Memory Hierarchy & Caching

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
§ Motivation
again, recall the Von Neumann architecture
— a stored-program computer with programs and data stored in the same memory
“memory” is an *idealized* storage device that holds our programs (instructions) and data (operands)
colloquially: “RAM”, random access memory
~ big array of byte-accessible data
execlp("/bin/echo", "/bin/echo", "hello", NULL);

but our code & data clearly reside on the hard drive (to start)!
in reality, “memory” is a combination of storage systems with very different access characteristics
common types of “memory”:
SRAM, DRAM, NVRAM, HDD
SRAM

- Static Random Access Memory
- One bit stored per flip-flop
  ~6+ transistors (switches) per flip-flop
- Data stable as long as charge applied
- Relatively simple design — very fast!
DRAM

- **Dynamic Random Access Memory**
- One bit per *capacitor*
- Higher density than SRAM
  - Much cheaper
- Volatile and unstable; requires “refresh”
NVRAM

- **Non-Volatile Random Access Memory**
- Data maintained without power
  - e.g., compact flash, SSD
- Limited writes per block (1M+)
- Slower than DRAM
HDD

- Hard Disk Drive
- Spinning magnetic platters with multiple read/write heads
  - Data access requires mechanical seek
- Super dense, super slow!
## Relative Speeds

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Access latency</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>8 - 32 words</td>
<td>0 - 1 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>On-board SRAM</td>
<td>32 - 256 KB</td>
<td>1 - 3 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>Off-board SRAM</td>
<td>256 KB - 16 MB</td>
<td>~10 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>DRAM</td>
<td>128 MB - 64 GB</td>
<td>~100 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>SSD</td>
<td>≤ 1 TB</td>
<td>~10,000 cycles</td>
<td>(µs)</td>
</tr>
<tr>
<td>HDD</td>
<td>≤ 4 TB</td>
<td>~10,000,000 cycles</td>
<td>(ms)</td>
</tr>
</tbody>
</table>
### Numbers Every Programmer Should Know

[http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html](http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html)

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory reference: 100ns</td>
<td>100ns</td>
</tr>
<tr>
<td>L1 cache reference: 1ns</td>
<td>1,000ns = 1μs</td>
</tr>
<tr>
<td>Branch mispredict: 3ns</td>
<td>2,000ns = 2μs</td>
</tr>
<tr>
<td>L2 cache reference: 4ns</td>
<td>10,000ns = 10μs</td>
</tr>
<tr>
<td>Mutex lock/unlock: 17ns</td>
<td>100ns</td>
</tr>
<tr>
<td>Send 2,000 bytes over commodity network: 1,000ns = 0.7μs</td>
<td></td>
</tr>
<tr>
<td>SSD random read: 15,000ns = 16μs</td>
<td></td>
</tr>
<tr>
<td>Read 1,000,000 bytes sequentially from memory: 10,000ns = 19μs</td>
<td></td>
</tr>
<tr>
<td>Disk seek: 4,000,000ns = 4ms</td>
<td></td>
</tr>
<tr>
<td>Read 1,000,000 bytes sequentially from disk: 2,000,000ns = 2ms</td>
<td></td>
</tr>
<tr>
<td>Packet roundtrip CA to Netherlands: 500,000,000ns = 500μs</td>
<td></td>
</tr>
<tr>
<td>1,000,000ns = 1ms = 1</td>
<td></td>
</tr>
</tbody>
</table>
Western Digital WD Black WD1002FAEX 1TB 7200 RPM SATA 6.0Gb/s 3.5" Internal Hard Drive -
- 7200 RPM 84MB Cache
- SATA 6.0Gb/s
- For Power Computing

Extra savings w/ promo code EMCXTVP23, ends 4/1

$99.99
Save: 5%
Free Shipping

$104.99

OCZ Octane OCT-25SAT3-1T 2.5" 1TB SATA III 2Xnm Synchronous Mode Multi-Level Cell (MLC)
- 2.5"
- 1TB
- SATA III

from $2,299.99
Free Shipping

Kingston HyperX Beast 64GB (8 x 8GB) 240-Pin DDR3 SDRAM DDR3 2133 Desktop Memory (with KHX-)
- DDR3 2133
- Timing 11-12-11
- Cas Latency 11

$599.99
Free Shipping

(16 × $599.99 ≈ $9600)

(from newegg.com)
would like:

