北京大学
PEKING UNIVERSITY

智能硬件体系结构

第九讲：缓存优化与多核多线程

主讲：陶耀宇、李萌

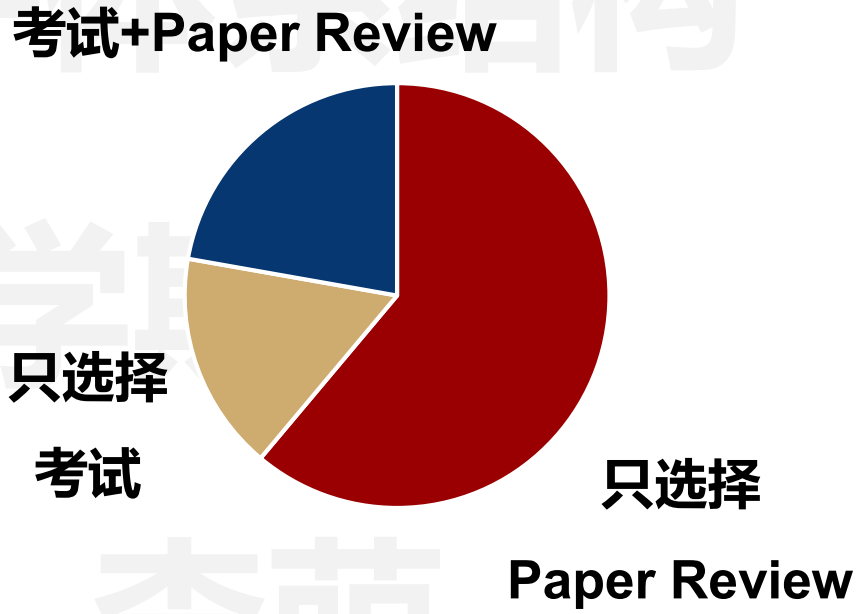
2024年秋季

注意事项

北京大学 PEKING UNIVERSITY

• 课程作业情况

- 第2次作业答案已放出
- 第3次作业**11月16日**截止（OoO等）
- 未来**3次作业（11.15-11.30、12.1-12.15、12.15-12.30）**
  - Cache设计、AI芯片架构、软硬加协同优化
- **Take Home考试**
  - **11月25中午12:00 – 11月26日凌晨11:59**
  - 每迟交一天扣10分
- **Paper Review**
  - 请自行阅读近5年内的体系结构相关论文1-2篇
  - 做7-8页左右的PPT
  - 推荐侧重在AI芯片架构



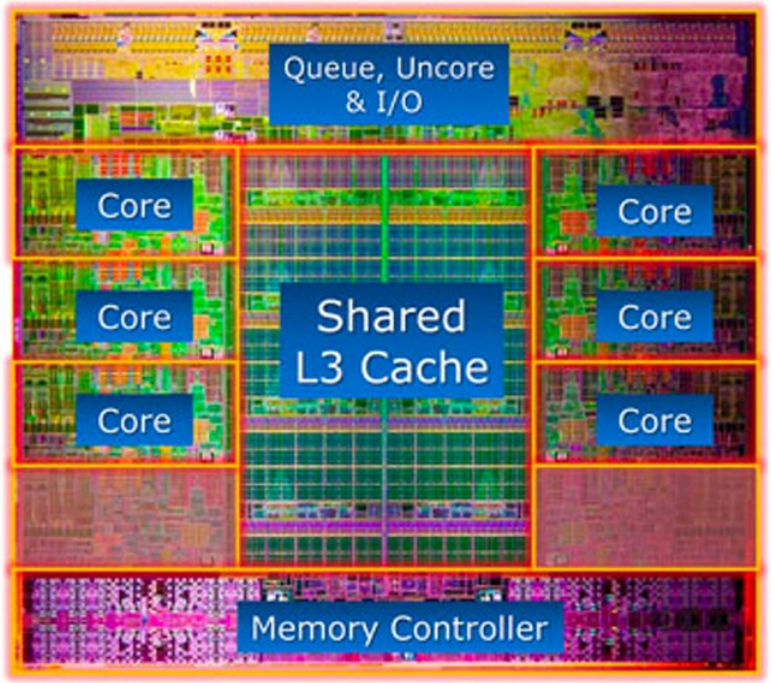
目 录

CONTENTS

缓存一致性的来源

- 多核共享cache

Intel® Core™ i7-3960X Processor Die Detail



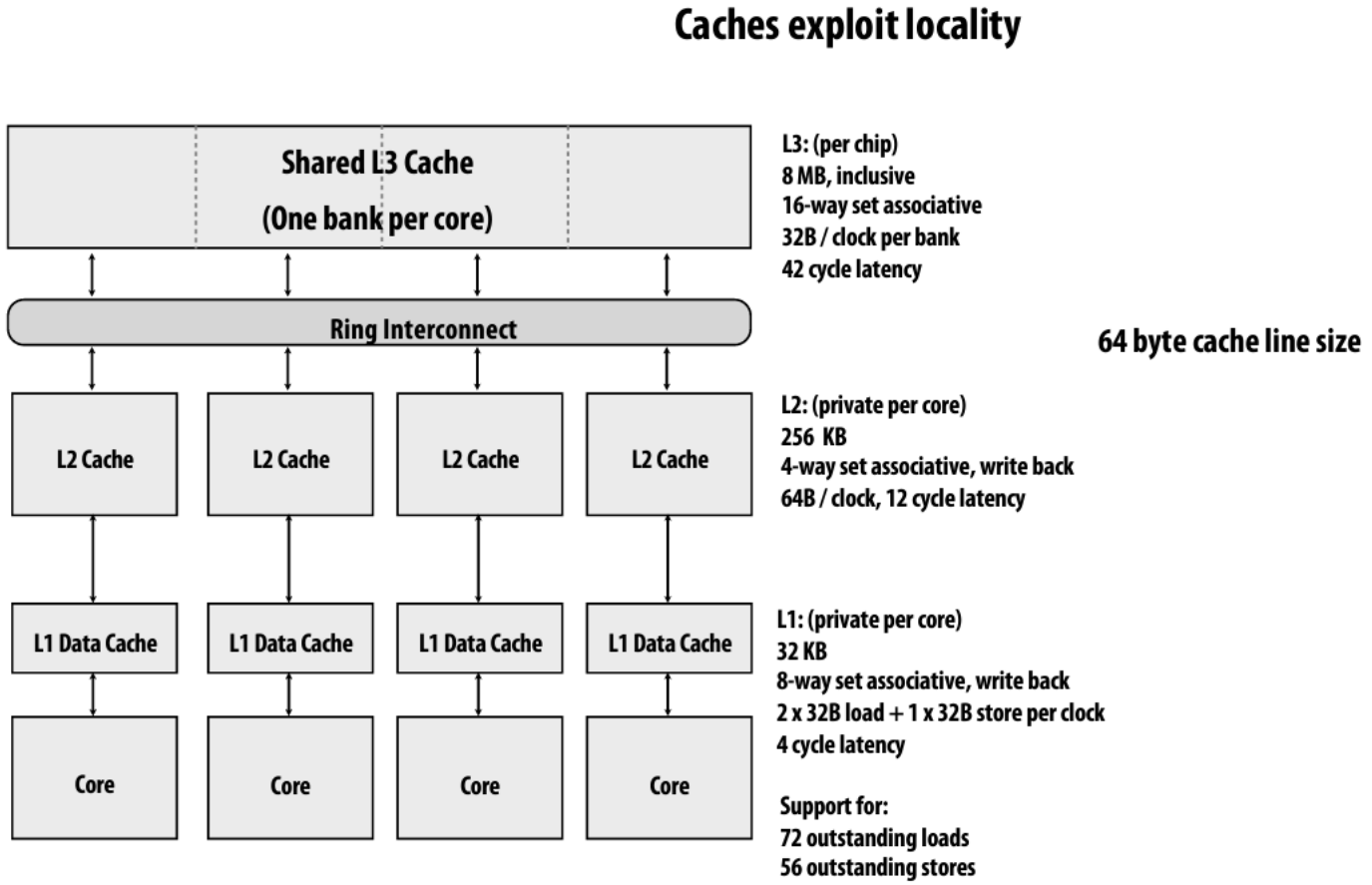
- 30% of the die area is cache

缓存一致性的来源

- 多核共享cache

3 Cs cache miss model

- Cold
- Capacity
- Conflict



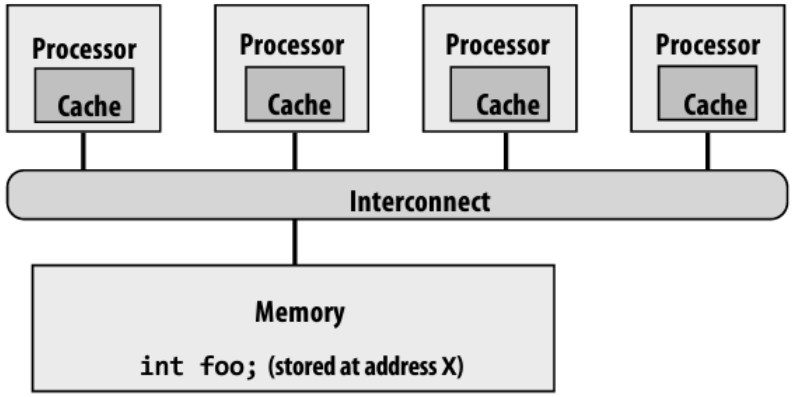
Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

缓存一致性的来源

- 多核共享cache

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



The chart at right shows the value of variable foo (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

缓存一致性的来源

北京大学 PEKING UNIVERSITY

- 缓存一致性协议

- **The logic we are about to describe is performed by each processor’s cache controller in response to:**
  - - Loads and stores by the local processor
  - - Messages from other caches on the bus
- **If all cache controllers operate according to this described protocol, then coherence will be maintained**
  - - The caches “cooperate” to ensure coherence is maintained

缓存一致性的来源

北京大学 PEKING UNIVERSITY

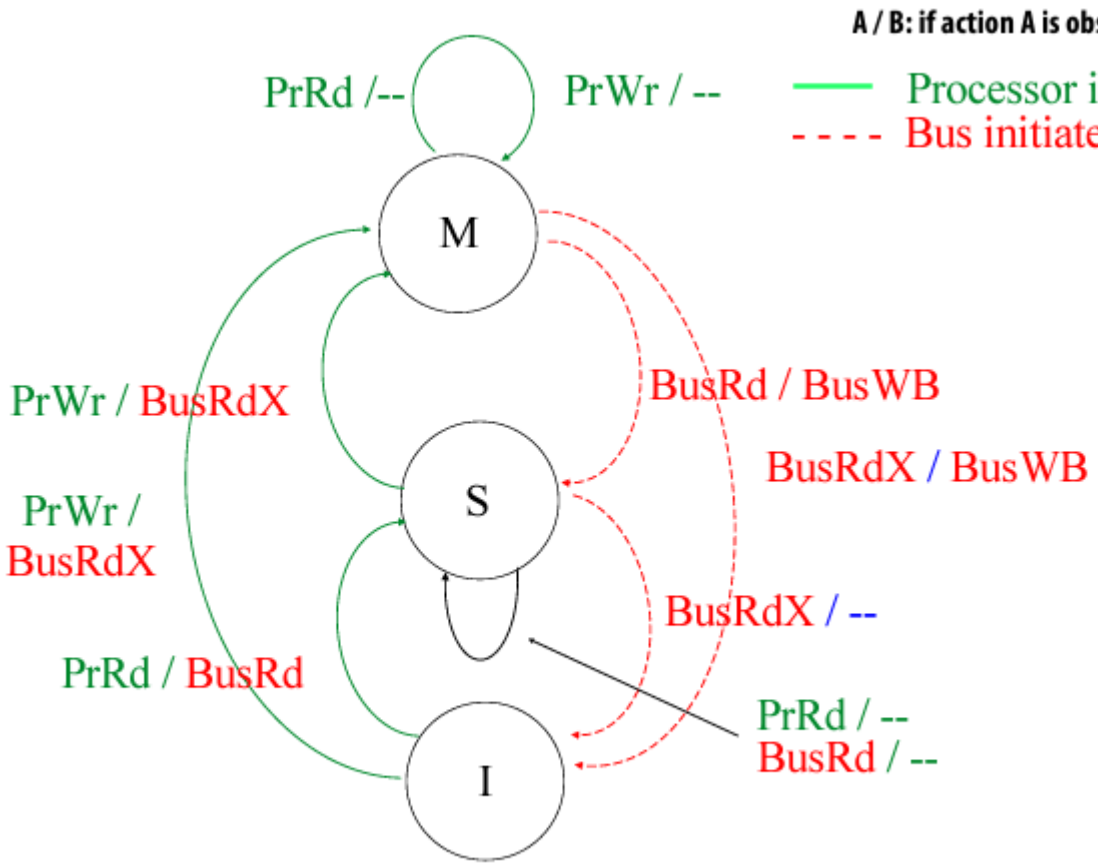
- **Invalidation-based write-back protocol**
  - **Key ideas:**
    - A line in the “modified” state can be modified without notifying the other caches
  - **Processor can only write to lines in the modified state**
    - Need a way to tell other caches that processor wants exclusive access to the line
    - We accomplish this by sending all the other caches messages
  - **When cache controller sees a request for modified access to a line it contains**
    - It must invalidate the line in its cache

