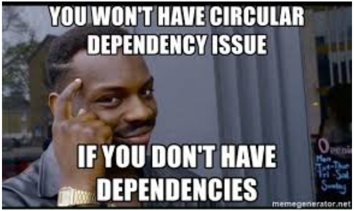
Lecture 13 — Dependencies and Speculation

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

February 17, 2022

Next topic: Dependencies



Dependencies are the main limitation to parallelization.

Example: computation must be evaulated as XY and not YX.

Not synchronization

Assume (for now) no synchronization problems.

Only trying to identify code that is safe to run in parallel.

Dependencies: Analogies

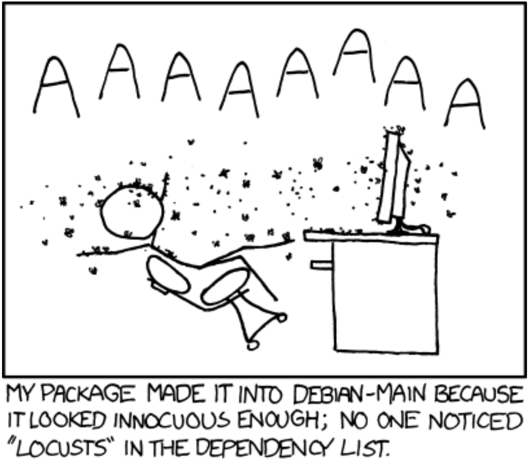
Must extract bicycle from garage before closing garage door.

Must close washing machine door before starting the cycle.

Must be called on before answering questions? (sort of)

Students must submit assignment before course staff can mark the assignment.

Must install package X before running package Y.



xkcd 797

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially vec[0] and vec[1] are 1)

```
let mut vec = vec![1; 32];
    /* */
vec[4] = vec[0] + 1;
vec[5] = vec[0] + 2;
```

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially vec[0] and vec[1] are 1)

```
let mut vec = vec![1; 32];
    /* */
vec[4] = vec[0] + 1;
vec[5] = vec[0] + 2;
```

Yes.

- There are no dependencies between these lines.
- However, this is not how we normally use arrays…

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for i in 1 .. vec.len() {
    vec[i] = vec[i-1] + 1;
}
```

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for i in 1 .. vec.len() {
    vec[i] = vec[i-1] + 1;
}
```

No, a[2] = 3 or a[2] = 2.

- Statements depend on previous loop iterations.
- An example of a loop-carried dependency.

Larger example: Loop-carried Dependencies

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
fn mandelbrot(x0: f64, y0: f64) -> i32 {
    let mut iterations = 0;
    let mut x = x0;
    let mut y = y0;
    let mut x_squared = x * x;
    let mut y_squared = y * y;
    while (x_squared + y_squared < 4f64) && (iterations < 1000) {
        y = 2f64 * x * y + y0;
        x = x_squared - y_squared + x0;
        x_squared = x * x;
        y_squared = y * y;
        iterations += 1;
    }
    return iterations;
}
```

How can we parallelize this?

Larger example: Loop-carried Dependencies

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
fn mandelbrot(x0: f64, y0: f64) -> i32 {
    let mut iterations = 0;
    let mut x = x0;
    let mut y = y0;
    let mut x_squared = x * x;
    let mut y_squared = y * y;
    while (x_squared + y_squared < 4f64) && (iterations < 1000) {
        y = 2f64 * x * y + y0;
        x = x_squared - y_squared + x0;
        x_squared = x * x;
        y_squared = y * y;
        iterations += 1;
    }
    return iterations;
}
```

How can we parallelize this?

- Run `mandelbrot` sequentially for each point, but parallelize different point computations.

