计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

review: Cache性能分析

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

Figure B.7 Summary of performance equations in this appendix. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

review：基本优化方法

- **基本Cache优化方法**

降低失效率： 引起失效的3C

1、增加Cache块的大小

2、增大Cache容量

3、提高相联度

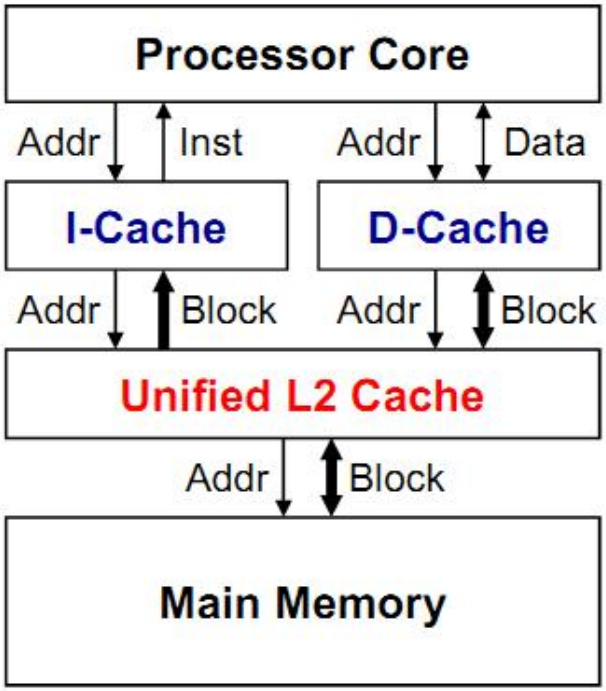
减少失效开销

4、多级Cache

5、使读失效优先于写失效

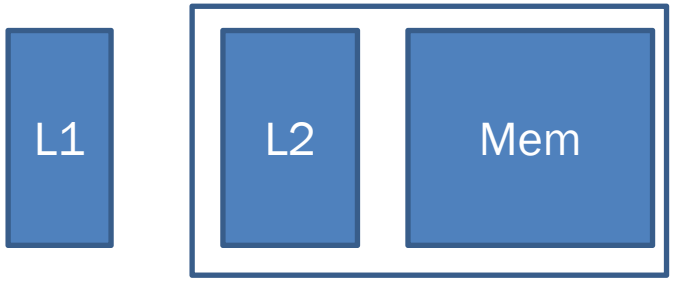
缩短命中时间

6、避免在索引缓存期间进行地址转换



多级cache的性能分析

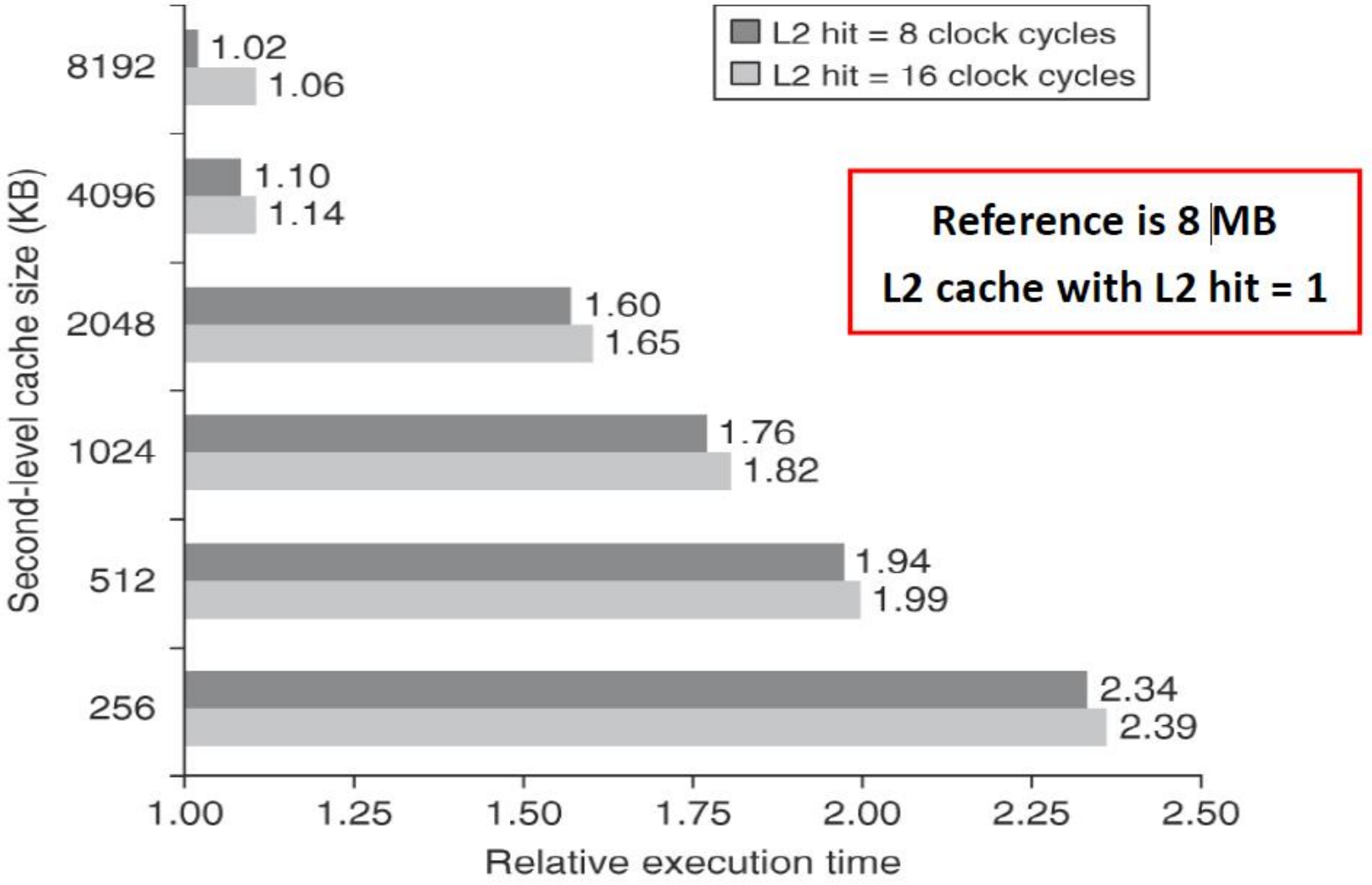
- **局部失效率：该级Cache失效次数 / 到达该级Cache的访存次数**
  - Miss rateL1 for L1 cache
  - Miss rateL2 for L2 cache
- **全局失效率：该级Cache失效次数/ CPU发出的访存总次数**
  - Miss rateL1 for L1 cache
  - Miss rateL1 × Miss rateL2 for L2 cache
  - 全局失效率是度量L2 cache性能的更好方法
- **性能参数**
  - AMAT = Hit TimeL1+Miss rateL1×Miss penaltyL1
  - Miss penaltyL1 = HitTimeL2 + Miss rateL2×Miss penaltyL2
  - AMAT = Hit TimeL1+Miss rateL1× (Hit TimeL2+Miss rateL2×Miss penaltyL2)



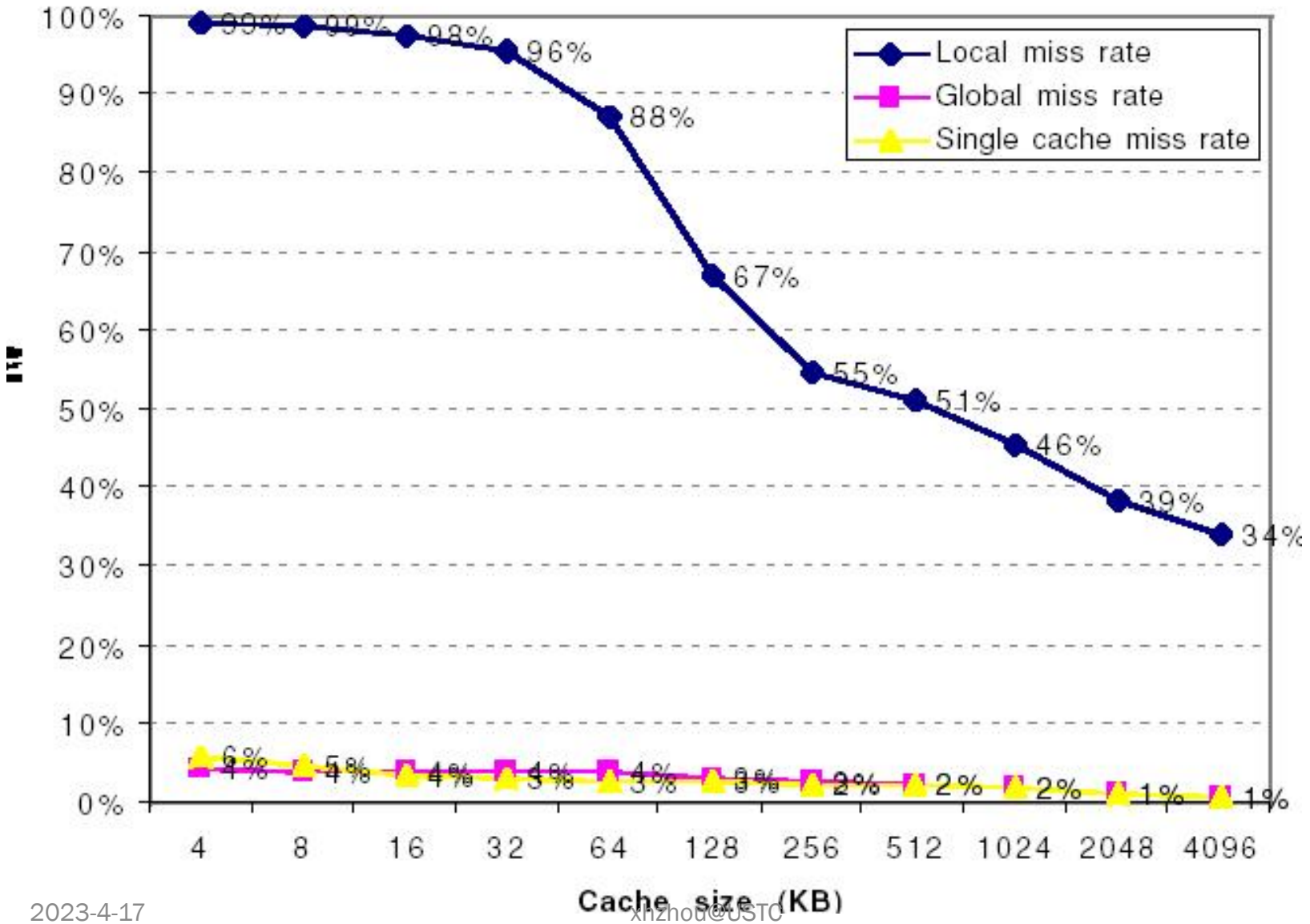
两级Cache的性能

- **Problem: 程序运行产生1000个存储器访问**
  - **失效次数**：I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
  - **命中时间和失效开销**：L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
  - **访存指令比例及理想CPI**：Load + Store frequency = 25%, CPIexecution = 1.1 (perfect cache)
  - 计算memory stall cycles per instruction 和有效的CPI
  - 如果没有L2 cache, 有效的CPI是多少?
- **Solution:**
  - L1 Miss Rate = (5 + 35) / 1000 = 0.04 (or 4% per access)
  - L1 misses per Instruction =0.04 × (1 + 0.25) = 0.05
  - L2 misses per Instruction =(8 / 1000) × (1+0.25) = 0.01
  - Memory stall cycles per Instruction =0.05 × 8 + 0.01 × 80 = 1.2
  - CPI_{L1+L2} = 1.1 + 1.2 = 2.3, CPI/CPIexecution = 2.3/1.1 = 2.1x slower
  - CPI_{L1only} =1.1 + 0.05 × 80 = 5.1 (worse)

