Enoch Hwang

✶ ehwang@cs.ucr.edu ✶

CS203 Advanced Computer Architecture
Notes
February 21, 1995

1. Amdahl's Law

$$\text{ExecutionTime}_{\text{After Improvement}} = \frac{\text{ExecutionTime}_{\text{Affected}}}{\text{Amount of Improvement}} + \text{ExecutionTime}_{\text{Unaffected}}$$

$$\text{Speedup} = \frac{\text{Performance}_{\text{After}}}{\text{Performance}_{\text{Before}}} = \frac{\text{ExecutionTime}_{\text{Before}}}{\text{ExecutionTime}_{\text{After}}}$$

$$= \frac{1}{(1-\text{FE}) + \frac{\text{FE}}{\text{SE}}}$$

where FE (fraction enhanced) = $\frac{\text{ExectionTime}_{\text{Affected}}}{\text{ExectionTime}_{\text{Before}}}$

SE (speedup enhanced) = Amount of Improvement

1.1. Corollary of Amdahl's Law

$$\text{Performance Improvement Limit} \leq \frac{1}{1 - \text{FractionEnhanced}}$$

2. CPU Performance Formula

CPU Execution Time = IC × CPI × CT

where IC = Instructions executed for the program

CPI = Average number of clock cycles per instruction

$$= \sum_i f_i \times \text{CPI}_i$$

where f_i = fraction of occurrence for category $i = \frac{\text{IC}_i}{\text{IC}}$

CT = Clock cycle time = $\frac{1}{\text{clock rate}}$

Machine *x* is *n* times faster than machine *y*: $\frac{\text{Performance}_x}{\text{Performance}_y} = n = \frac{\text{ExecutionTime}_y}{\text{ExecutionTime}_x}$

3. Pipelining

- *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution.
- The time required between moving an instruction one step down the pipeline is a machine cycle.
- The length of a machine cycle is determined by the time required for the slowest pipe stage.
- A functional unit is **fully pipelined** if it can start a new operation in every clock cycle. A floating point unit is fully pipelined if a floating point instruction can be issued in every cycle.

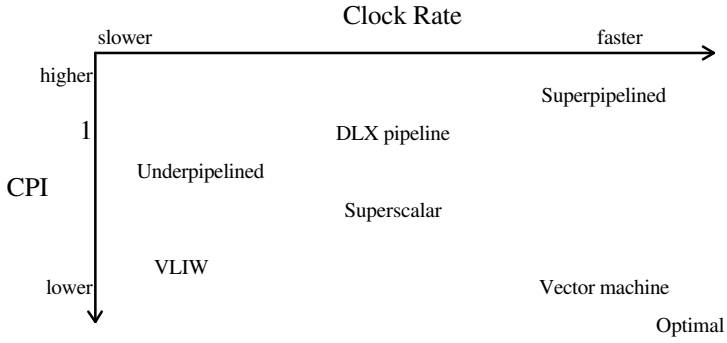
- If the stages are perfectly balanced, then the time per instruction on the pipelined machine--assuming ideal conditions (i.e., no stalls)--is equal to

$$\frac{\text{Time per instruction on nonpipelined machine}}{\text{Number of pipe stages}}$$

- Under these conditions, the speedup from pipelining equals the number of pipe stages. However, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum value.
- Pipelining yields a reduction in the average execution time per instruction. This reduction can be obtained by:
  - decreasing the clock cycle time (increasing the clock frequency) of the pipelined machine.
  - and/or by decreasing the number of clock cycles per instruction (CPI).
- Pipelining increases the CPU instruction throughput--the number of instructions completed per unit time--but it does not reduce the execution time of an individual instruction.

$$\text{Pipeline speedup} = \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}}$$

$$\text{Ideal CPI} = \frac{\text{CPI without pipelining}}{\text{Pipeline depth}}$$



DLX pipeline

- achieves a CPI very close to 1 (ignoring memory-system stalls).
- reasonable clock speed.
- example of a RISC or load/store machine.
- have 5 stages: IF ID EX Mem WB

Extended DLX pipeline page 285

- similar to DLX pipeline but with mutilple EX units
-
  EX
  IF ID EX Mem WB
  EX
- to handle longer floating point operations.
- if we have a sequence of two instructions: DIVF, ADDF,
  it is possible for the ADDF to finish before DIVF because the DIVF takes more cycles to complete. This is call *out-of-order-completion*.

Underpipelined

- lump multiple DLX pipestages into one.
- clock cannot run as fast.
- CPI will be only marginally lower.

