# 332
# Advanced Computer Architecture
# Chapter 3

## Caches and Memory Systems

January 2003

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd ed)*, and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course *(CS252, Jan 2001)*



## Review: Cache performance

- Miss-oriented Approach to Memory Access:

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalt\ y \right) \times CycleTime$$

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalt\ y \right) \times CycleTime$$

  - $CPI_{Execution}$ includes ALU and Memory instructions

- Separating out Memory component entirely
  - AMAT = Average Memory Access Time
  - $CPI_{ALUOps}$ does not include memory instructions

$$CPUtime = IC \times \left( \frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalt\ y \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalt\ y_{Inst}\right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalt\ y_{Data}\right) \end{aligned}$$



## Average memory access time:

$$AMAT = HitTime + MissRate \times MissPenalt\ y$$

## There are three ways to improve cache performance:

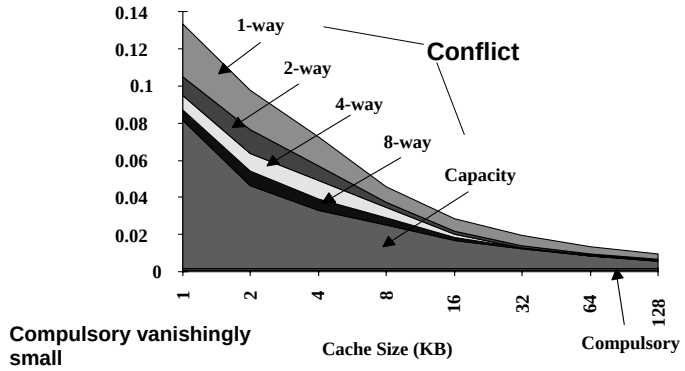
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.



## Reducing Misses

- Classifying Misses: 3 Cs
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.
    *(Misses in even an Infinite Cache)*
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
    *(Misses in Fully Associative Size X Cache)*
  - *Conflict*—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.
    *(Misses in N-way Associative, Size X Cache)*
- More recent, 4th "C":
  - *Coherence* - Misses caused by cache coherence.

## 3Cs Absolute Miss Rate (SPEC92)

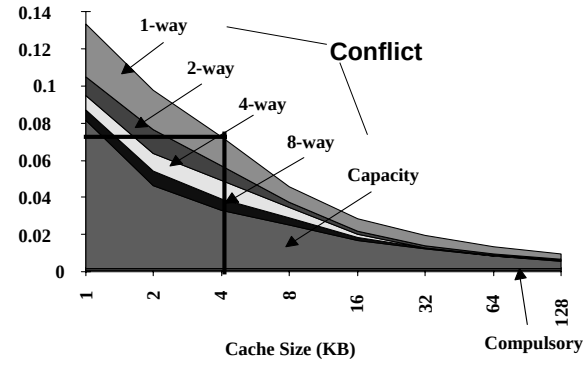



Compulsory vanishingly small



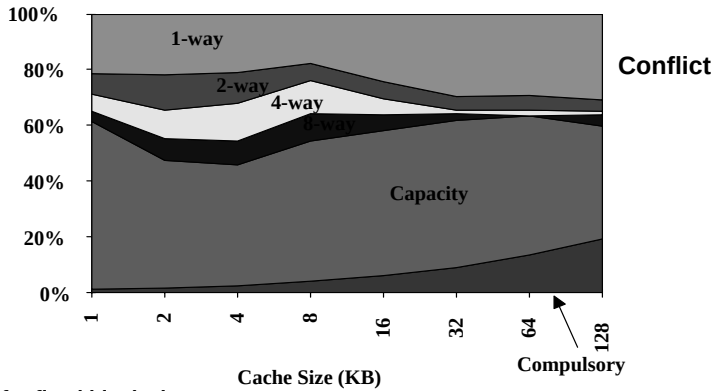
## 2:1 Cache Rule (of thumb!)

miss rate 1-way associative cache size X
= miss rate 2-way associative cache size X/2






## 3Cs Relative Miss Rate




Flaws: for fixed block size
Good: insight => invention



## How Can Reduce Misses?

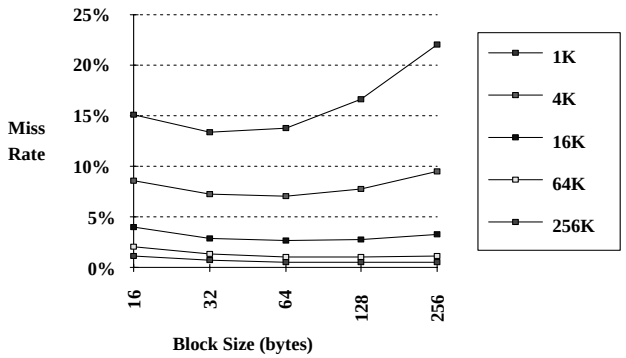
- 3 Cs: Compulsory, Capacity, Conflict
- In all cases, assume total cache size not changed:
- What happens if:

1) Change Block Size:
Which of 3Cs is obviously affected?

2) Change Associativity:
Which of 3Cs is obviously affected?

3) Change Compiler:
Which of 3Cs is obviously affected?

