PYTHON PERFORMANCE TUNING AND OPTIMIZATION

MODULE 11

1

APPLICATION PERFORMANCE

2

What Is Performance?

> What are some different aspects of performance?
  - Memory footprint
  - Startup time
  - Scalability
  - Responsiveness
  - Throughput
  - Efficiency
  - Utilization
  - Latency
  - Capacity
  - Degradation

Memory Footprint

> The memory footprint is the amount of memory used by your application and your JVM on a system. Considerations include the following:
  - Does the application run on a dedicated system?
  - Does the application share the machine with other applications and/or JVMs?
  - As memory footprint grows, does it affect Virtual Memory?
    - Accessing data in virtual memory is much slower than accessing data in RAM.

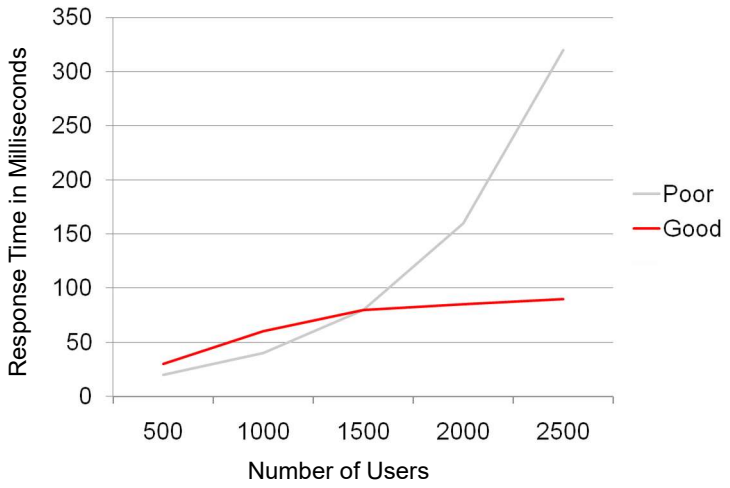
Startup Time

> *Startup time*: The time taken for an application to start
  - For client applications, this can be very important.
  - For server applications, it is less important.
  - Time 'til Performance

Scalability

> *Scalability*: How well an application performs as the load on it increases
  - An application may perform well in development but poorly after it is deployed.
  - If an application's response times grow *exponentially* when under load, its scalability is poor.
  - If an application's response times grow *linearly* when under load, its scalability is good.
  - Scalability can be measured at a number of different levels in a complex system.

Application Scalability



Responsiveness

> *Responsiveness*: How quickly an application or system responds with a requested piece of data
  - Examples
    - How quickly a desktop UI responds to an event
    - How fast a website returns a page
    - How fast a database query is returned
  - Large pause times are not acceptable.
  - The focus is on responding in short periods of time.

Throughput

- *Throughput*: Focusing on maximizing the amount of work by an application in a specific period of time
  - Examples
    - The number of instructions per second that can be executed
    - The number of jobs that a batch program can complete in an hour
    - The number of database queries that can be completed in an hour
  - High pause times are acceptable.
  - The focus is on how much work can be done over longer periods of time.

Low Latency

Data passing	Latency	Light over a fibre	Throughput on at a time
Method call	Inlined: 0 Real call: 50 ns.	10 meters	20,000,000/sec
Shared memory	200 ns	40 meters	5,000,000/sec
SYSV Shared memory	2 µs	400 meters	500,000/sec
Low latency network	8 µs	1.6 km	125,000/sec
Typical LAN network	30 µs	6 km	30,000/sec
Typical data grid system	800 µs	160 km	1,250/sec
60 Hz power flickers	8 ms	1600 km	120/sec
4G request latency in UK	55 ms	11,000 km	18/sec

What is an ultra low GC?

> Another reason to use ultra low garbage rates is your caches are not being filled with garbage and they work much more efficiently and consistently.
> If you have a web server which is producing 300 MB/s of garbage, this means less than 5% of the time you will be pausing for a GC.
> However, it does mean that you could be filling a 32 KB L1 CPU in around 0.1 milli-seconds. Your L2 cache fills with garbage every milli-second.

So, without a GC pause, no more pauses?

> A high percentage of pauses are not from the GC.
> The biggest ones are, but take away GC pauses and still see
> IO delays. These can be larger than GC pauses.
  – Network delays
  – Waiting for databases
  – Disk reads/writes
> OS interrupts
  – It is not uncommon for your OS to stop your process for 5 ms or more.
> Lock contention pauses.

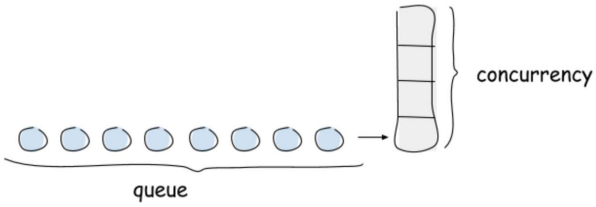
Optimizing for latency instead of throughput

> Average latency is largely an inverse of your throughput and no better.

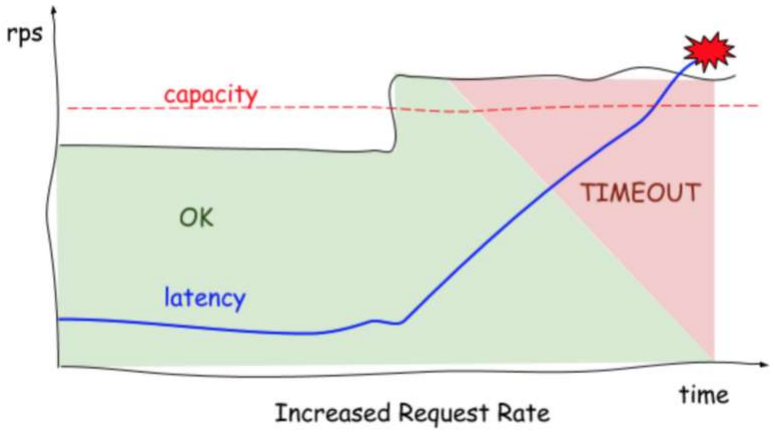
> Using standard deviation for average latency is misleading at best as the distribution for latencies are not a normal distribution or anything like it.

> From Little's Law

– Average latency = concurrency / throughput



Optimizing for latency instead of throughput



Capacity

> The capacity is the amount of work parallelism a system possesses
  – The number of units of work (e.g., transactions) that can be simultaneously ongoing in the system.
> Capacity is obviously related to throughput, and we should expect that as the concurrent load on a system increases, throughput (and latency) will be affected.
> For this reason, capacity is usually quoted as the processing available at a given value of latency or throughput.

Utilization

> One of the most common performance analysis tasks is to achieve efficient use of a system's resources.
> Ideally, CPUs should be used for handling units of work, rather than being idle (or spending time handling OS or other housekeeping tasks).
> Depending on the workload, there can be a huge difference between the utilization levels of different resources.
> For example, a computation-intensive workload (such as graphics processing or encryption) may be running at close to 100% CPU but only be using a small percentage of available memory.

Efficiency

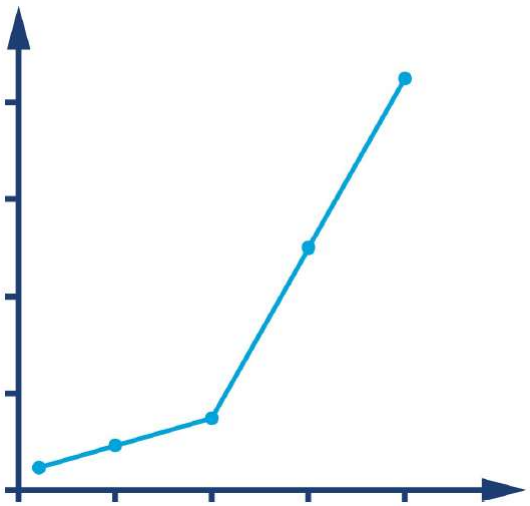
> Dividing the throughput of a system by the utilized resources gives a measure of the overall efficiency of the system.
> Intuitively, this makes sense, as requiring ***more resources*** to produce ***the same throughput*** is one useful definition of being ***less efficient***.
> It is also possible, when one is dealing with larger systems, to use a form of ***cost accounting to measure efficiency***.
> If solution **A** has a total dollar cost of ownership (TCO) twice that of solution **B** for the same throughput then it is, clearly, ***half as efficient***.

Degradation

> If we increase the load on a system, either by increasing the number of requests (or clients) or by increasing the speed requests arrive at, then we may see a change in the observed latency and/or throughput.
> Note that this change is dependent on utilization.
> If the system is underutilized, then there should be some slack before observables change, but if resources are fully utilized then we would expect to see throughput stop increasing, or latency increase.
> These changes are usually called the degradation of the system under additional load.

