2_measuring_performance_of_memory_hierarchy

October 11, 2018

0.1 measuring memory latency

In this notebook we will investigate the distribution of latency times for different size arrays.

1. **Goal 1:** Measure the effects of caching **in the wild**
2. **Goal 2:** Undestand how to study long-tail distributions.

0.1.1 Import modules

```
In [1]: %pylab inline
        from numpy import *
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: import time
        from matplotlib.backends.backend_pdf import PdfPages

        from os.path import isfile,isdir
        from os import mkdir
        import os
```

Set log directory This script generates large files. We put these files in a separate directory so it is easier to delete them later.

#run this cell only once

```
In [3]: ## Remember the path for home and  log directories
        home_base,=!pwd
        log_root=home_base+'/logs'
        if not isdir(log_root):
            mkdir(log_root)
```

```
In [4]: from lib.measureRandomAccess import measureRandomAccess
        from lib.PlotTime import PlotTime
        from lib.create_file import create_file,tee
```

0.1.2 defining memory latency

Latency is the time difference between the time at which the CPU is issuing a read or write command and, the time the command is complete.

- This time is very short if the element is already in the L1 Cache,
- and is very long if the element is in external memory (disk or SSD).

0.1.3 setting parameters

- We test access to elements arrays whose sizes are:
  - m_legend=['Zero','1KB','1MB','1GB','10GB']
- Arrays are stored **in memory** or **on disk**
- We perform 100,000 read/write ops to random locations in the array.
- We analyze the **distribution** of the latencies as a function of the size of the array.

```
In [5]: m_list=[0]+[int(10**i) for i in [3,6,9,10]]
        m_legend=['Zero','1KB','1MB','1GB','10GB']
        L=len(m_list)
        k=100000 # number of pokes
        print('m_list=',m_list)
```

```
m_list= [0, 1000, 1000000, 1000000000, 10000000000]
```

0.1.4 Set working directory

This script generates large files. We put these files in a separate directory so it is easier to delete them later.

```
In [6]: TimeStamp=str(int(time.time()))
        log_dir=log_root+'/'+TimeStamp
        mkdir(log_dir)
        %cd $log_dir
        stat=open('stats.txt','w')

        def tee(line):
            print(line)
            stat.write(line+'\n')
```

```
/Users/josem/src/docencia/bde/Section1-Spark-Basics/0.MemoryLatency/logs/1539244488
```

```
In [7]: _mean=zeros([2,L])    #0: using disk, 1: using memory
        _std=zeros([2,L])
        _block_no=zeros([L])
        _block_size=zeros([L])
        T=zeros([2,L,k])
```

In [8]:
```
# %load /Users/yoavfreund/academic.papers/Courses/BigDataAnalytics/BigData_spring2016/
import time

stat=open('stats.txt','w')

def tee(line):
    print(line)
    stat.write(line+'\n')

def create_file(n,m,filename='DataBlock'):
    """Create a scratch file of a given size

    :param n: size of block
    :param m: number of blocks
    :param filename: desired filename
    :returns: time to allocate block of size n, time to write a file of size m*n
    :rtype: tuple

    """
    t1=time.time()
    A=bytearray(n)
    t2=time.time()
    file=open(filename,'wb')
    for i in range(m):
        file.write(A)
        if i % 100 == 0:
            print('\r',i,",", end=' ')
    file.close()
    t3=time.time()
    tee('\r              \ncreating %d byte block: %f sec, writing %d blocks %f sec' %
    return (t2-t1,t3-t2)
```