# 1. Reduce Misses via Larger Block Size






# 2. Reduce Misses via Higher Associativity

- 2:1 Cache Rule of thumb:
  - The Miss Rate of a direct-mapped cache of size N
  - Is the same as the Miss Rate of a 2-way set-associative cache size of size N/2
    - on average, over a large suite of benchmarks
- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%



# Example: Avg. Memory Access Time vs. Miss Rate

- Example: assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

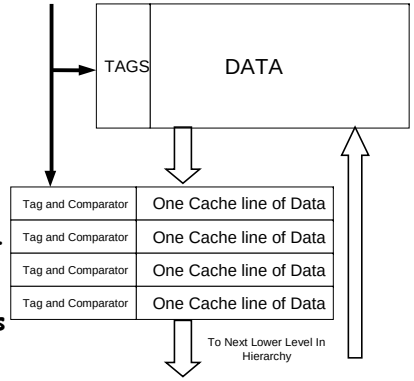
| Cache Size | Associativity | | | |
|---|---|---|---|---|
| (KB) | 1-way | 2-way | 4-way | 8-way |
| 1 | 2.33 | 2.15 | 2.07 | 2.01 |
| 2 | 1.98 | 1.86 | 1.76 | 1.68 |
| 4 | 1.72 | 1.67 | 1.61 | 1.53 |
| 8 | 1.46 | 1.48 | 1.47 | 1.43 |
| 16 | 1.29 | 1.32 | 1.32 | 1.32 |
| 32 | 1.20 | 1.24 | 1.25 | 1.27 |
| 64 | 1.14 | 1.20 | 1.21 | 1.23 |
| 128 | 1.10 | 1.17 | 1.18 | 1.20 |

(Red means A.M.A.T. not improved by more associativity)



# 3. Reducing Misses via a "Victim Cache"

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines



## 4. Reducing Misses via "Pseudo-Associativity"

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)


- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R10000 L2 cache, similar in UltraSPARC



## 5. Reducing Misses by Hardware Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in "stream buffer"
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty



## 6. Reducing Misses by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution
- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - Can be incorrect. Frees HW/SW to guess!
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth



## 7. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

## Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality



## Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
       for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
       for (j = 0; j < 100; j = j+1)
              x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality



## Loop Fusion Example

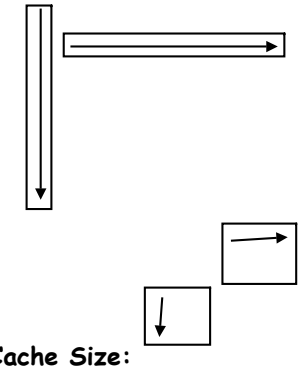
```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {    a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality



## Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1){
       r = r + y[i][k]*z[k][j];};
     x[i][j] = r;
    };
```



- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

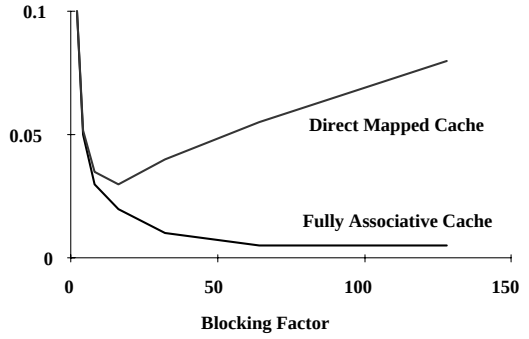
## Blocking Example

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
   for (j = jj; j < min(jj+B-1,N); j = j+1)
     {r = 0;
      for (k = kk; k < min(kk+B-1,N); k = k+1) {
        r = r + y[i][k]*z[k][j];};
      x[i][j] = x[i][j] + r;
     };
```

- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $N^3/B+2N^2$
- Conflict Misses Too?



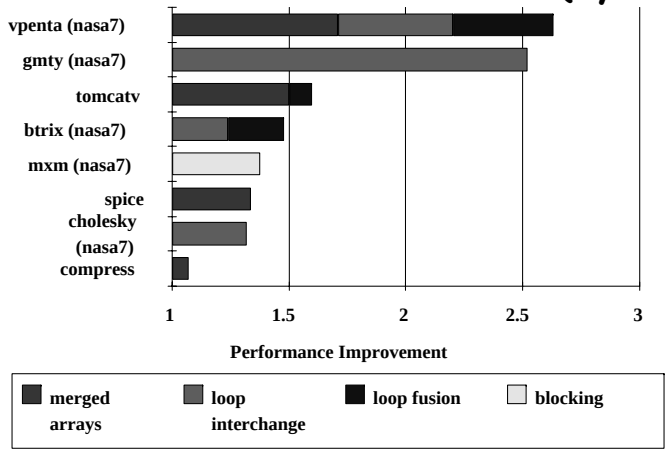
## Reducing Conflict Misses by Blocking




- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache



## Summary of Compiler Optimizations to Reduce Cache Misses (by hand)






## Summary: Miss Rate Reduction

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \text{Miss rate} \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

- 3 Cs: Compulsory, Capacity, Conflict
  1. Reduce Misses via Larger Block Size
  2. Reduce Misses via Higher Associativity
  3. Reducing Misses via Victim Cache
  4. Reducing Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr, Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Misses by Compiler Optimizations
- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - Can be incorrect. Frees HW/SW to guess!

## Review: Improving Cache Performance

1. Reduce the miss rate,
2. *Reduce the miss penalty,* or
3. Reduce the time to hit in the cache.



## Write Policy: Write-Through vs Write-Back

- Write-through: all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a *valid* bit
- Write-back: all writes simply update cache
  - Can't just discard cached data - may have to write it back to memory
  - Cache control bits: both *valid* and *dirty* bits
- Other Advantages:
  - Write-through:
    - memory (or other processors) always have latest data
    - Simpler management of cache
  - Write-back:
    - much lower bandwidth, since data often overwritten multiple times
    - Better tolerance to long-latency memory?

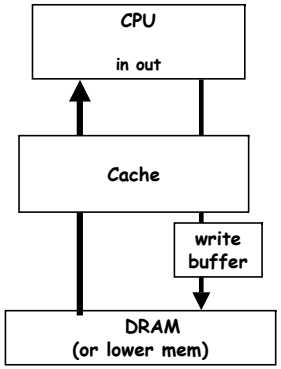


## Write Policy 2: Write Allocate vs Non-Allocate (What happens on write-miss)

- Write allocate: allocate new cache line in cache
  - Usually means that you have to do a "read miss" to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- Write non-allocate (or "write-around"):
  - Simply send write data through to underlying memory/cache - don't allocate new cache line!



## 1. Reducing Miss Penalty: Read Priority over Write on Miss




- Consider write-through with write buffers
  - RAW conflicts with main memory reads on cache misses
    - Could simply wait for write buffer to empty, before allowing read
    - Risks serious increase in read miss penalty (old MIPS 1000 by 50% )
    - Solution:
      - Check write buffer contents before read; if no conflicts, let the memory access continue
- Write-back also needs buffer to hold displaced blocks
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

## 2. Reduce Miss Penalty: Early Restart and Critical Word First

- Don't wait for full block to be loaded before restarting CPU
  - *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- Generally useful only in large blocks,
- (Access to contiguous sequential words is very common – but doesn't benefit from either scheme – are they worthwhile?)

block

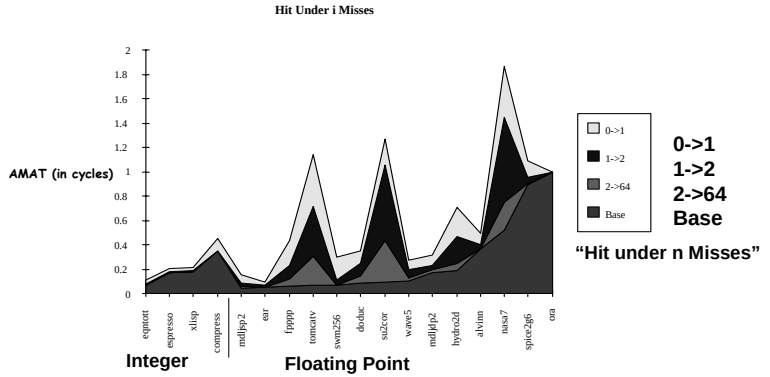


## 3. Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- *Non-blocking cache* or *lockup-free cache* allows data cache to continue to supply cache hits during a miss
  - requires full/empty bits on registers or out-of-order execution
  - requires multi-bank memories
- "*hit under miss*" reduces the effective miss penalty by working during miss instead of ignoring CPU requests
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses



## Value of Hit Under Miss for SPEC




0->1
1->2
2->64
Base

"Hit under n Misses"

- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss



## 4: Add a second-level cache

- L2 Equations

$$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times Miss\ Penalty_{L1}$$

$$Miss\ Penalty_{L1} = Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$$

$$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times (Hit\ Time_{L2} + Miss\ Rate_{L2} + Miss\ Penalty_{L2})$$

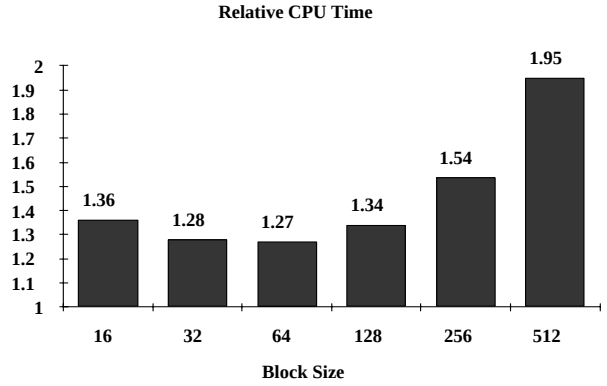
- Definitions:
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* (Miss $rate_{L2}$)
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU* (Miss $Rate_{L1} \times$ Miss $Rate_{L2}$)
  - Global Miss Rate is what matters

# Reducing Misses: Which apply to L2 Cache?

- Reducing Miss Rate
  1. Reduce Misses via Larger Block Size
  2. Reduce Conflict Misses via Higher Associativity
  3. Reducing Conflict Misses via Victim Cache
  4. Reducing Conflict Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr, Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Capacity/Conf. Misses by Compiler Optimizations



# L2 cache block size & A.M.A.T.




- 32KB L1, 8 byte path to memory



# Reducing Miss Penalty Summary

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \text{Miss rate} \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

- Four techniques
  - Read priority over write on miss
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache
- Can be applied recursively to Multilevel Caches
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse



# What happens on a Cache miss?

- For in-order pipeline, 2 options:
  - Freeze pipeline in Mem stage (popular early on: Sparc, R4000)