Superpipelined

- machines with higher clock rate and deeper pipelines.
- pipelining all functional units.
- may have 10 pipeline stages instead of the 5.
- slightly higher CPI.
- more complexed pipeline scheduling and pipeline control.

Superscalar

- clock rates very close to DLX but smaller CPI, i.e. CPI < 1
- have 1 integer and 1 floating point unit.
- can issue 1 integer and 1 FP operation per cycle.

VLIW

- have much lower CPI.
- can issue 2 integer, 2 FP, 2 mem reference, and 1 branch instruction per cycle.
- issues many instructions at a time.

Vector machine

- are superpipelined machines that work on vectors
- operations on vectors is like issuing multiple independent operations at a time.
- operates on one vector per cycle. One vector is like one complete loop in the DLX.

3.1. Pipeline Hazards

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
   - the most common instances arise when some functional unit is not fully pipelined.
   - another cause is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction.
   - Solution: use *forwarding* (*bypassing, short-circuiting*). Need to do for two instructions.
   - RAW (read after write). Most common type of hazard.
   - WAR (write after read). This cannot happen in DLX because all reads are early (in ID) and all writes are late (in WB).
   - WAW (write after write). Same comment as WAR.
   - RAR (read after read). Not a hazard.
   - Load instructions have a delay or latency of one cycle.
   - *pipeline interlock* detects a hazard and stalls the pipeline until the hazard is cleared.
   - A *delayed load* is a load requiring that the following instruction not use its result.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

- Hazards in pipelines will make it necessary to stall the pipeline.

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycle}}$$

- If we assume that the ideal CPI is 1, then simplify to

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycle}}$$

Pipeline stall cycles (say from branches) = Branch frequency × Branch penalty

- *Pipeline scheduling* is when the compiler tries to schedule the pipeline to avoid these stalls by rearranging the code sequence to eliminate the hazards.
- A load requiring that the following instruction not use its result is called a *delayed load*.
- In general, the deeper the pipeline, the worse the branch penalty in clock cycles.

3.1.1. Reducing pipeline branch penalties

1. Freeze the pipeline.
   - simple.
2. Predict the branch as not taken.
   - simply allowing the hardware to continue as if the branch were not taken.
3. Predict the branch as taken.
4. Pipeline scheduling of the delayed branch.

Strategy	Requirements	Improves performance when?
From before	Branch must not depend on the rescheduled instructions.	Always.
From target	Must be OK to execute instruction if branch is not taken.	When branch is taken.
From fall through	Must be OK to execute instruction if branch is taken.	When branch is not taken.

From before

```
ADD R1, R2, R3
if R2 = 0 then L1
delay slot

L1:
```

becomes

```
if R2 = 0 then L1
ADD R1, R2, R3

L1:
```

From target

```
L1: SUB R4, R5, R6

Add R1, R2, R3
if R1 = 0 then L1
delay slot
```

becomes

```
L1:
ADD R1, R2, R3
if R1 = 0 then L1
SUB R4, R5, R6
```

From fall through

```
ADD R1, R2, R3
if R1 = 0 then L1
delay slot
SUB R4, R5, R6
L1:
```

becomes

```
ADD R1, R2, R3
if R1 = 0 then L1
SUB R4, R5, R6
L1:
```

3.2. Advanced Pipelining - Dynamic Scheduling

- out-of-order execution
- scoreboarding - a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences.

- the goal of ascoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible.

3.2.1. hardware branch prediction page 307

- Using a *branch-prediction buffer* - a small memory indexed by the lower portion of the branch instruction address. The memory contains a bit that says whether the branch was recently taken or not - 1 bit prediction.
- 2-bit prediction. A prediction must miss twice in a row before it is changed.
- does not help in the DLX pipeline because the DLX pipeline finds out both whether the branch is taken and what the target of the branch is at the same time (in the EX stage).
- A *branch-target buffer* is a branch-prediction cache that stores the predicted address for the next instruction after a branch.
  - because we are predicting the next instruction address and will send it out before decoding the instruction, we must know whether the fetched instruction is predicted as a taken branch.
  - if the PC of the fetched instruction matches a PC in the branch-target buffer, fetching begins immediately at the corresponding predicted PC.
  - unlike the branch-prediction buffer, the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch.

3.3. Advanced Pipelining - More Instruction-Level Parallelism

- Need to decrease the CPI to less than one.
- Loop unrolling, followed by scheduling.

