Profiling

G S Nitesh Narayana

Workshop Repo

https://gitlab.com/hprcse/workshops/2019/hpc_workshop

What is Profiling?

It's a tool to optimise things for you?

Does the dirty work of making your slow code fast!?

Image Credits: smartbear.com

NO!!

unfortunately(?) U have to make your code faster!!

What it can do is, help you do that!!

How?

Well that's why we need to learn profiling

- where your program spent its time.
- which functions called which other functions while it was executing.
- which pieces of your program are slower than you expected.

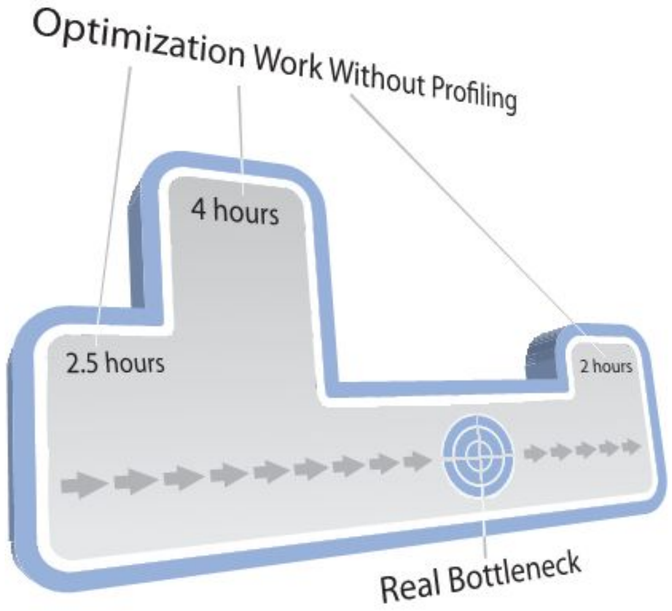
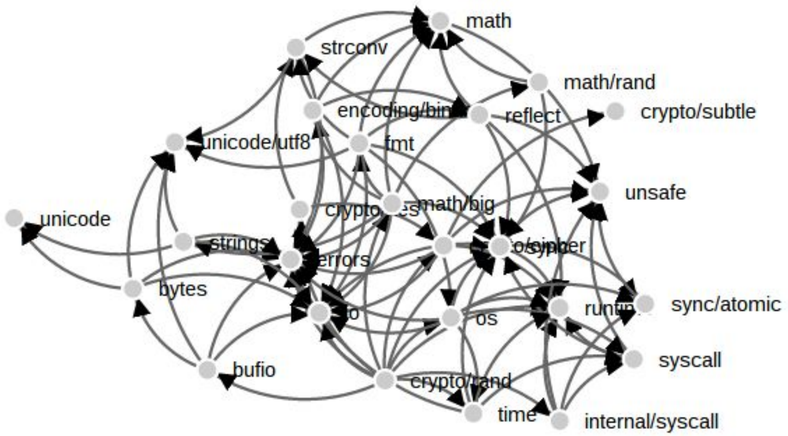


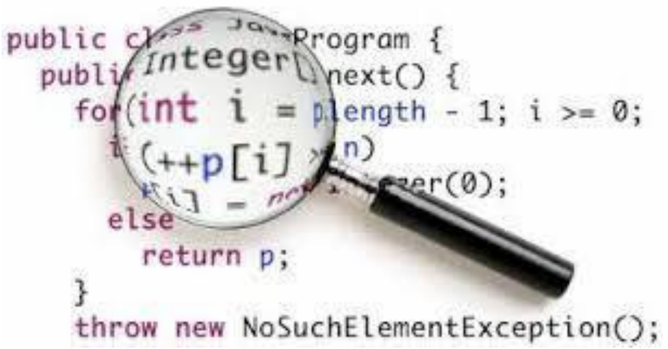
Image Credits: smartbear.com

“ Its a tool that does dynamic program analysis”

Dynamic



Static



Algorithmic

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Image Credits: Google

Will be Discussing

- GPROF
- LIKWID
- Valgrind

Also what each needs...

There is a bonus for you in the end!!

