Lecture 20

by Ethan Glasserman, Matt Sheets and Emi Nietfeld

November 15th, 2012

Code in L20 directory of the CS61-lectures repository

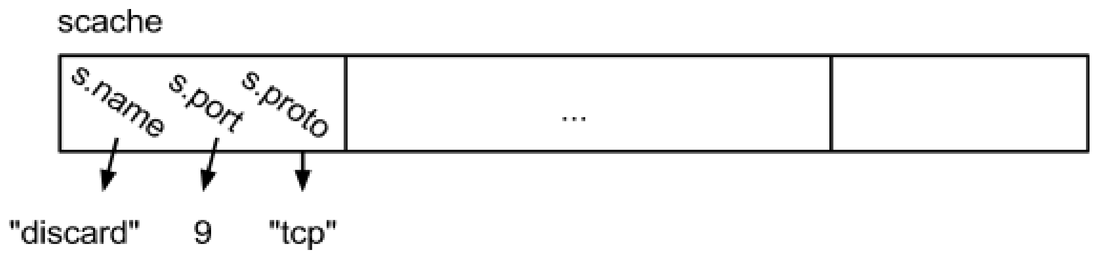
Agenda
-Finish up processor cache optimization
    -Data layout (Hot topic in theoretical CS!)
-Parallelism & Server Programming

Let's recall that in our `serveicetranslate` function from Lecture 19, we had:

```
while(next line) {
      getservbyname(line);
      print"<line>, <port number>";
}
```

Using tools like `strace` and `perf` we discovered the following performance problems/ bottlenecks:
1) `next line` function was completely "dumb" - fixed in Lecture 19
2) `getservbyname` library is inefficient; improved by introducing a CACHE instead of always reading from disk

CACHE design:



Array of structs containing the server *name*, *port*, and *protocol*.

20x improvement! But is it coherent?

Short Answer: This cache is not coherent, but it probably doesn't matter.
Long Answer: In order to be coherent the cache would need to continually check for updates to the file on disk. However, because the `getservbyname` library file on disk will probably only change on rare occasions (i.e. OS updates), this is not necessarily an issue.

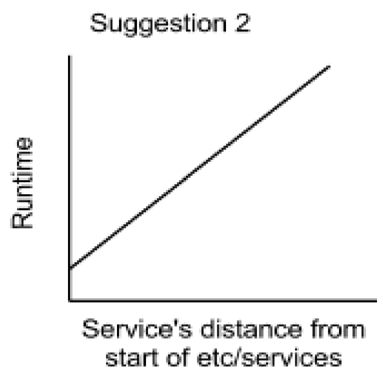
In general, how do we measure the the cost/consequences of incoherence?
-Ask *"How long the incoherence can last?"*
-For `getservbyname` the worst case is the duration of the program, which is proportional to the size of the `etc/services` file. From our experimentation, this turns out to be roughly ½ a second. We'd be more worried if we saw a longer period of incoherence.

Cache Optimization

Pointer vs. String comparison
Because of small number of protocols (only 2), it is better to store the protocol strings once, and then store a pointer to the correct string in the cache. Pointer comparison is more efficient than string comparison. Alleviates time and memory overhead. No need to load extra copy of protocol into processor cache. Roughly 20% advantage in time.

Algorithmic Transformation
Largest benefit occurs from algorithmic transformation.
-We implemented an algorithmic transformation earlier when we added the cache.
-Right now, we know strcmp is taking up a lot of the run-time
    -We COULD use a hash-table for constant-time lookup
    -But hash-tables are tricky to implement, so we're going to do binary search

Hypothesis Testing and Finding Runtime
Goal: Find the runtime for our current strategy
    -Strategy 1: Run 1 input 10 or 100 time, then plot time and input size (linear by nature of our code - doesn't tell us much)
    -Strategy 2: Create targeted input files to run the experiment
        -Pick one name near the middle, one at the end
            -Test hypothesis: if O(n) time, one at the end will take twice as long as the one in the middle
        -It worked! Try with name at the beginning.
        -Try looking for something not in the file, which would take about as long as finding the last thing if it's a linear search.



