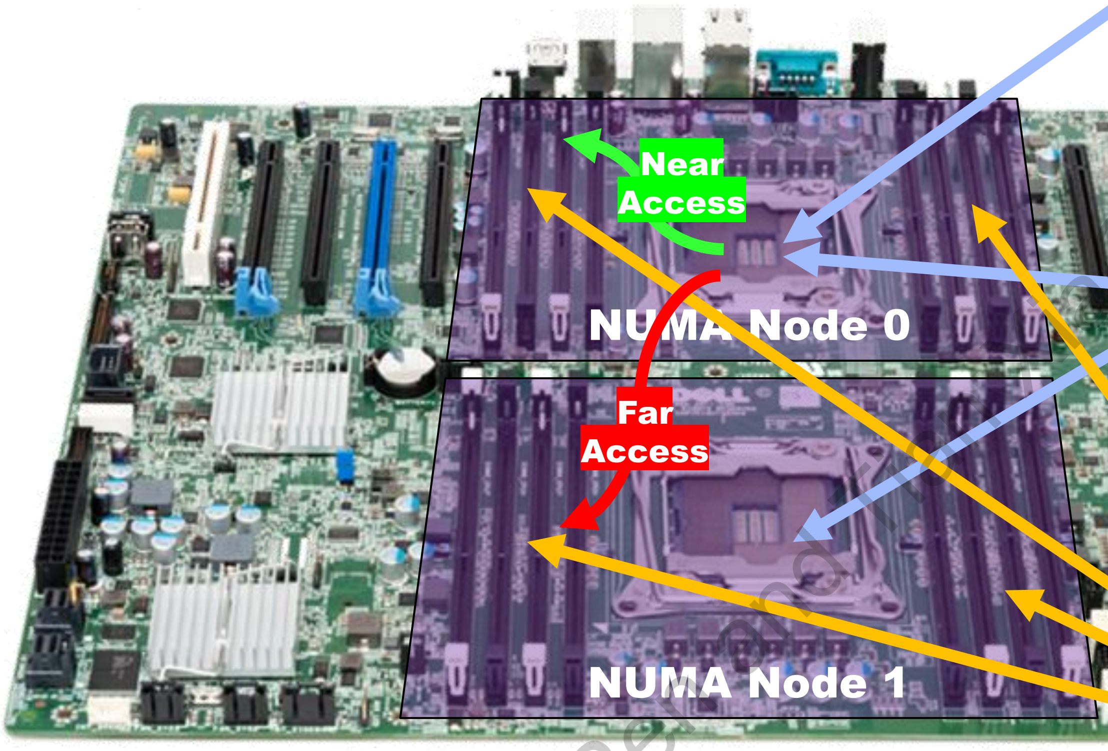
#Core: 6~64 #Threads: 12~128

阿里

倚天 710@5nm

128 Core

Multi-/Many-Core @ Intel, AMD, ARM, …



**PROCESSOR TYPE:
INTEL XEON E5-2600
V4 SERIES**

**NUMBER OF PROCESSORS
SUPPORTED: UP TO 2
PROCESSORS**

**DELL PRECISION TOWER
7910 WORKSTATION**

**NUMBER OF MEMORY SLOTS:
16 TOTAL MEMORY SLOTS (8
SLOTS FOR EACH CPU)**

MEMORY TYPE: DDR4

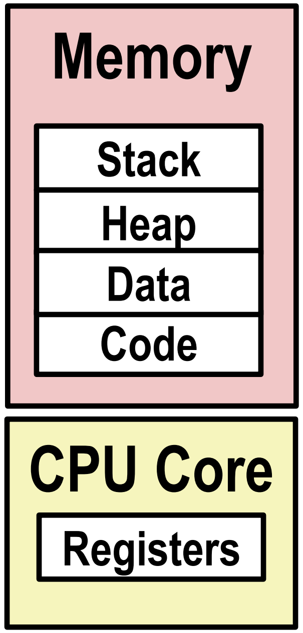
**MEMORY SIZE CAPACITY: UP
TO 1.0TB (FULLY POPULATED)**

Terminology (Program & Process)

- **Program**: An executable task
  – Example: Calculating the *sum of 1,000,000* number
- **Process**: An instance of a running program or portion of program
  – Example a running instance of *sum of 1,000,000*
- **Process** provides each program with two **key abstractions**:

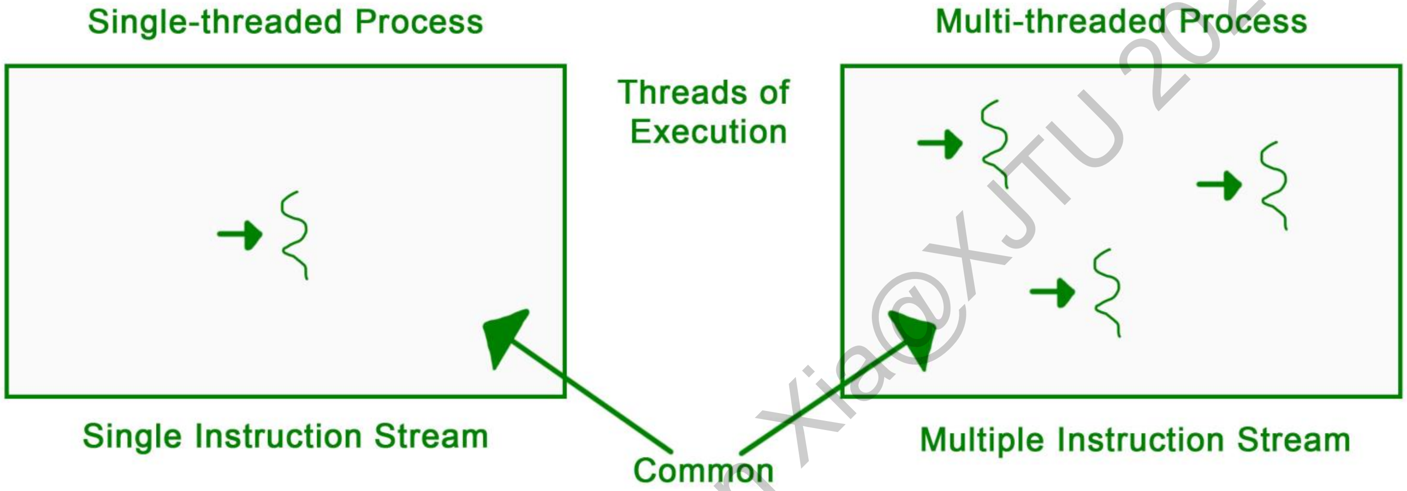
  ***Logical control flow*：** Each process seems to have exclusive use of the CPU

  ***Private address space(Virtual Memory)*：** Each process seems to have exclusive use of main memory.



Thread as the subset of a process

(a.k.a the lightweight process)

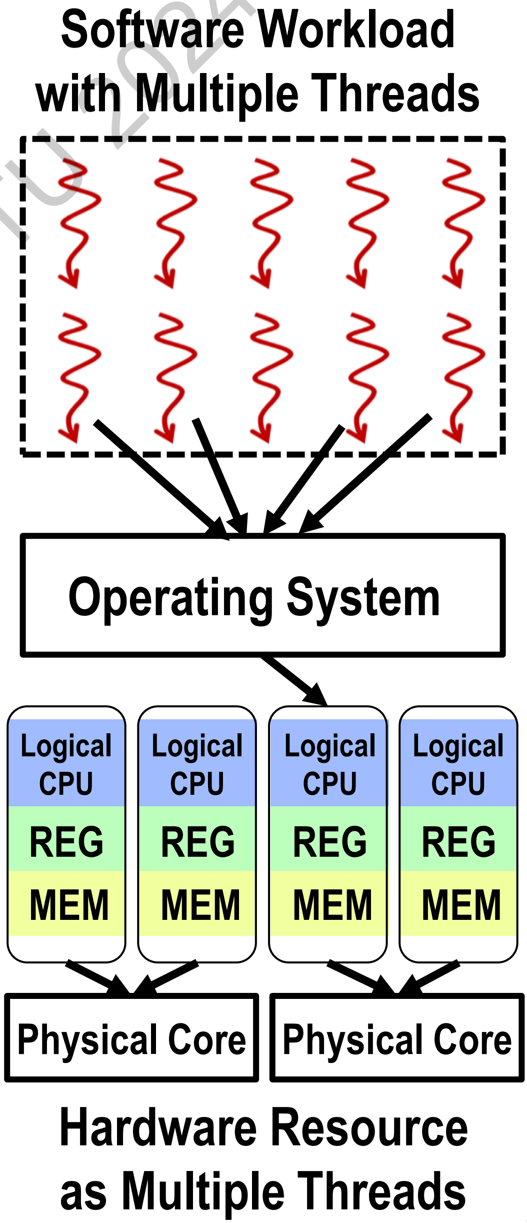


- **Sequential flow** of instructions that performs some task, each thread has:
  - ***- Dedicated PC (program counter)***
  - ***- Separate registers***
  - ***- Private variables (local stack variables)***
  - ***- Accesses the shared memory (static variables, global heap)***
- **Threads communicate** implicitly by writing/reading shared variables (next lecture!)
- **Threads coordinate** by synchronizing on shared variables (next lecture!)

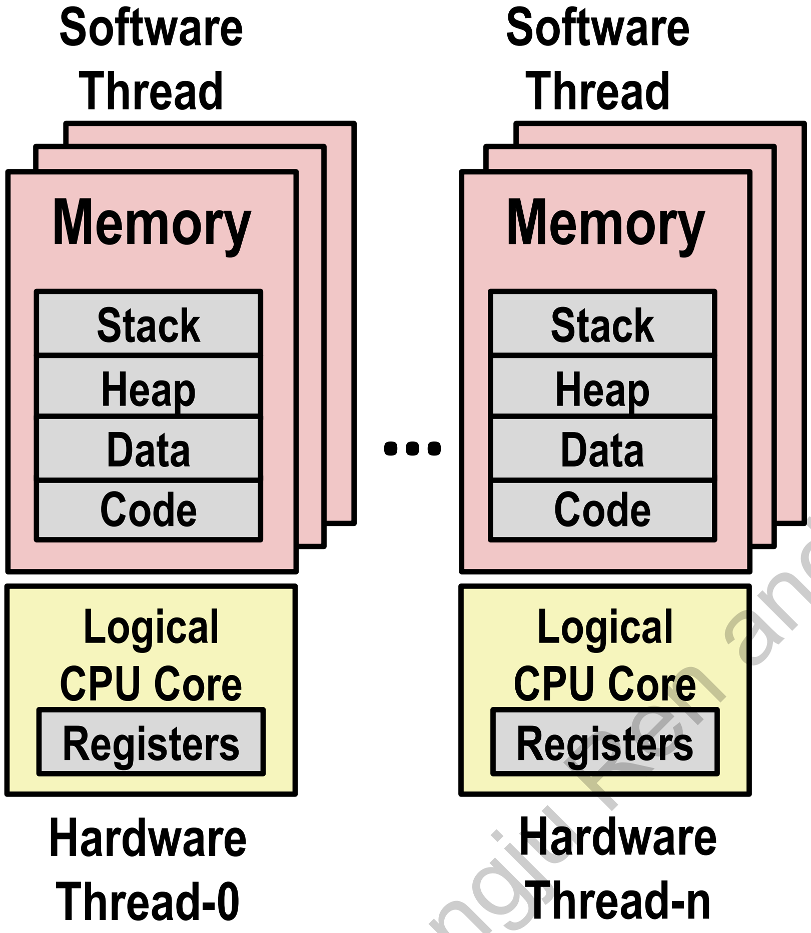
Threads (A Concept View)

- **Appearance of execution to Software (Programmer view point, *Software Thread*):**
  - Example: using ***1 or 100 threads*** to conduct the *sum of 1,000,000*
  - Each thread requires an **abstraction of hardware** to make multi-processing possible
  - The **smallest unit of processing** assigned and scheduled by operating system(OS)

- **Appearance of execution to Hardware (Architect view point, *Hardware Thread*) :**
  - Can be thought of as the **physical/logical CPU** or cores.
  - Example: iPhone **6 core/6 thread**; Laptop i7 CPU with **4 core/8 thread**; Lab Server Xeon CPU with **24 core/48 thread**
  - **One hardware thread** can run **many software threads** by **time-slicing** by the OS.



