Chapter 15 -- performance features

```
Architectural Features used to enhance performance
--------------------------------------------------

What is a "better" computer?  What is the "best" computer?
  The factors involved are generally cost and performance.

  COST FACTORS: cost of hardware design
                cost of software design (OS, applications)
                cost of manufacture
                cost to end purchaser
  PERFORMANCE FACTORS:
                what programs will be run?
                how frequently will they be run?
                how big are the programs?
                how many users?
                how sophisticated are the users?
                what I/O devices are necessary?


  (this chapter discusses ways of increasing performance)

There are two ways to make computers go faster.

 1. Wait a year.  Implement in a faster/better/newer technology.
    More transistors will fit on a single chip.
    More pins can be placed around the IC.
    The process used will have electronic devices (transistors)
      that switch faster.

 2. new/innovative architectures and architectural features, and
    clever implementations of existing architectures.




MEMORY HIERARCHIES
------------------
Known in current technologies:  the time to access data
   from memory is an order of magnitude greater than a
   processor operation. (And note that this has been true
   for well more than a decade.)

   For example:  if a 32-bit 2's complement addition takes 1 time unit,
   then a load of a 32-bit word takes about 10 time units.

Since every instruction takes at least one memory access (for
the instruction fetch), the performance of computer is dominated
by its memory access time.

  (to try to help this difficulty, we have load/store architectures,
   where most instructions take operands only from memory.  We also
   try to have fixed size, SMALL size, instructions.)
```

what we really want:
very fast memory -- of the same speed as the CPU
very large capacity -- 512 Mbytes
low cost -- $50

these are mutually incompatible. The faster the memory, the more expensive it becomes. The larger the amount of memory, the slower it becomes.

What we can do is to compromise. Take advantage of the fact (fact, by looking at many real programs) that memory accesses are not random. They tend to exhibit LOCALITY.
LOCALITY -- nearby.
2 kinds:

Locality in time (temporal locality)
if data has been referenced recently, it is likely to be referenced again (soon!).

example: the instructions with in a loop. The loop is likely to be executed more than once. Therefore, each instruction gets referenced repeatedly in a short period of time.

example: The top of stack is repeatedly referenced within a program.

Locality in space (spatial locality)
if data has been referenced recently, then data nearby (in memory) is likely to be referenced soon.

example: array access. The elements of an array are neighbors in memory, and are likely to be referenced one after the other.

example: instruction streams. Instructions are located in memory next to each other. Our model for program execution says that unless the PC is explicitly changed (like a control instruction) sequential instructions are fetched and executed.

We can use these tendencies to advantage by keeping likely to be referenced (soon) data in a faster memory than main memory. This faster memory is called a CACHE.

```
        processor-cache   <----------------> memory
```

It is located very close to the processor. It contains COPIES of PARTS of memory.

A standard way of accessing memory, for a system with a cache:
(The programmer doesn't see or know about any of this)

memory access (for an instruction fetch or to get operands or to write results) goes to the cache.
If the data is in the cache, then we have a HIT.
The data is returned to to the processor (from the cache),

```
    and the memory access is completed.
 If the data is not in the cache, then we have a MISS.
    The memory access is then sent on to main memory.

 On average, the time to do a memory access is

       = cache access time + (% misses  *  memory access time)

This average (mean) access time will change for each program.
It depends on the program, and its reference pattern, and how
that pattern interracts with the cache parameters.




cache is managed by hardware

        Keep recently-accessed blocks of memory in the cache,
         this exploits temporal locality

        Break memory into aligned blocks (lines), this
         exploits spatial locality

        transfer data to/from the cache in blocks

        put block into a predefined location, its frame

        Each block has
         state (valid) -- is there anything in this frame?
         address tag -- a way to distinguish which block from
                        memory is currently in this frame.
         data  -- the block of data, a copy of what is in memory.

>>>> simple CACHE DIAGRAM here <<<<
```

```
A Simplified Example:

   Addresses are 5 bits.
   Blocks are 4 bytes.
   Memory is byte addressable.
   There are 4 blocks in the cache.

   Assume the cache is empty at the start of the example.

   (line number)  valid  tag  data (in hex)
       00           0     ?   0x?? ?? ?? ??
```

```
    01           0     ?   0x?? ?? ?? ??
    10           0     ?   0x?? ?? ?? ??
    11           0     ?   0x?? ?? ?? ??
```

