6
Cache Data Organization

18-548/15-548 Memory System Architecture
Philip Koopman
September 14, 1998

Required Reading: Cragon 2.2.7, 2.3-2.5.2, 3.5.8
Supplemental Reading: Hennessy & Patterson 5.3, 5.4

Assignments

- **By next class read about associativity:**
  - Cragon pg. 166-174
- **Homework 3 due Wednesday September 16**
- **Lab 2 due Friday September 23**

Where Are We Now?

- **Where we've been:**
  - Physical memory hierarchy -- size vs. speed
  - Virtual memory hierarchy -- mapping
- **Where we're going today:**
  - Details of data organization
    - Split vs. Unified caches
    - Block size tradeoffs
- **Where we're going next time:**
  - Associativity
  - Data management policies

Why Not Use a Huge Cache? (Revisited)

- **L1 is generally implemented as on-chip cache**
  - Going off-chip takes longer than staying on-chip
  - Slows down with address fanout, chip select, data bus drive for off-chip
- **Area**
  - On-chip cache must share chip with CPU; limits size to an extent
- **Address translation**
  - Bits available to address cache usually limited to those not mapped by virtual memory
- **Compulsory misses may dominate anyway**
  - Caches bigger than program+data size (working set) don't help
  - Transaction processing has small working sets with short lives

How Do You Make the Most of Cache?

- **Keep cache arrays small & simple**
  - Faster clock cycle
  - Reduced area consumption
- **Make the best use of a limited resource (next few lectures)**
  - Arrange data efficiently
    - Split vs. unified caches
    - Block and sector size vs. overhead for tags and flag bits
  - Minimize miss rate while keeping cycle time & area usage low
    - Use associativity in just the right amount
  - Use good management policies

Preview

- **Design Target Miss Rates**
  - A conservative starting point for designs
- **Split *vs.* Unified Caches**
  - I-cache
  - D-cache
  - Unified Cache
- **Sectors & Blocks**
  - Bigger is better...
    Except when bigger is worse

DESIGN TARGET MISS RATES (DTMR)

DTMR Overview

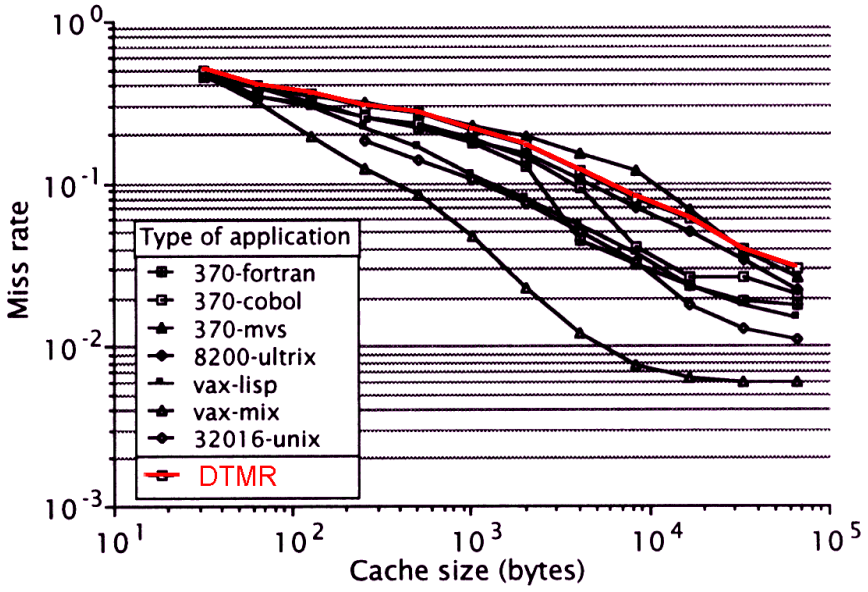
- **1985 ISCA paper by Smith, updated by Flynn in his book**
  - Conservative estimate of expectations
- **Idea is baseline performance data to estimate effects of changes in baseline cache structure**
  - Unified cache
  - Demand fetch
  - Write-back
  - LRU replacement
  - Fully associative for large blocks; 4-way set associative for 4-8 byte blocks
- **The numbers are, in some sense, a convenient fiction**
  - Cache performance depends on workload and detailed machine characteristics
  - But, DTMR is useful to develop intuition

DTMR Benchmark Mix (based on Smith, 1985)