In [9]:
```
Random_pokes=[]
Min_Block_size=1000000
for m_i in range(len(m_list)):

    m=m_list[m_i]
    blocks=int(m/Min_Block_size)
    if blocks==0:
        _block_size[m_i]=1
        _block_no[m_i]=m
    else:
        _block_size[m_i]=Min_Block_size
        _block_no[m_i]=blocks
    (t_mem,t_disk) = create_file(int(_block_size[m_i]),int(_block_no[m_i]),filename='Bl

    (_mean[0,m_i],_std[0,m_i],T[0,m_i]) = measureRandomAccess(m,filename='BlockData'+st
    T[0,m_i]=sorted(T[0,m_i])
```

```
            tee('\rFile pokes _mean='+str(_mean[0,m_i])+', file _std='+str(_std[0,m_i]))

            (_mean[1,m_i],_std[1,m_i],T[1,m_i]) = measureRandomAccess(m,filename='',k=k)
            T[1,m_i]=sorted(T[1,m_i])
            tee('\rMemory pokes _mean='+str(_mean[1,m_i])+', Memory _std='+str(_std[1,m_i]))

            Random_pokes.append({'m_i':m_i,
                                'm':m,
                                'memory__mean': _mean[1,m_i],
                                'memory__std': _std[1,m_i],
                                'memory_largest': T[1,m_i][-1000:],
                                'file__mean': _mean[0,m_i],
                                'file__std': _std[0,m_i],
                                'file_largest': T[0,m_i][-1000:]
                        })
        print('='*50)
```

```
creating 1 byte block: 0.000002 sec, writing 0 blocks 0.000582 sec
File pokes _mean=1.1614322662353516e-07, file _std=3.793294754862043e-07
Memory pokes _mean=1.0772943496704101e-07, Memory _std=3.2698493840319536e-07

creating 1 byte block: 0.000002 sec, writing 1000 blocks 0.003074 sec
File pokes _mean=1.7788701057434083e-05, file _std=7.385846211782043e-06
Memory pokes _mean=1.677107810974121e-07, Memory _std=3.799225482056638e-07

creating 1000000 byte block: 0.000251 sec, writing 1 blocks 0.001992 sec
File pokes _mean=2.1449880599975586e-05, file _std=1.6309410905446207e-05
Memory pokes _mean=2.101755142211914e-07, Memory _std=4.1441149213990107e-07

creating 1000000 byte block: 0.000189 sec, writing 1000 blocks 1.807132 sec
File pokes _mean=0.00033956289768218993, file _std=0.0025453737077633085
Memory pokes _mean=1.334238052368164e-06, Memory _std=4.933203884108723e-06

creating 1000000 byte block: 0.000247 sec, writing 10000 blocks 18.197443 sec
File pokes _mean=0.00036260735034942626, file _std=0.00040367432393033234
Memory pokes _mean=6.293690204620361e-06, Memory _std=1.7873528824431927e-05
==================================================
```

```
In [10]: fields=['m', 'memory__mean', 'memory__std','file__mean','file__std']
         print('| block size | mem mean  | mem std | disk mean | disk std |')
         print('| :--------- | :----------- | :--- | :-------- | :------- |')
         for R in Random_pokes:
             tup=tuple([R[f] for f in fields])
             print('| %d | %6.3g | %6.3g |  %6.3g | %6.3g |'%tup)
```

```
| block size | mem mean  | mem std | disk mean | disk std |
| :--------- | :----------- | :--- | :-------- | :------- |
```

```
| 0 | 1.08e-07 | 3.27e-07 |  1.16e-07 | 3.79e-07 |
| 1000 | 1.68e-07 | 3.8e-07 |  1.78e-05 | 7.39e-06 |
| 1000000 | 2.1e-07 | 4.14e-07 |  2.14e-05 | 1.63e-05 |
| 1000000000 | 1.33e-06 | 4.93e-06 |  0.00034 | 0.00255 |
| 10000000000 | 6.29e-06 | 1.79e-05 |  0.000363 | 0.000404 |
```

0.1.5 Mean and std of latency for random access

* Note that `zero` is not really zero. * system time is accurate to micro-second, not nano-second. * SSD random access latency is $10^{-5} - 10^{-4}$