不同大小的L2cache对应的执行时间



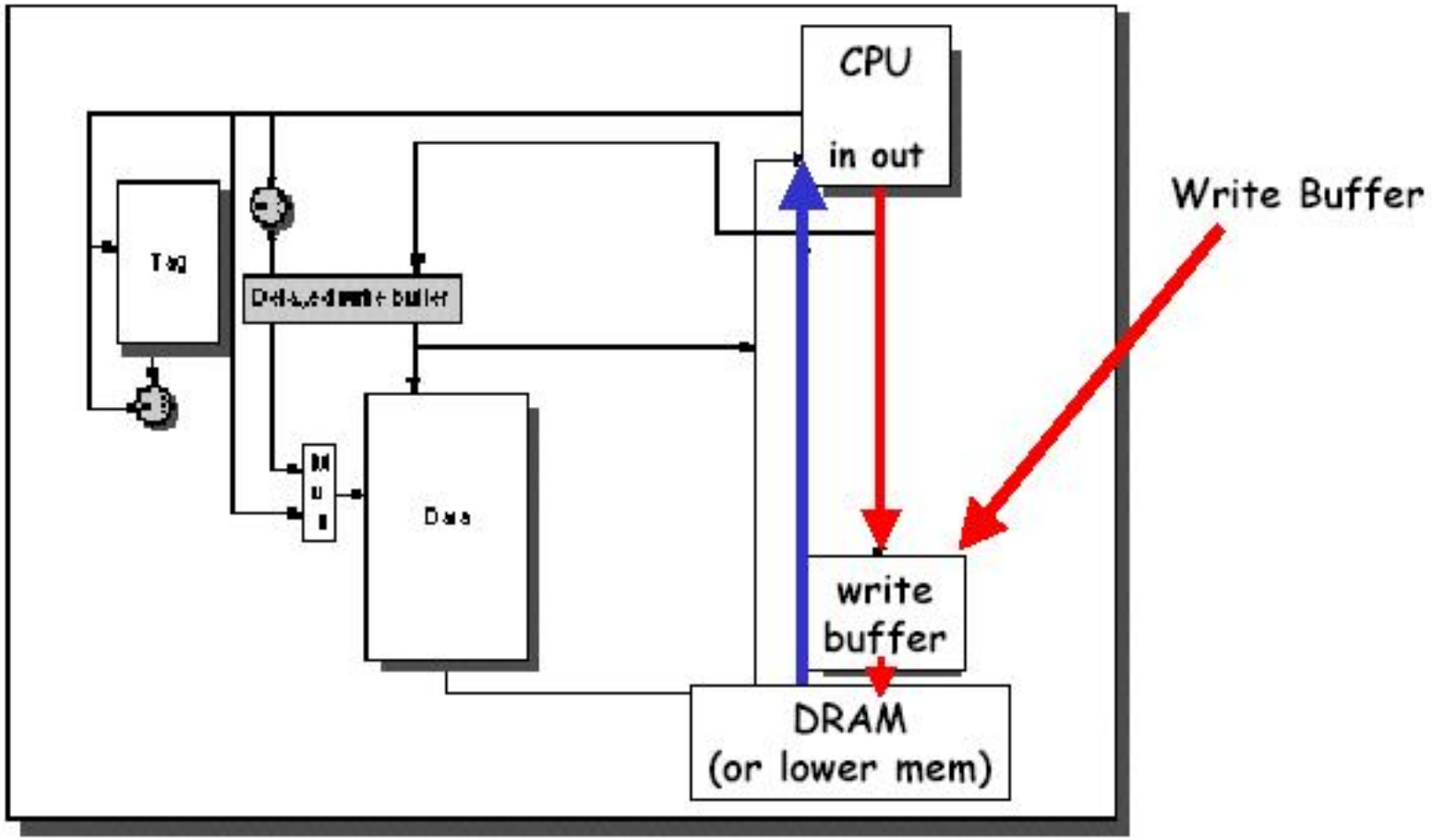
Miss rates versus cache size for multilevel caches



两级Cache的一些研究结论

- **在L2比L1大得多的情况下，两级Cache全局失效率 和 容量与第二级Cache相同的单级Cache的失效率接近**
- **局部失效率不是衡量第二级Cache的好指标**
  - 它是第一级Cache失效率的函数
  - 不能全面反映两级Cache体系的性能
- **第二级Cache设计需考虑的问题**
  - 容量：一般较大，可能无容量失效，有强制性失效和冲突失效
    - 相联度对第二级Cache的作用
    - Cache可以较大，以减少失效次数
  - 多级包容性问题：第一级Cache中的数据是否总是同时存在于第二级Cache中。
    - 如果L1和L2的块大小不同，增加了多级包容性实现的复杂性

让读优先于写图示



让读失效优先于写

- **由于读操作为大概率事件，需要读失效优先，以提高性能**
- **Write-Through Cache ->Write Buffer (写缓冲)，特别对写直达法更有效**
  - Write Buffer: **CPU不必等待写操作完成，即将要写的数据和地址送到Write Buffer后，CPU继续作其他操作。**
  - 写缓冲导致对存储器访问的复杂化
    - **在读失效时写缓冲中可能保存有所读单元的最新值，还没有写回**
    - 例如，直接映射、写直达、512和1024映射到同一块。则
    - SW R3, 512(R0)　　//命中，直接修改Cache块，并将新的值 写入write Buffer
    - LW R1, 1024(R0)　　//失效，将512(R0)对应块换成1024(R0)单元所在块
    - LW R2, 512(R0)　　//失效，从主存中调入512(R0)单元所在块
  - 解决办法
    - 推迟对读失效的处理，直到写缓冲器清空，导致新的问题——读失效开销增大。
    - **在读失效时，检查写缓冲的内容，如果没有冲突，而且存储器可访问，就继续处理读失效**
  - 写回法时，也可以利用写缓冲器来提高性能
    - 把脏块放入缓冲区，然后读存储器，最后写存储器
- **Write-Back Cache ->Victim Buffer**
  - 被替换的脏块放到了victim buffer
  - 问题：在脏块被写回前，需要处理读失效。 victim buffer 可能含有该读失效要读取的块
    - Solution: 查找victim buffer，如果命中直接将该块调入Cache

4.2 Cache 基本优化方法

Cache 性能分析 → 降低失效率 → 减少失效开销 → 缩短命中时间

平均访存时间 = 命中时间 + 失效率×失效开销

缩短命中时间

虚拟地址Cache VS. 物理地址Cache

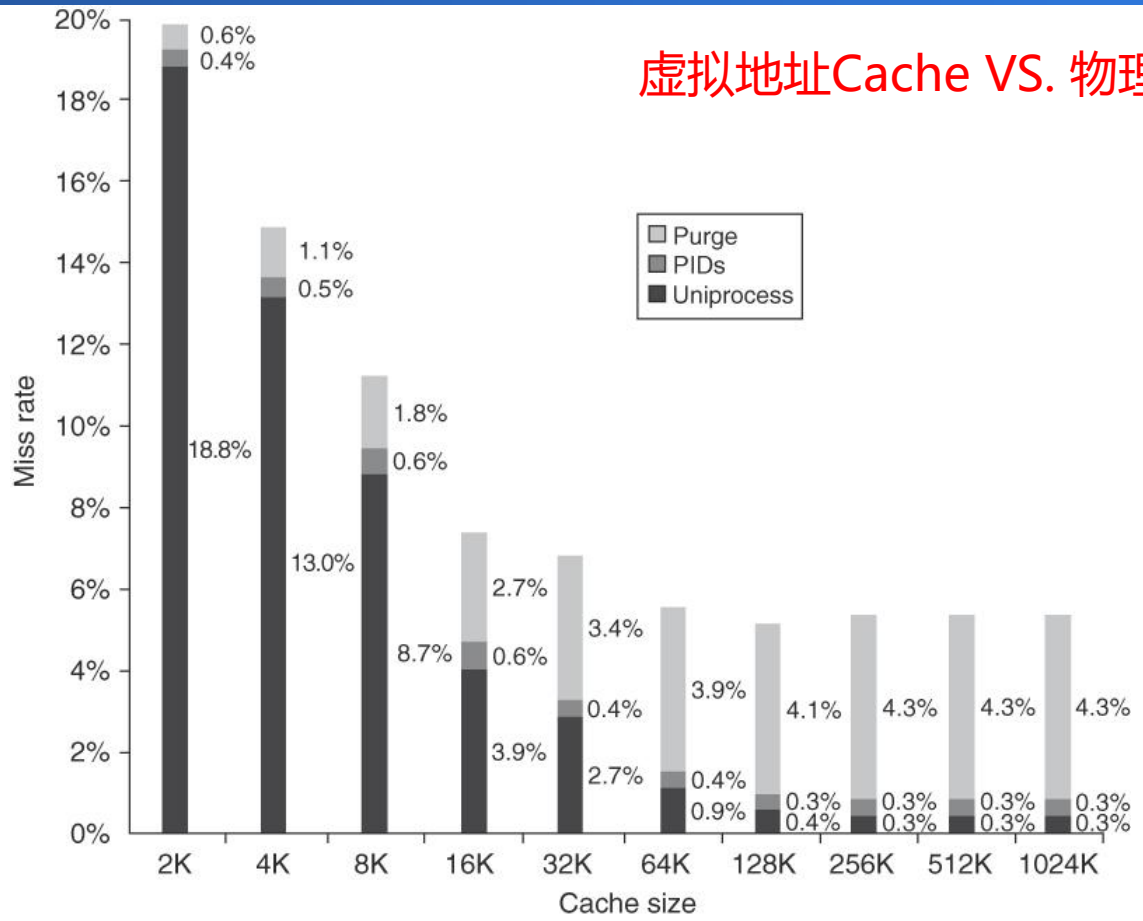
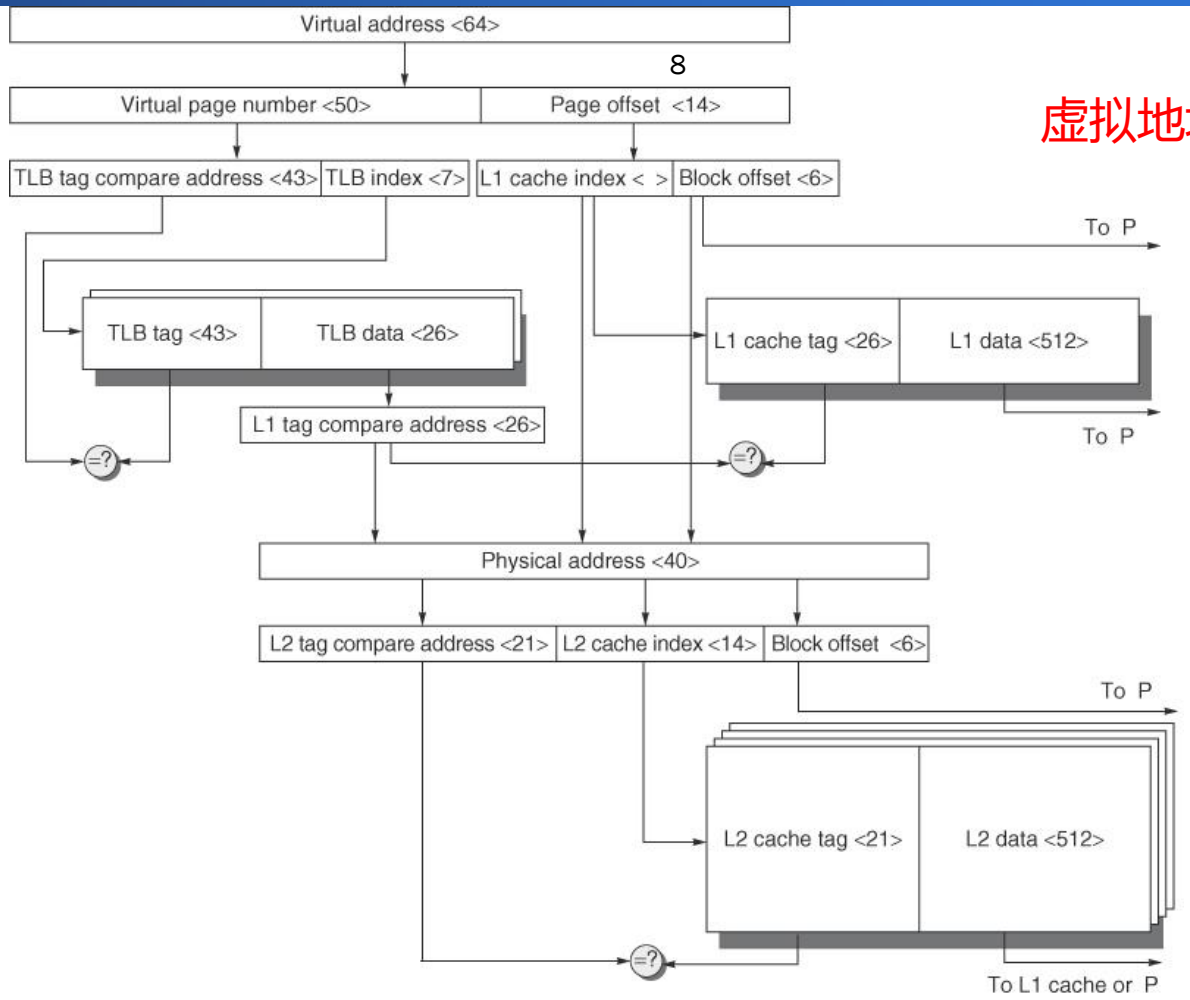


Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.



虚拟地址转换与Cache定位 并行

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KB, and the L2 cache is a four-way set associative with a total of 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

4.1 Cache的基本概念

存储系统的层次结构

Cache基本知识

4.2 Cache的基本优化方法

4.3 Cache的高级优化方法

4.4 存储器技术与优化

4.5 虚拟存储器 - 基本原理

4.3 Cache的高级优化方法

- **缩短命中时间**
- **增加Cache带宽**
- **减小失效开销**
- **降低失效率**
- **通过并行降低失效开销或失效率**

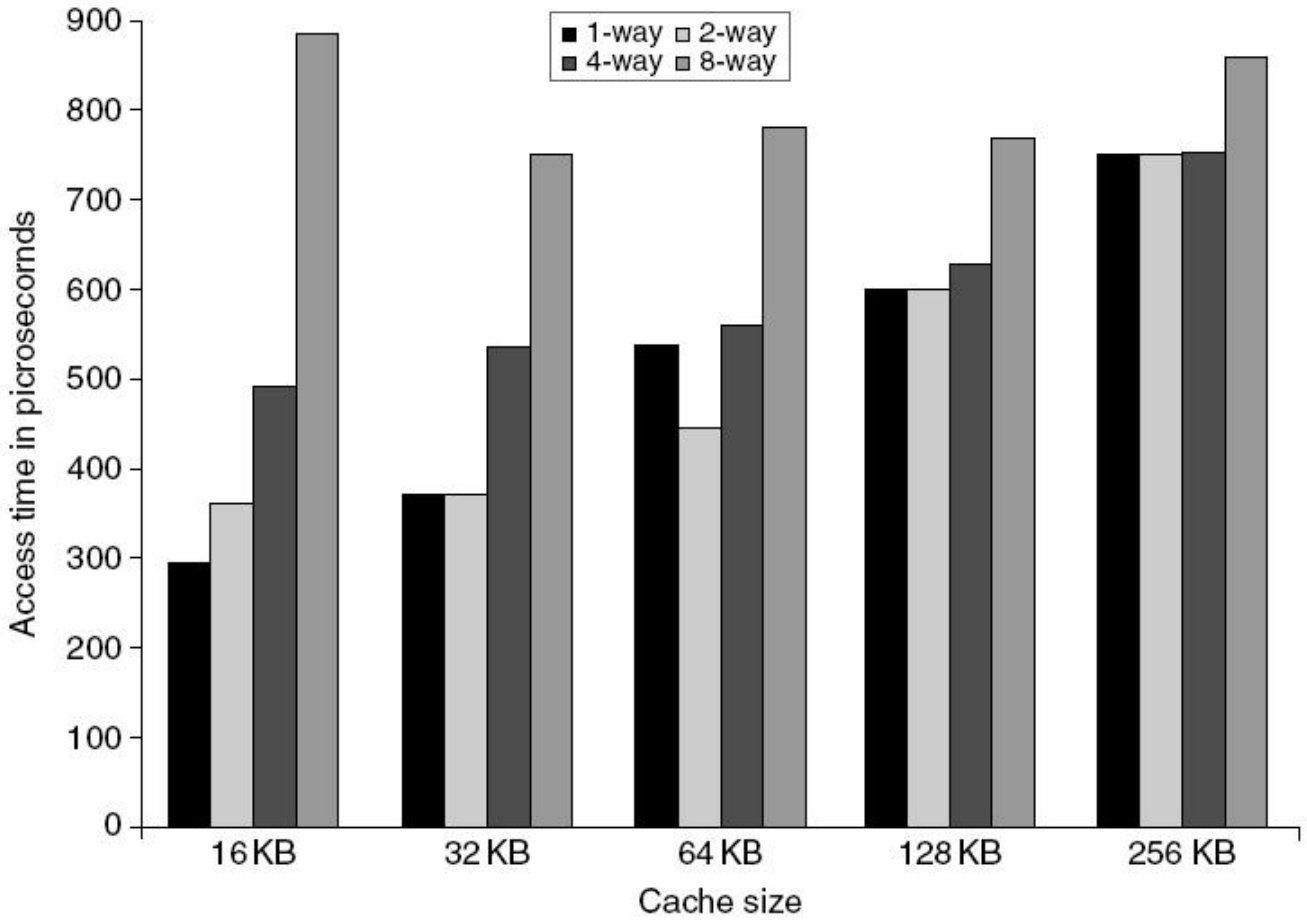
高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

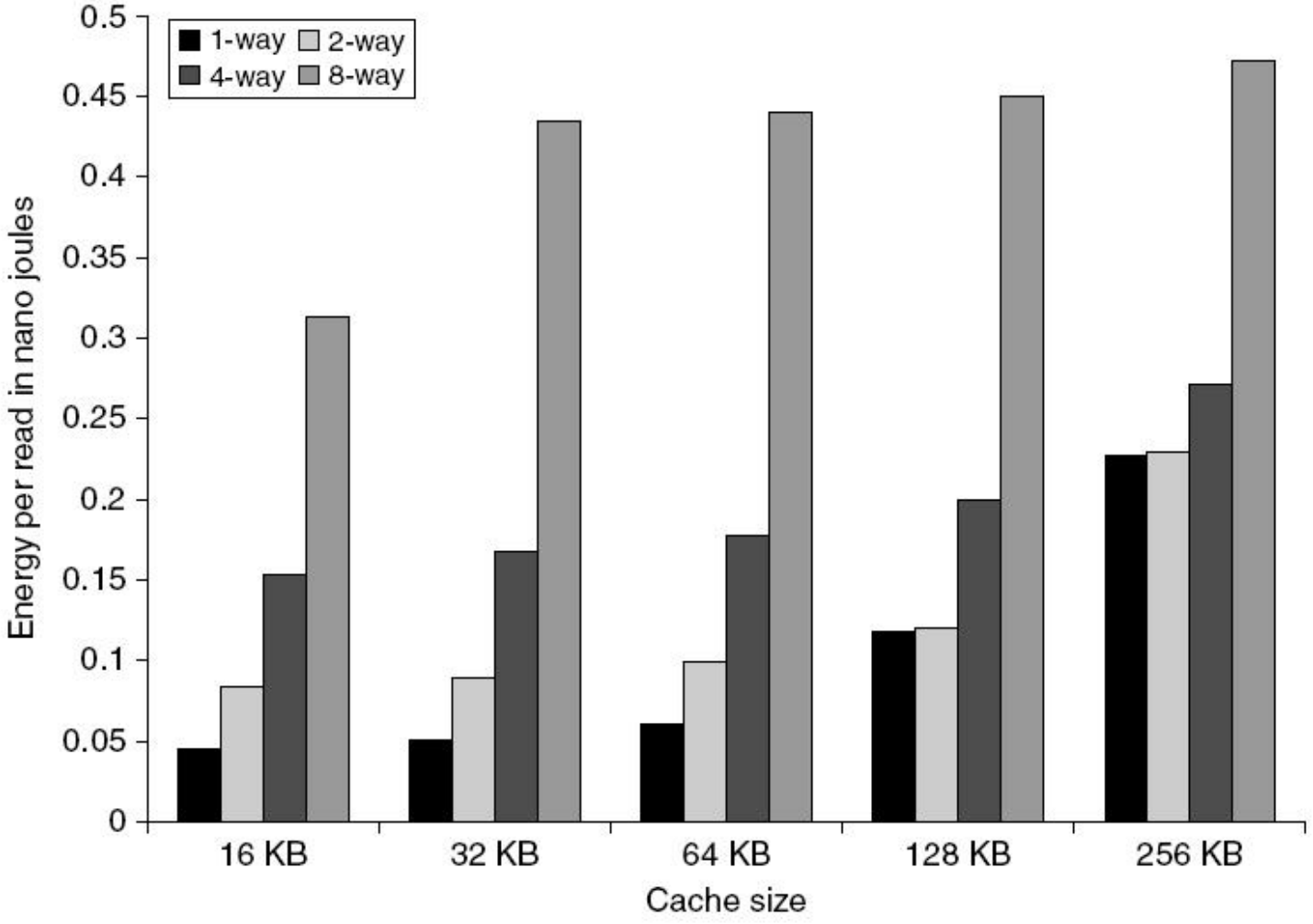
1、Small and simple first level caches

- **Small and simple first level caches**
  - 容量小，一般命中时间短，有可能做在片内
  - 另一方案，保持Tag在片内，块数据在片外，如DEC Alpha
  - 第一级Cache应选择容量小且结构简单的设计方案
- **Critical timing path:**
  - 1) 定位组, 确定tag的位置
  - 2) 比较tags,
  - 3) 选择正确的块
- **Direct-mapped caches can overlap tag compare and transmission of data**
  - 数据传输和tag 比较并行
- **Lower associativity reduces power because fewer cache lines are accessed**
  - 简单的Cache结构、可有效减少tag比较的次数，进而降低功耗

