Principle of Locality: Memory Hierarchies

- Text and data are not accessed randomly
- Temporal locality
  - Recently accessed items will be accessed in the near future (e.g., code in loops, top of stack)
- Spatial locality
  - Items at addresses close to the addresses of recently accessed items will be accessed in the near future (sequential code, elements of arrays)
- Leads to memory hierarchy at two main interface levels:
  - Processor - Main memory -> Introduction of caches
  - Main memory - Secondary memory -> Virtual memory (paging systems)

Processor - Main Memory Hierarchy

- Registers: Those visible to ISA + those renamed by hardware
- (Hierarchy of) Caches: plus their enhancements
  - Write buffers, victim caches etc…
- TLB's and their management
- Virtual memory system (O.S. level) and hardware assists (page tables)
- Inclusion of information (or space to gather information) level per level
  - Almost always true

Questions that Arise at Each Level

- What is the unit of information transferred from level to level ?
  - Word (byte, double word) to/from a register
  - Block (line) to/from cache
  - Page table entry + misc. bits to/from TLB
  - Page to/from disk
- When is the unit of information transferred from one level to a lower level in the hierarchy?
  - Generally, on demand (cache miss, page fault)
  - Sometimes earlier (prefetching)

Questions that Arise at Each Level (c'ed)

- Where in the hierarchy is that unit of information placed?
  - For registers, directed by ISA and/or register renaming method
  - For caches, in general in L1
    - Possibility of hinting to another level (Itanium) or of bypassing the cache entirely, or to put in special buffers
- How do we find if a unit of info is in a given level of the hierarchy?
  - Depends on mapping;
  - Use of hardware (for caches/TLB) and software structures (page tables)

Questions that Arise at Each Level (c'ed)

- What happens if there is no room for the item we bring in?
  - Replacement algorithm; depends on organization
- What happens when we change the contents of the info?
  - i.e., what happens on a write?

Caches (on-chip, off-chip)

- Caches consist of a set of entries where each entry has:
  - block (or line) of data: information contents
  - tag: allows to recognize if the block is there
  - status bits: valid, dirty, state for multiprocessors etc.
- Capacity (or size) of a cache:
  number of blocks * block size

  i.e., the cache metadata (tag + status bits) is not counted in the cache capacity
- Notation
  - First-level (on-chip) cache: L1;
  - Second-level (on-chip/off-chip): L2; third level (Off-chip) L3

Cache Organizations

- Direct-mapped cache.
  - A given memory location (block) can only be mapped in a single place in the cache. Generally this place given by:

    (block address) mod (number of blocks in cache)
  - To make the mapping easier, the number of blocks in a direct-mapped cache is a power of 2.
  - There have been proposals for caches, for vector processors, that have a number of blocks that are Mersenne prime numbers (the modulo arithmetic for those numbers has some “nice” properties)

Cache Organizations (c’ed)

- Fully-associative cache.
  - A given memory location (block) can be mapped anywhere in the cache.
  - No cache of decent size is implemented this way but this is the (general) mapping for pages (disk to main memory), for small TLB’s, and for some small buffers used as cache assists (e.g., victim caches, write caches).

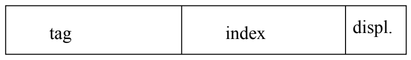
Cache Organizations (c’ed)

- Set-associative cache.
  - Blocks in the cache are grouped into sets and a given memory location (block) maps into a set. Within the set the block can be placed anywhere. Associativities of 2 (two-way set-associative), 3, 4, 8 and even 16 have been implemented.
  - Direct-mapped = 1-way set-associative
  - Fully associative with m entries is m-way set associative
- Capacity
  - Capacity = number of sets * set-associativity * block size

Cache Hit or Cache Miss?

- How to detect if a memory address (a byte address) has a valid image in the cache:
- Address is decomposed in 3 fields:
  - *block offset* or *displacement* (depends on block size)
  - *index* (depends on number of sets and set-associativity)
  - *tag* (the remainder of the address)
- The tag array has a width equal to *tag*

Hit Detection



Example: cache capacity C, block size b

Direct mapped: displ = $\log_2 b$; index = $\log_2(C/b)$; tag = 32 -index - displ

N -way S.A: displ = $\log_2 b$; index = $\log_2(C/bN)$; tag = 32 -index - displ

So what does it mean to have 3-way (N=3) set-associativity?

Why Set-associative Caches?

- Cons
  - The higher the associativity the larger the number of comparisons to be made in parallel for high-performance (can have an impact on cycle time for on-chip caches)
  - Higher associativity requires a wider tag array (minimal impact)
- Pros
  - Better hit ratio
  - Great improvement from 1 to 2, less from 2 to 4, minimal after that but can still be important for large L2 caches
  - Allows parallel search of TLB and caches for larger (but still small) caches (see later)

Replacement Algorithm

- None for direct-mapped
- Random or LRU or pseudo-LRU for set-associative caches
  - Not an important factor for performance for low associativity. Can become important for large associativity and large caches

Writing in a Cache

- On a write hit, should we write:
  - In the cache only (write-back) policy
  - In the cache and main memory (or higher level cache) (write-through) policy
- On a write miss, should we
  - Allocate a block as in a read (write-allocate*)*
  - Write only in memory *(*write-around*)*

The Main Write Options

- Write-through (aka store-through)
  - On a write hit, write both in cache and in memory
  - On a write miss, the most frequent option is write-around
  - Pro: consistent view of memory (better for I/O); no ECC required for cache
  - Con: more memory traffic (can be alleviated with write buffers)
- Write-back (aka copy-back)
  - On a write hit, write only in cache (requires dirty bit)
  - On a write miss, most often write-allocate (fetch on miss) but variations are possible
  - Pro-con reverse of write through

Classifying the Cache Misses: The 3 C's

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large blocks
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses.
- There is a fourth C: coherence misses (cf. multiprocessors)

Example of Cache Hierarchies

MICRO	L1	L2
Alpha 21064	8K(I), 8K(D), WT, 1-way, 32B	128K to 8MB,WB, 1-way,32B
Alpha 21164	8K(I), 8K(D), WT, 1-way, 32B ,D l-u fr.	96K, WB, on-chip, 3-way,32B,l-u free
Alpha 21264	64K(I), 64K(D),?, 2-way, ?	up to 16MB
Pentium	8K(I),8K(D),both, 2-way, 32 B	Depends
Pentium Pro	8k(I),8K(D), WB, 4-way(I),2-way(D), 32B,l-u free	256K,32B,4-way, tightly-coupled

Examples (c'ed)

PowerPC 620	32K(I),32K(D),WB 8-way, 64B	1MB TO 128MB, WB, 1-way
MIPS R10000	32K(I),32K(D),l-u, 2-way, 32B	512K to 16MB, 2-way, 32B