The Processor Memory Hierarchy

---

Copyright 2019, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.
Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Not in EaC2e.
Array Layout

Row-Major Order
• Lay out as a sequence of consecutive rows
• Rightmost subscript varies fastest
• A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Storage Layout

Stride One Access: successive references in the loop are to adjacent locations in virtual memory.
Stride one access maximizes spatial reuse & effectiveness of hardware prefetch units.
In row-major order, stride one is along rows.

Declared arrays in C (and most languages)
A couple of lectures back …

Array Address Calculations

In scientific codes, array address calculations are a major cost

• Each additional dimension adds more arithmetic

• Efficiency in address calculation is a critical issue for performance
  ♦ A[i+1,j], A[i,j], A[i,j+1] should all have some common terms
  ♦ Horner’s rule evaluation hides the common terms
    → Improving these calculations is a matter of algebra

• Improving the efficiency of array address calculations has been a major focus of code optimization (& hardware design) for the last 40 years
  ♦ Generate “better” code
  ♦ Transform it to fit the context
  ♦ Design memory systems that support common array access patterns
    → Optimize hardware for stride one access
    → Include sophisticated prefetch engines that recognize access patterns
### Does Layout Matter?

#### Which loop is faster?

- **Loop 1**:  
  ```c
  for (x=0; x<n; x++)
      for (y=0; y<n; y++)
          A[x][y] = 0;
  ```

- **Loop 2**:  
  ```c
  for (y=0; y<n; y++)
      for (x=0; x<n; x++)
          A[x][y] = 0;
  ```

- **Loop 3**:  
  ```c
  p = & A[0][0];
  t = n * n;
  for (x=0; x<t; x++)
      *p++ = 0;
  ```

- **Conventional wisdom**:  
  ```c
  bzero((void *) &A[0][0],(size_t) n * n * sizeof(int));
  ```

**All three loops have distinct performance**

- **Loop 1**:
  - 0.51 seconds on 10,000 x 10,000 array
  - ~5x faster compared to Loop 2

- **Loop 2**:
  - 1.65 seconds on 10,000 x 10,000 array
  - ~15x slower than Loop 1

- **Loop 3**:
  - 0.11 seconds on 10,000 x 10,000 array
  - ~15x faster than Loop 2

**All data collected on a quiescent, multiuser Intel T9600 @ 2.8 GHz using code compiled with gcc 4.1 –O3**

A couple of lectures back …
A couple of lectures back ...

Address Space Layout

Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).
- Arrays live on one of the stacks, in the global area, or in the heap.

The picture shows one virtual address space.

- The hardware supports one virtual address space per process.
- How does a virtual address space map into physical memory?
Multiple Virtual Address Spaces?

The Big Picture

- **Compiler’s view**
- **OS’ view**
- **1980 Hardware view**

**TLB** is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of **TLB**.

**virtual address spaces** (one per process)

**Physical address space** (big vector of memory)
Unified address space

TLB is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.

Physical address space (big vector of memory)

Mapping Virtual Address Spaces

Add a couple of high-end bits so we can treat the process address spaces as one large address space for mapping purposes.
Mapping Virtual Address Spaces

Of course, the “Hardware view” is no longer that simple
• Multiple levels of cache
• Caches shared among cores
• Caches exclusive to one core
• Multiple levels of (shared) TLB

All of the addresses must map in a way that works for the code generated to run in a single virtual address space

Most processors have > 1 core
Cache Memory

Modern hardware features multiple levels of cache & of TLB

- **L1** is typically private to one core
  - Separate L1 code & data caches
- **L2** (and beyond) is typically shared between cores and between instructions (I) and data (D)
  - Most chips use inclusive caches
    - Item in L1 ⇒ in L2 ⇒ in L3 ...
    - Some chips use exclusive caches
- Most caches are set associative
  - 2, 4 or 8 way
- **TLBs** are also associative
  - Little documentation
  - Difficult to diagnose
Cache Memory

The primary function of a cache is to provide fast memory “near” the core

• **L1** is “a couple of cycles” and small
• **L2** is slower than **L1** and larger; **L3** is slower, again, and larger
• **RAM** is even slower and going to disk is glacial

My Laptop:

