ECE 459 W23 Final Exam Solutions

J. Zarnett, P. Lam., H. Chen

April 26, 2023

(1) Many of these are examples, so answers will vary.

a. Deadlock – Rust cannot prevent things like lock ordering problems, so if thread 1 acquires locks in the opposite order to thread 2, it can end up in a deadlock.

b. Leader election – in a database system, if the leader system (does the writes) goes offline, an "election" is held and the different servers vote on which of the read-replicas should become the leader. Because it may take a while to get answers from all servers, the system recovers from the crash faster if the election ends as soon as a majority is reached.

c. Original has a speedup of 2.454. 6 CPUs is 2.927. Dev is 2.597. So more CPUs!

d. It might help – we can probably assume that most points are far away from the current point, according to our definition of close and far. That being the case, we can probably assume all points are far and then subtract the ones that aren't if we need. I think when the nbody problem was an assignment we even recommended that.

e. Profiling/monitoring/logging are all interesting ways to potentially detect abuse of the system. You want to use monitoring and logging to look for outliers, i.e., accounts using way too much CPU time or well-above normal resources, and then investigate why with profiling tools to find out whether the use is legitimate or not.

f. In this program, alter `MAX` as needed.

```rust
const MAX: usize = 100_000_000;

fn main() {
    let m = Mutex::new(0);
    for _i in 0..MAX {
        *m.lock().unwrap() += 1;
    }
}
```

g. If samples are sufficiently random, then the time average approach should work. The reason the system needs to be aperiodic is because we might get the wrong answer if the sampled value is a function of time and the sample time is also a function of time. Breaking the link means

You might also argue that aperiodic samples of a periodic process is functionally equivalent to periodic samples of an aperiodic process.

h. Another possibility for making the performance worse is database rollbacks. If the now-faster part does some insertion to the database it may cause other transactions in progress to be rolled back and restart, thus delaying their completion.

i. Degree courses have a spectrum of interchangeability but are never totally interchangeable. Many courses have some interchangeability: electives are based on choosing X of Y courses (even if Y is large); many other courses have antirequisites which are at least a partial substitute. A few courses have no good replacements for them.

You could say that courses are totally interchangeable if you have a scenario where you need a course just to meet minimum term load and it doesn't count towards the degree, but I'll be super surprised if anyone suggests that answer.

j. An example of strength reduction is taking a multiplication and replacing it with some number of additions. In the simplest example `x *= 2` can be replaced with `x = x + x` which may actually be faster as multiplication units are more complex than addition ones.

(2)

For the last part:

- reducing function call overhead: inlining, devirtualization, tail call elimination
- improve locality/cache perf: inlining, loop unrolling, loop fusion
- specializing for a particular case:

(3)

Here’s my implementation:

```
impl Flush for Buffer {
    fn add_to_buffer(&self, input: &[u8]) {
        let mut buf = self.storage.lock().unwrap();
        for i in input {
            if buf.index == buf.array.len() {
                println!("Buffer␣is␣full...");
                Self::do_flush(&mut buf, &self.destination)
            }
            let idx = buf.index;
            buf.array[idx] = *i;
            buf.index += 1;
        }
    }
    fn flush(&self) {
        let mut buf = self.storage.lock().unwrap();
        Self::do_flush(&mut buf, &self.destination);
    }
}

impl Buffer {
    fn do_flush(buf: &mut MutexGuard<Buf>, destination: &Mutex<File>) {
        println!("Flushing...");
        let mut dest = destination.lock().unwrap();
        dest.write_all(&buf.array[0..buf.index]).unwrap();
        dest.flush().unwrap();
        buf.array.iter_mut().for_each(|element| *element = 0);
        println! {"Flush␣wrote␣{}␣bytes.", buf.index}
        buf.index = 0;
    }
}
```

(4)

(5)

One sample implementation can use Rayon:

```
// Cargo.toml

[dependencies]
rayon = "1.7"

// rainbow_table.rs

/// build the rainbow table
pub fn build(&mut self) {
    // first hash
    self.table.par_iter_mut().for_each(|(key, value)| {
        let hash = RainbowTable::hash(&key);
        *value = Some(hash);
    });

    // reduce and hash
    for _ in 1..self.chain_length {
        self.table.par_iter_mut().for_each(|(_key, value)| {
            let guess = RainbowTable::reduce(&value.clone().unwrap());
            let hash = RainbowTable::hash(&guess);
            *value = Some(hash);
        });
    }
}
```

(6)

The two largest components are `query_before` (18.24%) and `sequence` (38.30%). The speedup calculation for `query_before` would be $100/(100 - 18.24) = 1.22\times$. Similarily, for `sequence` it would be $100/(100 - 38.30) = 1.62\times$.

I chose the `query_before` since it seems easier to optimize.

Explain its logic:

The original implementation is basically iterating document elements up to the given location and add the elements matching selector's criteria to a `Vec`. However, when examining how it gets called, we can observe that the only thing that matters to the parent caller function is the length of the vector.

Optimization:

Therefore, considering that `comemo` will try to memorize the function by hashing its return values, so it takes time to memorize the `query_before` function. The hash of `Vec` is a combination of values obtained by calling `hash` on each field, essentially appending bytes. This can be verified by examining the call graph of the following toy example:

```
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;

fn main() {
    let vec = vec![1, 2, 3, 4, 5];

    // Calculate the hash of the entire vector
    let mut hasher = DefaultHasher::new();
    vec.hash(&mut hasher);
    let vec_hash = hasher.finish();

    // Calculate the hash of the hashes of the individual elements of the vector
    let mut hasher_hashes = DefaultHasher::new();
    hasher_hashes.write_usize(5);
    for element in vec {
        element.hash(&mut hasher_hashes);
    }
    let hashes_hash = hasher_hashes.finish();

    // Compare and print the hash values
    println!("Hash␣of␣the␣entire␣vector:␣{}", vec_hash);
    println!("Hash␣of␣the␣hashes␣of␣the␣vector's␣elements:␣{}", hashes_hash);
    println!(
        "Are␣both␣hash␣values␣equal?␣{}",
        vec_hash == hashes_hash
    );
}
```

The optimization is straightforward. Instead of returning `Vec`, we return `usize`, which also necessitates changes to the caller functions. As a result, a reasonable speedup was achieved (from 8 seconds to 5 seconds on my laptop).

```rust
// in src/model/introspect.rs

/// Query for all matching element up to the given location.
// pub fn query_before(&self, selector: Selector, location: Location) -> Vec<Content> {
pub fn query_before(&self, selector: Selector, location: Location) -> usize {
    let mut matches = vec![];
    for elem in self.all() {
        if selector.matches(elem) {
            matches.push(elem.clone());
        }
        if elem.location() == Some(location) {
            break;
        }
    }
    // matches
    matches.len()
}

// in library/src/meta/counter.rs

pub fn at(&self, vt: &mut Vt, location: Location) -> SourceResult<CounterState> {
  // ...
  // let offset = vt.introspector.query_before(self.selector(), location);
  let offset = vt.introspector.query_before(self.selector(), location).len();
  // ...
}

pub fn both(&self, vt: &mut Vt, location: Location) -> SourceResult<CounterState> {
  // ...
  // let offset = vt.introspector.query_before(self.selector(), location);
  let offset = vt.introspector.query_before(self.selector(), location).len();
  // ...
}

// in library/src/meta/state.rs

pub fn at(self, vt: &mut Vt, location: Location) -> SourceResult<Value> {
  // ...
  // let offset = vt.introspector.query_before(self.selector(), location);
  let offset = vt.introspector.query_before(self.selector(), location).len();
  // ...
}
```