- **49 traces**
  - Variety of applications
  - Includes Operating System effects
- **6 machine architectures**
  - IBM S/370, IBM S/360, Vax, M68000, Z8000, CDC 6400
- **7 languages**
  - Fortran, 370 Assembler, APL, C, LISP, AlgolW, Cobol
- **Suggested use for DTMR:**
  - Provides a conservative baseline for estimating good cache parameters
    - Paper concludes "caches always work", and are worthwhile to use
  - Gives a starting point for detailed simulations
- **Data tables are in Appendix A of Flynn**
  - Also gives adjustment multipliers for various parameters

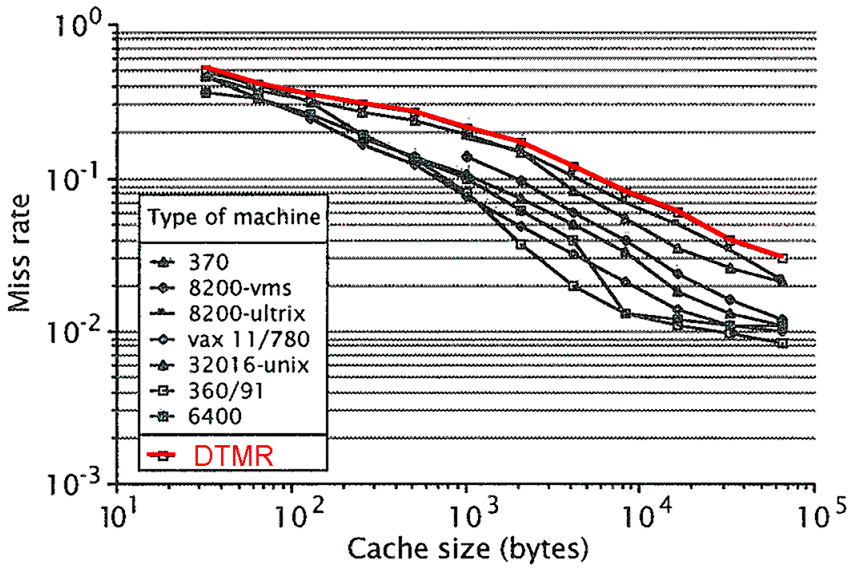
Effect of Application Environments

- **Fully associative, 16-byte blocks** (Flynn Fig. 5.11)



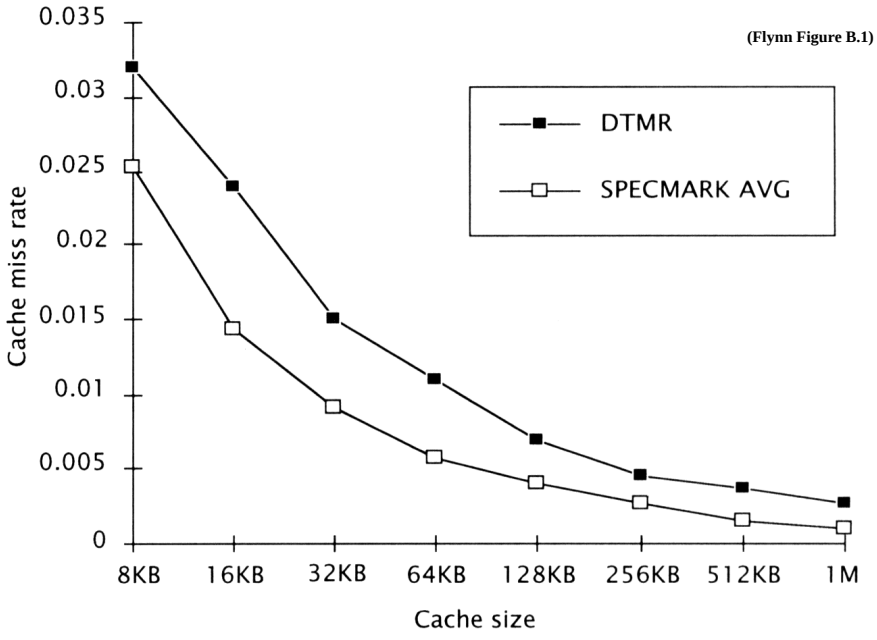
Effect of Machine

(Flynn Fig. 5.12)



DTMR Compared to SPECMARK

(Flynn Figure B.1)



Diminishing Returns

- **Even for "well behaved" programs you get diminishing returns by increasing cache size**
  - baseline DTMR data for 8x cache size increases:
    - 20% miss ratio with 1 KB cache
    - 7.5% miss ratio with 8 KB cache -- 2.7x improvement for 8x size increase
    - 3% miss ratio with 64 KB cache -- 2.5x improvement for 8x size increase
    - 1.5% miss ratio with 512 KB cache -- 2.0x improvement for 8x size increase+
  - And, of course, larger memory arrays will cycle more slowly...