- 4 cycles
- 12 cycles
- 21 cycles
Cache Memory

The primary function of a cache is to provide fast memory “near” the core

• L1 is “a couple of cycles” and small
• L2 is slower than L1 and larger; L3 is slower, again, and larger

The other function of a cache is to map addresses

• Cache is organized into blocks, or lines
• Each line consists of a tag and a set of words
Cache Memory

The primary function of a cache is to provide fast memory “near” the core
- **L1** is “a couple of cycles” and small
- **L2** is slower than **L1** and larger; **L3** is slower, again, and larger

The other function of a cache is to map addresses
- Cache is organized into blocks, or lines
- Each line consists of a tag and a set of words

A full cache is a set of lines
- Address maps into tag, index, and offset

To make good use of cache memory, the code must reuse values.
Spatial reuse refers to the use of more than one word in a line.
Temporal reuse refers to reuse of the same word over time.
Cache Memory

Caches differ in how they apportion the tag and index bits

- A **direct-mapped** cache has one line per **index**
- Cache lookup is simple
  - *The index bits are an ordinal index into the set of lines*

**Direct-mapped cache**

A direct mapped cache has $2^s$ lines.

*Capacity* is the sum of the sizes of the lines.
Cache Memory

Caches differ in how they apportion the tag and index bits

- A set-associative cache has multiple lines per index
  - index maps to a set, lookup matches tags within the set
  - Small associative tag memory for each set (parallel search)

2-way Set-Associative Cache

A set-associative cache has \(2^s\) sets. For a given total size, \(s\) is smaller than in direct mapped. The tag is longer; the index is shorter.
Cache Memory

Caches differ in how they apportion the tag and index bits

- A **set-associative** cache has multiple lines per **index**

**4-way Set-Associative Cache**

```
Way 0
Set 0:  tag data data ...
Set 1:  tag data data ...
Set 2:  tag data data ...
Set 2*: tag data data ...

Way 1
Set 0:  tag data data ...
Set 1:  tag data data ...
Set 2:  tag data data ...
Set 2*: tag data data ...

Way 2
Set 0:  tag data data ...
Set 1:  tag data data ...
Set 2:  tag data data ...
Set 2*: tag data data ...

Way 3
Set 0:  tag data data ...
Set 1:  tag data data ...
Set 2:  tag data data ...
Set 2*: tag data data ...
```
Cache Memory

Caches differ in how they apportion the tag and index bits

- A **set-assOCIATIVE** cache has multiple ways.

**4-way Set-Associative Cache**

Why not go to 8-way, or 16-way, or 32-way, or fully associative

- Each set has a small associative memory to allow parallel search on the tag bits
- The logic to do that lookup in parallel grows as the \( \log_2 \) of the number of ways
  - Each doubling of the number of ways adds another layer to the network of gates
  - Larger associativity means longer access times (**more cost, no gain**)
- Direct, 2-way, & 4-way are common
The Role of the Translation Lookaside Buffer (TLB)

• **TLB** is a small **cache** that maps virtual addresses to physical addresses
  ♦ Holds a subset of the *(active)* virtual pages that are in memory
  ♦ Each line contains a physical page address, tag is the full virtual address†

• In a **load**, the lookup in a physically addressed cache uses the **TLB** to translate the virtual address to a physical address
  ♦ A **TLB** hit allows the load to proceed
  ♦ A **TLB** miss invokes a search to bring the desired page into the **TLB**, followed by a continuation or a re-issue of the **load**
    → **TLB holds only recently accessed pages** *(locality matters, a lot)*

• Lookup in a direct-mapped **L1** cache
  ♦ Must translate virtual to physical, then lookup line by the index bits
  ♦ If tags match, return the line; if not, fetch the line and replace old contents

• Lookup in an associative **L1** cache
  ♦ Must translate virtual to physical, then lookup index, then match tags

† Remember the high-order bits we added to unify the multiple virtual address spaces? They keep the **TLB** entries for different processes distinct.
The Role of the Translation Lookaside Buffer (TLB)

What happens on a TLB miss?