缓存一致性

- **MSI write-back invalidation protocol**
  - **Key tasks of protocol**
    - Ensuring processor obtains exclusive access for a write
    - Locating most recent copy of cache line’s data on cache miss
  - **Three cache line states**
    - Invalid (I): same as meaning of invalid in uniprocessor cache
    - Shared (S): line valid in one or more caches, memory is up to date
    - Modified (M): line valid in exactly one cache (a.k.a. “dirty” or “exclusive” state)
  - **Two processor operations (triggered by local CPU)**
    - PrRd(read)
    - PrWr(write)
  - **Three coherence-related bus transactions (from remote caches)**
    - BusRd: obtain copy of line with no intent to modify
    - BusRdX: obtain copy of line with intent to modify
    - BusWB: write dirty line out to memory

缓存一致性

- Cache Coherence Protocol: MSI State Diagram



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

缓存一致性

- Cache Coherence Protocol: MSI State Diagram

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read x	S	--	--	BusRd	Memory
2. P3 read x	S	--	S	BusRd	Memory
3. P3 write x	I	--	M	BusRdX	Memory
4. P1 read x	S	--	S	BusRd	P3's cache
5. P2 read x	S	S	S	BusRd	Memory
6. P2 write x	I	M	I	BusRdX	Memory

缓存一致性

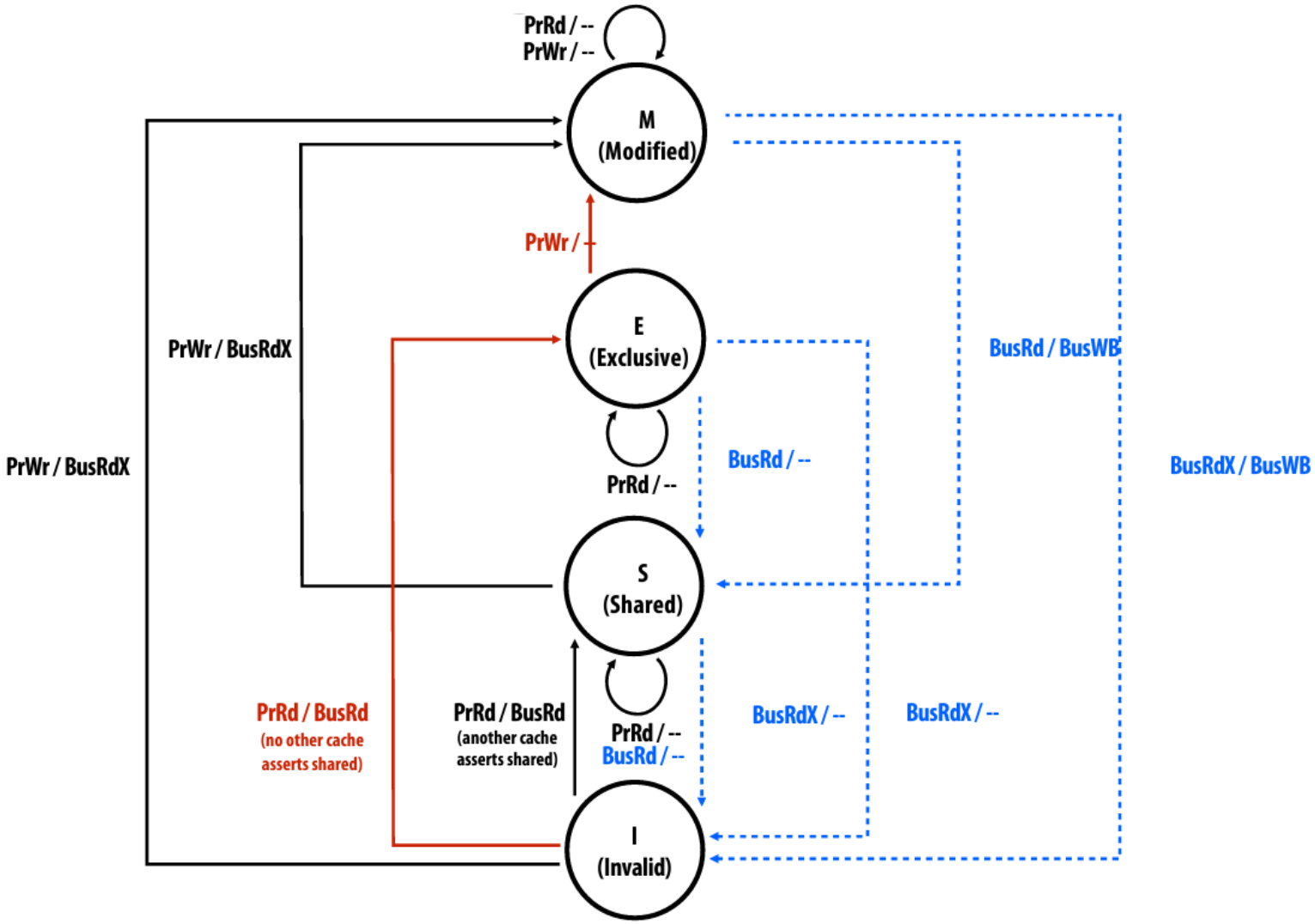
北京大学 PEKING UNIVERSITY

- Cache Coherence Protocol: MESI invalidation protocol

  - **MSI requires two interconnect transactions for the common case of reading an address, then writing to it**
    - Transaction 1: BusRd to move from I to S state
    - Transaction 2: BusRdX to move from S to M state
  - **This inefficiency exists even if application has no sharing at all**
  - **Solution: add additional state E (“exclusive clean”)**
    - Line has not been modified, but only this cache has a copy of the line
    - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
    - Upgrade from E to M does not require an bustransaction

缓存一致性

- Cache Coherence Protocol: MESI invalidation protocol

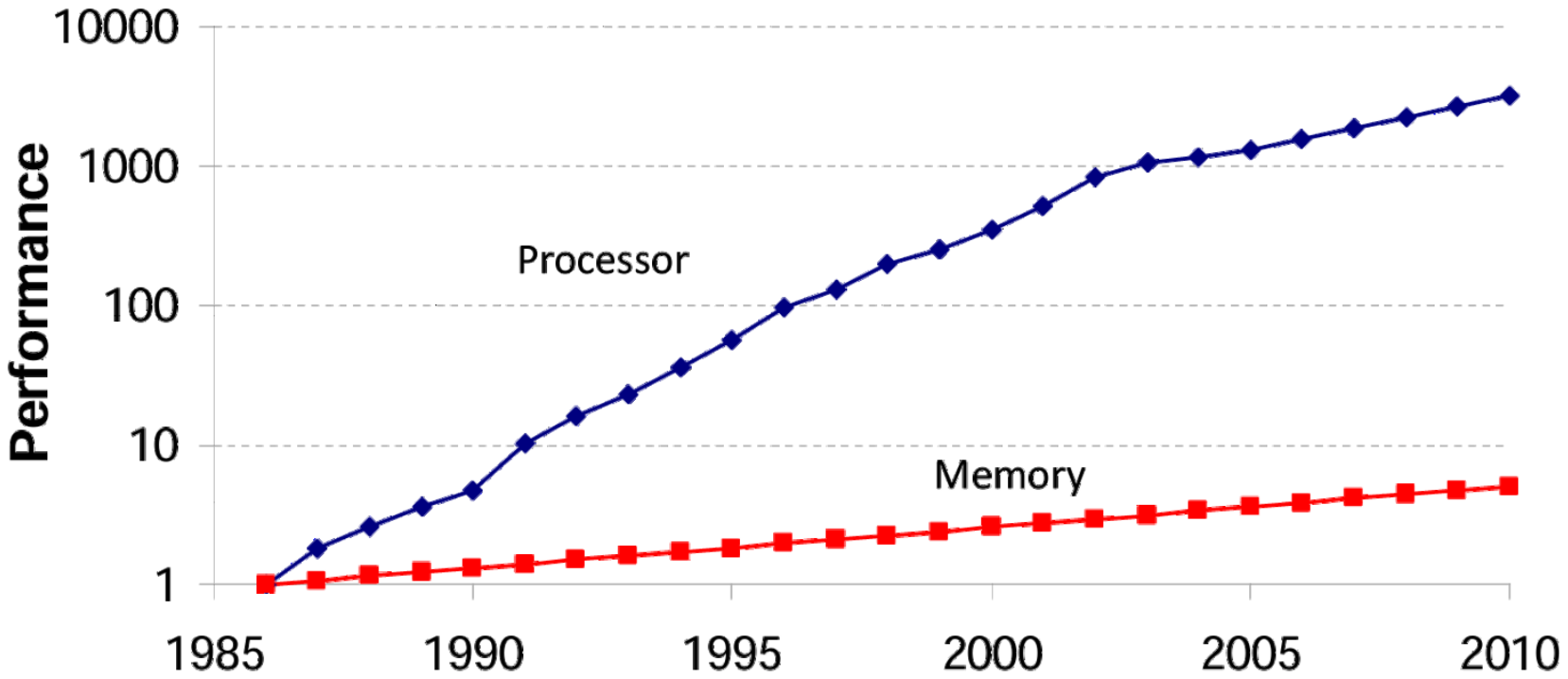


目 录

CONTENTS

缓存预读取优化机制

- Memory Wall



Source: Hennessy & Patterson, *Computer Architecture: A Quantitative Approach, 4th ed.*

Today: 1 mem access ≈ 500 arithmetic ops

How to reduce memory stalls for existing SW?

缓存预读取优化机制

北京大学 PEKING UNIVERSITY

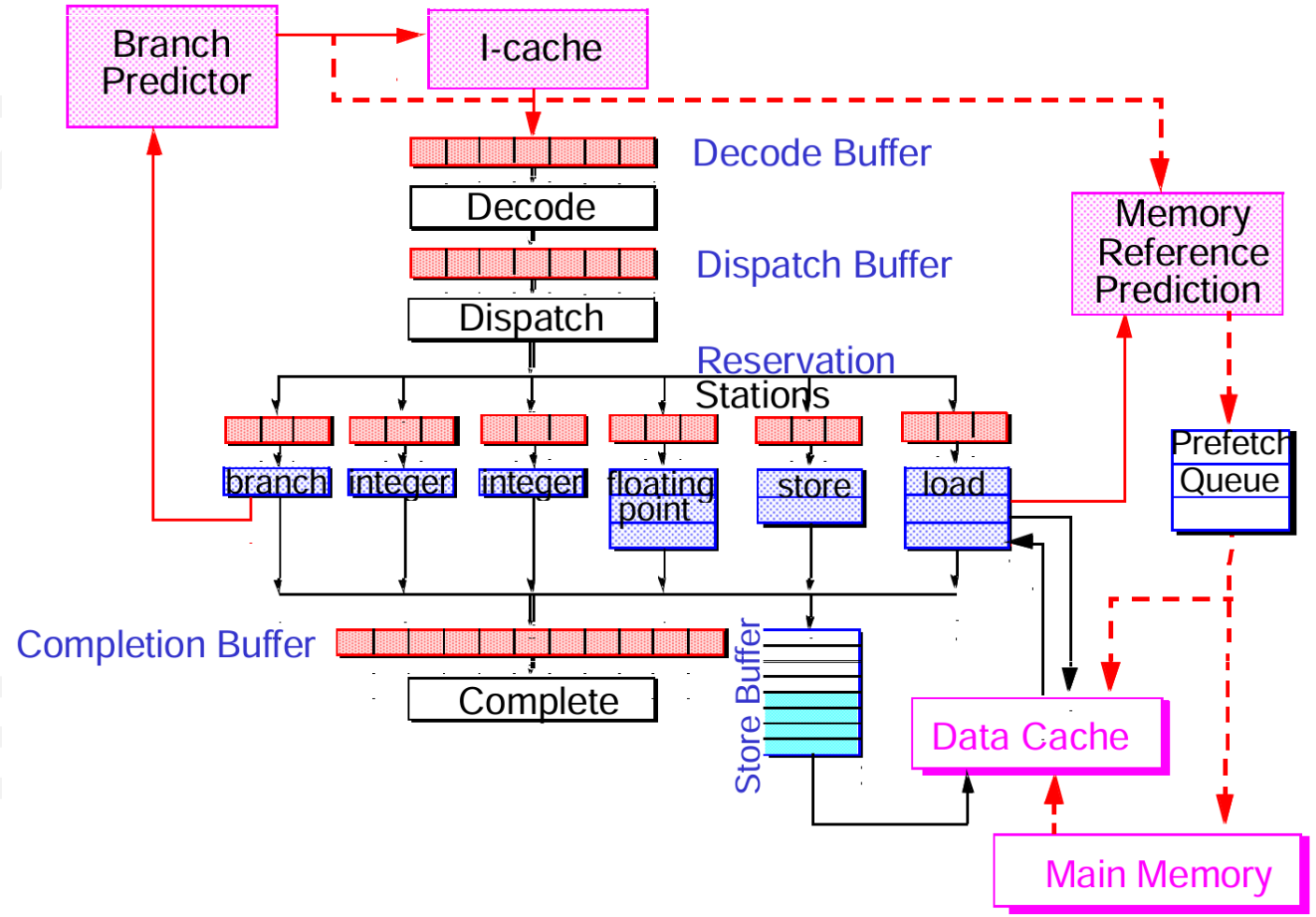
- 什么是预读取（Prefetch）

- Fetch memory ahead of time
- Targetscompulsory, capacity, & coherence misses

- Big challenges:
  - **knowing “what” to fetch**
    - Fetching useless info wastes valuable resources
  - **“when” to fetch it**
    - Fetching too early clutters storage
    - Fetching too late defeats the purpose of “pre”-fetching