- **Prediction:**
  - Eventually cache memory will run out of steam, and we'll need some other technology to bridge the main-memory/CPU speed gap
  - But, that's a problem for another course...

INSTRUCTION CACHE

Making the Most of Limited Cache

	SPARC Integer	SPARC Floating Point	MIPS Integer	MIPS Floating Point	IBM S/360	VAX	Average
Instruction	.79	.80	.76	.77	.50	.65	**.71**
Data Read	.15	.17	.15	.19	.35	.28	**.22**
Data Write	.06	.03	.09	.04	.15	.07	**.07**

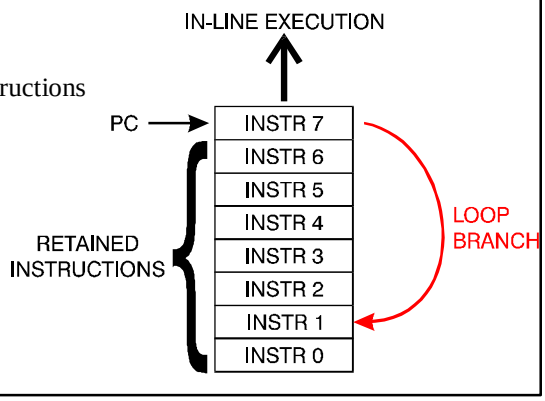
- **Optimize for the Common Case:** (Cragon Table 1.1)
  - Instructions ~71%
  - Data Read ~22%
  - Data Write ~7%
    - » (But, don't forget Amdahl's Law -- data ends up being important too!)

Exploiting Instruction Locality

- **Instruction buffers: for look-behind**
- **Instruction queues: prefetching**
- **Instruction caches: look-ahead & look-behind**

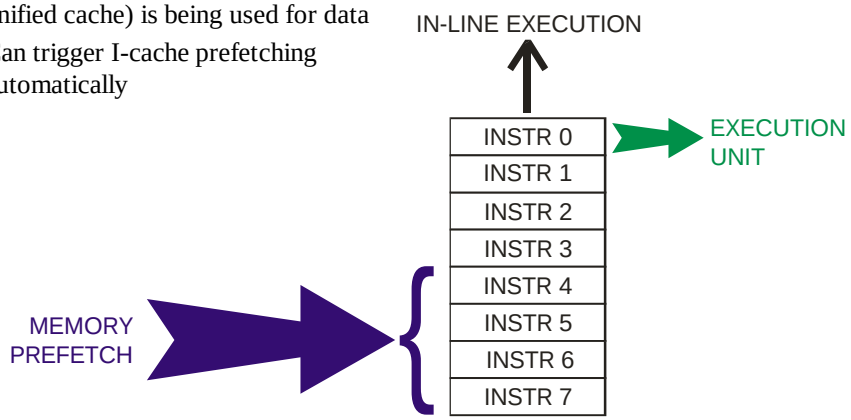
Instruction Buffer (Loop Cache)

- **Retains last *n* instructions executed in FIFO queue**
- **Short backward branches freeze queue and execute from it**
- **Useful for cache-less processors running scientific code**
  - CDC 6600 had 8 60-bit registers in the instruction queue
    - Up to: sixteen 30-bit instructions; thirty-two 15-bit instructions; combinations
  - Cray 1 buffered 256 16-bit instructions



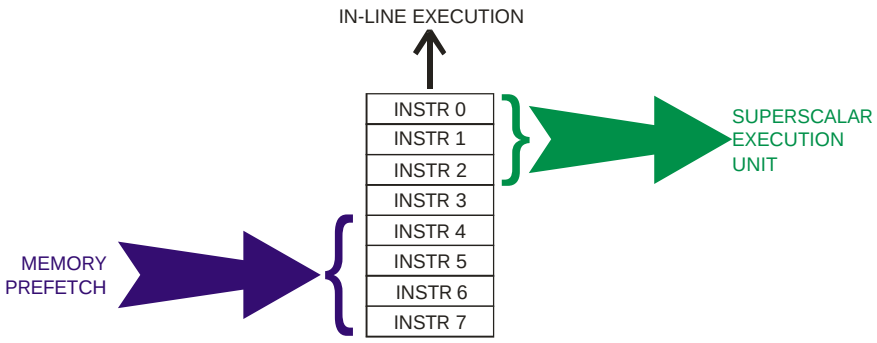
Instruction Queues (prefetch)