L1 Size and Associativity



L1 Size and Associativity



2、Way Prediction

- **为改进命中时间，预测被选中的路(way)**
  - 预测错误会导致更长的命中时间
  - 预测的准确性
    - 90%+ for two-way
    - 80%+ for four-way
    - I-cache比D-cache具有更好的准确性
  - 90年代中期第一次用于MIPS R10000
  - 用于ARM Cortex-A8
- **Way Prediction的扩展方法也可降低功耗：将直接映像方式查找与Way Prediction选择结合**
  - 也称“路选择”“Way selection”
  - 可有效降低功耗，但一旦预测错误会有更长的命中时间

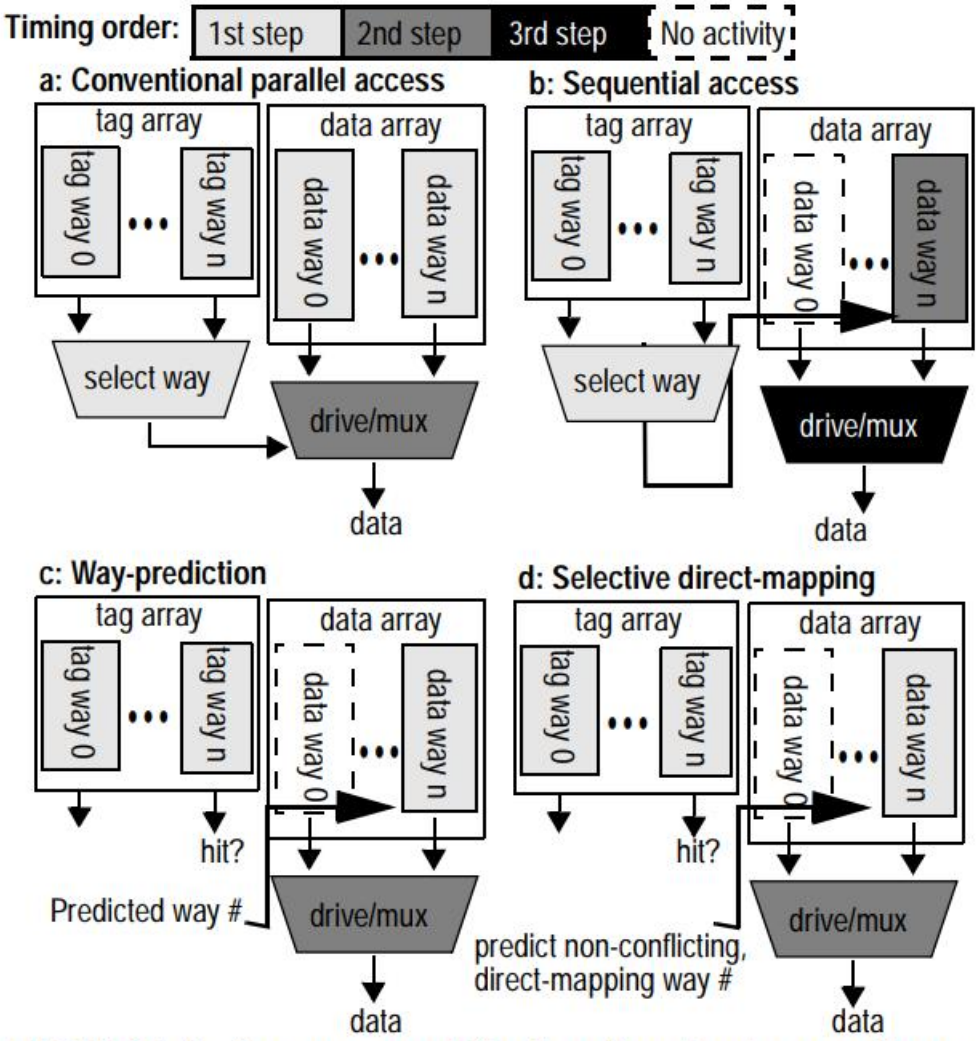


FIGURE 1: Access and timing for design options.

4.3 Cache的高级优化方法

- **缩短命中时间**
- **增加Cache带宽**
- **减小失效开销**
- **降低失效率**
- **通过并行降低失效开销或失效率**

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

3、Pipelining Cache

- **实现Cache访问的流水化**
  - 提高Cache的带宽，有利于采用高相联度的缓存
  - L1 cache的访问由多个时钟周期构成
    - Pentium：1 cycle
    - Pentium Pro – Pentium III：2 cycles
    - Pentium 4 – Core i7：4 cycles
    - IBM Power7：3 cycles
- **缺点：增加流水线的段数**
  - 增加了分支预测错误造成的额外开销
  - 增加了Load指令与要使用其结果的指令间的latency
  - 增加了I-Cache和D-Cache的延时

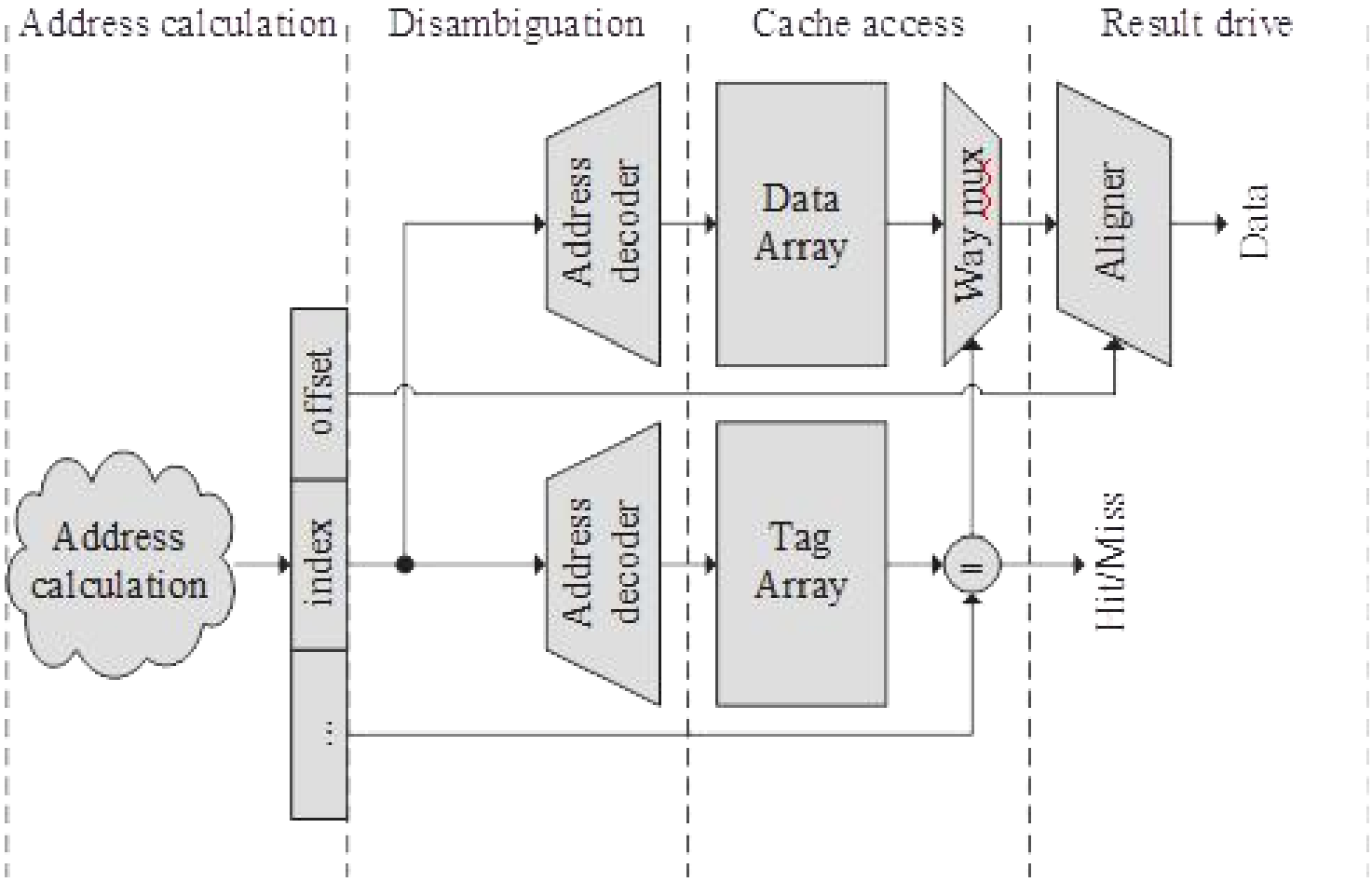


FIGURE 2.2: Parallel tag and data array access pipeline.

