ECE 2162
Caches
Memory Latency is *Long*

- 60-100ns not totally uncommon
- Quick back-of-the-envelope calculation:
  - 2GHz CPU
  - \( \rightarrow 0.5\text{ns / cycle} \)
  - 100ns memory \( \rightarrow 200 \) cycle memory latency!

- Solution: Caches
Locality and Caches

• Data Locality
  – Temporal: if data item needed now, it is likely to be needed again in near future
  – Spatial: if data item needed now, nearby data likely to be needed in near future

• Exploiting Locality: Caches
  – Keep recently used data in fast memory close to the processor
  – Also bring nearby data there
Storage (Memory) Hierarchy

- Core
  - I/DL1
- L2
- L3
- Memory

Speed +
Capacity -

Capacity +
Speed -
Cache Basics

• Fast (but small) memory close to processor
• When data referenced
  – If in cache, use cache instead of memory
  – If not in cache, bring into cache
    (actually, bring entire block of data, too)
  – Maybe have to kick something else out to do it!
• Important decisions
  – Placement: where in the cache can a block go?
  – Lookup: how do we find a block in cache?
  – Replacement: what to kick out to make room in cache?
  – Write policy: What do we do about stores?

Key: Optimize the average memory access latency
Cache Basics

• Cache consists of block-sized lines
  - Line size typically power of two
  - Typically 16 to 128 bytes in size

• Example
  - Suppose block size is 128 bytes
    • Lowest seven bits (of the word address) determine offset within block
  - Read data at address A=0x7ffffa3f4
  - Block address: 0x7ffffa380

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Blk_Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7ffffa380</td>
<td>7</td>
</tr>
</tbody>
</table>
Cache Lookup

• When address referenced, need to
  – Find whether its data is in the cache
  – If it is, find where in the cache
  – This is called a cache lookup

• Each cache line must have
  – A valid bit (1 if line has data, 0 if line empty)
    • We also say the cache line is valid or invalid
  – A tag to identify which block is in the line
    (if line is valid)
Simplistic Example

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest $k$ bits of the block address will index a block in the cache.
  - If the block is valid and the tag matches the upper $(m-k)$ bits of the $m$-bit address, then that data will be sent to the CPU.

- Here is a diagram of a $2^{10}$ cache blocks.
Cache Placement

• Placement
  – Which memory blocks are allowed into which cache lines

• Placement Policies
  – Direct mapped (block can go to only one line)
  – Fully Associative (block can go to any line)
  – Set-associative (block can go to one of N lines)
    • E.g., if N=4, the cache is 4-way set associative
    • Other two policies are extremes of this (E.g., if N=1 we get a direct-mapped cache)
Direct Mapped

- Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses.
2-way set associative cache

- Two tag comparators are needed.
Fully Associative Cache

- A fully associative cache is expensive to implement.
  - Because there is no index field in the address anymore, the entire block address must be used as the tag, increasing the total cache size.
  - Data could be anywhere in the cache, so we must check the tag of every cache block. That’s a lot of comparators!
Address Decomposition

- Let’s say we have a cache with $2^k$ blocks, each containing $2^n$ bytes.
- We can determine where a byte of data belongs in this cache by looking at its address in main memory.
  - $k$ bits of the address will select one of the $2^k$ cache blocks.
  - The lowest $n$ bits are now a block offset that decides which of the $2^n$ bytes in the cache block will store the data.

```
   (m+n)-bit Address
   (m-k) bits   k bits
     Tag        Index

   n-bit Block Offset
```
Cache Replacement

• Need a free line to insert new block
  – Which block should we kick out?

