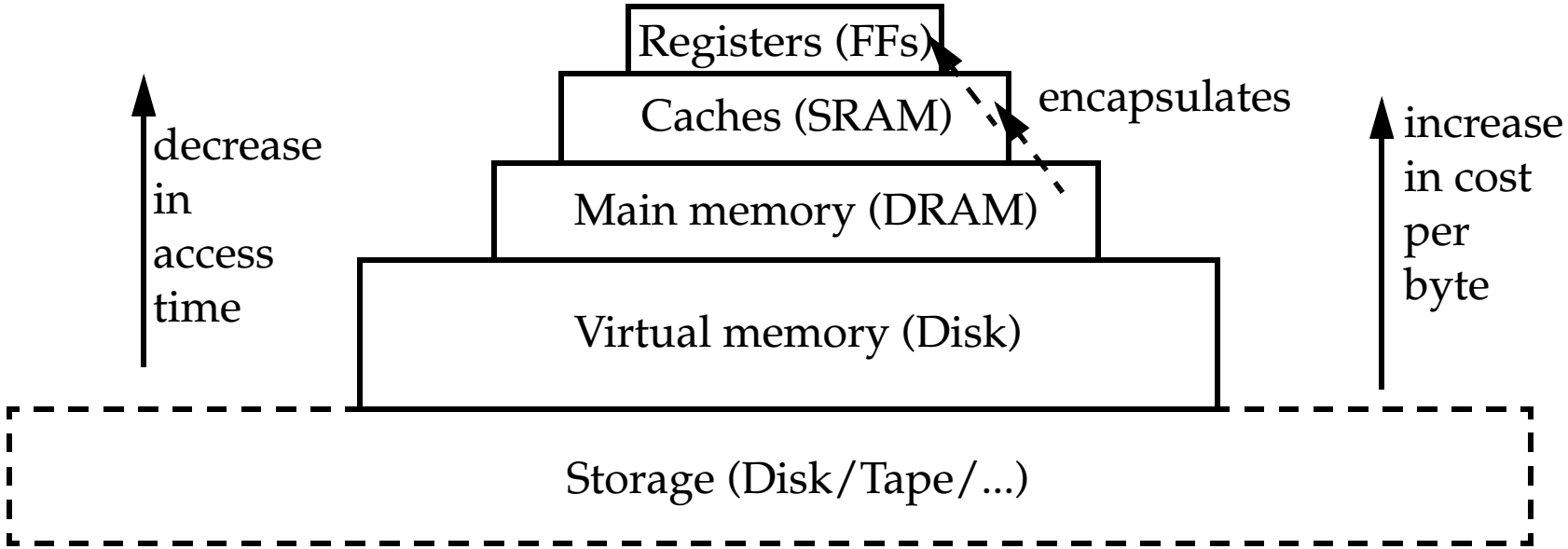
Memory Hierarchy Overview

The principle of locality says that programs do NOT access code and data uniformly.

Also, smaller hardware is faster, and faster hardware is more expensive.

This has led to a **memory hierarchy**:



Performance enhancements are realizable by keeping frequently used code/data in fast memory and the rest in slower memory.

Characterizing Memory Hierarchy

Four questions can be posed about **any 2 levels** of the memory hierarchy:

- *Where can a block be placed in the upper level ?*
  Block placement.

- *How is a block found if it is in the upper level ?*
  Block identification.

- *Which block should be replaced on a miss ?*
  Block replacement.

- *What happens on a write ?*
  Write strategy.

We'll focus on the interface between static and dynamic RAM (the CPU's memory cache) and dynamic RAM and disk (virtual memory).

Memory System Performance

A formula to evaluate the effectiveness of the memory hierarchy:

$$\text{Memory stall cycles} = \text{IC} \times \text{Mem refs per instruction} \times \text{Miss rate} \times \text{Miss penalty}$$

We will use a related formula to evaluate the performance of various memory system configurations.

There are several factors in this equation:

- *IC * Mem refs per instruction*

  This is the frequency with which the CPU uses memory.

  A memory system that need only satisfy 1-2 references per cycle is easier to build than one that satisfies 4-5.

- *Miss rate*

  This is the fraction of references that are not satisfied in the upper level.

  They require an access to the lower, slower level to be satisfied.

- *Miss penalty*

  The penalty is the length of time it takes to access the lower level.

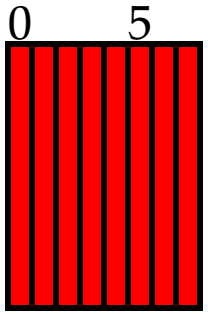
  A low miss rate is not much help if the miss penalty is very high.

Basic Cache Summary (from 411)

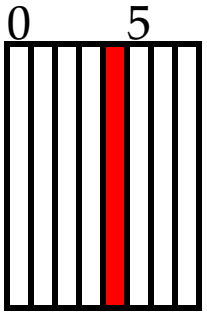
If the term *cache* is used without any modifiers, it usually means the fast memory closest to the CPU.

Recently, *cache* has been used for everything from files to WWW pages.

Block Placement: Three possibilities:

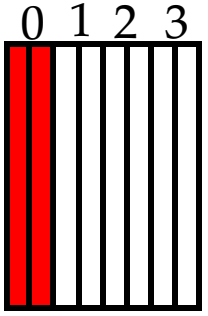


Fully associative



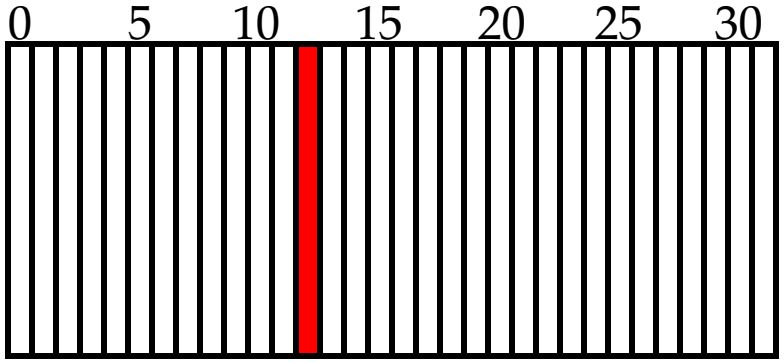
12 mod 8 = 4

Direct mapped



12 mod 4 = 0

Set associative



Basic Cache Summary

- *Direct mapped*

  Block can only go in one place in the cache (usually *address* MOD *number of blocks* in cache).

- *Fully associative*

  Block can go anywhere in cache.

- *Set associative*

  Block can go in one of a **set** of places in the cache.

  A **set** is a group of blocks in the cache.

  In a set-associative cache, a block is first mapped to a set by using *block address* MOD *number of sets* in the cache.

  A block may then be placed anywhere in that set.

  If **sets** have *n* blocks, the cache is said to be *n-way* set associative.

Note that direct mapped is the same as *1-way* set associative, and fully associative is *m-way* set-associative (for a cache with m blocks).

Basic Cache Summary

Block Identification: Finding data in the cache.

Components of an address as they relate to the cache:

Stored in cache and used in comparison with CPU address

Selects set

Selects data within the block

- *Block offset*

  The first few bits of the address give the offset of the byte within a block.

- *Block address (index)*

  Used to pick a set from the cache.

- *Tag*

  Only the tag is stored in the cache.

  All tags within a set are searched in parallel.

- *Valid bit*

  Indicates that the block in this location contains valid data.

  Otherwise, a random sequence of bits could be mistaken for a valid entry that matched the tag.

Basic Cache Summary

Block Replacement: Which block is replaced ?

This only applies to fully associative and set associative caches.

For direct mapped, each block can only go in one location.

- *Random*

  Choose a block from the set at random.

- *LRU*

  Choose the least-recently used block.

  Replace the block that has been unused for the longest time.

  This requires extra bits in the cache to keep track of accesses.

  It turns out that LRU isn't much better than random replacement.

Basic Cache Summary

Write Strategy: What happens on a write ?

All instruction access are reads and most data accesses are reads (DLX, 9% stores and 26% loads).

Making the common case fast means optimizing caches for reads.

The common case is also the easy case to handle since tag checking and reading can occur in *parallel*.

Plus, extra bytes read can be ignored.

However, Amdahl's law reminds us that we cannot ignore writes.

Problem: Tag checking and writing can NOT occur in parallel.

Therefore, writing is usually slower than reading.

Plus, extra bytes can NOT be written.

Basic Cache Summary

Write policy

This determines what happens when a block is written to the cache, and when the write is communicated to the lower level (main memory).