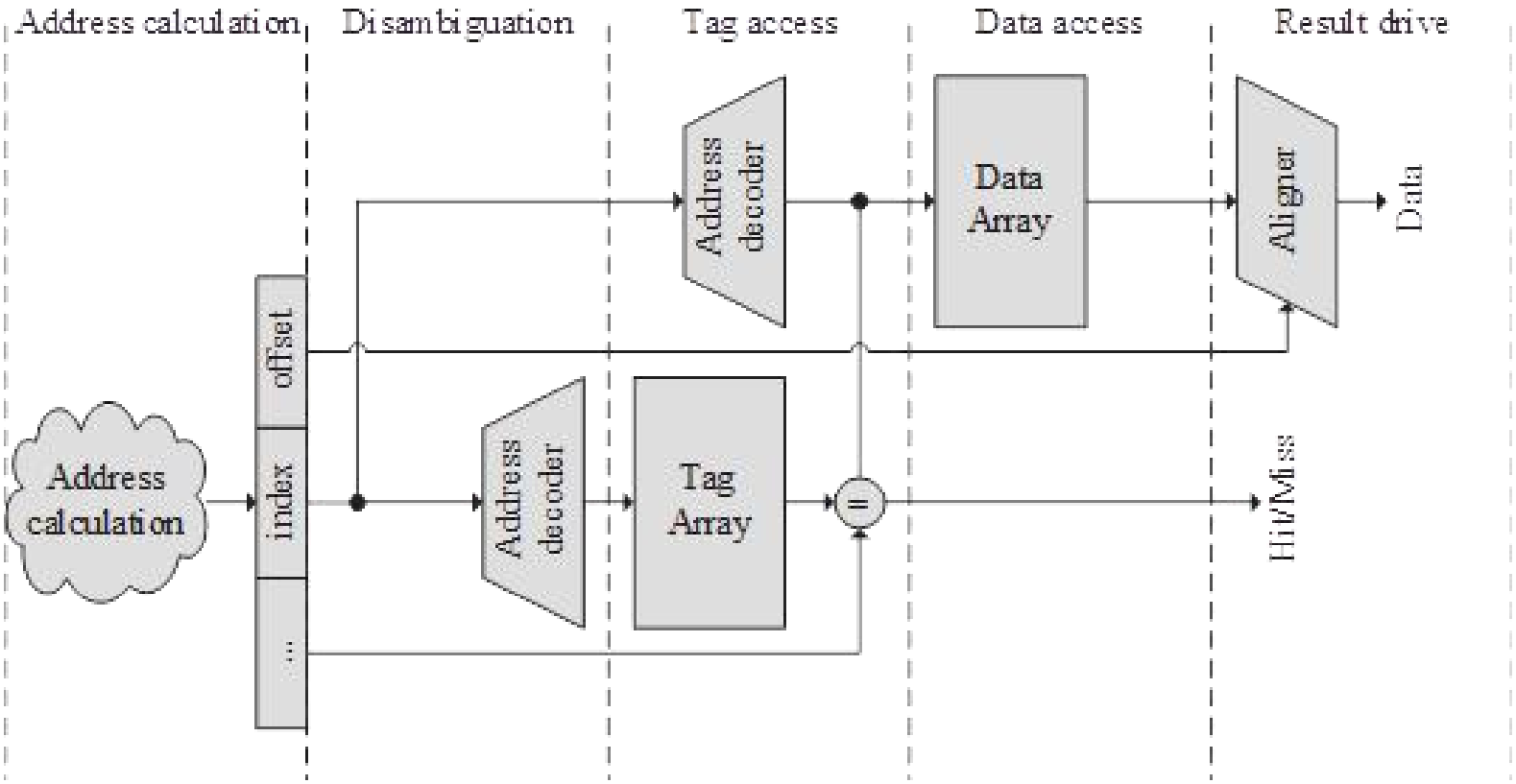
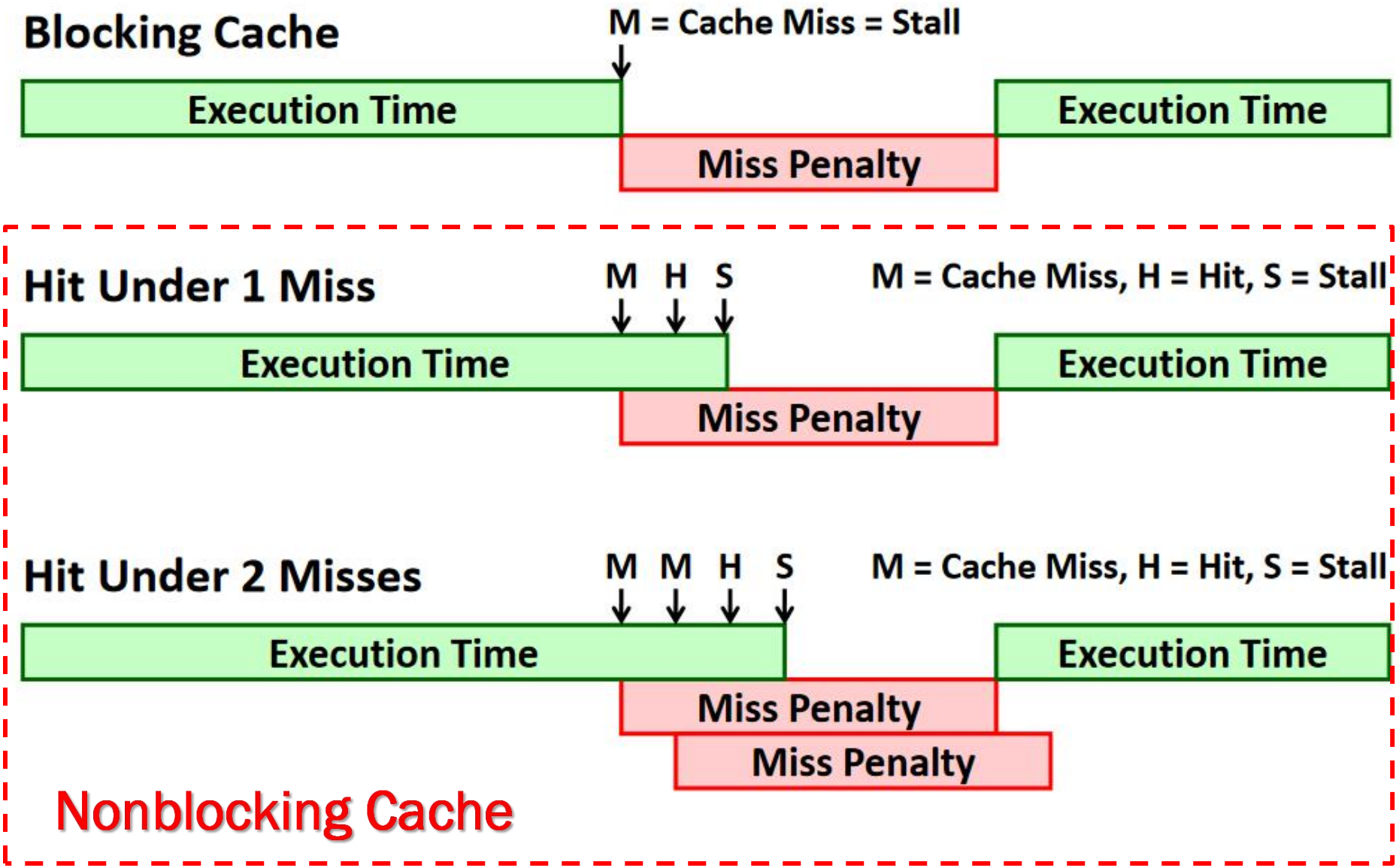


FIGURE 2.4: Serial tag and data array access pipeline.

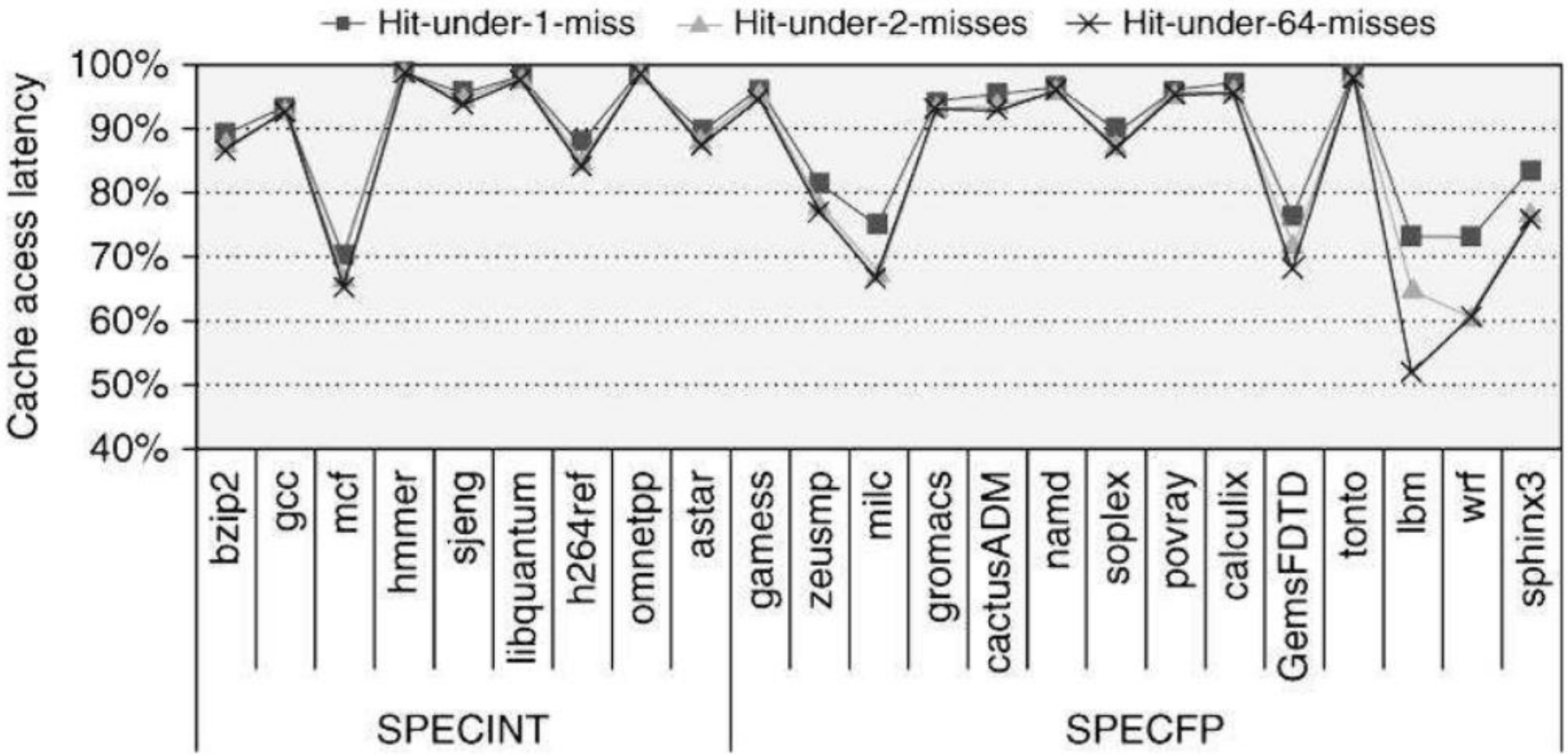
4、Nonblocking Caches

- **允许在Cache失效下继续命中**
  - 在Cache失效时，CPU无需stall
  - 主要用于乱序执行和多线程处理器
- **Hit under a Miss**
  - 减少有效的失效开销
  - 增加Cache的带宽
- **Hit under Multiple Misses**
  - 针对多个未解决的Cache失效
  - 可能会更多地减少有效的失效开销
  - 增加了Cache控制器的复杂性
  - 存储系统可以支持多个失效时的存储服务

Nonblocking Cache Timeline



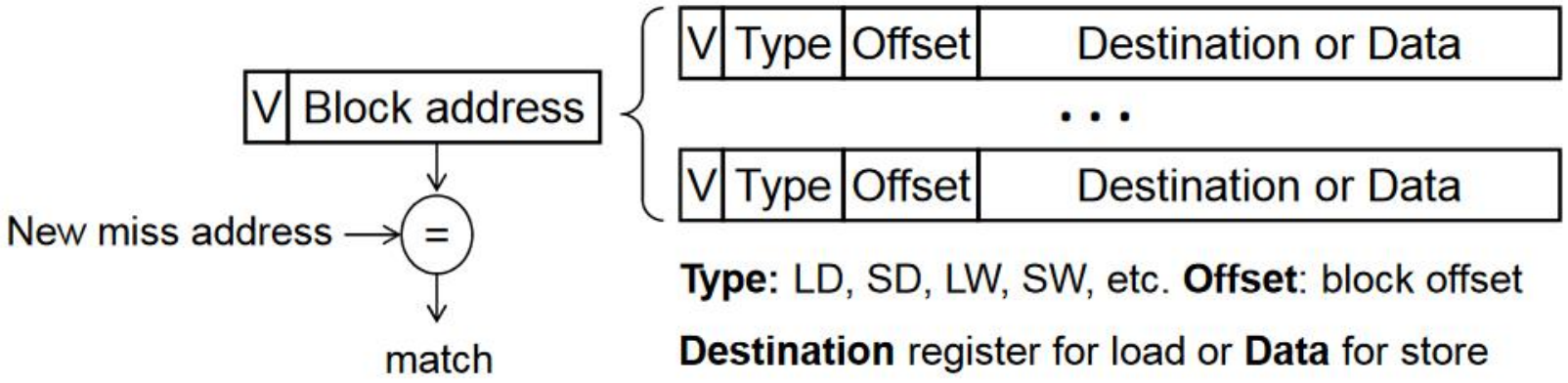
Effectiveness of Non-Blocking Cache



Reduces the miss penalty by not stalling the processor on a cache miss

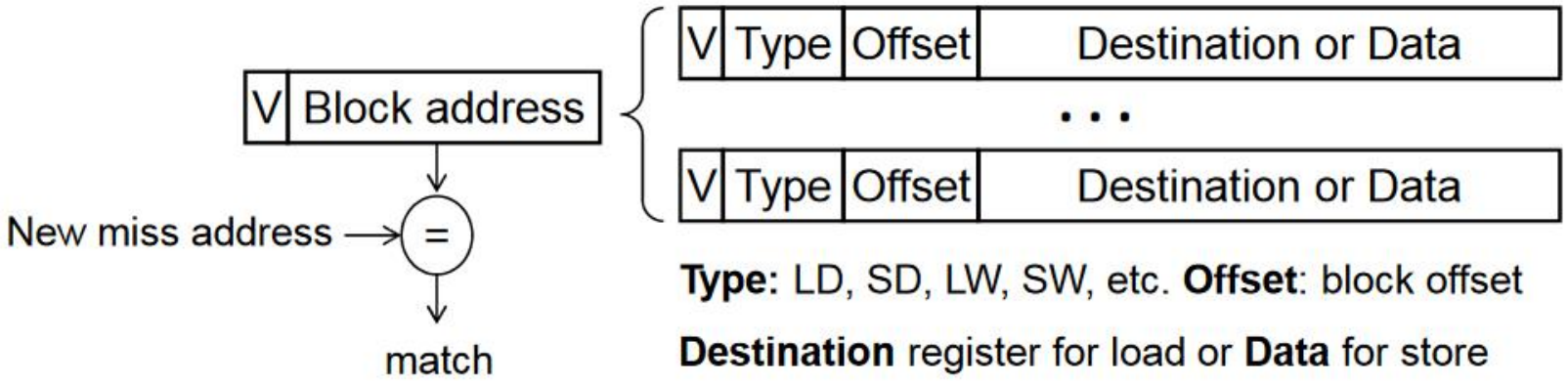
Miss Status Holding Register (MSHR)