缓存预读取优化机制

- 预读取（Prefetch）缓存设计示意图



基于软件的预读取

北京大学 PEKING UNIVERSITY

- 需要编译器和指令集的支持

- Compiler/programmer places prefetch instructions
  - Requires ISA support
  - Why not use regular loads?
  - Found in ISA’s such as SPARC V-9
- Prefetch into
  - register (binding)
  - caches (non-binding): preferred in multiprocessors

```
for (I = 1; I < rows; I++)
    for (J = 1; J < columns; J++)
    {
        prefetch(&x[I+1,J]);
        sum = sum + x[I,J];
    }
```

基于硬件的预读取

- 无需要编译器和指令集的支持

- **What to prefetch?**
  - one block spatially ahead?
  - use address predictors ->
    work well for regular patterns (e.g., x, x+8, …)
- **When to prefetch?**
  - On every reference
  - On every miss
  - When prior prefetched data is referenced
  - Upon last processor reference
- **Where to put prefetched data?**
  - Buffers、caches

基于硬件的预读取

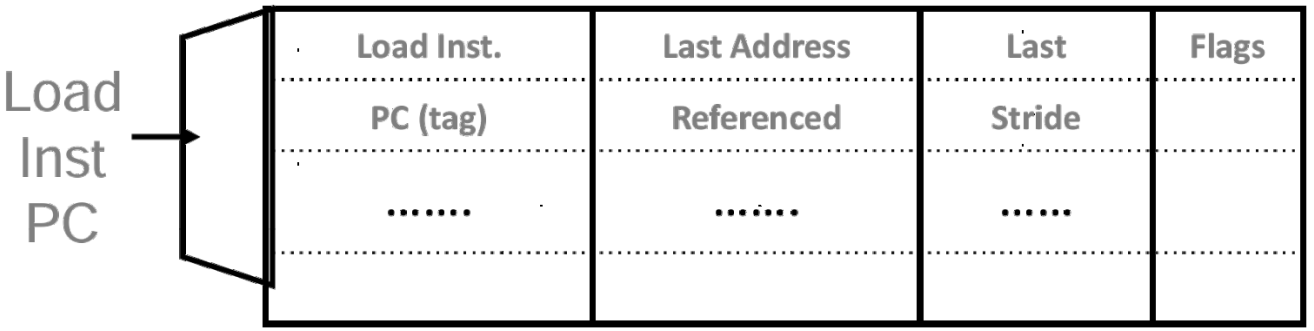
北京大学 PEKING UNIVERSITY

- 利用空间局部性的顺序预读取
  - **Works well for I-cache**
    - Instruction fetching tend to access memory sequentially
  - **Doesn't work very well for D-cache**
    - More irregular access pattern
    - regular patterns may have non-unit stride (e.g. matrix code)
  - Relatively easy to implement
    - Large cache block size already have the effect of prefetching
    - After loading one-cache line, start loading the next line automatically if the line is not in cache and the bus is not busy
  - **What if you fetch at the wrong time ….**
    - Branches, etc…

基于硬件的预读取

- Stride Prefetchers
  - **Access pattern for a particular static load is more predictable**

Reference Prediction Table (**RPT**)



  - Remember previously executed loads, their PC, the last address referenced, stride between the last two references
  - When executing a load, look up in **RPT** and compute the distance between the current data addr and the last addr
    - **If the new distance matches the old stride**
      ⇒ found a pattern, go ahead and prefetch "current addr+stride"
    - Update "last addr" and "last stride" for next lookup

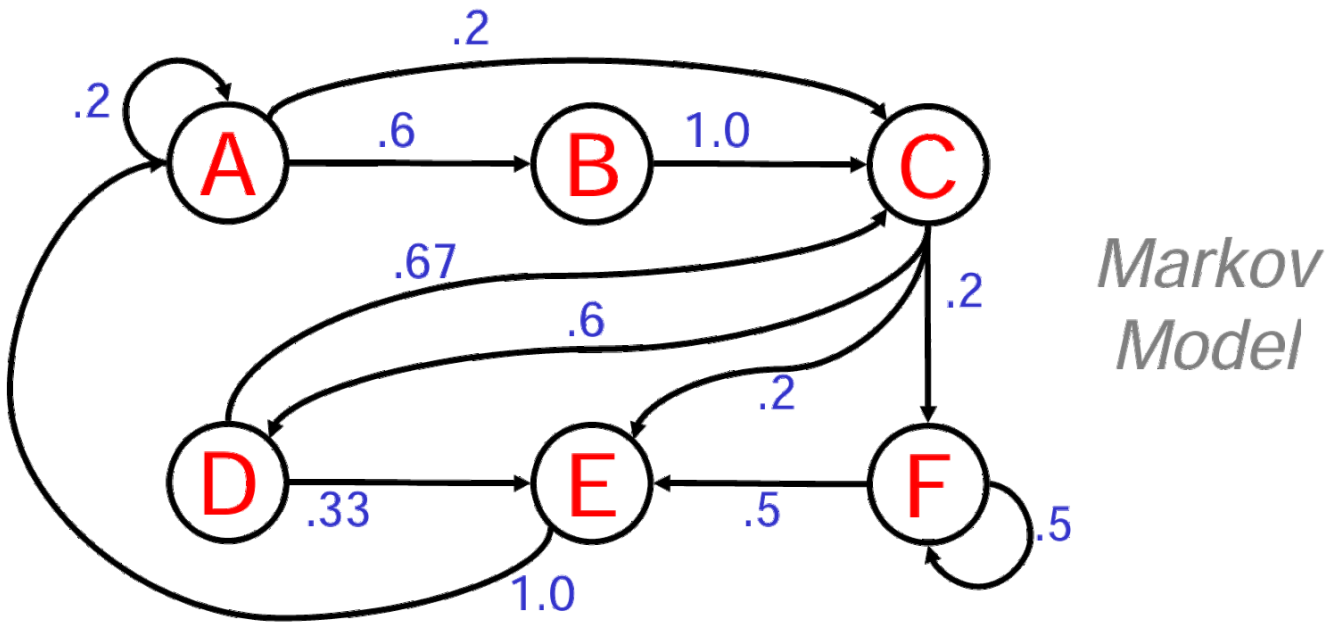
基于硬件的预读取

- Correlation-Based Prefetchers

Consider the following history of Load addresses emitted by a processor

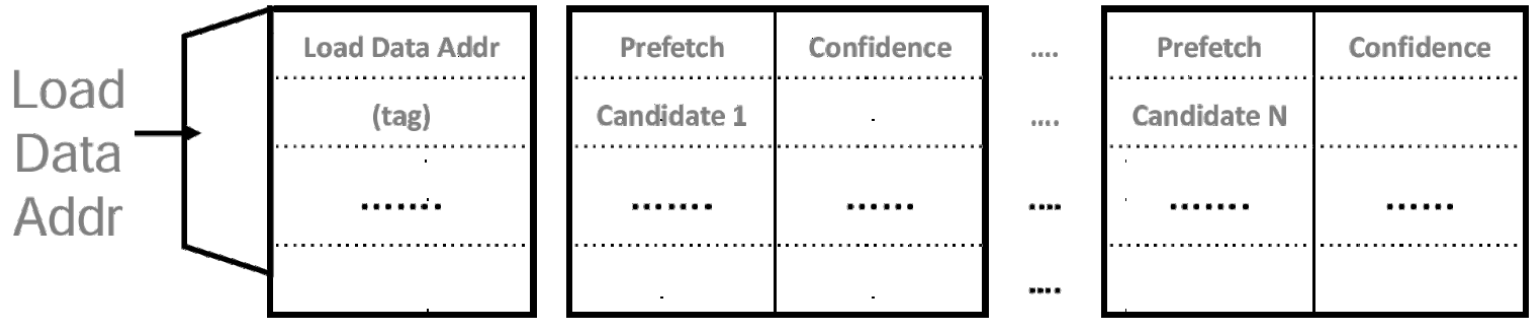
A, B, C, D, C, E, **A, C**, F, F, E, **A, A**, B, C, D, E, **A ,B** C, D, C

After referencing a particular address (say A or E), are some addresses more likely to be referenced next



基于硬件的预读取

- **Correlation-Based Prefetchers**



- **Track the likely next addresses after seeing a particular addr.**
- Prefetch accuracy is generally low so prefetch
  up to *N* next addresses to increase coverage (but this wastes bandwidth)
- Prefetch accuracy can be improved by using longer history
- Decide which address to prefetch next by looking at the last *K* load addresses instead of just the current one
  - e.g. index with the XOR of the data addresses from the last K loads
  - Using history of a couple loads can increase accuracy dramatically
  - This technique can also be applied to just the load miss stream

基于硬件的预读取

- 更多的预读取机制

Miss addresses are correlated

- Intuition: Miss sequences repeat
  - Because code/data traversals repeat

- Temporal Address Correlation
  - Prior evidence: [Joseph 97][Luk 99][Chilimbi 01][Lai 01]
  - Contrast: temporal locality

stream = ordered address sequence

目 录

CONTENTS

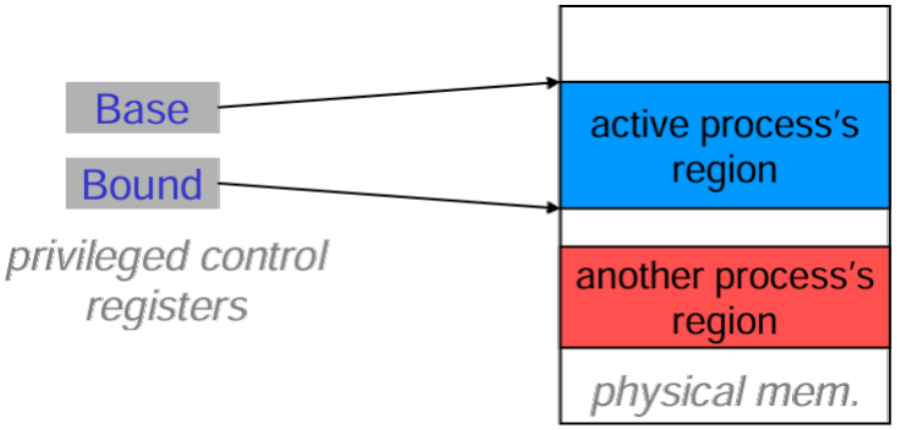
虚拟缓存机制

北京大学 PEKING UNIVERSITY

- **VM provides each process with the illusion of a large, private, uniform memory**
  - **Part A: Protection**
    - each process sees a large, contiguous memory segment without holes
      each process's memory space is private, i.e. protected from access by other processes
  - **Part B: Demand Paging**
    - capacity of secondary memory (swap space on disk)
    - at the speed of primary memory (DRAM)
  - Based on a common HW mechanism: address translation
    - user process operates on "virtual" or "effective" addresses
    - HW translates from virtual to physical on each reference
      - controls which physical locations can be named by a process
      - allows dynamic relocation of physical backing store (DRAM vs. HD)
    - VM HW and memory management policies controlled by the OS

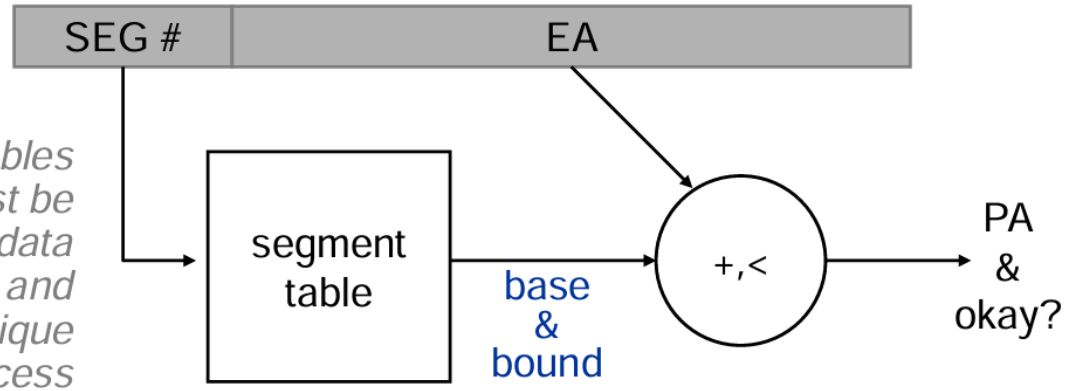
虚拟缓存机制

- Base and Bound Registers 与 Segmented Address Space



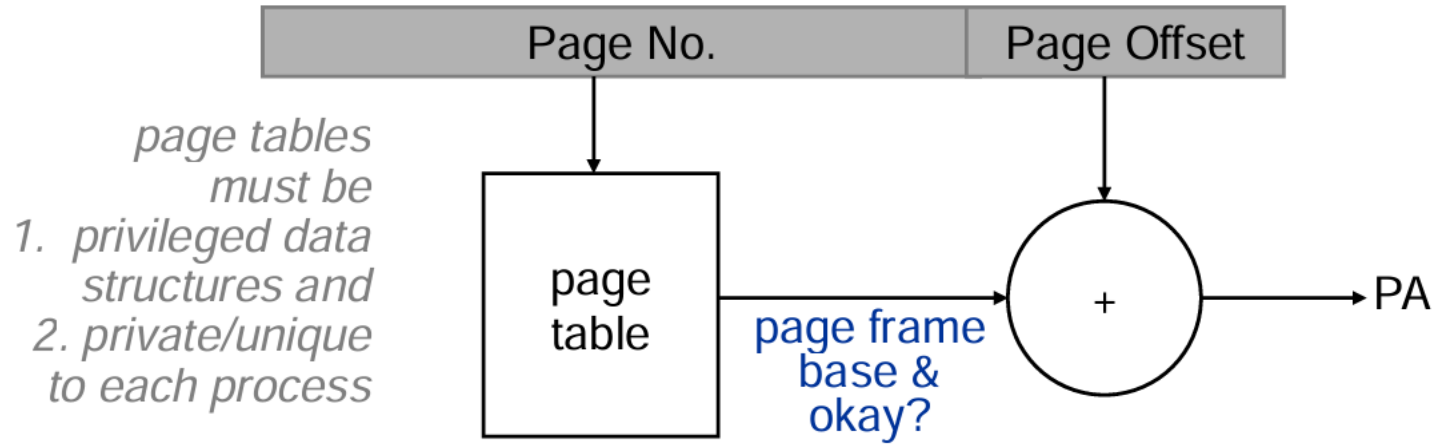
- **segment == a base and bound pair**
  - segmented addressing gives each process multiple segments
    - initially, separate code and data segments
      - 2 sets of base-and-bound reg's for inst and data fetch
        - allowed sharing code segments
    - became more and more elaborate: code, data, stack, etc.
    - also (ab)used as a way for an ISA with a small EA space to address a larger physical memory space

segment tables must be
1. privileged data structures and
2. private/unique to each process



