CS 4400
Computer Systems

LECTURE 11

Machine-dependent optimizations

Branch prediction

Profiling and improving performance

Recall: Running Example

```
/* most recent version of "combine" */
void combine4(vec_ptr v, data_t* dest) {
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;

  for(i = 0; i < length; i++)
    acc = acc OPER data[i];
  *dest = acc;
}
```

For our machine:

Operation	*Latency*
`int`, +	1
`int`, *	3
`float`, +	3
`float`, *	4
`double`, *	5

- Can we further reduce the CPEs?
- How low can we go?

CPEs	int		float-pt		
	+	*	+	F *	D *
combine1 (-O1)	12.00	12.00	12.00	12.01	13.00
combine4 (-O1)	2.00	3.00	3.00	4.00	5.00

Example: Loop Unrolling

```
void combine5(vec_ptr v, data_t* dest) {
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = 0;
  int limit = length - 2;    /* specific to 3 */

  /* combine 3 elements at a time */
  for(i = 0; i < limit; i+=3)
    acc = acc + data[i] + data[i+1] + data[i+2];

  /* finish any remaining elements */
  for(; i < length; i++)
    acc = acc + data[i];

  *dest = acc;
}
```

- **Reduction in loop overhead is critical in achieving CPE that matches integer addition latency.**
- Integer multiplication improved due to automatic reassociation (more later)
- Why no improvement for floating-point?

CPEs	int		float-pt		
	+	*	+	F *	D *
combine4	2.00	3.00	3.00	4.00	5.00
combine5 x2	2.00	1.50	3.00	4.00	5.00
combine5 x3	1.00	1.00	3.00	4.00	5.00

Superscalar Processors

- ***Superscalar processors*** perform multiple operations simultaneously.
- A ***functional unit*** is a subsystem of the CPU with a specific purpose.
  - integer add, integer mult, float add, float mult, load, store
- After unrolling, our example code is ***limited by the latency of the functional units*** (for all types and ops).

Parallelism

- However, some of the functional units are ***pipelined***.
  - i.e. they can start a new operation before the previous is finished.
- We're currently not taking advantage of this capability and instead causing the processor to stall.
- Ideas why?

Parallelism

- Some of the functional units are ***pipelined***.
  - They can start a new operation before the previous is finished.
- Code like our example cannot take advantage of this capability and causes the processor to stall.
- Ideas why – **Data dependency!**

Data Dependency Diagram

```
/* most recent version of "combine" */
void combine4(vec_ptr v, data_t* dest)
{
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = 0;

  for(i = 0; i < length; i++)
    acc = acc + data[i];
  *dest = acc;
}
```

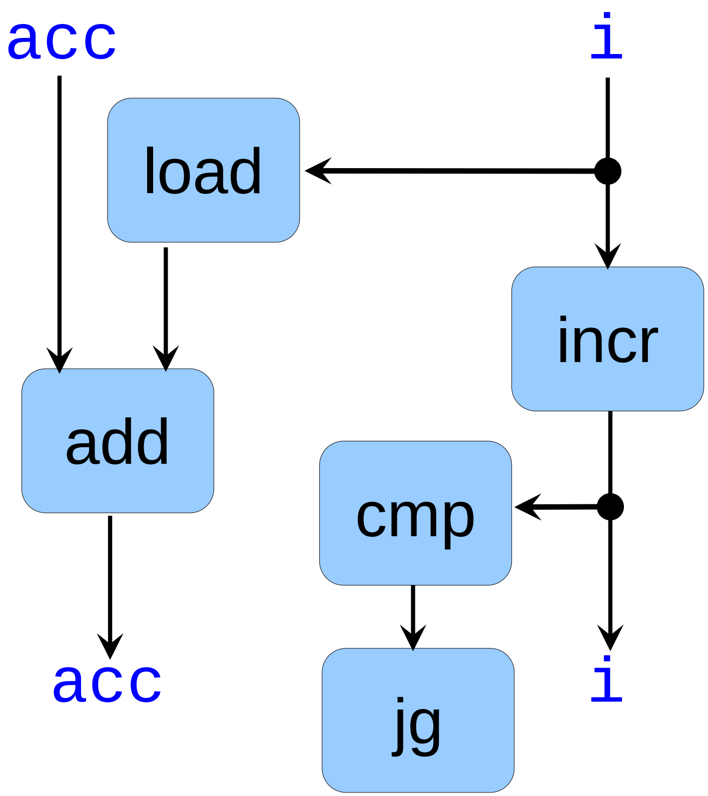
```
L:
  acc = acc + M[data + 4i]
  i = i + 1
  compare i, length
    jump to L if i < length
```