1. a lot of memory
2. fast access to memory
3. to not spend $$$ on memory
an exercise in compromise: the memory hierarchy
idea: use the *fast but scarce* kind as much as possible; fall back on the *slow but plentiful* kind when necessary
boundary 1: SRAM ⇔ DRAM
§Caching
cache  |kaSH|
verb
store away in hiding or for future use.
cache |kaSH|
noun
• a hidden or inaccessible storage place for valuables, provisions, or ammunition.
• (also cache memory) Computing an auxiliary memory from which high-speed retrieval is possible.
assuming SRAM cache starts out empty:

1. CPU requests data at memory address $k$
2. Fetch data from DRAM (or lower)
3. Cache data in SRAM for later use
after SRAM cache has been populated:

1. CPU requests data at memory address \( k \)
2. Check SRAM for cached data first; if there (“hit”), return it directly
3. If not there, fetch from DRAM
essential issues:

1. *what* data to cache

2. *where* to store cached data;
   i.e., how to *map* address $k \rightarrow$ cache slot
   - keep in mind SRAM $\ll$ DRAM
1. take advantage of *localities of reference*
   
   a. *temporal* locality  
   b. *spatial* locality
a. **temporal** (time-based) locality:

- if a datum was accessed recently, it’s likely to be accessed again soon

- e.g., accessing a loop counter; calling a function repeatedly
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}

movl $0x0000000a,0xf8(%rbp) ; store n
movl $0x00000001,0xf4(%rbp) ; store fact
jmp 0x100000efd

movl 0xf4(%rbp),%eax         ; load fact
movl 0xf8(%rbp),%ecx         ; load n
imull %ecx,%eax               ; fact * n
movl %eax,0xf4(%rbp)          ; store fact
movl 0xf8(%rbp),%eax
subl $0x01,%eax
movl %eax,0xf8(%rbp)          ; store n
movl 0xf8(%rbp),%eax
cmpl $0x01,%eax
jg 0x100000ee8

(memory references in bold)
- 2 writes, then 6 memory accesses per iteration!
- map addresses to cache slots
- keep required data in cache
- avoid going to memory
- may need to write data back to free up slots
- occurs without knowledge of software!
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}

compiler optimization: registers as “cache”
reduce/eliminate memory references in code
using registers is an important technique, but doesn’t scale to even moderately large data sets (e.g., arrays)
one option: manage cache mapping directly from code

```assembly
;; fictitious assembly
movl $0x00000001,0x0000(%cache)
movl $0x0000000a,0x0004(%cache)
imull 0x0004(%cache),0x0000(%cache)
decl 0x0004(%cache)
compl $0x01,0x0004(%cache)
jne 0x100000f10
movl 0x0000(%cache),0xf4(%rbp)
movl 0x0004(%cache),0xf8(%rbp)
```
awful idea!

- code is tied to cache implementation; can’t take advantage of hardware upgrades (e.g., larger cache)

- cache must be shared between processes (how to do this efficiently?)
caching is a hardware-level concern — job of the memory management unit (MMU) but it’s very useful to know how it works, so we can write cache-friendly code!
b. **spatial** (location-based) locality:

- after accessing data at a given address, data nearby are likely to be accessed

- e.g., sequential control flow; array access (with *stride n*)
```c
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

main() {
    int i, sum = 0;
    for (i=0; i<10; i++) {
        sum += arr[i];
    }
}
```

**stride length = 1 int (4 bytes)**

```
100001060 01000000 02000000 03000000 04000000
100001070 05000000 06000000 07000000 08000000
100001080 09000000 0a000000
```

```
100000f08 leaq 0x00000151(%rip),%rcx
100000f0f nop
100000f10 addl (%rax,%rcx),%esi
100000f13 addq $0x04,%rax
100000f17 cmpq $0x28,%rax
---100000f1b jne 0x100000f10
```
Modern DRAM is designed to transfer bursts of data (~32-64 bytes) efficiently.

Idea: transfer array from memory to cache on accessing first item, then only access cache!
2. *where* to store cached data?
   i.e., how to *map* address $k \rightarrow$ cache slot
§Cache Organization
<table>
<thead>
<tr>
<th>index</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
e.g., request for memory address 1001
e.g., request for memory address 1001
i) **direct** mapping

e.g., request for memory address 1001
i) **direct** mapping

- **Cache**
  - Index: 00, 01, 10, 11
  - Address: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

- **Memory**
  - Address: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

- Example: Request for memory address **1001**
i) **direct** mapping

pre-determined mapping for each memory address
i) **direct** mapping

for cache with $2^n$ slots, use *lower n bits of address* to compute *cache index*
i) **direct** mapping

why not *upper* bits?
— defeats spatial locality
(adjacent data *collide*)
i) **direct mapping**

