# Computer Architecture
# ELE 475 / COS 475
# Slide Deck 9: Advanced Caches

David Wentzlaff

Department of Electrical Engineering

Princeton University

PRINCETON UNIVERSITY

PRINCETON
School of Engineering and Applied Science

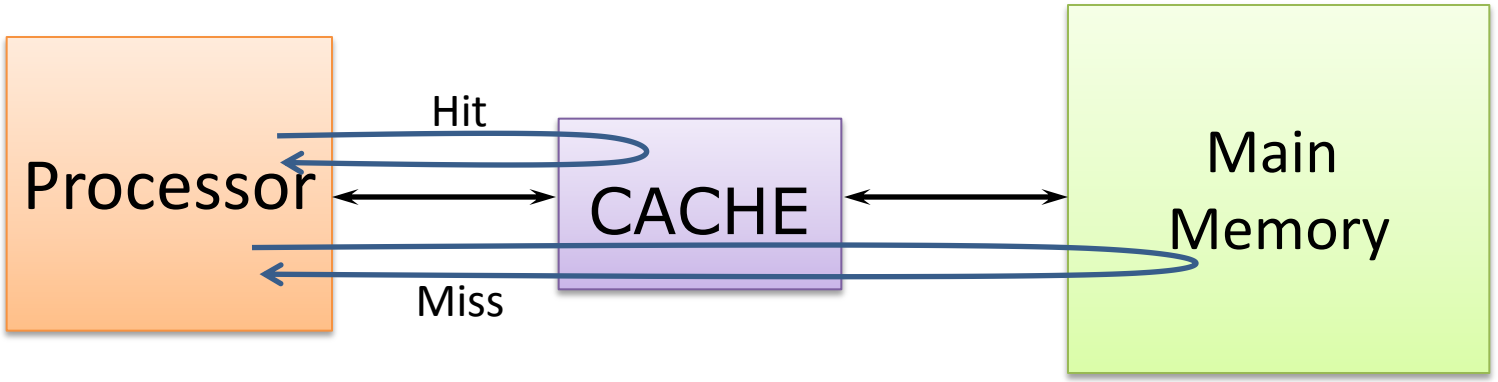
# Agenda

- Review
  - Three C's
  - Basic Cache Optimizations
- Advanced Cache Optimizations
  - Pipelined Cache Write
  - Write Buffer
  - Multilevel Caches
  - Victim Caches
  - Prefetching
    - Hardware
    - Software
  - Multiporting and Banking
  - Software Optimizations
  - Non-Blocking Cache
  - Critical Word First/Early Restart

# Agenda

- Review
  - Three C's
  - Basic Cache Optimizations
- Advanced Cache Optimizations
  - Pipelined Cache Write
  - Write Buffer
  - Multilevel Caches
  - Victim Caches
  - Prefetching
    - Hardware
    - Software
  - Multiporting and Banking
  - Software Optimizations
  - Non-Blocking Cache
  - Critical Word First/Early Restart

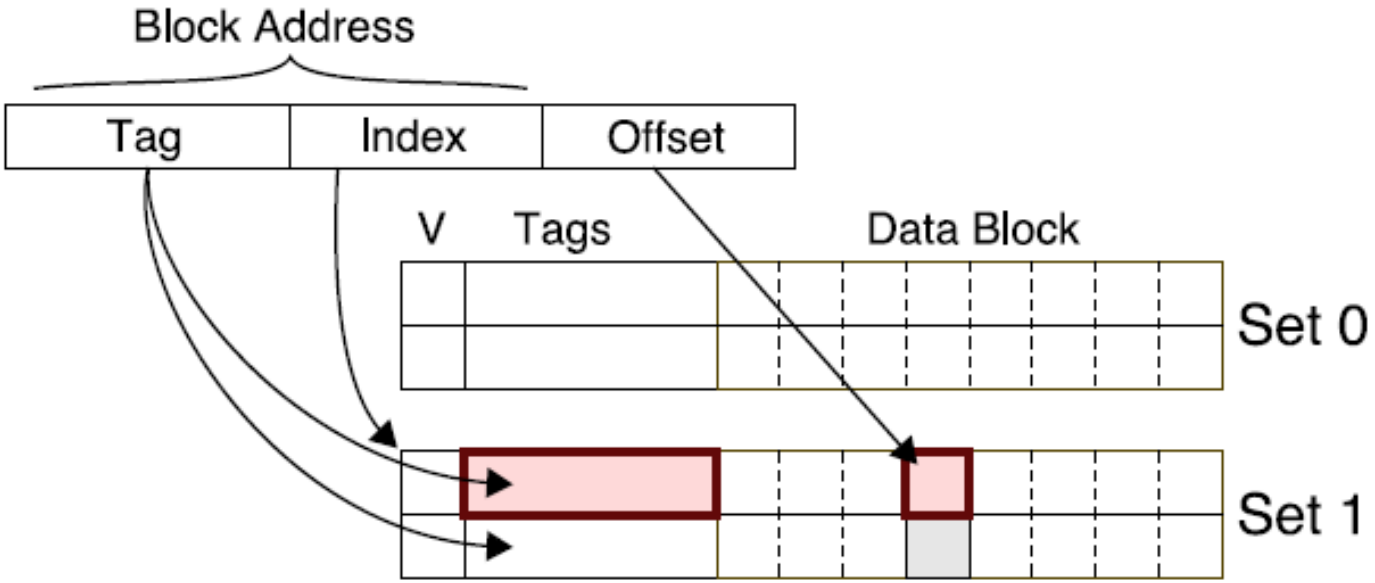
# Average Memory Access Time





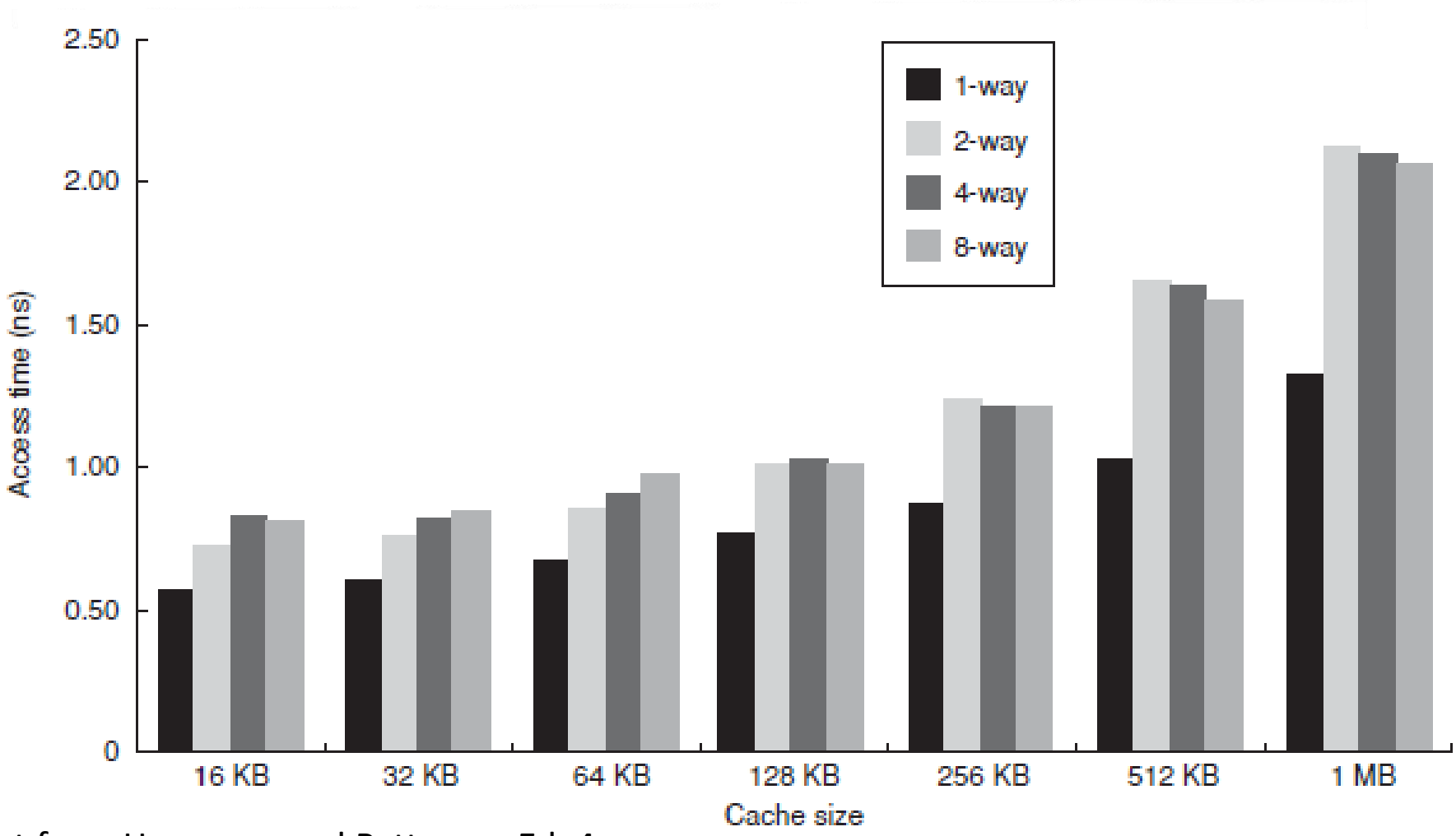
- Average Memory Access Time = Hit Time + ( Miss Rate * Miss Penalty )

# Categorizing Misses: The Three C's



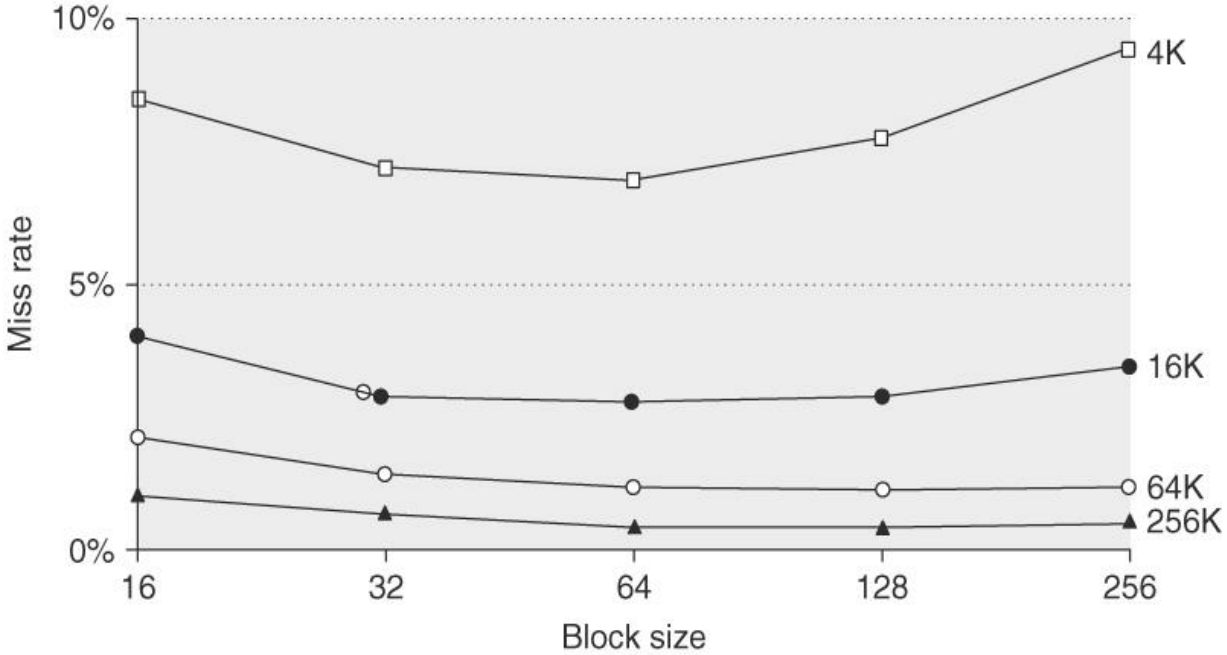

- **Compulsory** – first-reference to a block, occur even with infinite cache
- **Capacity** – cache is too small to hold all data needed by program, occur even under perfect replacement policy (loop over 5 cache lines)
- **Conflict** – misses that occur because of collisions due to less than full associativity (loop over 3 cache lines)