- **Can provide small but effective I-cache mechanism**
  - Prefetches sequential instructions in advance
  - Can keep instructions flowing (to a degree) even if bus (or single-ported unified cache) is being used for data
  - Can trigger I-cache prefetching automatically



Instruction Queues in Superscalar CPUs

- **Aids in decoding variable-length & multiple instruction issue**
  - Instruction decode is relative to head of queue
    - Cache & VM misses decoupled from instruction decoding
  - Can smooth memory bandwidth demands (to a degree), even if large instructions are issued quickly
  - Pentium prefetches both in-line and one branch-target stream



Trace Cache

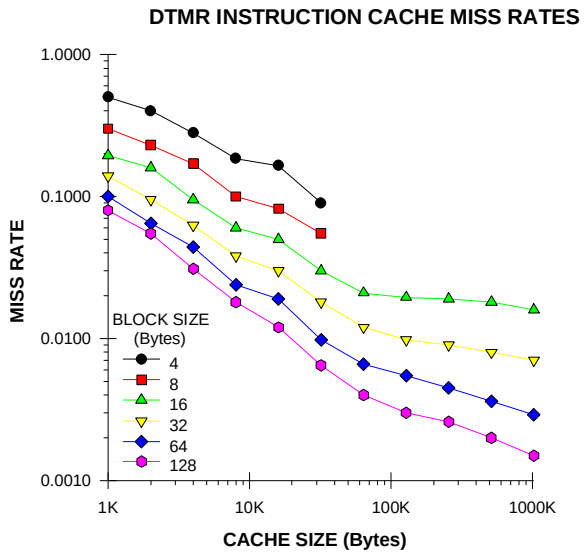
- **Capture paths (traces) through memory**
  - Expansion on prefetch queue idea
    - Prefetches based on branch target prediction
    - Retains paths taken through instruction memory in a cache so it will be there next time the first instruction of the trace segment is encountered
  - Includes effects of branch target prediction
  - Name comes from trace scheduling (multiflow VLIW machine)

Instruction-Only Caches

- **Separate cache just for instructions**
  - Full cache implementation with arbitrary addressability to contents
  - Single-ported cache used at essentially 100% of bandwidth
    - Every instruction has an instruction
    - But not every instruction has a data load/store...
- **Often implemented assuming only reads**
  - Consistency can be a problem if writes actually happen (and they do ... discussed in a few minutes)

Instruction Cache DTMR

- **Associative, demand fetch, write back, LRU**

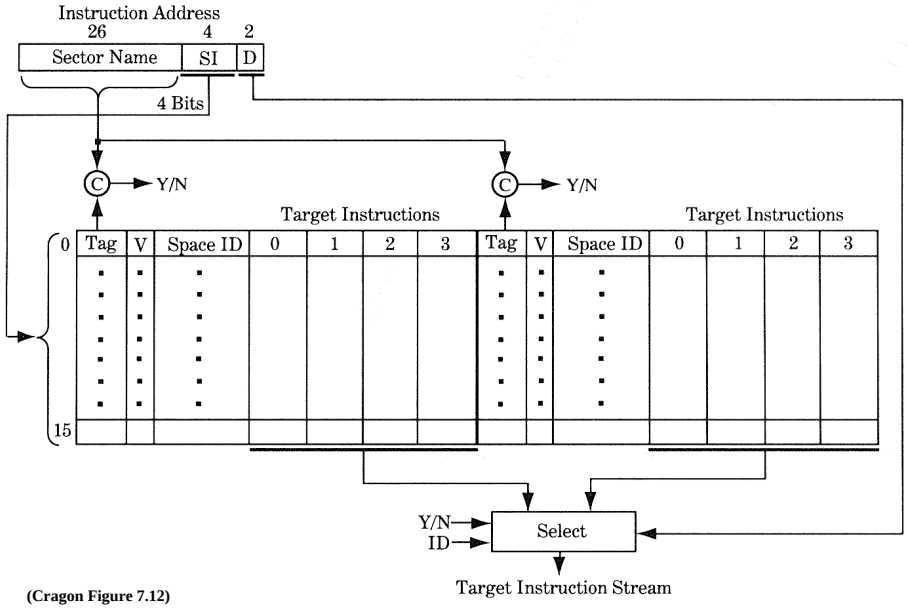


(After Flynn Fig. 5.28)

Branch Target Cache

- **Special I-cache -- holds instructions at branch target**
  - Used in AMD 29000 to make most of very small I-cache, no D-cache
    - Embedded controller; low cost (*e.g.*, laser printers)
- **Hides latency of DRAM access**
  - In-line instructions fetched in page mode from DRAM
  - Branching causes delay for new DRAM page fetch
  - Branch Target Cache keeps instructions flowing during DRAM access latency
- **Used in conjunction with branch prediction strategies**
  - AMD 29000 predicts branch taken if BTC hit; otherwise keeps fetching in-line
  - Branch prediction details beyond scope of this course