Stride

Now consider this example—is it parallelizable? (Again, all elements initially 1.)

```
for i in 4 .. vec.len() {
    vec[i] = vec[i-4] + 1;
}
```

Stride

Now consider this example—is it parallelizable? (Again, all elements initially 1.)

```
for i in 4 .. vec.len() {
    vec[i] = vec[i-4] + 1;
}
```

Yes, to a degree. We can execute 4 statements in parallel at a time:

- `vec[4] = vec[0] + 1, vec[8] = vec[4] + 1`
- `vec[5] = vec[1] + 1, vec[9] = vec[5] + 1`
- `vec[6] = vec[2] + 1, vec[10] = vec[6] + 1`
- `vec[7] = vec[3] + 1, vec[11] = vec[7] + 1`

We can say that the array accesses have stride 4

Dependencies limit the amount of parallelization.

```
let mut acct: Account = Account {
    balance: 0.0f32
};
f(&mut acct);
g(&mut acct);

/* */

fn f (a: &mut Account) {
    a.balance += 50.0f32;
}
fn g (a: &mut Account) {
    a.balance *= 1.01f32;
}
```

What are the possible outcomes after executing `g()` and `f()`

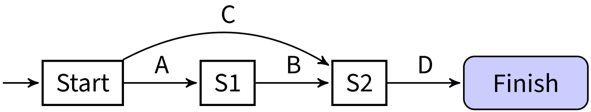
Summary of Memory-carried Dependencies

Well, turns out our memory-carried dependencies are the hazards:

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

Should be familiar with critical paths from other courses (Gantt charts).

Consider the following diagram (edges are tasks):



- B depends on A, C has no dependencies, and D depends on B and C.
- Can execute A-then-B in parallel with C.
- Keep dependencies in mind when calculating speedups for more complex programs.

Speculation: architects use it to predict branch targets.

Image Credit: Diacritica

Roll the dice and see how we do!

If you're a regular, can they guess your order?

And how much time would they save?

Breaking Dependencies

We need not wait for the branch to be evaluated.

We'll use speculation at a coarser-grained level: speculatively parallelize code.

Two ways: speculative execution and value speculation.

Speculative Execution: Example

Consider the following code:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let val = long_calculation(x, y);
    if val > threshold {
        return val + second_long_calculation(x, y);
    }
    return val;
}
```

Will we need to run `second_long_calculation`?

Consider the following code:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let val = long_calculation(x, y);
    if val > threshold {
        return val + second_long_calculation(x, y);
    }
    return val;
}
```

Will we need to run `second_long_calculation`?

- OK, so: could we execute `long_calculation` and `second_long_calculation` in parallel if we didn't have the conditional?

Speculative Execution: Assume No Conditional

Yes, we could parallelize them. Consider this code:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return long_calculation(x, y);
    });
    let t2 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = t1.join().unwrap();
    let v2 = t2.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

We do both the calculations in parallel and return the same result as before.

- What are we assuming about `long_calculation` and `second_long_calculation`?

The current thread is a valid thread for doing work and we don't have to create two threads and join two threads.

We can create one and maybe have less overhead.

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = long_calculation(x, y);
    let v2 = t1.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

Check Assumptions

T_1: time to run `long_calculation`.

T_2: time to run `second_long_calculation`.

p: probability that `second_long_calculation` executes.

In the normal case we have:

$$T_{\text{normal}} = T_1 + pT_2.$$

S: synchronization overhead.
Our speculative code takes:

$$T_{\text{speculative}} = \max(T_1, T_2) + S.$$

Exercise. When is speculative code faster? Slower?
How could you improve it?

Shortcomings of Speculative Execution

Consider the following code:

```
fn do_other_work(x: i32, y: i32) -> i32 {
    let val = long_calculation(x, y);
    return second_long_calculation(val);
}
```

Now we have a true dependency; can't use speculative execution.

But: if the value is predictable, we can execute `second_long_calculation` using the predicted value.

This is value speculation.

Value Speculation Implementation

This code does value speculation:

```
fn do_other_work(x: i32, y: i32, last_value: i32) -> i32 {
    let t = thread::spawn(move || {
        return second_long_calculation(last_value);
    });
    let val = long_calculation(x, y);
    let v2 = t.join().unwrap();
    if val == last_value {
        return v2;
    }
    return second_long_calculation(val);
}
```

Note: this is like memoization (plus parallelization).

Estimating Impact of Value Speculation

T_1: time to run `long_calculation`.

T_2: time to run `second_long_calculation`.

p: probability that `second_long_calculation` executes again.

S: synchronization overhead.

In the normal case, we have:

$$T = T_1 + T_2.$$

This speculative code takes:

$$T = \max(T_1, T_2) + S + pT_2.$$

Exercise. Again, when is speculative code faster? Slower? How could you improve it?