- *Write-through*

  In this scheme, the block is written both to the cache and main memory.

- *Write back* (also *copy back*)

  In this scheme, only the block in cache is modified.

  Main memory is modified when the block must be replaced in the cache.

  This requires the use of a *dirty bit* to keep track of which blocks have been modified.

Write-through adv: Read misses don't result in writes, memory hierarchy is consistent and it is simple to implement.

Write back adv: Writes occur at speed of cache and main memory bandwidth is smaller when multiple writes occur to the same block.

Basic Cache Summary

Write misses

If a miss occurs on a write (the block is not present), there are two options.

- *Write allocate*

  The block is loaded into the cache on a miss before anything else occurs.

- *Write around* (no write allocate)

  The block is only written to main memory

  It is not stored in the cache.

In general, *write-back* caches use *write-allocate*, and *write-through* caches use *write-around*.

This is true in the former case because it is hoped that subsequent writes to that block will be captured by the cache.
In the latter case, subsequent writes to that block will still go to memory.

Basic Cache Summary

Write buffers

To avoid stalling on writes, many CPUs use a write buffer.

A small cache that can hold a few values waiting to go to main memory.

This buffer helps when writes are clustered.

It does not entirely eliminate stalls since it is possible for the buffer to fill if the burst is larger than the buffer.

Write merging

Blocks are often larger than a machine word.

Many write buffers can *merge* memory writes to save both write buffer space and memory traffic.

For example, two writes to the same location can be collapsed or, two writes to sequential locations can be merged into a single buffer space.

Basic Cache Summary

Split vs. unified caches

- *Unified cache*

  All memory requests go through a single cache.

  This requires less hardware, but also has lower bandwidth and more opportunity for collisions.

- *Split I & D cache*

  A separate cache is used for instructions and data.

  This uses additional hardware, though there are some simplifications (the I cache is read-only).

Cache Performance

Average memory access time is a useful measure to evaluate the performance of a memory-hierarchy configuration.

$$\text{Avg mem access time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

It tells us how much penalty the memory system imposes on each access (on average).

It can easily be converted into clock cycles for a particular CPU.

But leaving the penalty in nanoseconds allows two systems with *different clock cycles times* to be compared to a single memory system.

Cache Performance

There may be different penalties for Instruction and Data accesses.

In this case, you may have to compute them separately.

This requires knowledge of the fraction of references that are instructions and the fraction that are data.

The text gives 75% instruction references to 25% data references.

We can also compute the write penalty separately from the read penalty.

This may be necessary for two reasons:

- Miss rates are different for each situation.
- Miss penalties are different for each situation.

Treating them as a single quantity yields a useful CPU time formula:

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

An Example

Compare the performance of a **64KB unified** cache with a split cache with **32KB data** and **16KB instruction**.

The miss penalty for either cache is 100 ns, and the CPU clock runs at 200 MHz.

Don't forget that the cache requires an extra cycle for load and store hits on a unified cache because of the structural conflict.

Calculate the effect on **CPI** rather than the average memory access time.

Assume miss rates are as follows (Fig. 5.7 in text):

- 64K Unified cache: 1.35%
- 16K instruction cache: 0.64%
- 32K data cache: 4.82%

Assume a data access occurs once for every 3 instructions, on average.

An Example

The solution is to figure out the penalty to CPI separately for instructions and data.

First, we figure out the miss penalty in terms of clock cycles: 100 ns/5 ns = 20 cycles.

For the unified cache, the per-instruction penalty is (0 + 1.35% x 20) = **0.27** cycles.

For data accesses, which occur on about 1/3 of all instructions, the penalty is (1 + 1.35% x 20) = 1.27 cycles per access, or **0.42** cycles per instruction.

The total *penalty* is **0.69 CPI**.

In the split cache, the per-instruction penalty is (0 + 0.64% x 20) = **0.13** CPI.

For data accesses, it is (0 + 4.82% x 20) x (1/3) = **0.32** CPI.

The total *penalty* is **0.45 CPI**.

In this case, the split cache performs better because of the lack of a stall on data accesses.

Effects of Cache Performance on CPU Performance

- Low CPI machines suffer more relative to some fixed CPI memory penalty.
  A machine with a CPI of 5 suffers little from a 1 CPI penalty.
  However, a processor with a CPI of 0.5 has its execution time *tripled* !

- Cache miss penalties are measured in cycles, not nanoseconds.
  This means that a faster machine will stall more cycles on the same memory system.

Amdahl's Law raises its ugly head again:
  Fast machines with low CPI are affected significantly from memory access penalties.

Improving Cache Performance

The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important.

15 distinct organizations characterize the effort of system architects in reducing average memory access time.

These organizations can be distinguished by:

- Reducing the miss rate.
- Reducing the miss penalty.
- Reducing the time to hit in a cache.

Reducing Cache Misses

Components of miss rate: All of these factors may be reduced using various methods we'll talk about.

- *Compulsory*

  Cold start misses or **first reference misses**: The first access to a block can NOT be in the cache, so there must be a compulsory miss.

  These are suffered regardless of cache size.

- *Capacity*

  If the cache is too small to hold all of the blocks needed during execution of a program, misses occur on blocks that were discarded earlier.

  In other words, this is the difference between the compulsory miss rate and the miss rate of a finite size fully associative cache.

- *Conflict*

  If the cache has sufficient space for the data, but the block can NOT be kept because the set is full, a conflict miss will occur.

  This is the difference between the miss rate of a non-fully associative cache and a fully-associative cache.

  These misses are also called **collision** or **interference** misses.

Reducing Cache Miss Rate

To reduce cache miss rate, we have to eliminate some of the misses due to the three C's.

We cannot reduce *capacity* misses much except by making the cache larger.

We can, however, reduce the *conflict* misses and *compulsory* misses in several ways:

- *Larger cache blocks*

  Larger blocks decrease the compulsory miss rate by taking advantage of spatial locality.

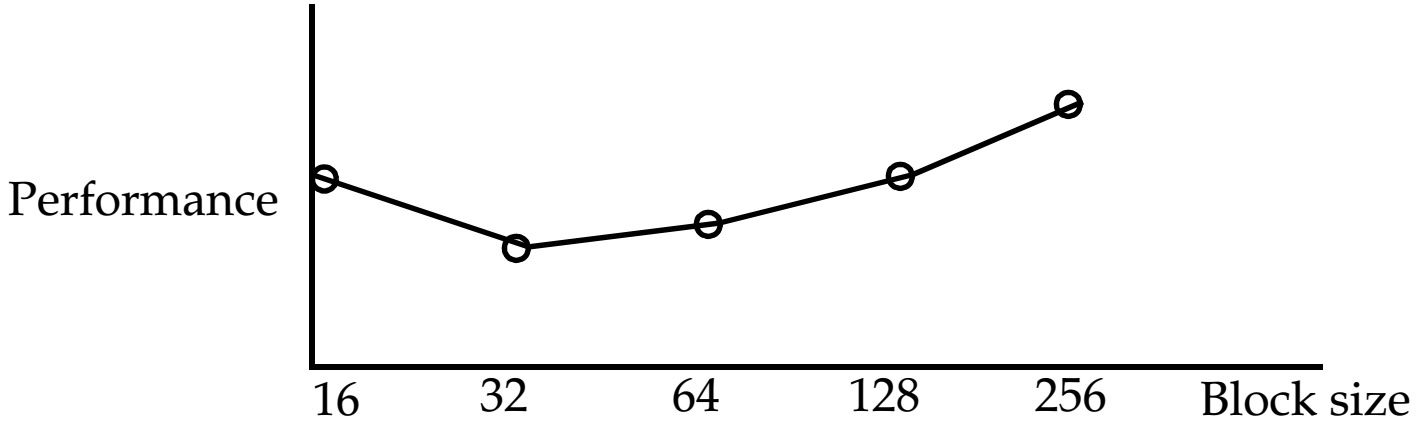
  However, they may increase the miss penalty by requiring more data to be fetched per miss.

  In addition, they will almost certainly increase *conflict* misses since fewer blocks can be stored in the cache.

  And maybe even *capacity* misses in small caches.

Reducing Cache Miss Rate

- *Larger cache blocks*



The performance curve is U-shaped because:

Small blocks have a higher miss rate and

Large blocks have a higher miss penalty (even if miss rate is not too high).

High latency, high bandwidth memory systems encourage large block sizes since the cache gets more bytes per miss for a small increase in miss penalty.

32-byte blocks are typical for 1-KB, 4-KB and 16-KB caches while 64-byte blocks are typical for larger caches.