Reading Performance Graphs

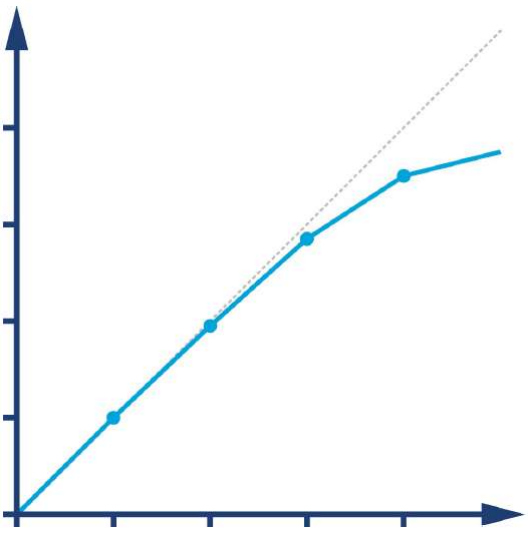
> Latency under increasing load



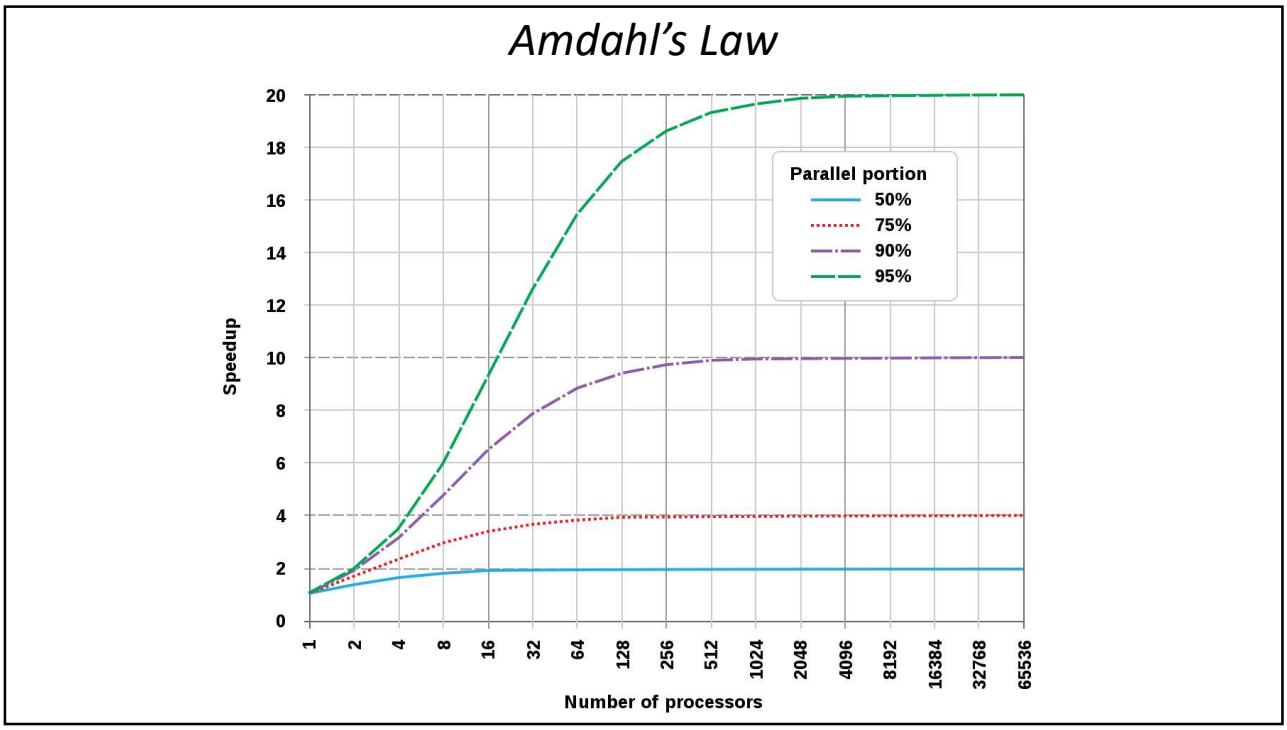
19

Reading Performance Graphs

> Latency under increasing load: *Near-linear scaling*

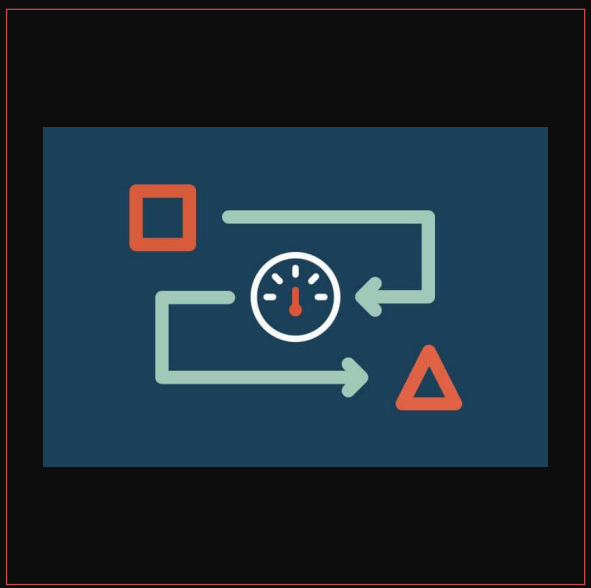


20



21

PERFORMANCE METHODOLOGY



22

Performance Methodology

> Performance methodology
  – Monitor
  – Profile
  – Tune
> Definitions
> Incorporating performance methodology into development

Performance Monitoring

> *Definition*: An act of *non-intrusively* collecting or observing performance data from an operating or running application.
  – For troubleshooting
    • Identify problems
    • Determine problem characteristics
  – For development
    • Test application for meeting requirements
    • Responsiveness or throughput
  – Operating system and JVM tools

Performance Profiling

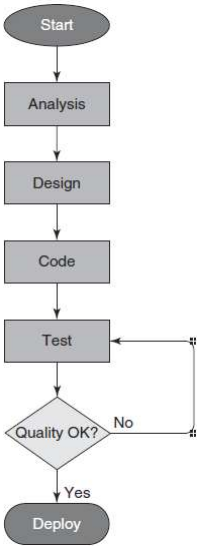
> *Definition*: An act of collecting or observing performance data from an operating or running application.
  - Usually more intrusive than monitoring
  - Usually a narrower focus than monitoring
  - Commonly done in qualification, testing, or development environments.

Performance Tuning

> *Definition*: An act of changing tunables, source code and/or configuration attributes for the purposes of improving application responsiveness and/or application throughput.
  - Relies on performance requirements
  - Requires monitoring and/or profiling activities
  - Performance goals
    - Responsiveness
    - Throughput

Forces

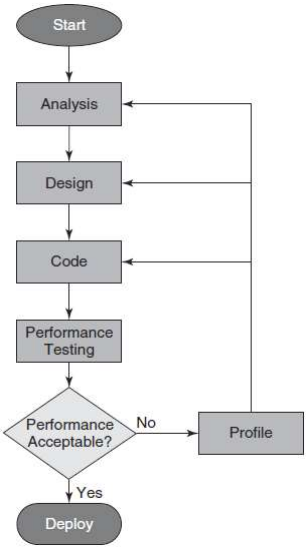
> The traditional software development process consists of four major phases: analysis, design, coding, and testing



27

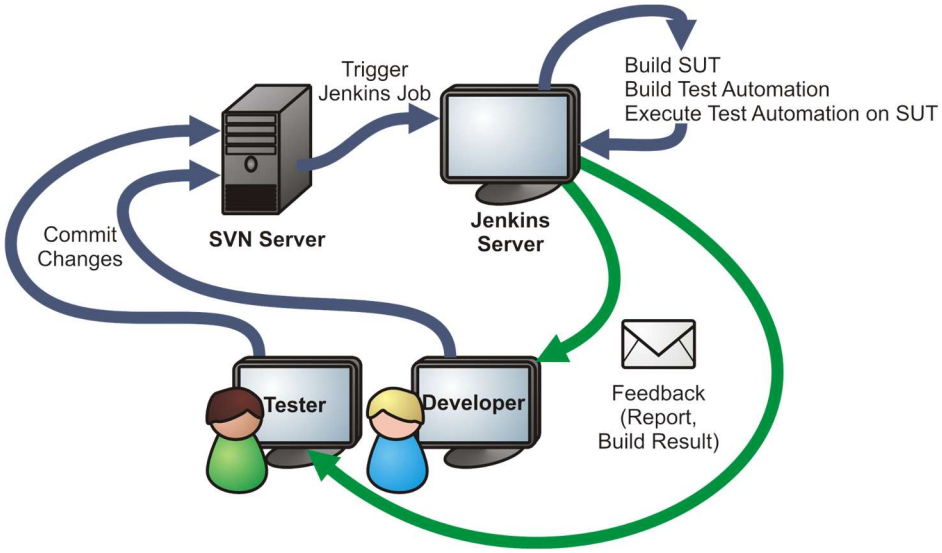
Wilson & Kesselman's Performance Process

> Integrate performance testing into automated build processes
> Performance regression
> Track performance at each coding change committed to the source code base



28

Continuous Performance Validation



Systemic-Quality-Driven

> Systemic qualities are requirements on the system that are non-functional or related to the quality of service.

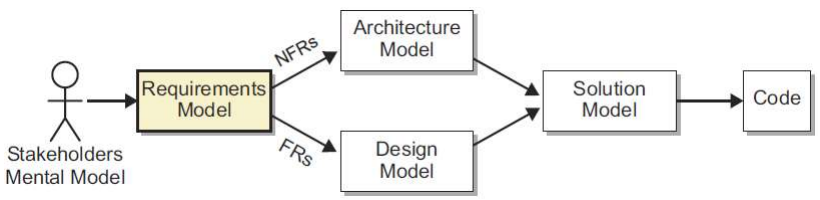
> Examples include:

– Performance – Such as responsiveness and latency

– Reliability – The mitigation of component failure

– Scalability – The ability to support additional load, such as more users

> Systemic qualities drive the architecture of the software.



NFRs

> What is the expected throughput of the application?

> What is the expected latency between a stimulus and a response to that stimulus?

> How many concurrent users or concurrent tasks shall the application support?

> What is the accepted throughput and latency at the maximum number of concurrent users or concurrent tasks?

> What is the maximum worst case latency?

> What is the frequency of garbage collection induced latencies that will be tolerated?

Two Approaches: Top Down and Bottom Up

> There are two commonly accepted approaches to performance analysis:

1. **Top down**
   - Focuses on the top level of the application
   - Drills down the software stack looking for problem areas and optimization opportunities
2. **Bottom up**
   - Begins at the lowest level of the software stack
   - At the CPU level looking at statistics such as CPU cache misses, inefficient use of CPU instructions
   - What constructs or idioms are used by the application

Top-Down Approach

- The most common approach utilized for performance tuning
- Commonly used when you have the ability to change the code at the highest level of the application software stack
- You begin by monitoring the application of interest under a load at which a stakeholder observes a performance issue
- An application is continuously monitored and as a result of a change in the **application's configuration**/**typical load** the application experiences a degradation in performance
- ***Performance*** and ***scalability*** requirements for the application change and the application in its current state cannot meet those new requirements.

Top-Down Approach

- Monitoring the application while it is running under a load of particular interest is the first step in a top-down approach
- Monitoring activity may include
  - Observing operating system level statistics,
  - Python interpreter statistics,
  - Application performance instrumentation statistics.

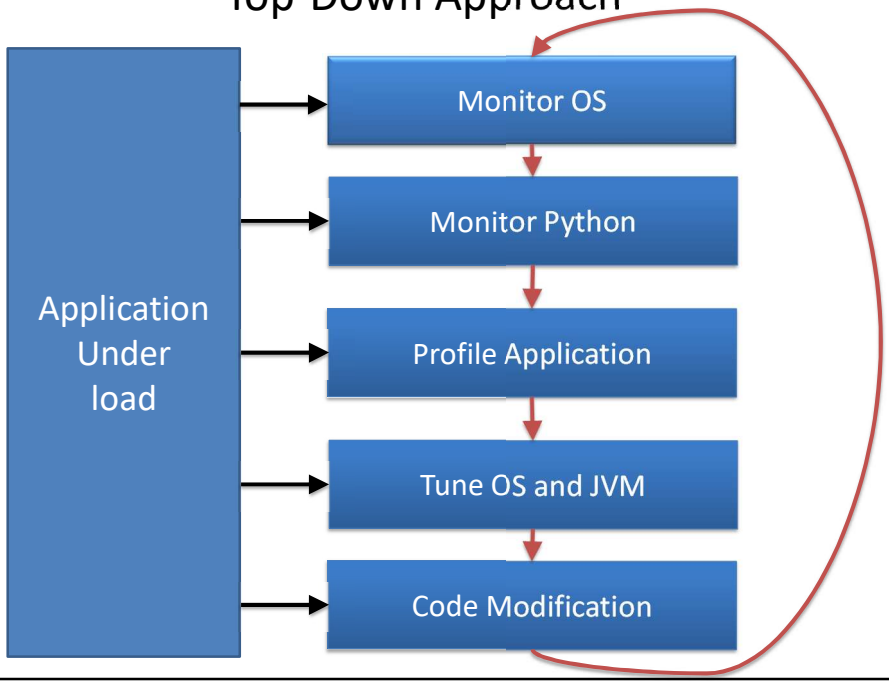
Top-Down Approach

- Tools and Techniques
  - Monitoring tools: Grafana, Prometheus.
  - Profilers: py-spy, cProfile, line_profiler.
- Metrics to Monitor
  - Response time, throughput, memory usage, CPU utilization.

Top-Down Approach

- Code Profiling
  - Use profilers to understand where time is spent.
  - Example: cProfile output for function-level performance.
- Log Analysis
  - Review application logs for delays.
- Database Query Analysis
  - Check query execution plans and latencies.