```
IF  ID  EX  Mem stall stall stall … stall Mem   Wr
    IF  ID  EX  stall stall stall … stall stall Ex Wr
```

  - Use Full/Empty bits in registers + MSHR queue
    - MSHR = "Miss Status/Handler Registers" (Kroft)
      Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.
      - Per cache-line: keep info about memory address.
      - For each word: register (if any) that is waiting for result.
      - Used to "merge" multiple requests to one memory line
    - New load creates MSHR entry and sets destination register to "Empty". Load is "released" from pipeline.
    - Attempt to use register before result returns causes instruction to block in decode stage.
    - Limited "out-of-order" execution with respect to loads. Popular with in-order superscalar architectures.
- Out-of-order pipelines already have this functionality built in… (load queues, etc).

## Average memory access time:

$$AMAT = HitTime + MissRate \times MissPenalty$$

## There are three ways to improve cache performance:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.



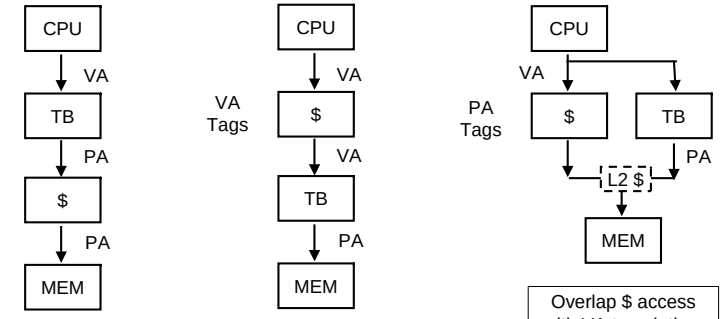
## Reducing the time to hit in the cache

- Why does the Alpha 21164 have 8KB Instruction and 8KB data cache + 96KB second level cache, all on-chip?

1. Keep the cache small and simple
2. Keep address translation off the critical path
3. Pipeline the cache access



## 2. Fast hits by Avoiding Address Translation



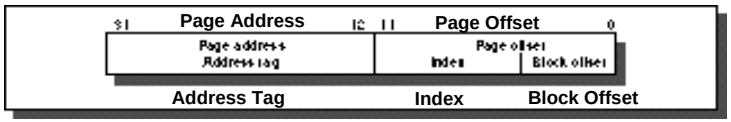



## 2. Fast hits by Avoiding Address Translation

- Send virtual address to cache? Called *Virtually Addressed Cache* or just *Virtual Cache* vs. *Physical Cache*
  - Every time process is switched logically must flush the cache; otherwise get false hits
    - Cost is time to flush + "compulsory" misses from empty cache
  - Dealing with *aliases* (sometimes called *synonyms/homonyms*): Two different virtual addresses map to same physical address,
    Two different physical addresses mapped to by the same virtual address in different contexts
  - I/O must interact with cache, so need virtual address
- Solution to aliases
  - HW guaranteess covers index field & direct mapped, they must be unique; called *page coloring*
- Solution to cache flush
  - Add *process identifier tag* that identifies process as well as address within process: can't get a hit if wrong process

## 2. Fast Cache Hits by Avoiding Translation: Index with Physical Portion of Address

- If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag




- Limits cache to page size: what if want bigger caches and uses same trick?
  - Higher associativity moves barrier to right
  - Page coloring



## 3: Fast Hits by pipelining Cache Case Study: MIPS R4000

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- What is impact on Load delay?
  - Need 2 instructions between a load and its use!



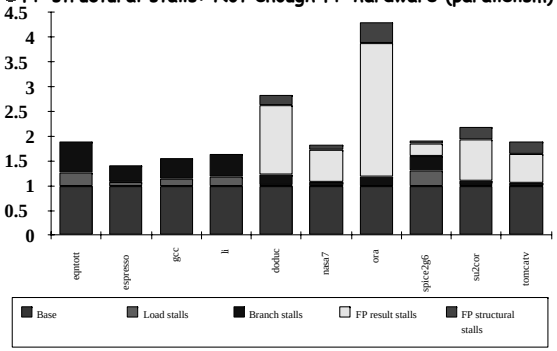
## Case Study: MIPS R4000

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TWO Cycle Load Latency | IF | IS | RF | EX | DF | DS | TC | WB |
| | | IF | IS | RF | EX | DF | DS | TC |