• If a load misses in the TLB, the next step is to consult the page table

• Page table is a list of virtual address, physical address pairs
  ♦ Walk the list, looking for the virtual address, at page granularity
  ♦ If we find the page, we swap that mapping into the TLB & proceed as in a TLB hit

To facilitate the page table lookup:

• The page table is usually ”locked down” in L2 or L3 cache

• Some processors provide hardware assist to search the page table
  ♦ Implies page table format is rigid and specified
  ♦ Page table can be quite large — how many 4KB pages in 8GB of RAM?

• Nonetheless, page table lookup is quite slow, compared to a hit in L1

And, the processor may miss in the page table ... (more later)
⇒ Causing a virtual memory miss (VM miss)
What Happens on a Load?

The hardware must find the data in this complex hierarchy

• Assume that the address is in a register, e.g. load $r_0 => r_1$
• Assume the addresses are translated appropriately

Sequence of Events for a load

1. Processor looks in L1 cache  
   ♦ *Index maps to a set, then an associative search on the tags in the set*  
   ♦ *If found (a cache hit), return the value; otherwise ...*

2. Processor looks in L2 cache  
   ♦ *Index maps to a set, then an associative search on the tags in the set*  
   ♦ *If found (a cache hit), return the value; otherwise ...*

3. And so on ...
What Happens on a Load?

What if the page is not in cache?

- The **hardware** loads the cache line from **RAM**
  - Assumes that the virtual page is already loaded into a physical page frame in **RAM**
  - If not, see “**VM miss**”

With inclusive caches:

- The cache line containing the data is loaded into all the levels of cache
- This act will trigger evictions through the replacement algorithm

With exclusive caches:

- The cache line containing the data is loaded into the **L1** cache
- This act will trigger an eviction in **L1**, which may ripple up through **L2 & L3**

If the page is not in **RAM**, ...
What Happens on a Load? (VM Miss)

What if the page is not in memory (RAM)?

• The OS must bring the page into memory from its swap space
  ♦ Swap space is an OS-managed file on disk

• When the OS created the address space, it allocated address space to it
  ♦ Remember fork() and exec()?
  ♦ As exec() fills the address space, those bytes populate the process’ swap space
  ♦ The OS has a map of where all those pages are in swap space

• The OS initiates a transfer from swap space to RAM
  ♦ Spinning disk must wait on rotational delay plus transfer time
  ♦ Solid state disk avoids rotational delay; likely to be faster
  ♦ When the data reaches RAM, the processor must pull it into cache

• The load must wait until the data hits the L1 cache

The takeaway: Locality matters (a lot)
What Happens on a Load?

What about virtual to physical address translation?

• The address in the load is a virtual address
• If the load misses in all caches, we need a physical address
• Caches can be designed to operate on either virtual or physical addresses
  ♦ L1 is typically indexed by virtual addresses
    → Removes TLB lookup from the critical path on an L1 cache access
  ♦ L2 and above are typically indexed by physical addresses

Physically-addressed cache ⇒ virtual address translation during lookup

• Involves understanding the map from virtual pages to physical pages
• Involves cooperation between hardware and the operating system
  ♦ Worst case behavior involves walking the page tables (often locked in L2 or L3)
  ♦ Design of virtual memory systems is covered in a good OS course
What Happens on a Load?

Careful design can let the TLB lookup & index set lookup run in parallel

- By playing with the size of \( t \), \( s \), and \( o \), the cache designer can separate index lookup from virtual-to-physical translation
- If \( s + o = \log_2(\text{pagesize}) \) then the index and offset bits are the same in physical & virtual addresses
  - If \( s + o = \log_2(\text{pagesize}) \), then the processor can start both the L1 lookup to find the set and the TLB lookup to translate the address at the same time
  - By the time it has found the set, it should have the tag data from the physical address (unless the lookup misses in the TLB)
  - In effect, associativity lets cache capacity grow without increasing the number of bits in the index field of the address

- Do manufacturers play this game? Absolutely.
  - My laptop has a 32KB L1 cache, with 64 byte lines, for 512 lines.
  - It is 8-way set associative, which means 64 sets.
  - Thus, \( s = 6 \), \( o = 6 \), and \( s + o = 12 \) bits. \( 2^{12} = 4,096 \), which is the pagesize.

A 4-way 32KB cache would need 2x sets or 2x line size (1 more bit in \( s \) or \( o \)) to keep \( s + o = 12 \).