Binary Search Is Way Better

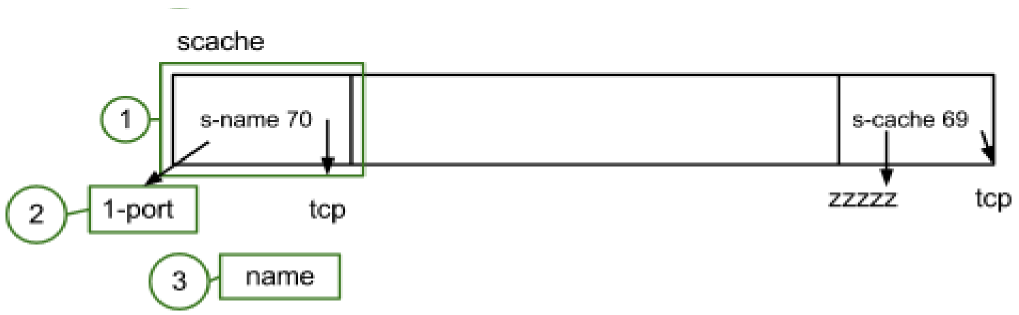
- -No “tcp” optimization on i2: 5.46 seconds
- -With “tcp” optimization on i2: 5.13 seconds
- -Binary Search: 0.82 seconds
- -A hash table would do even better!

Reducing Cache-Line Accesses

A binary search reduced our use of the cache. What else can we do?

-How many cache accesses do we need to compare something in our cache to the input?

- -Let’s look at the code!
- -We have to look up:
    - (1) cache line that contains the entry (from `etc/services`)
    - (2) separately allocated cache line that contains the port (from `etc/services`)
    - (3) separately allocated cache line that contains the name (from input)



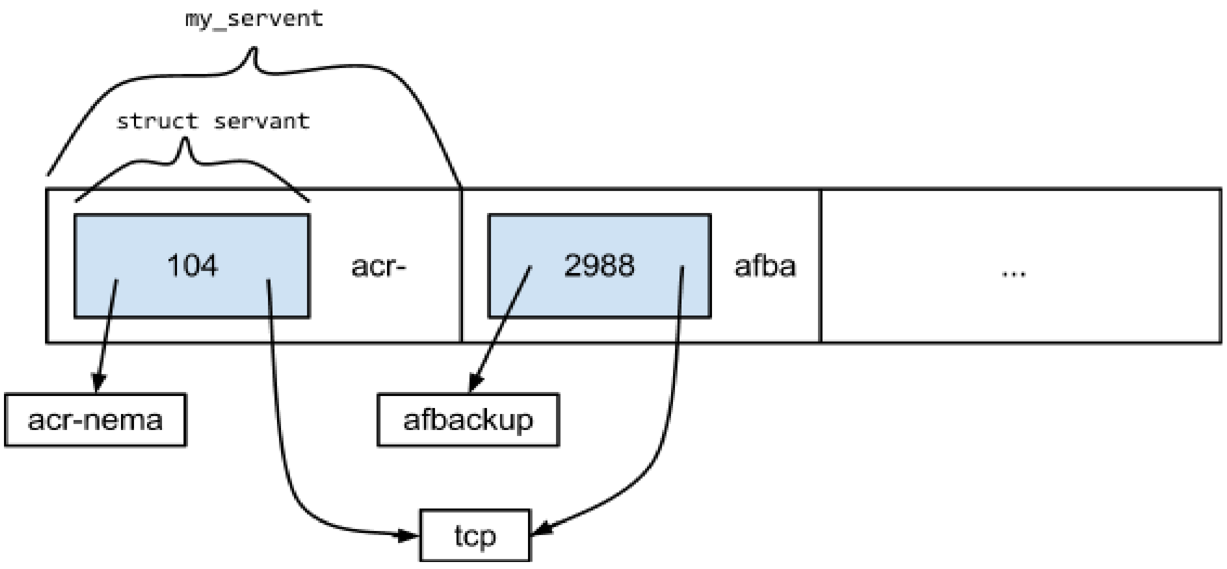
(NOTE: a cache line is an slot of the processor cache; it is 64 bytes of memory stored closer to the CPU than primary memory)

We’re looking up 1 piece of data from the input file, 2 from the `etc/services` file - let’s try to cut that down.

```
    struct servent {
            char *s_name;     /* official service name */
            char **s_aliases;       /* alias list */
            int s_port;       /* port number */
            char *s_proto;    /* protocol to use */
      }
```

`struct servant` is a 16 byte struct. Cache lines are 64 bytes, so we have more space in cache line.

