Computer Architecture
Course code: 0521292B

11. Optimizing Cache Performance

Jianhua Li
College of Computer and Information
Hefei University of Technology

How to Improve Cache Performance

- Three fundamental goals:
  - Reducing miss rate
    - Remind: reducing miss rate can reduce performance if more costly-to-re-fetch blocks are evicted
  - Reducing miss latency or miss cost
  - Reducing hit latency or hit cost

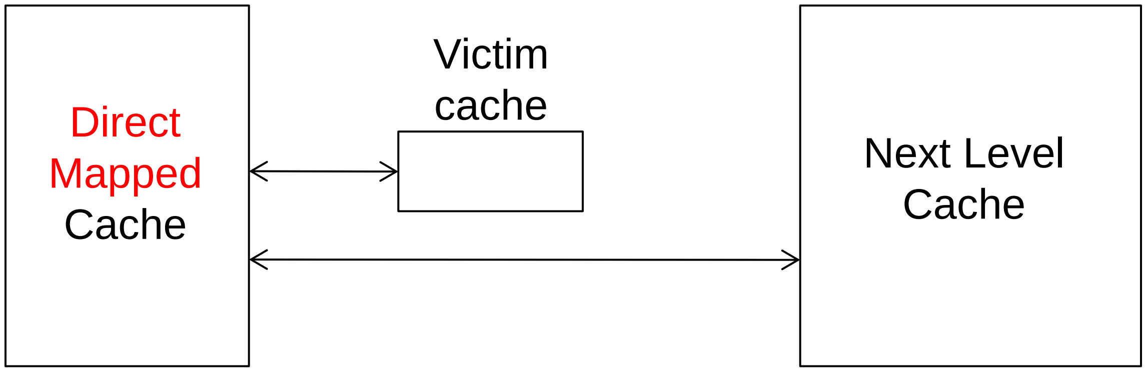
Basic Tricks to Improve Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Issues in multicore caches

Ways of Reducing Conflict Misses

- Instead of building highly-associative caches, many other approaches in the literature:
  - Victim Caches
  - Hashed/randomized Index Functions
  - Pseudo Associativity
  - Skewed Associative Caches
  - …

Victim Cache: Reducing Conflict Misses



- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990. 【若想掌握得透彻，请阅读原文。】
- Idea: Use a small fully associative buffer (victim cache) to store evicted blocks

  + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)

  -- Increases miss latency if accessed serially with L2;

  -- Adds complexity

Hashing and Pseudo-Associativity

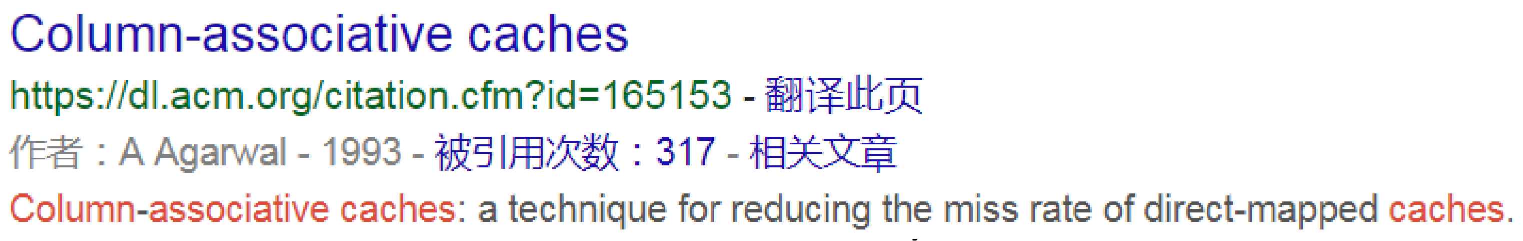
- Hashing: Use better “randomizing” index functions

  + can reduce conflict misses

  - by distributing the accessed memory blocks more evenly to sets
  - Example of conflicting accesses: stride access pattern where stride value equals number of sets in cache

  -- More complex to implement: can lengthen critical path

- Pseudo-associativity
  – View each "way" of the set as a separate direct-mapped cache
  – Ways are searched in sequence (first, second, …)


Column-associative caches
https://dl.acm.org/citation.cfm?id=165153 - 翻译此页
作者：A Agarwal - 1993 - 被引用次数：317 - 相关文章
Column-associative caches: a technique for reducing the miss rate of direct-mapped caches.

