CS 61 Lecture Notes - 10/8/13
Michelle Deng & Michael Chen

OPENING REMARKS.

Problem Set 2 notes.

- Example of an excellent submission: far surpasses `stdio` (e.g., 72x faster) for non-sequential, but ~ the same for sequential. This is reasonable--stdio optimized for sequential.
- **Tip: Start small**! Do a basic version, echoing what `stdio` does. Then improve for a few tests.
- Correct & slower > incorrect & faster.
- Notice that our testing infrastructure assesses memory allocation as well as time. Not grading memory usage, but can be helpful for debugging--if extraordinarily high memory usage, may signal a problem…
- Memmapping can speed up accesses. But it only works for files stored on disk, not piped inputs. Also, for large files, 32-bit machines may run out of memory, but 64-bit ones should be okay.

Topics for today:

- Strided access. (Associative caching)
- Explicit prefetching.
- Processor caches.

STRIDED ACCESS.

Definition. Stride *k* access pattern.
Predictable but ***non-sequential*** access pattern that reads / writes every *k*th byte. (Stride 1 is sequential.)

Examples:

- Dictionary headwords as strides accesses into the word list.
- Accessing one struct field in an array of structs.

`stdio` vs. our `r13-stridebyte`. Why is `stdio` slower? `stdio` reads/copies 4096 bytes every time, but we only need one 1 byte. So `stdio` actually does 4096 times more work than necessary. In that case, why is it not even slower? Because the main cost is in disk access, not in per-byte copies, and `stdio` only does one, just like our way.
`r17-multistdiostridebyte` solution: Have multiple buffers. Open multiple copies of a file, and have stdio keep track of different strides into the file. This comes out ~50% faster.

Side note: *direct map* vs. *fully associative* caching.
Recall that in a direct-mapped cache, each element can only be cached in (be mapped to) one specific slot, whereas in an associative cache, any element can go in any slot, and order of slot usage is determined by an eviction policy. `stdio` is direct-mapped: there's only one slot.

EXPLICIT PREFETCHING.

`r19-stdioensbyte`: Read some at beginning and some at end. This is actually a common access pattern where you read some at beginning, scan somewhere, and then read another chunk. Why common? Often files have headers that indicate where to subsequently read info desired.

Definition: Explicit prefetching.
Prefetching is loading stuff into buffer/cache before actual request. ***Explicit*** prefetching has the ***application*** request to do prefetching.

To speed things up, it helps to give OS some warning that you need stuff later.
`r20-stdioadviseendsbyke.c`: Contains `posix_fadvise` function. This tells OS that this program will need to access a certain offset later on. This function can be ignored without program crashing, but reliably gives a small advantage (improves r19 from 3.1 million bytes/s to 3.2 million bytes/s).

Definition: Coherence.
A ***coherent*** cache has contents always equal to corresponding blocks in slow storage.
So suppose you write from a buffer to a file and then read the file. If what you read is what you wrote, then the memory is coherent. Incoherence can occur if you haven't yet flushed the buffer between read accesses; `stdio` cache is not coherent.

PROCESSOR CACHES.

As processors have been getting faster and faster, memory has been actually getting slower. So to make programs run faster, modern machines use many layers of cache:
Disk --> primary memory --> series of level caches (level3 cache, then L2 cache, then L1 cache)--> Registers.
OS manages loading from disk to primary memory. Processor manages moving from primary memory to the level caches. But the concepts/mechanisms of the two are the same.

Cache line (block) size for primary memory: 64 bytes. "This is a factoid you should remember." (Compare this to 4096 bytes when loading from disk.) Why? You want bigger cache sizes for predictable accesses, e.g. from disk. But if unpredictable, loading large amounts at a time will often waste work. so smaller is better. Primary memory access is very jumpy / unpredictable, so there are many small blocks as opposed to one large one.

Demo.

`servicetranslate.c`: Essentially a database lookup of network protocols. This is quite slow --it takes over 1 s per lookup. Let's optimize!

Tactic: Use `strace`. We find that every time we make a lookup, the library opens and closes the entire database. Clearly not optimized for repeated lookups. Library function `setservent()` seems like it'll help us maintain the network connection, but for unclear reasons, it doesn't work. Luckily (cough), we're studying caches, so… let's make a cache!

Entire content of services is not that large, so we can just have a slot for each entry in it.

```
typedef struct scache_slot {
      struct servent entry;
} scache_slot;
static scache_slot* scache = NULL;
static size_t scache_size, scache_capacity;
```

Note: `servent` is a processor-defined struct.

```
struct servent {
      char* s_name;
      char* s_proto;
      int s_port;
}
```

We implement in `servicetranslate-02.c:` First time user calls `getservbyname`, populate the cache with all entries. Subsequently, just iterate over all contents of cache and use `strcmp` to identify entry with same name and protocol as corresponding user inputs.

This is much faster, but we can still optimize further by ~5x by altering our data structure.
Every time we read the `char*`s, we're following pointers to random places, so we're still doing extra work by reading 64 bytes. We want to make comparisons contained within the objects in the struct. This is an example of good vs. bad locality.
Solution: Have global strings. Then every time you have, say, `char*` "tcp" as a field, follow pointer to this global string "tcp", which will already be in the cache!
Note: when writing conditionals, make sure you're careful about order of conditions….