- **MSHR 包含正在等待处理的失效**
  - 相同的块可以包含多个未解决的Load/Store 失效
  - 可以有多个未解决的块地址
- **失效可以分为**
  - Primary: 第一次发起存取请求时的失效块
  - Secondary: 在后续过程中的失效
  - Structural Stall miss： MSHR 硬件资源耗尽



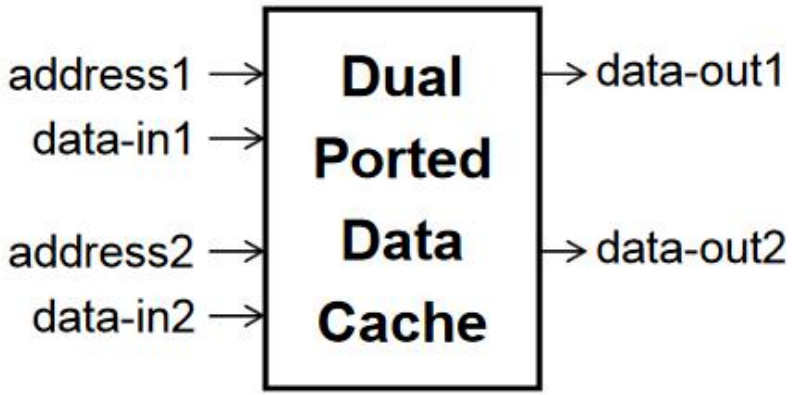
NonBlocking Cache Operation

- **当Cache失效时，检查MSHR(Miss-status Handling Registers)是否有匹配的块地址**
  - 如果有，为该地址分配新的load/store 表项
  - 如果没有，分配新的MSHR 和 load/store表项
  - 如果所有MSHR资源都分配完，则Stall（结构相关）
- **当从底层传输Cache块时**
  - 处理该块中的Load和Store指令引起的失效
  - Load：根据block offset从该块中装载数据到寄存器
  - Store：根据block offset将数据写入该块指定位置
  - 完成该块所有的失效的Load/Store后，释放MSHR中的对应表项



5、Multibanked Caches（1/2）

- **Multi-Ported Cache**
  - True Multi-ported Cache Design
    - 所有的控制和数据通路在Cache中是多份的
      - Address Decoder, way multiplexor, tag comparator, aligners
      - Tag Array, Data Array
    - 显著地增加了Cache的面积和访问时间
    - 没有商业处理器采用这种设计
  - Multi-Banked Cache
    - 将cache组织成多个banks
    - 每个bank是一个端口
    - 可以并行访问不同的Bank



5、Multibanked Caches （2/2）

- **将Cache组织为多个独立的banks，以支持并行访问**
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
- **根据块号进行顺序多体交叉编址**

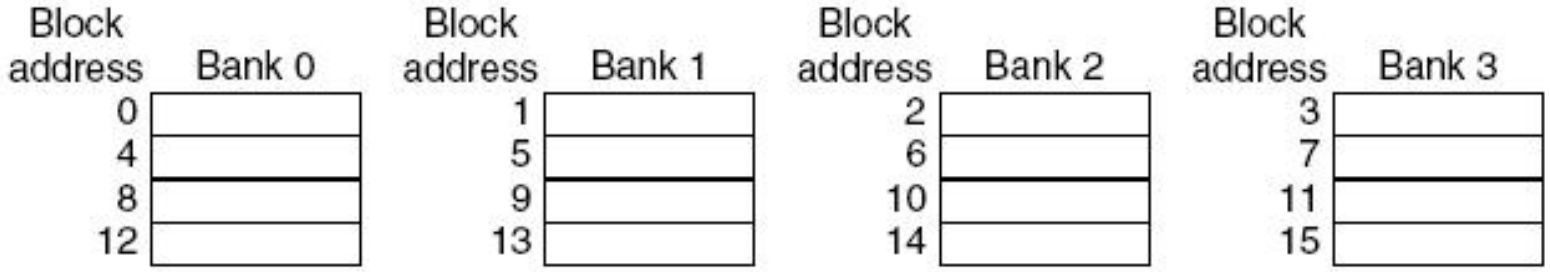


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

review：高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

4.3 Cache的高级优化方法

- **缩短命中时间**
- **增加Cache带宽**
- **减小失效开销**
- **降低失效率**
- **通过并行降低失效开销或失效率**

高级Cache优化方法

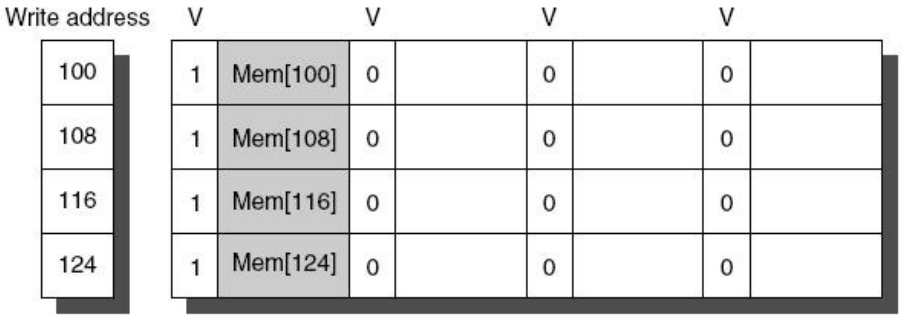
- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

6、Critical Word First, Early Restart

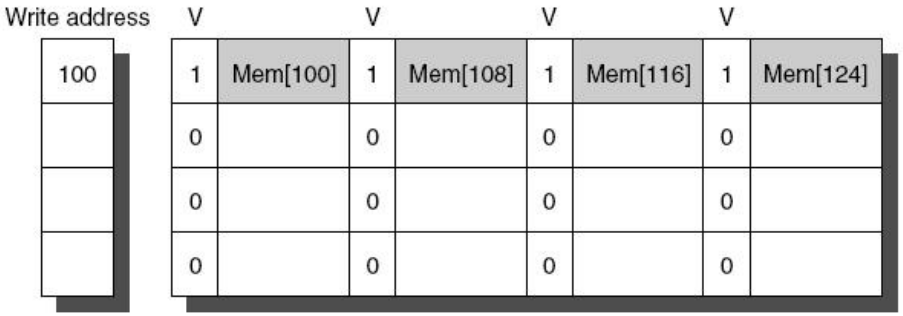
- **关键字优先**
  - 首先请求CPU所需要的字
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **提前重启**
  - 请求字的顺序不变
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **通常在块比较大时，这些技术才有效**

7、Merging Write Buffer

- 在向写缓冲器写入地址和数据时，如果写缓冲器中存在被修改过的块，就检查其地址，看看本次写入数据的地址是否与写缓冲器内的某个有效块地址匹配，如果匹配，就把新数据与该块合并，称为“合并写”
- 可以缓解由于写缓冲满而造成的CPU停顿
- 不适用于I/O地址空间？？



No write buffering



Write buffering

4.3 Cache的高级优化方法

- **缩短命中时间**
- **增加Cache带宽**
- **减小失效开销**
- **降低失效率**
- **通过并行降低失效开销或失效率**

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

8、编译器优化

- **无需对硬件做任何改动，通过软件优化降低失效率**
- **研究从两方面展开：减少指令失效 和 减少数据失效**
- **减少指令失效，重新组织程序（指令调度）而不影响程序的正确性**
  - 研究结果：通过使用profiling信息来判断指令组间可能发生的冲突，并将指令重新排序以减少失效。
  - 研究表明：
    - 容量为2KB, 块大小为 4Bytes的直接映象Icache，通过使用指令调度可以使失效率降低50%。容量增大到 8KB, 失效率可降低75%
    - 在有些情况下，当能够使某些指令不进入ICache时，可以得到最佳性能。即使不这样做，优化后（指令调度）的程序在直接映象Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。
- **减少数据失效，主要通过优化来改善数据的空间局部性和时间局部性，基本方法为：**
  - 数据合并
  - 内外循环交换，循环融合
  - 分块

编译器优化方法举例之一：循环合并

```
// Original Code
for (i = 0; i < N; i++)
      a[i] = b[i] + c[i];
for (i = 0; i < N; i++)
      d[i] = a[i] + b[i] * c[i];
```

Blocks are replaced in first loop then accessed in second

```
// After Loop Fusion
for (i = 0; i < N; i++) {
      a[i] = b[i] + c[i];
      d[i] = a[i] + b[i] * c[i];
}
```

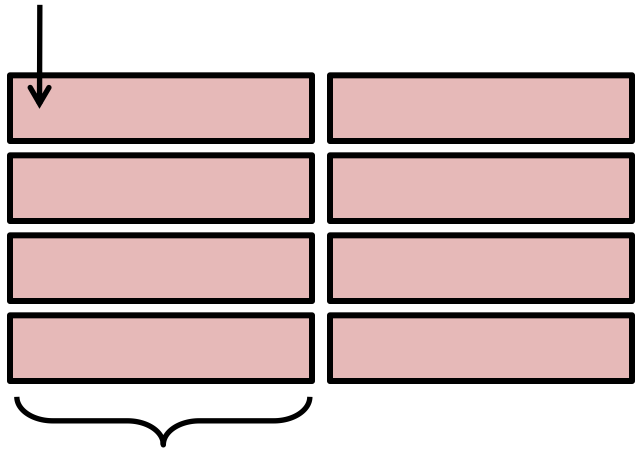
Revised version takes advantage of temporal locality

编译器优化方法举例之二：内外循环交换

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i< 16; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