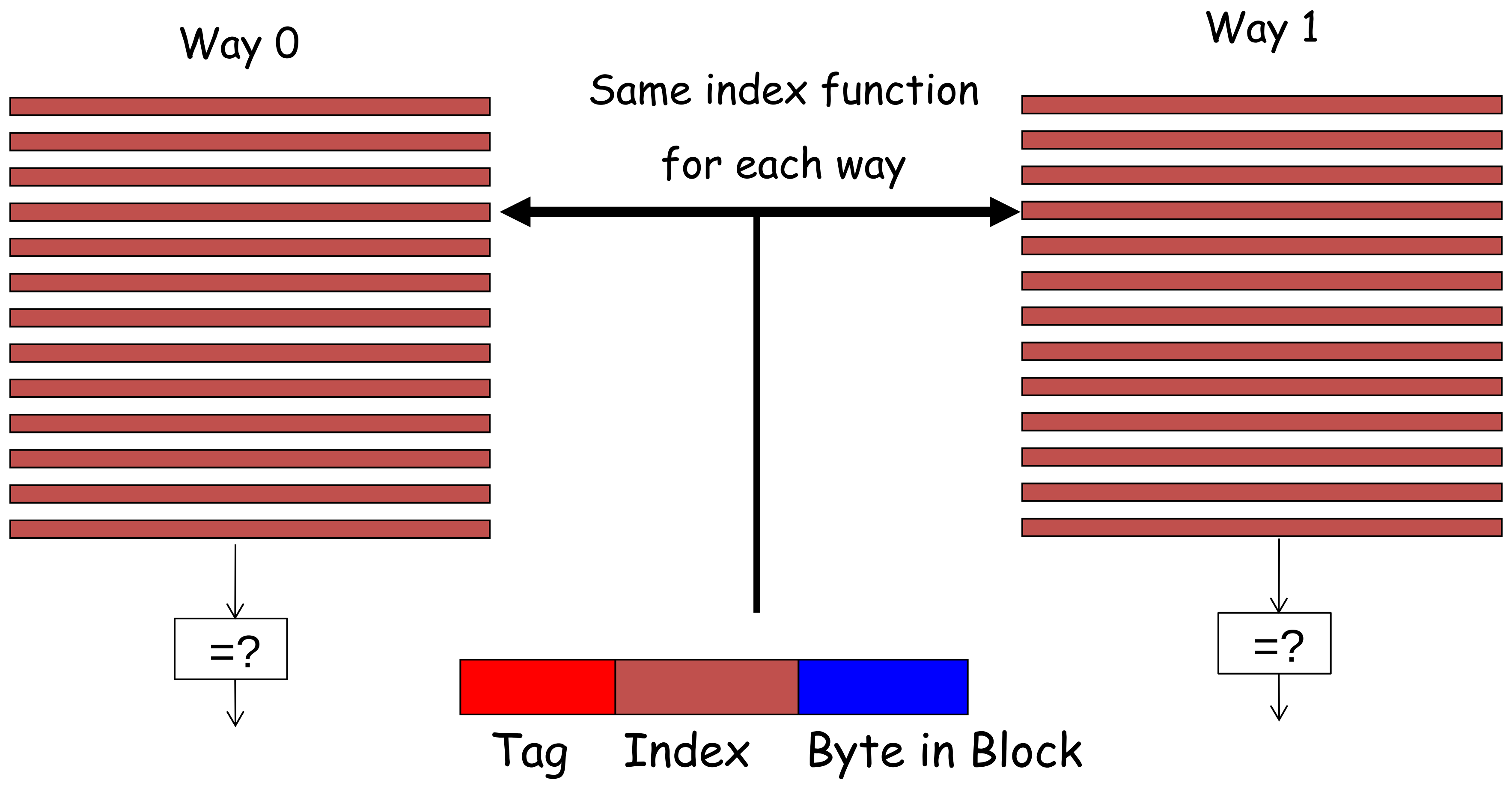
Skewed Associative Caches

- Idea: Reduce conflict misses by using different index functions for each cache way
- Seznec, "A Case for Two-Way Skewed-Associative Caches," ISCA 1993.