虚拟缓存机制

- Paged Address Space
  - **Segmented addressing creates fragmentation problems**
    - a system may have plenty of unallocated memory locations
    - they are useless if they do not form a contiguous region of a sufficient size
  - **In a Paged Memory System:**
    - PA space is divided into fixed size segments (e.g. 4kbyte), more commonly known as "page frames"
    - EA is interpreted as page number and page offset



虚拟缓存机制

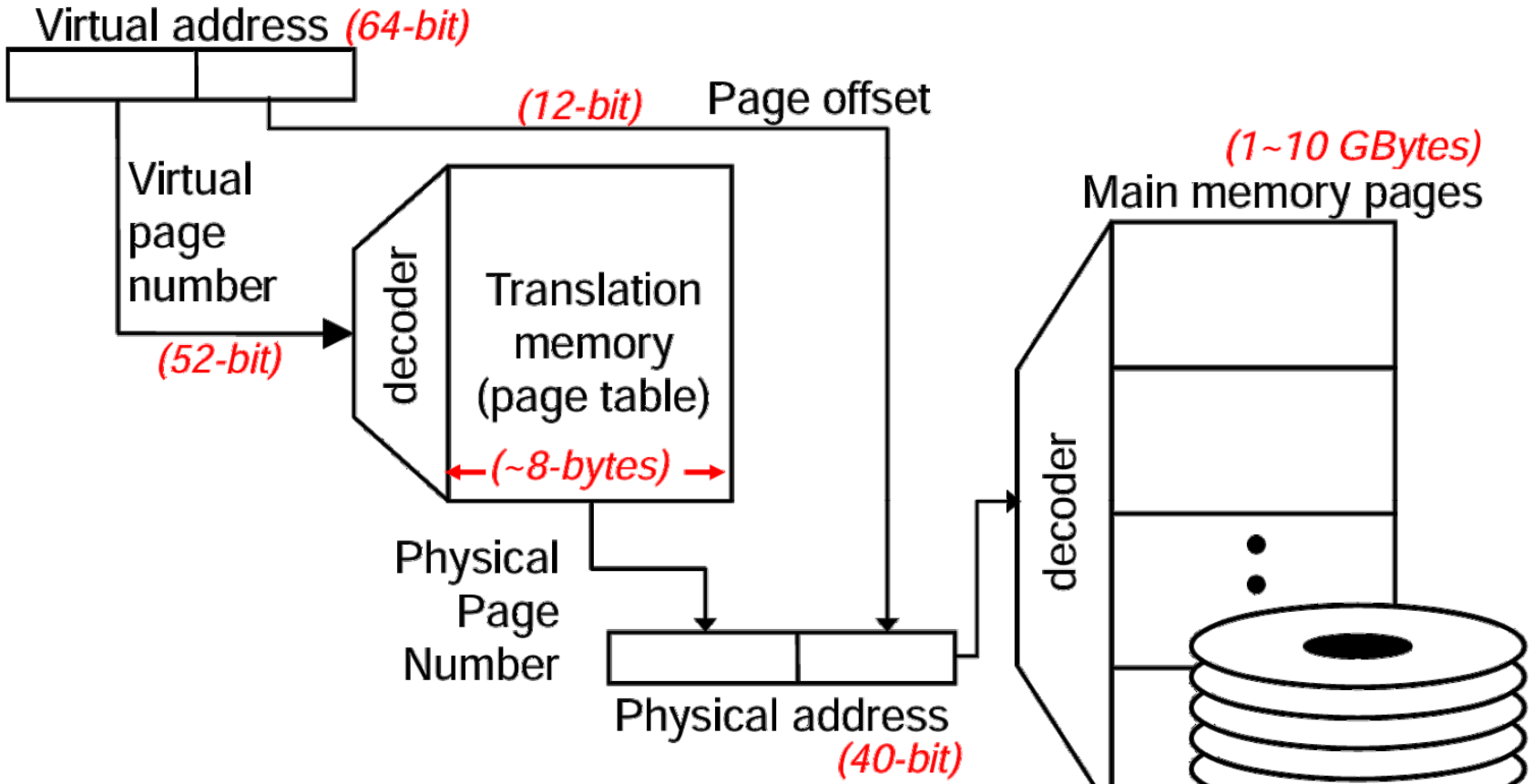
北京大学 PEKING UNIVERSITY

- **Demand Paging**

  - **Main memory and Disk as automatically managed memory hierarchies levels**
    - in the analogous to cache vs. main memory
  - Drastically different size and time scales ⇒very different design decisions
    - Early attempts von Neumann already described manual memory hierarchies
    - Brookner′ s interpretive coding, 1960
      - a software interpreter that managed paging between a 40kb main memory and a 640Kb drum
    - Atlas, 1962
      - hardware demand paging between a 32-page (512 word/page) main memory and 192 pages on drums
      - user program believes it has 192 pages

虚拟缓存机制

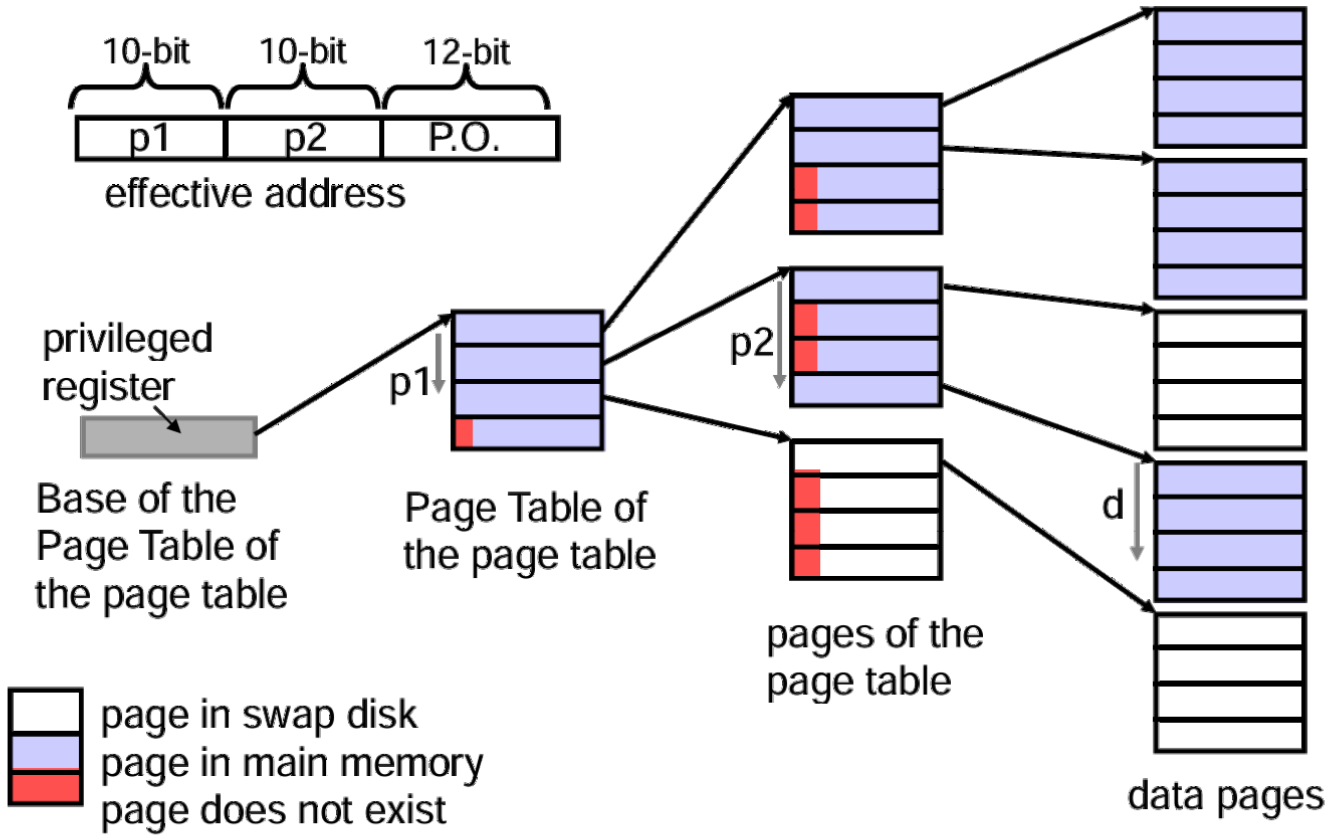
- Page-based Virtual Memory



Where to hold this translation memory and how much translation memory do we need?

虚拟缓存机制

- Hierarchical Page Table

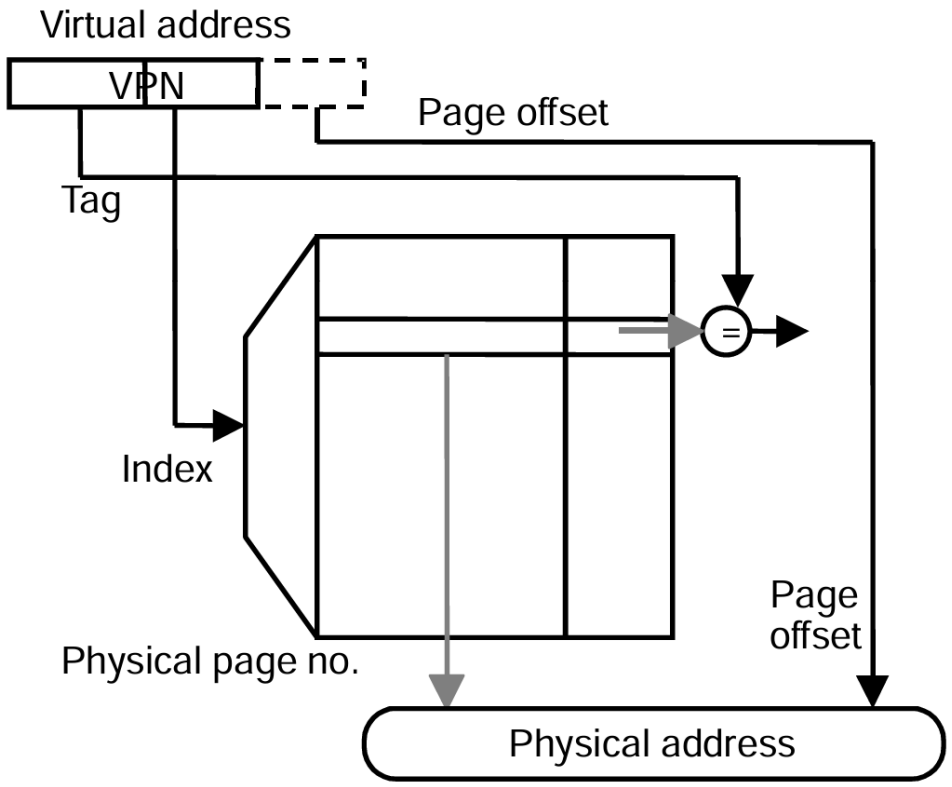


Storage of overhead of translation should be proportional to the size of physical memory and not the virtual address space

虚拟缓存机制

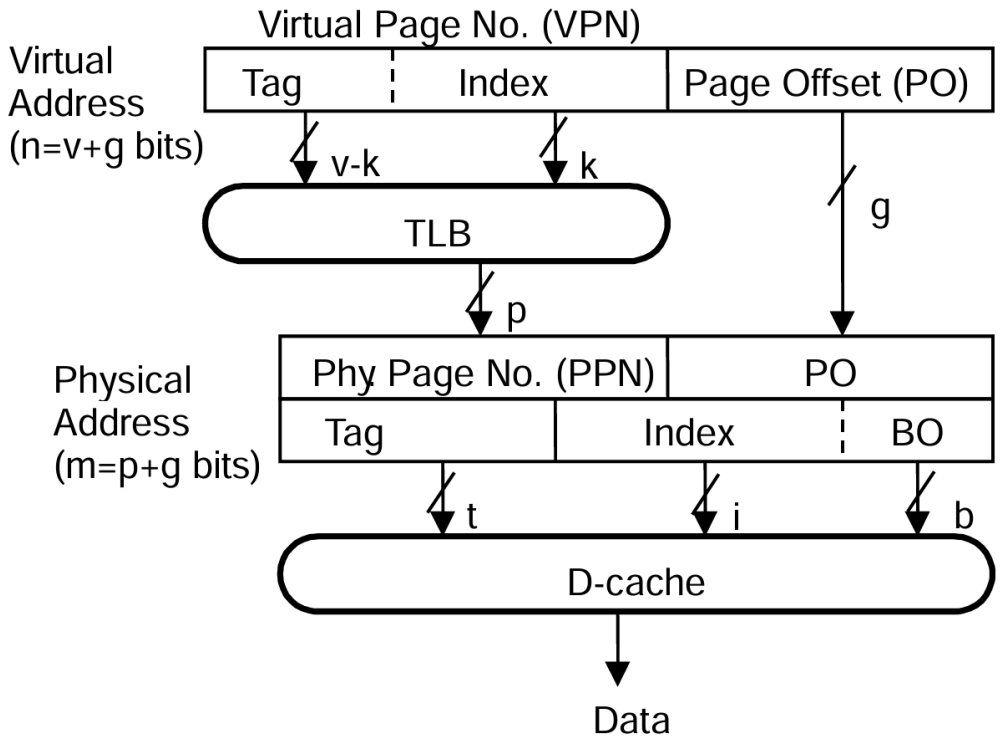
- **Translation Look-aside Buffer (TLB)**

- Essentially a cache of recent address translations
  - avoids going to the page table on every reference
  - indexed by lower bits of VPN (virtual page #)
    tag = unused bits of VPN + process ID
    data = a page-table entry
    i.e. PPN (physical page #) and access permission
  - status = valid dirty
    cache design choices (placement, replacement policy multi-level, etc) apply here too.