• Several strategies
  – Random (randomly selected line)
  – FIFO (line that has been in cache the longest)
  – LRU (least recently used line)
  – LRU Approximations
  – LFU
Approximating LRU

• LRU is pretty complicated (esp. for many ways)
  – Access and possibly update all counters in a set on every access (not just replacement)
• Need something simpler and faster
  – But still close to LRU
• NMRU – Not Most Recently Used
  – The entire set has one MRU pointer
  – Points to last-accessed line in the set
  – Replacement:
    Randomly select a non-MRU line
Approximating LRU

• Have LRU counter for each line in a set

• When line accessed
  – Get old value X of its counter
  – Set its counter to max value
  – For every other line in the set
    • Decrement it, but stops at 0

• When replacement needed
  – Select line whose counter is 0
Write Policy

• Do we update memory on writes?
  – Write-through
    • Memory immediately updated on each write
  – Write-back
    • Memory updated when line replaced

• Do we allocate cache lines on a write?
  – Write-allocate
    • A write miss brings block into cache
  – No-write-allocate
    • A write miss leaves cache as it was
Write-through caches

• A write-through cache solves the inconsistency problem by forcing all writes to update both the cache and the main memory.

• Why is this not always good?
  – Not efficient use of CPU-memory bandwidth
Write-back caches

- In a write-back cache, the memory is not updated until the cache block needs to be replaced (e.g., when loading data into a full cache set).
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory.
  - The cache block is marked “dirty” to indicate this inconsistency

\[ \text{Mem}[214] = 21763 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1101 0110</td>
<td>42803</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data.
Finishing the write back

- We don’t need to store the new value back to main memory unless the cache block gets replaced.
- For example, on a read from Mem[142], which maps to the same cache block, the modified cache contents will first be written to main memory.

Only then can the cache block be replaced with data from address 142.
Write-Back Caches

• Need a **Dirty** bit for each line
  – A dirty line has more recent data than memory
• Line starts as **clean** (not dirty)
• Line becomes dirty on first write to it
  – Memory not updated yet, cache has the only up-to-date copy of data for a dirty line
• Replacing a dirty line
  – Must write data back to memory (write-back)
Cache Performance

• Miss rate
  – Fraction of memory accesses that miss in cache
  – Hit rate = 1 – miss rate

• Average memory access time
  \[ \text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty} \]

• Memory stall cycles

\[
\text{CPUtime} = \text{CycleTime} \times (\text{Cycles}_{\text{Exec}} + \text{Cycles}_{\text{MemoryStall}})
\]

\[
\text{Cycles}_{\text{MemoryStall}} = \text{CacheMisses} \times (\text{MissLatency}_{\text{Total}} - \text{MissLatency}_{\text{Overlapped}})
\]
Improving Cache Performance

- AMAT = hit time + miss rate * miss penalty
  - Reduce miss penalty
  - Reduce miss rate
  - Reduce hit time

- Cycles_{MemoryStall} = CacheMisses x (MissLatency_{Total} - MissLatency_{Overlapped})
  - Increase overlapped miss latency
Reducing Cache Miss Penalty (1)

• Multilevel caches
  – Very Fast, small Level 1 (L1) cache
  – Fast, not so small Level 2 (L2) cache
  – May also have slower, large L3 cache, etc.

• Why does this help?
  – Miss in L1 cache can hit in L2 cache, etc.

\[
\text{AMAT} = \text{HitTime}_{L1} + \text{MissRate}_{L1} \times \text{MissPenalty}_{L1} \\
\text{MissPenalty}_{L1} = \text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2} \\
\text{MissPenalty}_{L2} = \text{HitTime}_{L3} + \text{MissRate}_{L3} \times \text{MissPenalty}_{L3}
\]
Multilevel Caches

• Global vs. Local Miss Rate
  – Global L2 Miss Rate
    # of L2 Misses / # of All Memory Refs
  – Local Miss Rate
    # of L2 Misses / # of L1 Misses
    (only L1 misses actually get to the L2 cache)
    (local misses should be used in previous equations)
  – MPKI often used (normalize against number of instructions)

• Exclusion Property
  – If block is in L1 cache, it is never in L2 cache
  – Saves some L2 space

• Inclusion Property
  – If block A is in L1 cache, it must also be in L2 cache
Inclusion vs. Exclusion in Multi-Level Caches