Reducing Cache Miss Rate

- *Higher associativity*

Conflict misses can be a problem for caches with low associativity (especially direct-mapped).

2:1 cache rule of thumb: a direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size N/2.

However, there is a limit -- higher associativity means more hardware and usually longer cycle times (increased hit time).

In addition, it may cause more capacity misses.

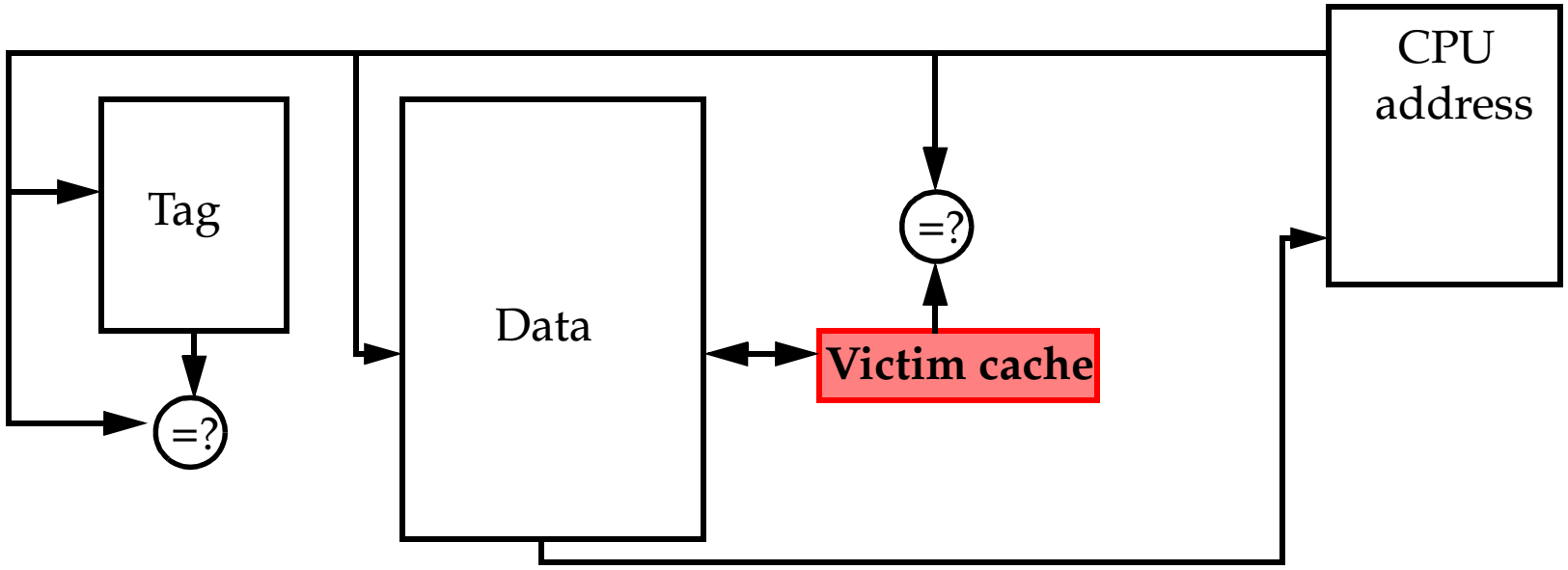
Nobody uses more than 8-way set-associative caches today, and most systems use 4-way or less.

The problem is that the higher hit rate is offset by the slower clock cycle time.

Reducing Cache Miss Rate

- *Victim caches*

A victim cache is a small (usually, but not necessarily) fully-associative cache that holds a few of the most recently replaced blocks or **victims** from the main cache.



Can improve miss rates without affecting the processor clock rate.

This cache is checked on a miss before going to main memory.
If found, the victim block and the cache block are swapped.

Reducing Cache Miss Rate

- *Victim caches*

  It can reduce *capacity* misses but is best at reducing *conflict* misses.

  It's particularly effective for small, direct-mapped data caches.

  A 4 entry victim cache handled from 20% to 95% of the conflict misses from a 4KB direct-mapped data cache.

- *Pseudo-associative caches*

  These caches use a technique similar to double hashing.

  On a miss, the cache searches a different *set* for the desired block.

  The second (*pseudo*) set to probe is usually found by inverting one or more bits in the original set **index**.

  Note that two separate searches are conducted on a miss.

  The first search proceeds as it would for direct-mapped cache.

  Since there is no associative hardware, hit time is fast if it is found the first time.

Reducing Cache Miss Rate

- *Pseudo-associative caches*

  While this second probe takes some time (usually an extra cycle or two), it is a lot faster than going to main memory.

  The secondary block can be swapped with the primary block on a "slow hit".

  This method reduces the effect of *conflict* misses.

  Also improves miss rates without affecting the processor clock rate.

- *Hardware prefetch*

  Prefetching is the act of getting data from memory before it is actually needed by the CPU.

  Typically, the cache requests the next consecutive block to be fetched with a requested block.

  It is hoped that this avoids a subsequent miss.

Reducing Cache Miss Rate

- *Hardware prefetch*

  This reduces *compulsory* misses by retrieving the data before it is requested.

  Of course, this may increase other misses by removing useful blocks from the cache.

  Thus, many caches hold prefetched blocks in a **special buffer** until they are actually needed.

  This buffer is faster than main memory but only has a limited capacity.

  Prefetching also uses main memory bandwidth.

  It works well if the data is actually used.

  However, it can adversely affect performance if the data is rarely used and the accesses interfere with 'demand misses'.

Reducing Cache Miss Rate

- *Compiler-controlled prefetch*

An alternative to hardware prefetching.

Some CPUs include *prefetching instructions*.

These instructions request that data be moved into either a register or cache.

These special instructions can either be *faulting* or *non-faulting*.

Non-faulting instructions do nothing (no-op) if the memory access would cause an exception.

Of course, prefetching does **not** help if it interferes with normal CPU memory access or operation.

Thus, the cache must be **nonblocking** (also called **lockup-free**).

This allows the CPU to overlap execution with the prefetching of data.

Reducing Cache Miss Rate

- *Compiler-controlled prefetch*

  While this approach yields better prefetch "hit" rates than hardware prefetch, it does so at the expense of executing more instructions.

  Thus, the compiler tends to concentrate on prefetching data that are likely to be cache misses anyway.

  Loops are key targets since they operate over large data spaces and their data accesses can be inferred from the loop index in advance.

- *Compiler optimizations*

  This method does NOT require any hardware modifications.

  Yet it can be the most efficient way to eliminate cache misses.

  The improvement results from better code and data organizations.

  For example, code can be rearranged to avoid conflicts in a direct-mapped cache, and accesses to arrays can be reordered to operate on **blocks of data** rather than processing **rows** of the array.

Reducing Cache Miss Rate

- *Compiler optimizations*

Merging arrays

This method combines two separate arrays (that might conflict for a single block in the cache) into a single interleaved array.

```
int val[SIZE];
int key[SIZE];
```

```
struct merge {
    int val;
    int key;
    };
struct merge MA[SIZE];
```

This brings together corresponding elements in both arrays, which are likely to be referenced together.

Reorganizing and fetching them at the same time can reduce misses.

This technique reduces misses by improving spatial locality.

Reducing Cache Miss Rate

- *Compiler optimizations*

Loop interchange

By switching the order in which loops execute, misses can be reduced due to improvements in spatial locality.

For example,

```
for (i = 0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
     a[j][i] = a[j][i] * 2;
     }
  }
```

These loops cause a miss on each memory access because of the long *stride* given by index j in the inner loop.

By switching the order of the loops, the stride is changed to 1, allowing the elements to be accessed in sequential order.

Reducing Cache Miss Rate

- *Compiler optimizations*

Loop Fusion

Many programs have separate loops that operate on the same data.

Combining these loops allows a program to take advantage of **temporal locality** by grouping operations on the same (cached) data together.