LD	ADDD	BNEZ
stall	stall	stall
ADDD	stall	
	SD	

- A Superscalar version of DLX with loop unrolling (4 times) and scheduling:

Integer instruction	FP instruction	Clock cycle
LD		1
LD		2
LD	ADDD	3
LD	ADDD	4
LD	ADDD	5
SD	ADDD	6
SD	ADDD	7
SD		8
SD		9
SUB		10
BNEZ		11
SD		12

12/5 = 2.4 clock cycles per element.

- A VLIW version with loop unrolling (9 times) and scheduling:

Mem Reference	Mem Ref 2	FP op 1	FP op 2	Int op / branch
LD	LD			
LD	LD			
LD	LD	ADDD	ADDD	
LD	LD	ADDD	ADDD	
LD	LD	ADDD	ADDD	
SD	SD	ADDD	ADDD	
SD	SD	ADDD	ADDD	
SD	SD			
SD	SD			SUB
SD	SD			BNEZ

10/10 = 1 clock cycle per element.

4. Vector Processors

- Vector functional units - each unit is fully pipelined and can start a new operation on every clock cycle.
- *start-up time* is the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used.
- *initiation rate* is the time per result once a vector instruction is running (usually 1).
- time to complete a single vector operation of length n = start-up time + $n \times$ initiation rate.
- 2 implementations to access the memory modules in parallel:
  - synchronize access - read all the modules at the same time from the same offset.
  - independent bank (module) phasing. should have
    number of memory modules (m) ≥ memory module access time (d)

4.1. Two Real-World Issues

page 364

4.1.1. Vector Length

- do "strip mining."

4.1.2. Stride

- s = stride, m = # of memory modules, d = access time, latency = startup time, n = length of vector.

  # of distinct modules = # of elements = $\frac{LCM(s,m)}{s} = \frac{m}{GCD(s,m)}$

  if $\frac{LCM(s,m)}{s} < d$ then

  IR $\neq$ 1 $\Rightarrow$ bank conflict $\Rightarrow$ stall

  do the ratio to get total access time: elements : d

  n :?

  else

  initiation rate (IR) = 1

  total access time = startup + n × initiation rate = startup + n

- Ex. m = 16, d = 12, n = 64, s=1 and 32. How long will it take to complete the load?
  For s = 1, $\frac{LCM(1,16)}{1}$ = 16 > 12 = d. Therefore, IR = 1. The load will take 12 +64 = 76 clock cycles.
  For s = 32, $\frac{LCM(32,16)}{32} = \frac{32}{32}$ = 1 < 12. Therefore, IR ≠ 1. The access time is $\frac{12 \times 64}{1}$ = 768 clock cycles.
  For s = 18, $\frac{LCM(18,16)}{18} = \frac{16}{GCD(18,16)} = \frac{16}{2}$ = 8 < 12. Access time is $\frac{12 \times 64}{8}$ = 96.

4.2. Compiler Technology for Vector Machines

page 371

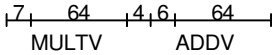
- To recognize whether a loop is vectorizable or not. If yest, then generate the approprate code.
- Vectorizable if a statement S1 uses a value computed by S2 in the same iteration.
  NOT vectorizable if S1 uses a value computed by S1 or S2 in an earlier iteration.
- Test: Given an array assignment statement
  A[a j +b] := A[c k + d]
  if GCD(a,c) divides (d-b), i.e. $\frac{(d-b)}{GCD(a,c)}$ has no remainder then
  may or may not have dependency.
  else
  no dependency.
- Example. Any dependency in the statement X(2*i + 3) = X(2*i) ?
  a=2, b=3, c=2, d=0. d-b = 0-3 = -3. GCD(a,c) = GCD(2,2) = 2. -3/2 has a remainder. ⇒ does not divide ⇒ no dependency.
- Example. Any dependency in the statement X(i) = X(i-2) ?
  a=1, b=0, c=1, d=-2. $\frac{-2-0}{GCD(1,1)} = \frac{-2}{1}$ = -2. ⇒ divides ⇒ may or may not have dependency. However from the statements,
  X(2) = X(0) i = 2
  X(3) = X(1) i = 3
  X(4) = X(2)
  ...
  we see that there is no dependency.

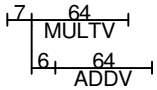
4.3. Enhancing Vector Performance

page 377

4.3.1. Chaining

- Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available.
- This is like the pipelining concept of *forwarding* that works on the individual elements of a vector.
- The results from the first functional unit in the chain are forwarded to the second function unit.