Data Dependency Diagram

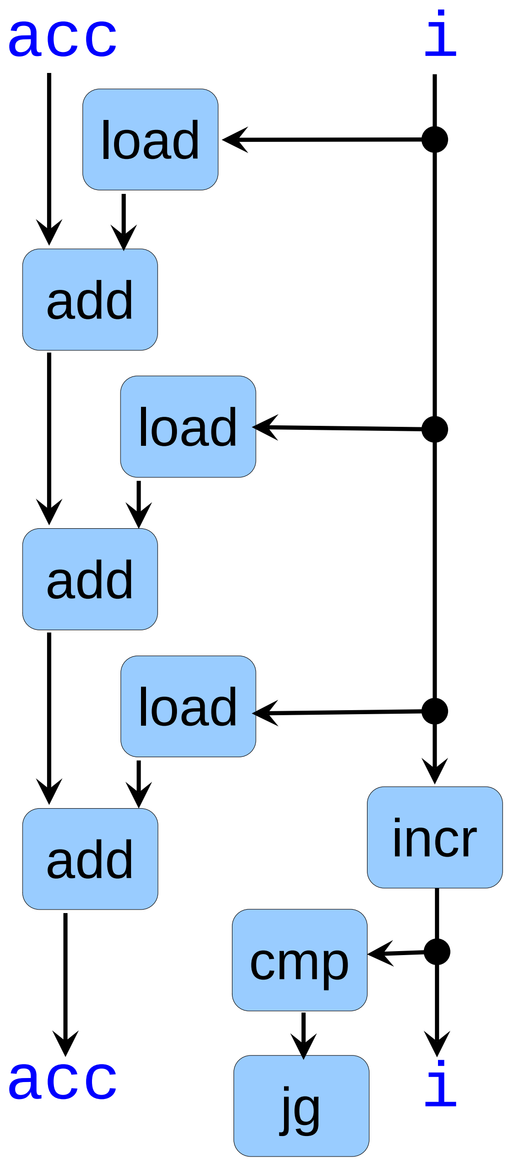
```
/* most recent version of "combine" */
void combine4(vec_ptr v, data_t* dest)
{
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = 0;

  for(i = 0; i < length; i++)
    acc = acc + data[i];
  *dest = acc;
}
```

```
L:
  acc = acc + M[data + 4i]
  i = i + 1
  compare i, length
    jump to L if i < length
```



Effects of Unrolling x3




```
For our superscalar,
out-of-order machine:
```

	latency	*issue*
`int, +`	1	0.33
`int, *`	3	1
`float, +`	3	1
`float, *`	4	1
`double, *`	5	1

Add cannot be issued until the previous add is complete!

`int`, `+`:

Add has 1-cycle latency

3-cycle critical path / 3 elements = 1.0 CPE

`float`, `+`:

Add has 3-cycle latency

9-cycle critical path / 3 elements = 3.0 CPE

Loop Splitting / Parallel Accumulators

```
/* unroll by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t*
dest) {
  int length = vec_length(v);
  int limit = length-1;
  data_t* data = get_vec_start(v);
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;
  int i;

  /* combine 2 elements at a time */
  for(i = 0; i < limit; i+=2) {
    acc0 = acc0 OPER data[i];
    acc1 = acc1 OPER data[i+1];
  }

  /* finish any remaining elements */
  for(; i < length; i++)
    acc0 = acc0 OPER data[i];

  *dest = acc0 OPER acc1;
}
```

- Split the set of combining operations into multiple parts and combine the results at the end.
- When will this (not) preserve the semantics of the original code?