A case for two-way skewed-associative caches - ACM Digital ...
dl.acm.org › citation ▾ 翻译此页
作者：A Seznec - 1993 - 被引用次数：364 - 相关文章
A case for two-way skewed-associative caches, Published by ACM 1993 Article. Bibliometrics Data Bibliometrics. · Citation Count: 97 · Downloads (cumulative): ...
Authors · References · Cited By · Publication

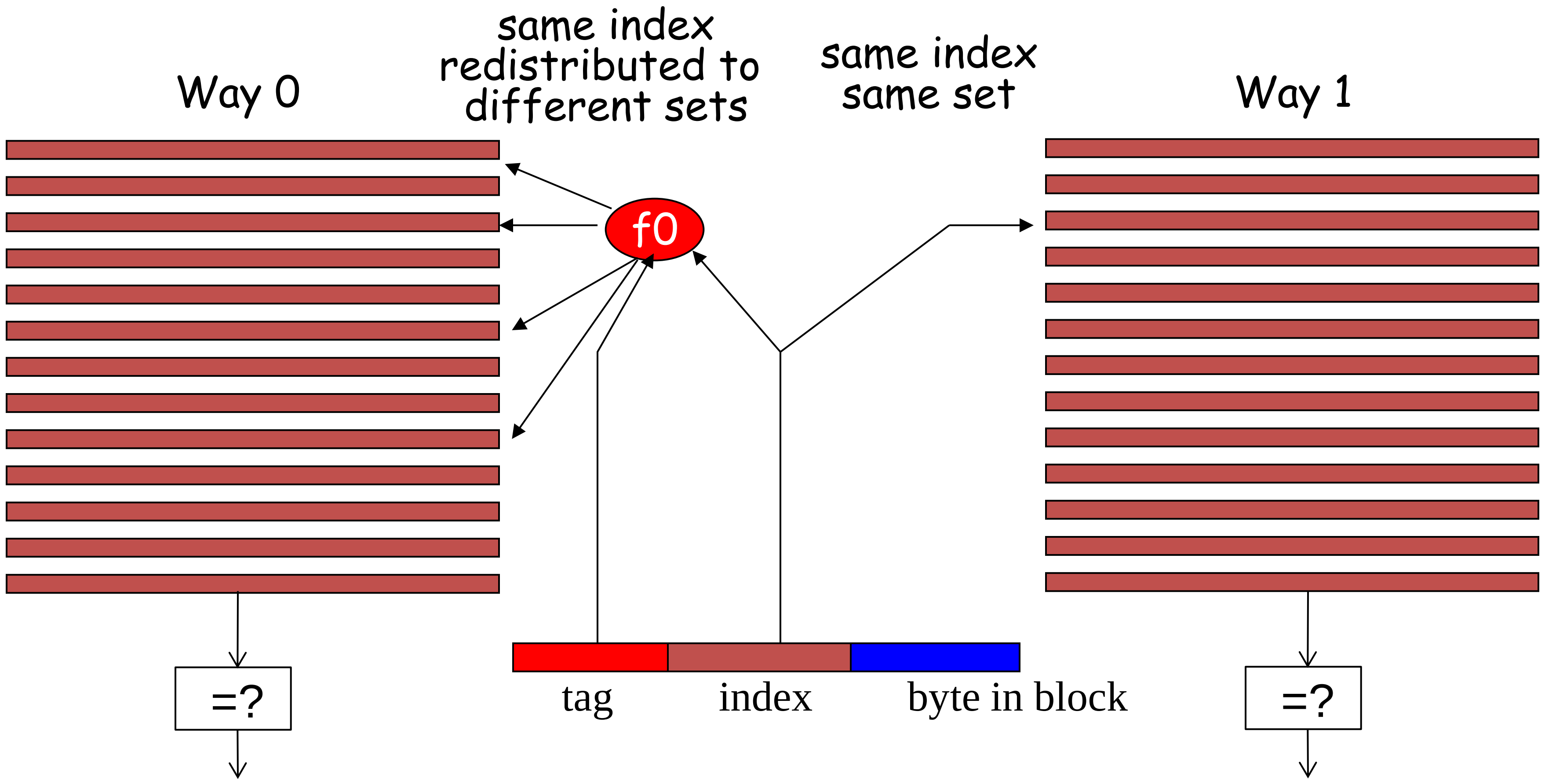
Skewed Associative Caches (I)

- Basic 2-way associative cache structure



Skewed Associative Caches (II)

- Skewed associative caches
  - Each bank has a different index function



Skewed Associative Caches (III)

- Idea: Reduce conflict misses by using different index functions for each cache way

- Benefit: indices are more randomized (memory blocks are better distributed across sets) due to hashing
  - Less likely two blocks have same index
    - Reduced conflict misses

- Cost: additional latency of hash function

Example Software Approaches

- Restructuring data access patterns
- Restructuring data layout
  - Loop interchange
  - Data structure separation/merging
  - Blocking
  - …

Restructuring Data Access Patterns (I)

- Idea: Restructure data layout or access patterns
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called loop interchange
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, …【请看教材】
- What if multiple arrays? Unknown array size at compile time?

