Please stop charging your phone in public ports

"Just by plugging your phone into a [compromised] power strip or charger, your device is now infected, and that compromises all your data," Drew Paik of security firm Authentic8 explained.

Public charging stations and wi-fi access points are found in places like airports, planes, conference centers and parks, so people can always have access to their phones and data. But connecting your phone to an unknown port has its risks.

The cord you use to charge your phone is also used to send data from your phone to other devices. If a port is compromised, there's no limit to what information a hacker could take, Paik explained.

•  http://money.cnn.com/2017/02/15/technology/public-ports-charging-bad-stop/
Administrivia

- Lab 3 due Thursday (2/23)
- Homework 4 due Tuesday (2/28)
- Optional Section on Thursday: Caches
Peer Instruction Question

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  - Vote at http://PollEv.com/justinh
  - A. 2
  - B. 4
  - C. 8
  - D. 16
  - E. We’re lost...

- If addresses are 16 bits wide, how wide is the Tag field?
  \[ k = \log_2(K) = 7 \text{ bits} \]
  \[ s = \log_2(S) = 1 \text{ bit} \]
  \[ t = m - s - k = 8 \text{ bits} \]
Cache Read

1) Locate set
2) Check if any line in set is valid and has matching tag: hit
3) Locate data starting at offset

$E = \text{blocks/lines per set}$

$S = \# \text{ sets} = 2^s$

Address of byte in memory:

- $t$ bits
- $s$ bits
- $k$ bits
- tag
- set index
- block offset

data begins at this offset

valid bit

$K = \text{bytes per block}$
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B

$S = 2^s$ sets

Address of int:

<table>
<thead>
<tr>
<th>bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

find set

set 0

set 1

set 2

... 

set S-1

8B in block
Example: Direct-Mapped Cache \((E = 1)\)

Direct-mapped: One line per set
Block Size \(K = 8\) B
Example: Direct-Mapped Cache \((E = 1)\)

Direct-mapped: One line per set
Block Size \(K = 8\) B

No match? Then old line gets evicted and replaced

This is why we want alignment!
Example: Set-Associative Cache \((E = 2)\)

2-way: Two lines per set
Block Size \(K = 8\) B

```
<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Address of `short int`:

Block offset

compare both

valid? + match: yes = hit
Example: Set-Associative Cache \((E = 2)\)

2-way: Two lines per set
Block Size \(K = 8\) B

No match?
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Address of \texttt{short int}:

\begin{tabular}{c}
\hline
\texttt{bits} & 0...01 & 100 \\
\hline
\end{tabular}

\begin{itemize}
\item valid? + match: yes = hit
\item short int \((2\) B) is here
\end{itemize}
Types of Cache Misses: 3 C’s!

- **Compulsory** (cold) miss
  - Occurs on first access to a block

- **Conflict** miss
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - *e.g.* referencing blocks 0, 8, 0, 8, ... could miss every time
  - Direct-mapped caches have more conflict misses than $E$-way set-associative (where $E > 1$)

- **Capacity** miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won’t fit, even if cache was fully-associative)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses
Core i7: Associativity

Processor package

Block/line size:
64 bytes for all

L1 i-cache and d-cache:
32 KiB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KiB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MiB, 16-way,
Access: 30-40 cycles

slower, but more likely to hit
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- Program optimizations that consider caches
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory

- What to do on a write-hit? (block/data already in $\$\$\$
  - Write-through: write immediately to next level
  - Write-back: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified ("dirty bit")

- What to do on a write-miss? (block/data not currently in $\$\$\$
  - Write-allocate: ("fetch on write") load into cache, update line in cache
    - Good if more writes or reads to the location follow
  - No-write-allocate: ("write around") just write immediately to memory

- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Write-back, write-allocate example

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.
Write-back, write-allocate example

\[ \text{mov } 0x\text{FACE}, F \]

1. check cache for addr F \(\rightarrow\) miss
2. fetch F from Mem (because write-allocate)

```
<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>G 0xBEEF</td>
<td>F 0xCafe</td>
</tr>
<tr>
<td>G 0xBEEF</td>
<td>G 0xBEEF</td>
</tr>
</tbody>
</table>
```

dirty bit
Write-back, write-allocate example

mov 0xFACE, F

Step 1: Bring F into cache

1. Fetch block
2. Write data into block
Write-back, write-allocate example