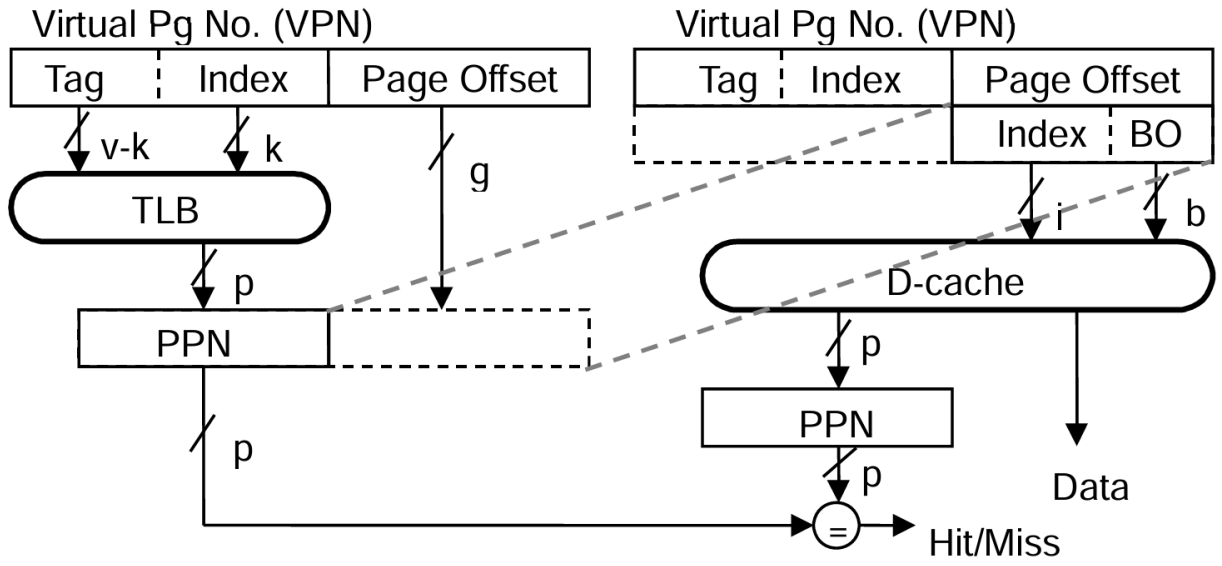
虚拟缓存机制

- **Physically-Indexed Cache 和 Virtually-Indexed Cache**



Physically-Indexed Cache

Parallel Access to TLB and Cache arrays

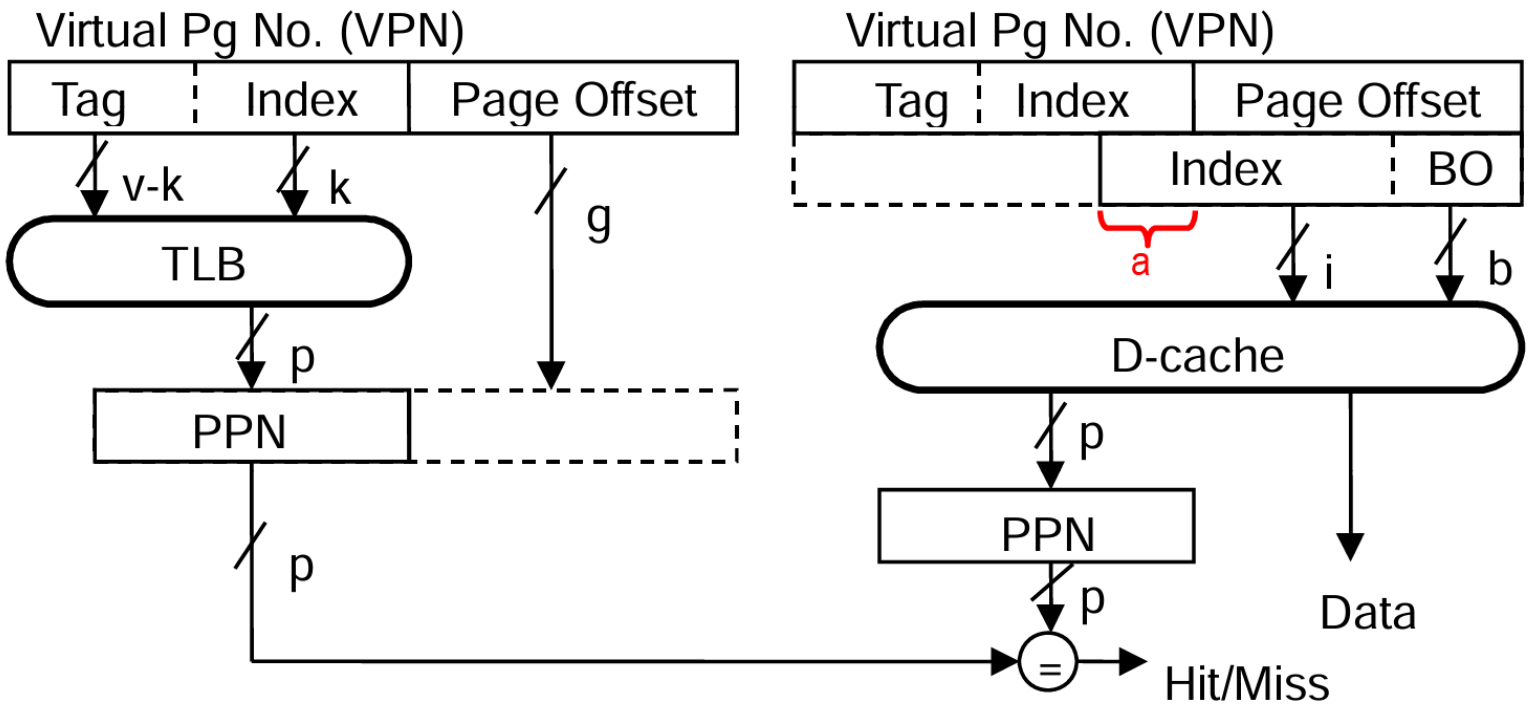


How large can a virtually indexed cache get?

Virtually-Indexed Cache

虚拟缓存机制

- **Large Virtually Indexed Cache**



If two VPNs differs in *a*, but both map to the same PPN then there is an aliasing problem

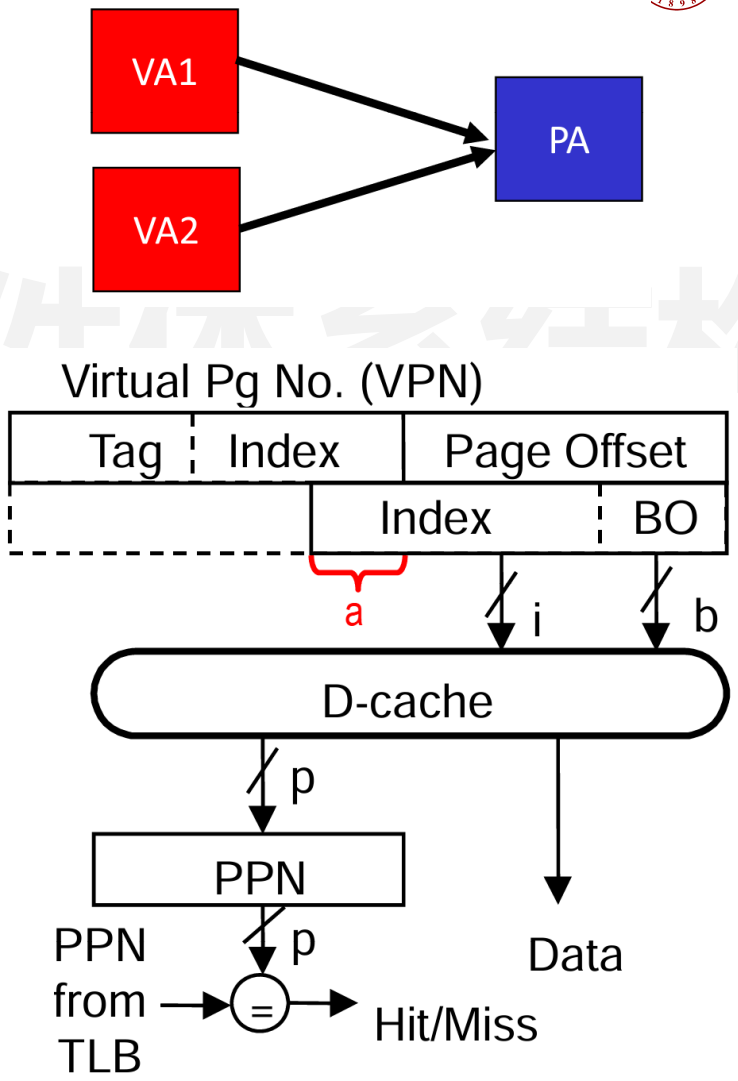
虚拟缓存机制

- Virtual Address Synonym (or Aliasing)

- **To Virtual pages that map to the same physical page**
- Within the same virtual address space across address spaces

- When VPN bits are used in indexing, two virtual addresses that map to the same physical address can end up sitting in two cache lines
- In other words, two copies of the same physical memory location may exist in the cache

  ⇒ modification to one copy won't be visible in the other won t be visible in the other



If the two VPNs do not differ in *a* then there is no aliasing problem

虚拟缓存机制

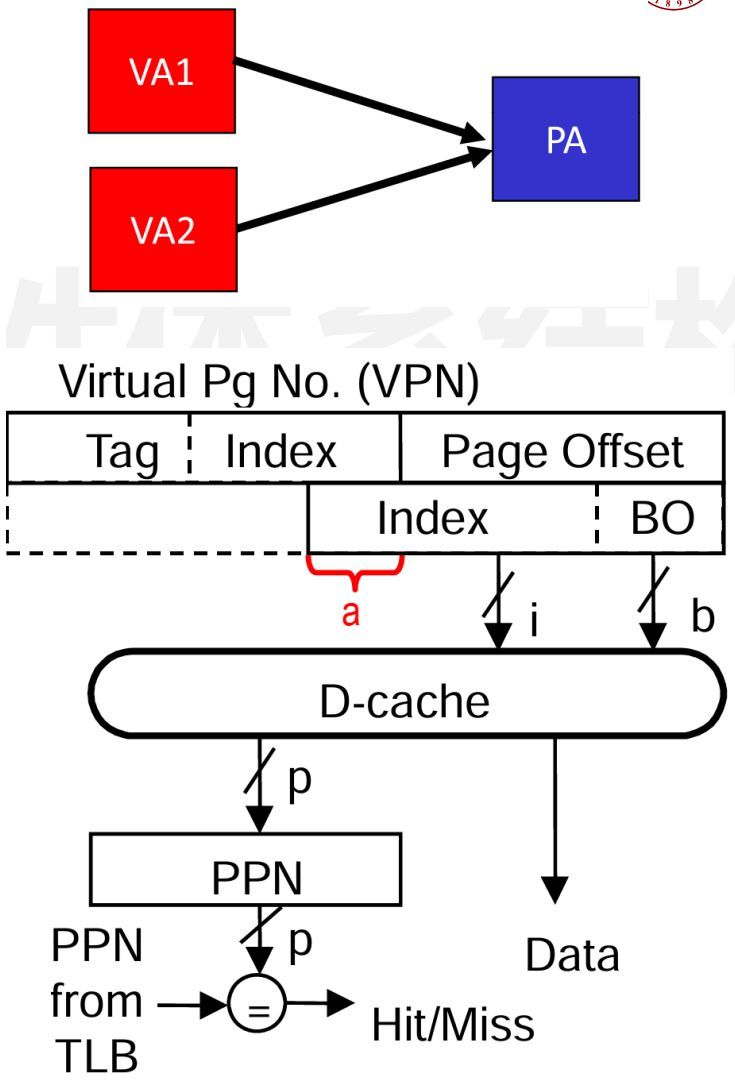
- **Virtual Address Synonym (or Aliasing) Solutions**

Limit cache size to page size times associativity
- get index from page offset

Search all sets in parallel
- 64K 4-way cache, 4K pages, search 4 sets (16 entries)
- Slow!

Restrict page placement in OS
- make sure index(VA) = index(PA)

Eliminate by OS convention
- single virtual space
- restrictive sharing model



目 录

CONTENTS

数据并行

- **Data-level parallelism (DLP)**

Clock rate and IPC are at odds with each other

- Pipelining
  - Fast clock
  - Increased hazards lower IPC
- Wide issue
  - Higher IPC
  - N^2 bypassing slows down clock

Can we get both fast clock and wide issue?