- **Inclusive caches**
  - Every block existing in the first level also exists in the next level
  - When fetching a block, place it in all cache levels. Tradeoffs:
    - Leads to duplication of data in the hierarchy: less efficient
    - Maintaining inclusion takes effort (forced evictions)
    - But makes cache coherence in multiprocessors easier
      - Need to track other processors’ accesses only in the highest-level cache

- **Exclusive caches**
  - The blocks contained in cache levels are mutually exclusive
  - When evicting a block, put it in the next level. Do a swap between the two
    - require all caches have the same line size
  + More efficient utilization of cache space
  + (Potentially) More flexibility in replacement/placement
  - More blocks/levels to keep track of to ensure cache coherence; takes effort
Maintaining Inclusion and Exclusion

• When does maintaining inclusion take effort?
  – When a block is evicted from L2, need to evict all corresponding subblocks from L1 \( \rightarrow \) keep 1 bit per subblock in L2 saying “also in L1”
  – When a block is inserted, make sure all higher levels also have it

• When does maintaining exclusion take effort?
  – When a block is inserted into any level, ensure it is not in any other
Reducing Cache Miss Penalty (2)

• Early Restart & Critical Word First
  – Block transfer takes time (bus too narrow)
  – Give data to loads before entire block arrive

• Early restart
  – When needed word arrives, let processor use it
  – Then continue block transfer to fill cache line

• Critical Word First
  – Transfer loaded word first, then the rest of block
    (with wrap-around to get the entire block)
  – Use with early restart to let processor go ASAP
Reducing Cache Miss Penalty (3)

• Increase Load Miss Priority
  – Loads can have dependent instructions
  – If a load misses and a store needs to go to memory, let the load miss go first
  – Need a write buffer to remember stores

• Merging Write Buffer
  – If multiple write misses to the same block, combine them in the write buffer
  – Use block-write instead of a many small writes
## Write Merge

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>Mem[100]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>Mem[108]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116</td>
<td>1</td>
<td>Mem[116]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>124</td>
<td>1</td>
<td>Mem[124]</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Kinds of Cache Misses

• The “3 Cs”
  – *Compulsory*: have to have these
    • Miss the first time each block is accessed
  – *Capacity*: due to limited cache capacity
    • Would not have them if cache size was infinite
  – *Conflict*: due to limited associativity
    • Would not have them if cache was fully associative
Reducing Cache Miss Penalty (4)

• Victim Caches: reducing conflict misses
  – Recently kicked-out blocks kept in small cache
  – If we miss on those blocks, can get them fast
  – Why does it work: conflict misses
    • Misses that we have in our N-way set-assoc cache, but would not have if the cache was fully associative
  – Example: direct-mapped L1 cache and a 16-line fully associative victim cache
    • Victim cache prevents thrashing when several “popular” blocks want to go to the same entry
Victim Cache – “Recycling”

- Holds victim blocks discarded from the L1 cache due to replacement.
- Small (otherwise an L2) and fully associative.
- Checked on a L1 miss. If found, block is swapped back to L1 (the block previously took its place is put into victim cache).
- Works better for small L1 caches since it saves victim blocks from conflict misses.
- Effective: a 4-entry vc can reduce ¼ of the misses in a 4KB L1 cache [Jouppi, 1990].
Reducing Cache Miss Penalty (5)
Handling Multiple Outstanding Accesses

• **Non-blocking or lockup-free** caches

• **Question**: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?

• **Idea**: *Keep track of the status/data of misses that are being handled in Miss Status Handling Registers (MSHRs)*
  – A cache access checks MSHRs to see if a miss to the same block is already *pending*.
    • If pending, a new request is not generated
    • If pending and the needed data available, data forwarded to later load
  – Requires buffering of outstanding miss requests
Non-Blocking Caches (and MLP)

- Enable cache access when there is a pending miss
- Enable multiple misses in parallel
  - Memory-level parallelism (MLP)
    - generating and servicing multiple memory accesses in parallel
  - Why generate multiple misses?
    - Enables latency tolerance: overlaps latency of different misses
  - How to generate multiple misses?
    - Out-of-order execution, multithreading, prefetching etc.
Overlap Miss Latencies