If you didn’t know,
- what address is $x$ from?
- is $x$ actually valid data?
i) **direct** mapping

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

must add some fields
- **tag** field: top part of mapped address
- **valid bit**: is it valid?
i) **direct** mapping

**Cache**

- **index**
- **valid**
- **tag**
- **data**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 | 01

i.e., x “belongs to” address 1001
i) **direct** mapping

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>z</td>
</tr>
</tbody>
</table>

assuming memory & cache are in sync, “fill in” memory
i) **direct** mapping

### Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>z</td>
</tr>
</tbody>
</table>

assuming memory & cache are in sync, “fill in” memory
i) **direct** mapping

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>z</td>
</tr>
</tbody>
</table>

what if a new request arrives for 1011?
i) **direct** mapping

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
</tbody>
</table>

what if new request arrives for 1011?
- cache “miss”: fetch a
i) **direct** mapping

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
</tbody>
</table>

what if new request arrives for 0010?
i) **direct** mapping

### Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
</tbody>
</table>

what if new request arrives for 0010?

- cache “hit”; just return y
i) **direct** mapping

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>w</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
</tbody>
</table>

what if new request arrives for 1000?
i) **direct** mapping

### Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
</tbody>
</table>

what if new request arrives for 1000?

- *evict* old mapping to make room for new
Given initial contents of a *direct-mapped* cache, determine if each request is a *hit* or *miss*. Also, show the final cache.

<table>
<thead>
<tr>
<th>Requests</th>
<th>Initial Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>hit/miss?</td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xAB</td>
<td></td>
</tr>
<tr>
<td>0x60</td>
<td></td>
</tr>
<tr>
<td>0xAB</td>
<td></td>
</tr>
<tr>
<td>0x83</td>
<td></td>
</tr>
<tr>
<td>0x67</td>
<td></td>
</tr>
<tr>
<td>0xAB</td>
<td></td>
</tr>
<tr>
<td>0x12</td>
<td></td>
</tr>
</tbody>
</table>
Problem: our cache (so far) implicitly deals with single bytes of data at a time

```c
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact *= n;
        n -= 1;
    }
}
```

But we frequently deal with >1 byte of data at a time (e.g., words)
Solution: adjust minimum granularity of memory ⇔ cache mapping

Use a “cache block” of $2^b$ bytes

† memory remains byte-addressable!
e.g., block size = 2 bytes  
total # cache lines = 4

Given request for 2-byte word @ 01110, which cache block to use?

<table>
<thead>
<tr>
<th>(line) index</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
</tr>
</tbody>
</table>
Given request for 2-byte word @ 01110, which cache block to use?

e.g., block size = 2 bytes  
total # cache lines = 4
e.g., block size = 2 bytes
total # cache lines = 4

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(line)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
</tr>
<tr>
<td>00010</td>
</tr>
<tr>
<td>00100</td>
</tr>
<tr>
<td>00110</td>
</tr>
<tr>
<td>01000</td>
</tr>
<tr>
<td>01010</td>
</tr>
<tr>
<td>01100</td>
</tr>
<tr>
<td>01110</td>
</tr>
<tr>
<td>10000</td>
</tr>
<tr>
<td>10010</td>
</tr>
<tr>
<td>10100</td>
</tr>
<tr>
<td>10110</td>
</tr>
<tr>
<td>11000</td>
</tr>
<tr>
<td>11010</td>
</tr>
<tr>
<td>11100</td>
</tr>
<tr>
<td>11110</td>
</tr>
</tbody>
</table>

- tag field
- index
- block offset

log_2(# lines) bits wide

log_2(block size) bits wide

e.g., block size = 2 bytes

total # cache lines = 4

block offset

log_2(block size) bits wide
e.g., cache with $2^{10}$ lines of 4-byte blocks
note: we assume that words in memory are *aligned*; i.e., they start at addresses that are *multiples of the word size*.

otherwise, must fetch $> 1$ word-sized block to access a single word!

\[
\begin{array}{ccc}
& w_0 & w_1 \\
w_2 & & \\
w_3 & & \\
\end{array}
\]

2 cache lines
```c
struct foo {
    char c;
    int i;
    char buf[10];
    long l;
};

struct foo f = { 'a', 0xDEADBEEF, "abcdefgHi", 0x123456789DEFAcEDEd };

main() {
    printf("%d %d %d\n", sizeof(int), sizeof(long), sizeof(struct foo));
}
```