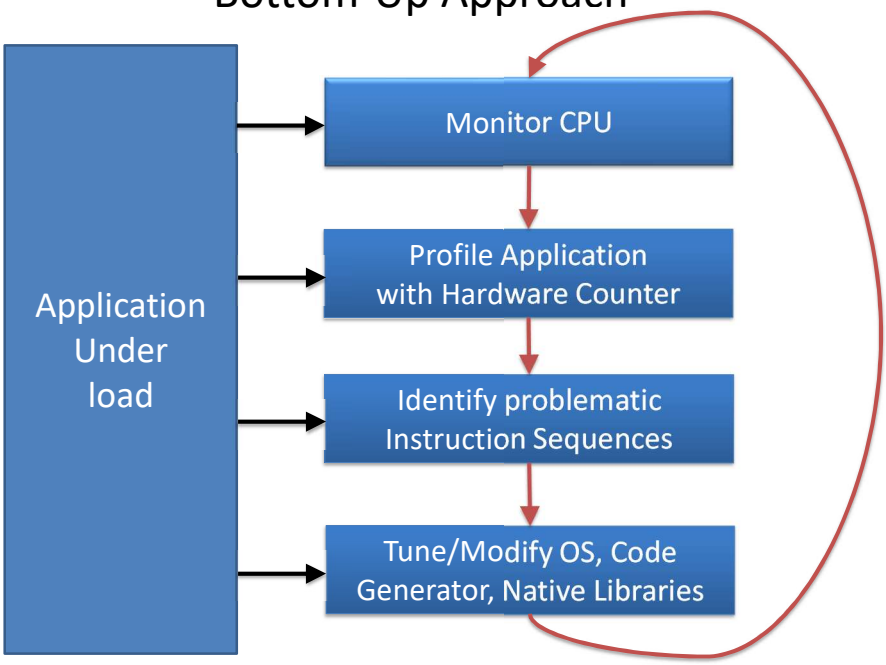
Top-Down Approach



Bottom-Up Approach

> The bottom-up approach is most used by performance specialists to ***improve*** the performance of an application on ***one platform*** relative to ***another***
  – where differences exists in the underlying
    • CPU,
    • CPU architecture,
    • Number of CPUs.

Bottom-Up Approach



Choosing Right Platform & Evaluating a System

> ***Choosing the Right CPU Architecture***

ARM, MIPS, x86, RISC-V

ASIC, SoC, ISA

GPU, TPU

> ***Evaluating a System's Performance***

OPERATION SYSTEM PERFORMANCE MONITORING

41

Objectives

After completing this lesson, you should be able to:

- Monitor CPU usage
- Monitor network I/O
- Monitor disk I/O
- Monitor virtual memory usage
- Monitor processes including lock contention

42

Why Are We Monitoring?

> Get a sense of what the problem is
> Identify the poor performance symptoms
> Based on the symptoms, diagnose the problem

Modern Hardware

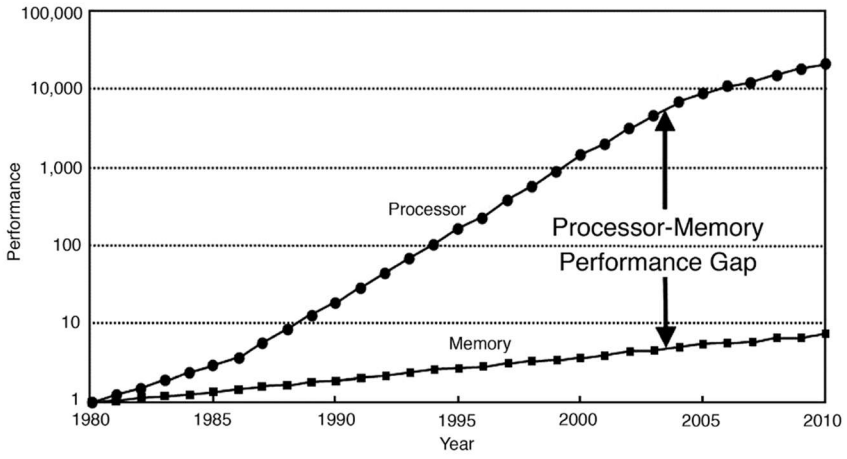
> Memory
> Cache
> Translation Lookaside Buffer
> Branch Prediction and Speculative Execution
> Hardware Memory Models

Memory

- Moore's Law: *The number of transistors on a mass-produced chip roughly doubles every 18 months.*
- The exponentially increasing number of transistors was initially used for faster and faster clock speed
- Faster clock speed means more instructions completed per second
- Faster chips require a faster stream of data to act upon
- Over time main memory could not keep up with the demands of the processor core for fresh data

Memory

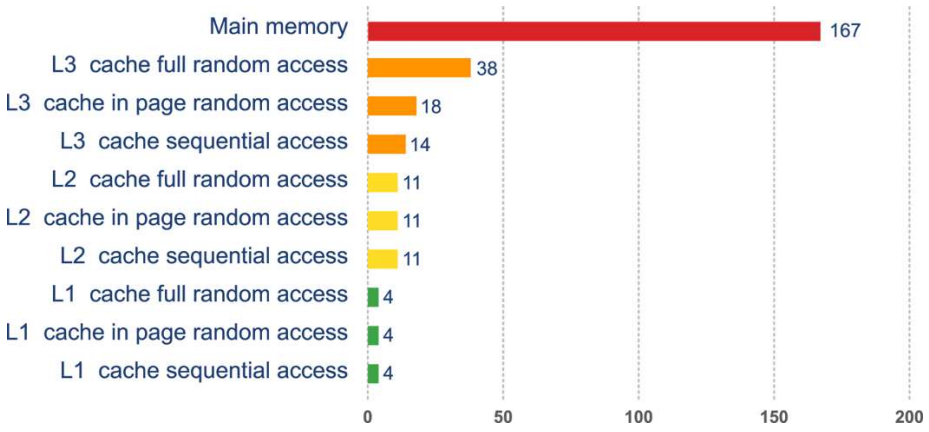
- if the CPU is waiting for data, then faster cycles don't help, as CPU will just have to idle until the required data arrives.



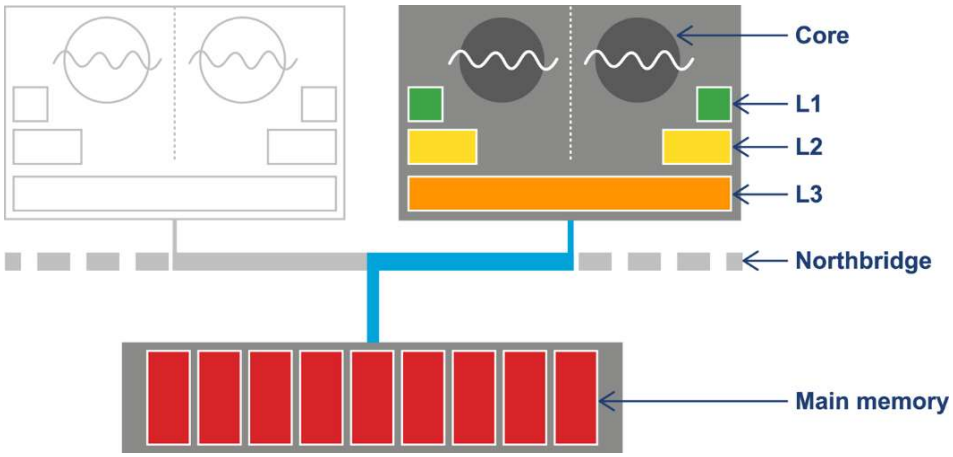
Memory Caches

> To solve this problem, CPU caches were introduced.
> These are memory areas on the CPU that are slower than CPU registers, but faster than main memory.
> The idea is for the CPU to fill the cache with copies of often-accessed memory locations rather than constantly having to re-address main memory

Memory Caches



Memory Caches



Memory Caches

> Although the addition of a caching architecture hugely improves processor throughput, it introduces a new set of problems.
> These problems include determining how memory is fetched into and written back from the cache.
> The solutions to this problem are usually referred to as ***cache consistency protocols***.

Cache Consistency Protocols

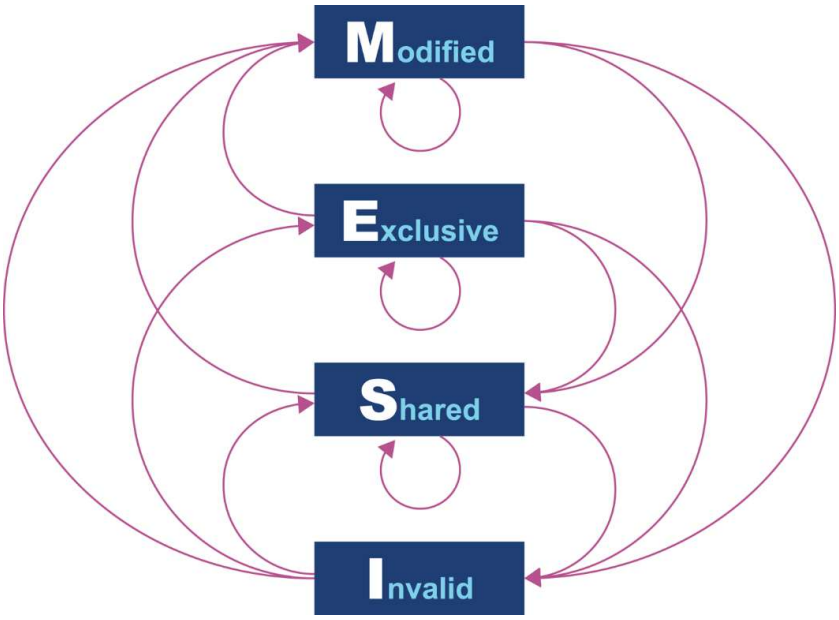
- At the lowest level, a protocol called **MESI** (and its variants) is commonly found on a wide range of processors.
- It defines four states for any line in a cache.
- Each line (usually 64 bytes) is either:
  - **Modified** (but not yet flushed to main memory)
  - **E**xclusive (present only in this cache, but does match main memory)
  - **S**hared (may also be present in other caches; matches main memory)
  - **I**nvalid (may not be used; will be dropped as soon as practical)

Cache Consistency Protocols

- *MESI allowable states between processors*

	M	E	S	I
M	-	-	-	Y
E	-	-	-	Y
S	-	-	Y	Y
I	Y	Y	Y	Y

Cache Consistency Protocols



Translation Lookaside Buffer

- One very important use is in a different sort of cache, the Translation Lookaside Buffer (TLB).
- TLB acts as a cache for the page tables that map virtual memory addresses to physical addresses
  - Greatly speeds up a very frequent operation
  - Access to the physical address underlying a virtual address
- Without the TLB, all virtual address lookups would take 16 cycles, even if the page table was held in the L1 cache

Branch Prediction and Speculative Execution

> One of the advanced processor tricks that appear on modern processors is branch prediction.

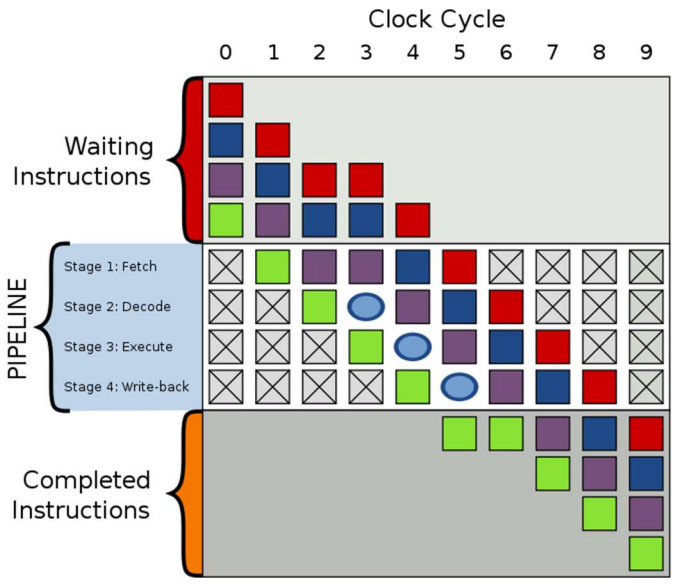
> This is used to prevent the processor from having to wait to evaluate a value needed for a conditional branch

> Modern processors have multistage instruction pipelines

> This means that the execution of a single CPU cycle is broken down into a number of separate stages.