```assembly
mov 0xFACE, F
```
Write-back, write-allocate example

mov 0xFACE, F
write-miss

mov 0xFEED, F
write-hit

Cache

Memory

Write hit!
Write 0xFEED to cache only
Write-back, write-allocate example

mov 0xFACE, F  mov 0xFEED, F  mov G, %rax

read miss

dirty bit

mov 0xFEED, F  mov 0xFACE, F

Cache

F  0xFEED  1

Memory

F  0xCafe
G  0xBEEF
Write-back, write-allocate example

mov 0xFACE, F  
mov 0xFEED, F  
mov G, %rax

1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into %rax
Peer Instruction Question

- Which of the following cache statements is FALSE?
  - Vote at http://PollEv.com/justinh

A. We can reduce compulsory misses by decreasing our block size
   - Smaller block size pulls fewer bytes into $ on a miss

B. We can reduce conflict misses by increasing associativity
   - More options to place blocks before evictions occur

C. A write-back cache will save time for code with good temporal locality on writes
   - Frequently-used blocks rarely get evicted, so fewer write-backs

D. A write-through cache will always match data with the memory hierarchy level below it
   - Yes, its main goal is data consistency

E. We’re lost...
Optimizations for the Memory Hierarchy

- Write code that has locality!
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How can you achieve locality?
  - Adjust memory accesses in code (software) to improve miss rate (MR)
    - Requires knowledge of both how caches work as well as your system’s parameters
  - Proper choice of algorithm
  - Loop transformations
Example: Matrix Multiplication

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}
\]
Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:

COLUMN of matrix (blue) is spread among cache blocks shown in red
Naïve Matrix Multiply

# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for (k = 0; k < n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];
Cache Miss Analysis (Naïve)

- **Scenario Parameters:**
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) \(B = 8\) doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- **Each iteration:**
  \[
  \frac{n}{8} + n = \frac{9n}{8} \text{ misses}
  \]

---

Ignoring matrix \(C\)
Cache Miss Analysis (Naïve)

- **Scenario Parameters:**
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- **Each iteration:**
  \[
  \frac{n}{8} + n = \frac{9n}{8}
  \]

- **Afterwards in cache:** (schematic)  
  - [Diagram of cache blocks]
Linear Algebra to the Rescue (1)

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)

- For example, multiply two 4×4 matrices:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22} \\
    a_{31} & a_{32} \\
    a_{41} & a_{42}
\end{bmatrix}
= \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix},
\] with \(B\) defined similarly.

\[
AB = \begin{bmatrix}
    (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
    (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]
Linear Algebra to the Rescue (2)

Matrices of size $n \times n$, split into 4 blocks of size $r$ ($n=4r$)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$$

- Multiplication operates on small “block” matrices
  - Choose size so that they fit in the cache!
  - This technique called “cache blocking”
Blocked Matrix Multiply

- Blocked version of the naïve algorithm:

```c
# move by rxr BLOCKS now
for (i = 0; i < n; i += r) {
    for (j = 0; j < n; j += r) {
        for (k = 0; k < n; k += r) {
            # block matrix multiplication
            for (ib = i; ib < i+r; ib++) {
                for (jb = j; jb < j+r; jb++) {
                    for (kb = k; kb < k+r; kb++) {
                        c[ib*n+jb] += a[ib*n+kb] * b[kb*n+jb];
                    }
                }
            }
        }
    }
}
```

- $r = \text{block matrix size (assume } r \text{ divides } n \text{ evenly)}$
Cache Miss Analysis (Blocked)

- **Scenario Parameters:**
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $r \times r$ fit into cache: $3r^2 < C$

- **Each block iteration:**
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$

Ignoring matrix $C$
Cache Miss Analysis (Blocked)

- Scenario Parameters:
  - Cache block size \( K = 64 \text{ B} = 8 \text{ doubles} \)
  - Cache size \( C \ll n \) (much smaller than \( n \))
  - Three blocks \( (r \times r) \) fit into cache: \( 3r^2 < C \)

- Each block iteration:
  - \( r^2 / 8 \) misses per block
  - \( 2n/r \times r^2 / 8 = nr / 4 \)

- Afterwards in cache (schematic)
Matrix Multiply Visualization

- Here $n = 100$, $C = 32$ KiB, $r = 30$

**Naïve:**

- Approximate cache misses:
  - $\approx 1,020,000$
  - cache misses

**Blocked:**

- Approximate cache misses:
  - $\approx 90,000$
  - cache misses

*shaded areas show blocks stored in the $*$
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code

---

Great general rules of thumb!
The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Slopes of spatial locality

Ridges of temporal locality

Working data set size

decreasing spatial locality

increasing

Stride (x8 bytes)
Mem
L1
L2
L3

Read throughput (MB/s)

memory performance

Size (bytes)
Learning About Your Machine

- **Linux:**
  - `lscpu`
  - `ls /sys/devices/system/cpu/cpu0/cache/index0/`
    - *Ex:* `cat /sys/devices/system/cpu/cpu0/cache/index*/size`
  - `cat /proc/cpuinfo | grep cache | sort | uniq`

- **Windows:**
  - `wmic memcache get <query>` *(all values in KB)*
  - *Ex:* `wmic memcache get MaxCacheSize`

- Modern processor specs: [http://www.7-cpu.com/](http://www.7-cpu.com/)