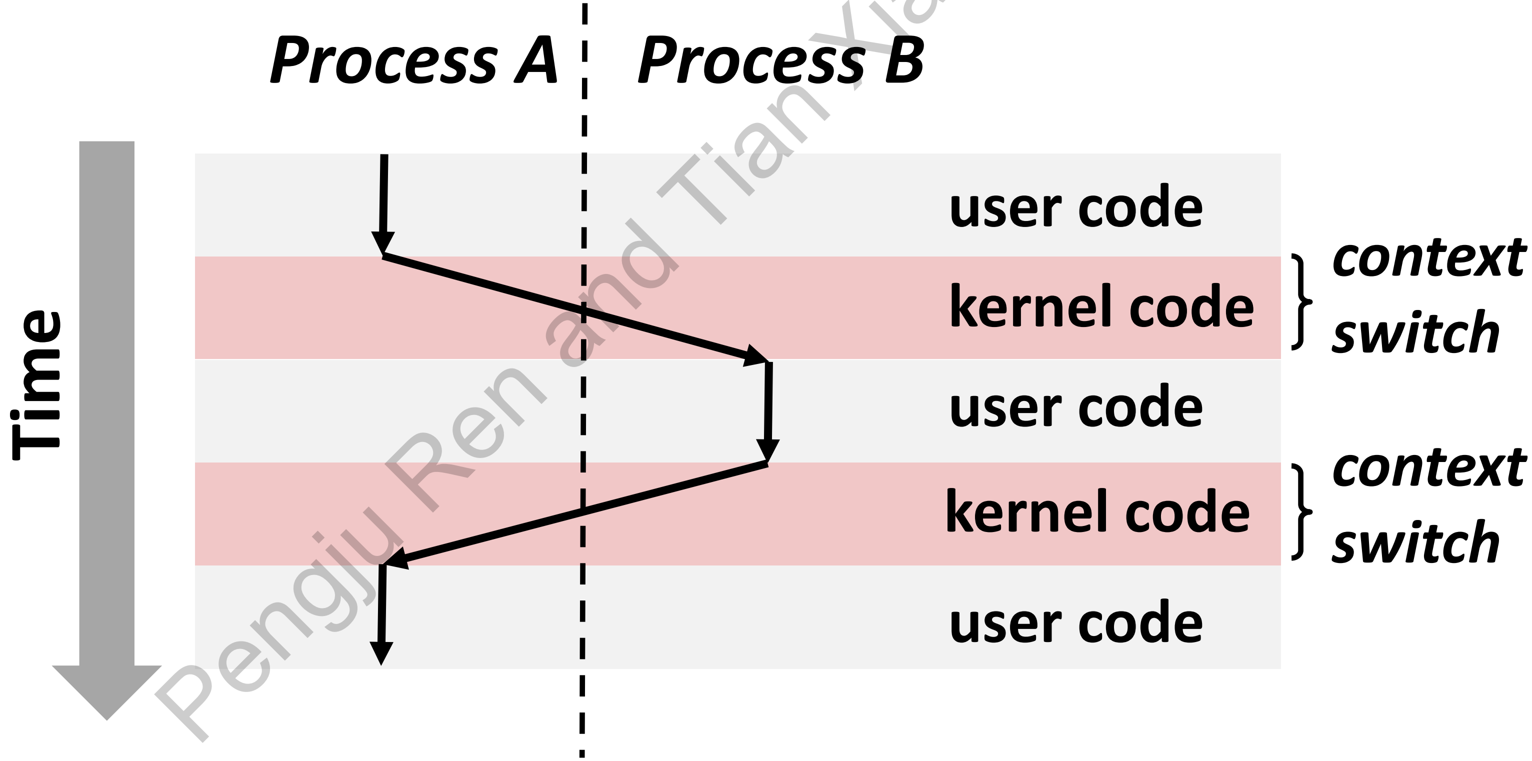
Process/Thread Execution



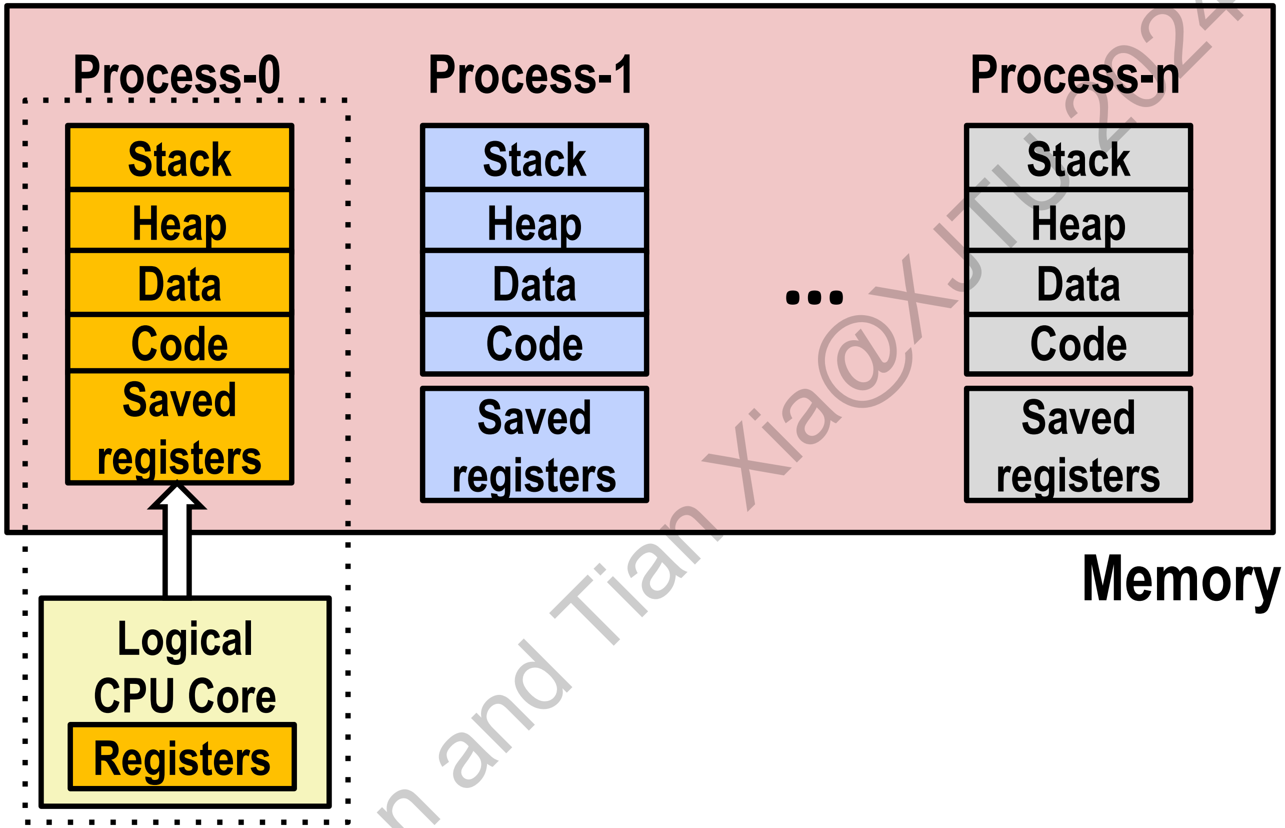
- **Ideally**, CPU runs threads **simultaneously**
  - □ Applications for many users
    - ➢ Web browsers, email clients, editors…
  - □ Background tasks
    - ➢ Monitoring network & I/O devices
- **Actually**, CPU runs threads with **limited hardware cores**
  - □ Process executions interleaved (**multi-tasking**)
  - □ Address spaces managed by **virtual memory** system
  - □ Save register values (**context**) for non-executing processes

Context Switching

- Control flow passes from one process to another via a ***context switch,*** conducted by ***Operating System (OS)***