> There can be several instructions in flight (at different stages of execution) at once.

Branch Prediction and Speculative Execution



i7: 4 Instructions per cycle, 16 stage pipeline

Hardware Memory Models

> The core question about memory that must be answered in a multicore system is

"How can multiple different CPUs access the same memory location consistently?"

Hardware Memory Models

> javac, JIT compiler, and CPU are all allowed to make changes to the order in which code executes.
> This is subject to the provision that any changes don't affect the outcome as observed by the current thread
> For example, let's suppose we have a piece of code like this:

```
myInt = otherInt;
intChanged = true;
```

> There is no code between the two assignments, so the executing thread doesn't need to care about what order they happen in, and thus the environment is at liberty to change the order of instructions?

Hardware Memory Models

	ARMv7	POWER	SPARC	x86	AMD64	zSeries
Loads moved after loads	Y	Y	-	-	-	-
Loads moved after stores	Y	Y	-	-	-	-
Stores moved after stores	Y	Y	-	-	-	-
Stores moved after loads	Y	Y	Y	Y	Y	Y
Atomic moved with loads	Y	Y	-	-	-	-
Atomic moved with stores	Y	Y	-	-	-	-
Incoherent instructions	Y	Y	Y	Y	-	Y

Operating Systems

> The point of an operating system is to control access to resources that must be shared between multiple executing processes.
> All resources are finite, and all processes are greedy
> The need for a central system to arbitrate and meter access is essential.
> Among these scarce resources, the two most important are usually memory and CPU time.
> Virtual addressing via the memory management unit (MMU) and its page tables is the key feature that enables access control of memory, and prevents one process from damaging the memory areas owned by another.

The Scheduler

> Access to the CPU is controlled by the process scheduler.
> It uses a queue known as the *run queue* as a waiting area for threads or processes that are eligible to run but which must wait their turn for the CPU.
> On a modern system there are effectively always more threads/processes that want to run than can, and so this CPU contention requires a mechanism to resolve it.
> The job of the scheduler is to respond to interrupts, and to manage access to the CPU cores.

Context Switches

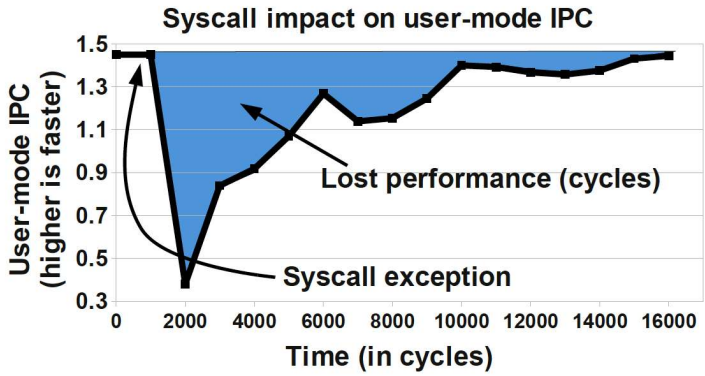
> A *context switch* is the process by which the OS scheduler removes a currently running thread or task and replaces it with one that is waiting.
> There are several different types of context switch, but broadly speaking, they all involve swapping the executing instructions and the stack state of the thread.

Context Switches

> A context switch can be a costly operation, whether between user threads or from user mode into kernel mode
> The latter case is particularly important, because a user thread may need to swap into kernel mode in order to perform some function partway through its time slice.
> However, this switch will force instruction and other caches to be emptied, as the memory areas accessed by the user space code will not normally have anything in common with the kernel.

Context Switches

> A context switch into kernel mode will invalidate the TLBs and potentially other caches.
> When the call returns, these caches will have to be refilled, and so the effect of a kernel mode switch persists even after control has returned to user space.

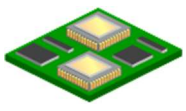


A Simple System Model

> A simple model for describing basic sources of possible performance problems
> The model is expressed in terms of operating system observables of fundamental subsystems and can be directly related back to the outputs of standard Unix command-line tools
> The model is based on a simple conception of a Java application running on a Unix or Unix-like operating system.

Monitoring CPU Usage Overview

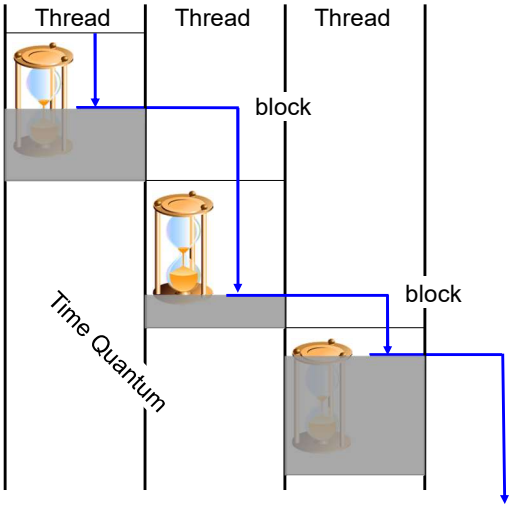
> Rationale for monitoring CPU usage
  – Get big picture view of CPU demand
  – Get per process measurement of CPU utilization
> Measurements of CPU usage
  – User (usr) time
  – System (sys) time
  – Idle time
  – Voluntary context switching (VCX)
  – Involuntary context switching (ICX)



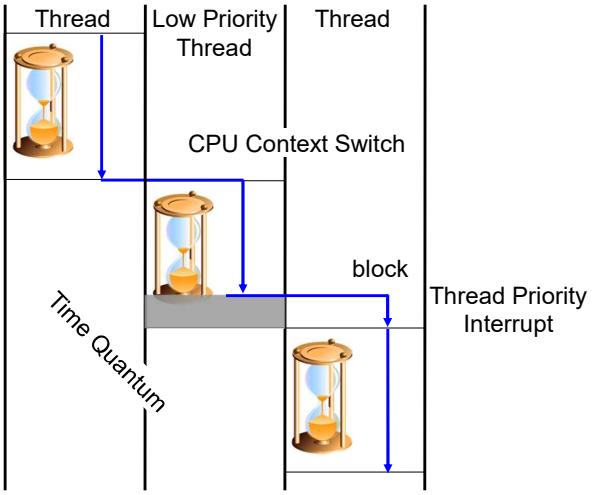
CPU Monitoring Performance Indicators

> The kind of CPU statistics that may warrant further review:
  - High sys/kernel CPU time
  - Idle CPU time

Voluntary Context Switching (VCX)



Involuntary Context Switching (ICX)



Tools For Monitoring CPU Usage

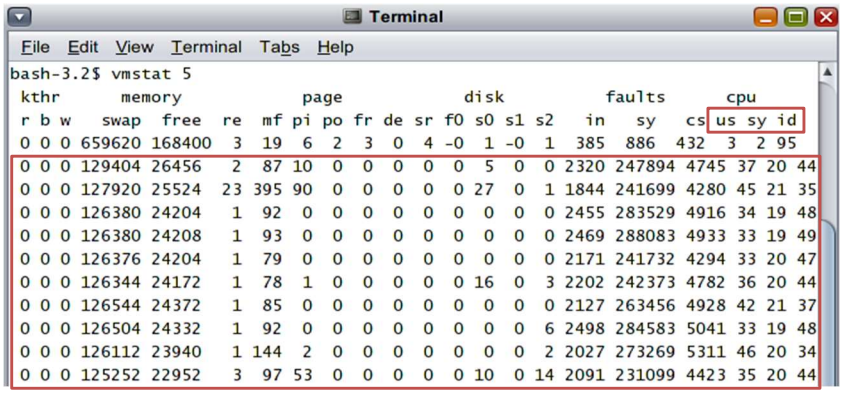
- Tools to monitor CPU utilization
  - `vmstat`
  - `mpstat`
  - `prstat`
  - `top`
  - Task Manager (Windows)
  - Performance Monitor (Windows)
  - Windows Resource Manager (Windows Server)
  - Gnome System Monitor (Linux)
  - `DTrace` (Linux)

Utilizing the CPU

- A key metric for application performance is CPU utilization.
- CPU cycles are quite often the most critical resource needed by an application, and so efficient use of them is essential for good performance.
- Applications should be aiming for as close to 100% usage as possible during periods of high load
- Two basic tools that every performance engineer should be aware of are **`vmstat`** and **`iostat`**.

CPU Usage: `vmstat`

```
$ vmstat 5
```



vmstat

1. The first two columns show the number of runnable (r) and blocked (b) processes.
2. In the memory section, the amount of swapped and free memory is shown, followed by the memory used as buffer and as cache.
3. The swap section shows the memory swapped in from and out to disk (**`si`** and **`so`**).
4. The block in and block out counts (**`bi`** and **`bo`**) show the number of 512-byte blocks that have been received from and sent to a block (I/O) device.
5. The number of interrupts (**`in`**) and the number of context switches per second (**`cs`**) are displayed.

vmstat

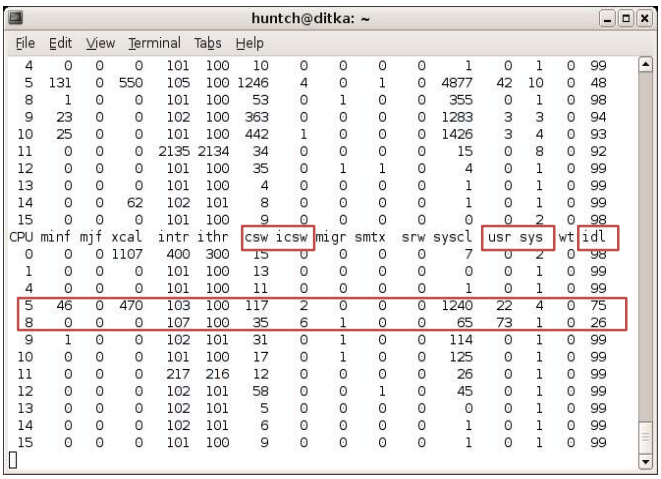
6. The CPU section contains a number of directly relevant metrics, expressed as percentages of CPU time. In order, they are user time (us), kernel time (sy, for “system time”), idle time (id), waiting time (wa), and “stolen time” (st, for virtual machines).

CPU Usage

- if we observe that the CPU utilization is not approaching 100% user time, then the next obvious question is, “Why not?”
- What is causing the program to fail to achieve that?
- Are involuntary context switches caused by locks the problem?
- Is it due to blocking caused by I/O contention?