We cannot store the name in scache, because the names can be of unbounded length.
Looking at the names list:
-To tell two names apart, we only need a small number of characters.
-Let's store the first *n = 4* characters of each name in the cache line
-We'll also store the `struct servent` which holds all of the information.
-if the first characters match, then we'll compare it to the name, port and protocol pointed to in `struct servant`.
    -If the characters match, *only then* look at the rest of the string



RESULT:
    -No "tcp" optimization on i2: 5.46 seconds
    -With "tcp" optimization on i2: 5.13 seconds
    -Binary Search: 0.82 seconds
    -Binary Search with a 'cache-friendly' data structure: .78 seconds
With larger inputs, this improvement would be much more significant

Caveat: There's no right way to optimize a cache for all inputs. Instead we looked at the set of inputs and decided what to cache

Lookup Complexity
Binary Search - O(log(N)) w*here N is the number of items in search*
Hash - O(1) i*deally*

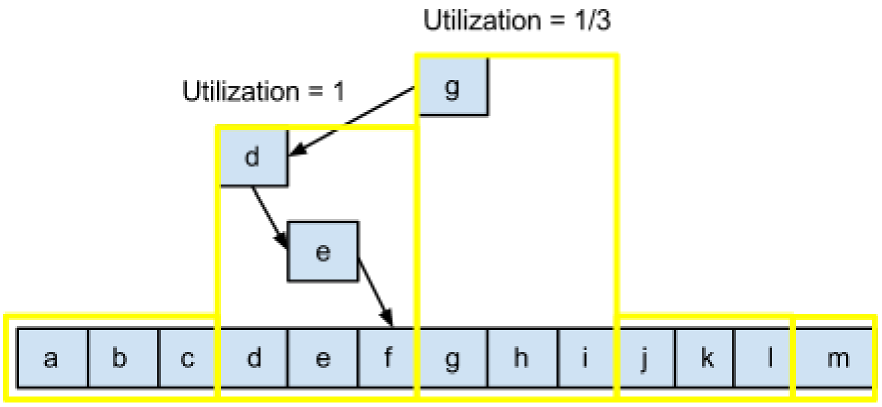
Unlike the hash table, Binary Search will output items in order
*tree let's us output things in order

A Closer Look at Binary Search

Think about the following array:

a	b	c	d	e	f	g	h	i	j	k	l	m

Performing a binary search for 'f', search might proceed as follows:

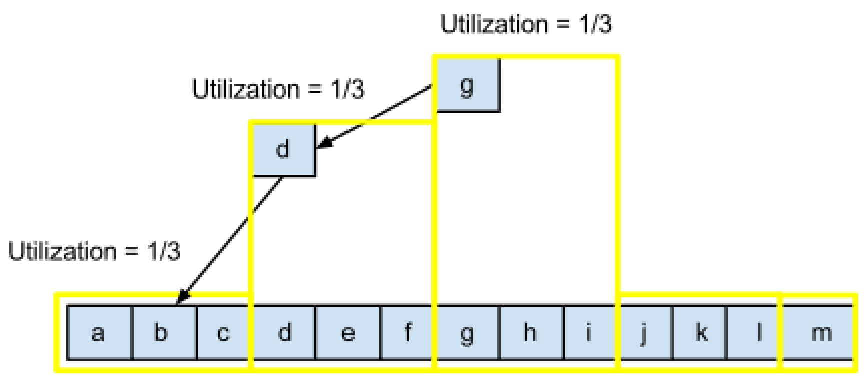


Let's say we broke this up into cache lines (yellow boxes above)
We would access a total 2 cache-lines to find 'f'

Question: What is the overall utilization of these cache-lines?

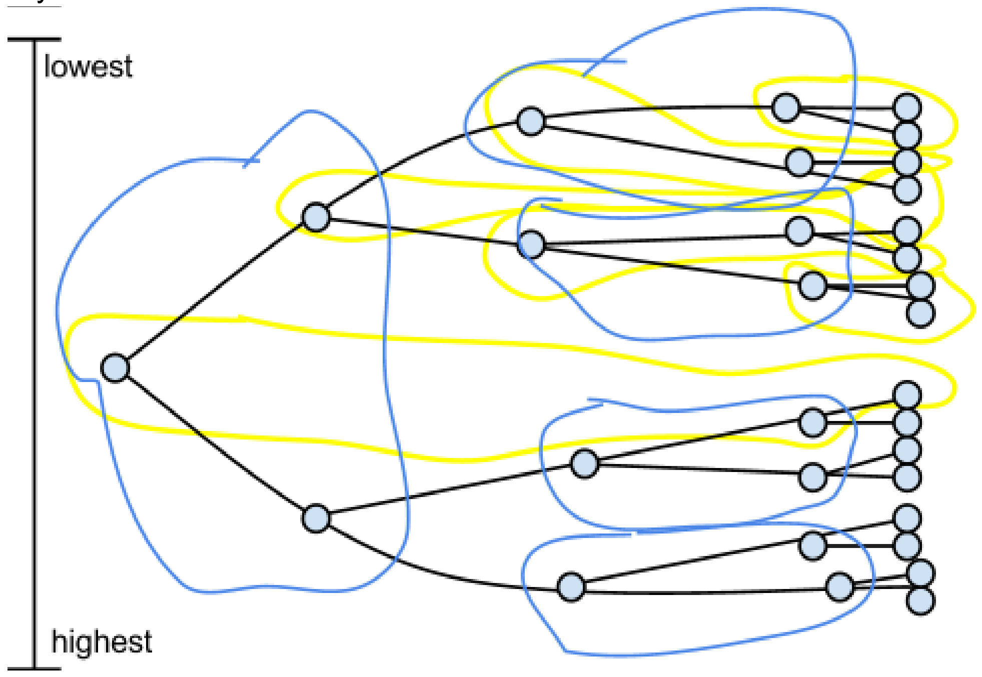
[g,h,i] has utilization ⅓
[d,e,f] has utilization 1