$ ./a.out
4 8 32

$ objdump -s -j .data a.out
a.out: file format elf64-x86-64
Contents of section .data:
   61000000 efbeadde 61626364 65666768 a.......abcdefgh
   69000000 00000000 edacef9d 78563412 i...........xV4.
Given: *direct-mapped* cache with *4-byte blocks*. Determine the average *hit rate* of *strlen* (i.e., the *fraction of cache hits* to total requests)
Assumptions:
- ignore code caching (in separate cache)
- buf contents are not initially cached
### C Function: `strlen`

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

**Example Outputs:**
- `strlen("")` returns 0
- `strlen("a")` returns 1
- `strlen("abcde")` returns 5
- `strlen("abcdefghi...jkl...")` returns 13
```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

**strlen(\0)**

**strlen(a \0)**

**strlen(a b c d e \0)**

**strlen(a b c d e f g h i j k l ...)**
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}

strlen(\0) or, if unlucky: a\0
strlen(a b c d e \0)
strlen(a b c d e f g h i j k l ...)

int strlen(char *buf) {
    pushq %rbp
    movq %rsp,%rbp
    mov $0x0,%eax    ; result = 0
    cmpb $0x0,(%rdi)  ; if *buf == 0
    je 0x10000500   ;   return 0
    add $0x1,%rdi    ; buf += 1
    add $0x1,%eax    ; result += 1
    movzbl (%rdi),%edx  ; %edx = *buf
    add $0x1,%rdi    ; buf += 1
    add $0x1,%eax    ; result += 1
    movzbl (%rdi),%edx  ; %edx = *buf
    add $0x1,%rdi    ; buf += 1
    test %dl,%dl      ; if %edx[0]≠0
    jne 0x1000004f2  ;   loop
    popq %rbp
    ret
```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

---

```assembly
strlen:           ; buf in %rdi
    pushq %rbp
    movq %rsp,%rbp
    mov $0x0,%eax    ; result = 0
    cmpb $0x0,(%rdi)  ; if *buf == 0
    je 0x10000500   ;   return 0
    add $0x1,%rdi    ; buf += 1
    add $0x1,%eax    ; result += 1
    movzbl (%rdi),%edx  ; %edx = *buf
    add $0x1,%rdi
    test %dl,%dl      ; if %edx[0] ≠ 0
    jne 0x1000004f2  ;   loop
    popq %rbp
    ret
```

—or, if unlucky:

— simplifying assumption: first byte of buf is aligned
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

The image shows a comparison of C code and its assembly representation. The code snippet defines the `strlen` function, which calculates the length of a string by incrementing a counter until a null character is encountered.

The assembly code above is a direct translation of the C code, with comments explaining each assembly instruction. The assembly code uses registers and memory operations to perform the same task as the C code, demonstrating the low-level nature of assembly language programming.

The image also includes visual representations of the function's behavior with different input strings, showing how the function processes each character until it encounters a null character, indicating the end of the string.
```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

- `strlen(`
- `strlen(a"
- `strlen(a b"
- `strlen(a b c d e "
- `strlen(a b c d e f g h i j k l "...
```
In the long run, hit rate = \( \frac{3}{4} = 75\% \)
Again: *direct-mapped* cache with *4-byte blocks*. Average *hit rate* of *sum*? *(arr not cached)*
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}