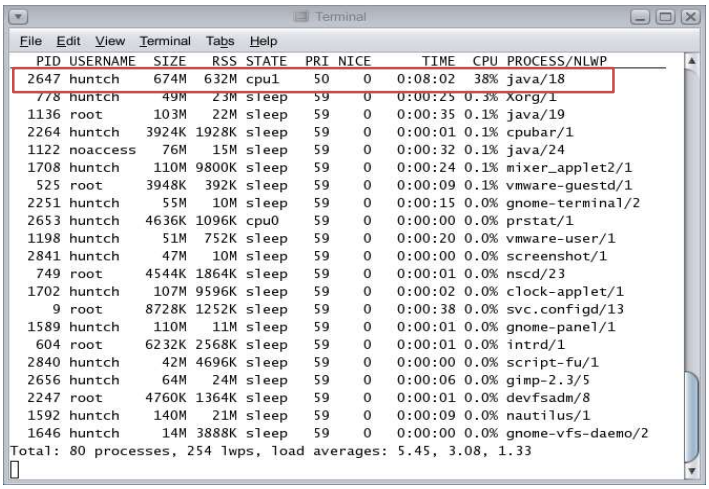
CPU Usage: `mpstat`

- **`$ mpstat 5`**

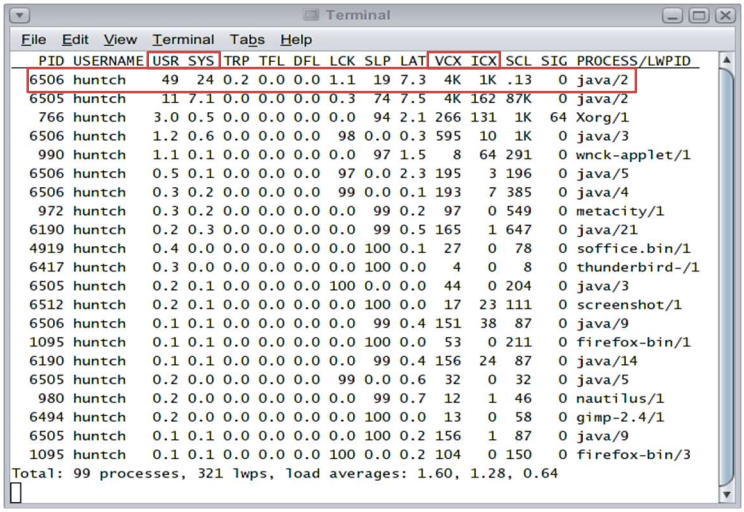


CPU Usage: `prstat`

```
$ prstat
```



CPU Usage: `prstat -m`

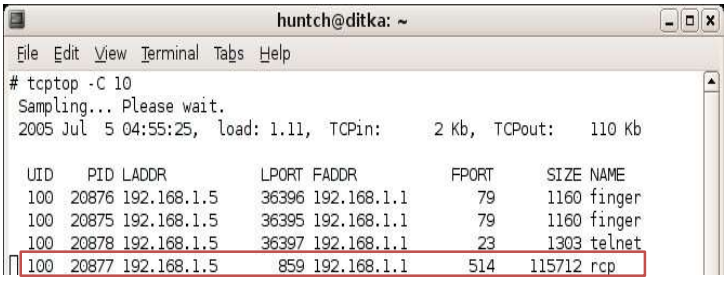


Monitoring Network I/O Overview

- Data of interest
  - Network utilization in terms of Transaction Control Protocol (TCP) statistics and established connections
- Tools to monitor network I/O
  - **`netstat`** (Linux)
  - Performance Monitor (Windows)
  - **`DTrace`** (Linux)
  - **`tcptop`** (`DTrace` Toolkit)

Network I/O: Using `tcptop`

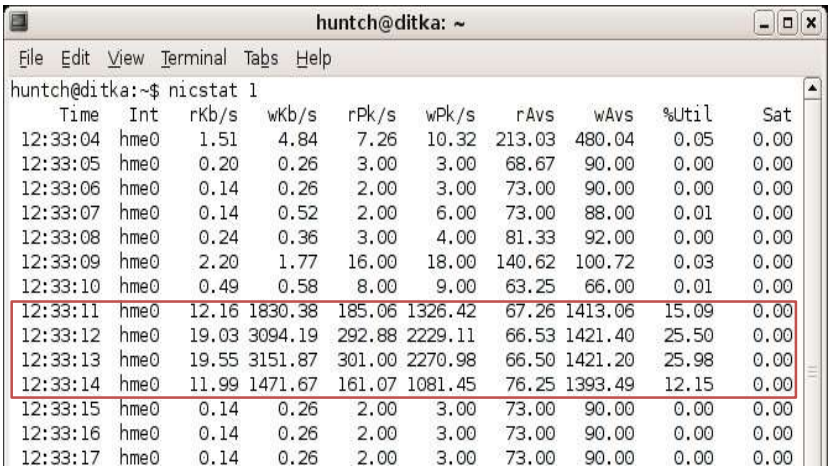
- **`tcptop`** can show per process TCP statistics:



The screen capture shows **`rcp`** generating 115 kb of traffic.

Network I/O: Using **nicstat**

> **nicstat 1**



Monitoring Disk I/O Overview

- Data of interest
  - Number of disk accesses
  - Latency and average latencies
- Tools to monitor disk I/O
  - **iostat** (Linux)
  - **iotop** (Linux)
  - **pidstat** (Linux)
  - Performance Monitor (Windows)
  - **DTrace**
- Disk caches

Disk I/O: `iotop`

> **`iotop`** reporting at a 5-second interval

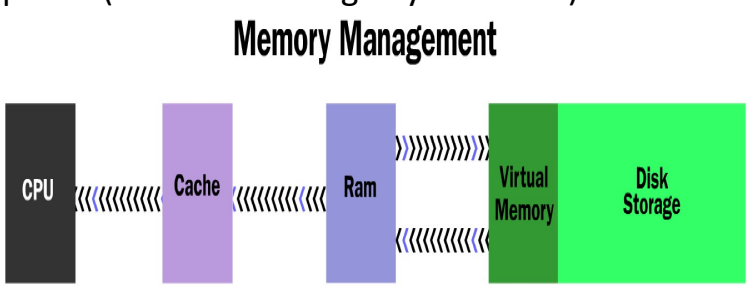
> **`DISKTIME`** reported in microseconds

Monitoring Virtual Memory: Overview

> Observe paging to identify swapping
  - Pages in (pi)
  - Pages out (po)
  - Scan rate (sr)

> Fixing the swapping problem

> Why is swapping bad for a Java application?
  - Want to explain? (Without looking at your notes)

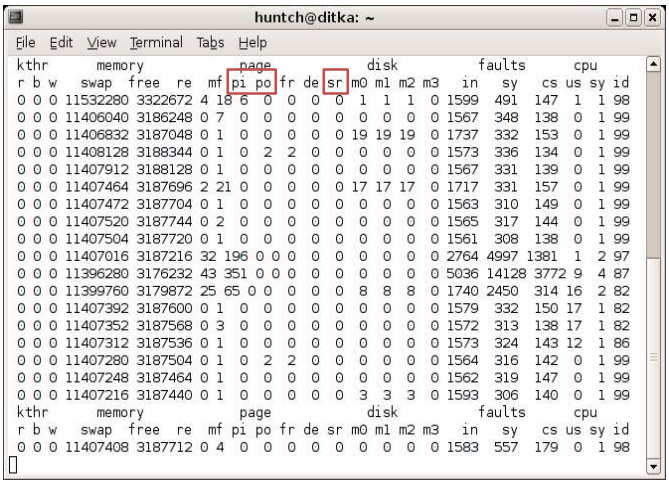


Virtual Memory Tools

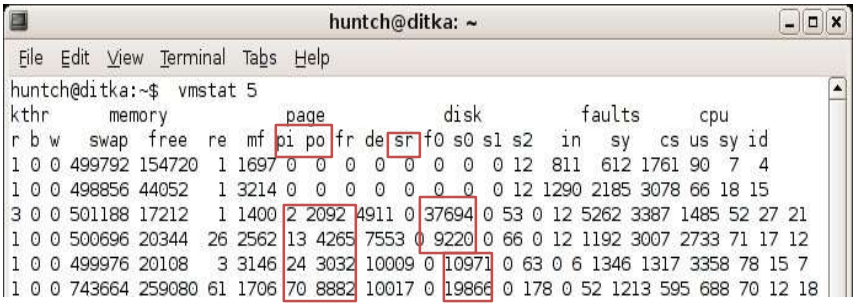
> Tools to monitor memory paging and usage
  - **`vmstat`** (Linux)
  - **`top`**
  - Performance Monitor (Windows)
  - **`DTrace`** (Linux)

Virtual Memory: `vmstat`

```
$ vmstat 5
```



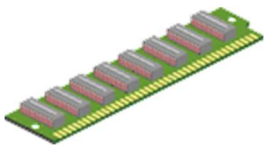
Virtual Memory: Swapping Example



- Data of interest
  - `pi` – pages in; `po` – pages out; `sr` – page scan rate
  - Watch for high scan rate (see rows 3 to 6), or increasing trend. Low scan rate is ok if they occur infrequently.

Virtual Memory: Fixing Swapping Problem

- Add physical memory.
- Reduce number of applications running on the machine.
- Anyone, or any combination of the above will help.



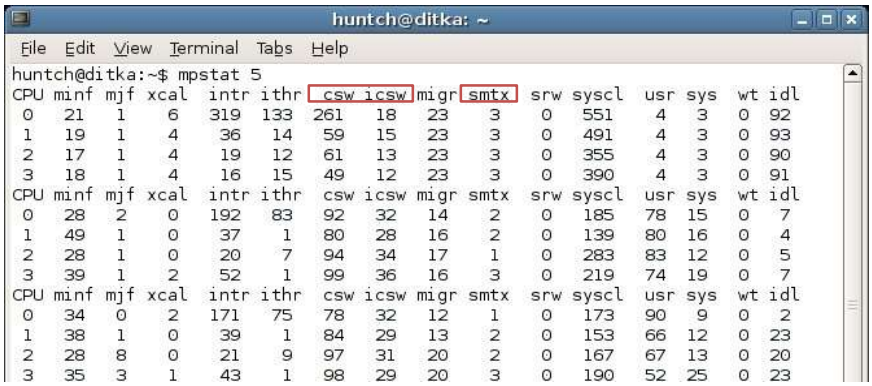
Monitoring Processes Overview

- Data of interest
  - Footprint size
  - Number of threads and thread state
  - CPU usage
  - Runtime stack
  - Context switches
  - Lock contention

Process Monitoring Tools

- **`ps`** (Linux)
- **`vmstat`** (Linux)
- **`mpstat`** (Linux)
- **`pidstat`** (Linux)
- Performance Monitor (Windows)
- **`top`**
- **`DTrace`** (Linux)

Processes: `mpstat`



Monitoring the Kernel

- Data of interest
  - Kernel CPU utilization, locks, system calls, interrupts, migrations, run queue depth
- Tools to monitor the kernel
  - `vmstat` (Linux)
  - `mpstat` (Linux)
  - Performance Monitor (Windows)
  - `DTrace` (Linux)

Kernel: `vmstat`

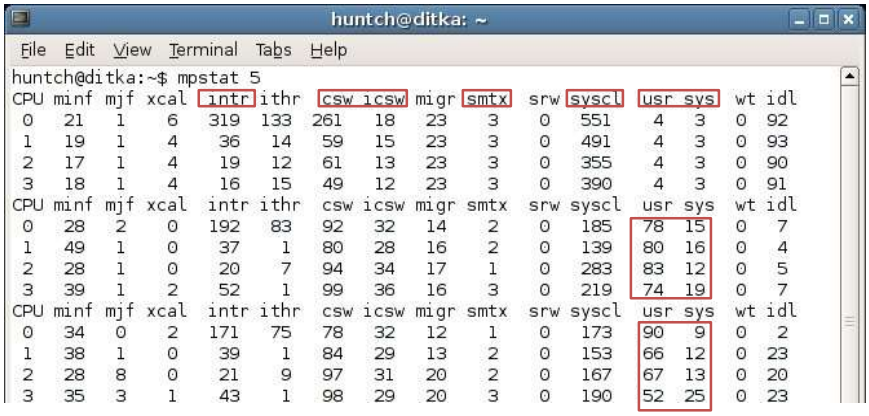
```
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 4  0      0 959476 340784 1387176    0    0     0     0 1030  8977 63 35  1  0
 3  0      0 959444 340784 1387176    0    0     0     0 1014  7981 62 36  2  0
 6  0      0 959460 340784 1387176    0    0     0    16 1019  9380 63 36  1  0
 1  0      0 958820 340784 1387176    0    0     0     0 1036  9157 63 35  2  0
 4  0      0 958500 340784 1387176    0    0     0    29 1012  8582 62 37  1  0
```

- The “us” column shows the percentage of user CPU utilization.
- The “sy” column shows the percentage of kernel or system CPU utilization.
- The “id” column shows the percentage of idle or available CPU.
- The sum of the “us” column and “sy” column should be equal to 100 minus the value in the “id” column, that is, 100 – (“id” column value).