| | | | IF | IS | RF | EX | DF | DS |
| | | | | IF | IS | RF | EX | DF |
| | | | | | IF | IS | RF | EX |
| | | | | | | IF | IS | RF |
| | | | | | | | IF | IS |
| | | | | | | | | IF |
| THREE Cycle Branch Latency (conditions evaluated during EX phase) | IF | IS | RF | EX | DF | DS | TC | WB |
| | | IF | IS | RF | EX | DF | DS | TC |
| | | | IF | IS | RF | EX | DF | DS |
| | | | | IF | IS | RF | EX | DF |
| | | | | | IF | IS | RF | EX |
| | | | | | | IF | IS | RF |
| | | | | | | | IF | IS |
| | | | | | | | | IF |

Delay slot plus two stalls
Branch likely cancels delay slot if not taken

# R4000 Performance

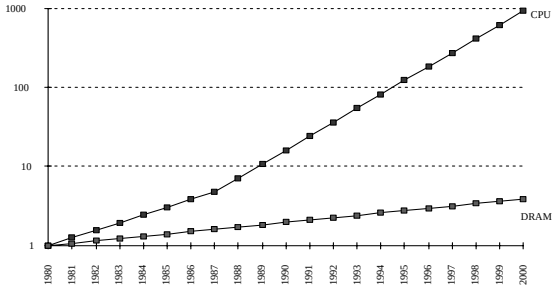
- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)






# What is the Impact of What You've Learned About Caches?

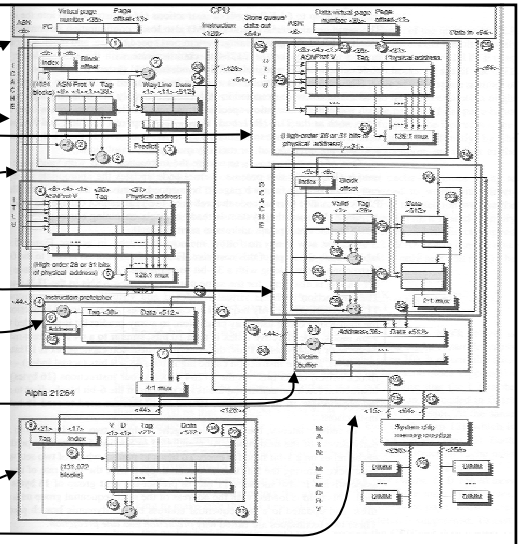
- 1960-1985: Speed = *f*(no. operations)
- 1990
  - Pipelined Execution & Fast Clock Rate
  - Out-of-Order execution
  - Superscalar Instruction Issue
- 1998: Speed = *f*(non-cached memory accesses)
- Superscalar, Out-of-Order machines hide L1 data cache miss (~5 clocks) but not L2 cache miss (~50 clocks)?





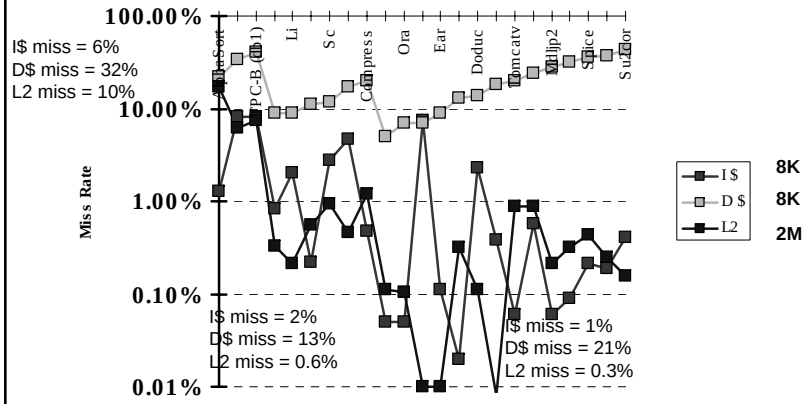

# Alpha 21064

- Processor issues 48-bit virtual addresses
- Separate Instr & Data TLB & Caches
- TLBs fully associative
- TLB updates in SW ("Priv Arch Libr")
- Caches 8KB direct mapped, write thru, virtually-indexed, physically tagged
- Critical 8 bytes first
- Prefetch instr. stream buffer
- 4 entry write buffer between D$ & L2$ incorporates victim buffer: to give read priority over write
- 2 MB L2 cache, direct mapped, WB (off-chip)
- 256 bit path to main memory, 4 x 64-bit modules



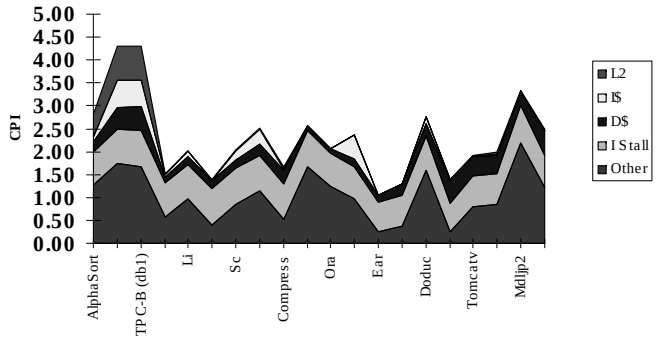



# Alpha Memory Performance: Miss Rates of SPEC92



# Alpha CPI Components

- Instruction stall: branch mispredict (green);
- Data cache (blue); Instruction cache (yellow); L2$ (pink)
  Other: compute + reg conflicts, structural conflicts






# Cache Optimization Summary

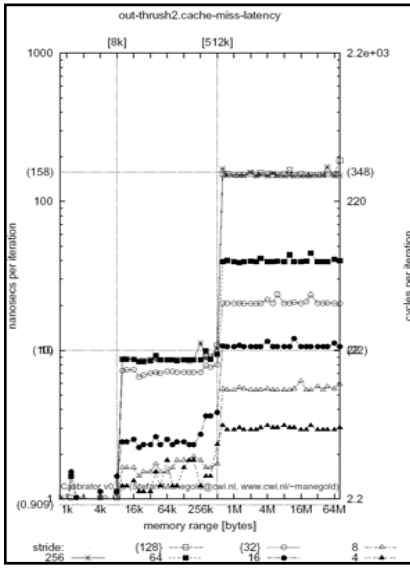
| | Technique | MR | MP | HT | Complexity |
|---|---|---|---|---|---|
| miss rate | Larger Block Size | + | – | | 0 |
| | Higher Associativity | + | | – | 1 |
| | Victim Caches | + | | | 2 |
| | Pseudo-Associative Caches | + | | | 2 |
| | HW Prefetching of Instr/Data | + | | | 2 |
| | Compiler Controlled Prefetching | + | | | 3 |
| | Compiler Reduce Misses | + | | | 0 |
| miss penalty | Priority to Read Misses | | + | | 1 |
| | Early Restart & Critical Word 1st | | + | | 2 |
| | Non-Blocking Caches | | + | | 3 |
| | Second Level Caches | | + | | 2 |



# Practical exercise: explore memory hierarchy on your favourite computer

- Download Stefan Manegold's "cache and TLB calibrator":
  - http://www.cwi.nl/~manegold/Calibrator/calibrator.shtml
    (or find installed copy in ~phjk/ToyPrograms/C/ManegoldCalibrator)
- This program consists of a loop which runs over an array repeatedly
  - The size of the array is varied to evaluate cache size
  - The stride is varied to explore block size