Steps in Profiling

Profile!

Profile!

Profile!

What is it?

What does it contain?

Call Graph

Flat Profile

How do you know who finished **FIRST**?!

Image Credits: Google

Therefore, you need measures to measure!

Time(?) is a start...

GNU PROF

Gives an execution profile of the program

yes i know what you are thinking 😁

Image Credits: Google

Hands on Keyboard!

What's inside the analysis.txt !?🧐

Contains Call Graph and Flat Profile

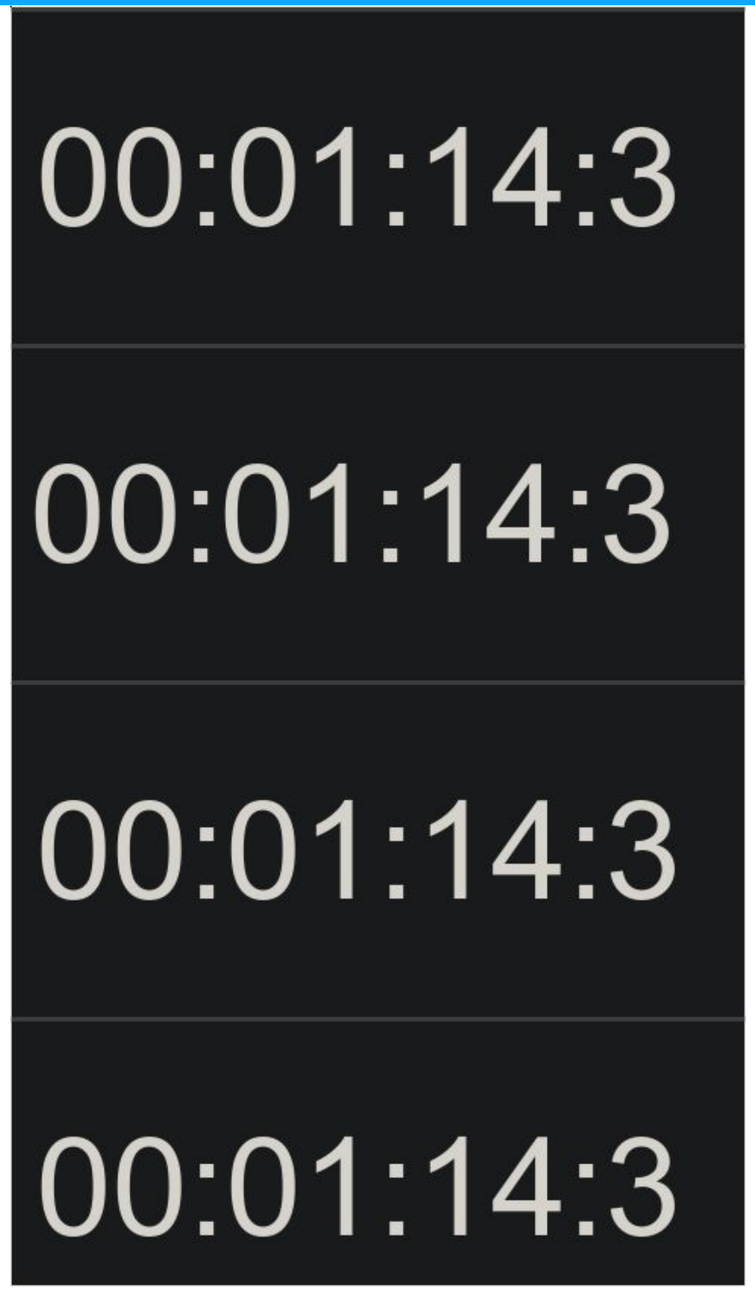
- Flat Prof shows how much time your program spent in each function and how many times that function was called.
- Call Graph shows, for each function, which functions called it, which function it called, how many times.

Let's open in!!