AMD 29000 Branch Target Cache



(Cragon Figure 7.12)

DATA CACHE

Data Caches

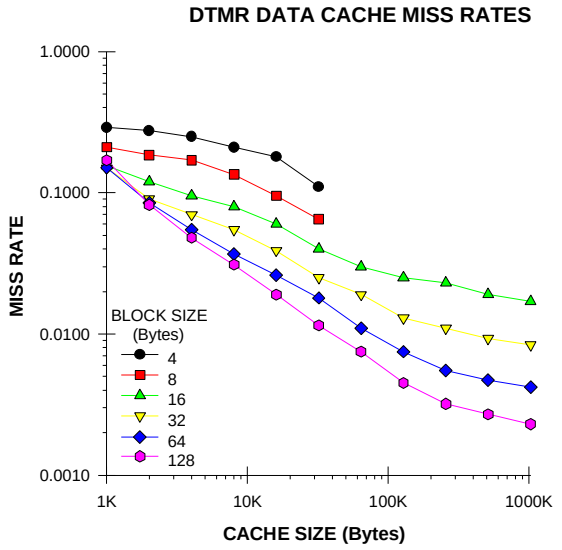
- **Must support reads & writes**
  - Approximately 75% of data accesses reads
  - Approximately 25% of data accesses writes

	SPARC Integer	SPARC Floating Point	MIPS Integer	MIPS Floating Point	IBM S/360	VAX	Average
Instruction	.79	.80	.76	.77	.50	.65	**.71**
Data Read	.15	.17	.15	.19	.35	.28	**.22**
Data Write	.06	.03	.09	.04	.15	.07	**.07**

- **Probably dual-ported for superscalars with multiple concurrent loads/stores**
  - The two data elements probably aren't in same cache block

Data Cache DTMR

- **Fully associative, demand fetch, write allocate, write back, LRU**



(After Flynn Fig. 5.29)

Special Data Caches

- **Translation Lookaside Buffer**
  - Stores address translation information between virtual and physical addresses
- **"Stack Cache" used by CRISP processor (a.k.a. Hobbit chip)**
  - Kept top of activation record stack for C programs in small on-chip cache
  - 32-word cache gave 19% data hit rate
    - Memory-to-memory addressing model
    - One way of looking at it is hardware-managed instead of compiler-managed register allocation

D-Cache / I-Cache Consistency

- **I-cache contents can become stale if modified**
  - Self-modifying code
    - Hand-written code
    - Incremental compilers/interpreters
    - Just-In-Time compilation
  - Intermingled data & instructions (*e.g.*, FORTRAN)
- **Approaches with split I- & D-cache**
  - Ignore and have SW flush cache when necessary (*e.g.*, loader flushes)
  - Trap with page faults (if data & code aren't intermingled)
  - Permit duplicate lines; invalidate I-cache line on D-cache write
  - Do not permit duplicate lines; invalidate either I-cache or D-cache when other obtains copy
- **No-duplicate-line policy can hurt FORTRAN performance**
- **Or, you can just use a unified cache...**

Concept In Real Life...

- **Name a real-life situation that is analogous to a split cache situation**
  - There should be a distinction between the sides of the "split"
  - Have you noticed problems with load-balancing?

UNIFIED CACHE

Split or Unified Cache?

- **Split cache**
  - Separate I-cache optimized for Instruction stream
  - Separate D-cache optimized for read+write
  - Can independently tune caches
  - Provides increased bandwidth via replication (2 caches accessed in parallel)