- In a chained sequence the initiation rate is equal to one per clock cycle if the functional units in the chained operations are all fully pipelined.
- Example, MULTV V1, V2, V3
  ADDV V4, V1, V5
  ADDV must be stalled if no chaining. However, with chaining, ADDV can start earlier.
  Assume start-up-time for MULTV = 7, start-up-time for ADDV = 6, stall penalty = 4 clcyes.
  Without chaining 

. Total = 145 cycles for 128 operations $\Rightarrow$ $\frac{128 operations}{145 cycles} = 0.9 \frac{operations}{cycles}$.
  With chaining 

. Total = 7+64+6 = 77 cycles for 128 operations $\Rightarrow$ $\frac{128 operations}{77 cycles} = 1.7 \frac{operations}{cycles}$.

4.3.2. Conditionally executed statements

- Use vector-mask (VM).
- Example, for i = 1 to 64
  if A(i) ≠ 0 then
  A(i) = A(i) - B(i)
  This loop cannot normally be vectorized because of the **if** statement. However, we can use the vector mask.
  1. set the vector mask VM to 1 if A(i) ≠ 0.
  2. do the subtract under vector mask, i.e. the subtract operation is executed only on the vector element whose corresponding entry in the VM is 1.
- disadvantages: (1) execution time is not decreased. (2) if the VM serves only to disable the storing of the result, then if the operation is a divide, then divide-by-zero errors may occur.

4.3.3. Sparse matrices

- *scatter - gather* operations.

4.3.4. Vector reduction

- A loop that reduces an array to a single value. Example

```
      dot = 0
      do 10 i = 1,64
10    dot = dot + A(i) × B(i)
```

not vectorizable, but we can split it into two halves...

```
      do 10 i = 1,64
10    dot(i) = A(i) × B(i)        // vectorizable
      do 20 i = 1,64
20    dot = dot + dot(i)          // not vectorizable
```

5. Memory-Hierarchy Design

- *Temporal locality* and *spatial locality*.
- *Hit rate* is the fraction of cache accesses found.
- *Miss rate* = (1 - hit rate) is the fraction of cache accesses not found.
- *Hit time* is the time to access the cache.
- *Miss penalty* is the time to replace a block in the cache with the corresponding block from the memory, plus the time to deliver this block to the CPU.
- The memory address is divided into 2 pieces, 1) the *block-frame address* is the higher-order of the address. 2) the *block-offset address* is the lower-order of the address.
- Average memory-access time = hit time + miss rate × miss penalty.

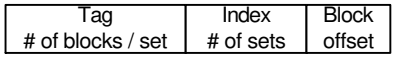
5.1. Caches

5.1.1. Where can a block be placed in a cache?

1. *Direct mapped* if each block has only one place it can appear in the cache. (1-way set associative).
2. *Fully associative* if a block can be placed anywhere in the cache. (*n*-way set associative). (block-frame address) modulo (number of **blocks** in cache).
3. *Set associative* if a block can be placed in a restricted set of blocks in the cache. A set is a group of two or more blocks in the cache. Once the set is determined, it can be placed anywhere within that set. *n-way set associative* if there are *n* blocks in a set.
   (block-frame address) modulo (number of **sets** in cache).

5.1.2. How is a block found if it is in the cache?

- include an address tag on each block that gives the block frame address.
- all possible tags are searched in parallel using a TLB for fully associative and set-associative.
- there is also a *valid bit* to the tag to say whether or not this entry contains a valid address.



- increasing associativity increases the number of blocks per set → decreasing the size of the index → increasing the size of the tag.
- fully associative = 1 set with *m* blocks.
- direct mapped = *m* sets each with 1 block.

5.1.3. Which block should be replaced on a cache miss?

- with fully associative or set associative placement, there are several blocks to choose from.
- random
- least-recently used

5.1.4. What happens on a write?

- *write through* - data is written to both the cache and the memory.
- *write back* - data is written only to the cache. The modified cache block is written to memory only when it is replaced. A *dirty bit* is used to see whether the block needs to be written back.
- Three sources of cache misses:

1. *compulsory* - the first access to a block is not in the cache, so the block must be brought into the cache.
2. *capacity* - if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
3. *conflict* - a block can be discarded and later retrieved if too many blocks map to its set.

- spatial locality → larger block sizes reduce compulsory misses → reduce the number of blocks in the cache → increase conflict misses.

Techniques for Fast Address Translation

- *translation-lookaside buffer* (TLB).