0.1.6 Digging deeper

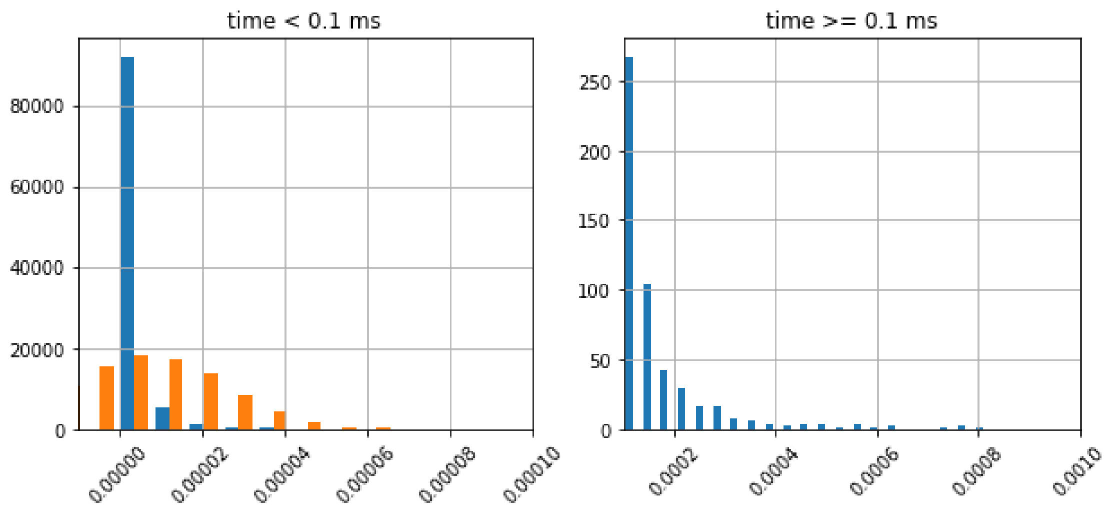
- THe mean and std are the first statistics to look at.
- but the distribution might have more to tell us.

0.1.7 First, lets try histograms

```
In [11]: m_i=4
         Disk_Mem=1
         print('Disk Block of size %2.1g GB'%(m_list[m_i]/1e9))
         print('\r latency mean='+str(_mean[1,m_i])+',  std='+str(_std[1,m_i]))

         _mean_t=_mean[Disk_Mem,m_i]
         _std_t=_std[Disk_Mem,m_i]
         _normal=random.normal(loc=_mean_t,scale=_std_t,size=T.shape[2])
         tmp=T[Disk_Mem,m_i]
         print(' Fraction of zeros=',sum(tmp==0)/len(tmp))
         figure(figsize=(10,4))
         subplot(121)
         thr=1e-4
         hist([tmp[tmp<thr],_normal[_normal<thr]],bins=20);
         #ylim([0,20000])
         xlim([-thr/10,thr])
         title('time < %3.2g ms'%(thr*1e3))
         xticks(rotation=45)
         grid()
         subplot(122)
         hist([tmp[tmp>=thr],_normal[_normal>=thr]],bins=40);
         xlim([thr,thr*10])
         #ylim([0,20])
         title('time >= %3.2g ms'%(thr*1e3))
         xticks(rotation=45);
         grid();
```

```
Disk Block of size 1e+01 GB
 latency mean=6.293690204620361e-06,  std=1.7873528824431927e-05
```

```
Fraction of zeros= 0.05603
```