Example: Loop Splitting

CPEs	int		float-pt		
	+	*	+	F *	D *
combine4	2.00	3.00	3.00	4.00	5.00
combine5: unroll x2	2.00	1.50	3.00	4.00	5.00
combine6: unroll x2, split x2	1.50	1.50	1.50	2.00	2.50

- For integers, `combine6` will give the same results as for all previous versions (even when overflow occurs).
- For floats, `combine6` may give different results due to rounding and underflow.
  - Does the performance gain outweigh the risk?

Data Dependency Diagram

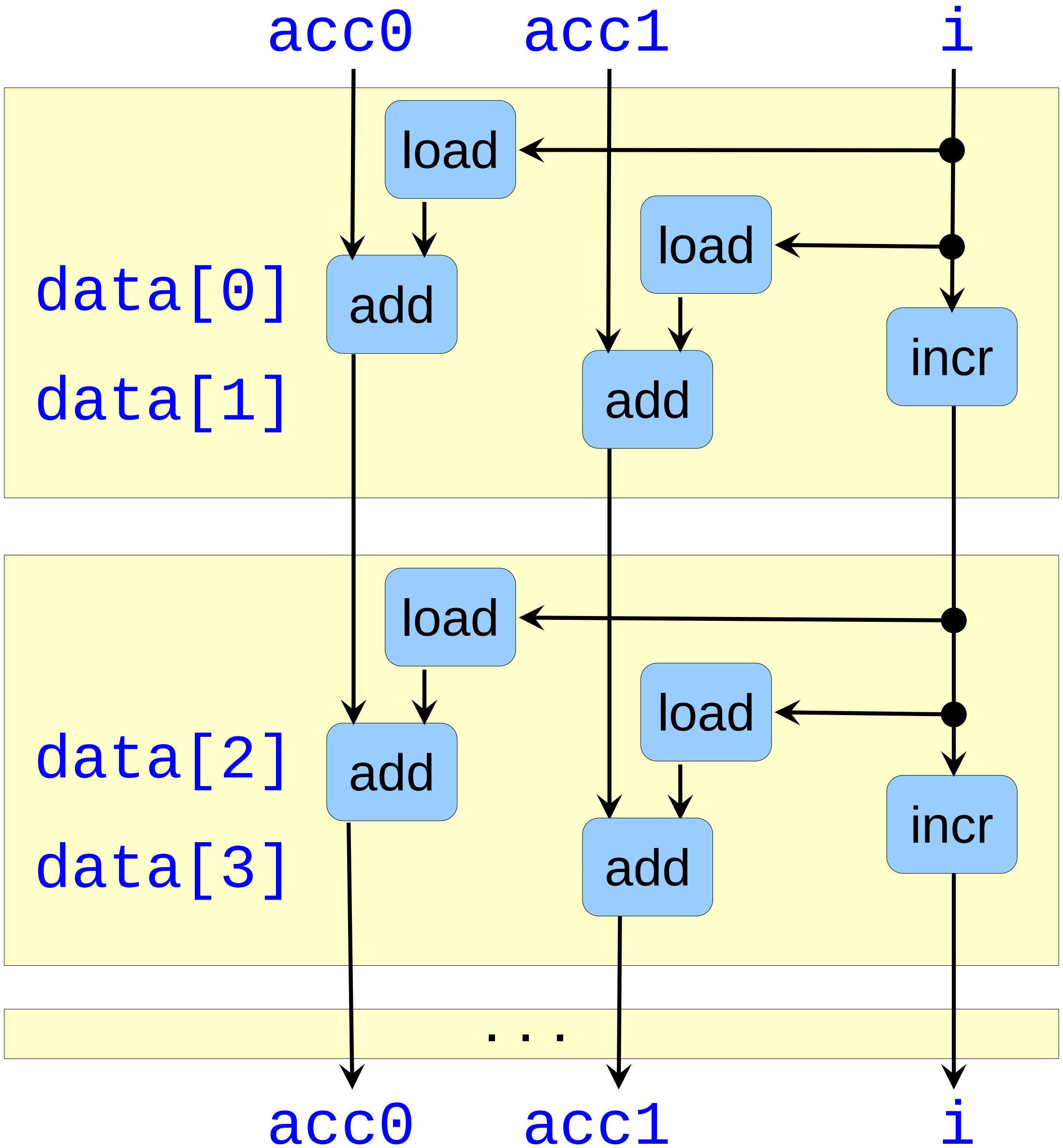
```
/* unroll by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t*
dest) {
  int length = vec_length(v);
  int limit = length-1;
  data_t* data = get_vec_start(v);
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;
  int i;

  /* combine 2 elements at a time */
  for(i = 0; i < limit; i+=2) {
    acc0 = acc0 OPER data[i];
    acc1 = acc1 OPER data[i+1];
  }

  /* finish any remaining elements */
  for(; i < length; i++)
    acc0 = acc0 OPER data[i];

  *dest = acc0 OPER acc1;
}
```