Kernel: **`vmstat`**

```
kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr f0 s0 s1 s2   in   sy   cs us sy id
0 0 0 672604 141500 10  40 36  6 10  0 20  0  3  0  2  425 1043  491  4  3 93
1 1 0 888460 632992  7  32 97  0  0  0  0  0 21  0 12  462 1099  429 32 19 49
0 1 0 887848 631772  4  35 128 0  0  0  0  0 30  0 13  325  575  314 38 13 49
0 1 0 887592 630844  6  26 79  0  0  0  0  0 40  0 11  324  501  287 36 10 54
1 0 0 887304 630160  5  33 112 0  0  0  0  0 50  0 16  369  899  367 37 11 52
0 1 0 886920 629092  4  30 101 0  0  0  0  0 26  0 18  354  707  260 39 14 46
```

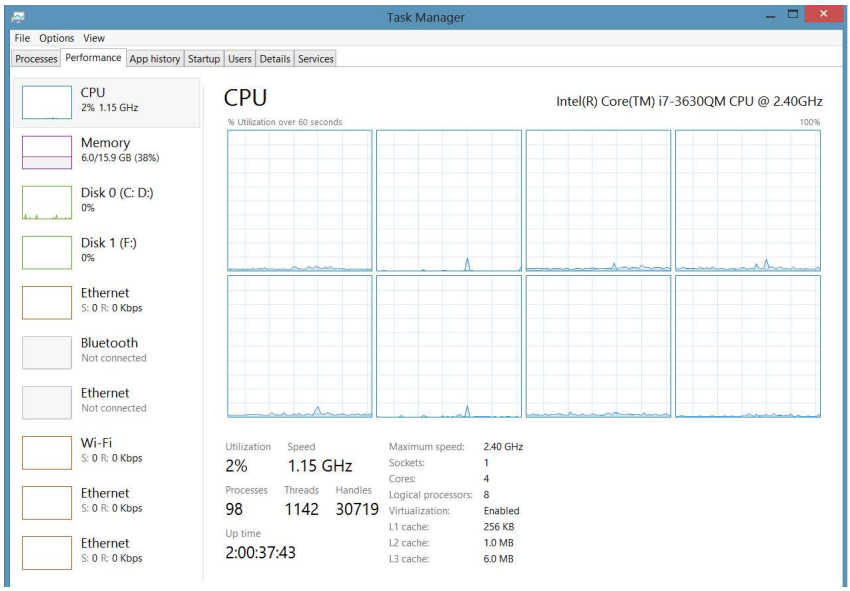
Kernel: **mpstat**



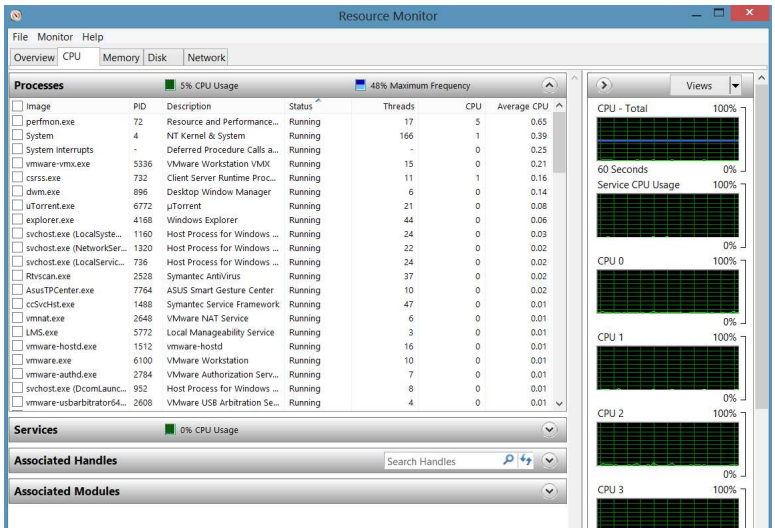
Kernel: **mpstat**

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0   28   2    0   192   83   92   32   14    2    0   185   78  15   0   7
 1   49   1    0    37    1   80   28   16    2    0   139   80  16   0   4
 2   28   1    0    20    7   94   34   17    1    0   283   83  12   0   5
 3   39   1    2    52    1   99   36   16    3    0   219   74  19   0   7
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0   34   0    2   171   75   78   32   12    1    0   173   90   9   0   2
 1   38   1    0    39    1   84   29   13    2    0   153   66  12   0  23
 2   28   8    0    21    9   97   31   20    2    0   167   67  13   0  20
 3   35   3    1    43    1   98   29   20    3    0   190   52  25   0  23
```

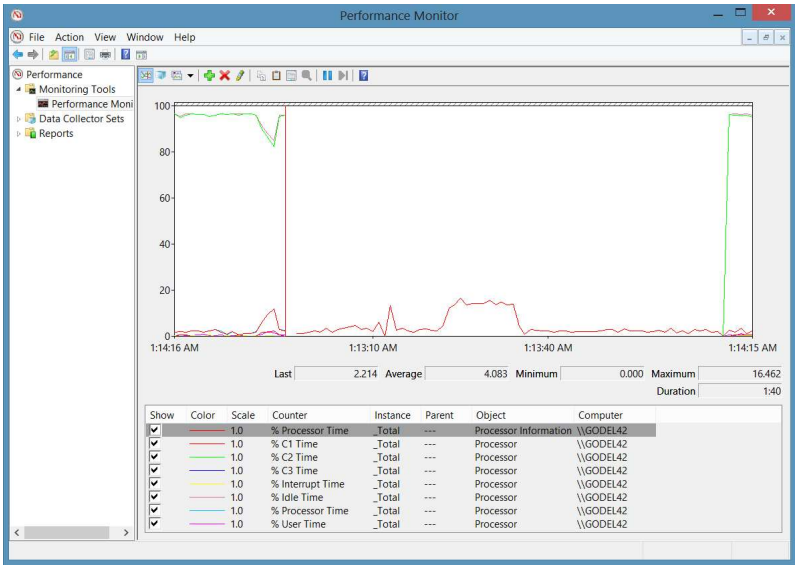
Monitoring CPU Utilization on Windows



Monitoring CPU Utilization on Windows



perfmon



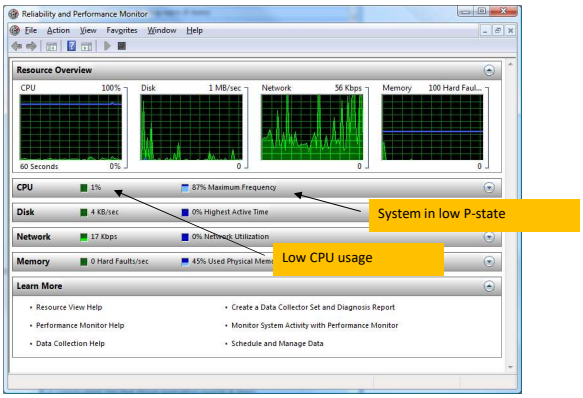
Processor Power Management in Windows

- Windows OS includes support for ACPI processor power management (PPM) features
  - support for processor performance states
  - support for processor idle sleep states on multiprocessor systems.
- PPM technologies are defined in the ACPI specification and are divided into three categories:
  - Processor performance states (ACPI “P-states”)
  - Linear stop clock throttle states (ACPI “T-states”)
  - Processor idle sleep states (ACPI “C-states”)
- For Intel, driver name is **`Intelppm.sys`**

```
powercfg -qh scheme_current sub_processor
```

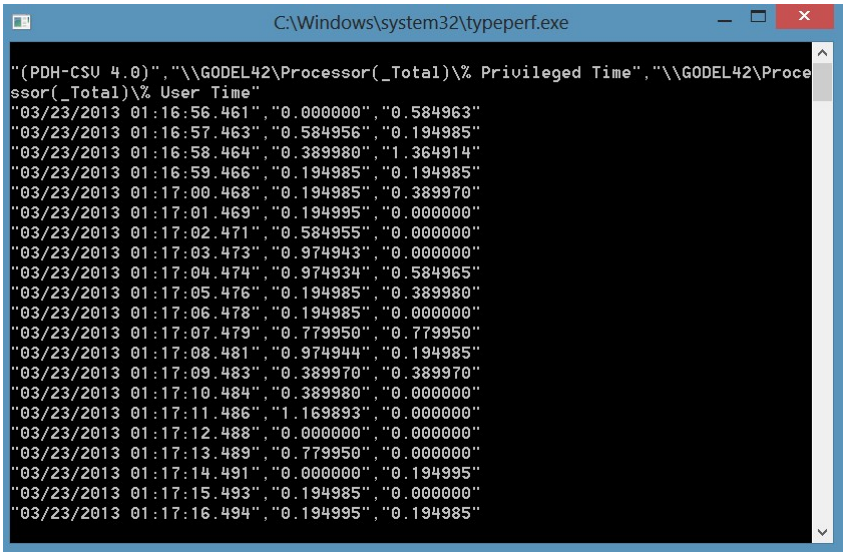
resmon

> Windows Reliability and Performance Monitor Showing Performance State Usage



typeperf

> typeperf "\Processor(_Total)\% Privileged Time" "\Processor(_Total)\% User Time"



typeperf

> You can also assemble a list of performance counters in a file and pass the name of the file to the typeperf command.

> For example, you can enter the following performance counters in a file named cpu-util.txt:

```
\Processor(_Total)\% Privileged Time
\Processor(_Total)\% User Time
```

> Then, invoke the **`typeperf`** command with the option **`-cf`** followed by the file name.

```
typeperf -cf cpu-util.txt
```


logman

> Helps to manage *Performance Monitor* and *performance* logs from the command line.