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative self              self     total
 time seconds   seconds calls  ms/call ms/call  name
 96.43 0.81      0.81       1   810.00  810.00   func3
 3.57 0.84       0.03       1    30.00  840.00   func1
 0.00 0.84       0.00       1     0.00  810.00   func2
 0.00 0.84       0.00       1     0.00    0.00   func4
```

```
Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 1.19% of 0.84 seconds

index % time    self  children    called     name
                0.03    0.81       1/1          main [2]
[1]    100.0    0.03    0.81         1        func1 [1]
                0.00    0.81       1/1          func2 [3]
-----------------------------------------------
                                                <spontaneous>
[2]    100.0    0.00    0.84                  main [2]
                0.03    0.81       1/1          func1 [1]
                0.00    0.00       1/1          func4 [5]
-----------------------------------------------
                0.00    0.81       1/1          func1 [1]
[3]     96.4    0.00    0.81         1        func2 [3]
                0.81    0.00       1/1          func3 [4]
-----------------------------------------------
                0.81    0.00       1/1          func2 [3]
[4]     96.4    0.81    0.00         1         func3 [4]
-----------------------------------------------
                0.00    0.00       1/1          main [2]
[5]      0.0    0.00    0.00         1         func4 [5]
```

Take some time and play with it!



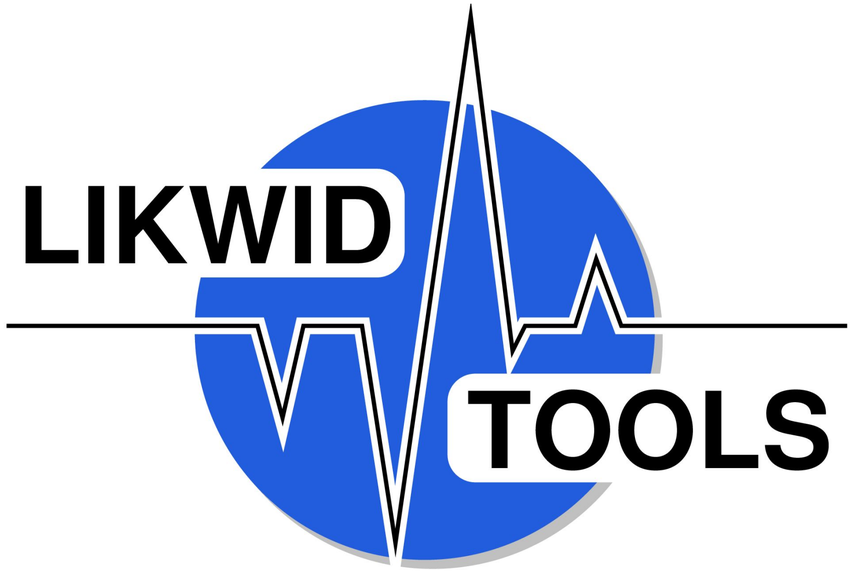
Consider this scenario😱

Need a more accurate/reliable measure or may be some other measures...

GPROF

LIKWID

Valgrind



LIKWID

a mini toolsuite with various performance analysis measures

Image Credits: LIKWID group

About LIKWID

- Has a variety of tools.
- Gives us a chance to look into hardware metadata -> giving new measures.
- The MSRs for starters...

MSRs

- A Model-Specific Register , one of those hardware registers that helps us debug, trace, measure the program.
- We need them for reading CPU performance counters when an operation is done.
- Find the readings in */dev/cpu/*/msr*

The Tools

- likwid-topology: print thread, cache and NUMA topology
- likwid-perfctr: configure and read out hardware performance counters on Intel, AMD and ARMv8 processors
- likwid-powermeter: read out RAPL Energy information and get info about Turbo mode steps

Cont.

- likwid-pin: pin your threaded application (pthread, Intel and gcc OpenMP to dedicated processors)
- likwid-bench: Micro benchmarking platform
- likwid-features: Print and manipulate cpu features like hardware prefetchers

Cont..

- likwid-genTopoCfg: Dumps topology information to a file
- likwid-mpirun: Wrapper to start MPI and Hybrid MPI/OpenMP applications (Supports Intel MPI, OpenMPI, MPICH and SLURM)
- likwid-perfscope: Frontend to the timeline mode of likwid-perfctr, plots live graphs of performance metrics using gnuplot
  -

Cont...

- likwid-memsweeper: Sweep memory of NUMA domains and evict cachelines from the last level cache

- likwid-setFrequencies: Tool to control the CPU and Uncore frequencies (x86 only)

Permission denied? Couldn't Find them?