When Can We Speculate?

Required conditions for safety:

- `long_calculation` and `second_long_calculation` must not call each other.
- `second_long_calculation` must not depend on any values set or modified by `long_calculation`.
- The return value of `long_calculation` must be deterministic.

General warning: Consider side effects of function calls.

Side Effects

"Oh yes. It's mentioned here, under side-effects."

Image Credit: Kes, Cartoonstock

Side Effects

As a general warning: Consider the side effects of function calls.

They have a big impact on parallelism. Side effects are problematic, but why?

For one thing they're kind of unpredictable.

Side effects are changes in state that do not depend on the function input.

Side Effects

Calling a function or expression has a side effect if it has some visible effect on the outside world.

Some things necessarily have side effects, like printing to the console.

Others are side effects which may be avoidable if we can help it, like modifying a global variable.


BARRY ALLEN'S TIME TRAVEL LOGIC
HARRISON WELLS:
DON'T DO IT.
REVERSE FLASH:
DON'T DO IT.
@JUSTICELEAGUE.MEMES
IG | BLERD.VISION
JAY GARRICK:
DON'T DO IT...
VIA 9GAG.COM
BARRY'S FUTURE SELF:
DON'T DO IT!
GOD HIMSELF:
DON'T. F*CKING. DO IT.
IMMA DO IT.

STM: Introduction

Instead of programming with locks, we have transactions on memory.

- Analogous to database transactions

An old idea; recently saw some renewed interest.

A series of memory operations either all succeed; or
all fail (and get rolled back), and are later retried.

STM: Benefits

Simple programming model: need not worry about lock granularity or deadlocks.

Just group lines of code that should logically be one operation in an `atomic` block!

It is the responsibility of the implementer to ensure the code operates as an atomic transaction.

Set up a transaction with an atomic block:

```
let x = atomically(|trans| {
    var.write(trans, 42)?; // Pass failure to parent.
    var.read(trans) // Return the value saved in var.
});
```

STM: Implementing a Motivating Example

```
struct Account {
    balance: TVar<f32>,
}

fn transfer_funds(sender: &mut Account, receiver: &mut Account, amount: f32) {
    atomically(|tx| {
        let sender_balance = sender.balance.read(tx)?;
        let receiver_balance = receiver.balance.read(tx)?;
        sender.balance.write(tx, sender_balance - amount)?;
        receiver.balance.write(tx, receiver_balance + amount)?;
        Ok(0)
    });
}
```

[Note: bank transfers aren't actually atomic!]

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

STM: Drawbacks

Rollback is key to STM.
  But, some things cannot be rolled back.
  (write to the screen, send packet over network)

Nested transactions.
  What if an inner transaction succeeds,
yet the transaction aborts?

Limited transaction size:
  Most implementations (especially all-hardware)
have a limited transaction size.

Basic STM Implementation (Software)

In all atomic blocks, record all reads/writes to a log.

At the end of the block, running thread verifies that no other threads have modified any values read.

If validation is successful, changes are **committed**.
Otherwise, the block is **aborted** and re-executed.

Note: Hardware implementations exist too.

Basic STM Implementation Issues

Since you don't protect against dataraces (just rollback), a datarace may trigger a fatal error in your program.

```
fn what_could_go_wrong(x: TVar<i32
    >, y: TVar<i32>) {
    atomically(|t| {
        let old_x = x.read(t)?;
        let old_y = y.read(t)?;
        x.write(t, old_x + 1);
        y.write(t, old_y + 1);
        Ok(0)
    });
}
```

```
fn oh_no(x: TVar<i32>, y: TVar<i32>) {
    atomically(|transaction| {
        if x.read(transaction)? != y.
            read(transaction)? {
            loop { /* Cursed Thread */}
        }
        Ok(0)
    });
}
```

In this silly example, assume initially x = y. You may think the code will not go into an infinite loop, but it can.

STM Summary

Software Transactional Memory gives another approach to parallelism:
no need to deal with locks and their associated problems.

Currently slow, but a lot of research is going into improving it. (futile?)

Operates by either completing an atomic block, or retrying (by rolling back) until it successfully completes.