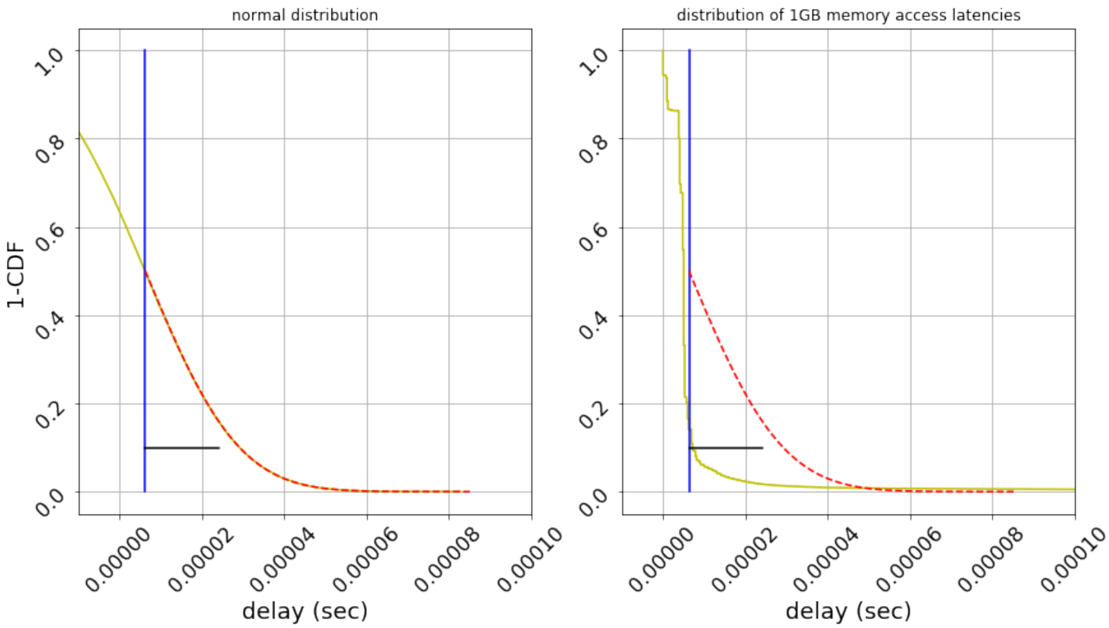
0.1.8 CDF instead of histogram

- Choosing ranges and bin-numbers for histograms can be hard.
- $CDF(a) = Pr(T \leq a)$ . . . .(T=latency)
- Plotting a CDF does not require choosing bins.
- We are interested in larger latencies, so we use instead

$$1 - CDF(a) = 1 - Pr(T \leq a) = Pr(T > a)$$

In [12]:
```
figure(figsize=(14,7))
subplot(121)
grid()
PlotTime(sort(_normal),_mean_t,_std_t,Color=['y','b','k','r'],LS=['-','-','-','--'],Lo
title('normal distribution')
xlabel('delay (sec)',fontsize=18)
xlim([-thr/10,thr])
ylabel('1-CDF',fontsize=18)
tick_params(axis='both', which='major', labelsize=16,rotation=45)
tick_params(axis='both', which='minor', labelsize=12,rotation=45)

#print('%d Memory Blocks of size %d bytes'%(m_list[m_i],n))
#print('\rMemory pokes _mean='+str(_mean[1,m_i])+', Memory _std='+str(_std[1,m_i]))
subplot(122)
grid()
PlotTime(sort(tmp),_mean_t,_std_t,Color=['y','b','k','r'],LS=['-','-','-','--'],LogLog
```

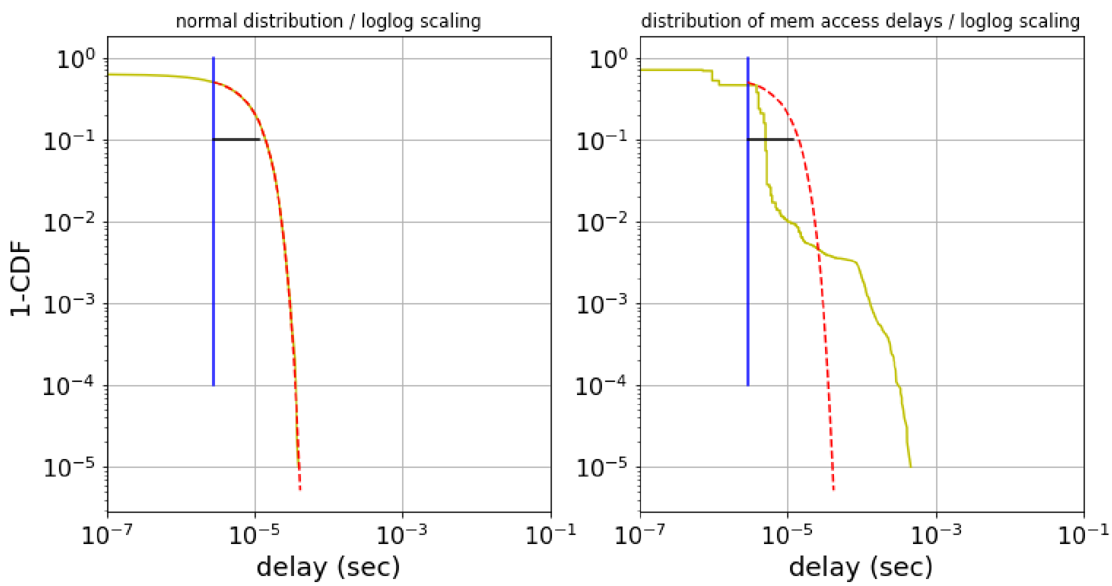
```
title('distribution of 1GB memory access latencies')
xlabel('delay (sec)',fontsize=18)
xlim([-thr/10,thr])
#ylim([0,0.001])
#ylabel('1-CDF',fontsize=18)
tick_params(axis='both', which='major', labelsize=16,rotation=45)
tick_params(axis='both', which='minor', labelsize=12,rotation=45)
```