sum(01 00 00 00 02 00 00 00 03 00 00 00, 3)
```c
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

`sum(01 00 00 00 02 00 00 00 03 00 00 00, 3)`

**each block is a miss! (hit rate=0%)**
use *multi-word* blocks to help with larger array strides (e.g., for word-sized data)
e.g., cache with $2^8$ lines of $2 \times 4$ byte blocks
Are the following (byte) requests hits? If so, what data is returned by the cache?

1. 0x0E9C
2. 0x8C30

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
<th>Byte 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>173</td>
<td>1</td>
<td>05</td>
<td>E2</td>
<td>6C</td>
<td>05</td>
<td>3B</td>
<td>53</td>
<td>0C</td>
<td>8E</td>
</tr>
<tr>
<td>1</td>
<td>2FB</td>
<td>1</td>
<td>9B</td>
<td>26</td>
<td>58</td>
<td>E0</td>
<td>EB</td>
<td>05</td>
<td>4A</td>
<td>4C</td>
</tr>
<tr>
<td>2</td>
<td>316</td>
<td>0</td>
<td>F8</td>
<td>3E</td>
<td>29</td>
<td>92</td>
<td>B2</td>
<td>52</td>
<td>69</td>
<td>2E</td>
</tr>
<tr>
<td>3</td>
<td>03A</td>
<td>1</td>
<td>95</td>
<td>07</td>
<td>51</td>
<td>3F</td>
<td>7B</td>
<td>00</td>
<td>DA</td>
<td>AC</td>
</tr>
<tr>
<td>4</td>
<td>1B9</td>
<td>0</td>
<td>9A</td>
<td>AB</td>
<td>9E</td>
<td>E3</td>
<td>20</td>
<td>03</td>
<td>C0</td>
<td>06</td>
</tr>
<tr>
<td>5</td>
<td>2C2</td>
<td>1</td>
<td>FB</td>
<td>7C</td>
<td>EC</td>
<td>25</td>
<td>C8</td>
<td>2B</td>
<td>3E</td>
<td>D6</td>
</tr>
<tr>
<td>6</td>
<td>315</td>
<td>1</td>
<td>E0</td>
<td>05</td>
<td>FB</td>
<td>E8</td>
<td>72</td>
<td>79</td>
<td>BE</td>
<td>D4</td>
</tr>
<tr>
<td>7</td>
<td>2C7</td>
<td>1</td>
<td>45</td>
<td>2D</td>
<td>92</td>
<td>74</td>
<td>C8</td>
<td>CB</td>
<td>92</td>
<td>85</td>
</tr>
</tbody>
</table>
What happens when we receive the following sequence of requests?

- 0x9697A, 0x3A478, 0x34839,
  0x3A478, 0x9697B, 0x3483A
problem: when a cache collision occurs, we must evict the old (direct) mapping — no way to use a different cache slot
ii) **associative** mapping

e.g., request for memory address **1001**
ii) **associative** mapping

e.g., request for memory address 1001
ii) **associative** mapping

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1001</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*use the full address as the “tag”*

- effectively a hardware lookup table
ii) **associative** mapping

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1001</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1100</td>
<td>y</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0001</td>
<td>w</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0101</td>
<td>z</td>
</tr>
</tbody>
</table>

- can accommodate requests = # lines without conflict

Memory