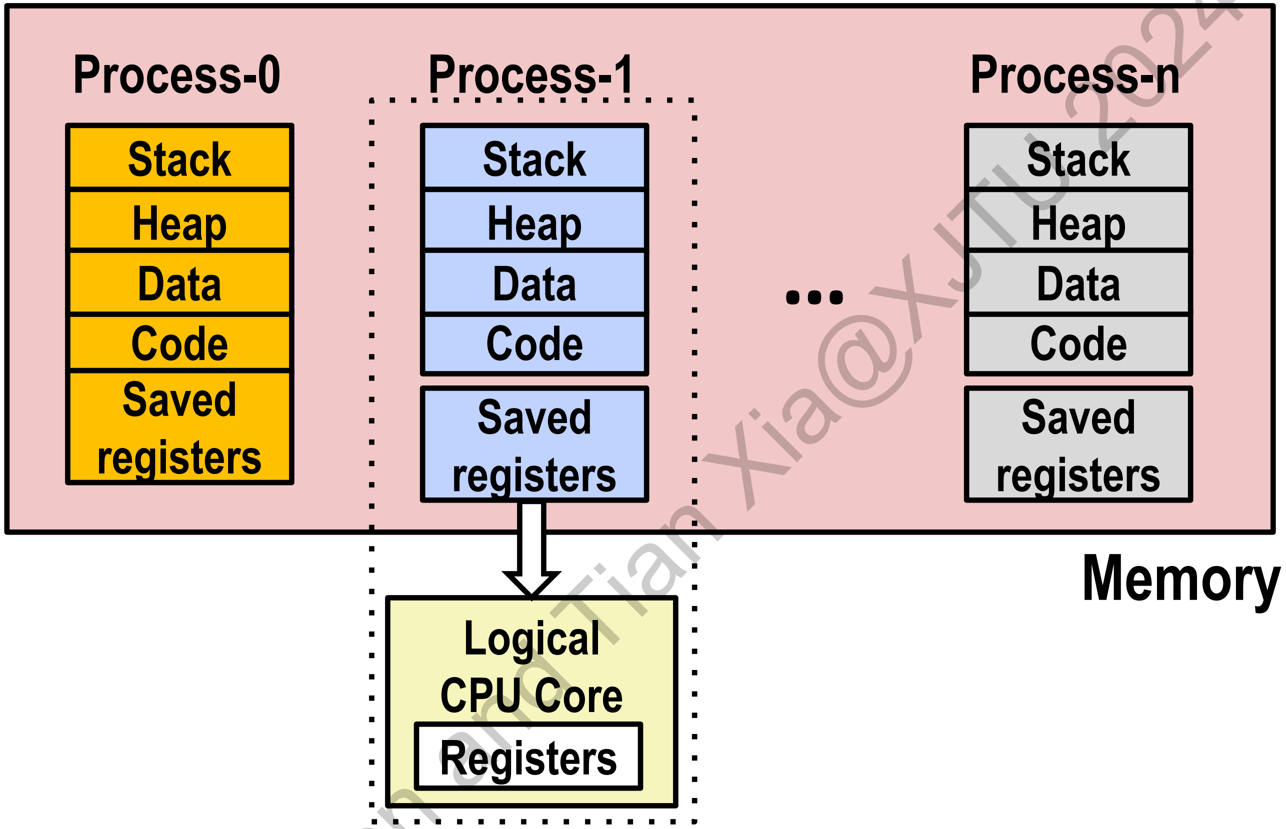


Process/Thread Context Switch



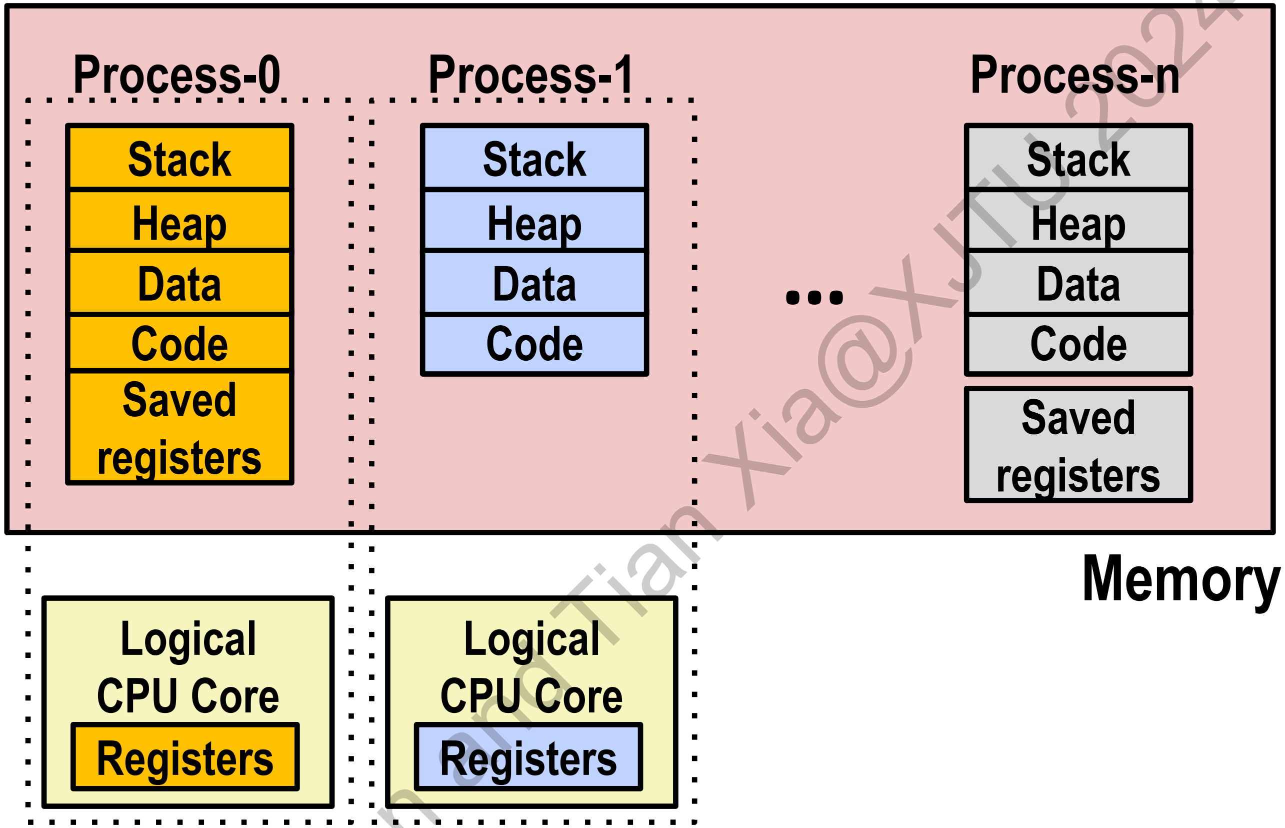
- OS saves current registers in memory

Process/Thread Context Switch



- OS saves current registers in memory
- OS schedules next thread for execution
- OS loads saved registers and switches address space (***context switch***)

Process/Thread on Multicore Processor



- **Multicore processors**
  - **Multiple CPU cores** on single chip
  - Each can execute a **separate process/thread**
    - **OS Schedules** process/thread onto CPU cores
  - **Share main memory** (and some **lower level caches**)

Recap: Terminology

- **Program**: An executable task
  – Example: Calculating the *sum of 1,000,000* number, could partition a single problem into multiple related tasks (threads) through ***parallel programming***
- **Process**: An instance of a running program or portion of program
  – Example a running instance of *sum of 1,000,000*
- **Software Thread** (Programmer View-point): it is an abstraction to the hardware to make multi-processing possible, the *smallest unit of processing* assigned and scheduled by operating system(OS)
  – Example: using *100 threads* to conduct the *sum of 1,000,000*
- **Hardware Thread** (Architecture View-point) : Can be thought of as the physical/logical CPU or cores. hardware thread can run many software threads by time-slicing by the OS.
  – Example: Your Laptop i7 CPU with 4 core/8 thread; Lab Server Xeon CPU with 24core/48 thread

Thread models : two very different implementations ***POSIX Threads*** and ***OpenMP***.

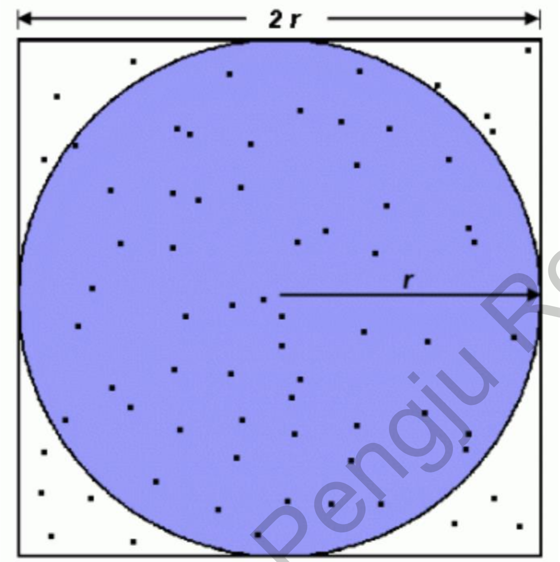
Thread-Level Parallelism（TLP）

- Many workloads can make use of **thread-level parallelism (TLP)**
  - TLP from multiprogramming ***(run one job faster using parallel threads)***
  - TLP from multithreaded applications ***(run several independent sequential jobs)***
- **Multi-threading:**
  - uses TLP to improve utilization of **a single processor**
- **Multi-/Many-core:**
  - **Duplicated Processors**, it plays a major role from the low end to the high end
- **Modern CPU do both**
  - Multiple or tens of cores with multiples threads per core

Thread-parallel programming

Software View-point*:*

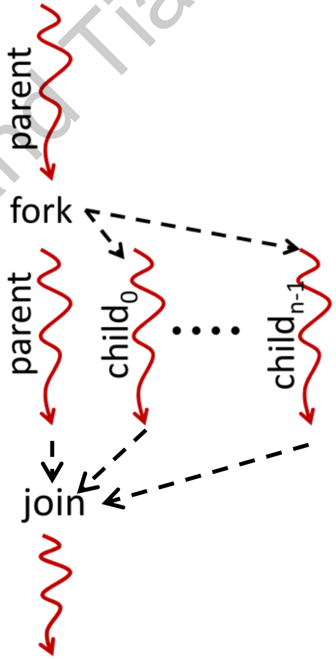
- All threads based on the **same program** that starts as a **single thread** process
- Software threads share the same **Virtual Address Space** but with private **PC**, **Reg File** and **Stacks**
- Different threads run concurrently on **different cores** or **interleaved on one core**



$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$



How about **uneven workload** distribution?

Idle threads

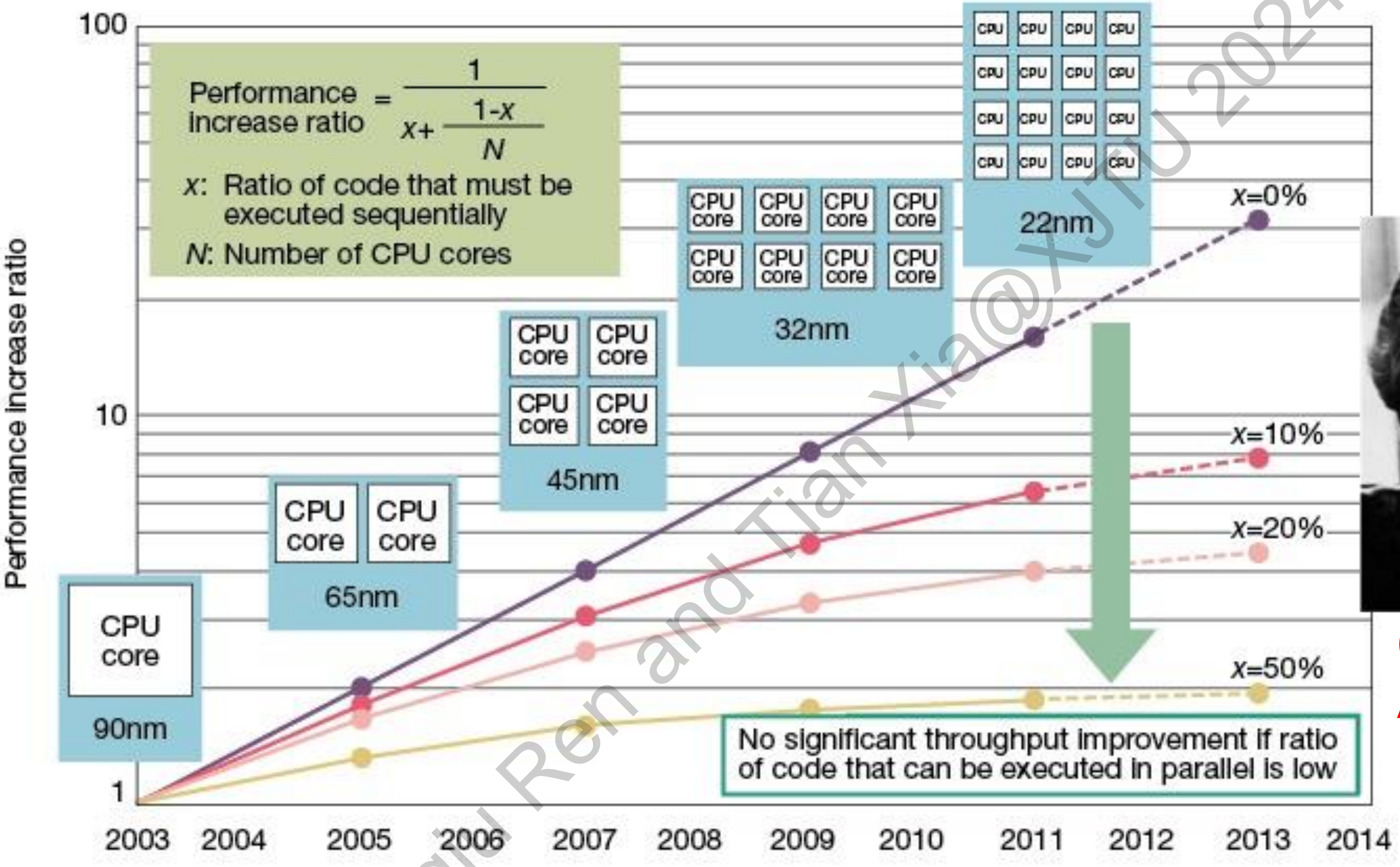
What's the difference using **single Sum** for all children or **Psum/Child**?

Threads stall for data(cache) coherence

Can this problem be solved **N times faster** ?