0.1.9 CDF + loglog plots

```
In [13]: figure(figsize=(12,6))
         subplot(121)
         grid()
         PlotTime(sort(_normal),_mean_t,_std_t,Color=['y','b','k','r'],LS=['-','-','-','--'])
         title('normal distribution / loglog scaling')
         xlabel('delay (sec)',fontsize=18)
         xlim([1e-7,1000*thr])
         ylabel('1-CDF',fontsize=18)
         tick_params(axis='both', which='major', labelsize=16)
         tick_params(axis='both', which='minor', labelsize=12)

         #print('%d Memory Blocks of size %d bytes'%(m_list[m_i],n))
         #print('\rMemory pokes _mean='+str(_mean[1,m_i])+', Memory _std='+str(_std[1,m_i]))
         subplot(122)
         grid()
         PlotTime(sort(tmp),_mean_t,_std_t,Color=['y','b','k','r'],LS=['-','-','-','--'])
         title('distribution of mem access delays / loglog scaling')
```

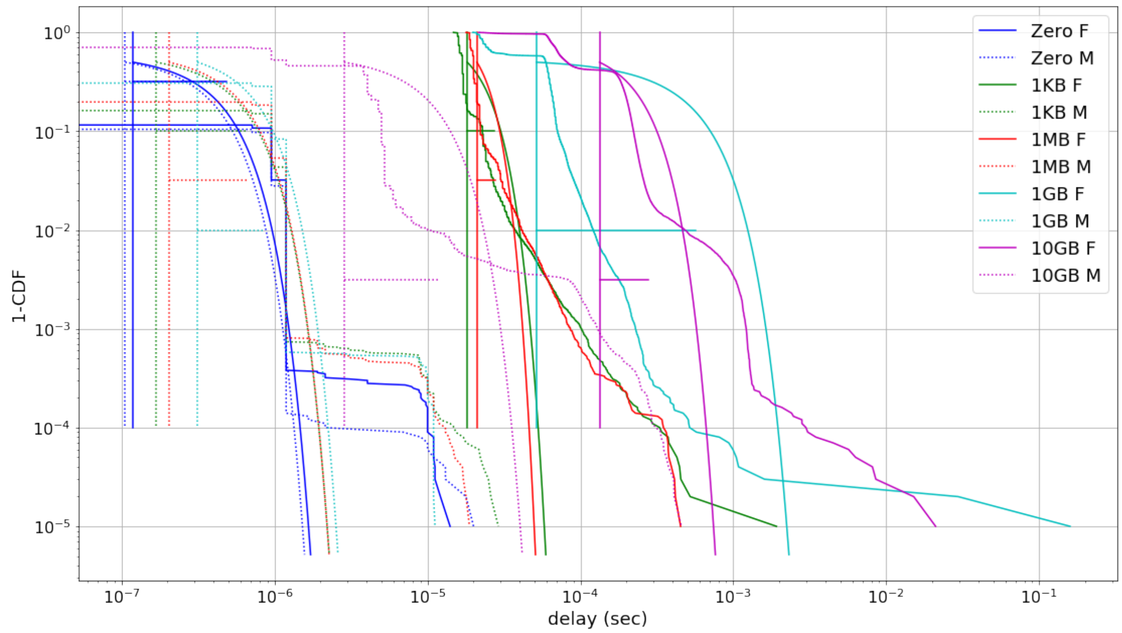
```
xlabel('delay (sec)',fontsize=18)
xlim([1e-7,1000*thr])
#ylabel('1-CDF',fontsize=18)
tick_params(axis='both', which='major', labelsize=16)
tick_params(axis='both', which='minor', labelsize=12)
```