Memory is small enough that we can make up a complete example. Assume little endian byte numbering.

```
  address   contents
  (binary)   (hex)
   00000    aa bb cc dd
   00100    00 11 22 33
   01000    ff ee 01 23
   01100    45 67 89 0a
   10000    bc de f0 1a
   10100    2a 3a 4a 5a
   11000    6a 7a 8a 9a
   11100    1b 2b 3b 4b
```

(1)
First memory reference is to the byte at address 01101.

The address is broken into 3 fields:

```
  tag   line number   byte within block
   0        11              01
```

On line 11, the block is marked as invalid, therefore we have a cache MISS.

The block that address 01101 belongs to (4 bytes starting at address 01100) is brought into the cache, and the valid bit is set.

```
(line number)  valid  tag  data (in hex)
    00           0     ?   0x?? ?? ?? ??
    01           0     ?   0x?? ?? ?? ??
    10           0     ?   0x?? ?? ?? ??
    11           1     0   0x45 67 89 0a
```

And, now the data requested can be supplied to the processor. It is the value 0x89.

(2)
Second memory reference is to the byte at address 01010.

The address is broken into 3 fields:

```
  tag   line number   byte within block
   0        10              10
```

On line 10, the block is marked as invalid, therefore we have a cache MISS.

The block that address 01010 belongs to (4 bytes starting at address 01000) is brought into the cache, and the valid bit is set.

```
(line number)  valid  tag  data (in hex)
    00           0     ?   0x?? ?? ?? ??
    01           0     ?   0x?? ?? ?? ??
    10           1     0   0xff ee 01 23
```

```
    11           1     0   0x45 67 89 0a

And, now the data requested can be supplied to the processor.
It is the value 0xee.

(3)
Third memory reference is to the byte at address 01111.

The address is broken into 3 fields:
  tag   line number   byte within block
   0        11             11

This line within the cache has its valid bit set, so there
is a block (from memory) in the cache.  BUT, is it the
block that we want?  The tag of the desired byte is checked
against the tag of the block currently in the cache.  They
match, and therefore we have a HIT.

The value 0x45 (byte 11 within the block) is supplied to
the processor.



(4)
Fourth memory reference is to the byte at address 11010.

The address is broken into 3 fields:
  tag   line number   byte within block
   1        10             10

This line within the cache has its valid bit set, so there
is a block (from memory) in the cache.  BUT, is it the
block that we want?  The tag of the desired byte is checked
against the tag of the block currently in the cache.  They
do NOT match.  Therefore, the block currently in the cache
is the wrong one.  It will be overwritten with the block
(from memory) that we now do want.

(line number)  valid  tag  data (in hex)
    00           0     ?   0x?? ?? ?? ??
    01           0     ?   0x?? ?? ?? ??
    10           1     1   0x6a 7a 8a 9a
    11           1     0   0x45 67 89 0a

The value 0x7a (byte 10 within the block) is supplied to
the processor.

(5)
Fifth memory reference is to the byte at address 11011.

The address is broken into 3 fields:
  tag   line number   byte within block
   1        10             11

This line within the cache has its valid bit set, so there
is a block (from memory) in the cache.  BUT, is it the
block that we want?  The tag of the desired byte is checked
against the tag of the block currently in the cache.  They
match, and therefore we have a HIT.

The value 0x6a (byte 11 within the block) is supplied to
```

```
   the processor.


Often
        cache:  instruction cache 1 cycle
                data cache 1 cycle
        main memory 20 cycles

Performance for data references w/ miss ratio 0.02 (2% misses)

        mean access time = cache-access + miss-ratio * memory-access
                         =       1     +   0.02     *  20
                         =       1.4


Typical cache size is 64K byte given a 64Mbyte memory
        20 times faster
        1/1000 the capacity
        often contains 98% of the references



Remember:

recently accessed blocks are in the cache (temporal locality)

the cache is smaller than main memory, so not all blocks are in the cache.

blocks are larger than 1 word (spatial locality)





This idea of exploiting locality is (can be) done at many
levels.  Implement a hierarchical memory system:

  smallest, fastest, most expensive memory         (registers)
  relatively small, fast, expensive memory         (CACHE)
  large, fast as possible, cheaper memory          (main memory)
  largest, slowest, cheapest (per bit) memory       (disk)



registers are managed/assigned by compiler or asm. lang programmer

cache is managed/assigned by hardware or partially by OS

main memory is managed/assigned by OS

disk managed by OS



Programmer's model:  one instruction is fetched and executed at
  a time.

Computer architect's model:  The effect of a program's execution are
  given by the programmer's model.  But, implementation may be
  different.
```

```
   To make execution of programs faster, we attempt to exploit
   PARALLELISM:  doing more than one thing at one time.

   program level parallelism:  Have one program run parts of itself
     on more than one computer.  The different parts occasionally
     synch up (if needed), but they run at the same time.
   instruction level parallelism (ILP):  Have more than one instruction
     within a single program executing at the same time.

 PIPELINING  (ILP)
 -----------------
  concept
  -------
    A task is broken down into steps.
    Assume that there are N steps, each takes the same amount of time.

    (Mark Hill's) EXAMPLE:  car wash

      steps:  P -- prep
              W -- wash
              R -- rinse
              D -- dry
              X -- wax

      assume each step takes 1 time unit

      time to wash 1 car (red) = 5 time units
      time to wash 3 cars (red, green, blue) = 15 time units

      which car      time units
                 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        red      P  W  R  D  X
        green                   P  W  R  D  X
        blue                                   P  W  R  D  X

    a PIPELINE overlaps the steps

      which car      time units
                 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        red      P  W  R  D  X
        green       P  W  R  D  X
        blue           P  W  R  D  X
        yellow            P  W  R  D  X
           etc.

          IT STILL TAKES 5 TIME UNITS TO WASH 1 CAR,
          BUT THE RATE OF CAR WASHES GOES UP!




    Pipelining can be done in computer hardware.

  2-stage pipeline
  ----------------
   steps:
     F -- instruction fetch
     E -- instruction execute (everything else)


     which instruction       time units
```

```
                     1  2  3  4  5  6  7  8 . . .
      1              F  E
      2                 F  E
      3                    F  E
      4                       F  E


      time for 1 instruction =  2 time units
        (INSTRUCTION LATENCY)

      rate of instruction execution = pipeline depth * (1 / time for     )
        (INSTRUCTION THROUGHPUT)                           1 instruction
                                    =        2      * (1 /   2)
                                    =   1 per time unit
```