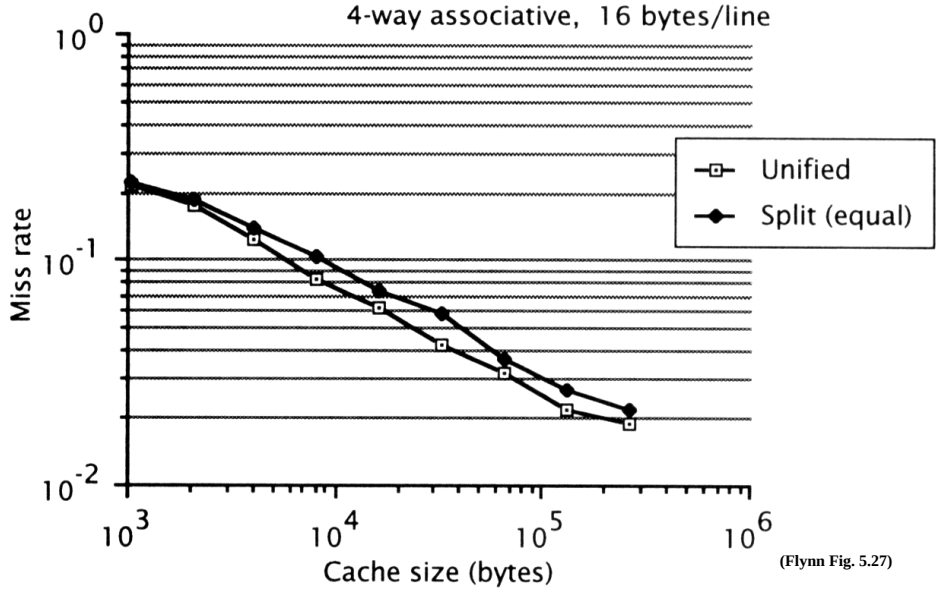
- **Unified cache**
  - Single cache holds both Instructions and Data
  - More flexible for changing instruction & data locality
  - No problem with instruction modification (self-modifying code, *etc.*)
  - Increased cost to provide bandwidth enough for instruction+data every clock cycle
    - Need dual-ported memory or cycle cache at 2x clock speed
    - Alternately, can take an extra clock for loads/stores for low cost designs; they don't happen for every instruction

Unified Caches

- **Instructions & Data in same cache memory**
- **Requires adding bandwidth for simultaneous I- and D-fetch, such as:**
  - Dual ported memory -- larger than single-ported memory
  - Cycle cache at 2x clock rate
  - Use I-fetch queue
    - Fetch entire block into queue when needed; larger than single instruction
    - 1-Cycle delay if I-fetch queue empty and need data
- **No problems with I-cache modifications; entire cache supports writes**
  - Single set of control logic (but, can have two misses in a single clock cycle)
- **Flexible use of memory**
  - Automatically uses memory for instruction or data as beneficial
  - Results in higher hit rate
- **Falling out of favor for L1 caches, but common for L2 caches**

Split vs. Unified Data

- **Unified size S compared to I-cache size S/2 + D-cache size S/2**



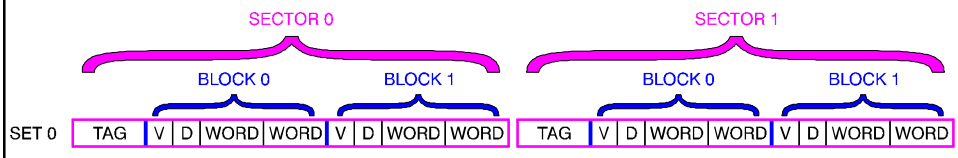
(Flynn Fig. 5.27)

Split & Unified TLBs

- **Similar tradeoffs to split & unified caches**
- **Split TLB provides address translation bandwidth**
  - Simultaneous Instruction & Data address translation
  - Can size TLBs depending on locality at the page level
    - Alpha 21164 has 64 D-TLB entries; 48 I-TLB entries
    - Pentium has 64 D-TLB entries; 32 I-TLB entries
- **Unified TLB provides more flexible allocation**
  - HP-PA 8000 has 96 entries in a unified TLB
  - Power PC 603e has 64 entries in a unified TLB

SECTORS & BLOCKS

Sectors Share Tag Among Blocks



- **Sectors reduce proportional tag overhead**
  - Single tag shared by several blocks; exploits spatial locality
- **H&P use of word "block" is actually for "sector"; many other authors as well**
  - "sub-block placement" equivalent to sector+block arrangement
  - "large" and "small" blocks usually equivalent to "large" and "small" sectors for performance trends.
- **Typical block size for on-chip cache now 32-64 bytes**

Why Large Sectors & Blocks?