Effects of Unrolling x2, Splitting x2



`float`, `+`:

3-cycle critical path / 2 elements

1.5 CPE

Example: Loop Splitting

CPEs	int		float-pt		
	+	*	+	F *	D *
combine4	2.00	3.00	3.00	4.00	5.00
combine5: unroll x2	2.00	1.50	3.00	4.00	5.00
combine6: unroll x2, split x2	1.50	1.50	1.50	2.00	2.50

- As seen in the text, all CPE approach 1.0 for *k*-way loop unrolling and *k*-way loop parallelism.
- Risks/downsides to increasing parallelism?

Reassociation Transformation

```
/* change associativity of combining ops */
void combine7(vec_ptr v, data_t* dest) {
  int length = vec_length(v);
  int limit = length-1;
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;
  int i;

  /* combine 2 elements at a time */
  for(i = 0; i < limit; i+=2) {
    acc = acc OPER (data[i] OPER data[i+1]);
  }

  /* finish any remaining elements */
  for(; i < length; i++)
    acc = acc OPER data[i];

  *dest = acc;
}
```

Regular unrolling x2, `combine5`*:*

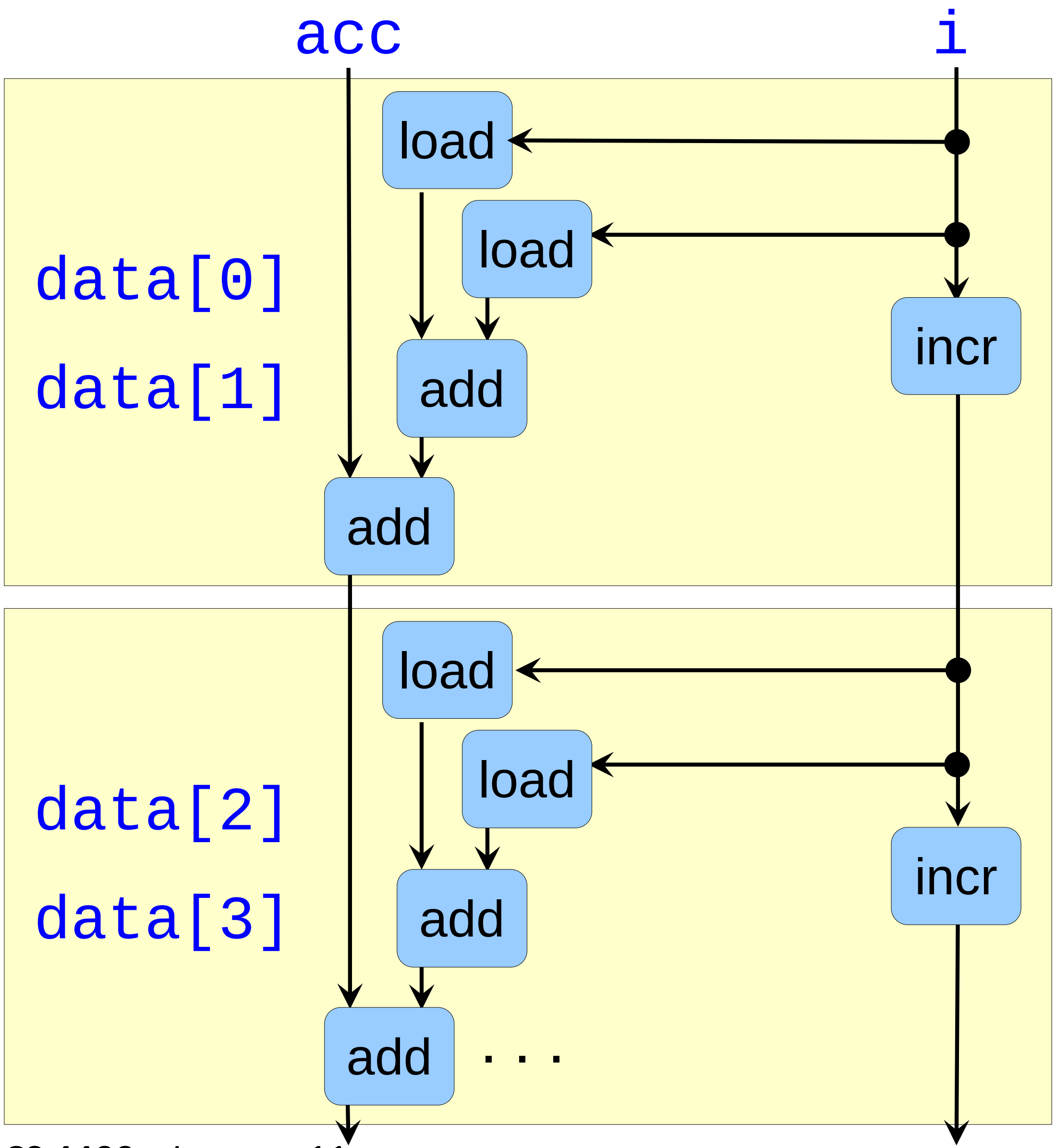
```
  acc = (acc OPER data[i]) OPER data[i+1];
```

Example: Reassociation

CPEs	int		float-pt		
	+	*	+	F *	D *
combine4	2.00	3.00	3.00	4.00	5.00
combine5: unroll x2	2.00	1.50	3.00	4.00	5.00
combine6: unroll x2, split x2	1.50	1.50	1.50	2.00	2.50
combine7: unroll x2, reassociate	2.00	1.51	1.50	2.00	2.97

- Again, as seen in the text, all CPE approach 1.0 for k-way loop unrolling and reassociation.
- The results for D * are likely due to a measurement error (expected to be 2.50).
- Why isn't integer addition the expected 1.0 when unrolling x2?

Effects of Unrolling x2, Reassociate



`float`, `+`:

3-cycle critical path / 2 elements

1.5 CPE

Branch Prediction

- Modern processors work ahead of the currently executing instructions
- ***Branch prediction***—Upon encountering a branch, the processor can guess which way to go
  - ***speculative execution***—the processor begins to fetch/decode instructions at the predicted branch target

Branch Prediction Outcomes

- Prediction is correct
  - “commit” to the results of the speculative execution
- Prediction incorrect
  - discard all of the speculatively-executed results
  - incurring a significant ***branch penalty***

Branch Prediction Outcomes

- Ideas for predicting branches?
- Our running example was not slowed by branch penalties. Prediction was correct almost always. Why?