Ignore the variables sum, i, j

blackboard

BlackBoard

a[0][0]~a[0][7]	T..T00BBB00
a[0][8]~a[0][15]	T..T01BBB00
a[1][0]~a[1][7]	T..T10BBB00
a[1][8]~a[1][15]	T..T11BBB00
a[2][0]~a[2][7]	T..T00BBB00
a[2][8]~a[2][15]	T..T01BBB00
a[3][0]~a[3][7]	T..T10BBB00
a[3][8]~a[3][15]	T..T11BBB00
a[4][0]~a[4][7]	T..T00BBB00
a[4][8]~a[4][15]	T..T01BBB00
a[5][0]~a[5][7]	T..T10BBB00
a[5][8]~a[5][15]	T..T11BBB00
a[6][0]~a[6][7]	T..T00BBB00
a[6][8]~a[6][15]	T..T01BBB00
a[7][0]~a[7][7]	T..T10BBB00
a[7][7]~a[7][15]	T..T11BBB00

a[i][j] in Memory

T...T	SS	BBB00

```
sum_array_rows(double a[16][16])
```

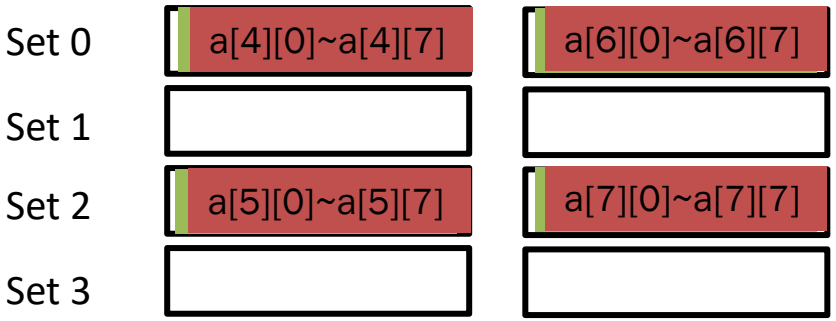
Cache容量：8 blocks
Block：32 B = 4 doubles
2路组相联

Addresses of form T...TSSBBB00
与E=1相比，Set位少1位，Tag位多1位

BlackBoard

a[0][0]~a[0][7]	T..T00BBB00
a[0][8]~a[0][15]	T..T01BBB00
a[1][0]~a[1][7]	T..T10BBB00
a[1][8]~a[1][15]	T..T11BBB00
a[2][0]~a[2][7]	T..T00BBB00
a[2][8]~a[2][15]	T..T01BBB00
a[3][0]~a[3][7]	T..T10BBB00
a[3][8]~a[3][15]	T..T11BBB00
a[4][0]~a[4][7]	T..T00BBB00
a[4][8]~a[4][15]	T..T01BBB00
a[5][0]~a[5][7]	T..T10BBB00
a[5][8]~a[5][15]	T..T11BBB00
a[6][0]~a[6][7]	T..T00BBB00
a[6][8]~a[6][15]	T..T01BBB00
a[7][0]~a[7][7]	T..T10BBB00
a[7][7]~a[7][15]	T..T11BBB00

```
int sum_array_cols(double a[16][16])
```



Cache容量：8 blocks
Block：32 B = 4 doubles
2路组相联

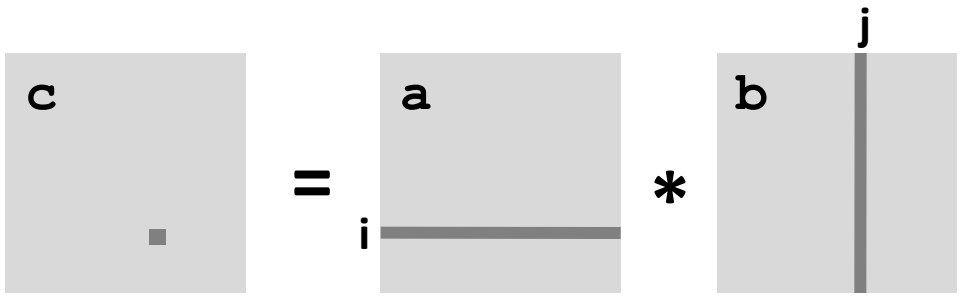
a[i][j] in Memory

Addresses of form T...TSSSBBB00
与E=1相比，Set位少1位，Tag位多1位

编译器优化方法举例之三：分块

基本的矩阵相乘

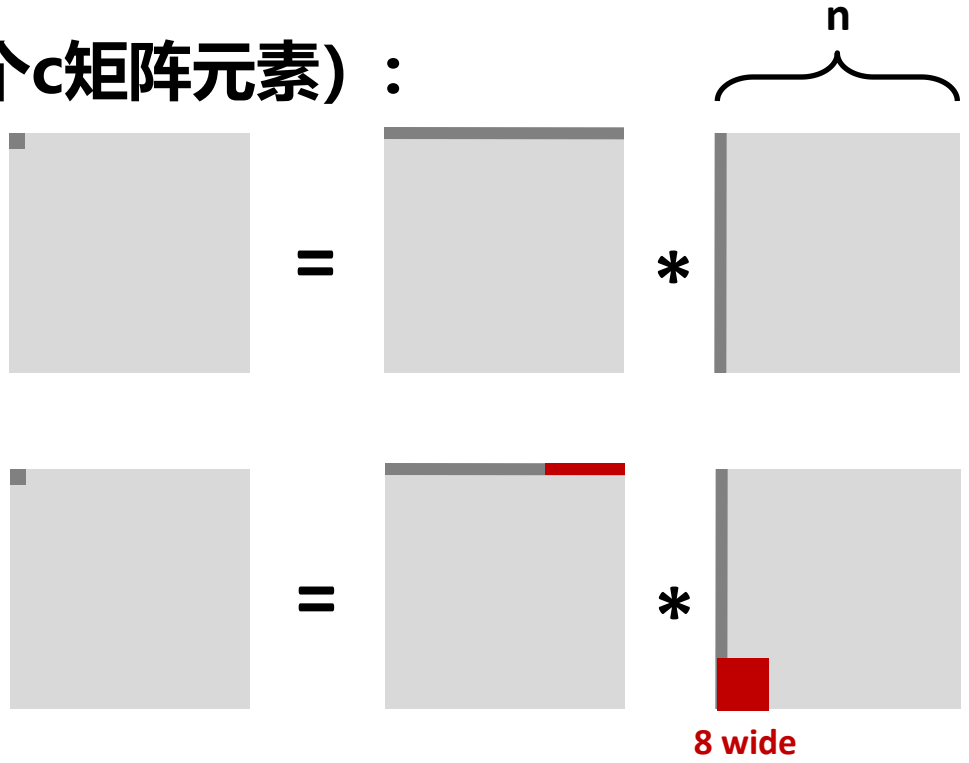
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
             for (k = 0; k < n; k++)
         c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)

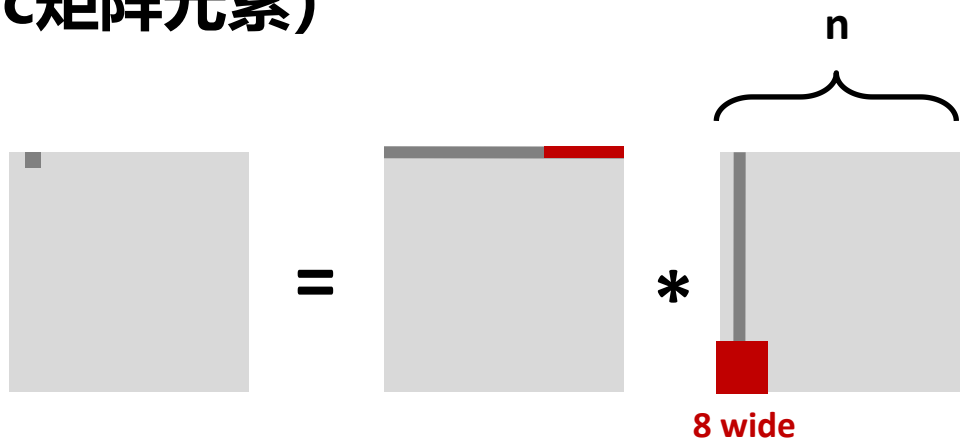
- **第一次循环（计算第一个c矩阵元素）：**
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)

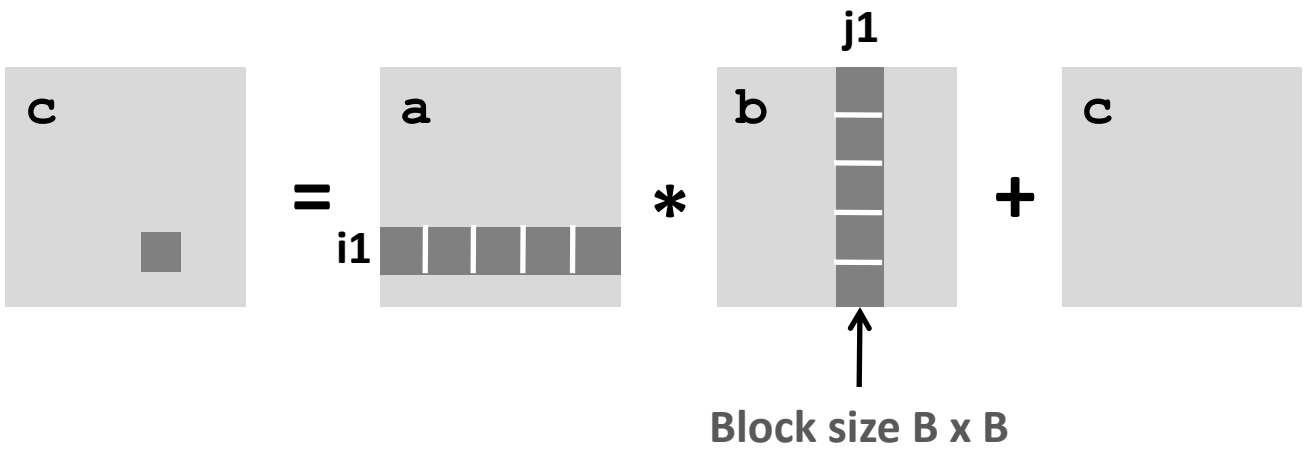
- **第2次循环（计算第2个c矩阵元素）**
  - Again:
    n/8 + n = 9n/8 misses



- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

矩阵分块相乘

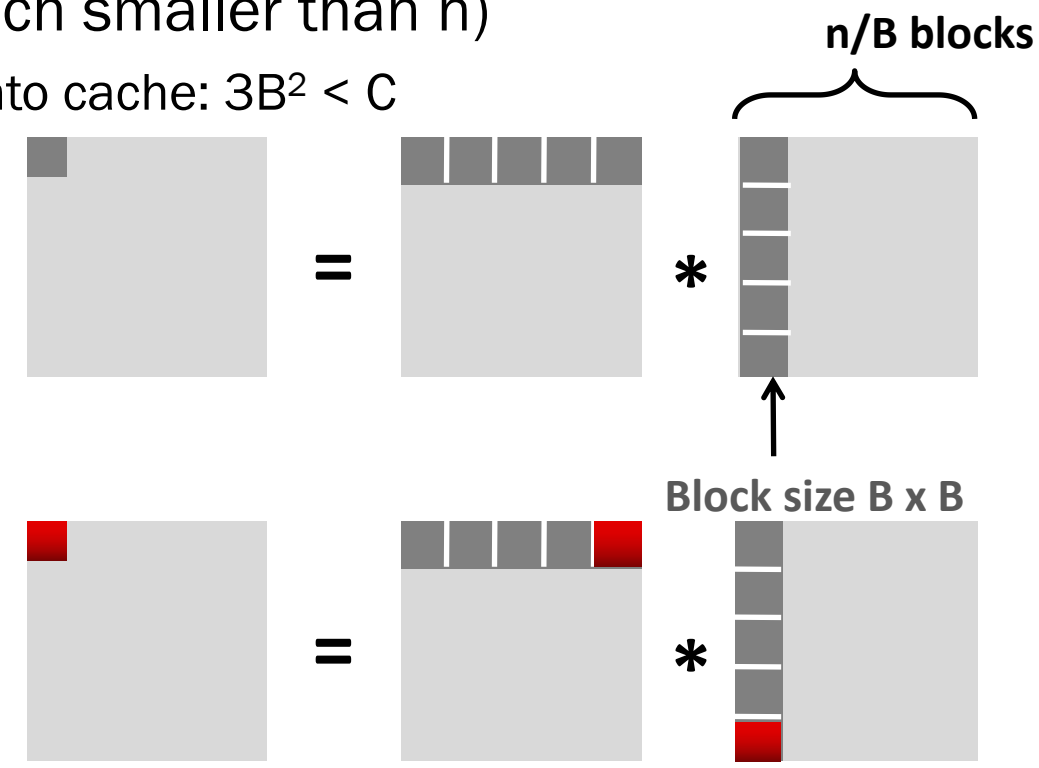
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
     for (j = 0; j < n; j+=B)
              for (k = 0; k < n; k+=B)
          /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i1++)
                      for (j1 = j; j1 < j+B; j1++)
                          for (k1 = k; k1 < k+B; k1++)
                          c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)
  - 可以存放三块子矩阵： fit into cache: $3B^2 < C$