- Yes, but with a parallelism model less general than ILP

Data-level parallelism (DLP)

- Single operation repeated on multiple data elements
- Less general than ILP: parallel insns are same operation

数据并行

- Data-level parallelism (DLP)

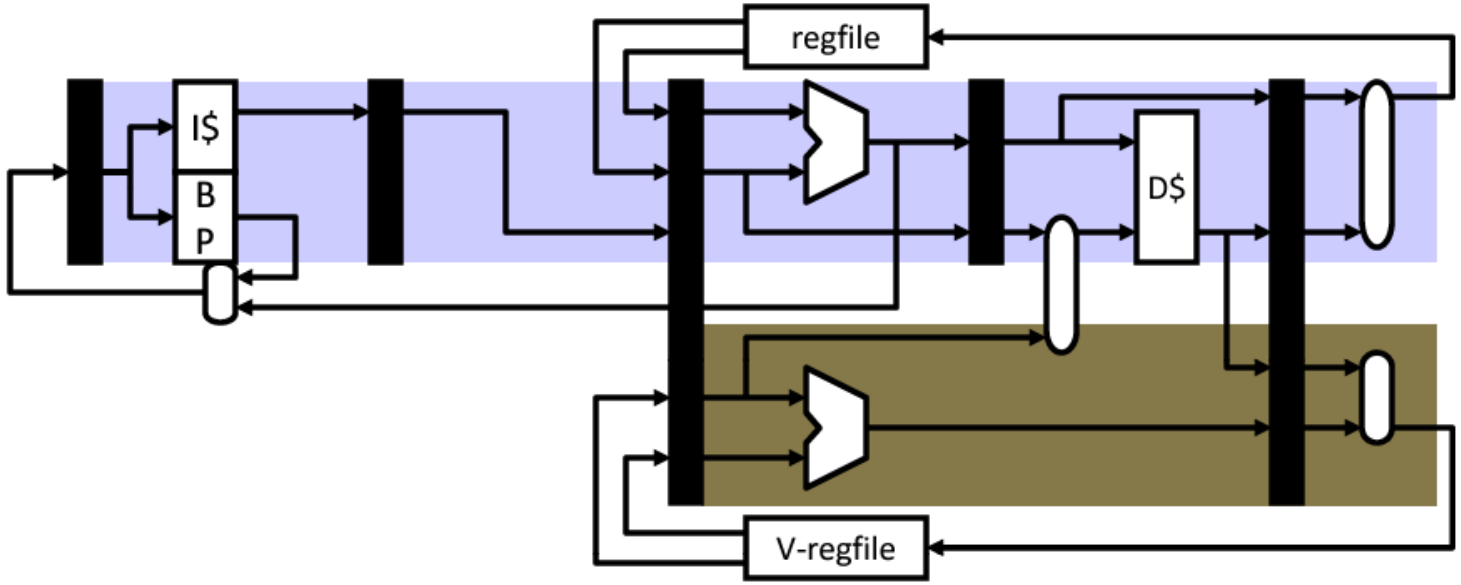
```
for (I = 0; I < 100; I++)
   Z[I] = A*X[I] + Y[I];
```

```
L0: ldf X(r1),f1          // I is in r1
    mulf f0,f1,f2         // A is in f0
    ldf Y(r1),f3
    addf f2,f3,f4
    stf f4,Z(r1)
    addi r1,4,r1
    blti r1,400,L0
```

One example of DLP: **inner loop-level parallelism**

- Iterations can be performed in parallel

- Exploiting DLP With Vectors

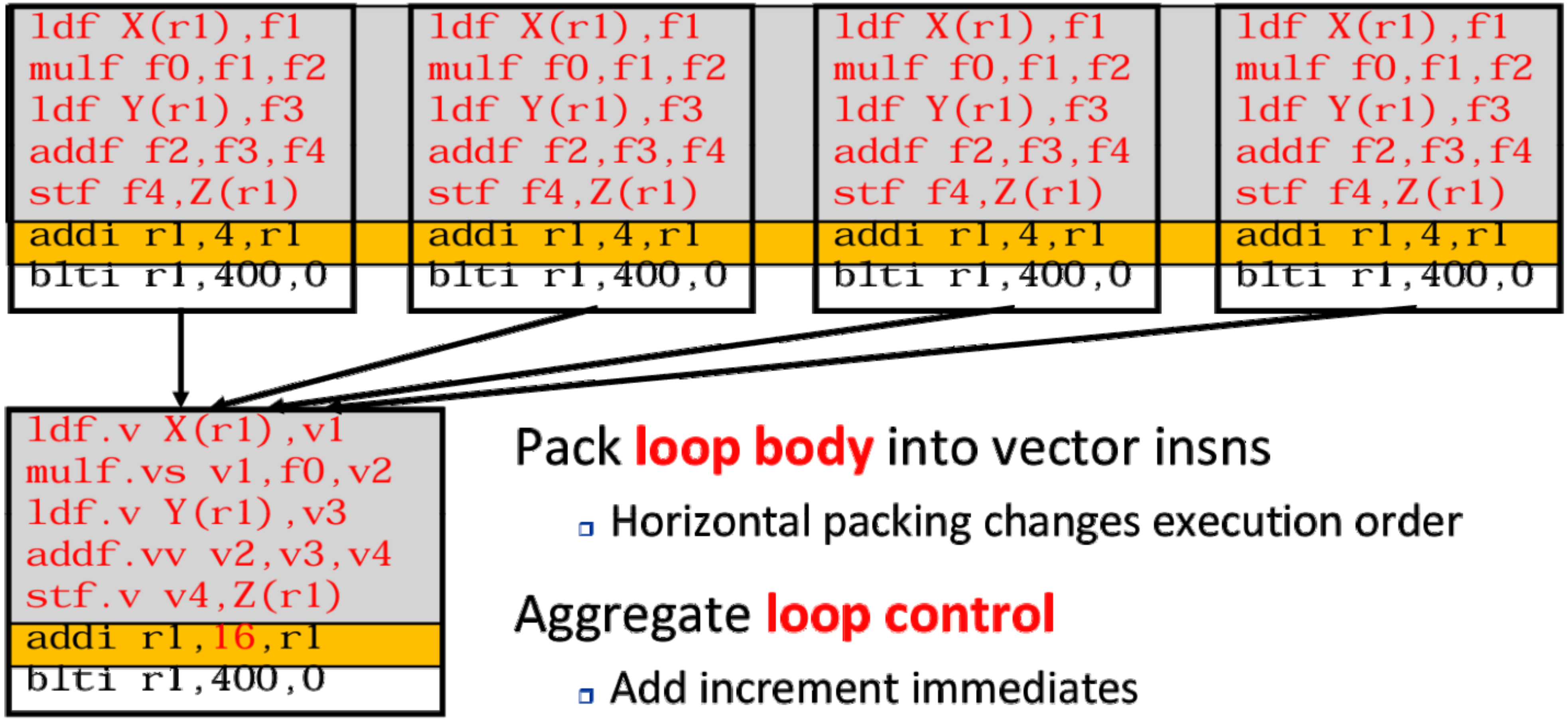


One way to exploit DLP: **vectors**

- Extend processor with **vector "data type"**
- Vector: array of MVL 32-bit FP numbers
  - **Maximum vector length (MVL)**: typically 8–64
- **Vector register file**: 8–16 vector registers (v0–v15)

数据并行

- **Exploiting DLP With Vectors**



Pack **loop body** into vector insns

- Horizontal packing changes execution order

Aggregate **loop control**

- Add increment immediates

- **MIPS-V Instructions**

Vector-vector instructions

- operate on two vectors
- produce a third vector
- addv v1, v2, v3

Vector-scalar instructions

- operate on one vector and one scalar
- addv v1, f0, v3

Vector ld/st instructions

- ld/st a vector from memory into a vector register
- operates on contiguous addresses
- lv [r1], v1 ; v[I] = M[r1+I]
- sv v1, [r1] ; M[r1+I] = v[I]

ld/st vector with stride

- vectors are not always contiguous in memory
- add non-unit stride on each access
- lvws [r1,r2], v1 ; v[I] = M[r1+I*r2]
- svws v1, [r1, r2] ; M[r1+I*r2] = v[I]

ld/st indexed

- indirect accesses through an index vector
- lvws [r1,v2], v1 ; v[I] = M[r1+v2[I]]
- svws v1, [r1, v2] ; M[r1+v2[I]] = v[I]

- MIPS-V Instructions

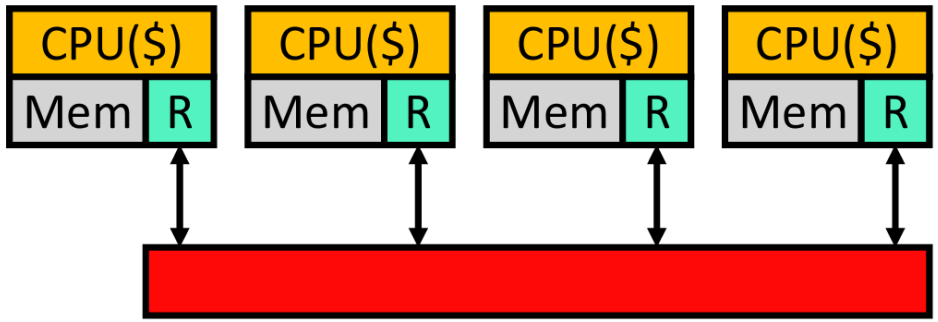
DAXPY: double-precision a * x + y

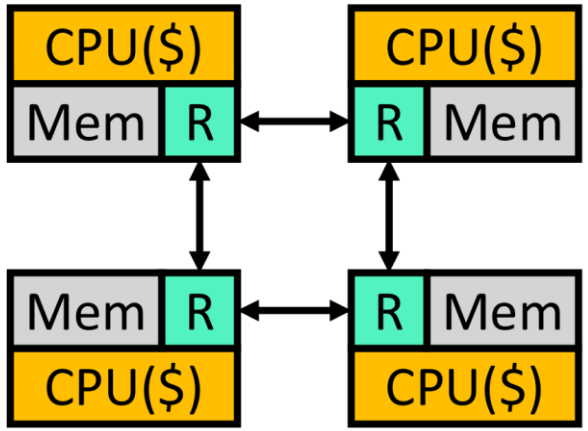
```
for (I=1; I<=64;I++)
  y[I] = a*x[I]+y[I]

VLR 64
ld      [a], f0
lv      [rx], v1
multv v1, f0, v2
lv      [ry], v3
addv v2, v3, v4
sv      v4, [ry]
```

6 instructions total as compared to 600 MIPS instructions!

多核架构的组织形式

- **Bus-based Multi-core 和 Network-on-chip：Chip Multiprocessor (CMP)**



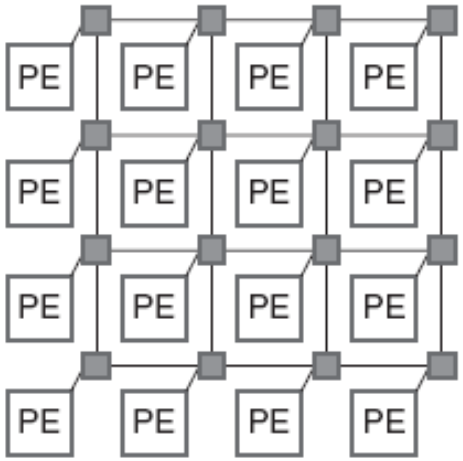


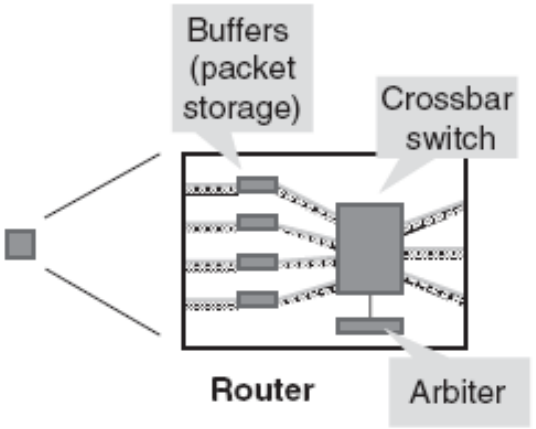
- Shared network: e.g., bus
  - Low latency
  - Low bandwidth: doesn’ t scale beyond ~16 processors
  - Shared property simplifies cach e coherence protocols (later)

- Point-to-point network: e.g., mesh or ring
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are complex

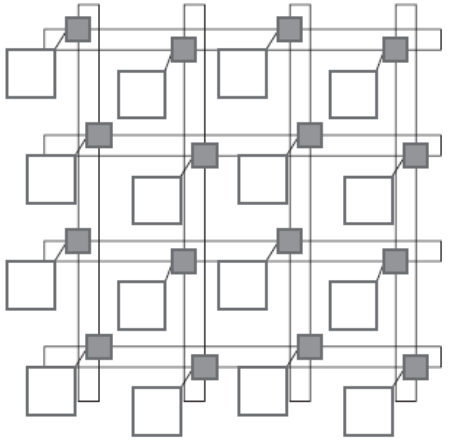
多核架构的组织形式

- Network-on-chip：Chip Multiprocessor (CMP)

