Lecture 12—Loop-carried Dependencies; Speculation

ECE 459: Programming for Performance

January 30, 2015

Last Time

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)

We also saw how to break WAR and WAW dependencies.

Plus, loop dependencies, and Mandelbrot example.

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 I

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.

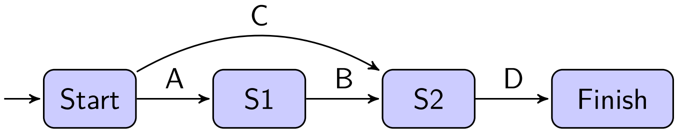
Part II

Parallelization Patterns

Critical Paths

Should be familiar with critcal 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.

Data and Task Parallelism

Data parallelism is performing *the same* operations on different input.
Example: doubling all elements of an array.

Task parallelism is performing *different* operations on different input.
Example: playing a video file: one thread decompresses frames, another renders.

Data Parallelism: Single Instruction, Multiple Data

We'll discuss SIMD in more detail later. An overview:

- You can load a bunch of data and perform arithmetic.
- Intructions process multiple data items simultaneously. (Exact number is hardware-dependent).

For x86-class CPUs, MMX and SSE extensions provide SIMD instructions.

SIMD Example

Consider the following code:

```
void vadd(double * restrict a, double * restrict b,
          int count) {
  for (int i = 0; i < count; i++)
    a[i] += b[i];
}
```

In this scenario, we have a regular operation over block data.

We could use threads, but we'll use SIMD.

SIMD Example—Assembly without SIMD

If we compile this without SIMD instructions on a 32-bit x86, (flags `-m32 -march=i386 -S`) we might get this:

```
loop:
  fldl   (%edx)
  faddl  (%ecx)
  fstpl  (%edx)
  addl   8, %edx
  addl   8, %ecx
  addl   1, %esi
  cmp    %eax, %esi
  jle    loop
```

Just loads, adds, writes and increments.

SIMD Example—Assembly with SIMD

Instead, compiling to SIMD instructions
(`-m32 -mfpmath=sse -march=prescott`) gives:

```
loop:
  movupd  (%edx),%xmm0
  movupd  (%ecx),%xmm1
  addpd   %xmm1,%xmm0
  movpd   %xmm0,(%edx)
  addl    16,%edx
  addl    16,%ecx
  addl    2,%esi
  cmp     %eax,%esi
  jle     loop
```

- Now processing two elements at a time on the same core.
- Also, no need for stack-based x87 code.

SIMD Overview

- Operations *packed*: operate on multiple data elements at the same time.
- On modern 64-bit CPUs, SSE has 16 128-bit registers.
- Very good if your data can be *vectorized* and performs math.
- Usual application: image/video processing.
- We'll see more SIMD as we get into GPU programming: GPUs excel at these types of applications.