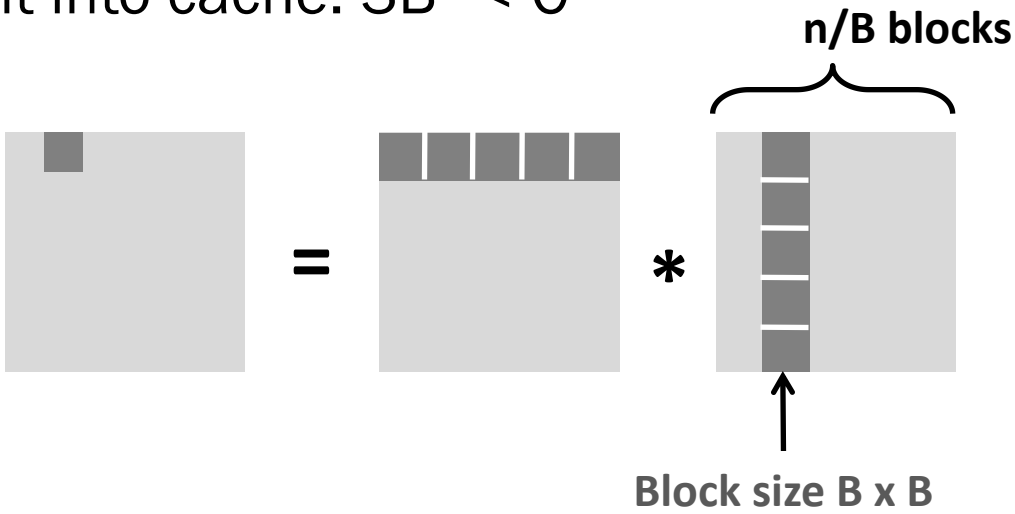
- **计算第1个子矩阵:**
  - $B^2/8$ misses for each block
    - **为什么?**
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)
  - 可以存放三块子矩阵： fit into cache: $3B^2 < C$

- **计算第2个子矩阵:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

分块技术小结

- **矩阵不分块相乘失效次数: (9/8) * n^3**
- **矩阵分块相乘失效次数: 1/(4B) * n^3**

- **建议：尽可能分块大一些, 但需保证$3B^2 < C$!**

- **分块与不分块的性能差异大的原因:**
  - 矩阵相乘具有固有的局部性:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - 编写Cache友好的程序，可以利用程序固有的局部性

4.3 Cache的高级优化方法

- **缩短命中时间**
- **增加Cache带宽**
- **减小失效开销**
- **降低失效率**
- **通过并行降低失效开销或失效率**

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

9、Hardware Prefetching

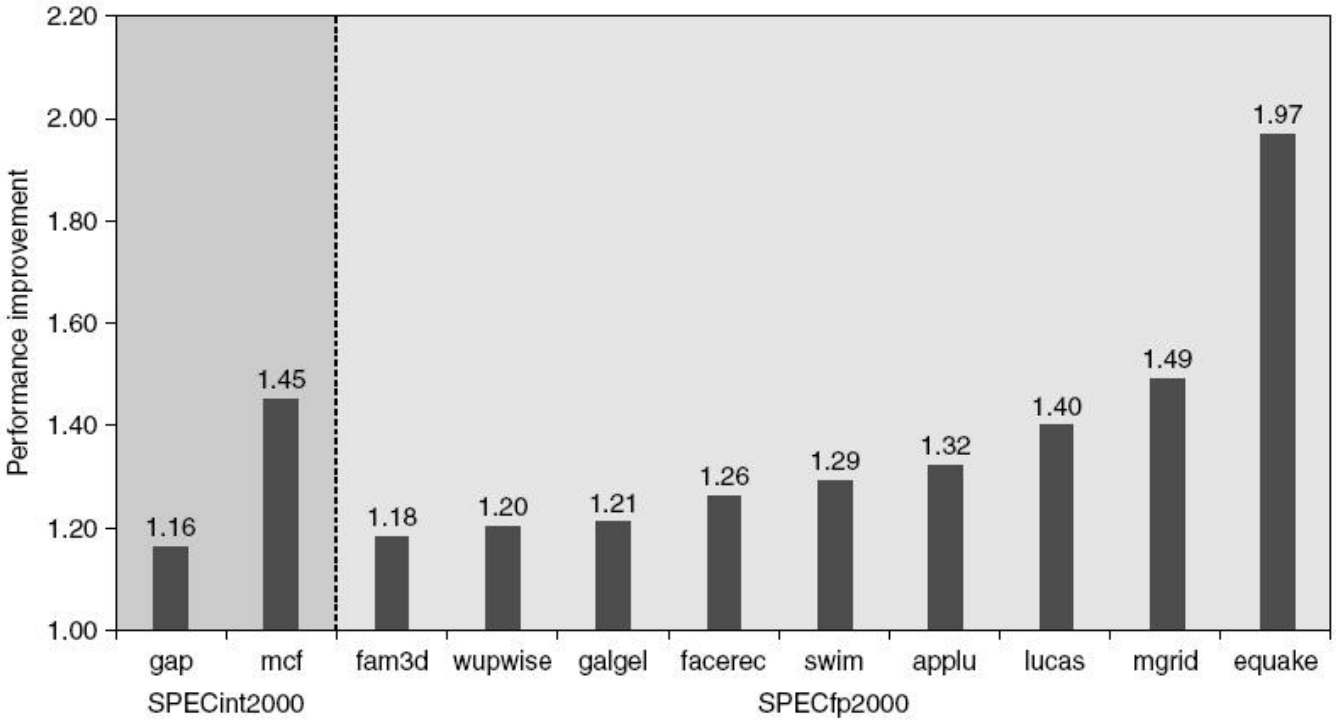
- **预取指令**
  - CPU在执行当前块代码时，硬件预取下一块代码
  - CPU可能马上就要执行这块代码，这样可以降低或消除Cache的访问失效
- **当块中有控制指令时，预取失效**
- **预取的指令可以放在Icache中，也可以放在其他地方（存取速度比Memory块的地方）**
- **AXP21064失效时，取2块指令块**
  - 目标块放在Icache，下一块放在ISB(指令流缓冲）中
  - 如果访问的块在ISB中，取消访存请求，直接从ISB中读，并发出对下一指令块的预取访存请求
- **研究结果：块大小为16字节，容量为4KB的直接映象Cache，1个块的指令流缓冲器，可以捕获15% - 25%的失效，4个块ISB可捕获50%的失效，16块ISB可捕获72%的失效**

硬件预取

- **预取数据**
  - 出发点：CPU访问一块数据，可能马上要访问下一块数据
  - Jouppi研究结果：块大小16字节，4KB直接映象Cache， 1Block DSB - 25%
  - 4Block DSB - 43%
  - Palacharla和Kessler 1994年研究
    - 一个具有两个64KB四路组相联(Icache, Dcache)的处理器来说，8Blocks流缓冲器能够捕获其50% - 70%的失效
- **举例：Alpha AXP21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变?**
  - 假设当指令不在指令Cache中，在预取缓冲区中找到时，需要多花1个时钟周期。
  - 假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期
  - 8KB指令Cache的失效率为1.10%，16KB指令cache的失效率为0.64%
  - AMAT（预取）= 命中时间+失效率*预取命中率*1+失效率*（1-预取命中率）*失效开销
- **注意：预取是利用存储器的空闲带宽，而不是与正常的存储器操作竞争。**

硬件预取

Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

10、Compiler Prefetching（1/2)

- **在ISA中增加预取指令，让编译器控制预取**
- **预取的种类**
  - 寄存器预取：把数据取到R中
  - Cache预取：只将数据取到Cache中，不放入寄存器
- **故障问题**
  - 两种类型的预取：故障性预取和非故障性预取
  - 所谓故障性预取：指在预取时若出现虚地址故障，或违反保护权限，就会有异常发生
  - 所谓非故障性预取：如导致异常就转化为空操作
- **只有在预取时，CPU可以继续执行的情况下，预取才有意义**
  - Cache在等待预取数据返回的同时，可以正常提供指令和数据，称为非阻塞Cache或非锁定Cache

由编译器控制预取 (2/2)

- **循环是预取优化的主要目标**
  - 失效开销较小时，将循环体展开一两次，调度好预取与执行的重叠
  - 失效开销较大时，将循环体展开多次，为较远循环预取数据
  - 由于发出预取指令要花费一条指令的开销，因此要避免不必要的预取
  - 重点放在那些可能导致失效的访问
- **举例1：P93/P70 Hennessy & Patterson 5th/中译本**
- **举例2：P94/P71 Hennessy & Patterson 5th/中译本**

举例1

Example： For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching.

Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

a和b均为双精度浮点数（8Bytes）
Cache组织：容量8KB，直接映像方式（1路组相联）
块大小：16B

```
for (j = 0; j < 100; j = j+1) {
      prefetch(b[j+7][0]);              //1 cycle
      /* b(j,0) for 7 iterations later */
      prefetch(a[0][j+7]);              // 1cycle
      /* a(0,j) for 7 iterations later */
      a[0][j] = b[j][0] * b[j+1][0];};   // 7 cycles

for (i = 1; i < 3; i = i+1)
      for (j = 0; j < 100; j = j+1) {
            prefetch(a[i][j+7]);
            /* a(i,j) for +7 iterations */
            a[i][j] = b[j][0] * b[j+1][0];}
```

举例2

Example：

Calculate the time saved in the example above.

(1) Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache.

(2) Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth.

(3) Here are the key loop times ignoring cache misses: The original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

小结：cache

Cache性能分析

- AMAT = Hit Time+Miss rate×Miss penalty
- AMAT = Hit TimeL1+Miss rateL1×
  (Hit TimeL2+Miss rateL2×Miss penaltyL2)

- **6种基本优化方法**
- **10 种高级优化方法**

小结-基本优化方法

- **降低失效率： 引起失效的3C**
  - 1、增加Cache块的大小
  - 2、增大Cache容量
  - 3、提高相联度
- **减少失效开销**
  - 4、多级Cache
  - 5、使读失效优先于写失效
- **缩短命中时间**
  - 6、避免在索引缓存期间进行地址转换

小结-高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.