This calculation depends on the order in which we're accessing elements.
Take for example a lookup of "b"



When are cache lines effectively designed?
-Locality is key.
-Although binary search follows predictable paths (from node to node), normally the next node will be in a difference cache line. If you store a node with its children in a single cache-line, utilization will always be 2/3rds, which is better.
-This idea is best seen through a binary search tree.

Binary tree with cache lines:



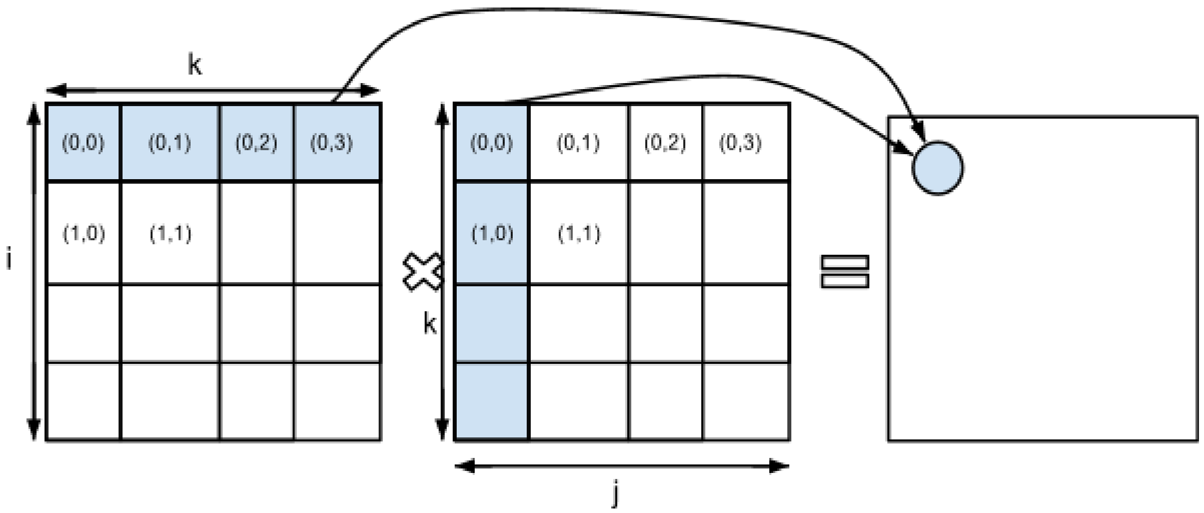
Normal cache lines in yellow. **Van Emde Boas** layout in blue.

Maximum number of cache lines looked up in old design is 5. Maximum number for Van Emde Boas is 3. Van Emde Boas encounters a new cache line every other level. A factor of two (roughly) fewer cache lines in the new layout as opposed to the old layout.

RESULT:
Old Design: 2.97s search time versus
New Design: 2.29s just by reorganizing what elements go in cache-lines together

Matrix Stuff - not like the movies :(

-We're going to multiply some matrices.
-Memory is inherently linear, but matrices are 2-dimensional, so we need to map from 2-dimensional data structure of a matrix into the 1-dimensional data structure of memory



We use a triply-nested loop designed as follows:

```
for(k) {
      for(i) {
            for(j) {
                  MELT
            }
      }
}
```

We want to access memory in as contiguous a manner as possible, though so we should put k on the inside:

```
for(i) {
      for(j) {
            for(k) {
                  MELT
            }
      }
}
```

Yet *kij* takes around 4 seconds and *ijk* takes around 7 seconds?
Woah! Mystery!! Ponder that, amigos!

(Note: other people have worked on these problems and libraries do exist that are quite fast - i.e. 2 seconds as opposed to 4)