# Reduce Hit Time: Small & Simple Caches




Plot from Hennessy and Patterson Ed. 4

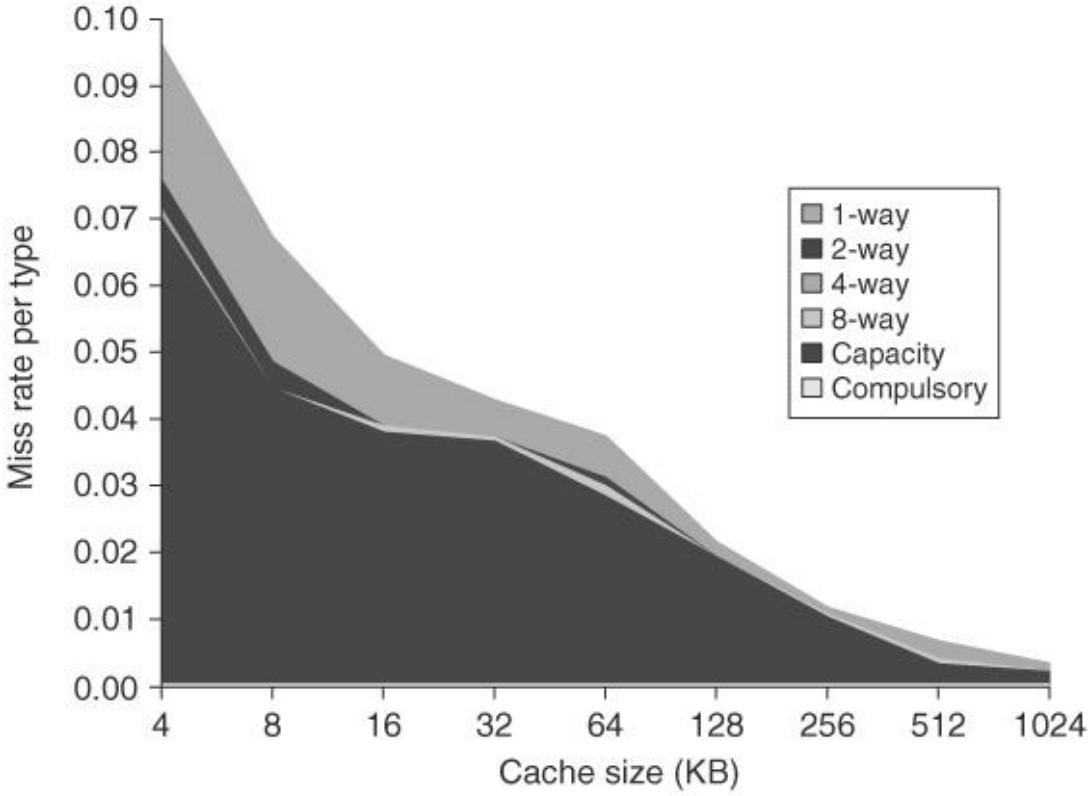
# Reduce Miss Rate: Large Block Size




- Less tag overhead
- Exploit fast burst transfers from DRAM
- Exploit fast burst transfers over wide on-chip busses

- Can waste bandwidth if data is not used
- Fewer blocks -> more conflicts

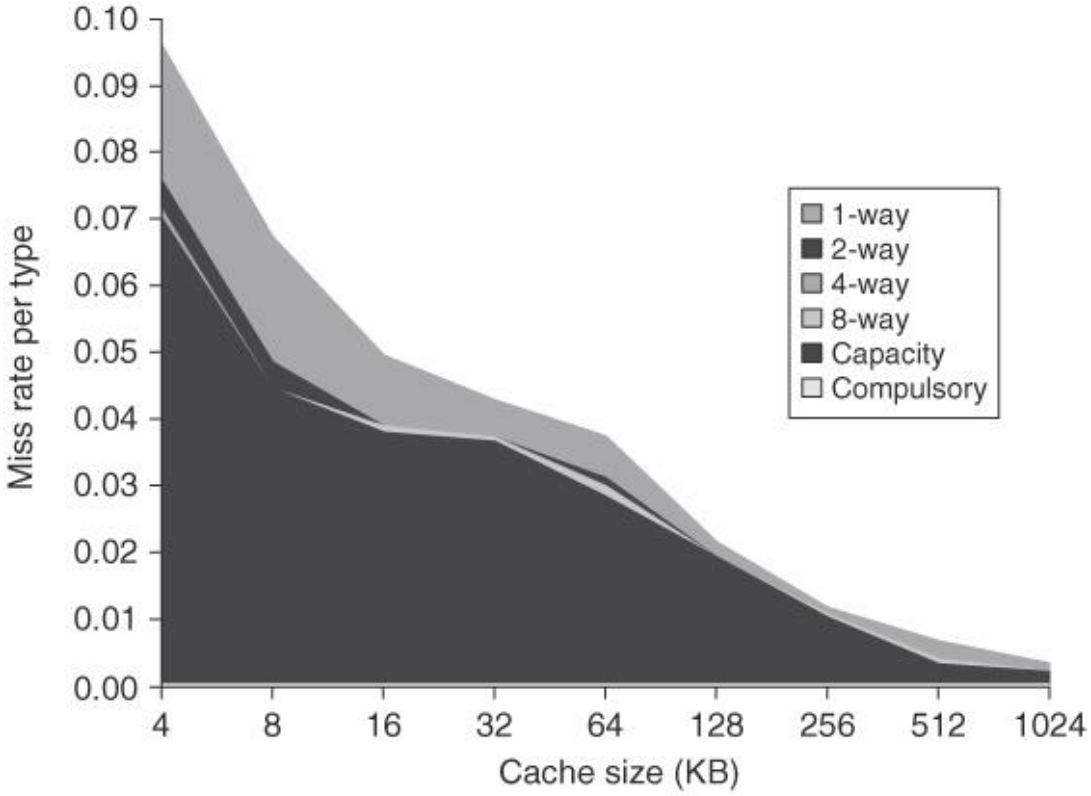
# Reduce Miss Rate: Large Cache Size




Empirical Rule of Thumb:

If cache size is doubled, miss rate usually drops by about $\sqrt{2}$

# Reduce Miss Rate: High Associativity



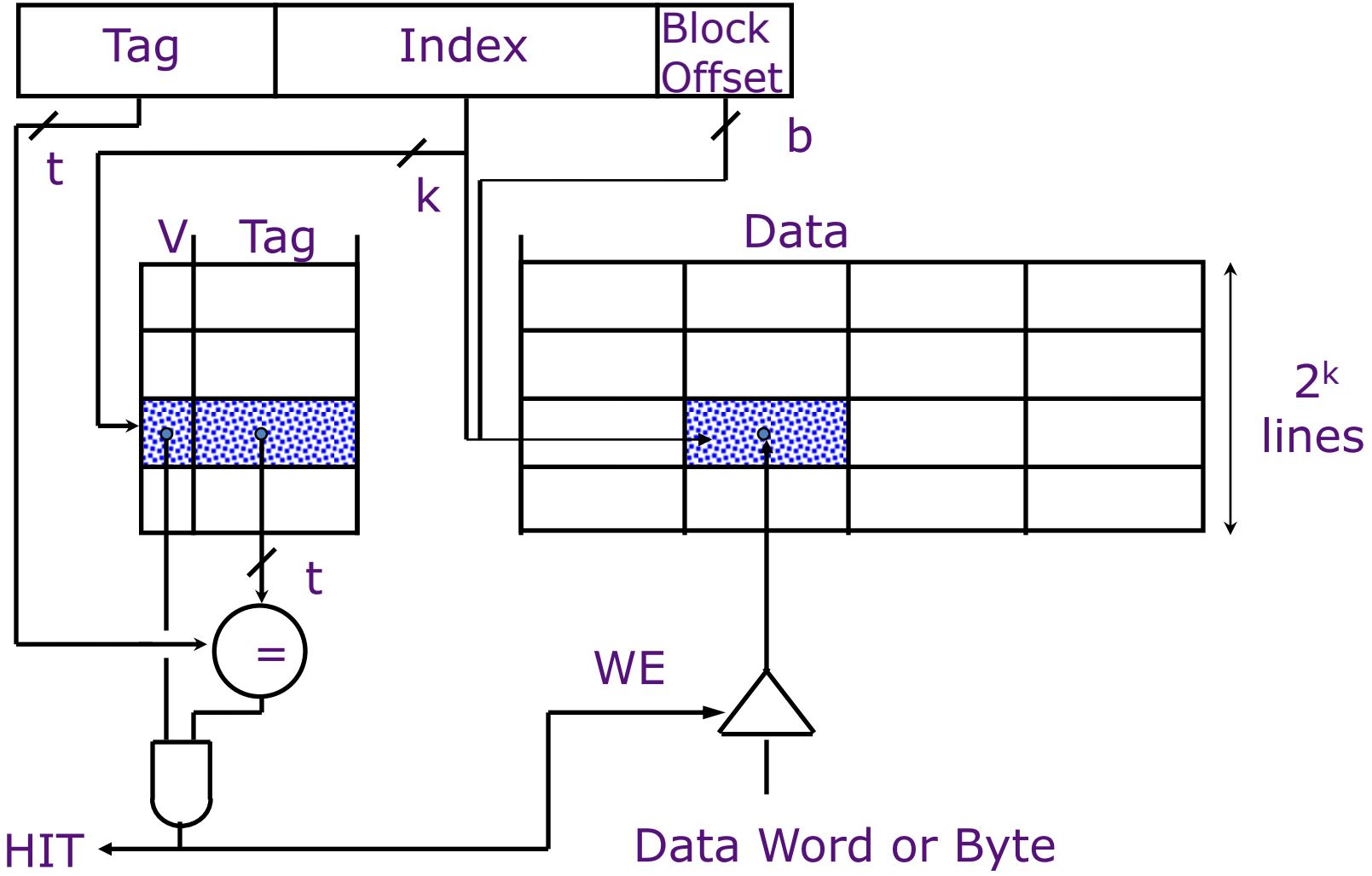

Empirical Rule of Thumb:

Direct-mapped cache of size N has about the same miss rate as a two-way set- associative cache of size N/2

# Agenda

- Review
  - Three C's
  - Basic Cache Optimizations
- Advanced Cache Optimizations
  - Pipelined Cache Write
  - Write Buffer
  - Multilevel Caches
  - Victim Caches
  - Prefetching
    - Hardware
    - Software
  - Multiporting and Banking
  - Software Optimizations
  - Non-Blocking Cache
  - Critical Word First/Early Restart

# Write Performance



# Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

Solutions:

- Design data RAM that can perform read and write concurrently, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check