## Memory hierarchy of a 2.2GHz Intel Pentium 4 Xeon

- Memory access latency is close to 1ns when loop reuses array smaller than 8KB level-1 cache
- While array is smaller than 512KB, access time is 2-8ns, depending on stride
- When array exceeds 512KB, accesses miss both level-1 and level-2 caches
- Worst case (large stride) suffers 158ns access latency
- Q:
  - How many instructions could be executed in 158ns?
  - what is the level-1 cache block size?
  - What is the level-2 cache block size?

## Instructions for running the Manegold calibrator

- Get a copy:
  - cp /homes/phjk/ToyPrograms/C/ManegoldCalibrator/calibrator.c ./
- Compile it:
  - gcc –O3 –o calibrator calibrator.s
- Find out CPU MHz
  - cat /proc/cpuinfo
- Run it; ./calibrator <CPUMHz> <size> <filename>
- Eg on media03:
  - ./calibrator 3000 64M media03
  - Output is delivered to a set of files "media03.*"
- Plot postscript graphs using generated gnuplot scripts:
  - gnuplot media03.cache-miss-latency.gp
  - gnuplot media03.cache-replace-time.gp
  - gnuplot media03.TLB-miss-latency.gp
- View the generated postscript files:
  - gv media03.cache-miss-latency.ps &



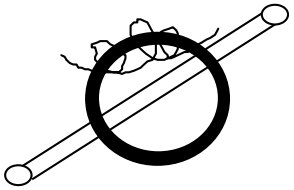
## Main Memory Background

- Performance of Main Memory:
  - Latency: Cache Miss Penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms, 1% time)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS* or *Row Access Strobe*
    - *CAS* or *Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor
    *Size*: DRAM/SRAM - *4-8*,
    *Cost/Cycle time*: SRAM/DRAM - *8-16*



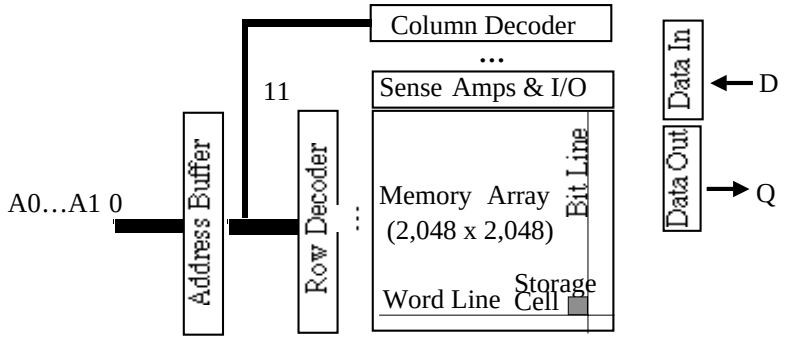
## Main Memory Deep Background

- "Out-of-Core", "In-Core," "Core Dump"?
- "Core memory"?
- Non-volatile, magnetic
- Lost to 4 Kbit DRAM (today using 64Kbit DRAM)
- Access time 750 ns, cycle time 1500-3000 ns





## DRAM logical organization (4 Mbit)




- Square root of bits per RAS/CAS

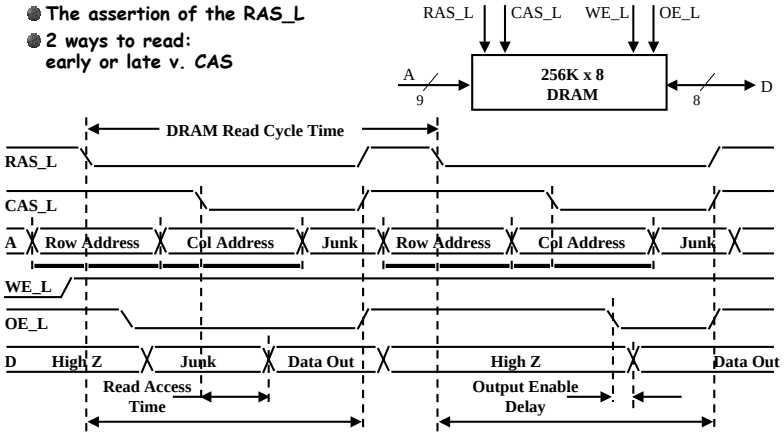
# 4 Key DRAM Timing Parameters

- $t_{RAC}$: minimum time from RAS line falling to the valid data output.
  - Quoted as the speed of a DRAM when buy
  - A typical 4Mb DRAM $t_{RAC}$ = 60 ns
  - Speed of DRAM since on purchase sheet?
- $t_{RC}$: minimum time from the start of one row access to the start of the next.
  - $t_{RC}$ = 110 ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns
- $t_{CAC}$: minimum time from CAS line falling to valid data output.
  - 15 ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns
- $t_{PC}$: minimum time from the start of one column access to the start of the next.
  - 35 ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns



# DRAM Read Timing

- Every DRAM access begins at:
  - The assertion of the RAS_L
  - 2 ways to read: early or late v. CAS




Early Read Cycle: OE_L asserted before CAS_L

Late Read Cycle: OE_L asserted after CAS_L



# DRAM Performance

- A 60 ns ($t_{RAC}$) DRAM can
  - perform a row access only every 110 ns ($t_{RC}$)
  - perform column access ($t_{CAC}$) in 15 ns, but time between column accesses is at least 35 ns ($t_{PC}$).
    - In practice, external address delays and turning around buses make it 40 to 50 ns