```
address  | Memory
----------|--------
0000      | w      
0001      |        |
0010      |        |
0011      |        |
0100      |        |
0101      | z      |
0110      |        |
0111      |        |
1000      |        |
1001      | x      |
1010      |        |
1011      |        |
1100      | y      |
1101      |        |
1110      |        |
1111      |        |
```
comparisons done in parallel (h/w): fast!
ii) **associative** mapping

- resulting ambiguity: what to do with a new request? (e.g., 0111)
associative caches require a *replacement policy* to decide which slot to evict, e.g.,

- **FIFO** (oldest is evicted)
- least frequently used (LFU)
- least recently used (LRU)
e.g., LRU replacement

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- requests: 0101, 1001
            1100, 0001
            1010, 1001
            0111, 0001

- requests:
  - 0101, 1001
  - 1100, 0001
  - 1010, 1001
  - 0111, 0001

<table>
<thead>
<tr>
<th>address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>w</td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>z</td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>a</td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>x</td>
</tr>
<tr>
<td>1010</td>
<td>b</td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>y</td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
e.g., LRU replacement

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
<th>last used</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0101</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1100</td>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0001</td>
<td>W</td>
<td>3</td>
</tr>
</tbody>
</table>

- requests: 0101, 1001
            1100, 0001
            1010, 1001
            0111, 1001

Memory

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
</tr>
</tbody>
</table>
e.g., LRU replacement

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
<th>last used</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1010</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1100</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0001</td>
<td>w</td>
<td>3</td>
</tr>
</tbody>
</table>

- requests: 0101, 1001
             1100, 0001
             1010, 1001
             0111, 1001

Memory

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
</tr>
</tbody>
</table>

w
z
a
x
b
y
e.g., LRU replacement

### Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
<th>last used</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1010</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1100</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0001</td>
<td>w</td>
<td>3</td>
</tr>
</tbody>
</table>

requests: 0101, 1001, 1100, 0001, 1010, 1001, 0111, 1001

### Memory

<table>
<thead>
<tr>
<th>address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>w</td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>z</td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>a</td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>x</td>
</tr>
<tr>
<td>1010</td>
<td>b</td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>y</td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
e.g., LRU replacement

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
<th>last used</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1010</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0111</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0001</td>
<td>w</td>
<td>3</td>
</tr>
</tbody>
</table>

- requests: 0101, 1001
             1100, 0001
             1010, 1001
             0111, 1001

Memory

<table>
<thead>
<tr>
<th>address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>w</td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>z</td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>a</td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>x</td>
</tr>
<tr>
<td>1010</td>
<td>b</td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>y</td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
e.g., LRU replacement

Cache

<table>
<thead>
<tr>
<th>index</th>
<th>valid</th>
<th>tag</th>
<th>data</th>
<th>last used</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1010</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>x</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0111</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0001</td>
<td>w</td>
<td>3</td>
</tr>
</tbody>
</table>

- requests: 0101, 1001
            1100, 0001
            1010, 1001
            0111, 1001
in practice, LRU is too complex (slow/expensive) to implement in hardware
use pseudo-LRU instead — e.g., track just MRU item, evict any other
even with optimization, a fully associative cache with more than a few lines is prohibitively complex / expensive
iii) **set associative mapping**

An address can map to a *subset* (≥ 1) of available cache slots.
1 valid bit per line

$B = 2^b$ bytes per cache block

$S = 2^s$ sets

$E$ lines per set

Cache size: $C = B \times E \times S$ data bytes
Understanding Cache Blocks

A cache block is divided into three parts: Valid, Tag, and Cache block.

Selected set

- Set 0:
  - Valid
  - Tag
  - Cache block

- Set 1:
  - Valid
  - Tag
  - Cache block

- Set S-1:
  - Valid
  - Tag
  - Cache block

The selection of the set is based on the tag and set index.

- **Tag**: m-1 bits
- **Set index**: s bits
- **Block offset**: b bits

The diagram illustrates the selection process and the layout of cache blocks.
(1) The valid bit must be set

(2) The tag bits in one of the cache lines must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte

\[ \text{Selected set (i):} \]

\[ \begin{array}{c|c}
1 & 1001 \\
1 & 0110 \\
\end{array} \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

\[ w_0, w_1, w_2, w_3 \]

\[ t \text{ bits} \quad s \text{ bits} \quad b \text{ bits} \]

\[ \begin{array}{c|c|c}
0110 & i & 100 \\
\end{array} \]

\[ m-1 \quad \text{Tag} \quad \text{Set index} \quad \text{Block offset}^0 \]
nomenclature:

- *n-way set associative* cache = \( n \) lines per set (each line containing 1 block)

- *direct mapped* cache: 1-way set associative

- *fully associative* cache: \( n = \) total \# lines
<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>973</td>
<td>0</td>
<td>05</td>
<td>E2</td>
<td>6C</td>
<td>05</td>
</tr>
<tr>
<td></td>
<td>C3B</td>
<td>1</td>
<td>0C</td>
<td>8E</td>
<td>FB</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>89B</td>
<td>0</td>
<td>58</td>
<td>E0</td>
<td>EB</td>
<td>05</td>
</tr>
<tr>
<td></td>
<td>64A</td>
<td>0</td>
<td>16</td>
<td>0C</td>
<td>F8</td>
<td>3E</td>
</tr>
<tr>
<td>1</td>
<td>929</td>
<td>0</td>
<td>B2</td>
<td>52</td>
<td>B9</td>
<td>2E</td>
</tr>
<tr>
<td></td>
<td>C3A</td>
<td>1</td>
<td>95</td>
<td>07</td>
<td>51</td>
<td>3F</td>
</tr>
<tr>
<td></td>
<td>B7B</td>
<td>0</td>
<td>DA</td>
<td>AC</td>
<td>B9</td>
<td>8E</td>
</tr>
<tr>
<td></td>
<td>99A</td>
<td>1</td>
<td>9E</td>
<td>E3</td>
<td>20</td>
<td>03</td>
</tr>
<tr>
<td>2</td>
<td>5C0</td>
<td>0</td>
<td>C2</td>
<td>B1</td>
<td>FB</td>
<td>7C</td>
</tr>
<tr>
<td></td>
<td>CEC</td>
<td>1</td>
<td>C8</td>
<td>2B</td>
<td>3E</td>
<td>D6</td>
</tr>
<tr>
<td></td>
<td>B15</td>
<td>1</td>
<td>E0</td>
<td>05</td>
<td>FB</td>
<td>E8</td>
</tr>
<tr>
<td></td>
<td>772</td>
<td>1</td>
<td>BE</td>
<td>D4</td>
<td>C7</td>
<td>79</td>
</tr>
<tr>
<td>3</td>
<td>745</td>
<td>1</td>
<td>92</td>
<td>74</td>
<td>C8</td>
<td>CB</td>
</tr>
<tr>
<td></td>
<td>992</td>
<td>1</td>
<td>3C</td>
<td>76</td>
<td>25</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>06C</td>
<td>1</td>
<td>66</td>
<td>41</td>
<td>2E</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>FAB</td>
<td>1</td>
<td>C0</td>
<td>4D</td>
<td>08</td>
<td>88</td>
</tr>
</tbody>
</table>

**Hits/Misses? Data returned if hit?**

1. 0xC3BC
2. 0xCEC9
So far, only considered *read* requests;
What happens on a *write* request?
- don’t really need data *from* memory
- but if cache & memory out of sync, may need to eventually reconcile them
<table>
<thead>
<tr>
<th></th>
<th>write-through</th>
<th>update memory &amp; cache</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>write hit</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write-back</td>
<td></td>
<td>update cache only (requires “dirty bit”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>write miss</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write-around</td>
<td></td>
<td>update memory only</td>
</tr>
<tr>
<td>write-allocate</td>
<td></td>
<td>allocate space in cache for data, then write-hit</td>
</tr>
</tbody>
</table>
logical pairing:

1. write-through + write-around
2. write-back + write-allocate
With write-back policy, eviction (on future read/write) may require data-to-be-evicted to be written back to memory first.
Given: 2-way set assoc cache, 4-byte blocks
DRAM accesses with hit policies (1) vs. (2)?
(1) write-through + write-around