Restructuring Data Access Patterns (II)

- Blocking for better cache utilization 【示例见教材】
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache
- But, there are still self-conflicts in a block
  1. there can be conflicts among different arrays
  2. array sizes may be unknown at compile/programming time

Restructuring Data Layout (I)

```
struct Node {
    struct Node* node;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

How to improve?

- Why does the code on the left have poor cache hit rate?
  – "Other fields" occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    struct Node* node;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure
- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

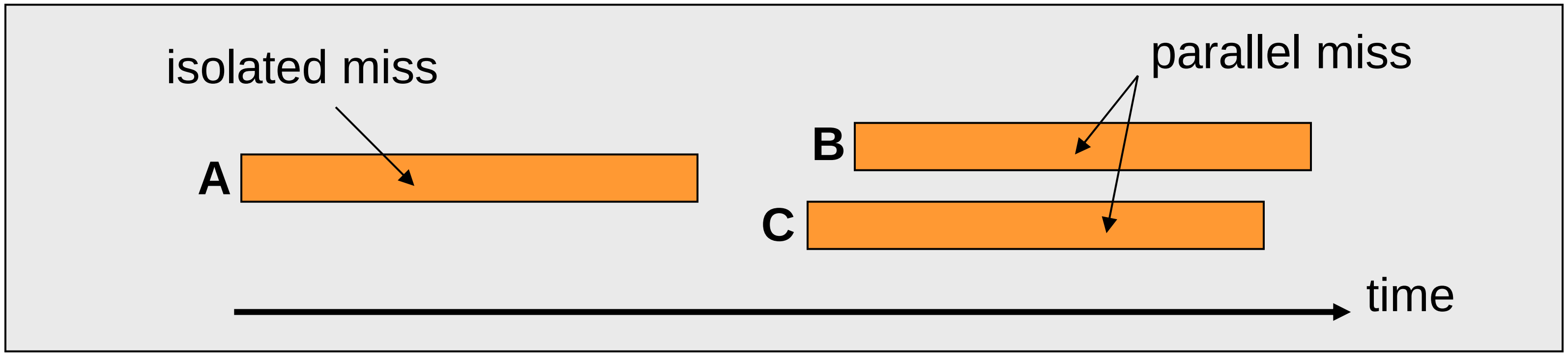
Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies【若感兴趣，请联系我】
  - Non-blocking caches (multiple cache misses in parallel)
  - Issues in multicore caches

Miss Latency/Cost

- What is miss latency or miss cost affected by?
  - Where does the miss get serviced from?
    - Local vs. remote memory
    - What level of cache in the hierarchy?
    - Row hit versus row miss
    - Queueing delays in the memory controller
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed?
    - …

Memory Level Parallelism (MLP)

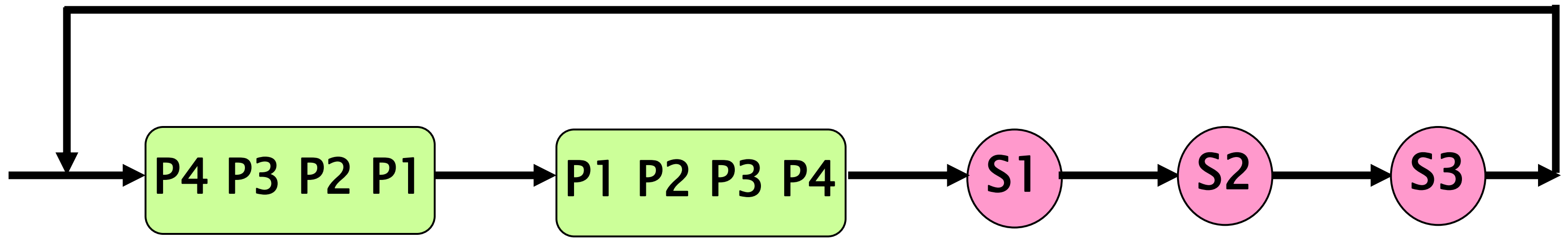


- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- Several techniques to improve MLP (e.g., out-of-order execution)
- MLP varies. Some misses are isolated and some parallel
