Lecture 16 — Dependencies and Speculation

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

Department of Electrical and Computer Engineering
University of Waterloo

December 15, 2018

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.

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?
  x = 43 or x = 42 or x = 2

Next, we'll classify dependencies.

Summary of Memory-carried Dependencies

		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)

Loop-carried Dependencies (1)

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

```
a[4] = a[0] + 1
a[5] = a[1] + 2
```

Loop-carried Dependencies (1)

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

```
a[4] = a[0] + 1
a[5] = a[1] + 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 (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1
```

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1
```

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

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

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:
  - a[4] = a[0] + 1, a[8] = a[4] + 1
  - a[5] = a[1] + 1, a[9] = a[5] + 1
  - a[6] = a[2] + 1, a[10] = a[6] + 1
  - a[7] = a[3] + 1, a[11] = a[7] + 1

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:
  - a[4] = a[0] + 1, a[8] = a[4] + 1
  - a[5] = a[1] + 1, a[9] = a[5] + 1
  - a[6] = a[2] + 1, a[10] = a[6] + 1
  - a[7] = a[3] + 1, a[11] = a[7] + 1

Always consider dependencies between iterations.

Larger example: Loop-carried Dependencies

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  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.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

How can we parallelize this?

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

Speculation: architects use it to predict branch targets.

- Need not wait for the branch to be evaluated.

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

Two ways: speculative execution and value speculation.

Speculative Execution: Example

Consider the following code:

```
void doWork(int x, int y) {
    int value = longCalculation(x, y);
    if (value > threshold) {
      return value + secondLongCalculation(x, y);
    }
    else {
      return value;
    }
}
```

Will we need to run `secondLongCalculation`?

Speculative Execution: Example

Consider the following code:

```
void doWork(int x, int y) {
    int value = longCalculation(x, y);
    if (value > threshold) {
      return value + secondLongCalculation(x, y);
    }
    else {
      return value;
    }
}
```

Will we need to run `secondLongCalculation`?

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

Speculative Execution: Assume No Conditional

Yes, we could parallelize them. Consider this code:

```
void doWork(int x, int y) {
  thread_t t1, t2;
  point p(x,y);
  int v1, v2;
  thread_create(&t1, NULL, &longCalculation, &p);
  thread_create(&t2, NULL, &secondLongCalculation, &p);
  thread_join(t1, &v1);
  thread_join(t2, &v2);
  if (v1 > threshold) {
    return v1 + v2;
  } else {
    return v1;
  }
}
```

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

- What are we assuming about `longCalculation` and `secondLongCalculation`?

Estimating Impact of Speculative Execution

T_1: time to run `longCalculation`.

T_2: time to run `secondLongCalculation`.

p: probability that `secondLongCalculation` 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:

```
void doWork(int x, int y) {
    int value = longCalculation(x, y);
    return secondLongCalculation(value);
}
```

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

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

This is value speculation.

Value Speculation Implementation

This Pthread code does value speculation:

```
void doWork(int x, int y) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2, last_value;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation,
                  &last_value);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 == last_value) {
      return v2;
    } else {
      last_value = v1;
      return secondLongCalculation(v1);
    }
}
```

Note: this is like memoization (plus parallelization).

Estimating Impact of Value Speculation

T_1: time to run `longCalculatuion`.

T_2: time to run `secondLongCalculation`.

p: probability that `secondLongCalculation` 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:

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

General warning: Consider side effects of function calls.

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.

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.

Purity

Code that allows multiple concurrent invocations without affecting the outcome is called reentrant or “pure”.

It is a desirable property to have code that is reentrant.

If a function is not reentrant, it may not be possible to make it thread safe.

And furthermore, a reentrant function cannot call a non-reentrant one (and maintain its status as reentrant).

Side Effects

Side effects are sort of undesirable, but not necessarily bad.

Printing to console is unavoidably making use of a side effect, but it's what we want.

When printing we can't have reentrant behaviour because two threads trying to write at the same time to the console would result in jumbled output.

Or alternatively, restarting the print routine might result in some doubled characters on the screen.

Non-Reentrant C Function

The trivial example of a non-reentrant C function:

```
int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}
```

Why is this non-reentrant?

How can we make it reentrant?

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important.

Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted, or we pause and resume.

Either way, if the code is not reentrant we will run into problems.

Thread Safe vs. Reentrant

Let us also draw a distinction between thread safe code and reentrant code.

A thread safe operation is one that can be performed from more than one thread at the same time.

On the other hand, a reentrant operation can be invoked while the operation is already in progress, possibly from within the same thread.

Or it can be re-started without affecting the outcome.

Thread Safe Non-Reentrant Example

```
int length = 0;
char *s = NULL;

// Note: Since strings end with a 0, if we want to
// add a 0, we encode it as "\0", and encode a
// backslash as "\\".


// WARNING! This code is buggy - do not use!


void AddToString(int ch)
{
  EnterCriticalSection(&someCriticalSection);
  // +1 for the character we're about to add
  // +1 for the null terminator
  char *newString = realloc(s, (length+1) * sizeof(char));
  if (newString) {
    if (ch == '\0' || ch == '\\') {
      AddToString('\\'); // escape prefix
    }
    newString[length++] = ch;
    newString[length] = '\0';
    s = newString;
  }
  LeaveCriticalSection(&someCriticalSection);
}
```

Is it thread safe? Sure - there is a critical section protected by the mutex `someCriticalSection`.

But is is re-entrant? Nope.

The internal call to `AddToString` causes a problem because the attempt to use `realloc` will use a pointer to `s`.

That is no longer valid because it got stomped by the earlier call to `realloc`.

Interestingly, functional programming languages (NOT procedural like C) such as Scala and so on, lend themselves very nicely to being parallelized.

Why?

Because a purely functional program has no side effects and they are very easy to parallelize.

Any impure function has to indicate that in its function signature.

Joel on Functional

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable.

- Joel Spolsky

OOP: `getBrain().damage()`

Object oriented programming kind of gives us some bad habits in this regard.

We tend to make a lot of `void` methods.

In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs.

If a function returns nothing, what does it do?

For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

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.

STM: Implementing a Motivating Example

```
transfer_funds(Account* sender, Account* receiver,
               double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

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

With locks we have two main options:

- Lock everything to do with modifying accounts
  (slow; may forget to use lock).
- Have a lock for every account
  (deadlocks; may forget to use lock).

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.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
     while (true) { }
}
```

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

STM Implementations

Note: Typically STM performance is no worse than twice as slow as fine-grained locks.

- Toward.Boost.STM (C++)
- SXM (Microsoft, C#)
- Built-in to the language (Clojure, Haskell)
- AtomJava (Java)
- Durus (Python)

STM Summary

Software Transactional Memory provides a more natural approach to parallel programming:

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.