```
$ sudo modprobe msr
$ sudo chmod +rw /dev/cpu/*/msr
```

- Depending on the number of CPUs and number of processors, the number of msr files may vary.
- Each file is given a MSR NUMBER. We use it to access the respective msr.

Installing LIKWID

- Link for the repo

  https://github.com/RRZE-HPC/likwid

- Follow the steps from that site.

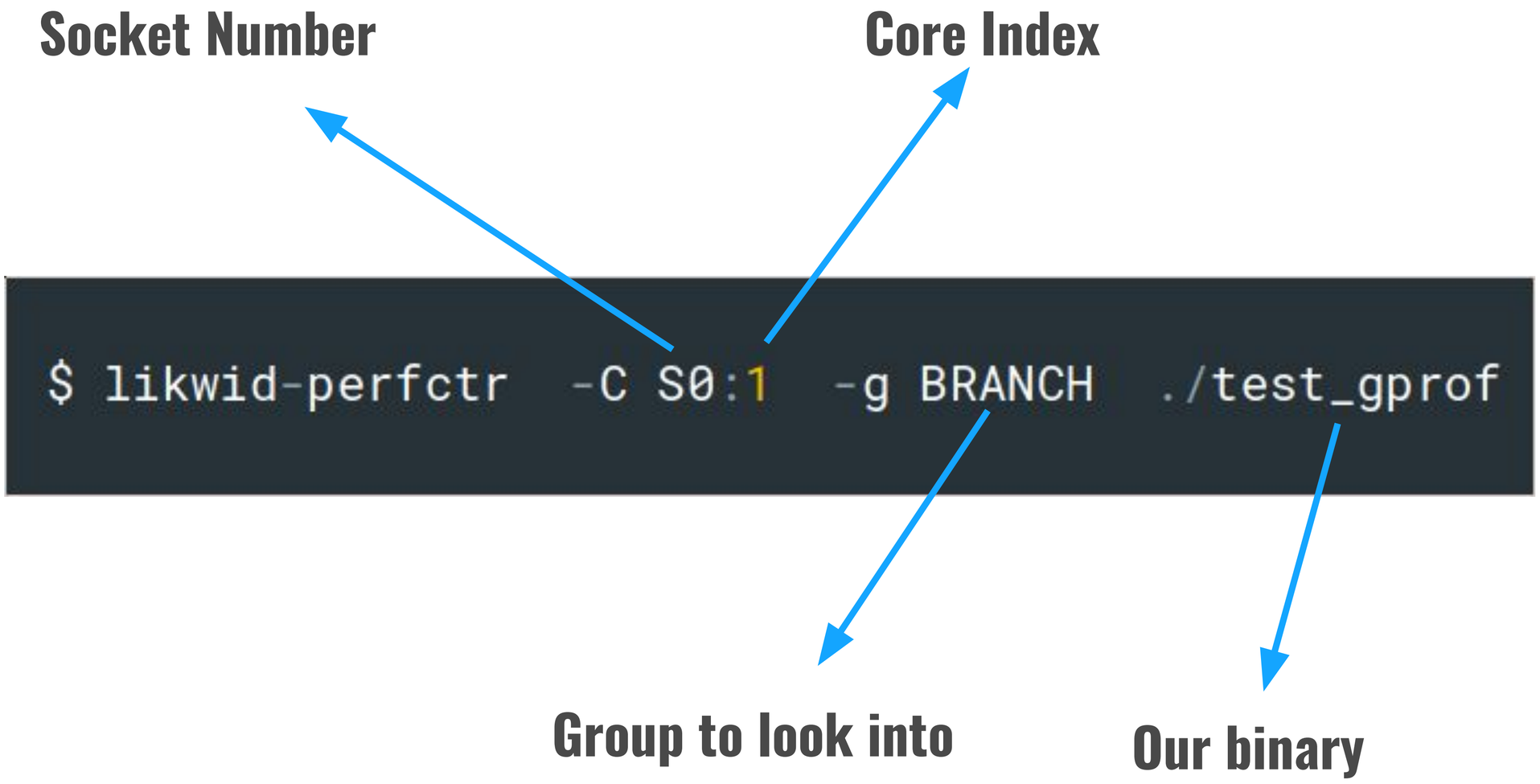
```
$ tar -xjf likwid-<VERSION>.tar.bz2
$ cd likwid-<VERSION>
$ vi config.mk
$ make
$ sudo make install
```

Hands on LIKWID! (perfctr)