```
cmd> set _mycnt="\Processor(_Total)\% Processor Time"
cmd> set _mylogfile=C:\tmp\ss64.blg
cmd> logman create counter ss64-CPU -f bincirc -v mmddhhmm
 -max 250  -c %_mycnt% -o %_mylogfile%
The command completed successfully.
cmd> logman start ss64-CPU
The command completed successfully.

cmd> logman stop ss64-CPU
The command completed successfully.
```

OPTIMIZATION

Memory Management

- Preallocate Memory
  - Use `array`, `np.array`, or custom buffers for predictable memory usage and faster access.
- Minimize Object Creation
  - Reduce unnecessary object creation with loops and comprehensions.
  - Favor `itertools` functions.
- Properly Handle Closures
  - Closures capture references, potentially causing memory leaks.
  - Use nonlocal or `functools.partial` when needed.
- Profile for Memory Bottlenecks
  - Use tools like `memory_profiler` to identify areas with high object churn and target them for optimization.

Memory Management

- Garbage collection tuning
  - Tune garbage collector settings based on your application's memory usage patterns.
  - Consider tools like Pympler to analyze memory allocation.
- Close file handles and connections
  - Ensure proper resource management by closing files, database connections, and network sockets explicitly to avoid memory leaks.
- Minimize object creation
  - Avoid creating unnecessary temporary objects within loops.
  - Consider object pooling for frequently used objects.

Memory Management

- Use weak references
  - Use weakref to hold non-critical references to objects, allowing them to be garbage collected without impacting others.

Data Structures and Algorithms

- Choose the Right Data Structure
  - Use sets for membership checks, dictionaries for fast key-value access, and lists for ordered sequences.
- Algorithm Efficiency Matters
  - Utilize appropriate sorting algorithms like Timsort or quicksort for large datasets.
- Avoid Nested Loops
  - Unroll inner loops where possible for clearer code and potential performance gains.
- Memoization for Repeated Calculations
  - Cache expensive function calls with techniques like functools.cache for repeated inputs.

Code Structure and Optimization

- Utilize Comprehensions
  - Use list/dictionary comprehensions instead of explicit loops for concise and often faster code.
- Vectorize Operations
  - Leverage NumPy for vectorized operations on large arrays, significantly boosting performance.
- Optimize Imports
  - Organize and defer imports strategically to avoid unnecessary module loading at runtime.
- Reduce Function Calls
  - Inline short functions within loops to minimize overhead and improve efficiency.

External Factors and Tools

- Profile Your Code
  - Use profilers like cProfile or hotshot to identify execution hotspots and target bottlenecks.
- Utilize Caching Mechanisms
  - Cache database queries, API calls, or expensive computations for better responsiveness.
- Consider Asynchronous Programming
  - Utilize non-blocking I/O libraries like asyncio for improved concurrency and responsiveness.
- Monitor System Resources
  - Track CPU, memory, and network usage to identify potential bottlenecks and resource limitations.

Continuous Improvement

- Benchmark Regularly
  - Track performance changes as your code evolves and identify regressions or potential improvements.
- Readability Matters
  - Prioritize maintainable and clear code while optimizing, even if it leads to slightly slower execution.
- Stay Updated
  - Utilize modern Python features and libraries for optimized performance and improved syntax.
- Seek Community Resources
  - Leverage online resources, forums, and profiling tools to learn from others and improve your optimization skills.

Advanced Techniques

- Cythonize critical sections
  - Translate performance-critical parts of your code to C for significant speedups.
- Numba for JIT compilation
  - Use Numba to JIT compile specific functions for improved performance of numerical code.
- Multithreading and multiprocessing
  - Leverage multiple cores or CPUs for parallel execution of independent tasks.
  - Be mindful of overhead and data synchronization challenges.
- Asynchronous programming
  - Use libraries like asyncio for non-blocking I/O operations and improve responsiveness, especially for network-bound applications.

Advanced Techniques

- Memory-mapped files
  - Share data directly between Python and C++ code for efficient data exchange and avoid unnecessary copying

Database Optimization

- Optimize queries
  - Use proper indexing, avoid unnecessary SELECT * statements, and utilize JOINs efficiently.
  - Consider database-specific query optimization strategies.
- Caching database results
  - Cache frequently used database queries to reduce server load and improve response times.
- Database connection pooling
  - Reduce overhead by maintaining a pool of pre-established database connections.

Testing and Monitoring

- Write unit tests for optimizations
  - Ensure your optimizations haven't introduced regressions.
  - Test both performance and functionality.
- Monitor application performance
  - Use tools like New Relic or Prometheus to track resource usage, identify bottlenecks, and measure the impact of optimization efforts.
- Continuous integration and deployment
  - Automate performance testing and optimization changes into your CI/CD pipeline for a proactive approach.

Introduction to Profiling

> What is Profiling?
  – Analyzing program performance
  – Identifying bottlenecks in the code
> Why Profile Your Code?
  – Optimize execution time
  – Improve resource usage
  – Enhance user experience

Benchmarking and Profiling

> Recognizing the slow parts of your program is the single most important task when it comes to speeding up your code
> When designing a performance-intensive program, the very first step is to write your code without bothering with small optimizations:

"Premature optimization is the root of all evil."

Donald Knuth

Benchmarking and Profiling

- Make it run
  - Ensure that it produces the correct results.
- Make it right
  - Ensure that the design of the program is solid.
- Make it fast
  - Once our program is working and is well structured, you can focus on performance optimization.

What is cProfile?

- cProfile is Python's built-in deterministic profiler.
- Provides detailed statistics about program execution.
- Advantages
  - Lightweight
  - Easy to use
  - Highly integrated with Python’s standard library

Installing and Setting Up cProfile

> Pre-installed with Python
  – No installation needed.
> Import directly

```
import cProfile
```

  – Use via command line

```
python -m cProfile [script.py]
```


Key Features of cProfile

> Execution Profiling
  – Tracks function calls
> Detailed Statistics
  – Call counts, execution times, etc.
> Output Formats
  – Plain text
  – Binary (for visualization tools)
> Ease of Integration
  – Works seamlessly with Python programs

Benchmarking with pytest-benchmark

```
from simul import Particle, ParticleSimulator
def test_evolve(benchmark):
    # ... previous code
    benchmark(simulator.evolve, 0.1)
```

```
============================================= test session starts ==============================================
platform linux -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
benchmark: 3.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=5.00us max_time=1.00s cal
ibration_precision=10 warmup=False warmup_iterations=100000)
rootdir: /home/gabriele/workspace/hiperf/chapter1, inifile:
plugins: benchmark-3.0.0
collected 2 items

test_simul.py .


------------------------------------------- benchmark: 1 tests -------------------------------------------
Name (time in ms)         Min      Max     Mean  StdDev   Median     IQR  Outliers(*)  Rounds  Iterations
----------------------------------------------------------------------------------------------------------
test_evolve           29.4716  41.1791  30.4622  2.0234  29.9630  0.7376          2;2      34           1
----------------------------------------------------------------------------------------------------------

(*) Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
========================================== 1 passed in 2.52 seconds ==========================================
```


Finding bottlenecks with cProfile

- Two profiling modules are available through the Python standard library
- The `profile` module
  - This module is written in pure Python and adds a significant overhead to the program execution.
  - Its presence in the standard library is because of its vast platform support and because it is easier to extend.
- The `cProfile` module
  - This is the main profiling module, with an interface equivalent to profile.
  - It is written in C, has a small overhead, and is suitable as a general-purpose profiler.

Using cProfile: Command Line

> Basic Command
  - python -m cProfile script.py
> Sort Options
  - **`-s time`**
    - Sort by execution time
  - **`-s calls`**
    - Sort by call count
> Saving Results:

```
python -m cProfile -o output.prof script.py
```


cProfile

> cProfile does not require any change in the source code and can be executed directly on an existing Python script or function.
> You can use cProfile from the command line in this way:

```
python -m cProfile simul.py
python -m cProfile -s tottime simul.py
python -m cProfile -o prof.out simul.py
```

Using cProfile: Programmatically

> The usage of cProfile as a Python module requires invoking the cProfile.run function in the following way:

```
import cProfile
def fun():
   # function logic

cProfile.run("fun()")
```


cProfile

> You can also wrap a section of code between method calls of a cProfile.Profile object:

```
profiler = cProfile.Profile()
profiler.enable()
fun()
profiler.disable()
profiler.print_stats(sort='time')
```

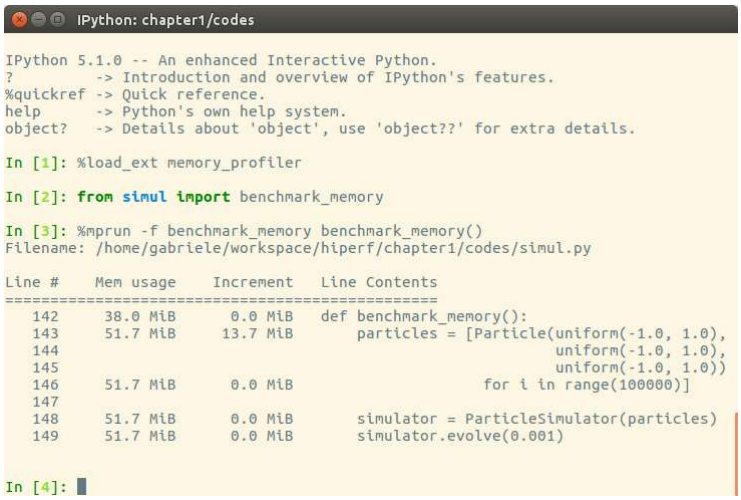
cProfile

```
IPython: chapter1/codes
(hperf) → codes ipython
Python 3.5.2 |Continuum Analytics, Inc.| (default, Jul  2 2016, 17:53:06)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: from simul import benchmark

In [2]: %prun benchmark()
         707 function calls in 1.231 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.230    1.230    1.230    1.230 simul.py:21(evolve)
        1    0.000    0.000    0.001    0.001 simul.py:118(<listcomp>)
      300    0.000    0.000    0.000    0.000 random.py:342(uniform)
      100    0.000    0.000    0.000    0.000 simul.py:10(__init__)
      300    0.000    0.000    0.000    0.000 {method 'random' of '_random.Random' objects}
        1    0.000    0.000    1.231    1.231 {built-in method builtins.exec}
        1    0.000    0.000    1.231    1.231 <string>:1(<module>)
        1    0.000    0.000    1.231    1.231 simul.py:117(benchmark)
        1    0.000    0.000    0.000    0.000 simul.py:18(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

In [3]:
```


Profiling memory usage with memory_profiler

```
IPython: chapter1/codes

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %load_ext memory_profiler

In [2]: from simul import benchmark_memory

In [3]: %mprun -f benchmark_memory benchmark_memory()
Filename: /home/gabriele/workspace/hiperf/chapter1/codes/simul.py

Line #    Mem usage    Increment   Line Contents
================================================
   142     37.8 MiB      0.0 MiB   def benchmark_memory():
   143     61.5 MiB     23.7 MiB       particles = [Particle(uniform(-1.0, 1.0),
   144                                                 uniform(-1.0, 1.0),
   145                                                 uniform(-1.0, 1.0))
   146     61.5 MiB      0.0 MiB                    for i in range(100000)]
   147
   148     61.5 MiB      0.0 MiB       simulator = ParticleSimulator(particles)
   149     61.5 MiB      0.0 MiB       simulator.evolve(0.001)


In [4]:
```

Profiling memory usage with memory_profiler



Profiling Metrics

- **ncalls**
  - Number of calls
- **tottime**
  - Time spent in a function (excluding subcalls)
- **percall**
  - Time per call
- **cumtime**
  - Cumulative time (including subcalls)
- **filename:lineno(function)**
  - Function location

Visualizing Profiling Data

> Tools for Visualization
  - `pstats` module
  - SnakeViz
  - PyProf2CallTree
> Example with SnakeViz:

```
pip install snakeviz
snakeviz output.prof
```


Best Practices for Profiling

> Profile critical sections of code.
> Use real-world data inputs.
> Compare different implementations.
> Visualize results for deeper insights.
> Avoid over-optimization.

Example: Profiling a Sorting Algorithm

```
import cProfile
import random

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

data = [random.randint(0, 1000) for _ in range(1000)]
cProfile.run('bubble_sort(data)')
```


Analyzing the Output