- These times do not include the time to drive the addresses off the microprocessor nor the memory controller overhead!



# DRAM History

- DRAMs: capacity +60%/yr, cost –30%/yr
  - 2.5X cells/area, 1.5X die size in ~3 years
- '98 DRAM fab line costs $2B
  - DRAM only: density, leakage v. speed
- Rely on increasing no. of computers & memory per computer (60% market)
  - SIMM or DIMM is replaceable unit
    => computers use any generation DRAM
- Commodity, second source industry
  => high volume, low profit, conservative
  - Little organization innovation in 20 years
- Order of importance: 1) Cost/bit 2) Capacity
  - First RAMBUS: 10X BW, +30% cost => little impact

# DRAM Future: 1 Gbit DRAM (ISSCC '96; production '02?)

| | Mitsubishi | Samsung |
|---|---|---|
| Blocks | 512 x 2 Mbit | 1024 x 1 Mbit |
| Clock | 200 MHz | 250 MHz |
| Data Pins | 64 | 16 |
| Die Size | 24 x 24 mm | 31 x 21 mm |
| Sizes will be much smaller in production | | |
| Metal Layers | 3 | 4 |
| Technology | 0.15 micron | 0.16 micron |



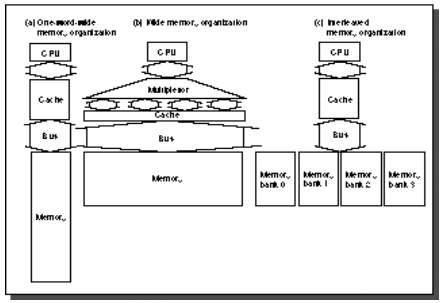
# Fast Memory Systems: DRAM specific

- Multiple CAS accesses: several names (page mode)
  - *Extended Data Out (EDO)*: 30% faster in page mode
- New DRAMs to address gap; what will they cost, will they survive?
  - *RAMBUS*: startup company; reinvent DRAM interface
    - Each Chip a module vs. slice of memory
    - Short bus between CPU and chips
    - Does own refresh
    - Variable amount of data returned
    - 1 byte / 2 ns (500 MB/s per chip)
    - 20% increase in DRAM area
  - *Synchronous DRAM*: 2 banks on chip, a clock signal to DRAM, transfer synchronous to system clock (66 - 150 MHz)
  - Intel claims RAMBUS Direct (16 b wide) is future PC memory?
- Niche memory or main memory?
  - e.g., Video RAM for frame buffers, DRAM + fast serial output



# Main Memory Organizations

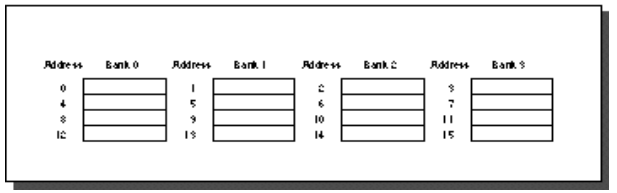
- *Simple*:
  - CPU, Cache, Bus, Memory same width (32 or 64 bits)
- *Wide*:
  - CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UtraSPARC 512)
- *Interleaved*:
  - CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved*





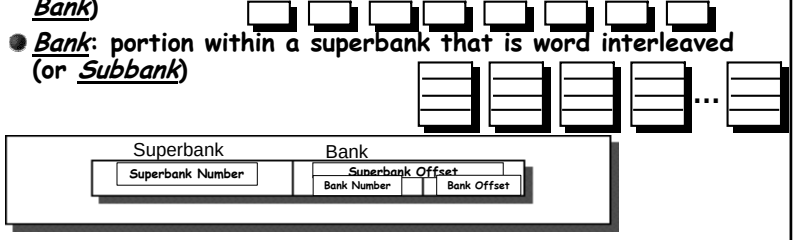

# Main Memory Performance

- Timing model (word size is 32 bits)
  - 1 to send address,
  - 6 access time, 1 to send data
  - Cache Block is 4 words
- *Simple M.P.* = 4 x (1+6+1) = 32
- *Wide M.P.* = 1 + 6 + 1 = 8
- *Interleaved M.P.* = 1 + 6 + 4x1 = 11



# Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses
  - Multiprocessor
  - I/O
  - CPU with Hit under n Misses, Non-blocking Cache
- ***Superbank***: all memory active on one block transfer (or ***Bank***)
- ***Bank***: portion within a superbank that is word interleaved (or ***Subbank***)






# Independent Memory Banks

- How many banks?

  number banks ≤ number clocks to access word in bank
  - For sequential accesses, otherwise will return to original bank before it has next word ready
  - (like in vector case)
- Increasing DRAM => fewer chips => harder to have banks



# Avoiding Bank Conflicts

- Lots of banks