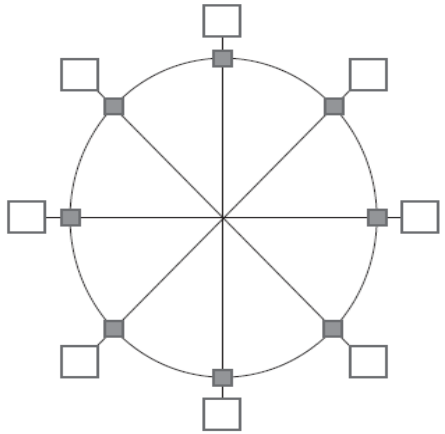




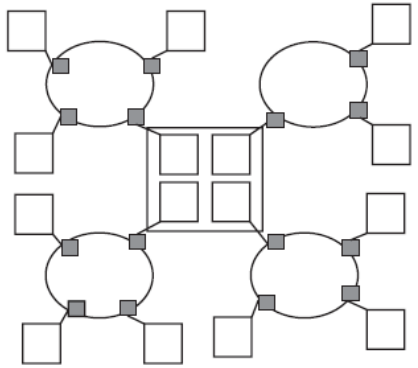
Mesh Topology



2-D torus topology



Octagon topology



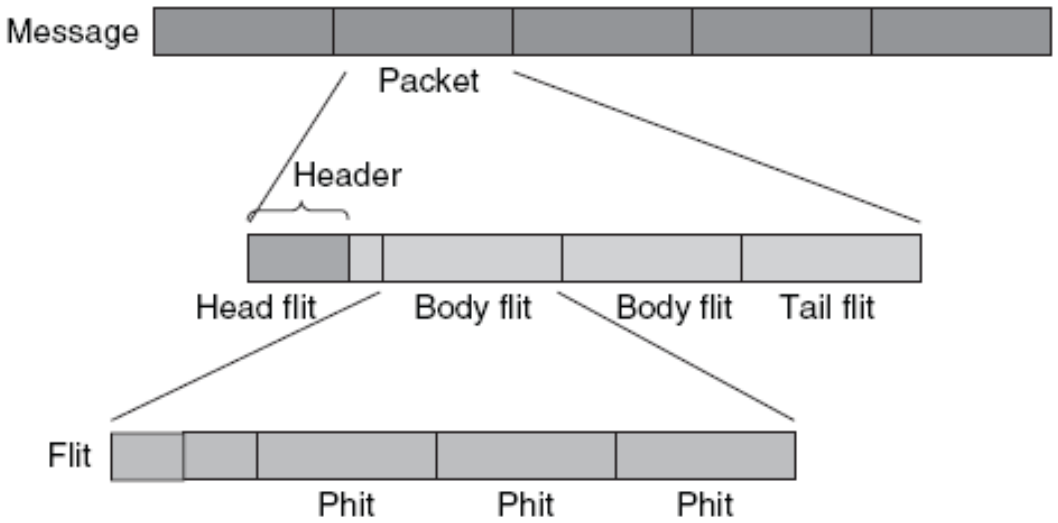
Irregular or ad hoc network topologies

多核架构的组织形式

- **Network-on-chip：Chip Multiprocessor (CMP)**

Switching strategies

- Determine how data flows through routers in the network
- Define granularity of data transfer and applied switching technique
  - phit is a unit of data that is transferred on a link in a single cycle
  - typically, phit size = flit size



多核架构的组织形式

北京大学 PEKING UNIVERSITY

- **Network-on-chip：Chip Multiprocessor (CMP)**

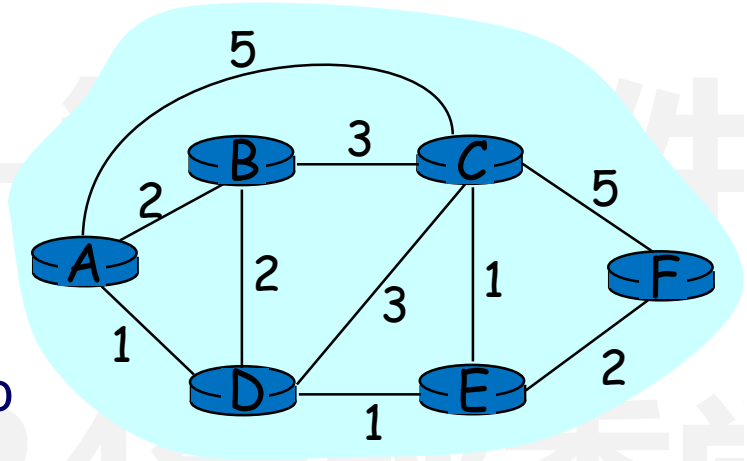
- Static and dynamic routing
  - **static routing**: fixed paths are used to transfer data between a particular source and destination
    - does not take into account current state of the network
  - advantages of static routing:
    - easy to implement, since very little additional router logic is required
    - in-order packet delivery if single path is used
  - **dynamic routing**: routing decisions are made according to the current state of the network
    - considering factors such as availability and load on links
  - path between source and destination may change over time
    - as traffic conditions and requirements of the application change
  - more resources needed to monitor state of the network and dynamically change routing paths
  - able to better distribute traffic in a network

多核架构的组织形式

- **Static Routing Tables**

Dijkstra's algorithm

- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ('source") to all other nodes
  - gives routing table for that node
- iterative: after k iterations, know least cost path to k dest.'s



CENTRAL ROUTING DIRECTORY

To Node \ From Node	1	2	3	4	5	6
1	—	1	5	2	4	5
2	2	—	5	2	4	5
3	4	3	—	5	3	5
4	4	4	5	—	4	5
5	4	4	5	5	—	5
6	4	4	5	5	6	—

Node 1 Directory

Destination	Next Node
2	2
3	4
4	4
5	4
6	4

Node 2 Directory

Destination	Next Node
1	1
3	3
4	4
5	4
6	4

Node 3 Directory

Destination	Next Node
1	5
2	5
4	5
5	5
6	5

Node 4 Directory

Destination	Next Node
1	2
2	2
3	5
5	5
6	5

Node 5 Directory

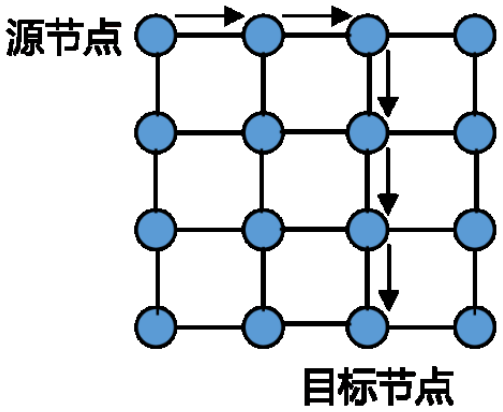
Destination	Next Node
1	4
2	4
3	3
4	4
6	6