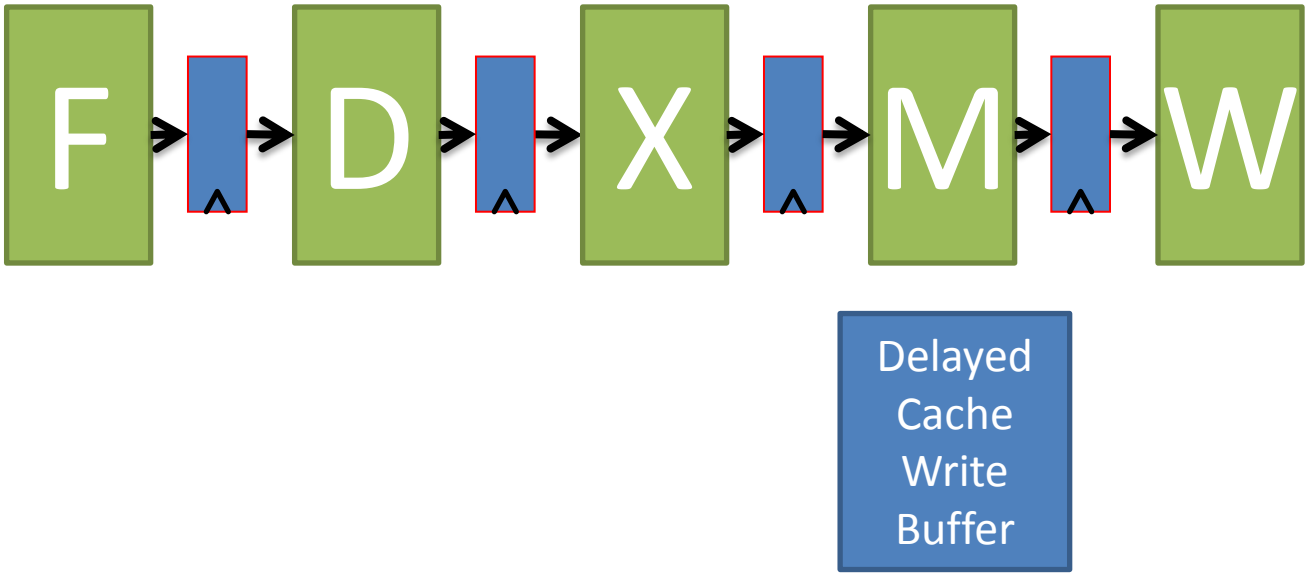
# Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

Solutions:

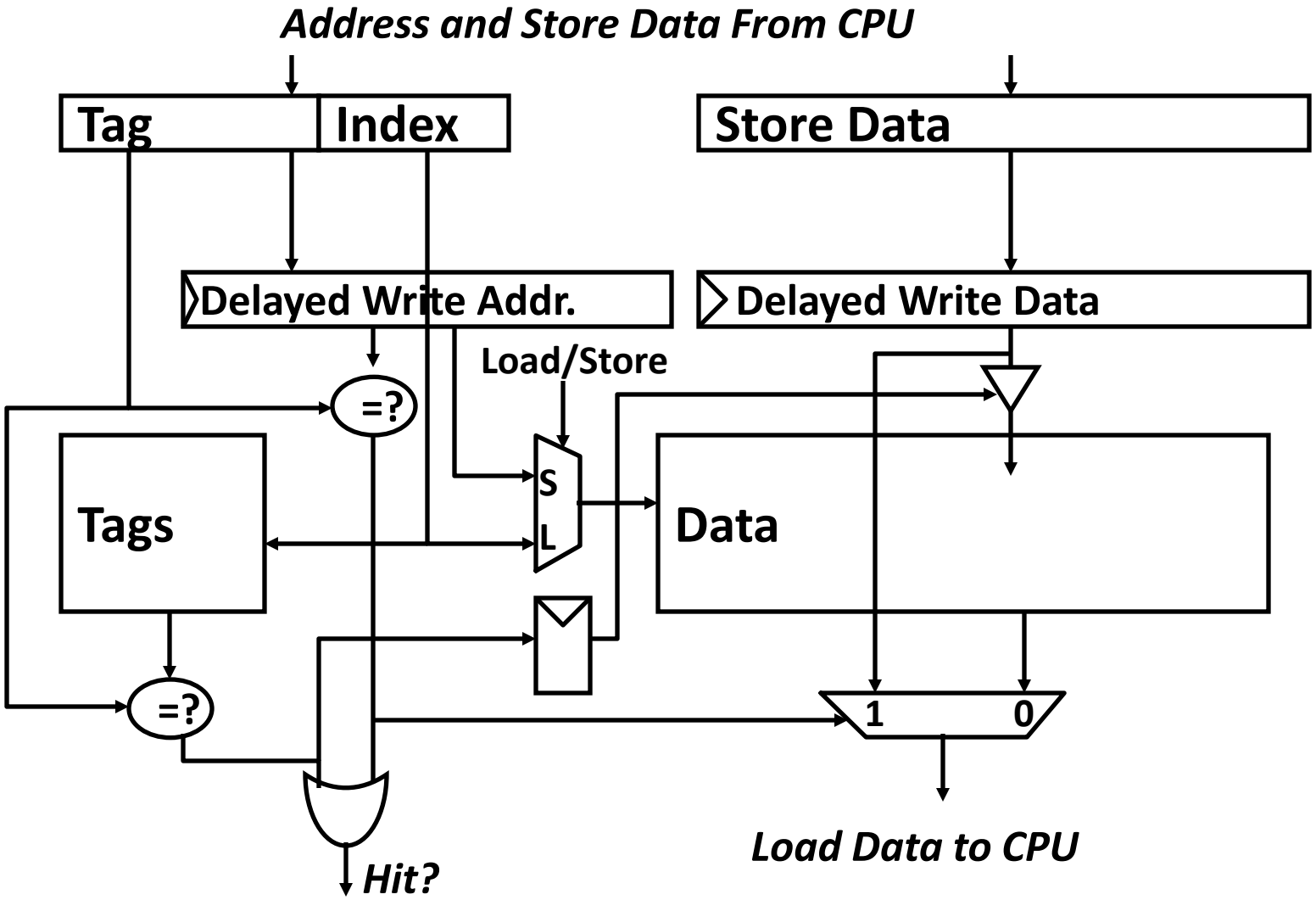
- Design data RAM that can perform read and write concurrently, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check

# Pipelining Cache Writes




Data from a store hit written into data portion of cache during tag access of subsequent store

# Pipelining Cache Writes




Data from a store hit written into data portion of cache during tag access of subsequent store

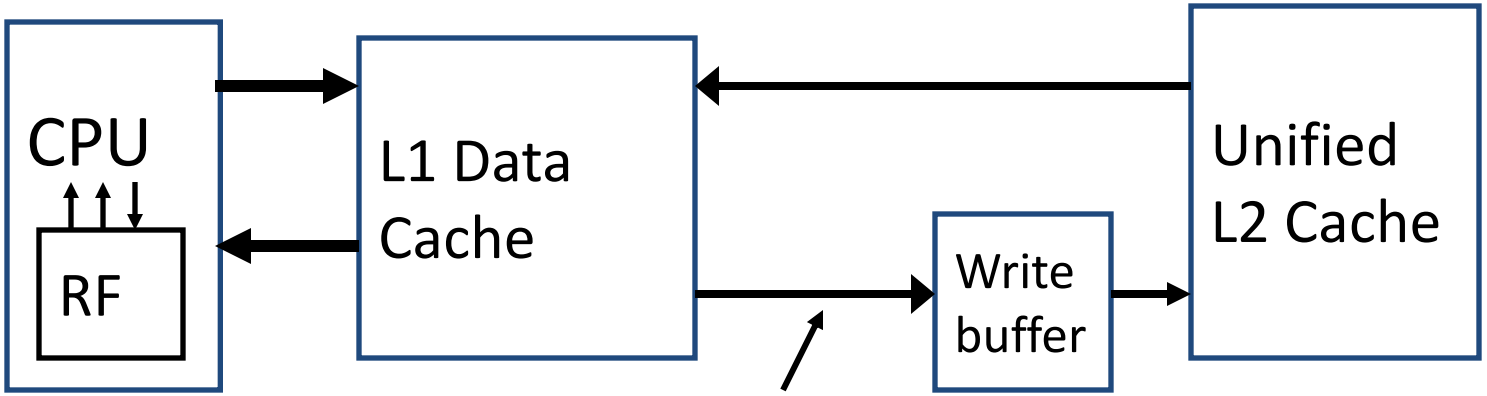
# Pipelined Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Pipelined Writes | | | | |

# Pipelined Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Pipelined Writes | | | - | + |

# Write Buffer to Reduce Read Miss Penalty




Evicted dirty lines for writeback cache
OR
All writes in writethrough cache

Processor is not stalled on writes, and read misses can go ahead of write to main memory

**Problem:** Write buffer may hold updated value of location needed by a read miss

**Simple scheme:** on a read miss, wait for the write buffer to go empty

**Faster scheme:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

# Write Buffer Efficacy

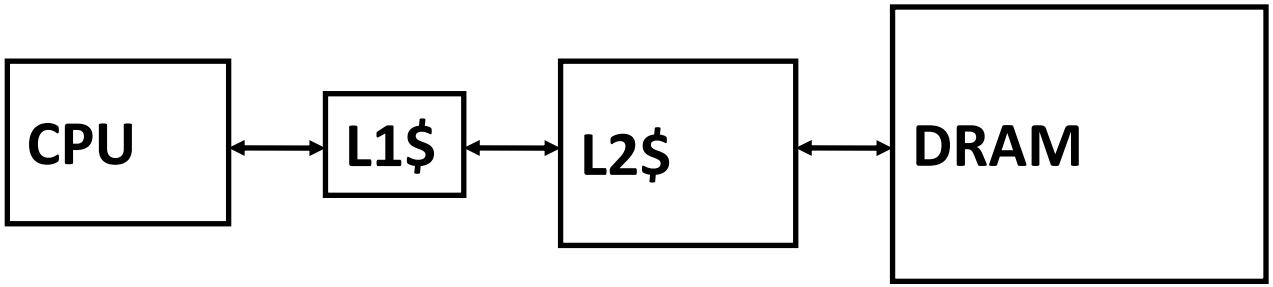
| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Write Buffer | | | | |

# Write Buffer Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Write Buffer | | + | | |

# Multilevel Caches

Problem: A memory cannot be large and fast
Solution: Increasing sizes of cache at each level




Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions

# Presence of L2 influences L1 design

- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time and reduced L1 miss penalty
  - Reduces average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn't go off-chip
  - At most one L1 miss request per L1 access (no dirty victim write back) simplifies pipeline control
  - Simplifies coherence issues
  - Simplifies error recovery in L1 (can use just parity bits in L1 and reload from L2 when parity error detected on L1 read)

# Inclusion Policy

- **Inclusive** multilevel cache:
  - Inner cache holds copies of data in outer cache
  - External coherence snoop access need only check outer cache
- **Exclusive** multilevel caches:
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

Why choose one type or the other?