5-stage pipeline
----

a popular pipelined implementation, that works really well for teaching about pipelines and also for load/store architectures

Its application to the Pentium would be problematic.

```
    steps:
       IF -- instruction fetch
       D  -- instruction decode
       OA -- operand access
       OP -- ALU operation (can be effective address calculation)
       R  -- store results

   which       time units
instruction   1   2   3   4   5   6   7  8 . . .
     1        IF  D   OA  OP  R
     2            IF  D   OA  OP  R
     3                IF  D   OA  OP  R

   INSTRUCTION LATENCY = 5 time units
   INSTRUCTION THROUGHPUT = 5 * (1 / 5) = 1 instruction per time unit
```

unfortunately, pipelining introduces other difficulties. . .

data dependencies
----

suppose we have the following code:

```
  mov  EAX, data1
  add  EBX, EAX
```

the data moved (loaded into a register) doesn't get written to EAX until R,
but the add instruction wants to get the data out of EAX it its D stage. . .

```
   which       time units
instruction   1   2   3   4   5   6   7  8 . . .
```

```
    mov       IF  D   OA  OP  R
                              ^
    add           IF  D   OA  OP  R
                      ^
```

the simplest solution is to STALL the pipeline.
(Also called HOLES, HICCOUGHS or BUBBLES in the pipe.)

```
    which       time units
 instruction   1   2   3   4   5   6   7   8 . . .
    mov       IF  D   OA  OP  R
                              ^
    add           IF  D   D   D   OA  OP  R
                      ^   ^   ^ (pipeline stalling)
```

A DATA DEPENDENCY (also called a HAZARD) causes performance to decrease.

more on data dependencies

Read After Write (RAW) --
(example given), a read of data is needed before it has been written

Given for completeness, not a difficulty to current pipelines in practice, since the only writing occurs as the last stage.