```
5 function calls in 0.050 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.050    0.050 <string>:1(<module>)
       1    0.050    0.050    0.050    0.050 exercise01.py:5(bubble_sort)
       1    0.000    0.000    0.050    0.050 {built-in method builtins.exec}
       1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
. . .
```

Profiling Larger Applications

> Example: Profiling a Web App

```
import cProfile
from my_web_app import app
cProfile.run('app.run()', 'output.prof’)
```

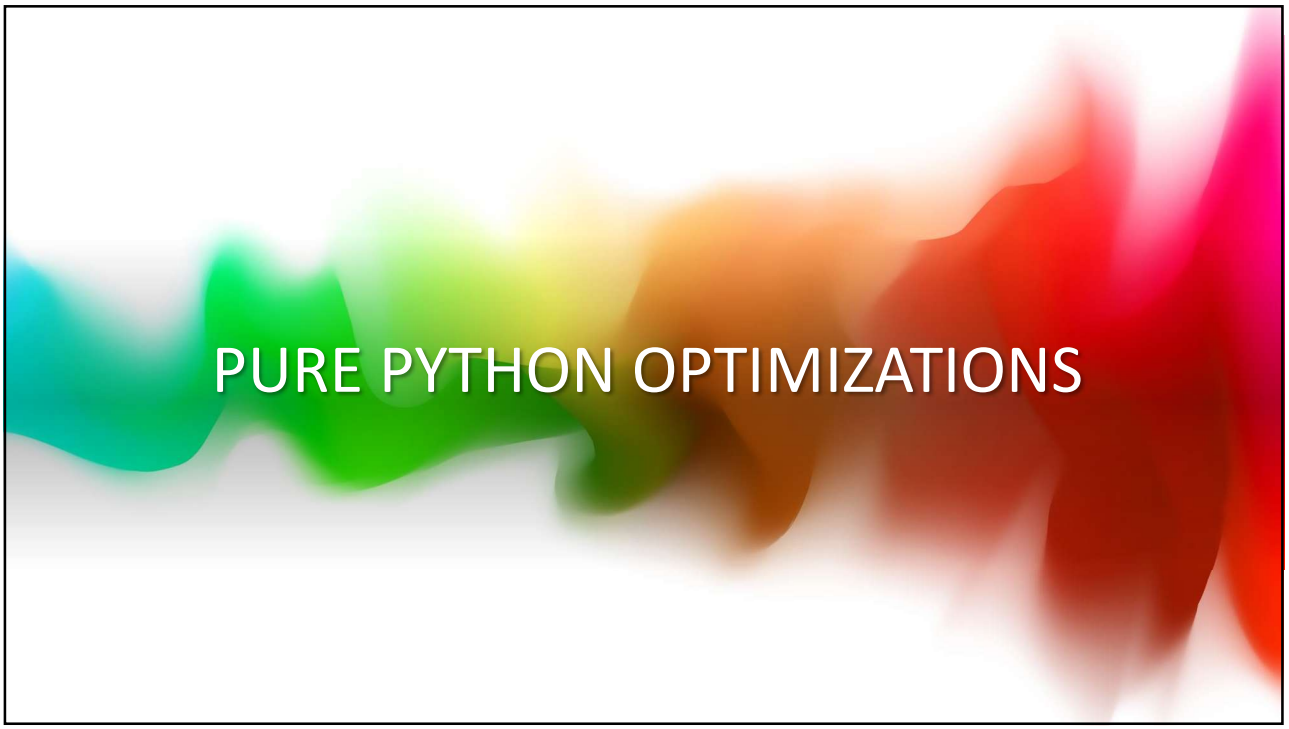
> Visualization
  > Use SnakeViz or KCachegrind

Profiling Decorator

```
import cProfile
import random


def profile(func):
    def wrapper(*args, **kwargs):
        profiler = cProfile.Profile()
        profiler.enable()
        result = func(*args, **kwargs)
        profiler.disable()
        profiler.print_stats(sort='time')
        return result
    return wrapper
```

Profiling Decorator

```
@profile
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

data = [random.randint(0, 1000) for _ in range(1000)]
bubble_sort(data)
```

Useful algorithms and data structures

> Algorithmic improvements are especially effective in increasing performance because they typically allow the application to scale better with increasingly large inputs
> Efficient Data Structures
  – Choose Wisely
  – Use **`set`** for membership tests
  – Use **`deque`** for queue operations
  – Use **`defaultdict`** for default values

Useful algorithms and data structures

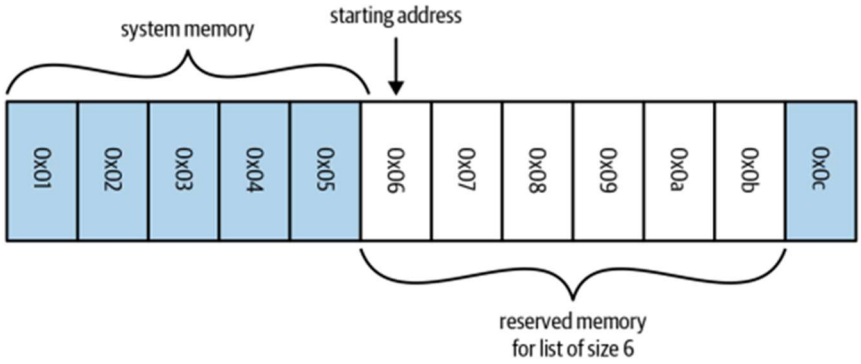
> Example:

```
from collections import defaultdict
counts = defaultdict(int)
for item in data:
   counts[item] += 1
```

Lists and deques

> Python lists are ordered collections of elements and, in Python, are implemented as resizable arrays.

> An array is a basic data structure that consists of a series of contiguous memory locations, and each location contains a reference to a Python object.



Lists and deques

```
>>> %%timeit l = list(range(10))
        ...: l[5]
        ...:
30.1 ns ± 0.996 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

>>> %%timeit l = list(range(10_000_000))
        ...: l[100_000]
        ...:
28.9 ns ± 0.894 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Linear Search

> What if we were given an array with an unknown order and wanted to retrieve a particular element?

```
def linear_search(needle, array):
    for i, item in enumerate(array):
        if item == needle:
            return i
    return -1
```


Binary Search

> Once a list has been sorted, we can find our desired element using a binary search
> Average case complexity is O(log n).

```
def binary_search(needle, haystack):
    imin, imax = 0, len(haystack)
    while True:
        if imin > imax:
            return -1
        midpoint = (imin + imax) // 2
        if haystack[midpoint] > needle:
            imax = midpoint
        elif haystack[midpoint] < needle:
            imin = midpoint+1
        else:
            return midpoint
```

Lists Versus Tuples

> If lists and tuples both use the same underlying data structure, what are the differences between the two?

> Lists are dynamic arrays
  – Mutable and allow for resizing.

> Tuples are static arrays
  – Immutable

> Tuples are cached by the Python runtime
  – No need to talk to the kernel to reserve memory every time we want to use one.

Using Built-in Functions

> Why Built-ins?
  – Implemented in C, faster than Python loops.

> Examples
  – **`sum`**, **`min`**, **`max`**, **`sorted`**
  – **`map`**, **`filter`**

Using Built-in Functions

> Example:

```
# Inefficient
result = []
for x in data:
  result.append(x**2)
# Optimized
result = map(lambda x: x**2, data)
```


List Comprehensions and Generators

> List Comprehensions
  – Faster and more concise than loops
> Generators
  – Use less memory by yielding items lazily

List Comprehensions and Generators

> Example:

```
# List comprehensions
quares = [x**2 for x in range(10)]
# Generator
squares_gen = (x**2 for x in range(10))
```


String Manipulations

> Avoid Concatenation in Loops
  – Use **`join`** for better performance

> Example

```
# Inefficient
result = ''
for s in strings:
    result += s
# Optimized
result = ''.join(strings)
```

Lazy Evaluation

> What is Lazy Evaluation?
  – Postponing computations until needed
> Techniques
  – Use iterators
  – Leverage **`itertools`** for combinatorics

Lazy Evaluation

> Example:

```
import itertools
result = itertools.islice(itertools.count(), 10)
```

Exception Handling Optimization

> Best Practices
  – Avoid using exceptions for flow control
  – Catch specific exceptions.

Exception Handling Optimization

> Example:

```
# Inefficient
try:
    result = data[index]
except IndexError:
    result = None
# Optimized
result = data[index] if index < len(data) else None
```

Avoiding Global Variables

> Why Avoid Globals?
  – Slower access compared to local variables.
> Solution
  – Pass variables as function arguments

Using `@lru_cache`

> Cache function results to avoid recomputation.

Using @lru_cache

> Example:

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fibonacci(n):
     if n < 2:
          return n
     return fibonacci(n-1) + fibonacci(n-2)
```


Parallel Processing with multiprocessing

> Utilize multiple CPU cores.

Parallel Processing with multiprocessing

> Example:

```
from multiprocessing import Pool

def square(x):
     return x**2

with Pool(4) as p:
     result = p.map(square, range(10))
```


Using **numpy** for Numerical Computations

> Highly optimized for array operations

Using **numpy** for Numerical Computations

> Example:

```
import numpy as np
data = np.array([1, 2, 3, 4])
result = data * 2
```


Reducing Function Call Overhead

> Inline critical functions to avoid call overhead

Reducing Function Call Overhead

> Example:

```
# Inefficient
def add(a, b):
     return a + b

result = add(x, y)

# Optimized
result = x + y
```


Optimize Imports

> Import only what you need

> Use local imports in specific functions

Avoid Repeated Calculations

```
# Inefficient
for i in range(len(data)):
    for j in range(len(data)):
        process(data[i], data[j])

# Optimized
length = len(data)
for i in range(length):
    for j in range(length):
        process(data[i], data[j])
```


Avoid Excessive Logging

- Why?
  - Reduces I/O overhead
- Best Practices
  - Log only essential information

Use **`with`** Statement for File Handling

> Benefits
  – Ensures proper resource cleanup

Use **`with`** Statement for File Handling

```
# Inefficient
f = open('file.txt')
data = f.read()
f.close()

# Optimized
with open('file.txt') as f:
     data = f.read()
```

Leverage Type Hints

> Why?
  – Improves readability and debugging
> Example

```
def add(a: int, b: int) -> int:
   return a + b
```


Batch Processing

> Processes large datasets in chunks, reducing processing time and resource usage
> Decreases overhead caused by multiple I/O operations, such as database writes or network calls.
> Handles larger datasets by dividing them into manageable pieces, ensuring system stability.
> Allows the system to focus on bulk operations, reducing the latency associated with individual processing.

When to Use Batch Processing:

- Database operations
  - Bulk insert or update records instead of performing one at a time.
- File handling
  - Process multiple files or records together
- Machine learning
  - Train models on batched data instead of single examples

Batch Database Operations

- Instead of inserting rows one at a time into a database:

```
# Inefficient: Single-row insertsfor row in data:
cursor.execute("INSERT INTO table_name VALUES (?, ?)", row)
# Optimized: Batch processing
cursor.executemany("INSERT INTO table_name VALUES (?, ?)", data)
connection.commit()
```

Batch File Processing

> Processing a large file line-by-line versus in batches:

```
# Inefficient: Line-by-line processing
with open('large_file.txt') as f:
    for line in f:
        process(line)
```


Batch File Processing

> Processing a large file line-by-line versus in batches:

```
# Optimized: Batch processing using chunks
def process_in_batches(file_path, batch_size=1000):
    with open(file_path) as f:
        batch = []
        for line in f:
            batch.append(line)
            if len(batch) == batch_size:
                process(batch)
                batch = []
        if batch:  # Process remaining lines
            process(batch)

process_in_batches('large_file.txt')
```