# Itanium-2 On-Chip Caches

(Intel/HP. 2002)

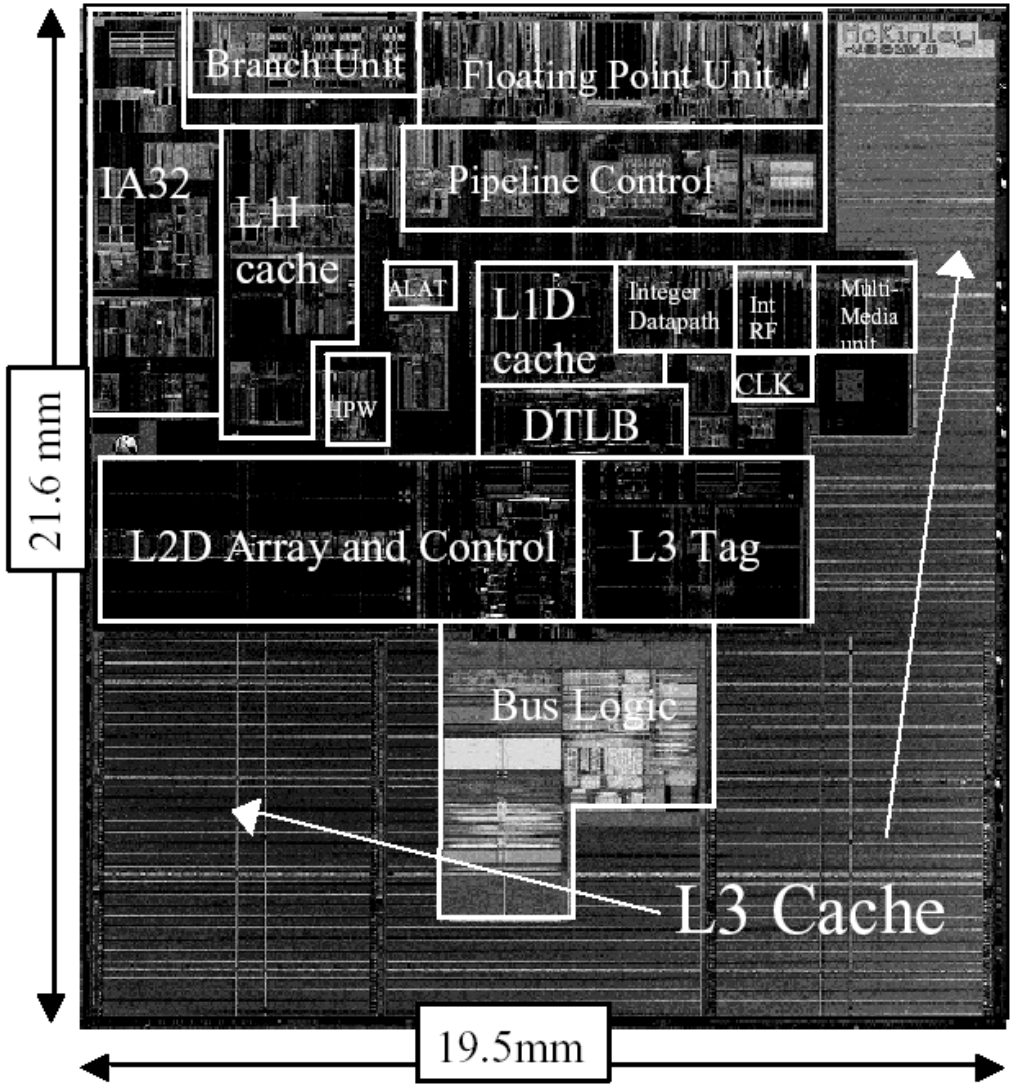



Image Credit: Intel

**Level 1:** 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

**Level 2:** 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency

**Level 3:** 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

# Power 7 On-Chip Caches [IBM 2009]

32KB L1 I$/core
32KB L1 D$/core
3-cycle latency

256KB Unified L2$/core
8-cycle latency

32MB Unified Shared L3$
Embedded DRAM
25-cycle latency to local slice

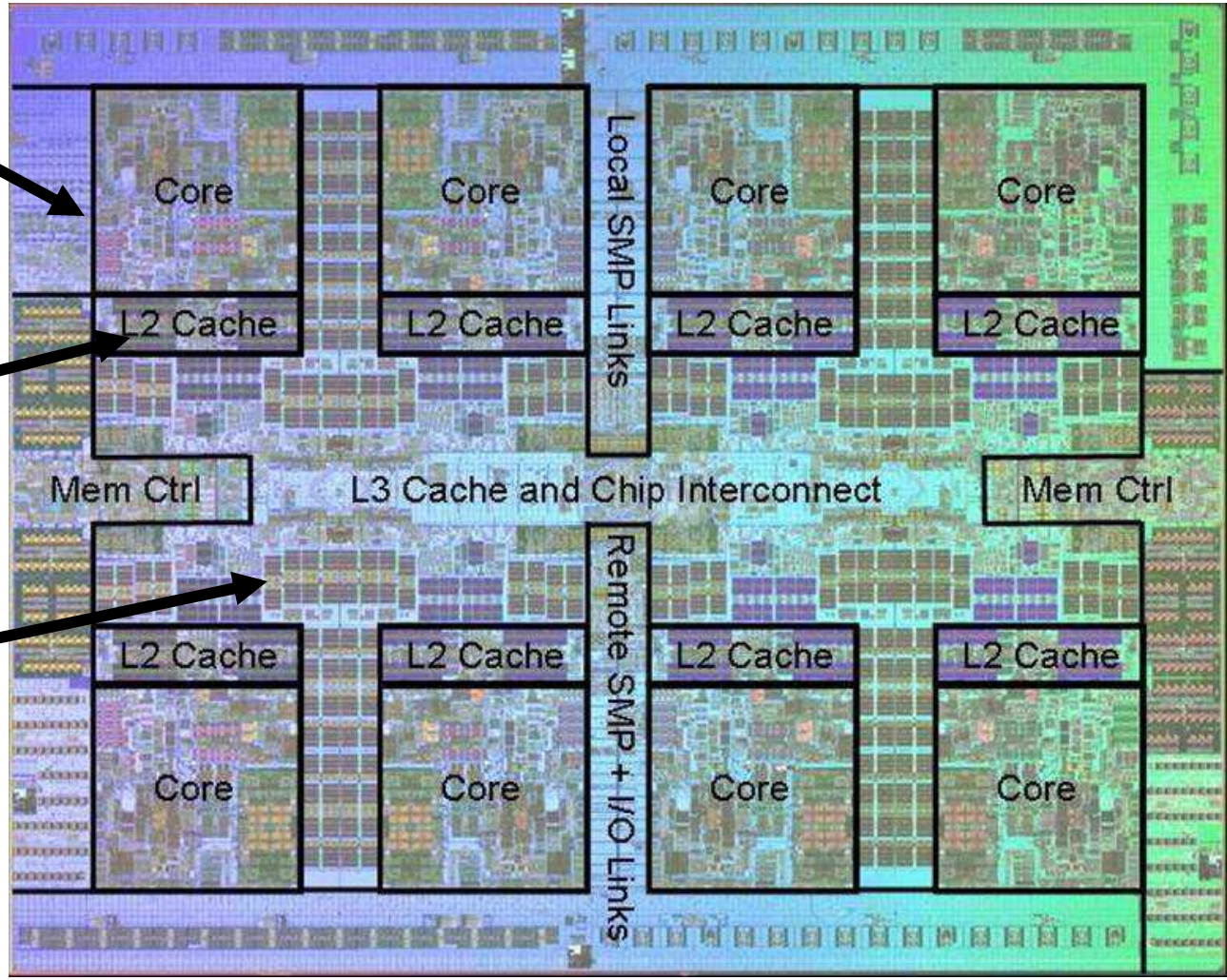



Image Credit: IBM
Courtesy of International Business Machines Corporation.

# Multilevel Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Multilevel Cache | | | | |

# Multilevel Cache Efficacy L1

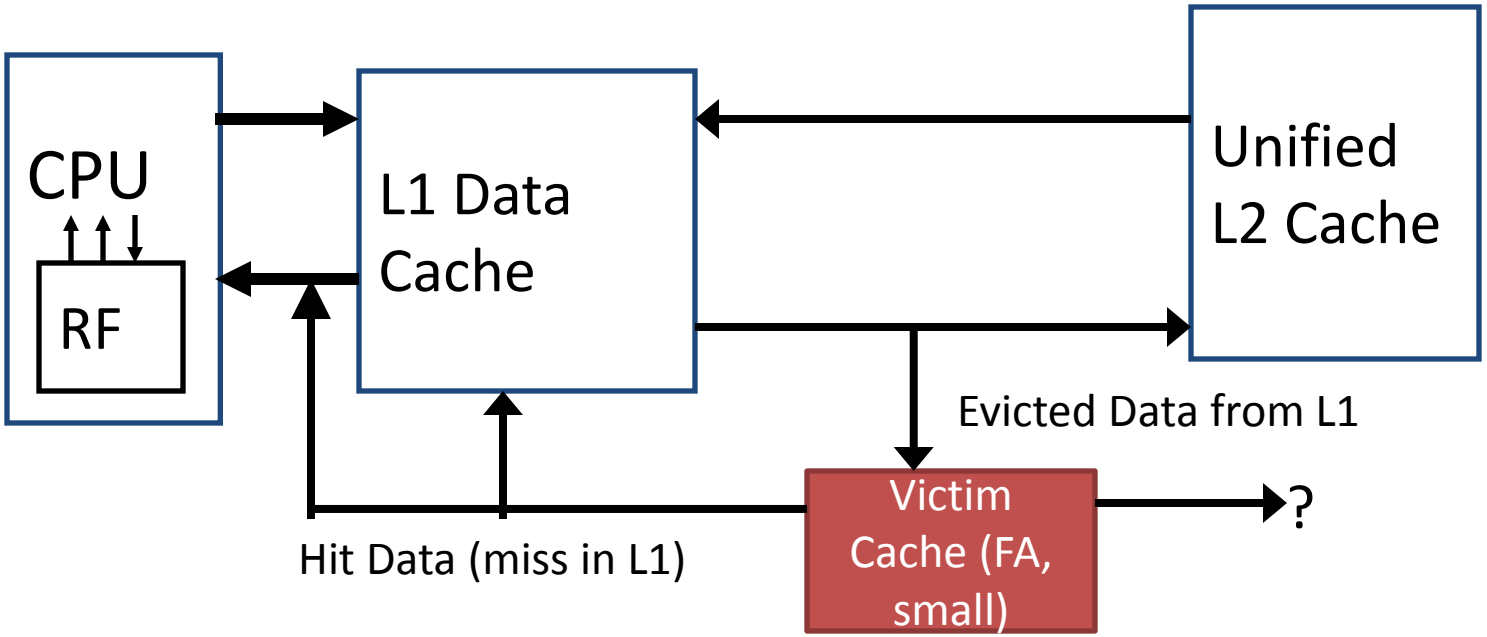
| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Multilevel Cache | | + | | |

# Multilevel Cache Efficacy L1, L2, L3

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Multilevel Cache | + | + | | |

# Victim Cache

- Small Fully Associative cache for recently evicted lines
  - Usually small (4-16 blocks)
- Reduced conflict misses
  - More associativity for small number of lines
- Can be checked in parallel or series with main cache
- On Miss in L1, Hit in VC:  VC->L1, L1->VC
- On Miss in L1, Miss in VC: L1->VC, VC->? (Can always be clean)