```
int x[256][512];
for (j = 0; j < 512; j = j+1)
     for (i = 0; i < 256; i = i+1)
          x[i][j] = 2 * x[i][j];
```

- Conflicts occur even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if $2^N$ words per bank



# Fast Bank Number

- Chinese Remainder Theorem

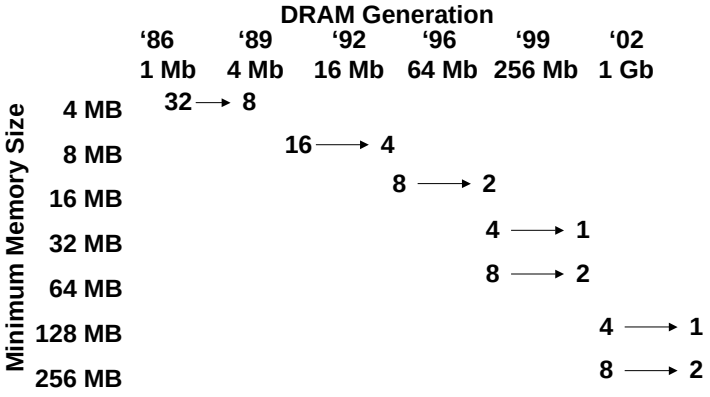
  As long as two sets of integers ai and bi follow these rules

  $$b_i = x \bmod a_i, 0 \le b_i < a_i, 0 \le x < a_0 \times a_1 \times a_2 \times \ldots$$

  and that ai and aj are co-prime if i ≠ j, then the integer x has only one solution (unambiguous mapping):
  - bank number = $b_0$, number of banks = $a_0$ (= 3 in example)
  - address within bank = $b_1$, number of words in bank = $a_1$ (= 8 in example)
  - N word address 0 to N-1, prime no. banks, words power of 2

| | | Seq. Interleaved | | | Modulo Interleaved | | |
|---|---|---|---|---|---|---|---|
| Bank Number: | | 0 | 1 | 2 | 0 | 1 | 2 |
| Address | | | | | | | within |
| Bank: | 0 | 0 | 1 | 2 | 0 | 16 | 8 |
| | 1 | 3 | 4 | 5 | 9 | 1 | 17 |
| | 2 | 6 | 7 | 8 | 18 | 10 | 2 |
| | 3 | 9 | 10 | 11 | 3 | 19 | 11 |
| | 4 | 12 | 13 | 14 | 12 | 4 | 20 |
| | 5 | 15 | 16 | 17 | 21 | 13 | 5 |
| | 6 | 18 | 19 | 20 | 6 | 22 | 14 |
| | 7 | 21 | 22 | 23 | 15 | 7 | 23 |

# DRAMs per PC over Time






# Need for Error Correction!

- Motivation:
  - Failures/time *proportional* to number of bits!
  - As DRAM cells shrink, more vulnerable
- Went through period in which failure rate was low enough without error correction that people didn't do correction
  - DRAM banks too large now
  - Servers always corrected memory systems
- Basic idea: add redundancy through parity bits
  - Simple but wasteful version:
    - Keep three copies of everything, vote to find right value
    - 200% overhead, so not good!
  - Common configuration: Random error correction
    - SEC-DED (single error correct, double error detect)
    - One example: 64 data bits + 8 parity bits (11% overhead)
    - Papers up on reading list from last term tell you how to do these types of codes
  - Really want to handle failures of physical components as well
    - Organization is multiple DRAMs/SIMM, multiple SIMMs
    - Want to recover from failed DRAM and failed SIMM!
    - Requires more redundancy to do this
    - All major vendors thinking about this in high-end machines



# Architecture in practice

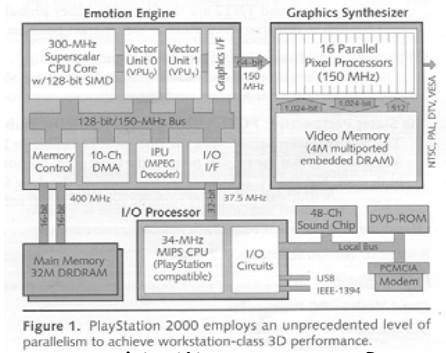



Figure 1. PlayStation 2000 employs an unprecedented level of parallelism to achieve workstation-class 3D performance.

Figure 2. PlayStation 2000 screenshot. (Source: Namco)

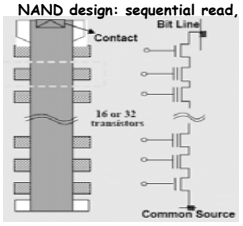
- (as reported in Microprocessor Report, Vol 13, No. 5)
  - Emotion Engine: 6.2 GFLOPS, 75 million polygons per second
  - Graphics Synthesizer: 2.4 Billion pixels per second
  - Claim: *Toy Story* realism brought to games!



# More esoteric Storage Technologies?

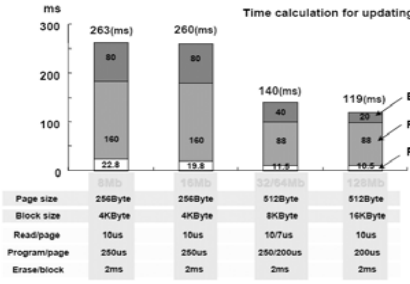
- FLASH
  - Mosfet cell with two gates
  - One "floating"
  - To program, charge tunnels via <7nm dielectric
  - Cells can only be erased (reset to 0) in blocks

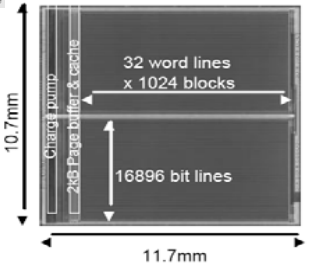
NAND design: sequential read, high density





Jan 2004: $1200



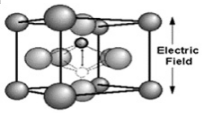



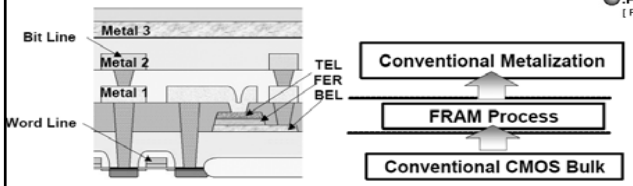

Gbit NAND Flash memory

bwrc.eecs.berkeley.edu/Classes/ICDesign/ EE241_s02/Lectures/lecture28-Flash.pdf

## More esoteric Storage Technologies?

- FRAM
  - Perovskite ferroelectric crystal forms dielectric in capactor, stores bit via phase change
  - 100ns read, 100ns write
  - Very low write energy (ca.1nJ)




Conventional Metalization

FRAM Process

Conventional CMOS Bulk

Additional FRAM process between conventional CMOS bulk and metalization

Compatible with conventional CMOS technology and existing CMOS cell libraries

- Fully integrated with logic fab process
- Currently used in Smartcards/RFID
- Soon to overtake Flash?

http://www.fma.fujitsu.com/fram/framDocs01.asp?grOut=Documentation&sec=Documentation



## Main Memory Summary

- Wider Memory
- Interleaved Memory: for sequential or independent accesses
- Avoiding bank conflicts: SW & HW
- DRAM specific optimizations: page mode & Specialty DRAM
- Need Error correction