EECS 370

Lab 11:
Cache-aware Programming

Array Access

Caches help us most with **sequential accesses** - i.e. arrays.

Consider the following code:

```
extern int arr[128];
for(int count = 0; count < 5; ++count)
    for(int idx = 0; idx < 128; ++idx)
        do_something(count, arr[idx]);
```

How can we improve this code for caching?

Idea: Temporal Locality

We can change the order of all calls to `do_something()` (assuming this still produces *correct* behavior):

```
extern int arr[128];
for(int idx = 0; idx < 128; ++idx)
    for(int count = 0; count < 5; ++count)
        do_something(count, arr[idx]);
```

Now for each `arr[idx]`, the array value is accessed completely before moving to the next array member.

Stride Accesses

Consider the following code:

```
extern int arr[128];
for(int offset = 0; offset < STRIDE_LEN; ++offset)
    for(int stride = 0; stride + offset < 128;
            stride += STRIDE_LEN)
        do_something(count, arr[stride + offset]);
```

Depending on the value of `STRIDE_LEN` this will work better for different caches.

Stride Accesses

For example, if `STRIDE_LEN` is 1 or 128, the **access pattern** is the same

```
With 1: stride + offset = 0, 1, 2, 3, ...
With 128: stride + offset = 0, 1, 2, 3, ...
```

For other strides, the access pattern is completely different:

```
With 8: 0, 8, 16, ... 120, 1, 9, 17, …
```

Is there any way we can optimize this?

Idea: Spatial Locality

Consider the following code:

```
extern int arr[128];
for(int stride = 0; stride < 128; stride += STRIDE_LEN)
    for(int offset = 0; offset < STRIDE_LEN; ++offset)
        do_something(count, arr[stride + offset]);
```

By inverting the loop order, we get a different access pattern.

The above code, for any `STRIDE_LEN`, accesses `0, 1, 2, 3, ...`

Loop Unrolling

Sometimes we can unroll loops to avoid branch penalties.
Consider the following code:

```
extern int arr[128];
for(int idx = 0; idx < 128; ++idx){
    do_something(0, arr[idx]);
    do_something(1, arr[idx]);
    do_something(2, arr[idx]);
    do_something(3, arr[idx]);
    do_something(4, arr[idx]);
}
```

Loop Unrolling

What are some of the issues here?

If we unroll the loop too much, then parts of the loop get evicted from the **instruction cache** and our runtime is increased.

At a certain point, the evictions might exceed the branch penalties.

Most of the stuff so far, your compiler might optimize for you.

Inline Functions

Branches cost both cycle penalties, and can hurt the cache if a branch target is far away.

If a function contains a small amount of code, the function call might take as many instructions as the function itself!

Inline Functions

Idea: If we can replace a function call with a small number of computations, we eliminate the branch penalty.

This can grow the code size and hurt the instruction cache, but most instruction caches are large enough to take this hit, especially to improve spatial locality.

Inline Functions

Inlining does this for us in C.

Consider this project 3 function:

```
static inline int opcode(int instruction) {
    return instruction>>22;
}
```

Every call to `opcode(instruction)` is replaced with `instruction>>22`

In ARM, the Branch is replaced with: `LSR Xd, Xn, #22.` No cache cost!

Inlining With Loop Unrolling

If we take our unrolled loop and if `do_something` is inlined, we've removed many branches!

```
extern int arr[128];
for(int idx = 0; idx < 128; ++idx){
    do_something(0, arr[idx]);
    do_something(1, arr[idx]);
    do_something(2, arr[idx]);
    do_something(3, arr[idx]);
    do_something(4, arr[idx]);
}
```

Differing Containers

Some data structures are very good for caches. Some are not. Consider a linked-list in C:

```
struct node {
   int data;
   struct node *next;
};

struct node *head = NULL;
```

If we go through the linked list and each next is far from current, our spatial locality is reduced.

But if we rarely iterate the list, and use it more for insertion and deletion, the linked list is better.

Differing Containers in C++

EECS 281 teaches how some STL containers are implemented.

Generally, containers with pointers are less optimal for caches. Index-based containers are more optimal for caches, but can be less optimal for the number of accesses.

Pointer based: trees (`set`, `map`), linked list (`list`, `forward_list`)
Index based: arrays (`vector`), `stack/queue/deque`,
heap (`priority_queue`)
Mixed: hash table (`unordered_set`, `unordered_map`)

Differing Containers in C++

This only matters if a program makes many accesses so that elements are regularly evicted from the cache.

Ex: If only one "path" in a tree is accessed, the path might fit in the cache.

Also, if the cache can hold the entire data structure, then no misses will occur.

Refer to EECS 281 for more.

Struct Alignment Revisited (EECS 281 Tips!)

Consider a byte-addressable architecture with the following data cache:
Cache size: 64 bytes Block size: 16 bytes

```
struct example{
       char a;
       float b;
       short c;
       unsigned int d;
       char e;
       int f;
};
```

Key: can realigning the struct help reduce mem usage AND latency?