# Victim Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Victim Cache | | | | |

# Victim Cache Efficacy L1

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Victim Cache | | + | | |

# Victim Cache Efficacy L1 and VC

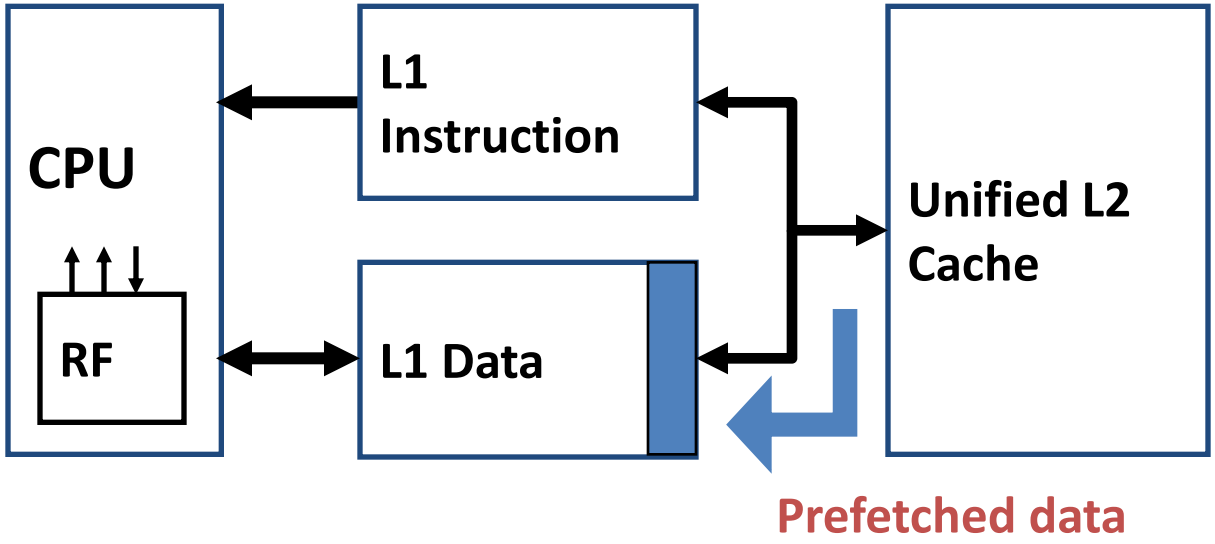
| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Victim Cache | + | + | | |

# Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *What types of misses does prefetching affect?*

# Issues in Prefetching

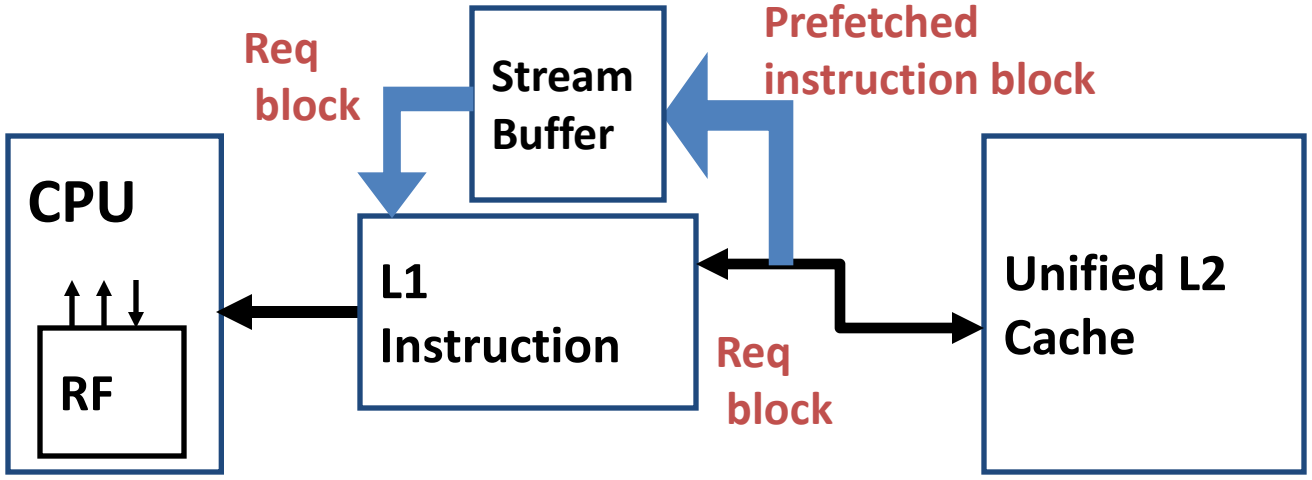
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



# Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)



# Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch b + 1 upon miss on b
- One Block Lookahead (OBL) scheme
  - Initiate prefetch for block b + 1 when block b is accessed
  - *Why is this different from doubling block size?*
  - Can extend to N-block lookahead
- Strided prefetch
  - If observe sequence of accesses to block b, b+N, b+2N, then prefetch b+3N etc.

**Example:** IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

# Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
 }
```

# Software Prefetching Issues

- Timing is the biggest issue, not predictability
  - If you prefetch very close to when the data is required, you might be too late
  - Prefetch too early, cause pollution
  - Estimate how long it will take for the data to come into L1, so we can set P appropriately
  - *Why is this hard to do?*