- How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks this assumption!
  - Eliminating an isolated miss helps performance more than eliminating a parallel miss
  - Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



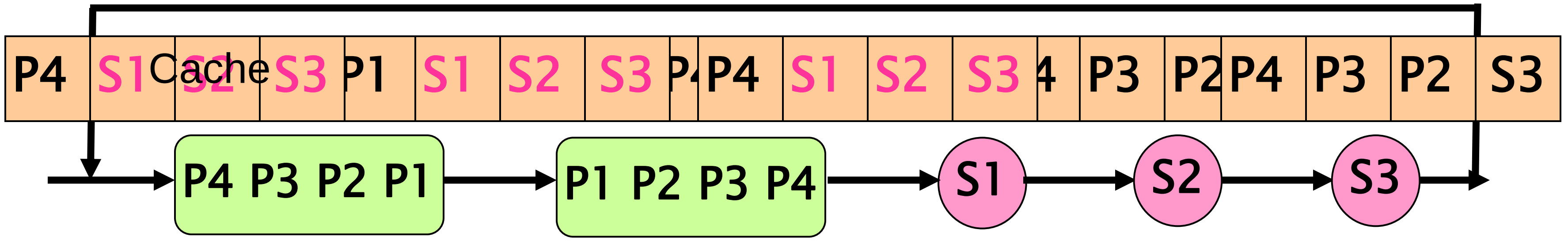
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

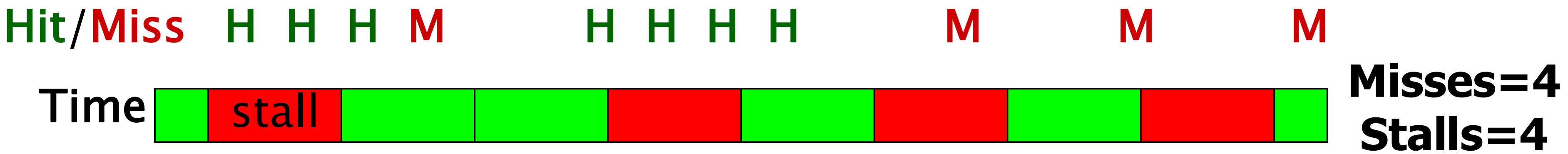
Two replacement algorithms:

1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

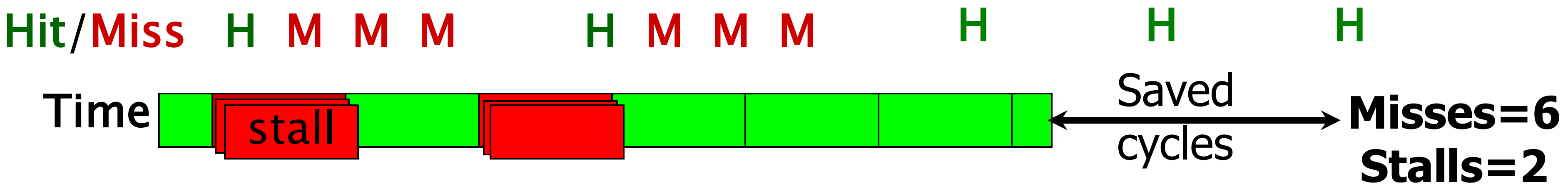
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady's OPT replacement



MLP-Aware replacement

MLP-Aware Cache Replacement

- How do we incorporate MLP into replacement decisions?

- Qureshi et al., “A Case for MLP-Aware Cache Replacement,” ISCA 2006.
  - Suggested reading for homework

【这是一个替换算法优化缓存性能的经典例子】

Enabling Multiple Outstanding Misses

Handling Multiple Outstanding Accesses

- Question: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?
- Goal I: Enable cache access when there is a pending miss
- Goal II: Enable multiple misses in parallel
  - Memory-level parallelism (MLP)
- Solution: Non-blocking or lockup-free caches
  - Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," ISCA 1981.

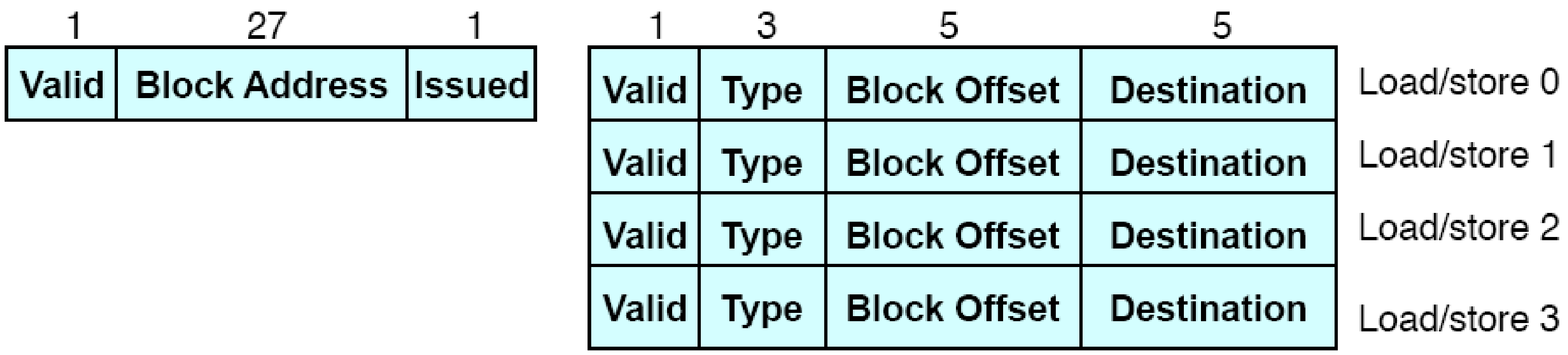
Handling Multiple Outstanding Accesses

- Idea: Keep track of the status/data of misses that are being handled using Miss Status Handling Registers (MSHRs)
  - A cache access checks MSHRs to see if a miss to the same block is already *pending.*