0.1.10 Characterize random access to storage

```
In [14]: pp = PdfPages('MemoryFigure.pdf')
         figure(figsize=(6,4))

         Colors='bgrcmyk'  # The colors for the plot
         LineStyles=['-',':']
         Legends=['F','M']

         fig = matplotlib.pyplot.gcf()
         fig.set_size_inches(18.5,10.5)

         for m_i in range(len(m_list)):
             Color=Colors[m_i % len(Colors)]
             for Type in [0,1]:

                 PlotTime(sort(T[Type,m_i]),_mean[Type,m_i],_std[Type,m_i],\
                      Color=Color,LS=LineStyles[Type],Legend=m_legend[m_i]+' '+Legends[Type],\
                      m_i=m_i)

         grid()
         legend(fontsize=18)
```

```
xlabel('delay (sec)',fontsize=18)
ylabel('1-CDF',fontsize=18)
tick_params(axis='both', which='major', labelsize=16)
tick_params(axis='both', which='minor', labelsize=12)
pp.savefig()
pp.close()
```



0.1.11 Characterize sequential access

- Random access degrades rapidly with the size of the block.
- Sequential access is **much** faster.
- We already saw that writing 10GB to disk sequentially takes 8.9sec, or less than 1 second for a gigbyte.
- Writing a 1TB disk at this rate would take ~1000 seconds or about 16 minutes.

In [15]:
```
import time
Consec=[]
Line='### Consecutive Memory writes:'
print(Line); stat.write(Line+'\n')
n=1000
r=np.array(list(range(n)))
Header="""
|   size (MB) | Average time per byte |
| ---------: | --------------: | """
tee(Header)
for m in [1,1000,1000000,10000000]:
    t1=time.time()
```

```
            A=np.repeat(r,m)
            t2=time.time()
            Consec.append((n,m,float(n*m)/1000000,(t2-t1)/float(n*m)))
            tee("| %6.3f | %4.2g |" % (float(n*m)/1000000,(t2-t1)/float(n*m)))
        A=[];r=[]
        stat.close()
```

```
### Consecutive Memory writes:

|   size (MB) | Average time per byte |
| ---------: | --------------: |
|  0.001 | 2.3e-08 |
|  1.000 | 7.5e-09 |
| 1000.000 | 7.1e-09 |
| 10000.000 | 9.2e-09 |
```

0.1.12 Consecutive Memory writes:

- We are measuring **bandwidth** rather than **latency**:
- We say that it take 8.9sec to write 10GB to SSD, we are **NOT** saying that to write one byte to SSD it take 8.9×10^{-10} second to write a **single** byte.
- This is because many write operations are occuring in parallel.
- Byte-rate for writing large blocks is about (100MB/sec)
- Byte-rate for writing large SSD blocks is about (1GB/sec)
- Comparison:
  - **Memory:** Sequential access: 100M/sec, random access: 10^{-9}sec for 10KB, $10^{-6} - 10^{-3}$ for 10GB
  - **SSD:** Sequential access: 1GB/sec, random access: $10^{-5} - 10^{-3}$sec for 10KB, $10^{-4} - 10^{-1}$ for 10GB

0.2 Collecting System Description

In this section you will find commands that list properties of the hardware on which this notebook is running.

0.2.1 Specify which OS you are using

Uncomment the line corresponding to your OS. Comment all of the rest.

```
In [16]: brand_name = "brand: Macbook"
         #brand_name = "brand: Linux"
         #brand_name = "brand: Windows"
```

0.2.2 For Mac users

The next cell needs to be run only by Mac OS users. If run on other OS platforms, it will throw error.

```
In [17]: if brand_name== "brand: Macbook":
             # To get all available information use !sysctl -a
             os_info = !sysctl kernel.osrelease kernel.osrevision kernel.ostype kernel.osversio
             cpu_info = !sysctl machdep.cpu.brand_string machdep.cpu.cache.L2_associativity mac
             cache_info = !sysctl kern.procname hw.memsize hw.cpufamily hw.activecpu hw.cachel
```

0.2.3 For Linux OS users

```
In [18]: if brand_name == "brand: Linux":
             os_info = !sysctl kernel.ostype kernel.osrelease
             os_version = !lsb_release -r
             memory_size = !cat /proc/meminfo | grep 'MemTotal'
             os_info += os_version + memory_size

             cache_L1i = !lscpu | grep 'L1i'
             cache_L1d = !lscpu | grep 'L1d'
             cache_L2 = !lscpu | grep 'L2'
             cache_L3 = !lscpu | grep 'L3'
             cache_info = cache_L1i + cache_L1d + cache_L2 + cache_L3

             cpu_type = !lscpu | grep 'CPU family'
             cpu_brand = !cat /proc/cpuinfo | grep -m 1 'model name'
             cpu_frequency = !lscpu | grep 'CPU MHz'
             cpu_core_count = !lscpu | grep 'CPU(s)'
             cpu_info = cpu_type + cpu_brand + cpu_frequency + cpu_core_count
```