- Stall CPU on miss
- Hit under miss
- Multiple out-standing misses
Potential

© 2003 Elsevier Science (USA). All rights reserved.
Miss Status Handling Register

• Also called “miss buffer”
• Keeps track of
  – Outstanding cache misses
  – Pending load/store accesses that refer to the missing cache block
• Fields of a single MSHR
  – Valid bit
  – Cache block address (to match incoming accesses, associative search)
  – Issued bit (1 if already issued to memory)
  – For each pending load/store
    • Valid, type (load/store), format(byte/halfword/..), block offset, destination register for load OR store buffer entry for stores
# Miss Status Handling Register

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Load/store 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Block Address</td>
<td>Issued</td>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Load/store 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Block Address</td>
<td>Issued</td>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Type</td>
<td>Block Offset</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load/store 3</td>
</tr>
</tbody>
</table>
MSHR Operation

- On a cache miss:
  - Search MSHR for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall

- When a subblock returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - Deallocate load/store entry in the MSHR entry
  - Write subblock in cache or MSHR
  - If no pending load/store, deallocate MSHR (after writing the block in cache)
Reducing Cache Miss Rate (1)

- Larger blocks
  - Helps if there is more spatial locality
Reducing Cache Miss Rate (2)

• Larger caches
  – Fewer capacity misses, but longer hit latency!

• Higher Associativity
  – Fewer conflict misses, but longer hit latency
  – Way Prediction
    • Speeds up set-associative caches – also saves energy
    • Predict which of N ways has our data, fast access as direct-mapped cache
    • If mispredicted, access again as set-associative cache
Reducing Cache Miss Rate (2)

• Pseudo Associative Caches
  – Similar to way prediction
  – Start with direct mapped cache
  – If miss on “primary” entry, try another entry

• Compiler optimizations
  – Loop interchange
Loop Interchange

- For loops over multi-dimensional arrays
  - Example: matrices (2-dim arrays)
- Change order of iteration to match layout
  - Gets better spatial locality
  - Layout in C: last index changes first

\[
\begin{align*}
\text{for}(j=0; j<10000; j++) & \quad \text{for}(i=0; i<40000; i++) \\
\quad c[i][j] &= a[i][j] + b[i][j]; & \quad c[i][j] &= a[i][j] + b[i][j];
\end{align*}
\]

- $a[i][j]$ and $a[i+1][j]$ are 10000 elements apart
- $a[i][j]$ and $a[i][j+1]$ are next to each other
Reducing Cache Miss Rate (3): Prefetching

• Idea: fetch data into the cache before processors request them
  – Can address compulsory misses (next slide)
  – Can be done by the programmer, compiler, or hardware

• Characteristics of ideal prefetching
  – You only prefetch data that are truly needed
    • Avoid bandwidth waste
  – You issue prefetch requests early enough
    • To hide the memory latency
  – You don’t issue prefetch quest too early
    • To avoid cache pollution
Software Prefetching

for (i=0; i<N; i++) {
    _prefetch(a[i+8]);
    _prefetch(b[i+8]);
    sum+=a[i]*b[i];
}

• Issues software prefetching
  - Takes up issue slots
    • OK with superscalar
  - Takes up system bandwidth
  - Must have non-blocking caches
  - Prefetch distance depends on specific system implementation
    • Non-portable code
  - Not easy to use for pointer based structures
  - Requires skilled programmer/compiler
Hardware Prefetching

• Same goal with software prefetching but initiated by hardware
  – Can tune to specific system implementation
  – Does not waste instruction issue bandwidth
  – More portable code

• Major design questions
  – Where to place a prefetch engine?
    • L1, L2,…
  – What to prefetch?
    • Next sequential cache line(s), strided patterns, pointers, …
  – Where to place prefetched data?
    • In the caches or special prefetch buffer
Simple Sequential Prefetching

• On a cache miss, fetch two sequential memory blocks
  – Exploits spatial locality in both instruction & data
  – Exploits high bandwidth for sequential accesses
• Intel’s term: Adjacent cache line prefetch, or Spatial prefetch
• Extend to fetching $N$ sequential memory blocks
  – Pick $N$ large enough to hide memory latency
Stream Prefetching