No

Challenges of Parallel Processing



No significant throughput improvement if ratio of code that can be executed in parallel is low

Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

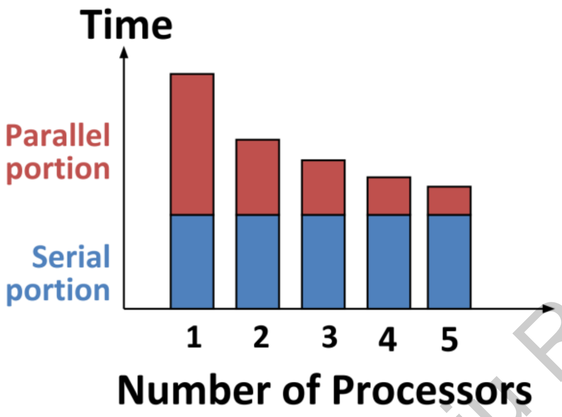
Caveat! Amdahl's Law

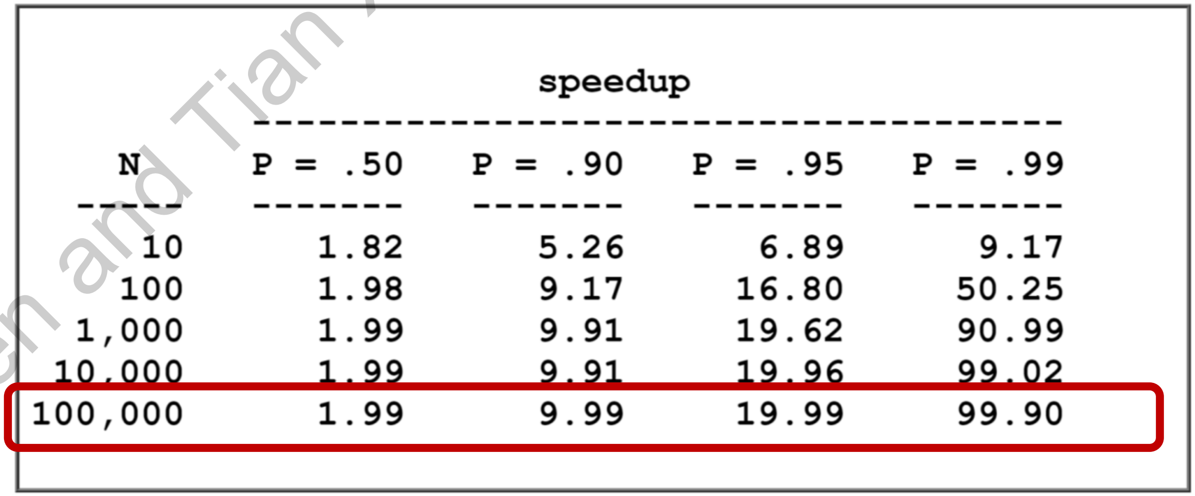
Amdahl’s Law

Amdahl's Law states that potential program speedup using N cores is determined by the proportion of code (P) that can be parallelized

$$\boldsymbol{Speedup = \frac{1}{P/N + S}}$$

It soon becomes obvious that there are limits to the scalability of parallelism:





	speedup			
N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

Insufficient parallelism and **long-latency remote communication** are the two biggest performance challenges in using multiprocessors.

Scalability

Strong scaling:

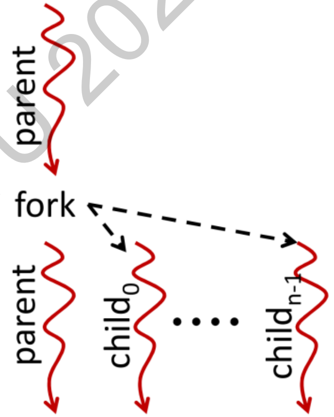
- The total **problem size stays fixed** as more processors are added.
- Goal is to run the same problem size **faster**
- Perfect scaling means problem is solved in **1/N** time (compared to serial execution)

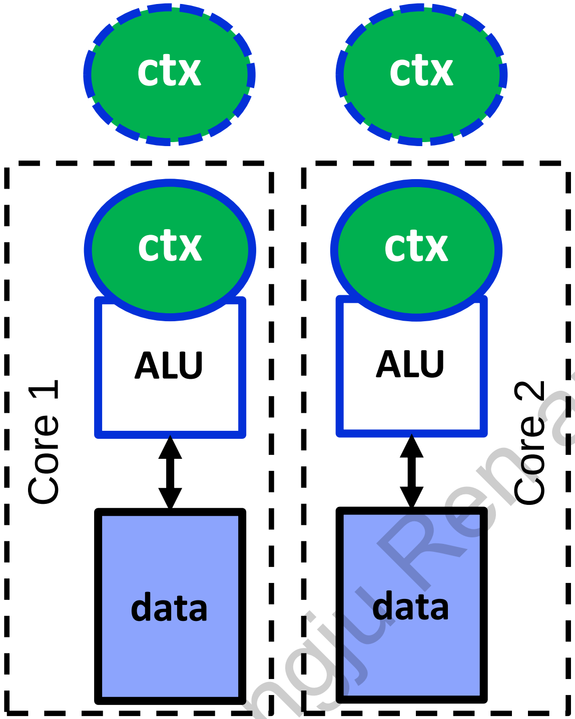
Weak scaling:

- The **problem size per processor stays fixed** as more processors are added.
- The **total problem size** is proportional to the number of processors used.
- Goal is to run larger problem in **same amount of time**
- Perfect scaling means problem ×**N** runs in same time as single processor run

Software Thread vs. Hardware Thread

Parent and Child acting as *software-threads*

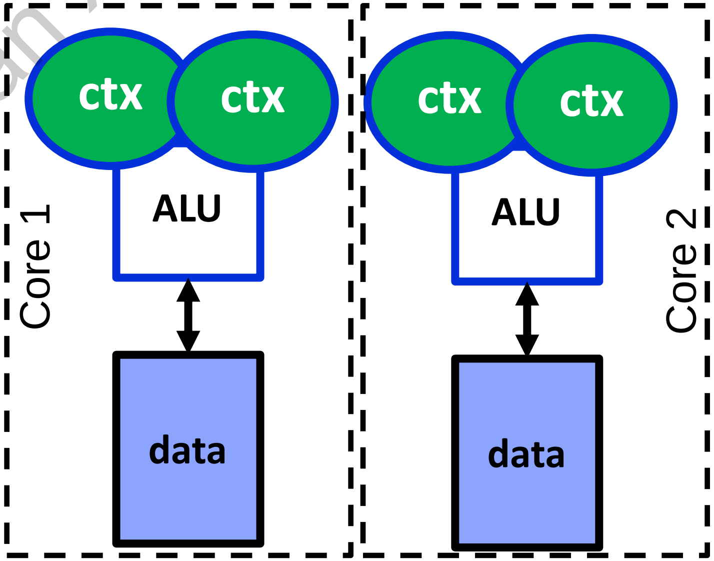




CPU: 2 core, 1 thread/core

hardware-threads: 2

(# Physical Core = 2)



CPU (SMT): 2 core, 2 thread/core

hardware-threads: 4

(# Physical Core = 2, # Logical Core = 4)

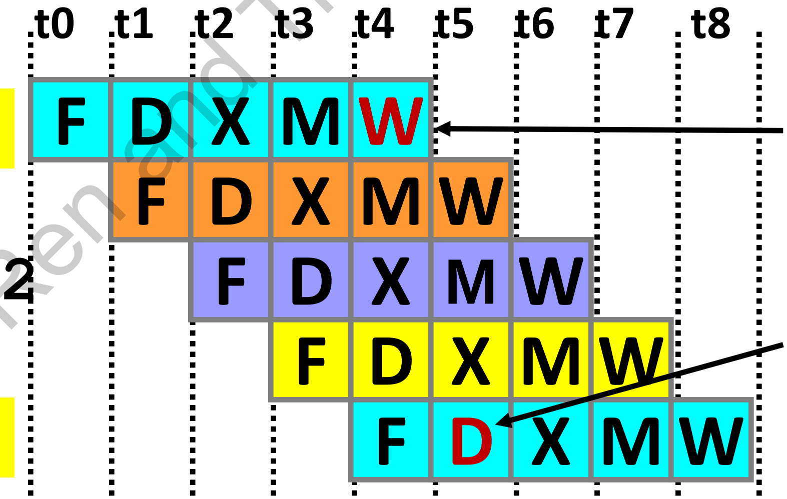
Motivation for Hardware Multithreading

How to **guarantee no dependencies** between instructions in a pipeline?

One way is to interleave execution of instructions from **different program threads** on **same pipeline**

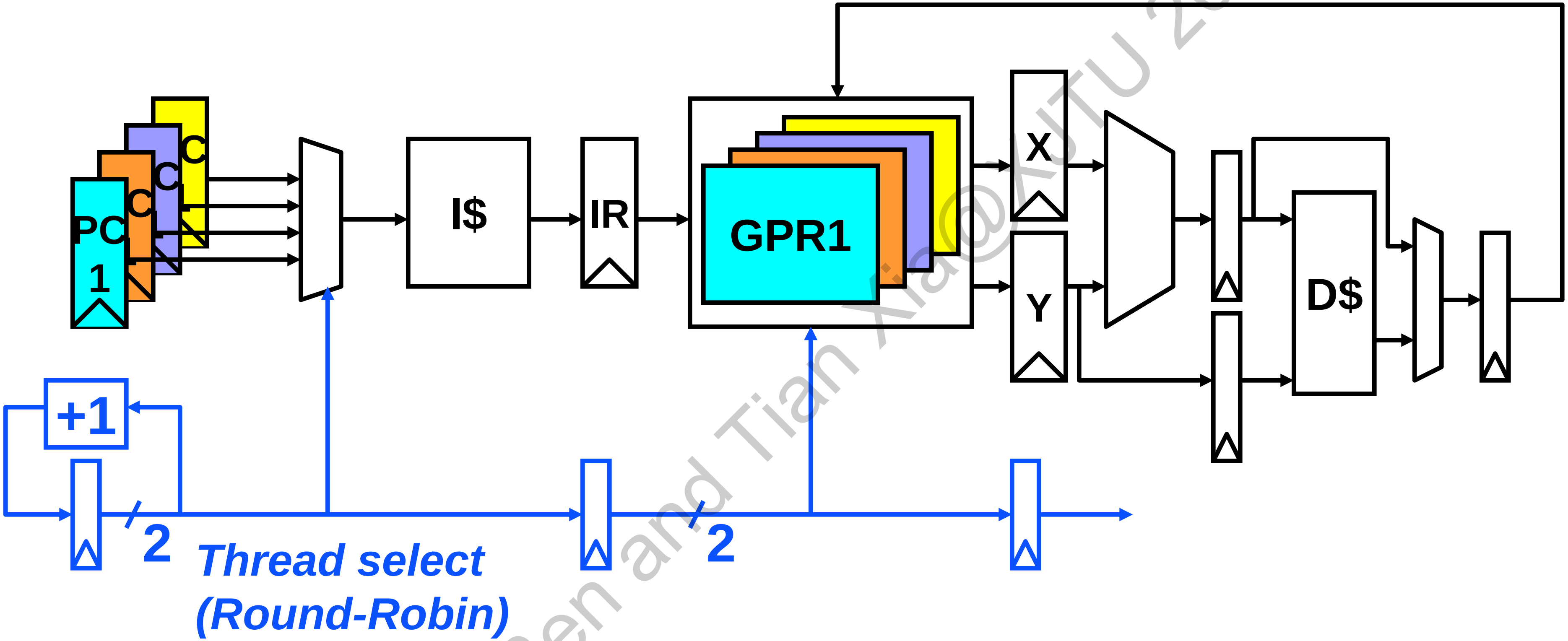
Interleave 4 threads, T1-T4, on ***non-bypassed*** *5-stage pipe*

```
T1:LD x1,0(x2)
T2:ADD x7,x1,x4
T3:XORI x5,x4,12
T4:SD 0(x7),x5
T1:LD x5,12(x1)
```



Prior instruction in a thread always completes write-back before next instruction in same thread reads register file

Simple Multithreaded Pipeline



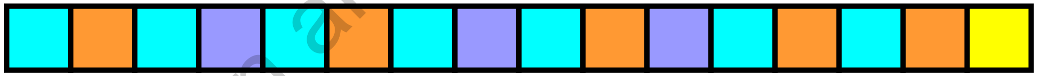
- Have to carry **thread select signal** down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as **multiple (albeit slower) CPUs**

Multithreading Costs for Modern Machine

- Each thread requires its **own user execution state**
  - Program Counter (PC)
  - General Purpose Registers
  - Renaming Table, etc.
- Also, needs its **own system state**
  - Virtual-memory page-table-base register (**PTBR**)
  - Exception-handling registers (e.g. **Exception Entry Register**)
- Other overheads:
  - Additional **cache/TLB conflicts** from competing threads
  - (or add **larger cache/TLB** capacity)
  - More **OS overhead** to schedule more threads (where do all these threads come from?)