0.2.4 For Windows users

```
In [19]: if brand_name =="brand: Windows":
             os_release  = !ver
             os_type     = !WMIC CPU get  SystemCreationClassName
             memory      = !WMIC ComputerSystem get TotalPhysicalMemory
             os_info     = os_release + os_type

             cpu_core_count  = !WMIC CPU get NumberOfCores
             cpu_speed       = !WMIC CPU get CurrentClockSpeed
             cpu_model_name  = !WMIC CPU get name
             cpu_info        = cpu_core_count + cpu_speed + cpu_model_name

             l2cachesize = !WMIC CPU get L2CacheSize
             l3cachesize = !WMIC CPU get L3CacheSize
             cache_info  = l2cachesize + l3cachesize
```

```
In [20]: # Print collected information
         description=[brand_name] + os_info + cache_info + cpu_info
```

```
    print("Main Harware Parameters:\n")
    print('\n'.join(description))
```

```
Main Harware Parameters:

brand: Macbook
sysctl: unknown oid 'kernel.osrelease'
kern.procname: sysctl
hw.memsize: 17179869184
hw.cpufamily: 280134364
hw.activecpu: 8
hw.cachelinesize: 64
hw.cpufrequency: 2500000000
hw.l1dcachesize: 32768
hw.l1icachesize: 32768
hw.l2cachesize: 262144
hw.l3cachesize: 6291456
hw.cputype: 7
machdep.cpu.brand_string: Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz
machdep.cpu.cache.L2_associativity: 8
machdep.cpu.cache.linesize: 64
machdep.cpu.cache.size: 256
machdep.cpu.core_count: 4
```

0.2.5 Summary of Macbook Pro hardware parameters

- Intel four cores
- Clock Rate: `2.50GHz` (0.4ns per clock cycle)

```
In [21]: # Writing all necesarry information int oa pickle file.
         import pickle
         with open(home_base+'/memory_report.pkl','wb') as pickle_file:
             pickle.dump({'description':description,
                          'Consec':Consec,
                          'Random_pokes':Random_pokes},
                         pickle_file)
```

0.3 Observations

- making measurements in the wild allows you to measure the performance of your hardware with your software.
- Measuring in the wild you discover unexpected glitches:
  - timer resolution is 1μsec
  - once every ~10,000 of a zero-time poke there is a 10^{-5}$ delay. Maybe a context switch?
- Latencies typically have long tails - Use loglog graphs.
- Memory latency varies from 10^{-9}sec to 10^{-6}sec depending on access pattern.

- SSD latency for random access varies from 10^{-5}sec to 10^{-1}sec.
- When reading or writing large blocks, we care about **throughput** or **byte-rate** not **latency**
- Typical throughputs: **Memory:** 100MB/sec **SSD:** 1GB/sec **Disk:** (fill in)

0.3.1 Impact on Big Data Analytics

- Clock rate is stuck at around 3GHz, and is likely to be stuck there for the forseeable future.
- **Faster** computers / disks / networks are **expensive**
- **focus on data access:** The main bottleneck on big data computation is moving data around, **NOT** calculation.
- The cost-effective solution is often a cluster of many cheap computers, each with many cores and break up the data so that each computer has a small fraction of the data.
- Data-Centers and the "Cloud"
- I invite you to use this notebook on your computer to get a better understanding of its memory access latency.
- If you are interest in way to make more accurate measurements of latency, try notebook 3.
- See you next time.

0.4 Clean-Up

This notebook generates large logs that can be deleted.
A summary of the results is placed in the file `memory_report.pkl`

```
In [22]: %cd $home_base

         !rm -rf logs
```

```
/Users/yoavfreund/academic.papers/Courses/BigDataAnalytics/BigData_spring2016/CSE255-DSE230-20
```