Branch Prediction Heuristics

- "Take all branches" shown to have a 60% success rate
- Take any branch to a lower address/Don't take any branch to a higher address
  - backward branches are used to reenter loops
  - forward branches are used for conditional computation
  - experiments show 65% success rate
- Advanced heuristics allow modern CPUs get it right >90% of the time

Performance Improvement

1. Choose appropriate algorithms and data structures.

Optimizations cannot save a program with poor asymptotic performance.

Performance Improvement

2. Avoid optimizations blockers and let the compiler generate efficient code.

Eliminate excessive function calls and unnecessary memory references. Move loop-invariant computations.

Performance Improvement

3. Try low-level optimizations when performance really matters.

Pointer vs array code, make the most of instruction pipelining.

Program Profiling

- When working with large programs, even knowing where to focus your optimization efforts can be difficult.
- Code profilers collect performance data as programs run.
  - Instrumentation code is incorporated with the original program code to detect the running time required by different parts.

Program Profiling

- Gnu's code profiler is `gprof`, which reports
  - CPU time spent on each function (relative importance of each)
  - number of calls to each function (dynamic behavior of program)
  - See text, `man`, web, etc. for how to use `gprof` and read output.

Amdahl's Law

- Amdahl's Law provides insight into the effectiveness of improving the efficiency of just one part of a system.

Amdahl's Law

- Let a be the fraction of time required by a critical component of the program
- Let k be the factor of improvement for this component

$$T_{\text{new}} = (1\text{-}a)\ T_{\text{old}} + (a\ T_{\text{old}}) \ /\ k$$

$$\text{Speedup} = 1 \ /\ (\ (1\text{-}a) + a/k\ )$$

$$\text{Speedup} = T_{\text{old}}/T_{\text{new}}$$

Example: Amdahl's Law

- Suppose that we have optimized a part of the program that takes 60% of the program's original running time (i.e. $a = 0.6$)
- We have improved the performance of this part by a factor of 3 (i.e. $k = 3$)
- What is our total speedup?

$$\text{Speedup} = 1 / ((1-a) + a/k)$$

Example: Amdahl's Law

- $a = 0.6$
- $k = 3$
- Total Speedup = 1 / (0.4 + 0.6 / 3) = 1.67

- Even though the improvement of the part is significant, the net improvement on the program is much less.

Special Case of Amdahl's Law

- What if k is ∞?
  - The program part now takes only a negligible amount of time.

$$\text{Speedup} = 1 / ((1-a) + a/k)$$

Special Case of Amdahl's Law

- What if k is ∞?
  - The program part now takes only a negligible amount of time.

$$\text{Speedup} = 1 \,/\, (\,(1\text{-}a) + a/k\,)$$

$$\text{Speedup}_{\infty} = 1 \,/\, (1\text{-a})$$

- *Example*: Let a = 0.6. Net speedup of overall program is still only 1 / 0.4 = 2.5

Impact of Amdahl's Law

- To have a significant impact on the overall program, it is critical to improve the performance of a very large fraction of the program.

Question

- Suppose you are charged with improving the overall performance of a system by a factor of 2. However, you determine that only 60% of the system can be improved. By what factor k must you improve this part to meet the overall goal?

$$T_{new} = (1\text{-}a)\ T_{old} + (a\ T_{old})\ /\ k$$

$$\text{Speedup} = 1\ /\ (\ (1\text{-}a) + a/k\ )$$

$$\text{Speedup} = T_{old} / T_{new}$$

Summary: Optimization

- Much can be done by the programmer to assist an optimizing compiler in generating efficient code.
- Some optimizations require a deeper look the assembly code generated and how the computation is being performed.

Summary: Optimization

- The programmer has little or no control over the branch structure generated by the compiler or the processor's prediction strategy.
- For large programs, focus on the parts that consume the most execution time (using a code profiler).