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 *superscalar*, *out-of-order* machine:

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

Can we further reduce the CPEs?

CPEs	int		float-pt		
	+	*	+	F *	D *
`combine4` -O1	2.00	3.00	3.00	4.00	5.00

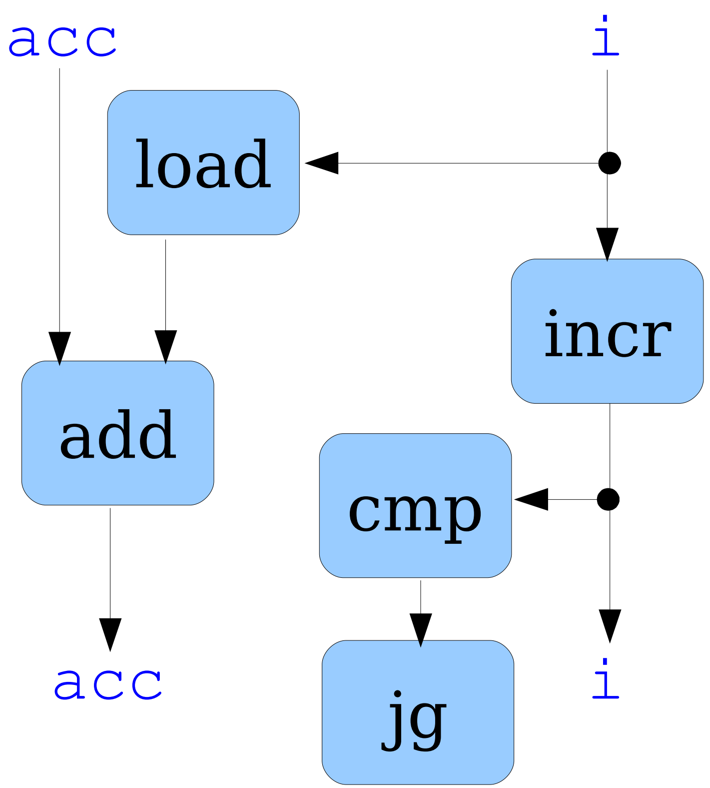
Integer Addition

```
/* 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
```

2.0 CPEs



How many instructions per iteration?
How many operate on the data?

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;
}
```

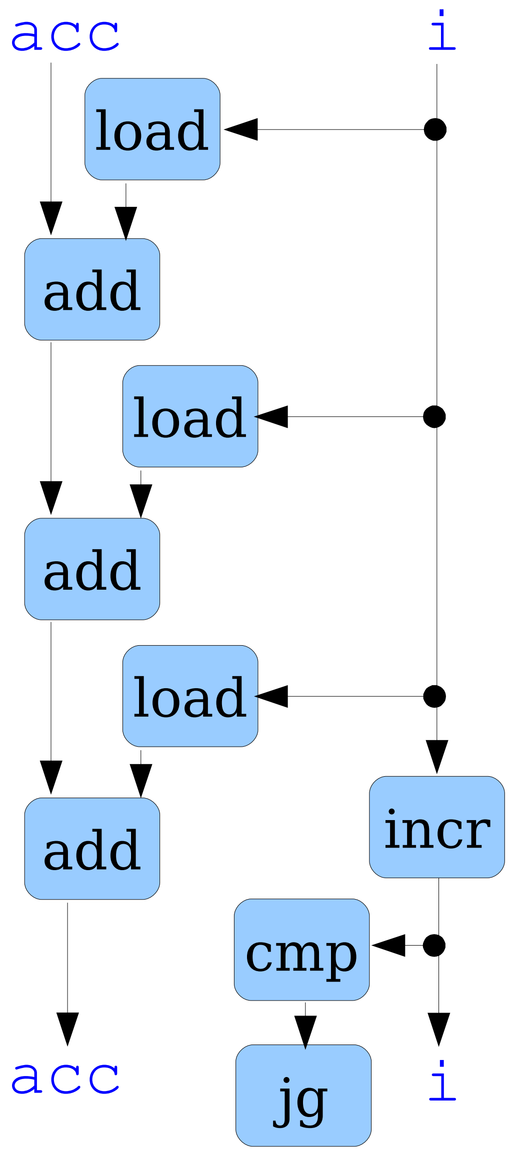
Reduction in loop overhead is critical in achieving CPE that matches integer addition latency.

For integer multiplication, the compiler is automatically applying 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

Effects of Unrolling x3



int, +:

each add has 1-cycle latency
add cannot be issued until the previous add is complete
3-cycle critical path / 3 elements
1.0 CPE

float, +:

each add has 3-cycle latency
add cannot be issued until the previous add is complete
9-cycle critical path / 3 elements
3.0 CPE

Parallelism

- 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).
- However, 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. Why?

Loop Splitting