```
$ likwid-perfctr -a
$ likwid-perfctr -e | less
```

Shows supported groups

supported counter registers and events


Socket Number
Core Index
$ likwid-perfctr -C S0:1 -g BRANCH ./test_gprof
Group to look into
Our binary

Execute and try to infer the output

LIKWID

Valgrind

BONUS(?)

Valgrind

a mega toolsuite with various performance analysis measures (memory based?)

Image Credits: Valgrind group

About Valgrind

- Has a variety of tools.
- It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.

If you don't want to worry about memory leaks start using

RUST or OCaml !!!

Icon Credits: Google

The Tools

- MemCheck: detects memory management problems.
- Cache-grind: Cache and Branch Predictor profiler, for analysis of the behaviour of caches
- Call-grind: extension to cache-grind which provides a call graph

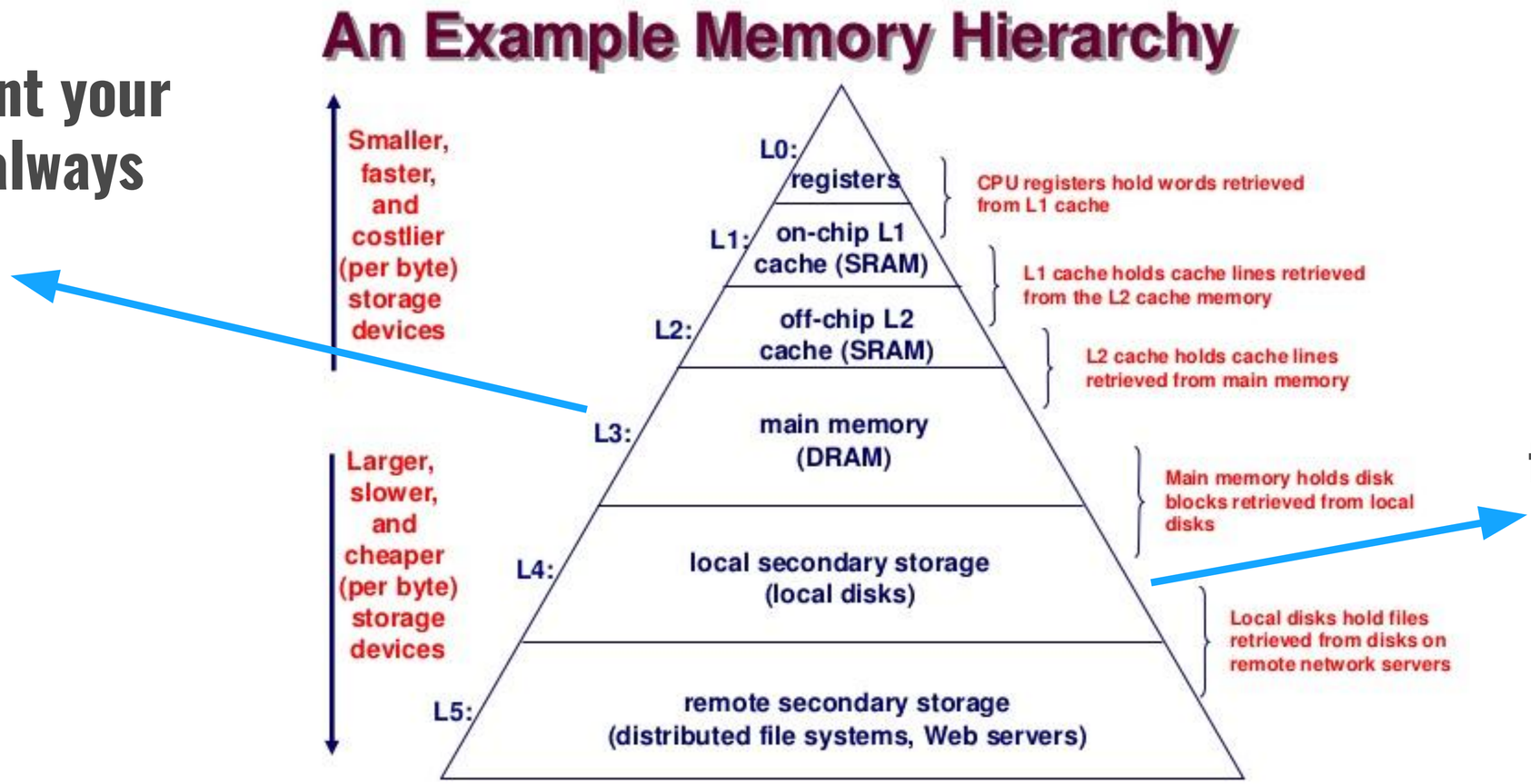
Cont.

- Massif: is a heap profiler. Monitors the program’s heap space
- Helgrind: is a thread debugger, for data races in multithreaded programs.
- DRD: similar to Helgrind but takes less memory to perform the analysis

There are other experimental tools in this suite , have a look at them

Why bother about memory leaks or management

you don't want your program to always come here



this is worse!!

Image Credits: Google & CMU

Try to avoid $ misses !

Installation