```assembly
movl $0x0000000a,0xf8(%rbp) ; write (around) to memory
movl $0x00000001,0xf4(%rbp) ; write (around) to memory
jmp 0x100000efd

; read from memory → cache / cache
movl 0xf4(%rbp),%eax
movl 0xf8(%rbp),%ecx
imull %ecx,%eax
movl %eax,0xf4(%rbp)
movl 0xf8(%rbp),%eax
subl $0x01,%eax
movl %eax,0xf8(%rbp)
movl 0xf8(%rbp),%eax
```

2 + 4 [first iteration] + 2 × # subsequent iterations
(1) write-back + write-allocate

```
movl $0x0000000a,0xf8(%rbp) ; allocate cache line
movl $0x00000001,0xf4(%rbp) ; allocate cache line
jmp 0x100000efd
movl 0xf4(%rbp),%eax ; read from cache
movl 0xf8(%rbp),%ecx ; read from cache
imull %ecx,%eax ; update cache
movl %eax,0xf4(%rbp) ; read from cache
movl 0xf8(%rbp),%eax ; read from cache
subl $0x01,%eax
movl %eax,0xf8(%rbp) ; update cache
movl 0xf8(%rbp),%eax ; read from cache
cmpl $0x01,%eax
jg 0x100000ee8
```

0 memory accesses! (but flush later)
i.e., write-back & write-allocate allow the cache to absorb multiple writes to memory
why would you ever want write-through / write-around?

- to minimize cache complexity
- if miss penalty is not significant
cache metrics:

- *hit time*: time to detect hit and return requested data

- *miss penalty*: time to detect miss, retrieve data, update cache, and return data
cache metrics:

- *hit time* mostly depends on cache complexity (e.g., size & associativity)

- *miss penalty* mostly depends on latency of lower level in memory hierarchy
catch:

- best hit time favors simple design (e.g., small, low associativity)

- but simple caches = high miss rate; unacceptable if miss penalty is high!
solution: use *multiple levels* of caching
closer to CPU: focus on optimizing hit time, possibly at expense of hit rate
closer to DRAM: focus on optimizing hit rate, possibly at expense of hit time
CPU

Core

L1 Data Cache  L1 Instr Cache

L2 Unified Cache

... L3 Shared, Unified Cache

multi-level cache
e.g., Intel Core i7

Core

- 32KB I, 4-way, ~4 cycles
- 32KB D, 8-way, ~4 cycles
- 256KB, 8-way, ~10 cycles
- 2MB, 16-way, ~40 cycles

multi-level cache
... but what does any of this have to do with systems programming?!?
§Cache-Friendly Code
In general, cache friendly code:

- exhibits *high locality* (temporal & spatial)
- maximizes cache *utilization*
- keeps *working set* size small
- avoids random memory access patterns
case study in software/cache interaction:

matrix multiplication
\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
b_{11} & b_{12} & b_{13} \\
b_{21} & b_{22} & b_{23} \\
b_{31} & b_{32} & b_{33}
\end{pmatrix}
= 
\begin{pmatrix}
c_{11} & c_{12} & c_{13} \\
c_{21} & c_{22} & c_{23} \\
c_{31} & c_{32} & c_{33}
\end{pmatrix}
\]

\[
c_{ij} = \begin{pmatrix} a_{i1} & a_{i2} & a_{i3} \end{pmatrix} \cdot \begin{pmatrix} b_{1j} & b_{2j} & b_{3j} \end{pmatrix}
= a_{i1} b_{1j} + a_{i2} b_{2j} + a_{i3} b_{3j}
\]
canonical implementation:

```
#define MAXN 1000
typedef double array[MAXN][MAXN];