Thread Scheduling Policies

- Fixed interleave *(CDC 6600 PPUs, 1964)*
  - – Each of N threads executes **one instruction every N cycles**
  - – If thread not ready to go in its slot, insert **pipeline bubble**
  - – Can potentially **remove bypassing and interlocking** logic
- Software-controlled interleave (*TI ASC PPUs, 1971*)
  - – OS allocates S **pipeline slots** amongst N threads (S>>N)
  - – Hardware performs **fixed interleave** over S slots, executing whichever thread is in that slot



- Hardware-controlled thread scheduling (*HEP, 1982*)
  - – Hardware **keeps track** of **which threads are ready** to go
  - – Picks next thread to execute based on **hardware priority** scheme
  - – **Coarse-grained** multithreading

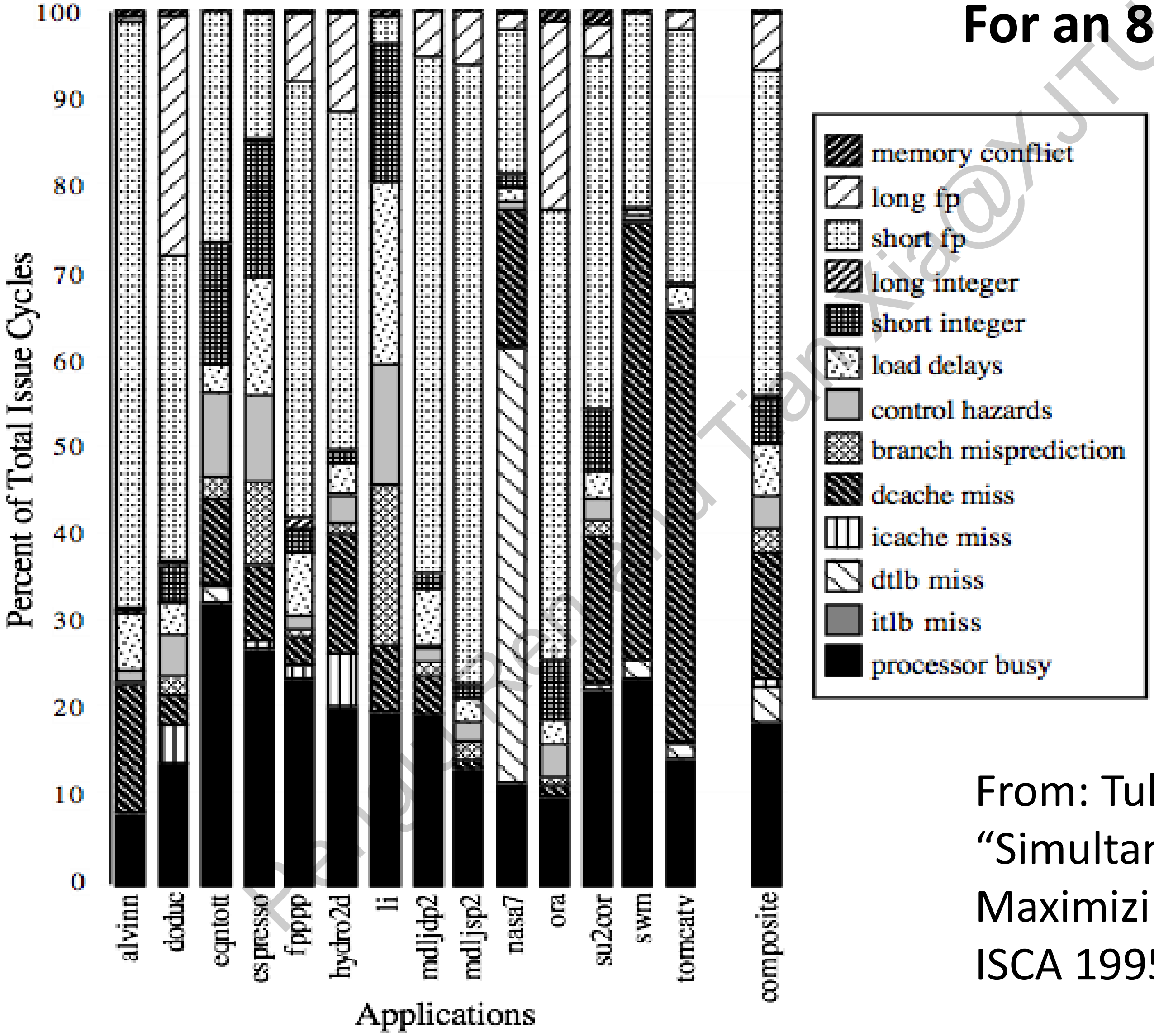
IBM PowerPC RS64-IV (2000)

- Commercial **Coarse-Grain Multithreading** CPU
- Based on PowerPC with **quad-issue in-order five-stage** pipeline
- Each physical CPU supports **two virtual CPUs**
- On every **L2 cache miss**, pipeline is **flushed** and physical CPU **execution switches** to the next thread
  - **short pipeline** minimizes flush penalty (4 cycles), small compared to memory access latency
  - flush pipeline to **simplify exception handling**

For most apps, most execution units lie idle in an Out-of-Order superscalar

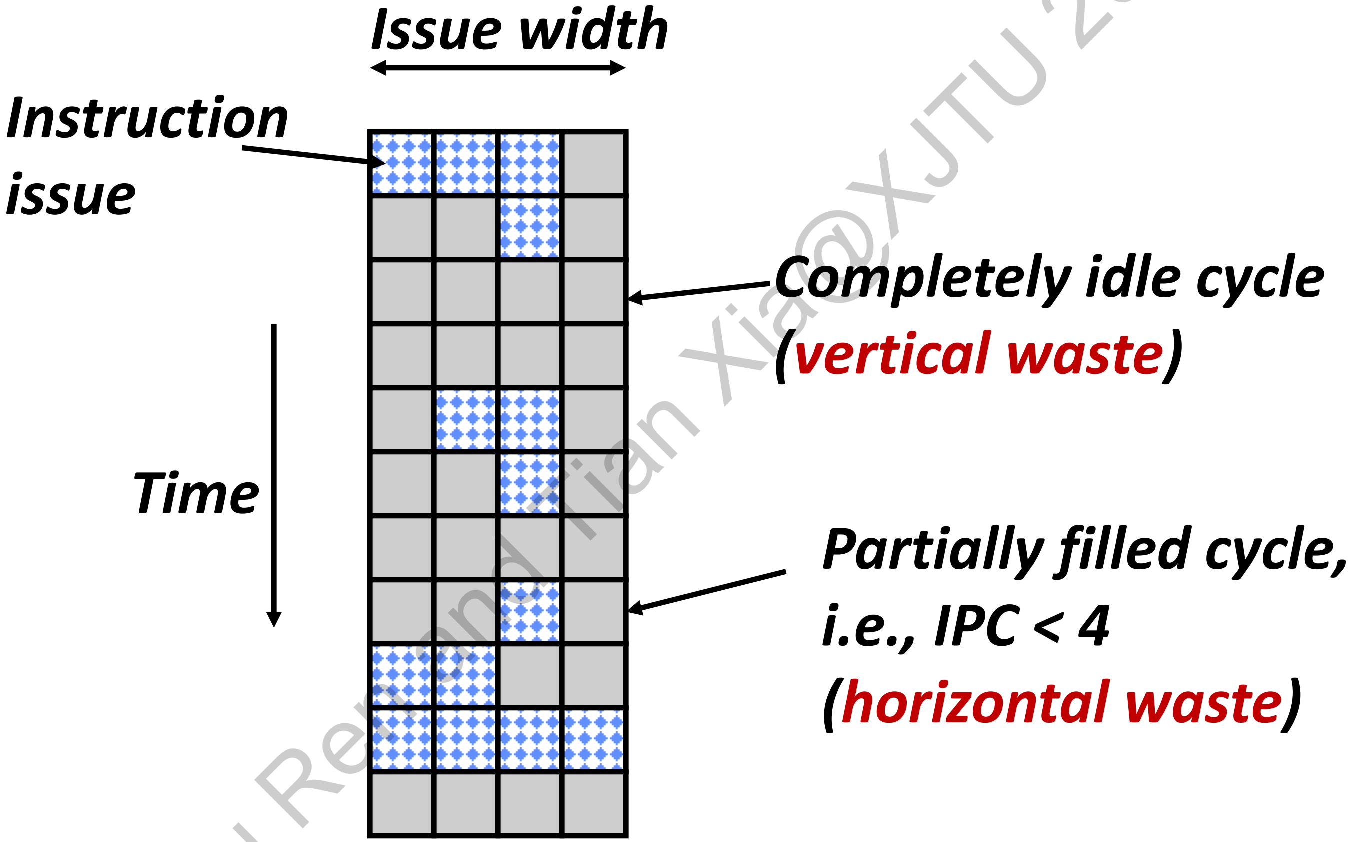
Theoretically, why is it faster?

For an 8-way superscalar.



From: Tullsen, Eggers, and Levy, “Simultaneous Multithreading: Maximizing On-chip Parallelism”, ISCA 1995.

SuperScalar Machine Efficiency



Issue width

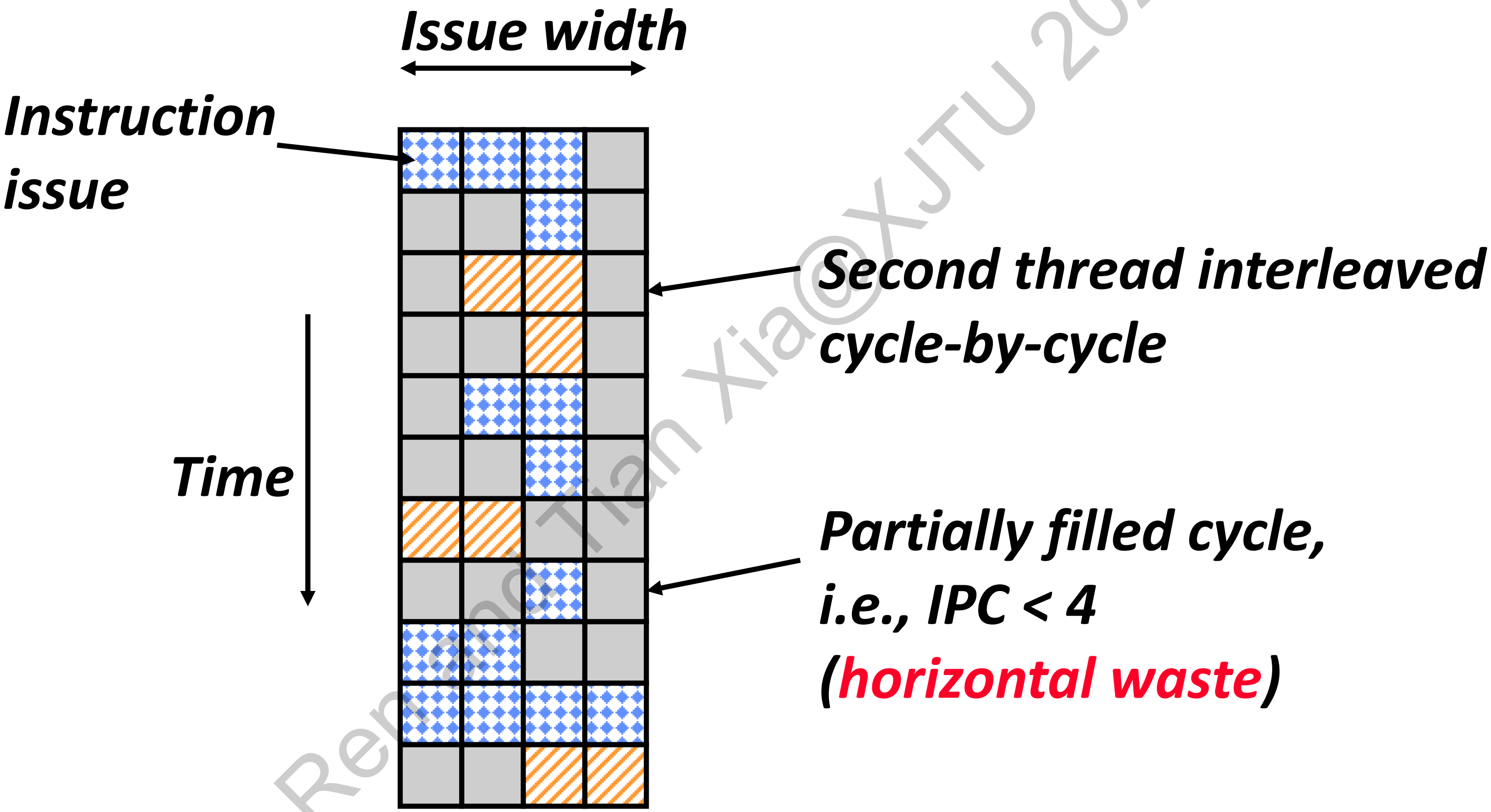
Instruction issue

Time

Completely idle cycle (vertical waste)