```
/* 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;
}
```

AKA

“iteration splitting”

Split the set of combining operations into multiple parts and combine the results at the end.

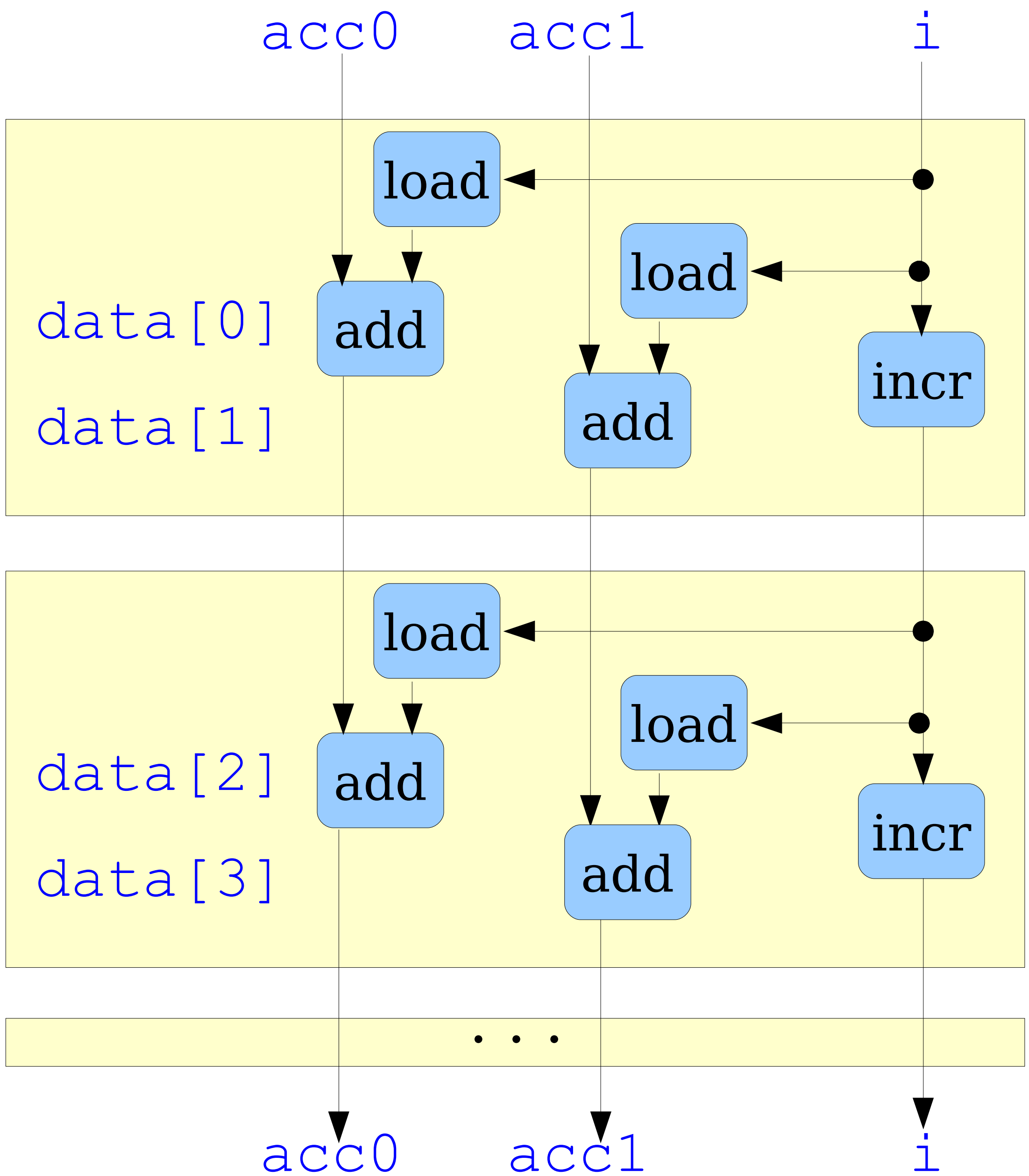
When will this preserve the semantics of the original code?

Example: Loop Splitting

	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, spilt 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.
- 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.
  - Won't happen in general, and big performance gain may outweigh risk.

Effects of Unrolling x2, Splitting x2



`float`, `+`:

3-cycle critical path / 2 elements

1.5 CPE

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`*:*

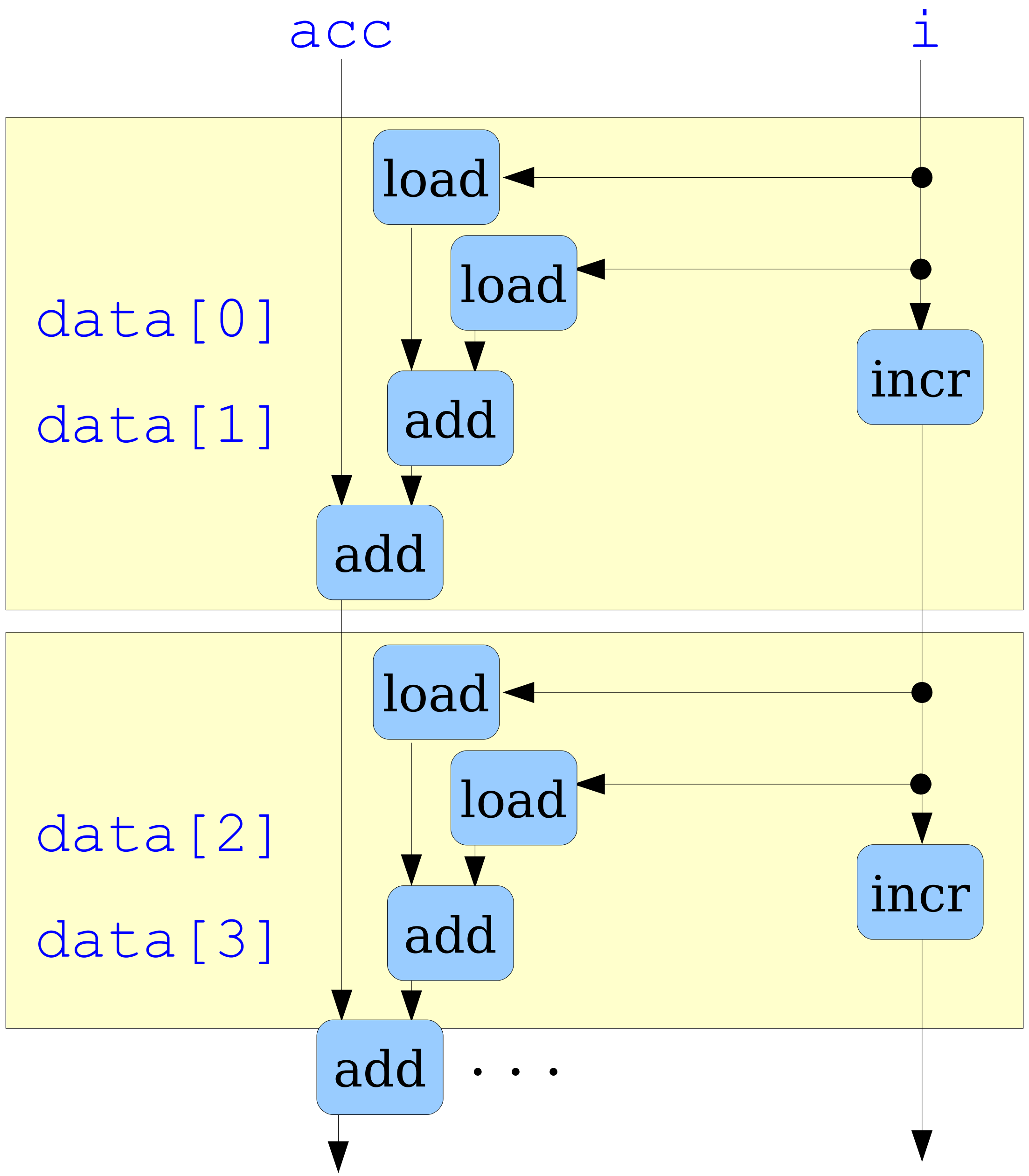
```
   acc = (acc OPER data[i]) OPER data[i+1];
```

Example: Reassociation

	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, spilt 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 well ahead of the currently executing instructions.
  - fetching and decoding new instructions from memory
  - works well so long as the instructions follow a simple sequence
- Upon encountering a branch, the processor must guess which way to go.
  - *speculative execution*—the processor begins to fetch/decode instructions at the predicted branch target
  - avoids modifying actual register and memory locations until the actual outcome of the branch is known

Branch Prediction Outcomes

- If the prediction is correct, the processor "commits" to the results of the speculative execution and continues.
- If the prediction is wrong, the processor discards all of the speculatively-executed results and restarts fetching and decoding instructions at the correct location.
  - incurring a significant *branch penalty*
- Ideas for predicting branches?
- Our running example was not slowed by branch penalties. Prediction was correct almost always. Why?

Branch Prediction Heuristics

- A common heuristic is to predict that any branch to a lower address will be taken, and any branch to a higher address will not be taken.