/* multiply (compute the inner product of) two square matrices
 * A and B with dimensions n x n, placing the result in C  */
void matrix_mult(array A, array B, array C, int n) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```
void kji(array A, array B, array C, int n) {
    int i, j, k;
    double r;

    for (k = 0; k < n; k++) {
        for (j = 0; j < n; j++) {
            r = B[k][j];
            for (i = 0; i < n; i++)
                C[i][j] += A[i][k]*r;
        }
    }
}
void kij(array A, array B, array C, int n) {
    int i, j, k;
    double r;

    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            r = A[i][k];
            for (j = 0; j < n; j++)
                C[i][j] += r*B[k][j];
        }
    }
}
cycles per iteration

array size (n)

ijk  kji  kij

135
essential problem: *working set size* grows beyond capacity of cache

smaller strides can help, to an extent (by leveraging spatial locality)
idea for optimization: deal with matrices in smaller chunks at a time that will fit in the cache — “blocking”
/ * "blocked" matrix multiplication, assuming n is evenly 
  * divisible by bsize */
void bijk(array A, array B, array C, int n, int bsize) {
  int i, j, k, kk, jj;
  double sum;

  for (kk = 0; kk < n; kk += bsize) {
    for (jj = 0; jj < n; jj += bsize) {
      for (i = 0; i < n; i++) {
        for (j = jj; j < jj + bsize; j++) {
          sum = C[i][j];
          for (k = kk; k < kk + bsize; k++) {
            sum += A[i][k]*B[k][j];
          }
          C[i][j] = sum;
        }
      }
    }
  }
}
/* "blocked" matrix multiplication, assuming n is evenly divisible by bsize */

void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;

    for (kk = 0; kk < n; kk += bsize) {
        for (jj = 0; jj < n; jj += bsize) {
            for (i = 0; i < n; i++) {
                for (j = jj; j < jj + bsize; j++) {
                    sum = C[i][j];
                    for (k = kk; k < kk + bsize; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
            }
        }
    }
}

Figure 1: Blocked matrix multiply.

Figure 2: Graphical interpretation of blocked matrix multiply.

Use 1 x bsize row sliver bsize times
Use bsize x bsize block n times in succession
Update successive elements of 1 x bsize row sliver
/* Quite a bit uglier without making previous assumption! */
void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;
    int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            C[i][j] = 0.0;

    for (kk = 0; kk < en; kk += bsize) {
        for (jj = 0; jj < en; jj += bsize) {
            for (i = 0; i < n; i++) {
                for (j = jj; j < jj + bsize; j++) {
                    sum = C[i][j];
                    for (k = kk; k < kk + bsize; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
            }
        }
    }
    /* Now finish off rest of j values */
    for (i = 0; i < n; i++) {
        for (j = en; j < n; j++) {
            sum = C[i][j];
            for (k = kk; k < kk + bsize; k++) {
                sum += A[i][k]*B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
/* Now finish remaining k values */
for (jj = 0; jj < en; jj += bsize) {
    for (i = 0; i < n; i++) {
        for (j = jj; j < jj + bsize; j++) {
            sum = C[i][j];
            for (k = en; k < n; k++) {
                sum += A[i][k]*B[k][j];
            }
            C[i][j] = sum;
        }
    }
}

/* Now finish off rest of j values */
for (i = 0; i < n; i++) {
    for (j = en; j < n; j++) {
        sum = C[i][j];
        for (k = en; k < n; k++) {
            sum += A[i][k]*B[k][j];
        }
        C[i][j] = sum;
    }
} /* end of bijk */
Another nice demo of software-cache interaction: the \textit{memory mountain} demo
/ * test - Iterate over first "elems" elements of array "data" *      with stride of "stride". */
void test(int elems, int stride) {
    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride) {
        result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* run - Run test(elems, stride) and return read throughput (MB/s). *      "size" is in bytes, "stride" is in array elements, and *      Mhz is CPU clock frequency in Mhz. */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(double);

    test(elems, stride); /* warm up the cache */
cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```c
#define MINBYTES (1 << 11) /* Working set size ranges from 2 KB */
#define MAXBYTES (1 << 25) /* ... up to 64 MB */
#define MAXSTRIDE 64 /* Strides range from 1 to 64 elems */
#define MAXELEMS MAXBYTES/sizeof(double)

double data[MAXELEMS]; /* The global array we'll be traversing */

int main() {
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */
    double Mhz; /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data */
    Mhz = mhz(0); /* Estimate the clock frequency */

    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++) {
            printf("%.1f\t", run(size, stride, Mhz));
        }
    }
}
```