    - If pending, a new request is **not** generated
    - If pending and the needed data available, data forwarded to later load
  - Requires buffering of outstanding miss requests

Miss Status Handling Register

- Also called "miss buffer"
- Keeps track of
  - Outstanding cache misses
  - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR entry
  - Valid bit
  - Cache block address (to match incoming accesses)
  - Control/status bits (prefetch, issued to memory, which sub-blocks have arrived, etc)
  - Data for each sub-block
  - For each pending load/store, keep track of
    - Valid, type, data size, byte in block, destination register or store buffer entry address

MSHR Entry Demo



MSHR Operation

- On a cache miss:
  - Search MSHRs for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall – structure hazard
- When a sub-block returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - Deallocate load/store entry in the MSHR entry
  - Write sub-block in cache or MSHR
  - If last sub-block, deallaocate MSHR (after writing the block in cache)

Non-Blocking Cache Implementation

- When to access the MSHRs?
  - In parallel with the cache?
  - After cache access is complete?

- MSHRs need not be on the critical path of hit requests
  - Which one below is the common case?
    - Cache miss, MSHR hit
    - Cache hit

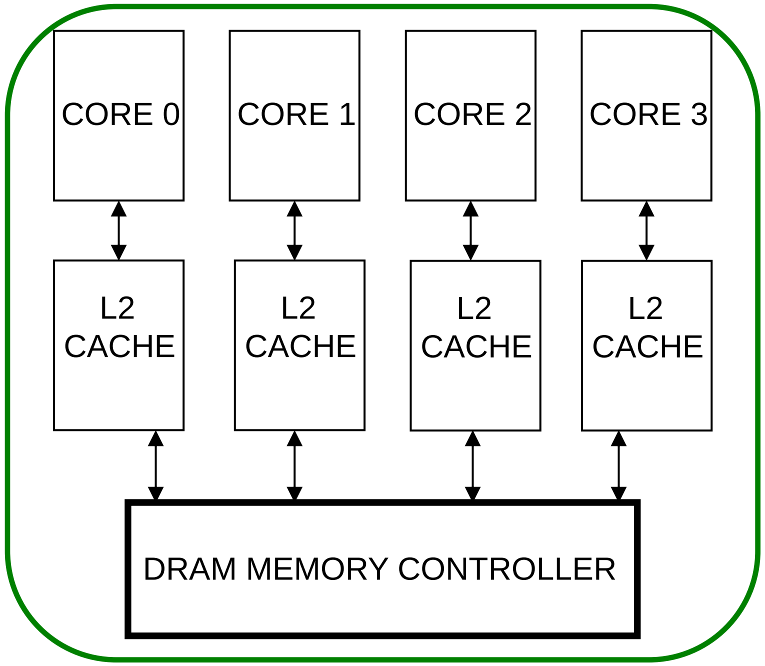
Multi-Core Issues in Caching

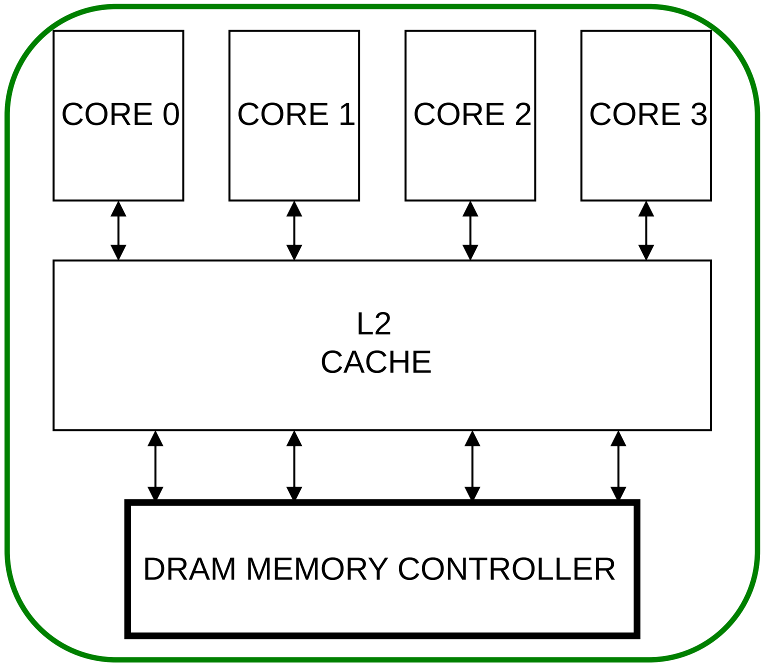