```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
 }
```

**Must consider cost of prefetch instructions**

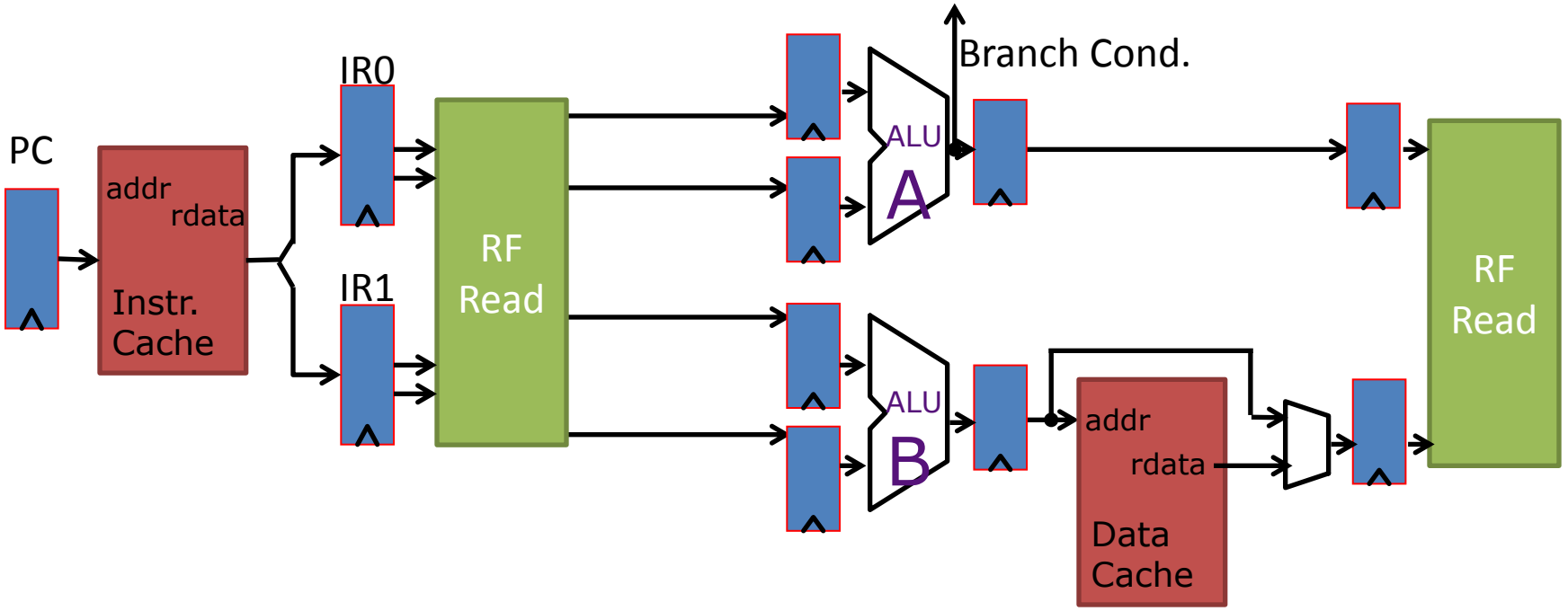
# Prefetching Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Prefetching | | | | |

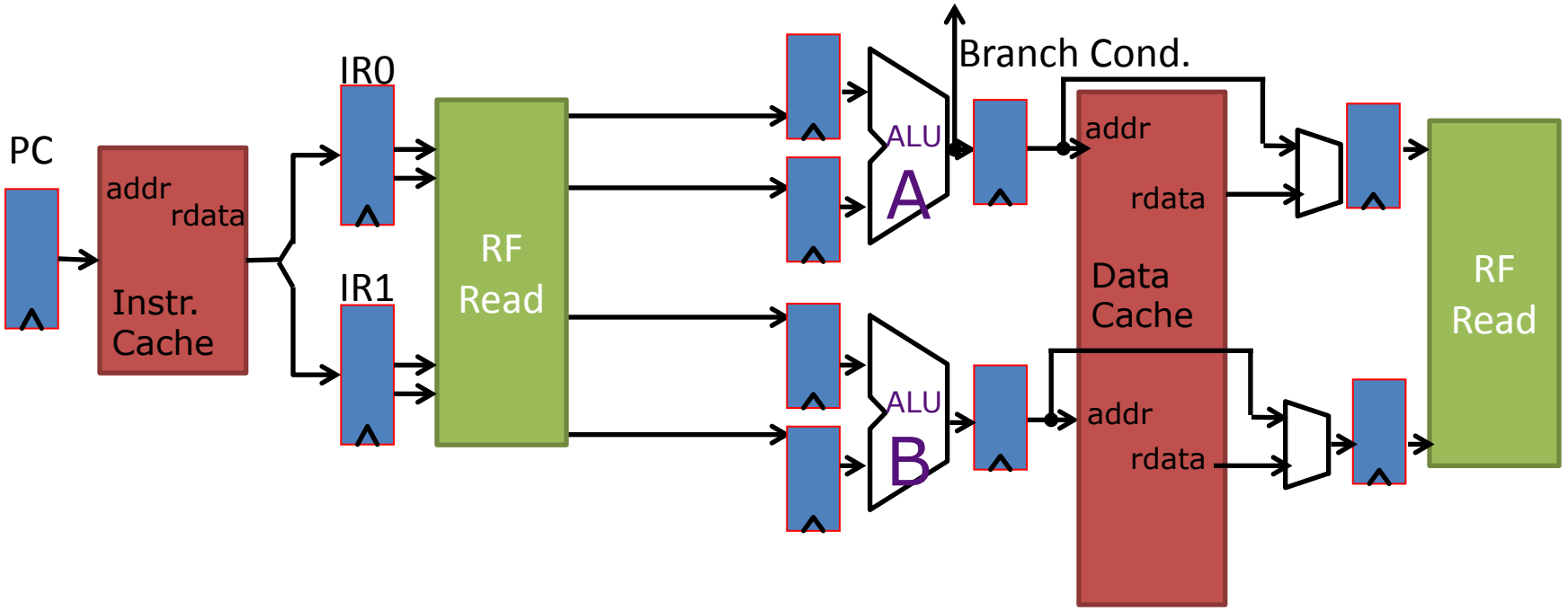
# Prefetching Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Prefetching | + | + | | |

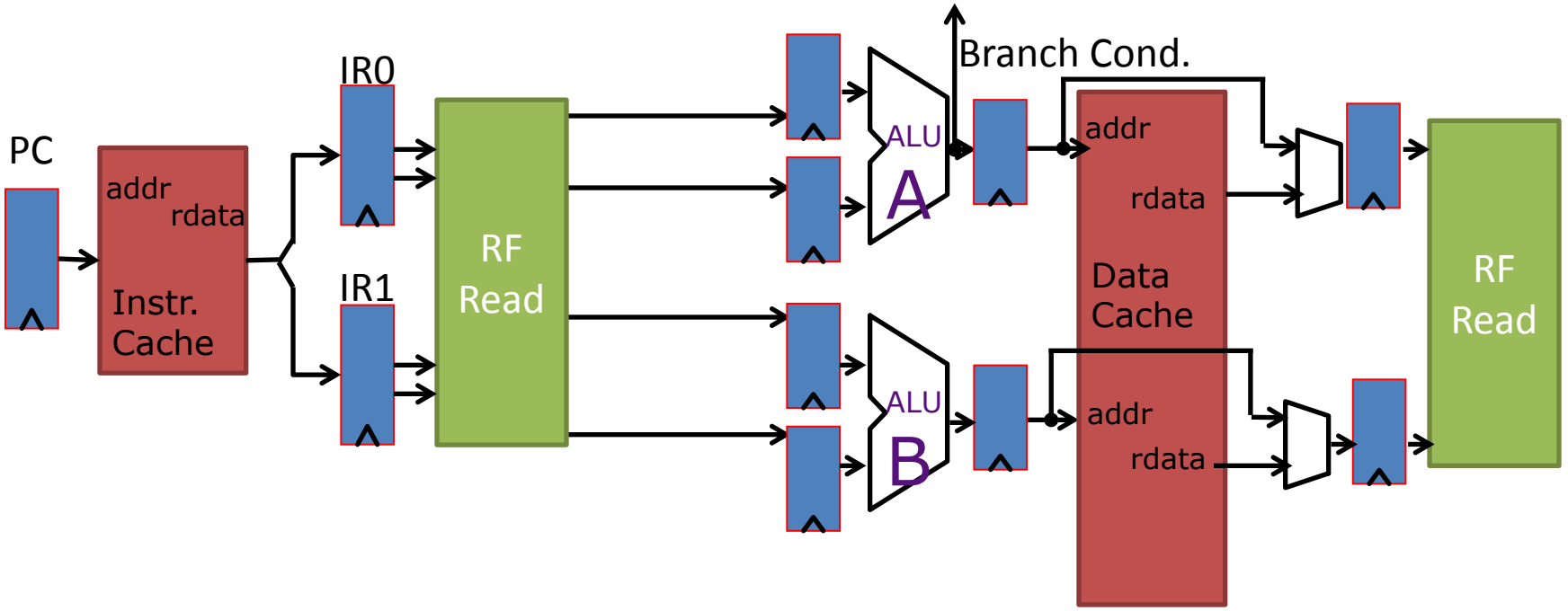
# Increasing Cache Bandwidth Multiporting and Banking



# Increasing Cache Bandwidth Multiporting and Banking



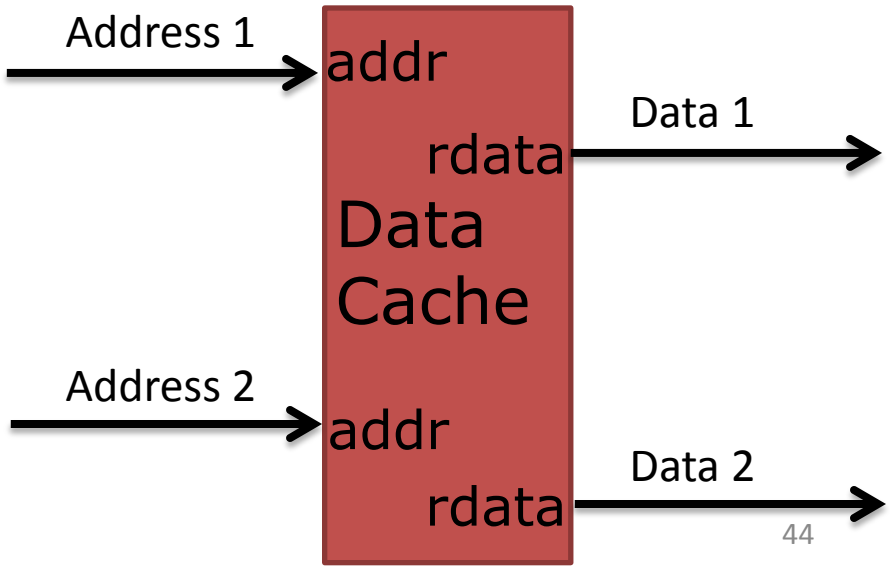
# Increasing Cache Bandwidth Multiporting and Banking




Challenge: Two stores to the same line, or Load and Store to same line

# True Multiport Caches

- Large area increase (could be double for 2-port)
- Hit time increase (can be made small)



# Banked Caches

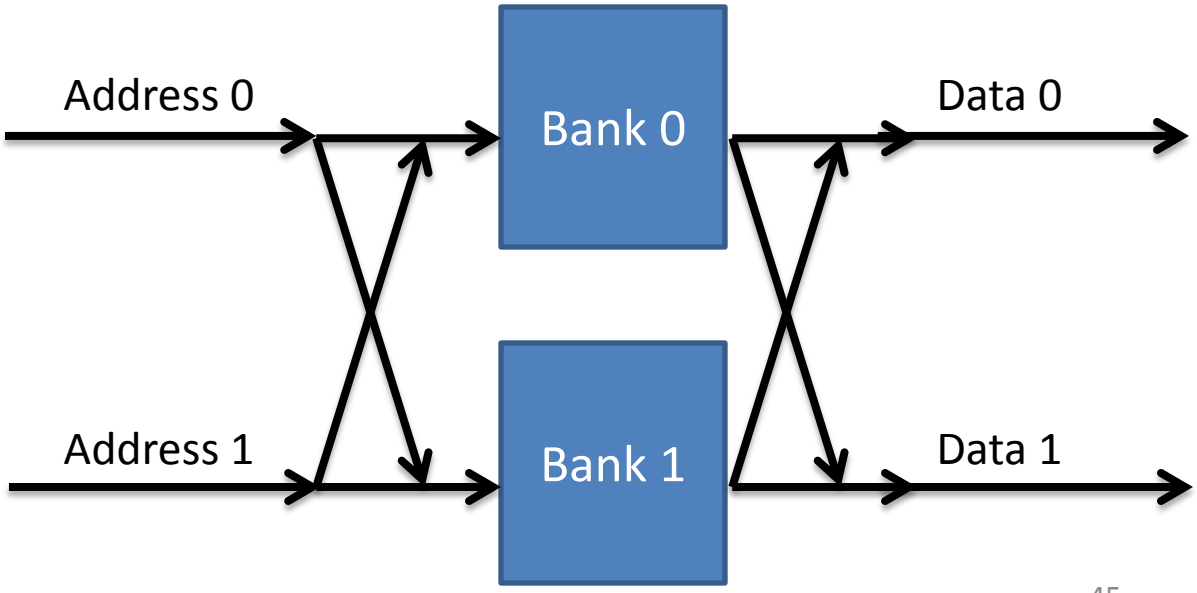
- Partition Address Space into multiple banks
  - Use portions of address (low or high order interleaved)

Benefits:

- Higher throughput

Challenges:

- Bank Conflicts
- Extra Wiring
- Uneven utilization



# Cache Banking Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Cache Banking | | | | + |

# Compiler Optimizations

- Restructuring code affects the data block access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data ("no-allocate" instruction hints or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations

# Loop Interchange