Blocking

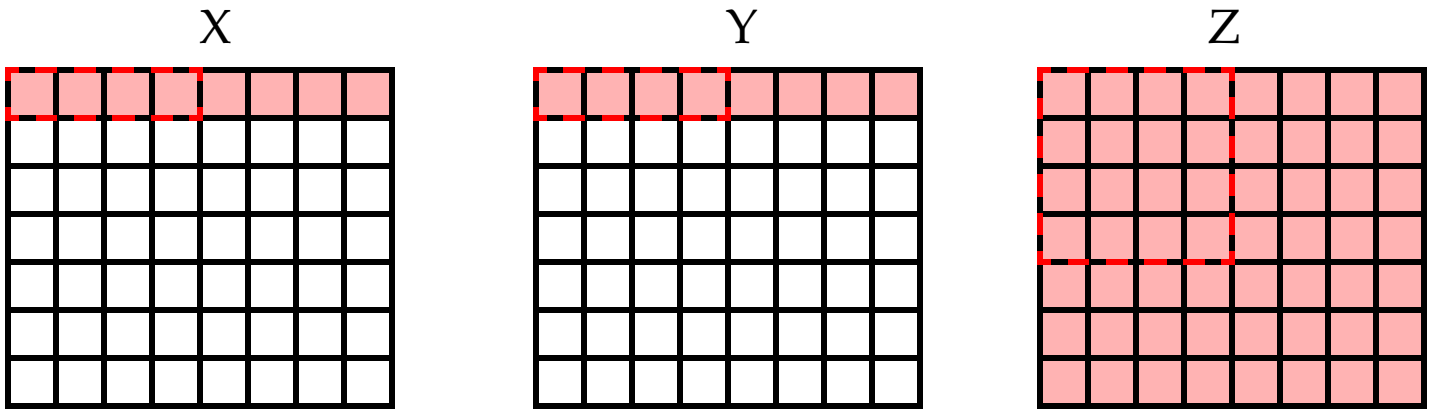
The above methods work well on array accesses that occur along one dimension only.

However, loops that access both rows and columns, such as matrix multiplication, are problems.

Reducing Cache Miss Rate

Blocking

Unoptimized matrix multiplication requires the cache to hold the minimum elements shown shaded below.



Data accessed to compute a row of X using X = Y*Z

Capacity misses can occur for large matrices since it may not be possible to store all the elements of Z in the cache.

Blocking operates on blocks (submatrices) as shown by the dotted line, and reduces the total number of memory words accessed by a factor of B (the blocking factor).

Reducing Cache Miss Rate

- *Compiler optimizations*

Blocking

Therefore, matrix multiplication is performed by multiplying the submatrices first.

Matrix Y benefits from spatial locality and Z benefits from temporal locality.

This method is also used to reduce the number of blocks that must be transferred between disk and main memory.
Therefore, the technique is effective for several levels of the hierarchy.

Given the increasing speed gap in processor speed and memory access times, these last two techniques will only increase in importance over time.

Reducing Cache Miss Penalty

- *Giving read misses priority*

If a system has a write buffer, writes can be delayed to come after reads.

The system must, however, be careful to check the write buffer to see if the value being read is about to be written.

```
SW 512(R0), R3
LW R1, 1024(R0)
LW R2, 512(R0)
```

Assume the write-through cache maps 512 to the same cache location as 1024.
Will R2 = R3 ?

A simple method of dealing with this problem:
Stall reads until the write buffer is empty.

However, this method increases the read miss penalty considerably since the write buffer in write-through is likely to have blocks waiting to be written.

An alternative is to check the write buffer for conflicts.
In cases like this, the write buffer acts as a victim cache.

Reducing Cache Miss Penalty

- *Using subblocks to reduce fetch time*

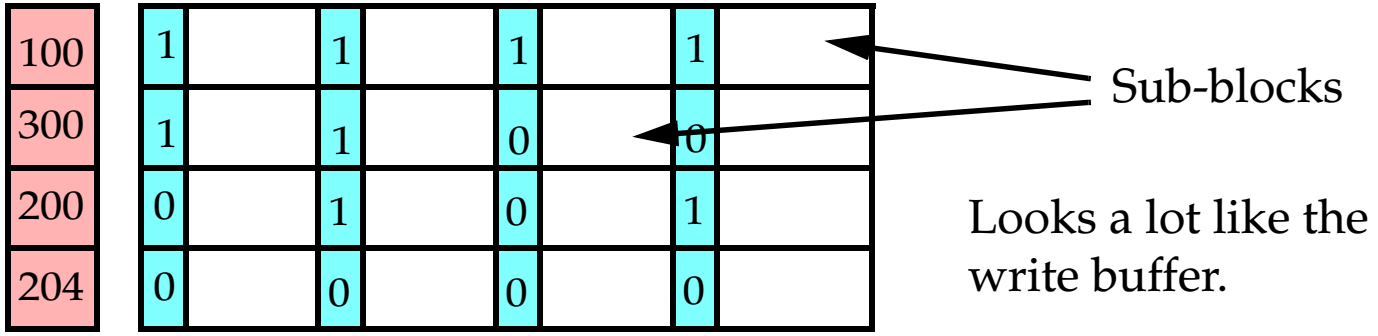
Tags can hurt performance by occupying too much space or by slowing down caches.

Using large blocks reduces the amount of storage for tags (and makes them shorter), optimizing space on the chip.

This may even reduce **miss rate** by reducing compulsory misses.

However, the *miss penalty* for large blocks is high, since the entire block must be moved between the cache and memory.

The solution is to divide each block into subblocks, each of which has a valid bit.



Reducing Cache Miss Penalty

- *Using subblocks to reduce fetch time*

  The tag is valid for the entire block, but only a sub-block needs to be read on a miss.

  Therefore, a block can no longer be defined as the minimum unit transferred between cache and memory.

  This results in a smaller miss penalty.

- *Early restart & critical word first*

  This strategy does NOT require extra hardware (like the previous two techniques).

  It optimizes the order in which the words of a block are fetched and when the desired word is delivered to the CPU.

  - *Early restart*

  With early restart, the CPU gets its data (and thus resumes execution) as soon as it arrives in the cache without waiting for the rest of the block.

Reducing Cache Miss Penalty

- *Early restart & critical word first*
  - *Critical word first*

  Instead of starting the fetch of a block with the first word, the cache can fetch the requested word first and then fetch the rest afterwards.

  In conjunction with *early restart*, this reduces the miss penalty by allowing the CPU to continue execution while most of the block is still being fetched.

- *Nonblocking caches*

  A **nonblocking** cache, in conjunction with out-of-order execution, can allow the CPU to continue executing instructions after a data cache miss.

  The cache continues to supply hits while processing read misses (*hit under miss*).

  The instruction needing the missed data waits for the data to arrive.

Reducing Cache Miss Penalty

- *Nonblocking caches*

  Complex caches can even have multiple outstanding misses (*miss under miss*).

  But this greatly increases cache complexity.

- *Second-level caches*

  This method focuses on the interface between the cache and main memory.

  We can add an second-level cache between main memory and a small, fast first-level cache.

  This helps satisfy the desire to make the cache fast and large.

  The second-level cache allows:

  The smaller first-level cache to fit on the chip with the CPU and fast enough to service requests in one or two CPU clock cycles.

  Hits for many memory accesses that would go to main memory, lessening the effective miss penalty.

Reducing Cache Miss Penalty

- *Second-level caches*

Performance of a multi-level cache:

The performance of a two-level cache is calculated in a similar way to the performance for a single level cache.

$$\text{Avg mem access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

So the miss penalty for level 1 is calculated using the hit time, miss rate, and miss penalty for the level 2 cache.

For two level caches, there are two miss rates:

- *Global miss rate*

  The number of misses in the cache divided by the total number of memory accesses generated by the CPU (Miss rate_{L1}*Miss $\text{rate}_{L2)}$.

- *Local miss rate*

  The number of misses in the cache divided by the total number of memory accesses to this cache (Miss rate_{L2} for the 2nd-level cache).

Reducing Cache Miss Penalty

Performance of a multi-level cache:

Note that the local miss rate for L2 is high because it's only getting the misses from the L1 cache (instead of all memory accesses).

In general, the global miss rate is a more useful measure since it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.

Desirable characteristics for an L2 cache:

- *Much larger than the L1 cache*

  Since L2 contains the same data as L1, making L2 about the same size as L1 causes it to have a high local miss rate.

  This is true since if we miss in L1, it is likely that we'll miss in L2 as well resulting in performance that is not much better than using main memory alone.

  Therefore, it must be much larger.

Reducing Cache Miss Penalty

Desirable characteristics for an L2 cache:

- *Higher associativity*

  The main reason for low associativity was fast, small caches.

  The L2 cache need be **neither**, and will benefit from the higher hit rate that more blocks per set provides.

- *Larger block size*

  This has the advantage of reducing compulsory misses that must go all the way to main memory.

  Since the L2 cache is large, the effect of increasing conflict misses (as is true for a smaller cache) is minimal.

Reducing Cache Miss Penalty

Inclusion

If all of the data in the L1 cache is also in the L2 cache, the L2 cache has the *multilevel inclusion property*.

Most caches enforce this property since it is easier to deal with cache consistency.