Caches in Multi-Core Systems

- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource
- How do we design the caches in a multi-core system?
- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Resource Sharing and Its Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory
- Why?

  + Resource sharing improves utilization/efficiency → throughput
    - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data

  + Reduces communication latency
    - For example, shared data kept in the same cache in multithreaded processors

  + Compatible with the **shared memory model**

Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it
- Sometimes reduces each or some thread’s performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → **inconsistent performance across runs【实时系统】**
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

- Need to efficiently and fairly utilize shared resources
- This is crucial for modern data center

Shared Caches Between Cores

- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence (single copy)
  - Shared data and locks do not ping pong between caches
- Disadvantages
  - Slower access
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Shared Caches: How to Share?

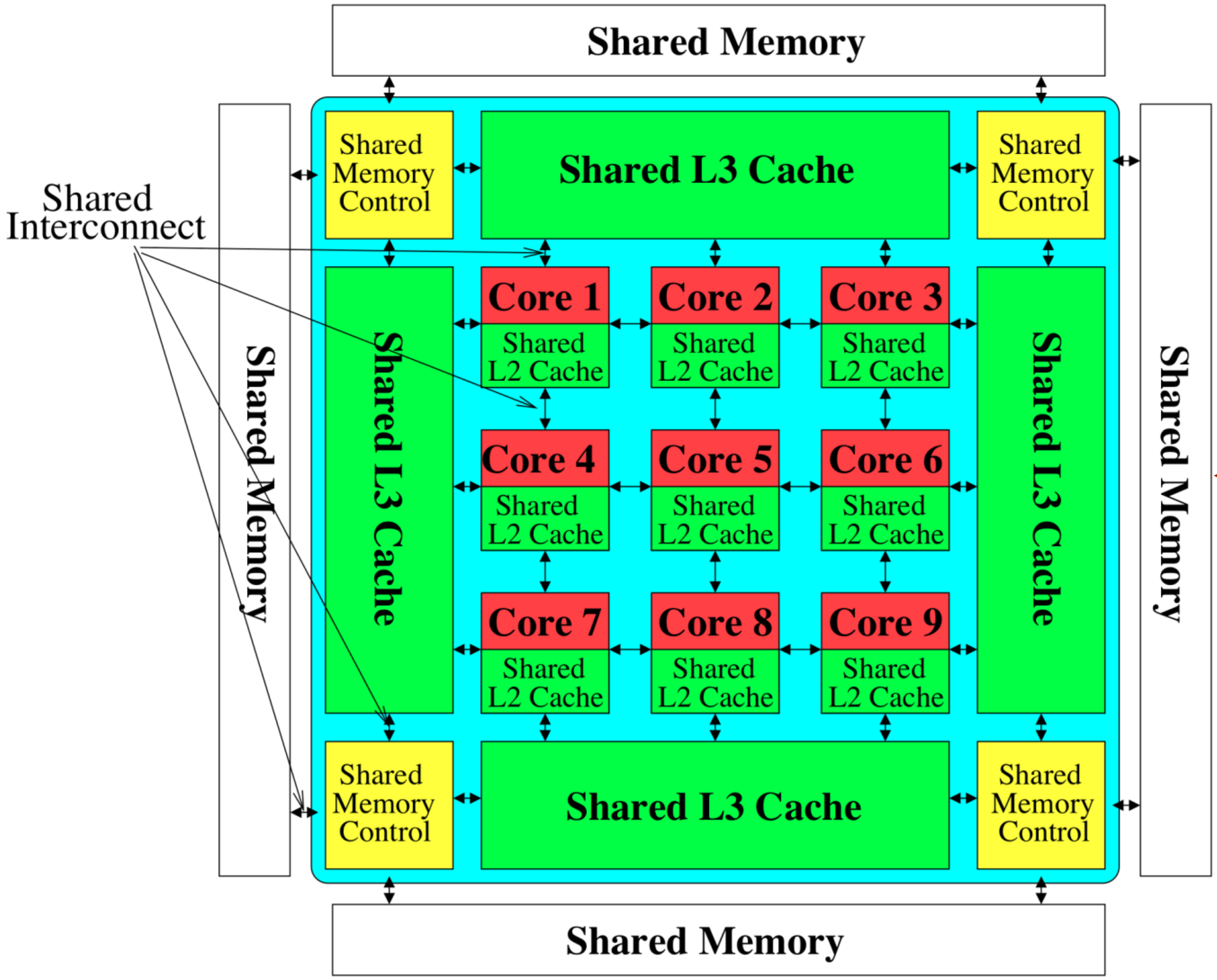
- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to
- Problems
  - Inefficient utilization of cache: LRU is not the best policy
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness

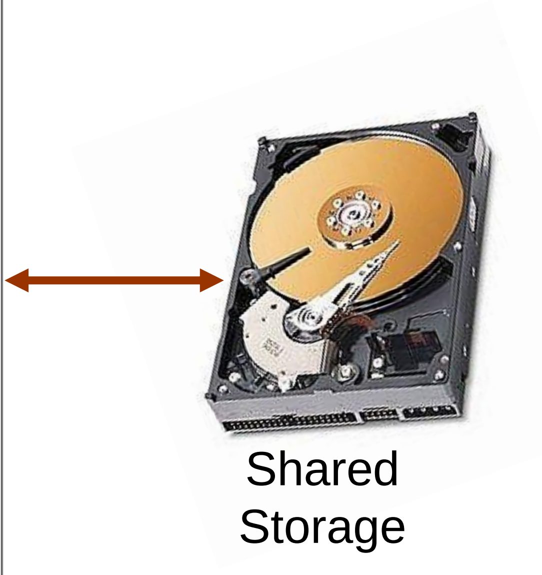
Optimization: Utility Based Cache Partitioning

- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space
- The high-level idea can be applied to other shared resources as well.

- Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
- Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

The Multi-Core System: A *Shared Resource* View





Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?
- Makes programmer's life difficult
  - An optimized program can get low performance (and performance varies widely depending on co-runners)
- Causes discomfort to user
  - An important program can starve
  - Examples from shared software resources
- Makes system management difficult
  - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?

Resource Sharing vs. Partitioning

- Sharing improves throughput
  - Better utilization of space
- Partitioning provides performance isolation (predictable performance)
  - Dedicated space

依然是开放问题!

- Can we get the benefits of both?
- Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
  - No wasted resource + QoS mechanisms for threads

Shared Hardware Resources

- Memory subsystem (in both multithreaded and multi-core systems)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks
- I/O subsystem (in both multithreaded and multi-core systems)
  - I/O, DMA controllers
  - Ethernet controllers
- Processor (**in multithreaded systems**)
  - Pipeline resources
  - L1 caches

Next Topic

Prefetching