Partially filled cycle, i.e., IPC < 4 (horizontal waste)

Vertical Multithreading



Issue width

Instruction issue

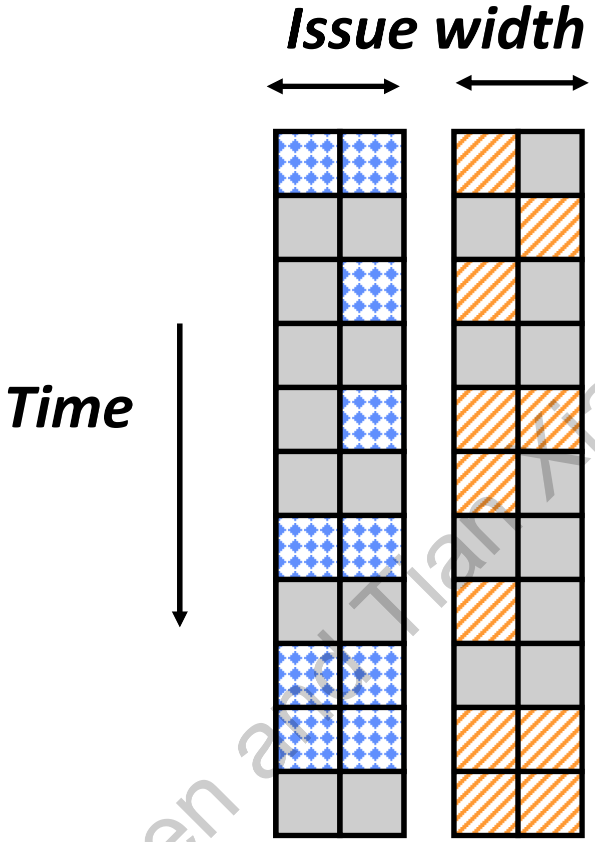
Time

Second thread interleaved cycle-by-cycle

Partially filled cycle, i.e., IPC < 4 (horizontal waste)

- Cycle-by-cycle (one or several clocks) interleaving removes vertical waste, but leaves some horizontal waste (fine-grained multithread)

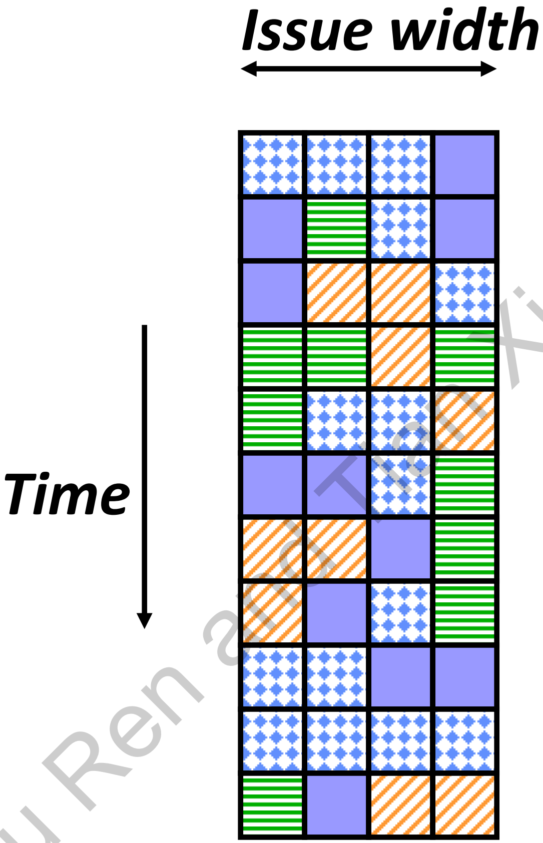
Chip Multiprocessing (CMP)



- What is the effect of splitting into multiple processors?
  - – reduces **horizontal** waste,
  - – leaves some **vertical** waste, and
  - – puts **upper limit on peak throughput** of each thread.

Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]

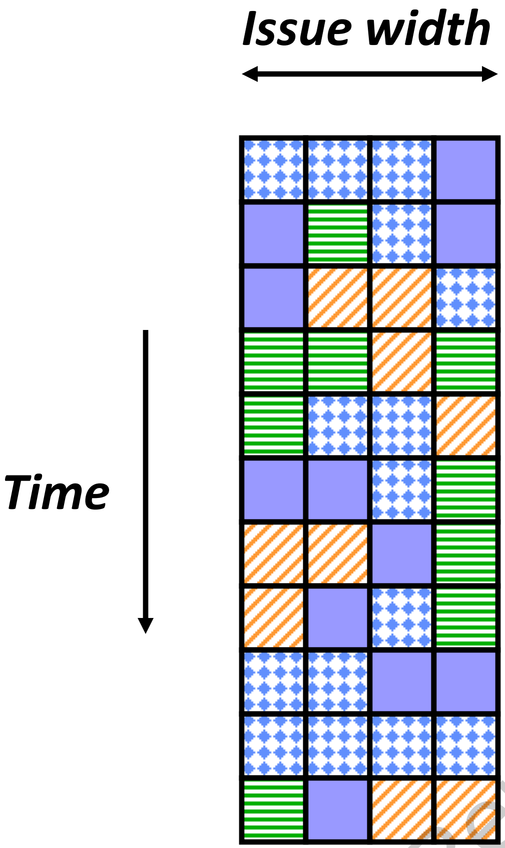


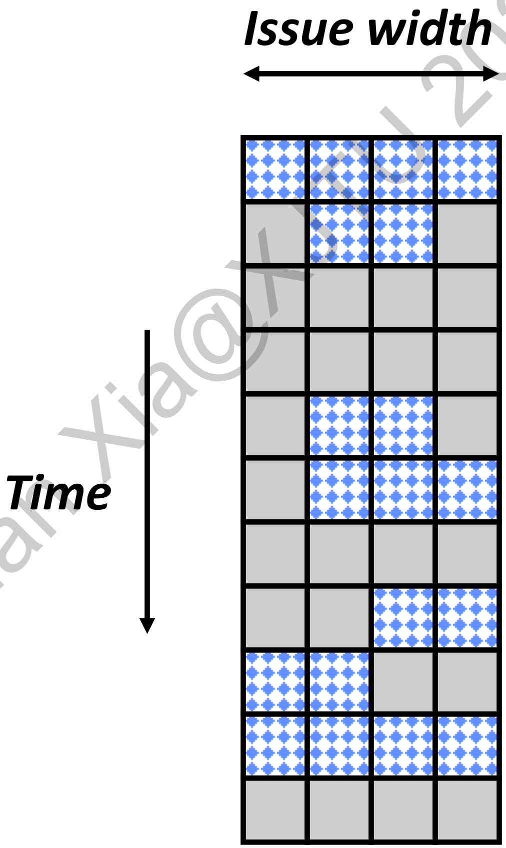
- Interleave multiple threads to multiple issue slots with **no restrictions**

Simultaneous Multithreading (SMT) for Out-of-Order Superscalars

- Add **multiple contexts** and fetch engines and allow instructions fetched from **different threads to issue simultaneously** (entering execution on same clock cycle). Gives better utilization of machine resources.
- Utilize **wide out-of-order superscalar issue queue** to find instructions to issue from multiple threads
- Out-of-Order instruction window **already has most of the circuitry required** to schedule from multiple threads
- Theoretically, any single thread can utilize **whole machine**
  - **Is this true in real implemented processors?**

SMT adaptation to parallelism type



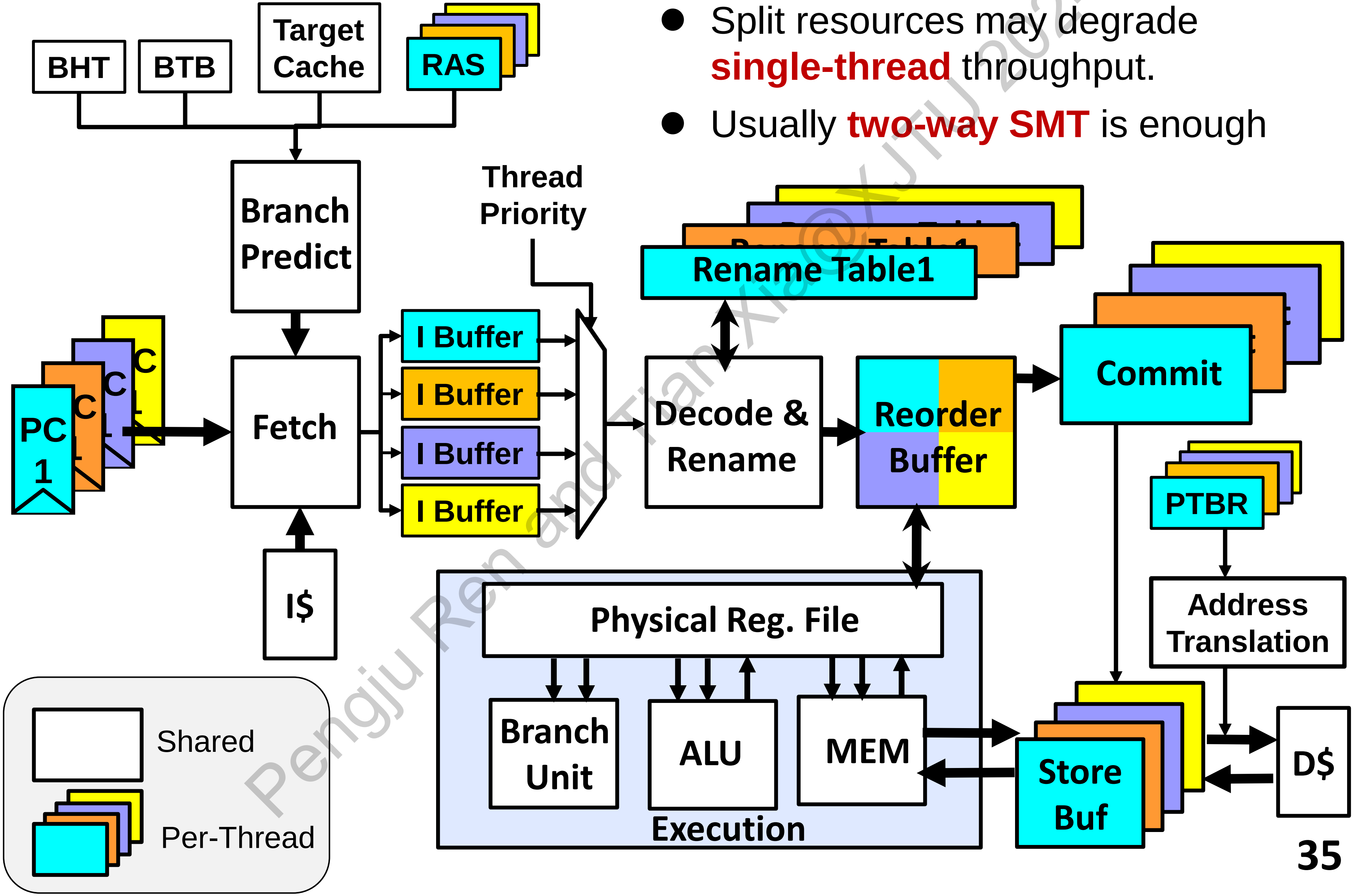


For regions with **high thread-level parallelism (TLP)**, entire machine width is shared by all threads