Consistency between I/O and caches (and between caches in a multiprocessor) can be determined by checking second-level cache.

Design of L1 and L2 caches

Although they can be designed separately, it is often helpful to know if there is going to be an L2 cache.

For example, *write-through* in L1 is much more effective if there is an L2 *writeback* cache to buffer repeated writes.

Similarly, a direct-mapped L1 cache can work fine if the L2 cache satisfies most of the conflict misses.

Reducing Cache Miss Penalty

L2 cache summary

In general, cache design trades fast hits for few misses.

For an L1 cache, fast hits are more important.

For L2, there are many fewer hits, so fewer misses becomes more important.

Therefore, *larger* caches with *higher associativity* and *larger blocks* are beneficial in L2 caches.

Reducing hit time

On many machines, cache access time limits the clock cycle rate !

Therefore, cache design affects more than average memory access time, it affects everything.

- *Small & simple caches*

  The less hardware that is necessary to implement a cache, the shorter the critical path through the hardware.

  Direct-mapped is faster than set associative for both reads and writes.

  In particular, tag checking can overlap data transmission (there is only one piece of data for each index).

  Fitting the cache on the chip with the CPU is also very important for fast access times.

  Therefore, fast clock cycle time encourages small direct-mapped caches.

Reducing hit time

- *Avoid address translation during indexing*

The CPU uses virtual addresses that must be mapped to a physical address.

The cache may either use virtual or physical addresses.

A cache that indexes by virtual addresses is called a *virtual cache*, as opposed to a *physical cache*.

A virtual cache reduces hit time since a translation from a virtual address to a physical address is not necessary on hits.

Also, address translation can be done in parallel with cache access, so penalties for misses are reduced as well.

So why doesn't anyone use them ?

Reducing hit time

- *Avoid address translation during indexing*

Virtual cache difficulties include:

- *Process switches require cache purging*

In virtual caches, different processes share the same virtual addresses even though they map to different physical addresses.

When a process is swapped out, the cache must be purged of all entries to make sure that the new process gets the correct data.

One solution: **PID tags**

Increase the width of the cache address tags to include a process ID (instead of purging the cache.)

The current process PID is specified by a register.

If the PID does not match, it is **not** a hit even if the address matches.

Reducing hit time

Virtual cache difficulties include:

- *Aliasing*

Two **different virtual** addresses may have the **same physical** address.

In such a case, it is possible to end up with two copies of the same block !

Anti-aliasing hardware

A hardware solution called **anti-aliasing** guarantees every cache block a unique physical address.

Every virtual address maps to the same location in the cache.

Page coloring

This software technique forces aliases to share some address bits.

Therefore, the virtual address and physical address match over these bits.

A *direct-mapped cache* that is 2^k bytes (where k is the number of matching bits) or smaller can **never** have duplicate physical addresses for blocks.

Reducing hit time

Virtual cache difficulties include:

- *Aliasing*

Using the page offset

An alternative to get the best of both virtual and physical caches.

If we use the page offset to index the cache, then we can overlap the virtual address translation process with the time required to **read** the tags.

Note that the page offset is unaffected by address translation.

However, this restriction forces the cache size to be smaller than the page size.

Since the index comes only from the "physical" portion of the virtual address (the page offset).

After doing both in parallel, the tag is checked against the **physical** address stored in the cache.

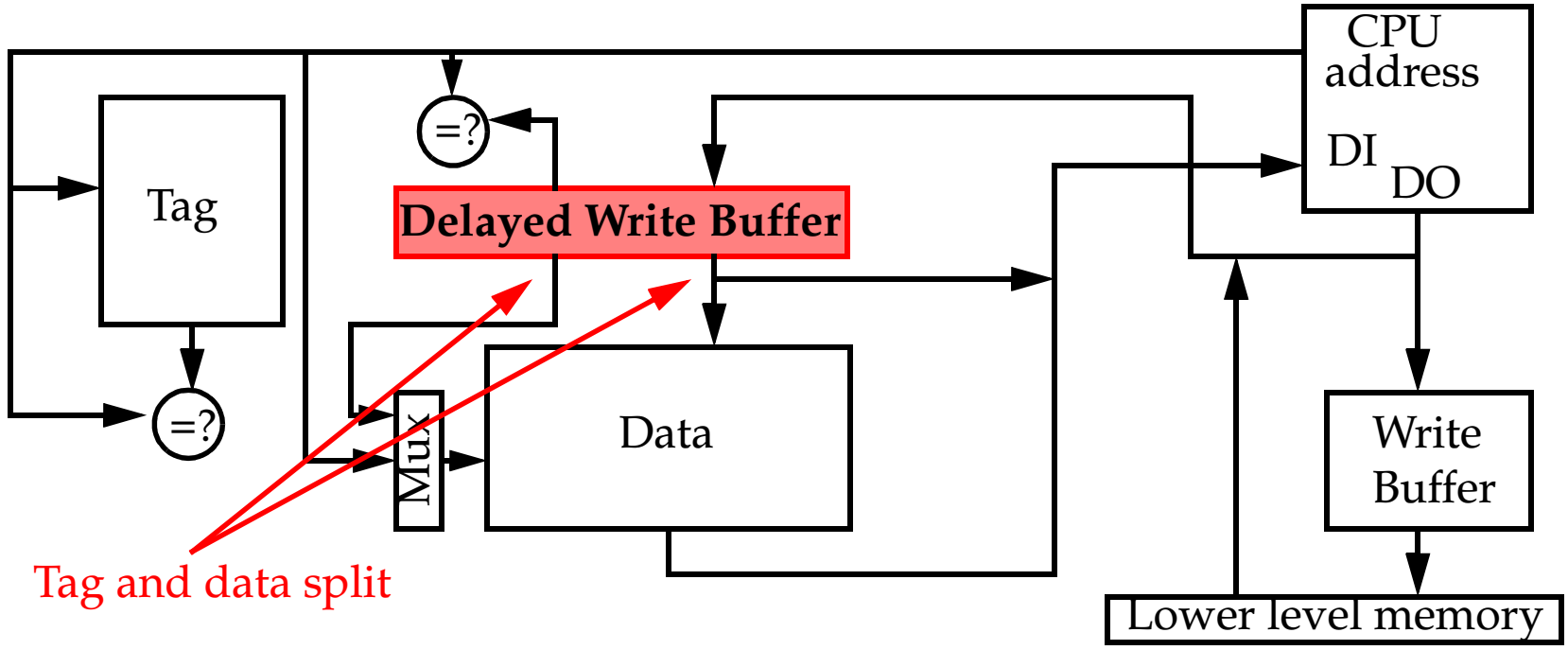
High associativity allows for larger cache sizes.

Reducing hit time

- *Pipelined writes*

Write hits take longer than read hits because tag checking is required before the data is written.

One solution is to pipeline the writes (Alpha AXP 21064):



The second stage of the write (cache is updated with new data) occurs during the first stage of the next write.

Allows tag checking and data writing to occur simultaneously.

Cache Optimization Summary

Technique	Miss Rate	Miss Pen.	Hit time	Hardware Complexity
Larger Block Size	+	-		0
Higher Associativity	+		-	1
Victim Caches	+			2
Pseudo-associative	+			2
Hardware Prefetching	+			2
Compiler-controlled Pre	+			3
Compiler Techniques	+			0
Giving Read Misses Priority		+		1
Subblock Placement		+		1
Early Restart/Crit Wd First		+		2
Nonblocking Caches		+		3
Second-Level Caches		+		2
Small and Simple Caches	-		+	0
Avoiding Address Trans.			+	2
Pipelining Writes			+	1

Main Memory

Main memory is usually made from DRAM while caches use SRAM.

SRAM is faster (by almost an order of magnitude).

However, it's also more expensive per bit and 1/4 to 1/8 as dense as DRAM (1 transistor versus 6 transistors).

We now turn to optimizing DRAM performance.

Performance measures include:

- *Latency*

  Important for caches.

- *Bandwidth*

  Important for I/O.

  Also for cache with second-level and larger block sizes.

Improving Main Memory Performance

Latency measures:

- *Access time*

  Time between when a read is requested and when the desired word arrives.

- *Cycle time*

  This is the minimum time between the starts of two accesses to memory.
  This is at least as long as access time, and is usually longer.

- *Refresh time*

  DRAMs must occasionally refresh their data.

  This is done by reading all of the cells in a row and writing them back.

  This must be done every few milliseconds.

  However, this operation consumes less than 5% of total time.

  This is true because the time necessary to refresh is proportional to the **square root** of the size of the DRAM.