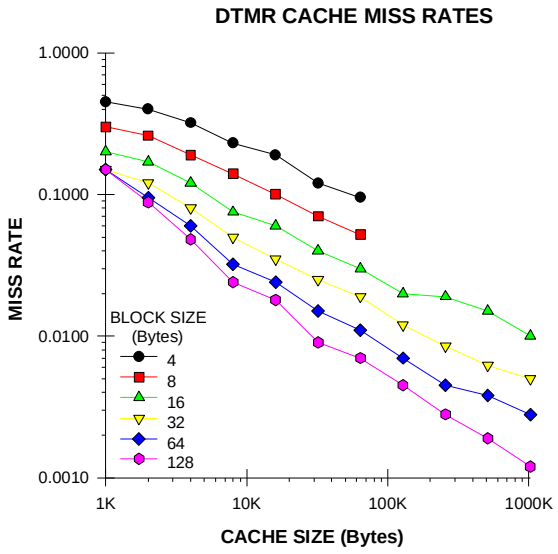
- **Reduced cost for tags**
  - Words per sector determines pro-rated overhead for tags
- **Large sector size**
  - Fewer tags needed
  - But, fewer unique locations to place data
    - P_{miss} tends to increase to extent that spatial locality is poor
- **Large blocks**
  - Fewer valid/dirty/shared bits needed
  - Exploits burst memory transfer modes
  - Provides bandwidth for I-fetching
    - Multiple instruction fetch for superscalar
    - Instruction queue load of multiple words
  - 64-bit or larger blocks provides double float load/store bandwidth
  - But, cache misses consume more fetch/store bandwidth (cache memory pollution)

Why Small Sectors & Blocks?

- **Reduces memory traffic & latency compared to larger blocks**
  - More flexibility in data placement, at cost of higher tag space overhead
- **Small sector size**
  - More unique locations to place data
  - But, more bits spent on tags (limiting case is 1 block/sector = 1 tag/block)
- **Smaller block size**
  - Simpler design (*e.g.*, direct mapped cache, block size of 1 word)
  - Fewer words to read from memory on miss
  - Fewer words to write to memory on write back eviction
  - Lower traffic ratio
  - But, does not exploit burst mode transfers; not necessarily fastest overall design

DTMR for Block Size

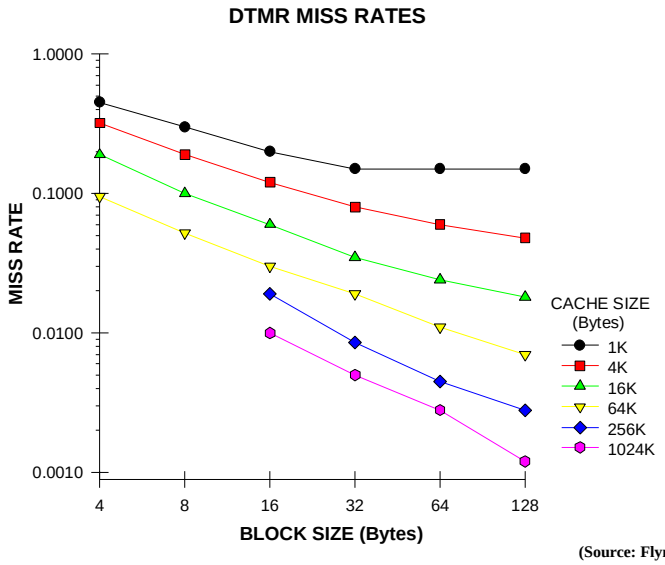
- **Associative, demand fetch, write allocate, write back, LRU**



(Source: Flynn Table A.1)

Miss Rate for Block Sizes

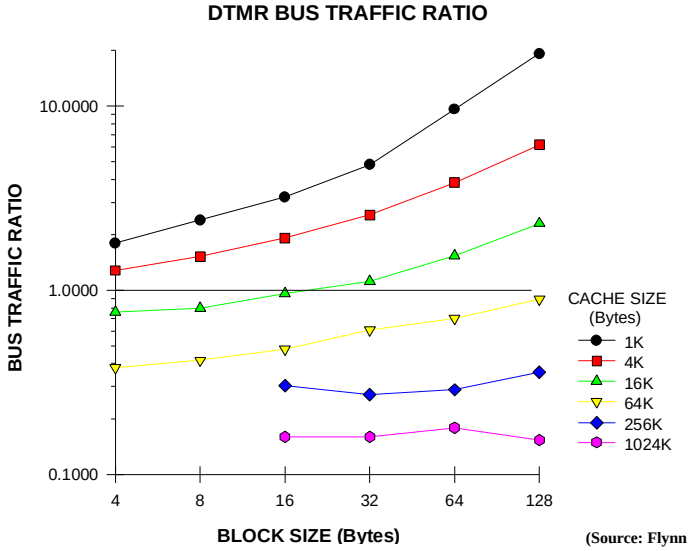
- **Miss rates go down until blocks are significant fraction of cache size**



(Source: Flynn Appendix A)