For regions with **low thread-level parallelism (TLP)**, entire machine width is available for instruction-level parallelism (ILP)

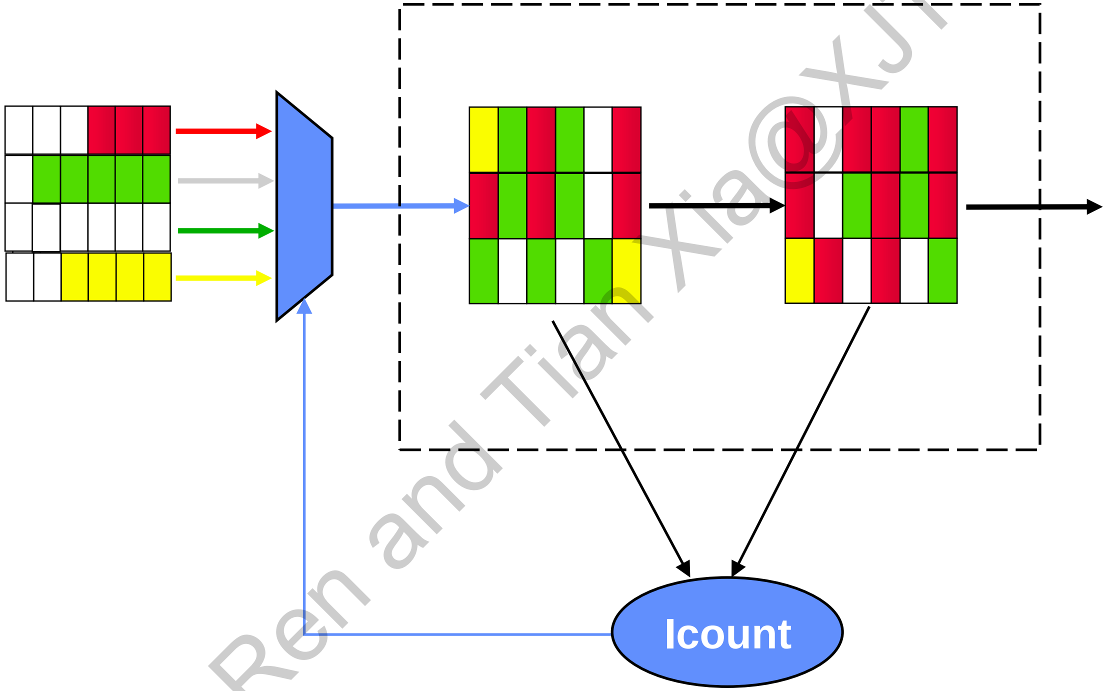
Multithreaded Design Discussion (Pipeline Stages)

- Split resources may degrade **single-thread** throughput.
- Usually **two-way SMT** is enough



Icount Choosing Policy

Fetch from thread with the **least instructions in flight**.



Why does this enhance throughput?

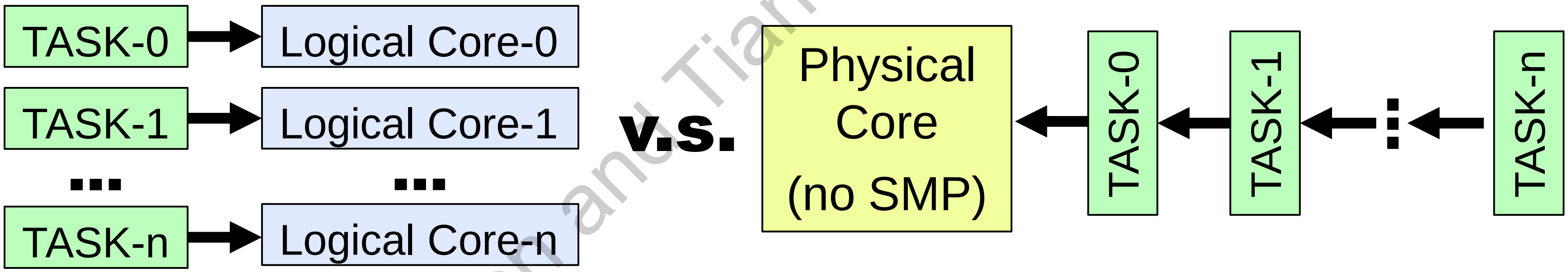
More instructions in flight ≈ more stalled conditions

Pentium-4 Hyperthreading (2002)

- Hyper-threading = SMT (in Intel world)
- First commercial SMT design (**2-way SMT**)
- **Logical processors** share nearly all resources of the physical processor
  - **Caches, execution units, branch predictors**
- **Chip (Die) area** overhead of hyper-threading ~ **+5%**
- When one **logical processor is stalled**, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running **only one active software thread** at almost **same speed** with or without hyper-threading
- Hyper-threading dropped on Out-of-Order P6 based follow on to Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), until revived with Nehalem generation machines in 2008.
- Intel Atom (in-order x86 core) has **2-way vertical** multithreading
  - Hyper-threading == (SMT for Intel Out-of-Order & Vertical for Intel In-Order)

Initial Performance of SMT

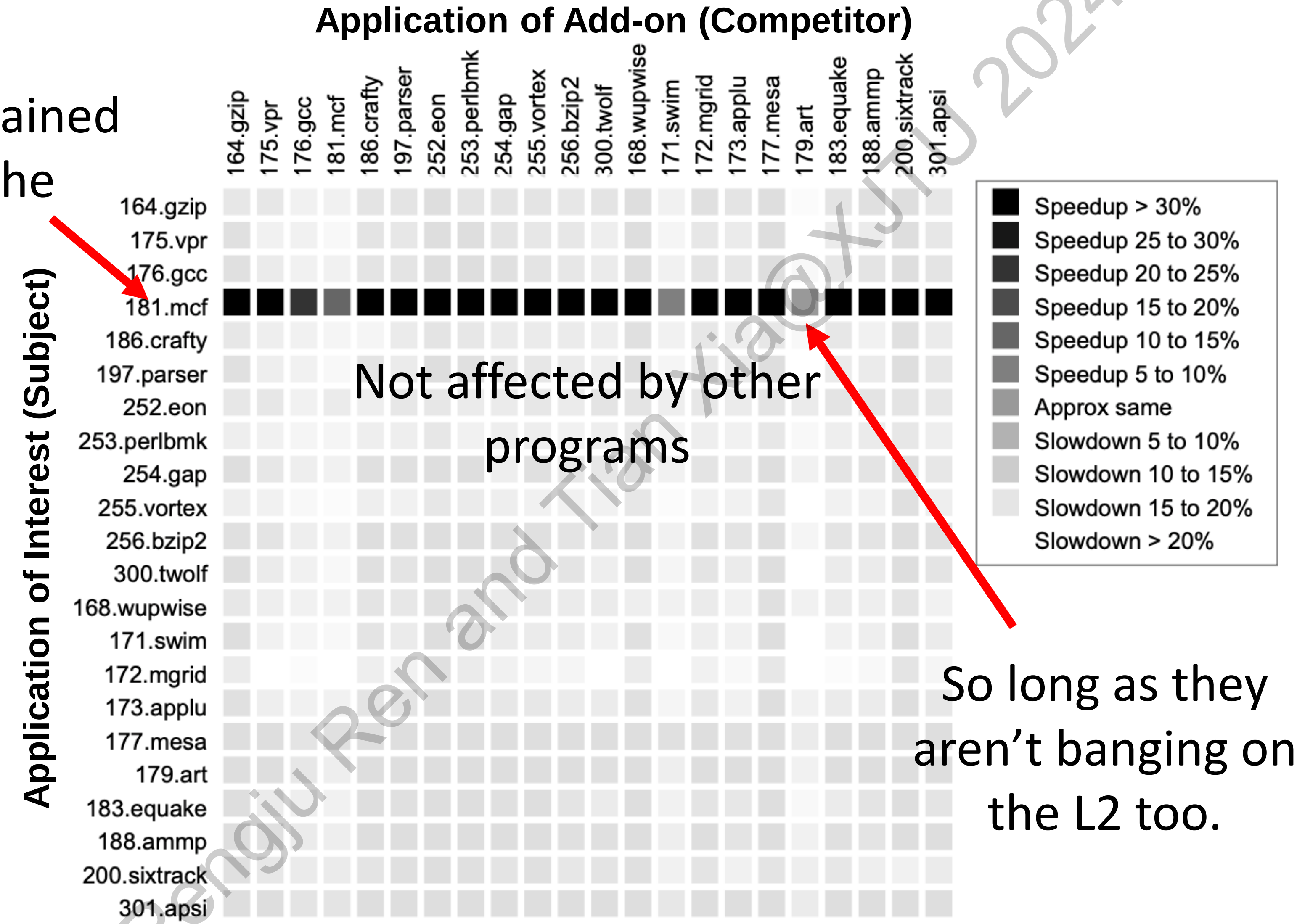
- Pentium-4 Extreme SMT yields **1.01x** speedup for *SPECint_rate* benchmark and **1.07x** for *SPECfp_rate* (with **5%** extra die size)
  - Pentium-4 is **dual-threaded SMT** (i.e. 2-way SMT)
  - *SPECRate* requires that each SPEC benchmark be run against a vendor-selected number of **copies of the same benchmark**
- Running on Pentium-4 each of 26 SPEC benchmarks **paired with every other** (26*26 runs) speedup 0.90--1.58 (**average 1.20x**)



- Power-5 processor gets **1.23** faster for *SPECint_rate* with SMT, **1.16** faster for *SPECfp_rate* (with **25%** extra die size)
- Power-5 running 2 copies of each app speedup **0.89--1.41**
  - Most gained some speedup
  - **Floating Point apps** had most cache conflicts and least gains