Improving Main Memory Performance

Amdahl suggested that memory capacity should grow linearly with CPU speed.

Memory capacity grows **four-fold** every **three** years to supply this demand.

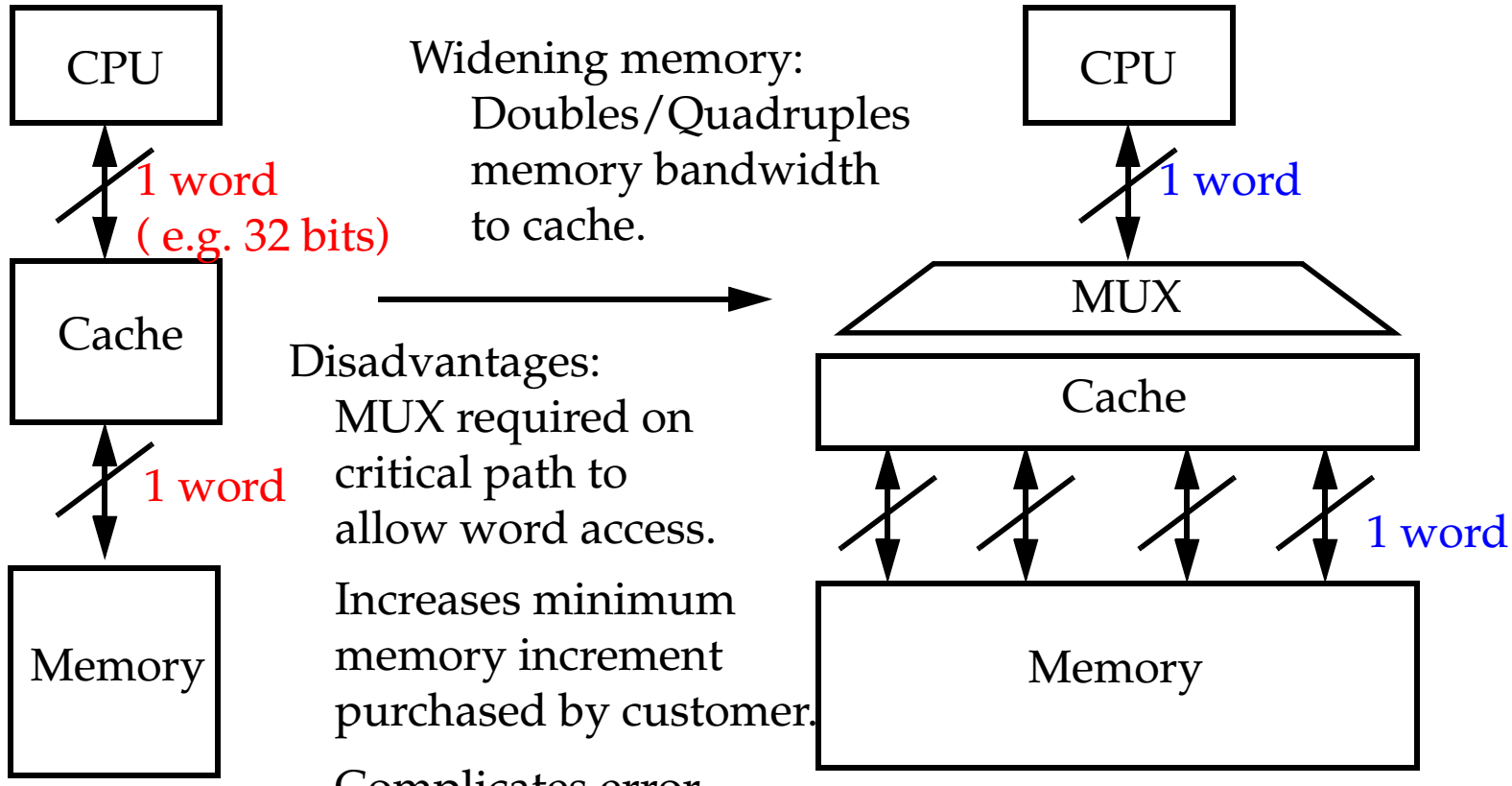
The CPU-DRAM performance gap is a problem, however, since DRAM performance improvement is only about 7% per year.

Cache innovations have addressed this problem to some degree.

We will now look at innovations in main memory organizations that are more cost effective.

Improving Main Memory Performance

- *Wider main memory*



Disadvantages:

MUX required on critical path to allow word access.

Increases minimum memory increment purchased by customer.

Complicates error correction.

Improving Main Memory Performance

- *Wider main memory*

DRAM chips are typically 1-8 bits wide.

Any number of them can be accessed in parallel without extra delay.

By increasing the width of memory, the CPU can get more bits in a single cycle.

This increases bandwidth between cache and memory.

For example, consider a cache with 4 word blocks.

Main memory might require:

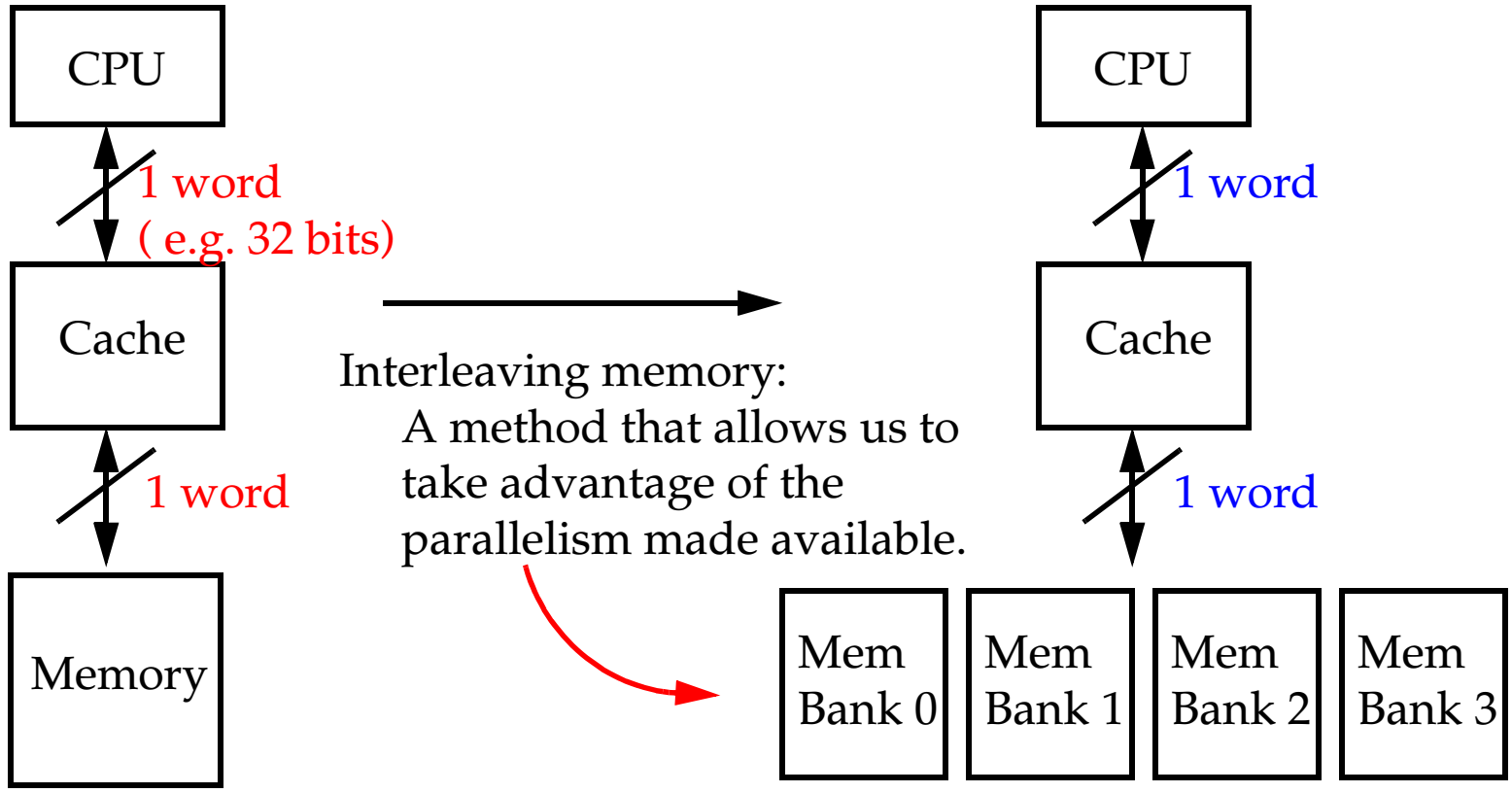
- 4 cycles to send the address.
- 40 cycles to access memory.
- 4 cycles to transfer over the bus.

