Lecture 08—Loop-carried Dependencies; Speculation

ECE 459: Programming for Performance

January 30, 2014

Last Time

Having compilers work for you: three-address code, `restrict`, `volatile`.

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)

We also saw how to break WAR and WAW dependencies.

Part I

Loop-carried Dependencies

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:
  - $a[4] = a[0] + 1$, $a[8] = a[4] + 1$
  - $a[5] = a[1] + 1$, $a[9] = a[5] + 1$
  - $a[6] = a[2] + 1$, $a[10] = a[6] + 1$
  - $a[7] = a[3] + 1$, $a[11] = a[7] + 1$

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

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.

Live Coding Demo: Parallelizing Mandelbrot

Refactor the code; create array for output.

Add a struct to pass offset, stride to thread.

Create & join threads.

Part II

Breaking Dependencies with Speculation

Breaking Dependencies

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 `longCalculatuion`.
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.
S: synchronization overhead.

In the normal case, we again have:

$$T = T_1 + pT_2.$$

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.