  - backward branches are used to reenter loops
  - forward branches are used for conditional computation
  - experiments show this heuristic to be correct 65% of the time
- Predicting all branches as taken has a 60% success rate.
- Much more sophisticated strategies have been devised and are in use. (Intel Pentium II, III claim 90-95% correct.)

Performance Improvement

1. Choose appropriate algorithms and data structures.

   Optimizations cannot save a program with poor asymptotic performance.

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

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

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.
- 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.
- Performance of the overall system depends on 2 things.

1. The significance of this part in the overall system.

   Let a be the fraction of time required by this part.

2. The improvement in speed for this part.

   Let k be the factor of improvement for this part.

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

Example: Amdahl's Law

- Suppose that we have optimized a part of the program that requires 60% of the program's original running time.
  - $a = 0.6$
- We have improved the performance of this part by a factor of 3.
  - $k = 3$
- 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.
- $\text{Speedup}_{\infty} = 1 / (1-a)$
- *Example*: Let $a = 0.6$. Net speedup of overall program is still only $1 / 0.4 = 2.5$
- To have a significant impact on the overall program, it is critical to improve the performance of a very large fraction of the program.

Clicker Question

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

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? *CLICK your one-digit answer.*

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

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.
- 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).
- *After the break*: How the memory hierarchy affects program performance and Lab 4—challenges you to make code run faster.