If the memory is only **one word** wide, a miss would require 4 x (4 + 40 + 4) = 192 cycles!

If the memory is enlarged to **4 words wide**, miss time is only 48 cycles.

Improving Main Memory Performance

- *Interleaved memory*



Banks are often one word wide, so bus width need not be changed.

However, several independent areas of memory can be accessed simultaneously.

Improving Main Memory Performance

- *Interleaved memory*

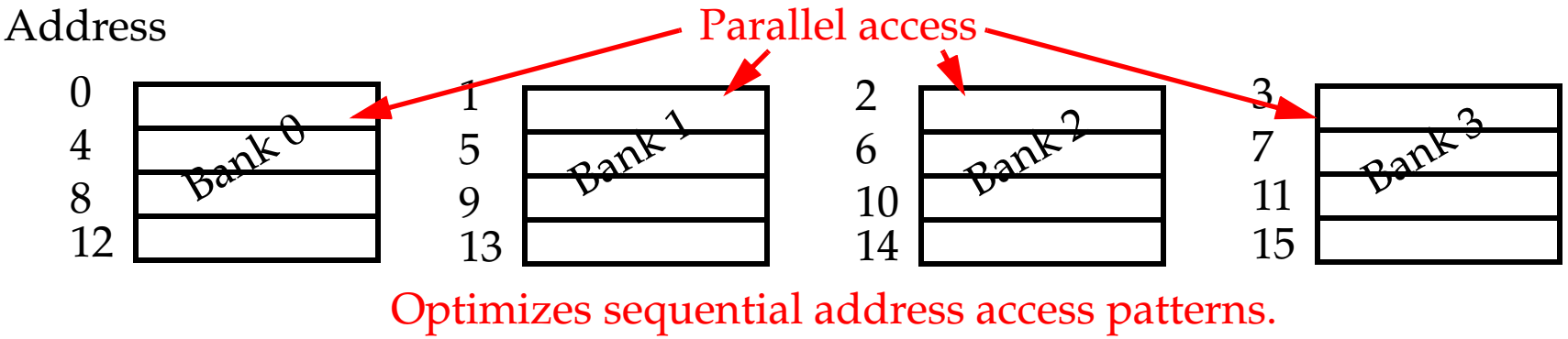
For example, we could fetch a block by

- Sending 1 address.
- Waiting for a single memory cycle.
- Transferring 4 words for a total time of 4 + 40 + (4 x 4) = 60 cycles.

Which is a little slower than wider memory (due to bus limitations) but it has several advantages.

Individual writes can also be overlapped if they are addressed to different banks.

One possible interleaving strategy: **Word interleaving**:



Optimizes sequential address access patterns.

Improving Main Memory Performance

- *Interleaved memory*

  Read access optimization is possible if, for example, cache block size is four words since parallel access is possible (no conflicts).

  Also, write-back caches make writes sequential as well as reads, improving efficiency even further.

  How many banks are sufficient ?

  One rule might be '# of banks >= # of clocks to access a word in a bank'.

  This allows up to 1 word per clock cycle in best case.

Improving Main Memory Performance

- *Independent memory banks*

The interleaved memory concept can be extended to remove **all** restrictions on memory access.

We assumed for interleaved memory that only a **single memory controller** was present.

This allowed the interleaving of sequential access patterns.

Address line sharing among the banks is possible in this scheme.

We can also use **multiple independent controllers**, e.g. one for I/O devices, one for cache reads and one for cache writes.

Banks are still accessed in parallel, but now there may be *multiple independent requests* serviced simultaneously.

This can be particularly useful with **nonblocking** caches (caches that allow multiple outstanding reads misses).

And multiprocessors.

Improving Main Memory Performance

- *Avoiding memory bank conflicts*

As with caches, programs can be modified to improve memory performance.

The most important is to keep all the banks running.

Programs that access all banks evenly will perform best.

However, data memory references are **not** random and may end up going to the same bank.

Using a *prime number* of memory banks makes this work well.

However, using a prime number makes the division operation expensive:

$$\text{Bank number} = \text{Address MOD Number of banks}$$

$$\text{Address within bank} = \frac{\text{Address}}{\text{Number of banks}}$$

Improving Main Memory Performance

- *Avoiding memory bank conflicts*

There are schemes that use a prime number of banks and fast modulo arithmetic to distribute memory accesses to many banks of memory.

For example, the following can be used:

Bank number = Address MOD Number of banks

Address within bank = Address MOD Number of words in bank

This avoids the use of an expensive 'non power of 2' division operation shown previously.

There is a proof that guarantees that the above mapping provides a unique mapping between an address and a memory location.

For numbers of the form 2^N-1, there is fast hardware to implement the operation.

Improving Main Memory Performance

The previous methods work with any memory technology.

We now look at techniques that take advantage of the nature of DRAMs.

The first three take advantage of the individual row access and column access operations that occur on a memory access.

DRAMs buffer a row of bits inside the DRAM for column access.

The size of the buffer is usually the square root of the DRAM size, e.g. 16Kbits for 64MBits.

In order to improve performance, DRAMs are designed to allow multiple accesses to this buffer, **eliminating** the row access time.

Improving Main Memory Performance

DRAM-specific interleaving

- *Nibble mode*

  The DRAM can supply three extra bits from locations sequential to the one just accessed.

  This can be done after each RAS (Row Access Strobe).

- *Page mode*

  The DRAM can act as an SRAM once a row has been selected.

  For example, random bits from the row can be selected by changing just the column address.

  This can occur until the next RAS or refresh.

- *Static column mode* (Extended Data Out [**EDO**] RAM)

  Very similar to page mode, except that it is not necessary to toggle (clock) the column access strobe line every time the column address changes.

These optimizations can improve bandwidth by a factor of **four**.

Improving Main Memory Performance

- *Synchronous DRAM (SDRAM)*

  In this type of DRAM, the clock is supplied to the RAM chip, and all signals are synchronized to it.

  This allows the RAM to run at higher speeds.

  Similarly, sequential data can be retrieved faster, at the rate of one bit per clock cycle.

- *VRAM*

  Video RAM is used to drive displays.

  It can be written or read using a normal interface or a special interface that **outputs rows** one bit at a time (good for video displays!).

Virtual memory

Virtual memory is just another level in the memory hierarchy.

It allows main memory to cache pages (blocks) normally stored on disk.

As with caches, the operations performed by virtual memory are transparent to properly-running user programs.

Similarity to caching.

- *Block = page*

  Blocks in caches are equivalent to pages in virtual memory.

  Pages are anywhere from 1 KB to 64 KB (though today's page sizes are usually 4+ KB).

- *Miss = page fault*

  A miss in a cache is analogous to a page fault.

  The only difference is the penalty.

  Millions of clock cycles for VM as compared to tens of clock cycles for caches.

Virtual memory

Similarity to caching.

- *Miss rate*

  The miss rate for VM is very low -- less than 0.001%.

  This means that fewer than one in **one million accesses** cause a VM miss, and it's often a *lot* fewer.

- *Size*

  Caches are 16 KB - 1 MB or more.

  The VM "cache" is 16 MB to 1024 MB or more -- a factor of 1000 larger.

Differences include:

- Replacement mechanism.

  In caches, it is primarily controlled by the hardware.

  In VM, replacement is primarily controlled by the OS.
- The number of bits in the address determines the size of VM where cache size is independent of the address size.

Two classes of VM: paging systems and segmentation systems.

Basic virtual memory caching questions

- *Where can a block be placed ?*

  Since miss penalties are very high, OS designers always choose lower miss rates over simple placement algorithms.

  Therefore, VM is *almost always* fully-associative (blocks can be placed anywhere in main memory).

- *How is a block found ?*

  Paging systems use a page table to translate virtual page numbers into physical page numbers.

  The physical address is constructed by concatenating the physical page number (found in the table) to the offset.

  Segmented systems use a similar structure except that the segment's physical address is ADDED to the offset.

  Note that the page table needs enough entries to map the entire virtual address space since it is accessed using virtual page numbers.

Basic virtual memory caching questions

- *How is a block found ?*

  This can result in a large amount of space dedicated just to the page table.

  One optimization is to use *hashing* to restrict the number of page table entries to the number of physical pages.

  This is called an **inverted page table**.

  Translation lookaside buffers (TLBs) are used to cache these translations, and reduce address translation time.