• **Continuous** version of prefetching
  - Stream buffer can fit N cache lines
  - On a miss, start fetching N sequential cache lines
  - On a stream buffer hit:
    • Move cache line to cache, start fetching line (N+1)

• In other words, stream buffer tries to stay N cache lines ahead

• Design issues
  - When is a stream buffer allocated/released
  - Can use multiple stream buffers to capture multiple streams
    • E.g. a program operating on 2 arrays
Stream Buffers (Jouppi, ISCA 1990)

- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
  - if hit, pop the entry from FIFO, update the cache with data
  - if not, allocate a new stream buffer to the new miss address (may have to recycle a stream buffer following LRU policy)
- Stream buffer FIFOs are continuously topped-off with subsequent cache lines whenever there is room and the bus is not busy
- Can incorporate stride prediction mechanisms to support non-unit-stride streams
  - See “Evaluating stream buffers as a secondary cache replacement”, ISCA 1994
Strided Prefetching

• Idea: detect and prefetch strided accesses
  – For (i=0; i<N; i++) A[i*1024]++
Strided Prefetching

• Stride detected using a PC-based table
  – For each PC, remember the stride
  – Stride detection
    • Remember the last address used for this PC
    • Compare to currently used address for this PC
  – Track confidence using a two bit saturating counter
    • Increment when stride correct, decrement when wrong

• When stride is 1, stream prefetching
Sandybridge Prefetching (Intel Core i7-2600K)


Two hardware prefetchers load data to the L1 DCache:

• **Data cache unit (DCU) prefetcher.** This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.

• **Instruction pointer (IP)-based stride prefetcher.** This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to 2K bytes.
Other Ideas of Prefetching

• Prefetch for pointer-based data structures
  – Predict if fetched data contain a pointer and follow it
  – Works for linked-lists, graphs, etc
  – Must be very careful
    • What is a pointer
    • How far to prefetch
Reducing Hit Time (1)

- Small & Simple Caches are faster
Reducing Hit Time (2)

• Avoid address translation on cache hits
• Software uses virtual addresses, memory accessed using physical addresses
• HW must translate virtual to physical
  – Normally the first thing we do
  – Caches accessed using physical address
  – Wait for translation before cache lookup
• Idea: index cache using virtual address, physical or virtual tag comparison
Reducing Hit Time (3)

• Pipelined Caches
  – Improves bandwidth, but not latency
  – Essential for L1 caches at high frequency
    • Even small caches have 2-3 cycle latency at N GHz
  – Also used in many L2 caches

• Trace Caches
  – For instruction caches
Multi-ported Caches

• Idea: allow for multiple accesses in parallel
  – Processor with many LSUs, I+D access in L2, ...

• Can be implemented in multiple ways
  – True multi-porting
  – Multiple banks

• What is difficult about multi-porting
  – Interaction between parallel accesses (especially for stores)
True Multi-porting

• Example:
  - Use 2-ported tag and data array
  - Problem: large area increase
    • Area proportional to ports
  - Problem: hit time increase
Multi-banked Caches

• Banking (Interleaving)
  – Bits in address determines which bank an address maps to
    • Address space partitioned into separate banks
    • Which bits to use for “bank address”?
  + No increase in data store area
  -- Cannot always satisfy multiple accesses
    Why?
  -- interconnection network needed in input/output

• Bank conflicts
  – Two accesses are to the same bank
Sun UltraSPARC T2
8-bank L2 Cache
## Summary of Advanced Cache Optimizations

<table>
<thead>
<tr>
<th>Cache optimization</th>
<th>Miss rate</th>
<th></th>
<th></th>
<th>Miss penalty</th>
<th>Hit time</th>
<th>bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>compulsory</td>
<td>Capacity</td>
<td>Conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-level</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Victim Cache</td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-blocking</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical word first</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetching</td>
<td>+</td>
<td></td>
<td>-</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-porting</td>
<td></td>
<td></td>
<td></td>
<td>~</td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>