```
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

*What type of locality does this improve?*

# Loop Fusion

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

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

# Loop Fusion

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

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

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

# Loop Fusion

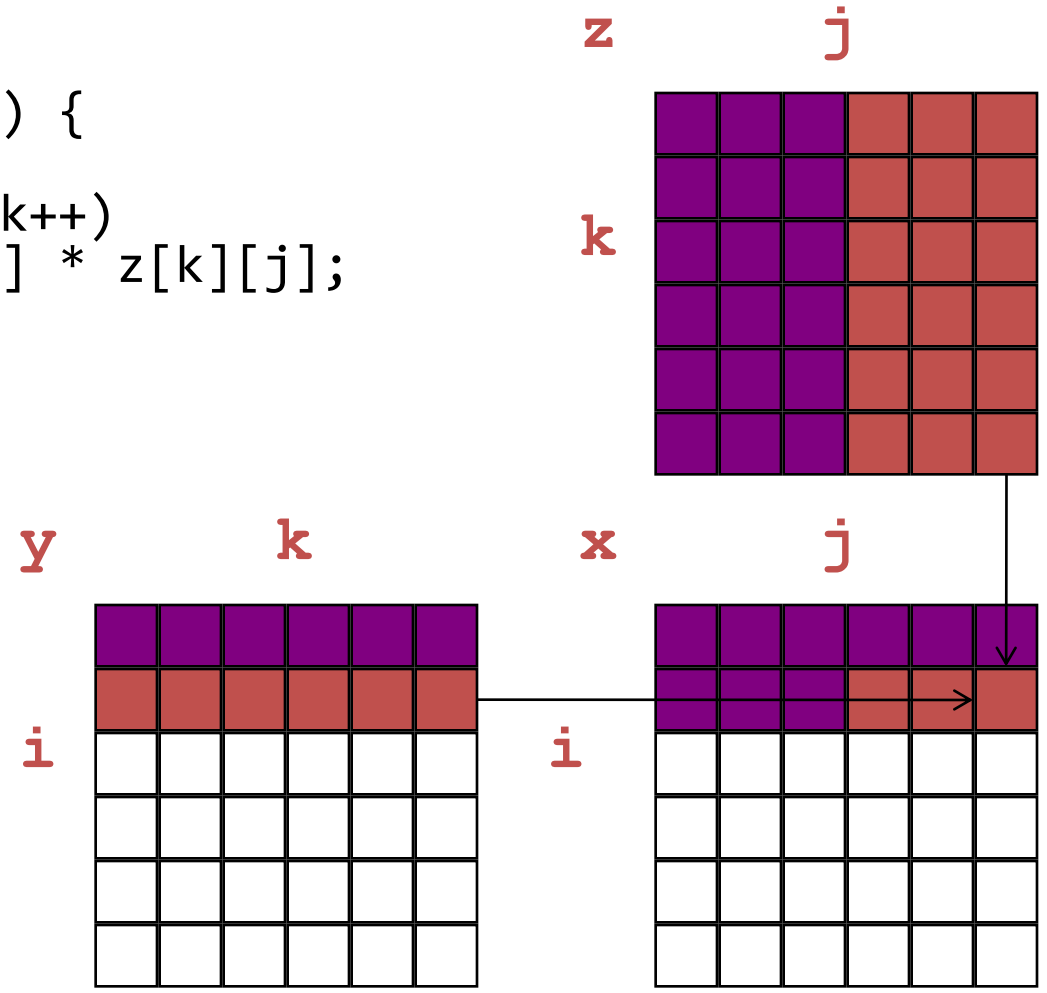
```
for(i=0; i < N; i++)
      a[i] = b[i] * c[i];

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

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

*What type of locality does this improve?*

# Matrix Multiply, Naïve Code

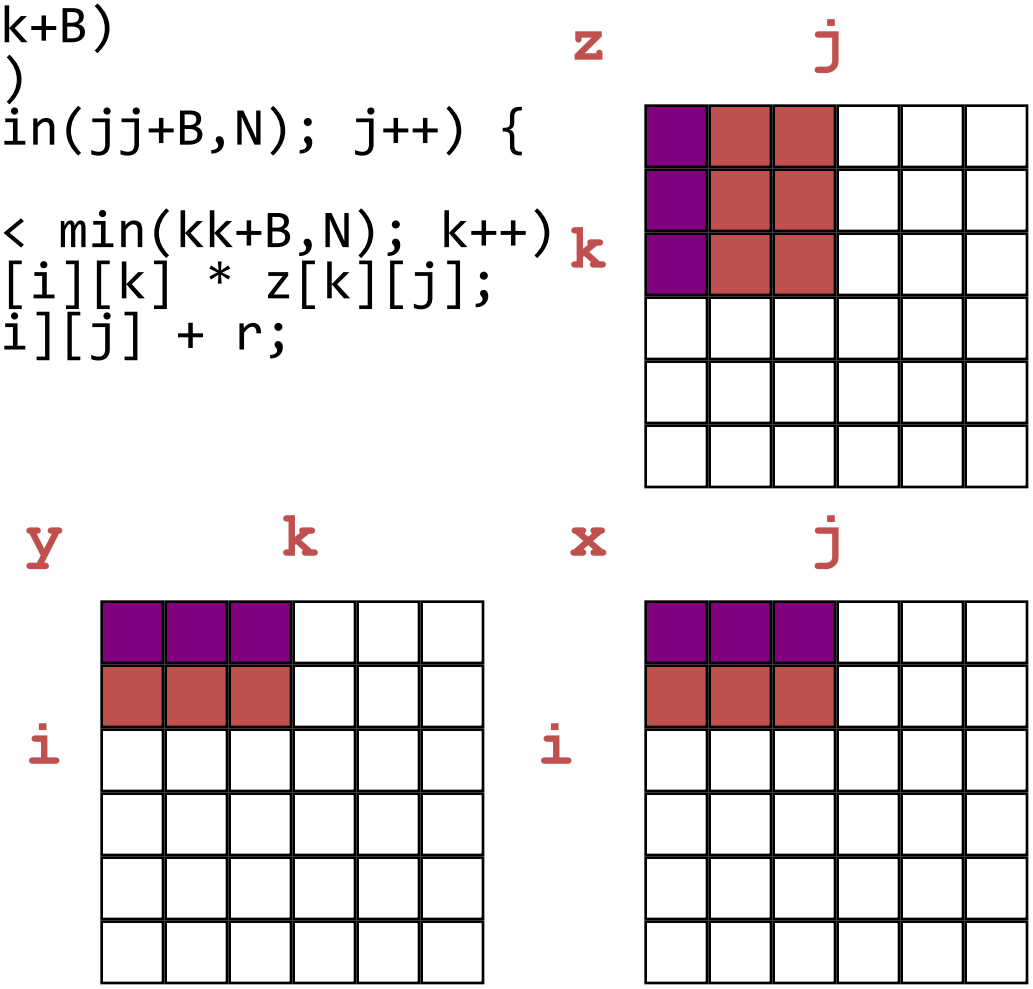
```
for(i=0; i < N; i++)
     for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = r;
     }
```




Not touched  Old access  New access

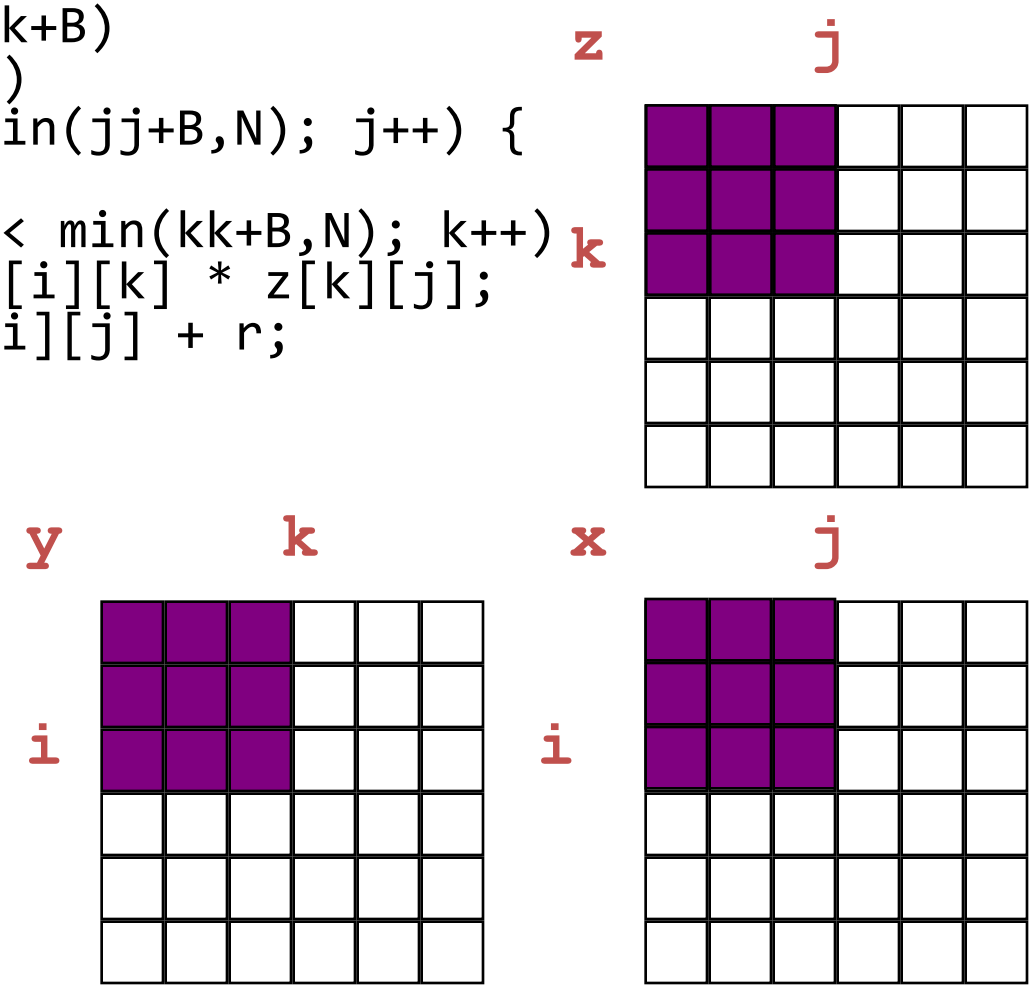
# Matrix Multiply with Cache Tiling/Blocking

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```



# Matrix Multiply with Cache Tiling/Blocking

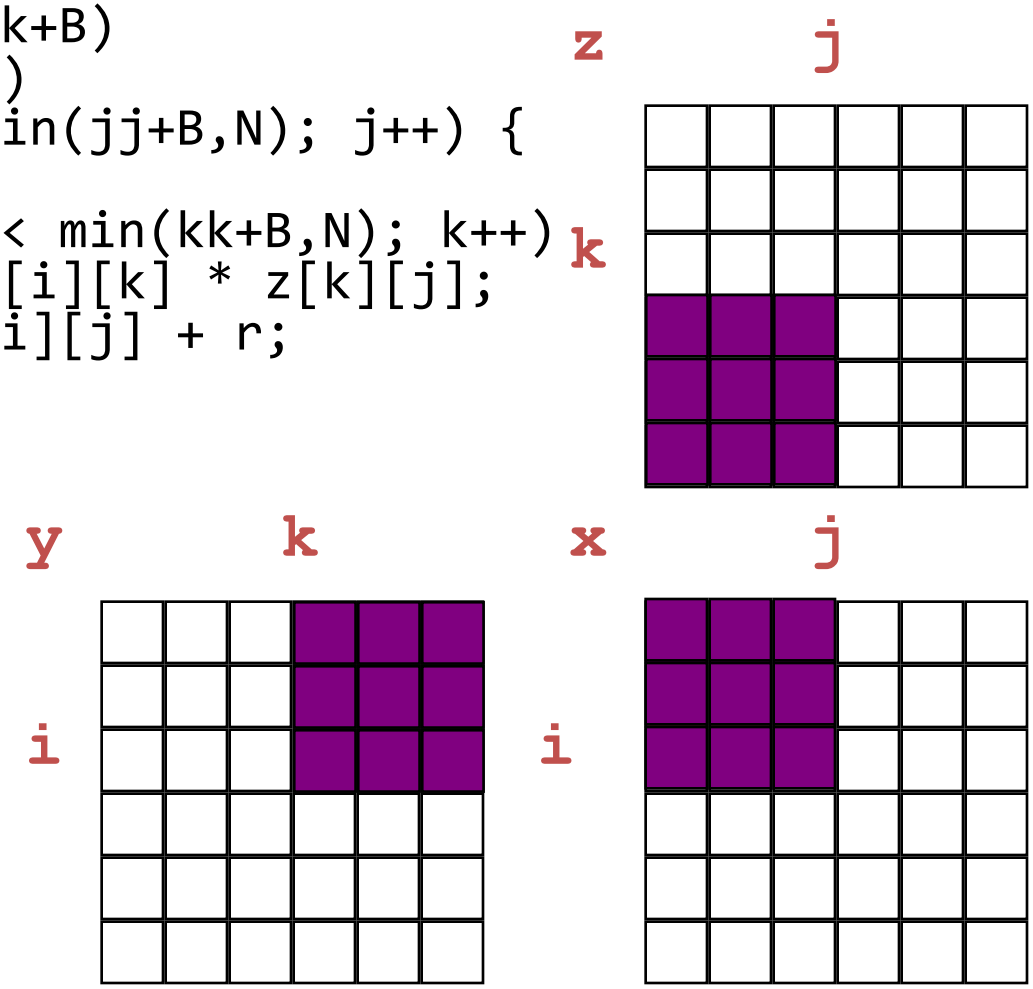
```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
       for(i=0; i < N; i++)
           for(j=jj; j < min(jj+B,N); j++) {
              r = 0;
              for(k=kk; k < min(kk+B,N); k++)
                 r = r + y[i][k] * z[k][j];
              x[i][j] = x[i][j] + r;
           }
```




***What type of locality does this improve?***

# Matrix Multiply with Cache Tiling/Blocking

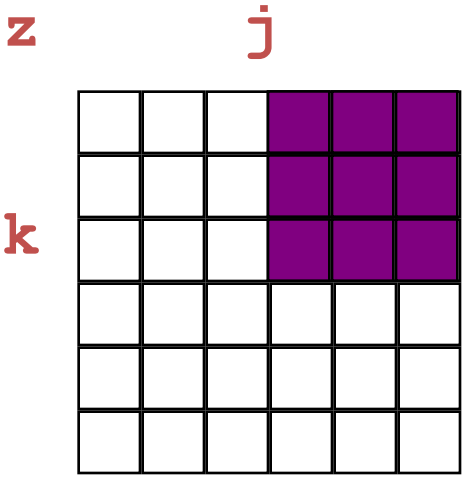
```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```