SMT Performance: Application Interaction

Mcf is Constrained by Cache



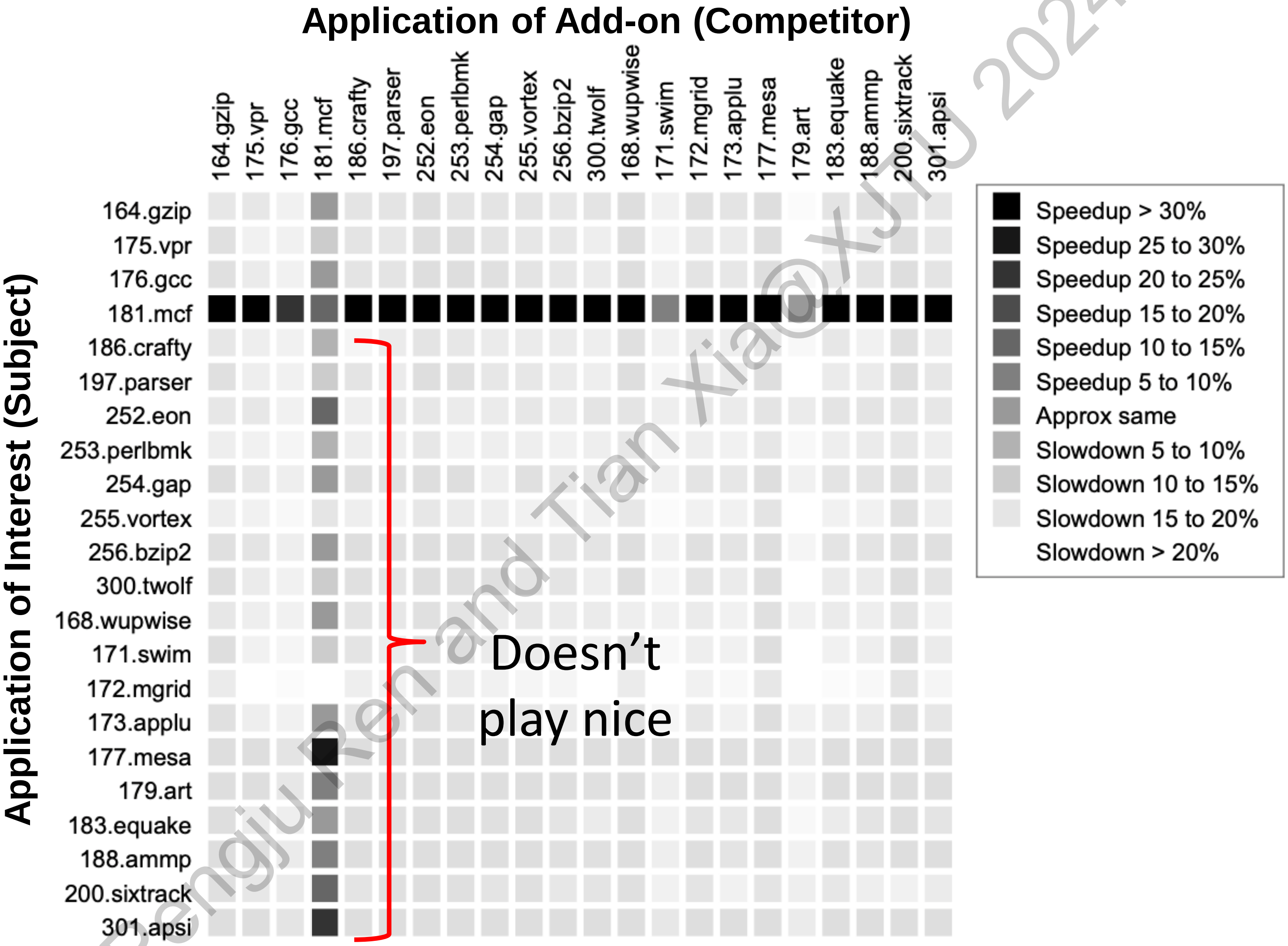
Not affected by other programs

So long as they aren't banging on the L2 too.

Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

https://www.spec.org/cpu2000/CINT2000/181.mcf/docs/181.mcf.html

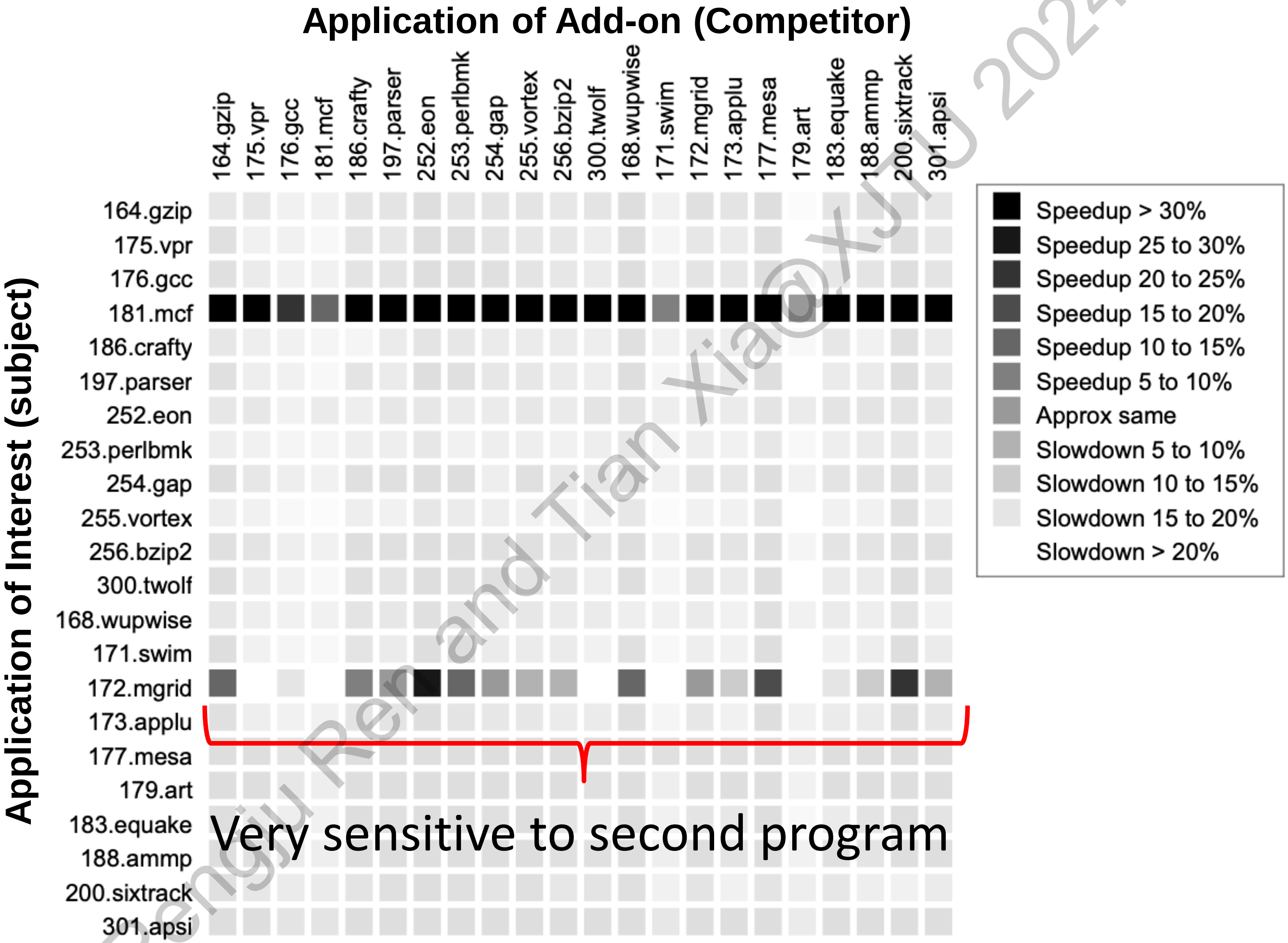
SMT Performance: Application Interaction



Doesn't play nice

Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

SMT Performance: Application Interaction



Very sensitive to second program

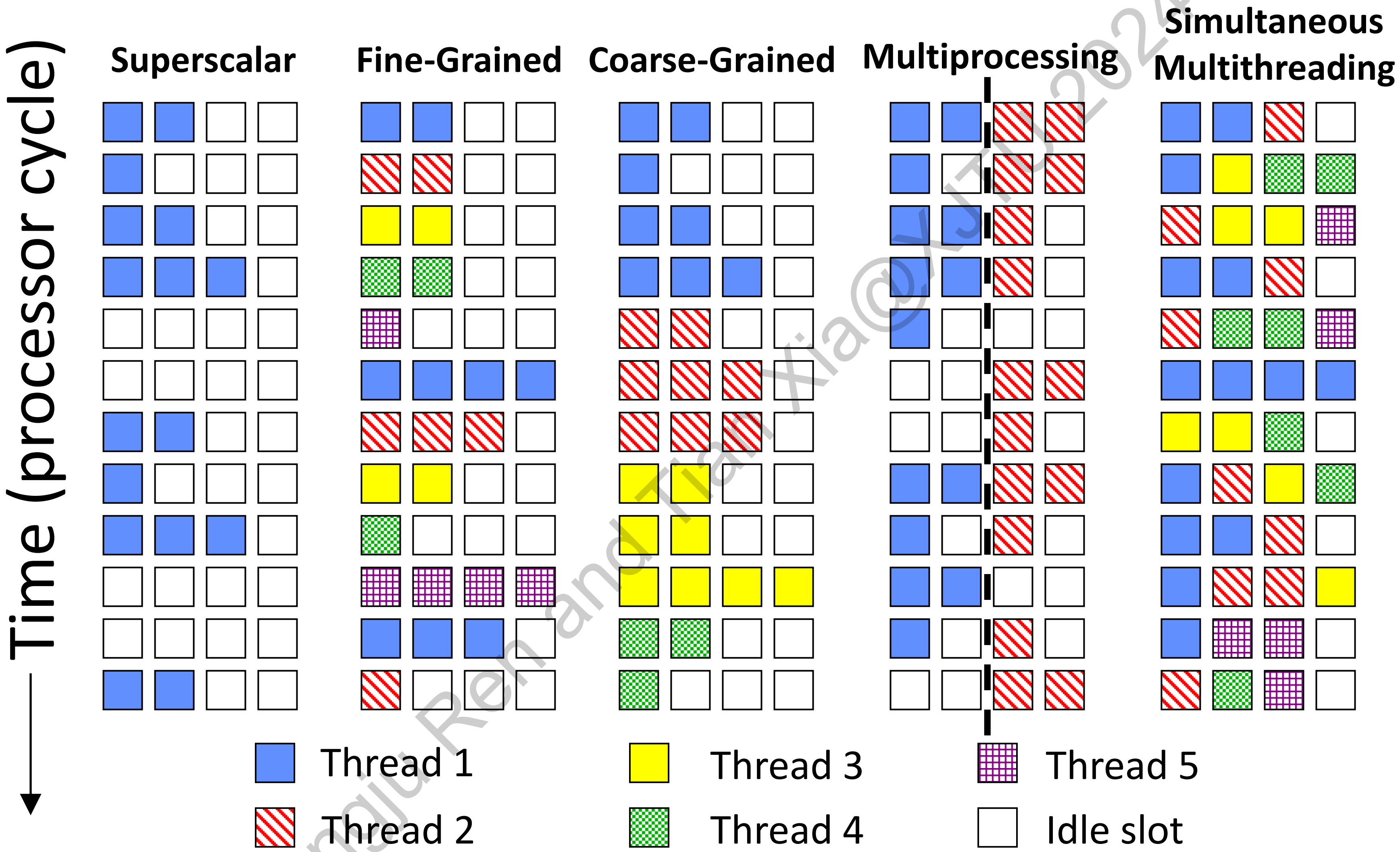
Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

SMT & Security

- Most hardware attacks rely on shared hardware resources to establish a **side-channel**
  - – E.g. Shared outer caches, DRAM row buffers
- SMT gives attackers high-bandwidth access to previously **private hardware resources** that are shared by co-resident threads:
- TLBs: TLBleed (June, ‘18)
- L1 caches: CacheBleed (2016)
- Functional unit ports: PortSmash (Nov, ’18)

OpenBSD 6.4 → **Disabled** HT in BIOS, AMD SMT to follow

Summary: Multithreaded Categories



Superscalar

Fine-Grained

Coarse-Grained

Multiprocessing

Simultaneous Multithreading

Time (processor cycle)

Thread 1

Thread 2

Thread 3

Thread 4

Thread 5

Idle slot

- **Moderate benefits** compared with extra die area (which is **still in debate**)
- Potential risks of **security** attacks.
- Successful in **commercial marketing**.

Next Lecture：Cache Coherence and Memory Consistency Model

(Thread-level Parallel)

Acknowledgements

- Some slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - David Patterson (UCB)
  - David Wentzlaff (Princeton University)

- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C