```
$ tar -zxvf valgrind<tab>
$ cd valgrind<tab>
$ ./configure
$ make -j4
$ sudo make install
```

Valgrind → Cache-Grind

- It gives the statistics of the form, keep them in mind !!:

Xyz

X∈ {I, D, LL}
y∈ {1, L}
z∈ {r,w, mr, mw}

I → Instruction
D → Data
LL → Last Level
r → read
w → write
mr → read miss
mw → write miss

Lets CacheGrind !

```
$ valgrind --tool=cachegrind <prog>
$ cg_annotate <file_name>
```

```
--branch-sim=yes
```

add this with first command for branch prediction analysis

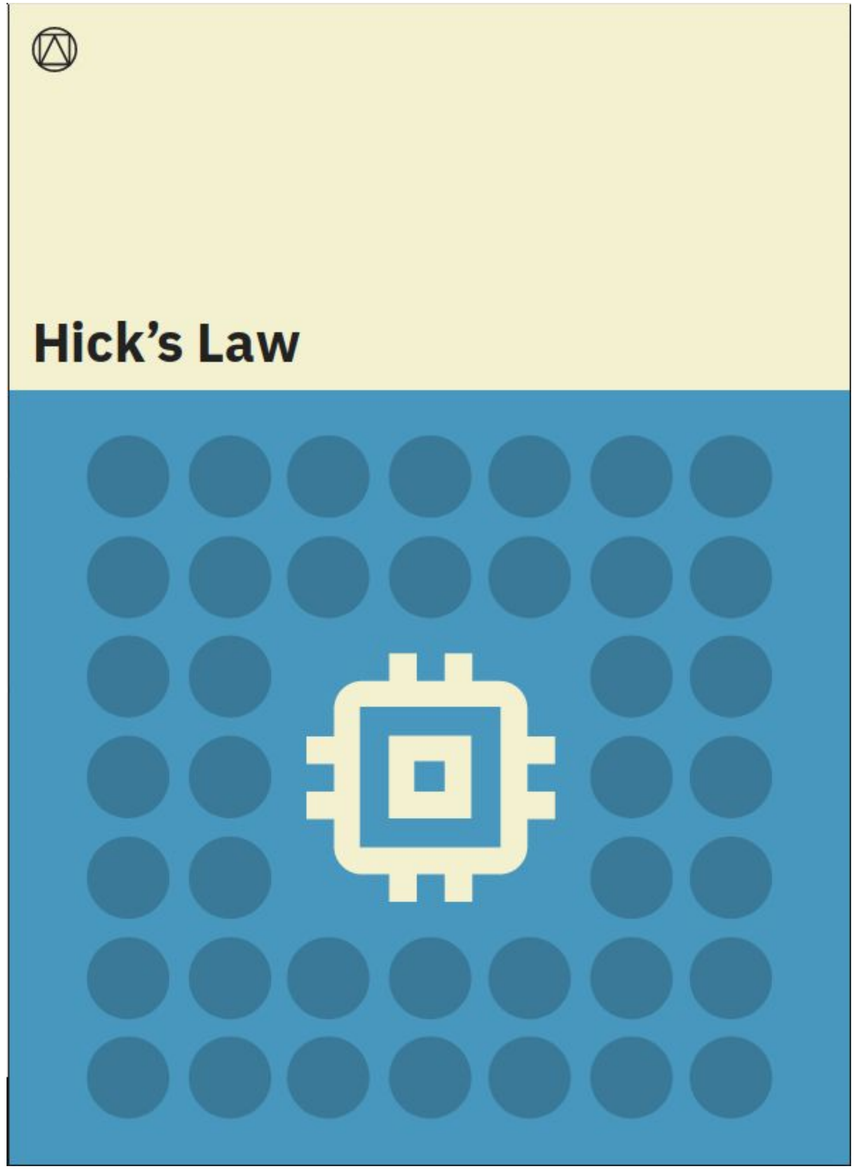
```
I1 cache:                65536 B, 64 B, 2-way associative
D1 cache:                65536 B, 64 B, 2-way associative
LL cache:                262144 B, 64 B, 8-way associative
Command:                 concord vg_to_ucode.c
Events recorded:         Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:            Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order:        Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Threshold:               99%
Chosen for annotation:
Auto-annotation:         off
```

if you want even the smallest counts to be shown