Bus Traffic & Block Size

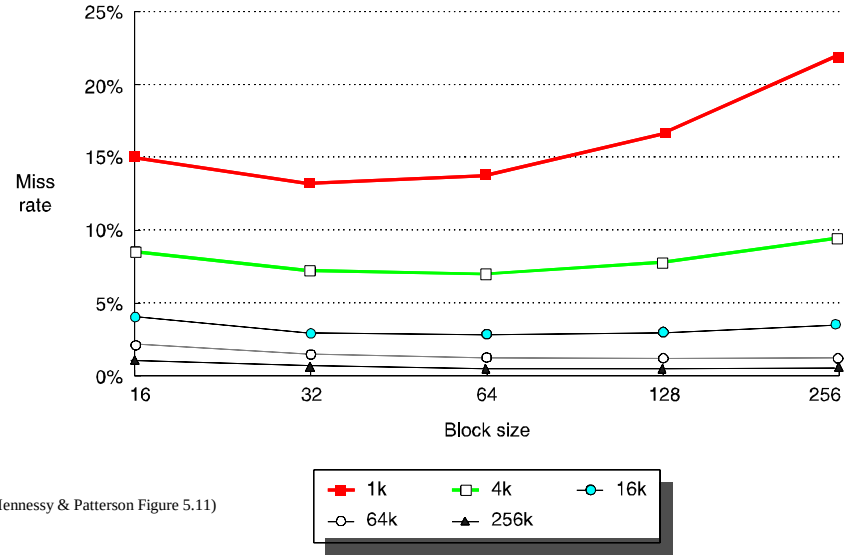
- **Bus traffic increases with block size, except (perhaps) with large caches**



(Source: Flynn Appendix A)

Effects of Extremely Large Sectors

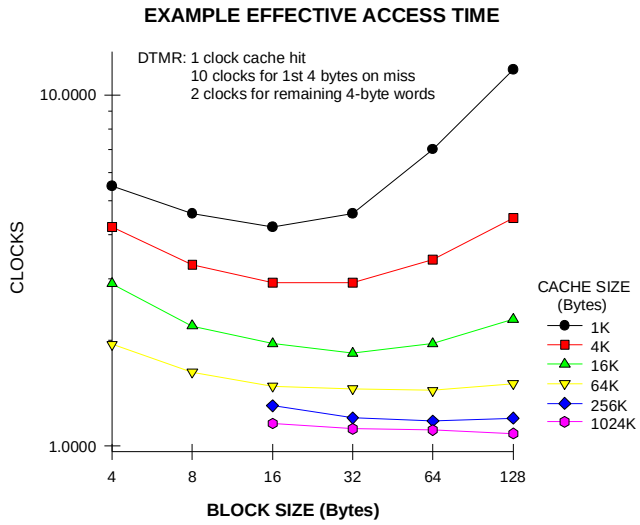
- **With only a few sectors in the cache, conflict misses increase dramatically**



(Hennessy & Patterson Figure 5.11)

Effective Access Time & Block Size

- t_{ea} goes down with larger block size as temporal locality is exploited
- t_{ea} goes back up when cache memory pollution becomes prevalent



Blocks Elsewhere

- **Virtual Memory System**
  - Page » Sector with 1 block
  - Large page may waste memory space if not fully filled
  - Small page has high overhead for address translation information (*e.g.*, requires more TLB entries for programs with good locality)

- **Disk Drives**
  - File system cluster (in DOS) » Cache Sector
  - Disk Sector » Cache block
  - Large sector size promotes efficient transfers, but wastes space with partially filled or only slightly modified sectors.

REVIEW

Review

- **Design Target Miss Rates**
  - A conservative starting point for designs, but a bit dated
- **Instructions and Data have different caching needs**
  - I-cache: prefetching, branches, "read-mostly"
  - D-cache: writes, poorer locality
  - Split cache gives bandwidth, unified cache gives flexibility
- **Cache sectors & blocks aren't quite the same**
  - Sectors account for amortized overhead of tags *vs.* miss rate
  - Block lengths are determined by data transfer widths and expected spatial locality
- **Sectors & Blocks**
  - Bigger is better -- when there is spatial locality
  - Smaller is better -- when there isn't enough spatial locality
    - Conflict misses when too few sectors in the cache
    - Traffic ratio goes up if not enough words are actually used from each block

Key Concepts

- **Latency**
  - It's pretty easy to get speedup by buffering instructions, even without a cache
- **Bandwidth**
  - Split I- & D- caches increase bandwidth, at cost of loss of flexibility
  - Larger blocks exploit any high-bandwidth transfer capabilities
- **Concurrency**
  - Split caches & split TLBs double bandwidth by using concurrent accesses
- **Balance**
  - Block size must balance miss rate against traffic ratio