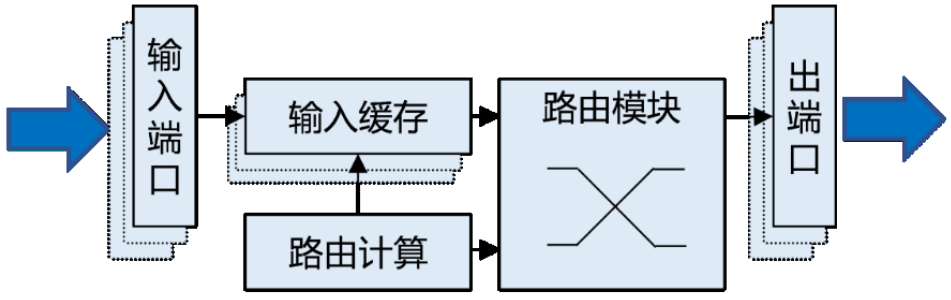
Node 6 Directory

Destination	Next Node
1	5
2	5
3	5
4	5
5	5

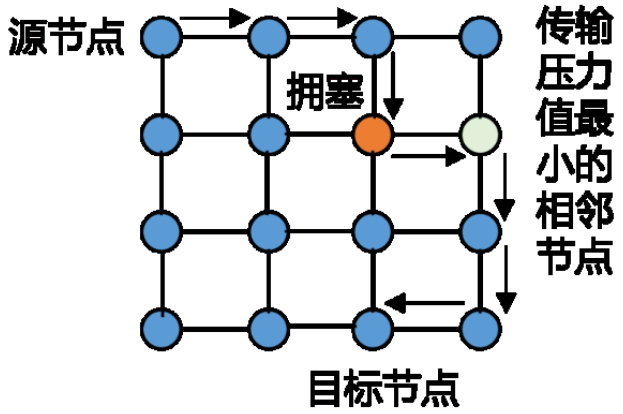
多核架构的组织形式

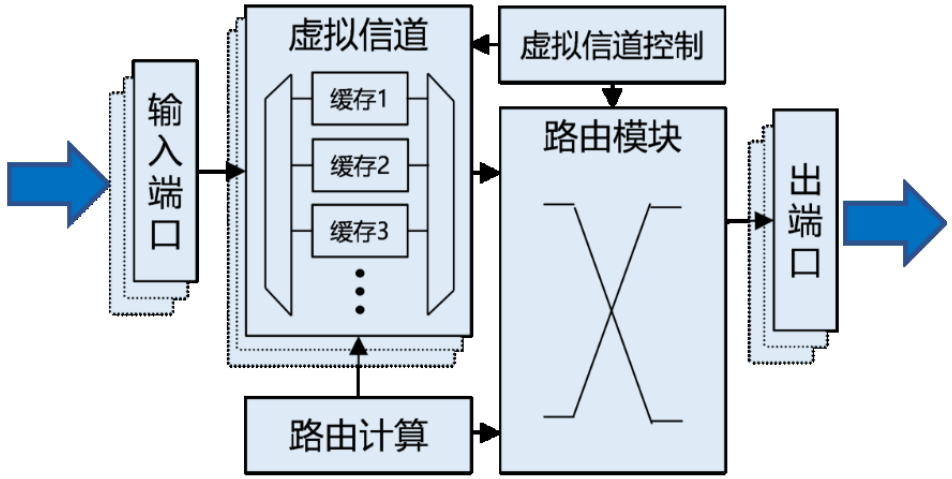
- Dynamic Routing





传统路由模块

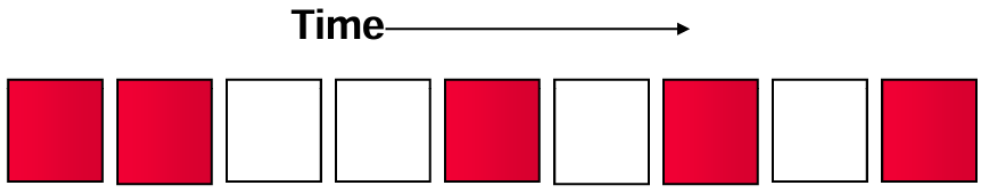




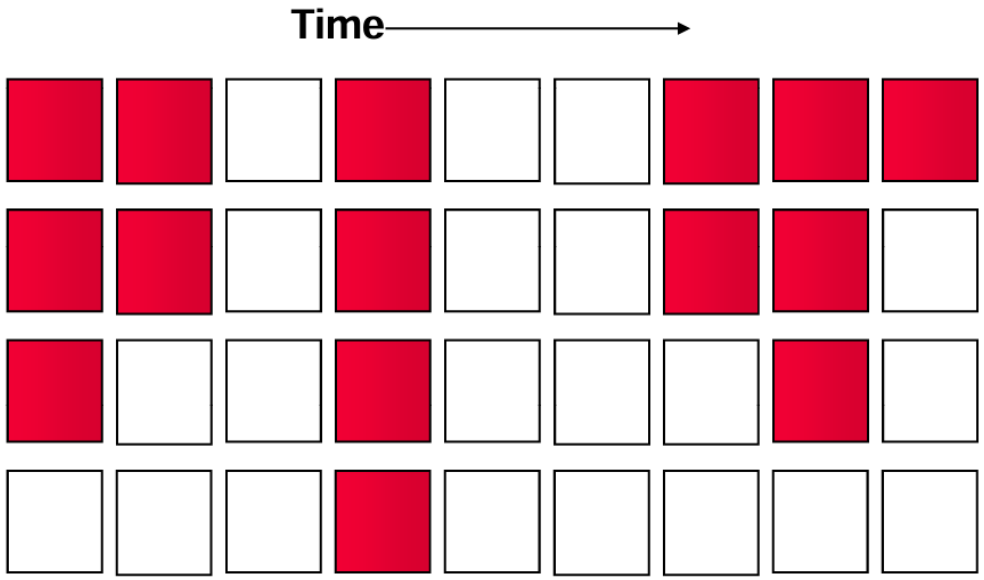
虚拟信道路由模块

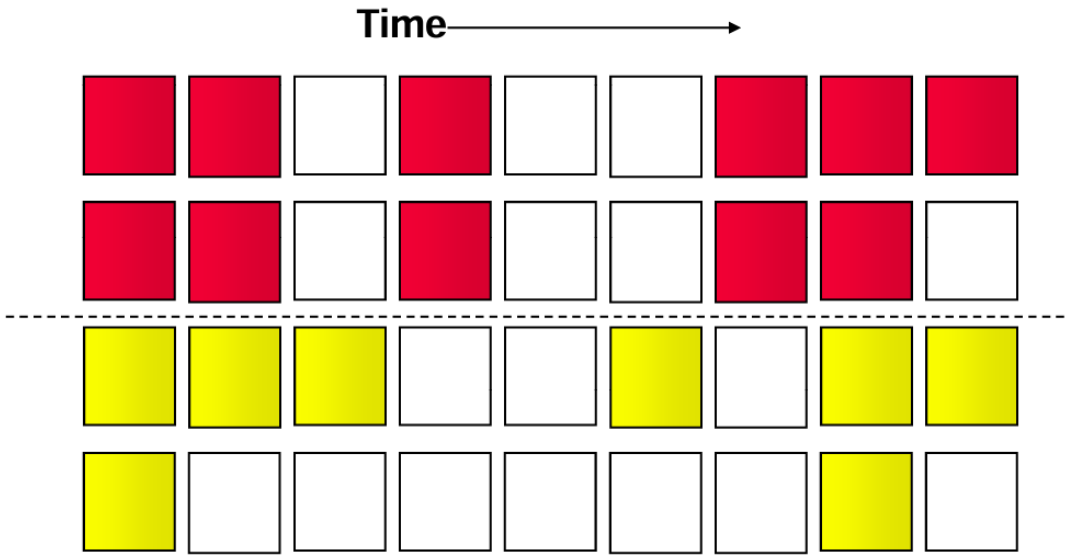
多线程组织形式

- Instruction Issue



Scalar Pipeline



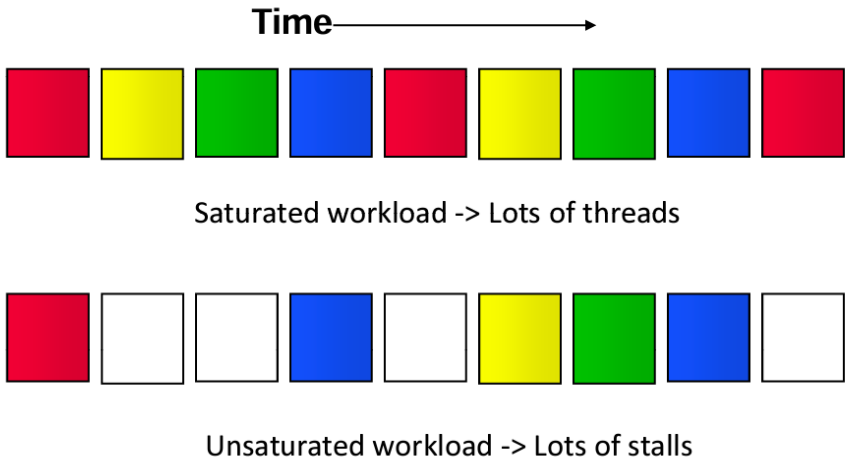


Limited utilization when only running one thread

Superscalar leads to more performance, but lower utilization

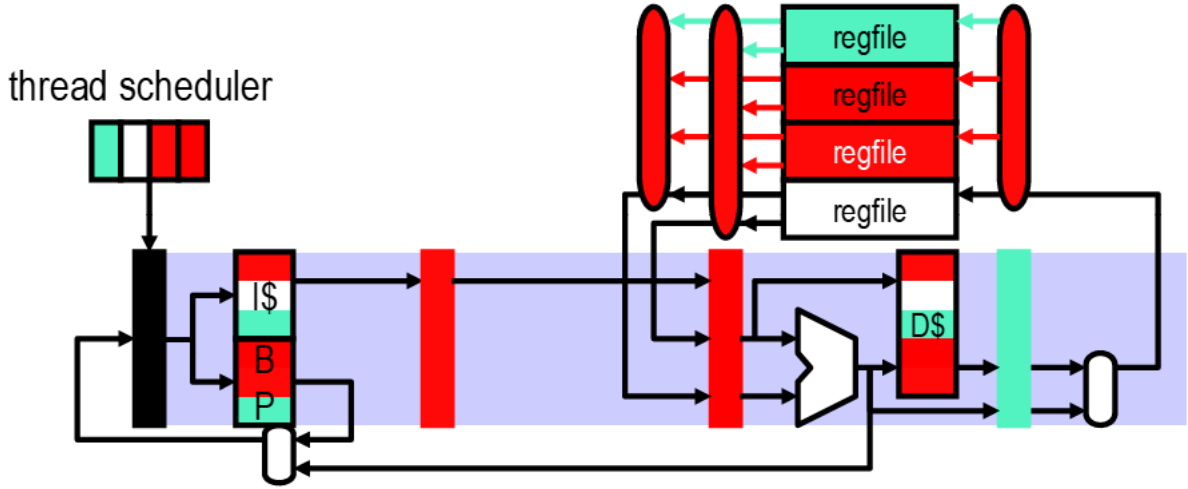
多线程组织形式

- **Fine Grained Multithreading (FGMT)**



Intra-thread dependencies still limit performance

- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once

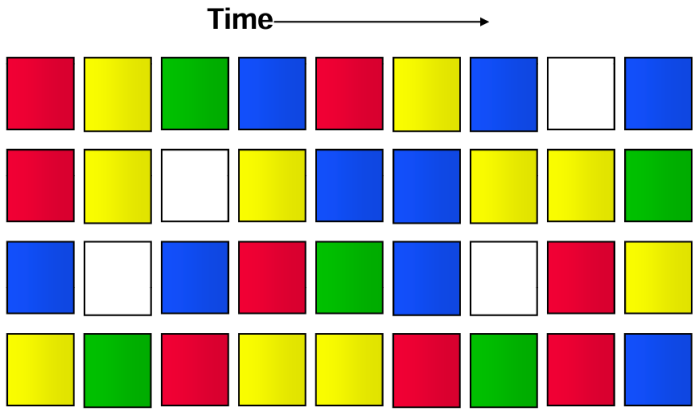


Many threads → many register files

多线程组织形式

北京大学 PEKING UNIVERSITY

- **Simultaneous Multithreading (SMT)**

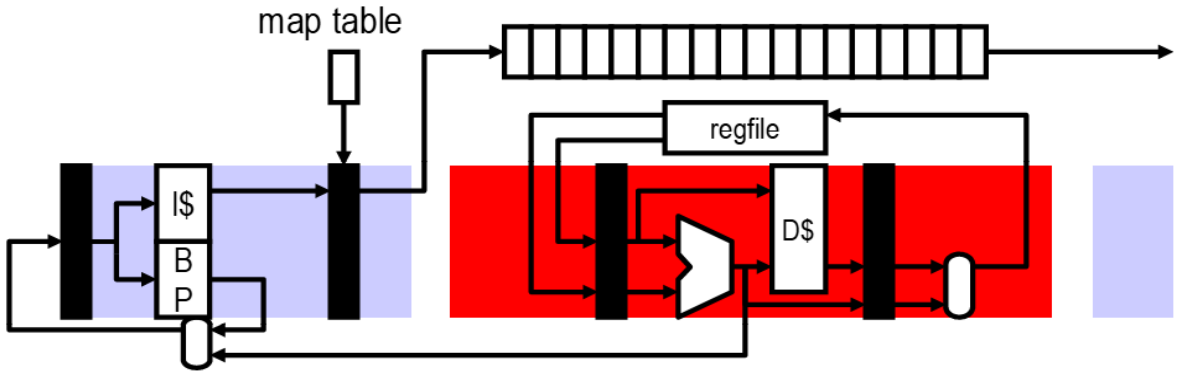


Maximum utilization of function units by independent operations

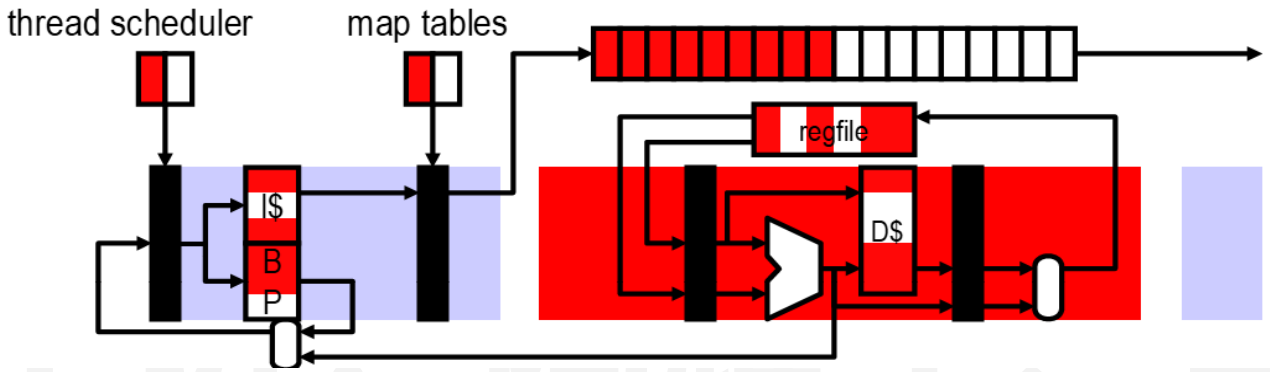
- Can we multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of D$ (L1) miss latency
- **Simultaneous multithreading (SMT)**
  - Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  - Sacrifices some single thread performance
  - Thread scheduling policy
    - Round-robin (just like FGMT)
  - Pipeline partitioning
    - Dynamic, hmmm…
  - Example: Pentium4 (hyper-threading): 5-way issue, 2 threads
  - Another example: Alpha 21464: 8-way issue, 4 threads (canceled)

多线程组织形式

- Simultaneous Multithreading (SMT)



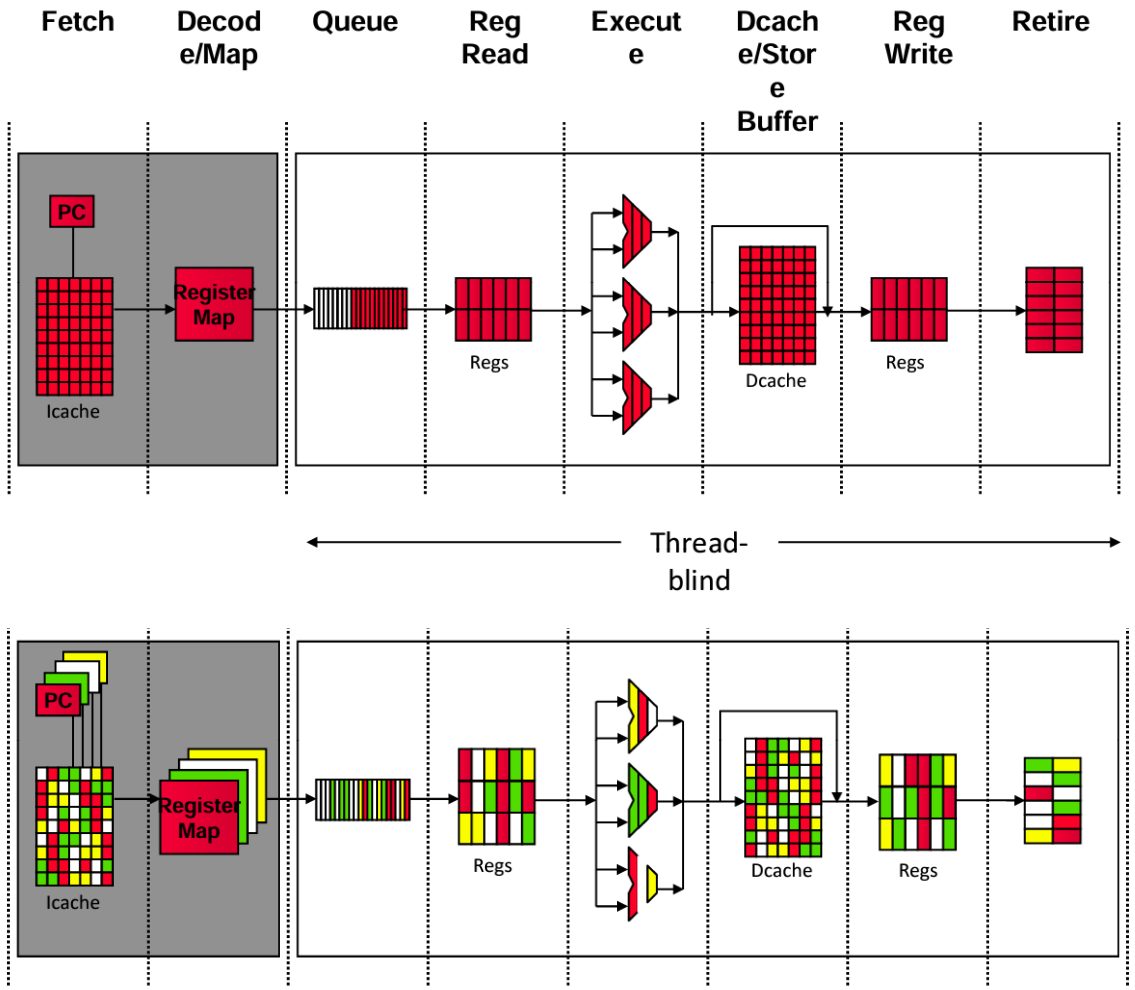
- SMT
  - Replicate map table, share physical register file



- Large map table and physical register file
  - #mt-entries = (**#threads** * #arch-regs)
  - #phys-regs = (**#threads** * #arch-regs) + #in-flight insns

多线程组织形式

- Simultaneous Multithreading (SMT) Pipeline



SMT Changes

Basic pipeline – unchanged

Replicated resources
- Program counters
- Register maps

Shared resources
- Register file (size increased)
- Instruction queue
- First and second level caches
- Translation buffers
- Branch predictor

多线程组织形式

北京大学 PEKING UNIVERSITY

- **Simultaneous Multithreading (SMT) vs Chip Multiprocessor (CMP)**

  - If you wanted to run multiple threads would you build a…
    - Chip multiprocessor (CMP): multiple separate pipelines?
    - A multithreaded processor (SMT): a single larger pipeline?
  - **Both will get you throughput on multiple threads**
    - CMP will be simpler, possibly faster clock
    - SMT will get you better performance (IPC) on a single thread
      - SMT is basically an ILP engine that converts TLP to ILP
      - CMP is mainly a TLP engine
  - **Again, do both**
    - Sun's Niagara (UltraSPARC T1)
    - 8 processors, each with 4-threads (coarse-grained threading)
    - 1Ghz clock, in-order, short pipeline (6 stages or so)
    - Designed for power-efficient "throughput computing"