```
$ cg_annotate <file_name> <source_file>
```

```
Ir  I1mr ILmr Dr  D1mr DLmr Dw D1mw DLmw

 .    .    .    .    .    .  .    .    .
 .    .    .    .    .    .  .    .    .  #include <stdio.h>
 .    .    .    .    .    .  .    .    .
 .    .    .    .    .    .  .    .    .  void swap(int *xp, int *yp)
56    0    0    0    0    0 42    0    0  {
42    0    0   28    0    0 14    0    0      int temp = *xp;
56    0    0   42    0    0 14    0    0      *xp = *yp;
42    0    0   28    0    0 14    0    0      *yp = temp;
42    0    0   28    0    0  0    0    0  }
```

Take your time and try not to miss the $ accesses !



“ The time it takes to make a decision increases with the number and complexity of choices.”

SOOO Many profiling tools around...

Image Credits: Jon Labonski UX design

How to Select Which Profiler!?

- Profiler Use Challenges
- Low Impact, Integrated Profiling
- Ease of Use
- Multiple measurements
- Detailed Reporting

Benchmarking Vs Profiling

- Select 2 programs
- Put them against each other
- Get a metric/score and Compare

YOU TELL ME!!!

End...

Bonus?

Intel VTune

So powerful you don't know you witnessed!

Only an Intro 😎

Why This

- Pinpoint **HOTSPOT** identification.
- Supports C, C++, DPC++, Google Go*, Fortran ,Python(!) & more...
- Microarchitecture level profiling!!
- Local and remote data collection
- GUI 😎
- and ofcourse **INTEL** and **much more.....**

VTune Flow

- As Standalone
  - GUI
  - Command line
- From Studio package
  - Intel Parallel Studio XE
  - Intel System Studio
  - Intel Media Server Studio
- Within Microsoft* Visual Studio

- WHERE
  - Local host
  - Remote system
  - Arbitrary host
  - Android device
- WHAT
  - Application
  - Process
  - System
- HOW
  - Hotspots
  - Microarchitecture
  - Parallelism
  - Platform analyses

- Summary window
- Source/Assembly pane

HOTSPOT?🔥

Where in a system or application there is **significant** amount of **activity**

Demo time!

Thanks!

Any questions?

You can find me at:
nitesh8998.gitlab.io