Write After Read (WAR) --
Write After Write (WAW) --

NOTE: there is no difficulty implementing a 2-stage pipeline due to DATA dependencies!

control dependencies
----

what happens to a pipeline in the case of control instructions?

PENTIUM CODE SEQUENCE:

```
        jmp  label1
        inc  eax
label1: mult ecx
```

```
    which       time units
 instruction   1   2   3   4   5   6   7  8 . . .
    jmp       IF  D   OA  OP  R
                              ^ (PC changed here)
    inc           IF  D   OA  OP  R
                  ^^  (WRONG instruction fetched here!)
```

whenever the PC changes (except for the PC update step)
we have a CONTROL DEPENDENCY.

```
       CONTROL DEPENDENCIES break pipelines.  They cause
       performance to plummet.

       So, lots of (partial) solutions have been implemented
       to try to help the situation.
         Worst case, the pipeline must be stalled such that
         instructions are going through sequentially.

       Note that just stalling doesn't really help, since
         the (potentially) wrong instruction is fetched
         before it is determined that the previous instruction
         is a branch.




BRANCHES and PIPELINING
-----------------------
 (or, how to minimize the effect of control dependencies on pipelines.)

 easiest solution (poor performance)
    Cancel anything (later) in the pipe when a jump is decoded.
    This works as long as nothing changes the program's state
    before the cancellation.  Then let the branch instruction
    finish (flush the pipe), and start up again.

       which       time units
   instruction   1   2   3   4   5   6   7  8 . . .
       jmp       IF  D   OA  OP  R
                              ^ (PC changed here)
       inc           IF
                     ^^ (cancelled)

 branch Prediction (static or dynamic)
   add lots of extra hardware to try to help.

   a)  (static)  assume that the branch/jump will not be taken
       When the decision is made, the hw "knows" if the correct
       instruction has been partially executed.

       If the correct instruction is currently in the pipe,
         let it (and all those after it) continue.  Then,
         there will be NO holes in the pipe.
       If the incorrect instruction is currently in the pipe,
         (meaning that the branch/jump was taken), then all instructions
         currently in the pipe subsequent to the branch must
         be BACKED OUT.

   b)  (dynamic) A variation of (a).
       Have some extra hw that keeps track of which branches have
       been taken in the recent past.  Design the hw to presume that
       a branch will be taken the same way it was previously.
       If the guess is wrong, back out as in (a).

       Question for the advanced student:  Which is better, (a) or (b)? Why?

   NOTE:  solution (a) works quite well with currently popular
      pipeline solutions, because no state information is changed
      until the very last stage of an instruction.  As long as
      the last stage hasn't started, backing out is a matter
```

```
      of stopping the last stage from occuring and getting the
      PC right.



 separate test from branch
   make the conditional test and address calculation
   separate instructions from the one that changes the PC.

   This reduces the number of holes in the pipe.


 squashing
   A fancy name for branch prediction that always presumes the
   branch will be taken,  and keeps a copy of the PC that will
   be needed in the case of backing out.




Amdahl's Law
------------

(Or why the common case matters most)

speedup = new rate / old rate

        = old execution time / new execution time


We program in some enhancement to part of our program.
  The fraction of time spent in that part of the code is f.
  The speedup of that part of the code (f) is S.

  ( Let an enhancement speedup f fraction of the time by speedup S)

speedup = [(1-f)+f]*old time / (1-f) * old time + f/S * old time

        =    1
          ---------
          1-f + f/S

Examples

            f       S           speedup
           ---     ---          -------
           95%     1.10         1.094
            5%     10           1.047
            5%     inf          1.052


lim                 1
                ---------         =  1/ 1-f
S --> inf       1-f + f/S

        f       speedup
       ---      -------
       1%       1.01
       2%       1.02
       5%       1.05
```

```
        10%      1.11
        20%      1.25
        50%      2.00
```

This says that we should concentrate on the common case!