- *Which block is replaced ?*

  Most operating systems use LRU or an approximation to it.

  The page table often includes a reference bit to help do LRU replacement.

Basic virtual memory caching questions

- *What happens on a write ?*

  VM is *always writeback* (capture as many writes as possible before writing the page to disk).

  Write-through does not make sense because of the very large access penalty.

  Thus, the page table uses a **dirty bit** to keep track which pages have been modified and must be written to disk before they are replaced.

  We do not want to write pages to disk that have not been modified.

Page tables imply that a memory reference requires **two** memory accesses.

One for the page table and one to get the data.

A TLB, which caches previous translations, can be effective in reducing memory references to the page table.

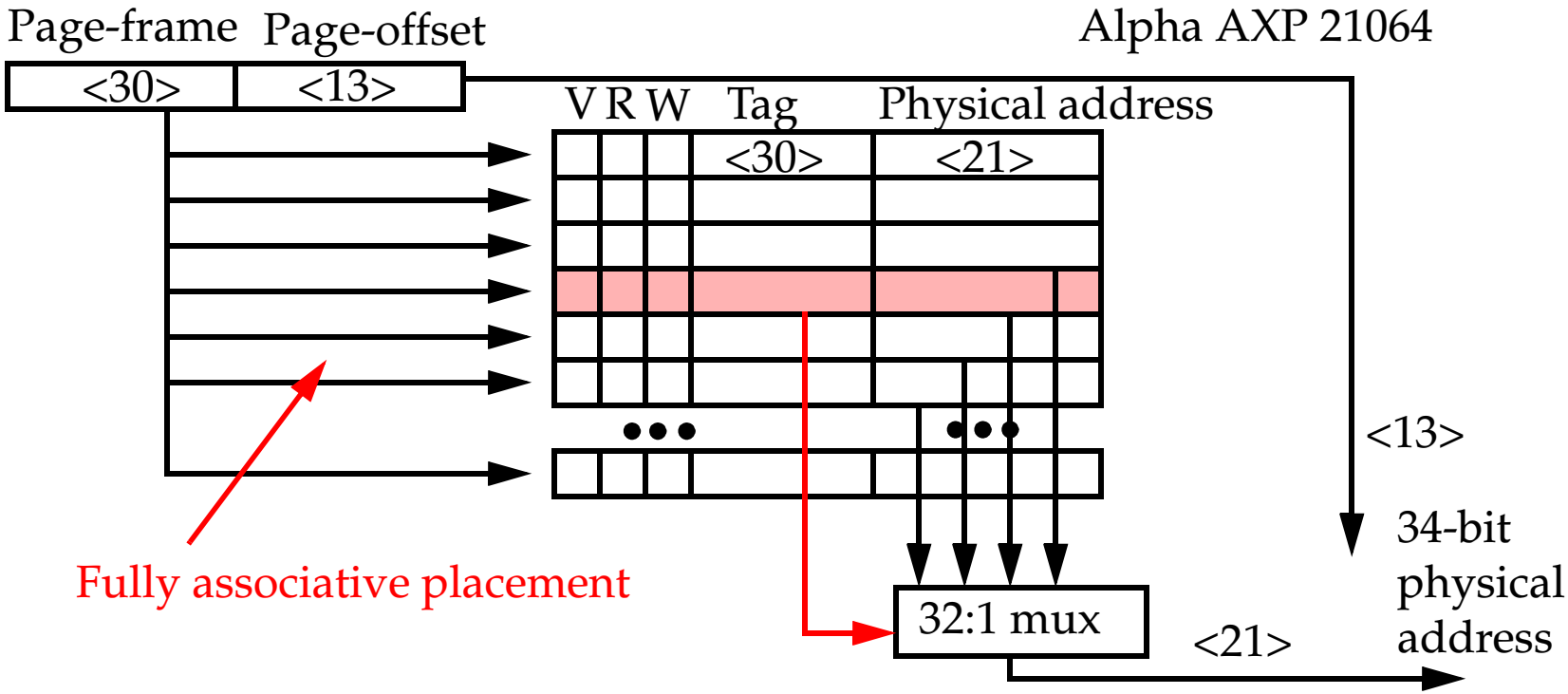
This works because of the principle of locality.

Translation Look-aside Buffer

Similar to a cache:

Tag holds the virtual address

Data portion holds the physical page frame number, protection field, valid bit, use bit and a dirty bit.



Translation Look-aside Buffer

As with normal caches, the TLB may be fully-associative, direct-mapped, or set-associative.

Replacement may be done in hardware or may be assisted by software.

For example, a miss in the TLB causes an exception which is handled by the OS, which places the appropriate page information into the TLB.

Hardware handling is faster, but software is more flexible.

Small, fast TLBs are crucial because they are on the *critical path* to accessing data from the cache.

This is particularly true if the cache is physically addressed.

Selecting a Page Sizes

Large page sizes are generally better because:

They reduce the size of the page table.

They are more efficient to transfer between memory and disk.

They allow a TLB to cache translations for more of memory.

The biggest drawback to large pages is that they may waste memory, **internal fragmentation**.

Assuming a process has three primary segments (text, heap and stack), the average wasted storage per process will be **1.5** times the page size.

When page size is 4 KB or 8 KB, this is negligible for machines with megabytes of memory.

For larger pages, e.g., 64 KB, lots of storage may be wasted.

Uses of Virtual Memory

- *Protection*

  VM is often used to protect one program from others in the system.

  Protection mechanisms must have hardware support.

  - *Base & bounds*

    Each reference must fall between two addresses, given by the base & bound registers.

    This method also allows some relocation.

  User processes cannot be allowed to change these registers, but the OS must be able to do so on a process switch.

  Therefore, the hardware must be able to:

  - Provide at least two modes of operations, user and kernel mode and a mechanism to switch between them.
  - Provide a protection mechanism for other portions of the CPU state to prevent user processes from being malicious.

    User/supervisor mode bit(s).

    Interrupt enable/disable bit(s).

Uses of Virtual Memory

- *Protection*

  Base and bound registers constitute the minimum protection system.

  Virtual Memory offers a more fine-grained alternative.

  Processes have their own page tables, which they cannot modify themselves.

  Permission flags are provided with each segment or page.

  Concentric rings of security and *capability lists* are more fined-grained alternatives, allowing more than two levels of protection.

  The OS course discusses these in detail.

Effects of CPU design on memory hierarchy

- *Superscalar & vector execution*

  A superscalar or vector machine may fetch several words per cycle.

  Clearly, the memory system must deliver the bandwidth to handle this, otherwise the benefit is lost.

  The brunt of the load falls upon the L1 cache.

  Bandwidth can be increased by widening the path to the cache or by providing extra ports to the cache.

  However, cache access is often the bottleneck in modern CPUs.

- *Speculative execution*

  Speculative execution and conditional instructions may generate invalid addresses that would not occur otherwise.

  The memory system must recognize and suppress these exceptions.

  Similarly, it must not stall the cache on a miss caused by a speculative instruction.

Effects of CPU design on memory hierarchy

- *I/O and cache consistency*

I/O devices move data from peripherals to memory.

This has two pitfalls:

Data written into memory is not automatically updated in the cache.

Data in a writeback cache is not written to memory immediately so memory has stale data.

One solution is to flush blocks from the cache that are used in the I/O operation.

This is done:

Before the I/O for a write (so the write operation uses up-to-date information).

After the I/O for the read (before the I/O should work as well. The CPU should not access the data as it is being read into memory).

An alternate method is simply to mark the blocks from I/O buffers as *uncacheable*.

Effects of CPU design on memory hierarchy

- *I/O and cache consistency*

  Other solutions include:

  - *Watch the I/O buses for addresses in the tag.*

    This eliminates the consistency problem.

    The drawback is that the checking slows down the cache.

  - *Do I/O directly into the cache.*

    This method guarantees consistency but it slows down the cache since both the CPU and I/O access it.

    Moreover, it displaces data in the cache with new data that is unlikely to be accessed soon by the CPU.

Fallacies and Pitfalls

- *Don't predict cache performance of program A from program B.*

  Programs vary widely in how they use cache.

  A scientific program may have a small tight code loop but access large quantities of data.

  On the other hand, a word processing program might operate on relatively little data but use lots of code.

- *Simulate plenty of memory references.*

  A CPU executes 100 million or more instructions per second.

  Simulating cache behavior using traces of only a few million traces can be misleading.

  Particularly since program locality behavior is not constant over the run of the entire program.

- *Don't ignore the OS.*

  The OS can miss or interfere with application programs, causing misses.