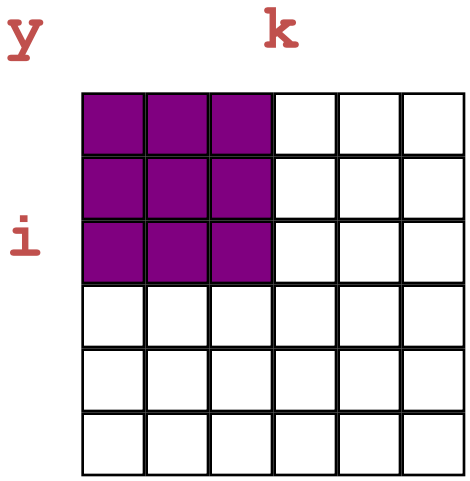


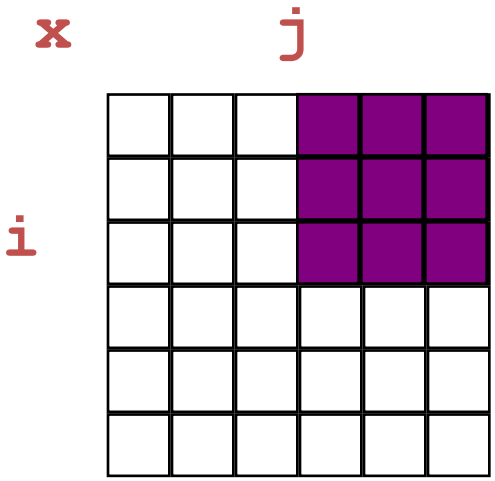

***What type of locality does this improve?***

# Matrix Multiply with Cache Tiling/Blocking

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```



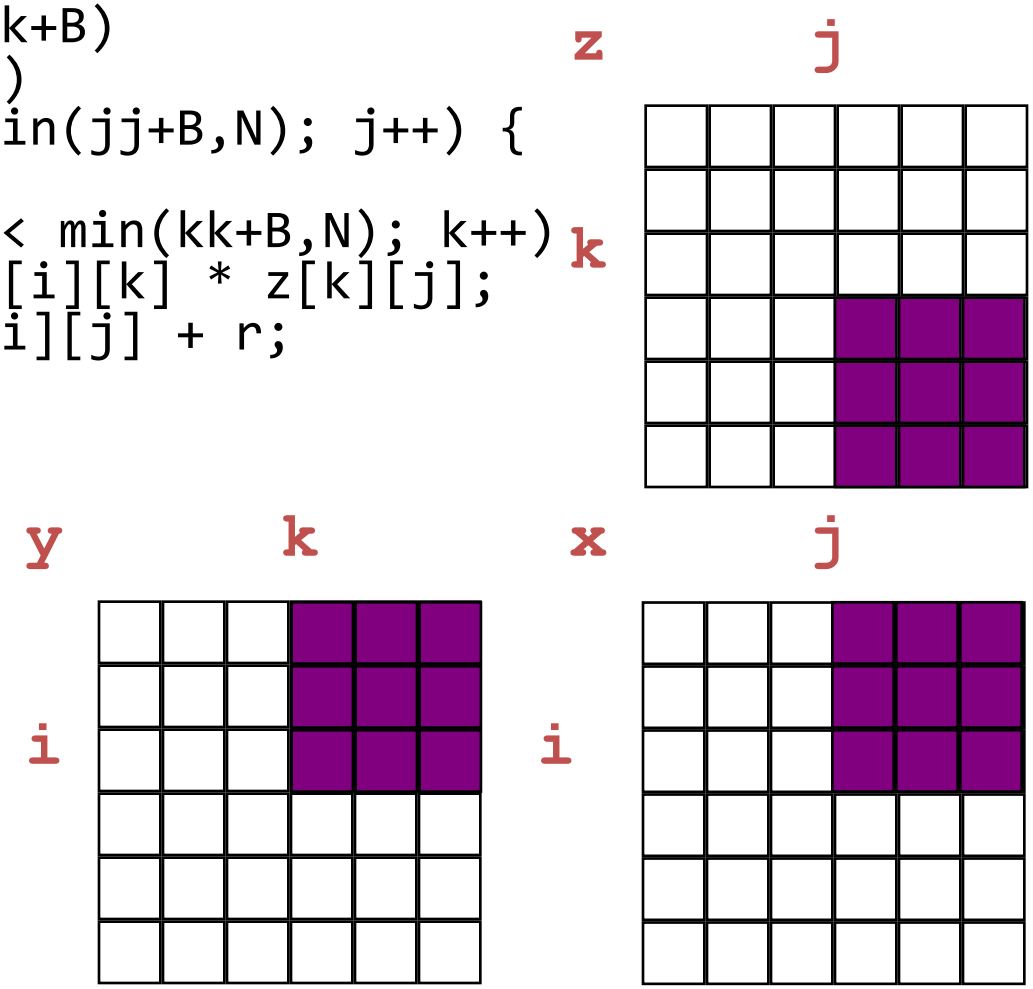




***What type of locality does this improve?***

# Matrix Multiply with Cache Tiling/Blocking

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
       for(i=0; i < N; i++)
           for(j=jj; j < min(jj+B,N); j++) {
              r = 0;
              for(k=kk; k < min(kk+B,N); k++)
                 r = r + y[i][k] * z[k][j];
              x[i][j] = x[i][j] + r;
           }
```




***What type of locality does this improve?***

# Compiler Memory Optimizations Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Compiler Optimization | | | | |

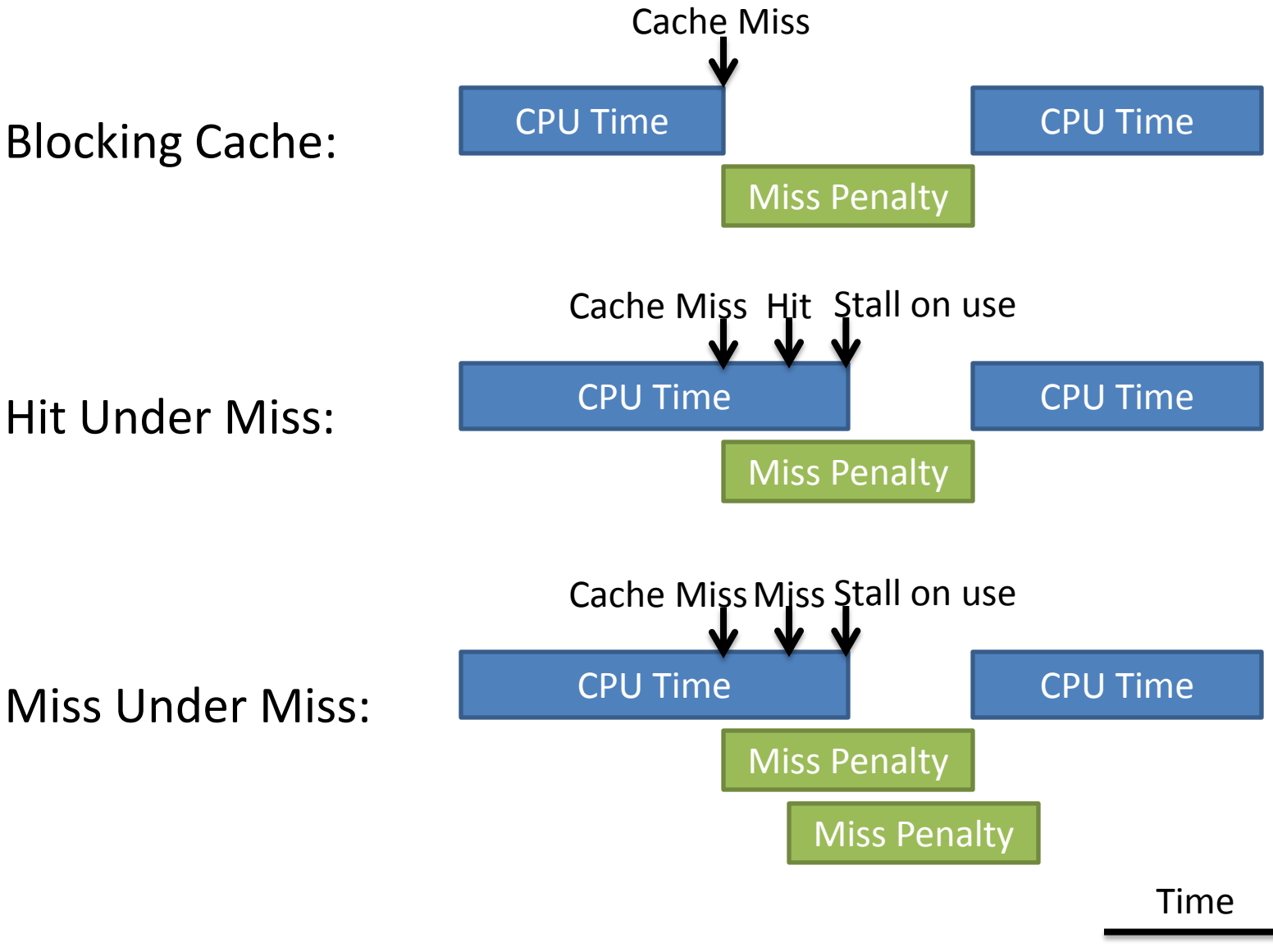
# Compiler Memory Optimizations Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Compiler Optimization | + | | | |

# Non-Blocking Caches (aka Out-Of-Order Memory System) (aka Lockup Free Caches)

- Enable subsequent cache accesses after a cache miss has occurred
  - Hit-under-miss
  - Miss-under-miss (concurrent misses)
- Suitable for in-order processor or out-of-order processors
- Challenges
  - Maintaining order when multiple misses that might return out of order
  - Load or Store to an already pending miss address (need merge)

# Non-Blocking Cache Timeline



# Miss Status Handling Register (MSHR)/ Miss Address File (MAF)

MSHR/MAF

| V | Block Address | Issued |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

**V**: Valid
**Block Address**: Address of cache block in memory system
**Issues:** Issued to Main Memory/Next level of cache

Load/Store Entry

| V | MSHR Entry | Type | Offset | Destination |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

**V**: Valid
**MSHR Entry**: Entry Number
**Type:** {LW, SW, LH, SH, LB, SB}
**Offset:** Offset within the block
**Destination:** (Loads) Register, (Stores) Store buffer entry

# Non-Blocking Cache Operation

On Cache Miss:

- Check MSHR for matched address
  - If found: Allocate new Load/Store entry pointing to MSHR
  - If not found: Allocate new MSHR entry and Load/Store entry
  - If all entries full in MSHR or Load/Store entry table, stall or prevent new LDs/STs

On Data Return from Memory:

- Find Load or Store waiting for it
  - Forward Load data to processor/Clear Store Buffer
  - Could be multiple Loads and Stores
- Write Data to cache

When Cache Lines is Completely Returned:

- De-allocate MSHR entry

# Non-Blocking Cache with In-Order Pipelines

- Need Scoreboard for Individual Registers

On Load Miss:

- Mark Destination Register as Busy

On Load Data Return:

- Mark Destination Register as Available

On Use of Busy Register:

- Stall Processor

# Non-Blocking Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Non-blocking Cache | | | | |

# Non-Blocking Cache Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Non-blocking Cache | | + | | + |

# Critical Word First

- Request the missed word from memory first.
- Rest of cache line comes after "critical word"
  - Commonly words come back in rotated order

Basic Blocking Cache:


Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

Blocking Cache with Critical Word first:


Order of fill: 3, 4, 5, 6, 7, 0, 1, 2

# Early Restart

- Data returns from memory in order
- Processor Restarts when needed word is returned

Basic Blocking Cache:


Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

Blocking Cache with Early Restart:


Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

# Critical Word First and Early Restart Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Critical Word First/Early Restart | | | | |

# Critical Word First and Early Restart Efficacy

| Cache Optimization | Miss Rate | Miss Penalty | Hit Time | Bandwidth |
|---|---|---|---|---|
| Critical Word First/Early Restart | | + | | |

# Agenda

- Review
  - Three C's
  - Basic Cache Optimizations
- Advanced Cache Optimizations
  - Pipelined Cache Write
  - Write Buffer
  - Multilevel Caches
  - Victim Caches
  - Prefetching
    - Hardware
    - Software
  - Multiporting and Banking
  - Software Optimizations
  - Non-Blocking Cache
  - Critical Word First/Early Restart

# Acknowledgements


- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)

- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750