HPES
High Performance Embedded Systems

EEE4120F

University of Cape Town • iYunivesithi yaseKapa • Universiteit van Kaapstad

High Performance Embedded Systems

Lecture 6:

Performance benchmarking, Metrics, Profiling & Testing

benchmarking

Lecturer:
Simon Winberg

Planned as short recorded lecture

Outline for Lecture

- Performance benchmarking
- Metrics of Performance
- Average Cycles Per Instruction (ACPI)
- Cost vs. Performance
- SWAP
- Profiling code (Reading: Valgrind)

Reminder: Performance Benchmarking

- "Don't loose sight of the forest for the trees…"
- Generally, the main objective is to make the system faster, use less power, use less resources…
- Most code doesn't need to be parallel.
- Important questions are…

Benchmarking: what to test

- What can be benchmarked? For DSP and HPC…
- Compiler
  - Converts High Level Language to Assembly language thus we benchmark compiler efficiency, such as how efficient is the generated assembly code?
- The Processor
  - Code in hand-crafted/inspected assembly (to make comparisons fair)
- Operating System
  - Interrupt latencies, overhead of operating system calls, limits on devices, kernel size, availability of services and facilities such as support for virtual memory and paged memory.
- Platform
  - Scalability of memory. Peripheral limits. Interfaces supported. Power use. Power saving features. OS's supported.
- Applications (e.g. representative operations for certain application domains – think 'DWARFS' as in Berkeley paper)

Benchmarking: what is usually measured

- For DSP/HPEC systems we typically benchmark:
  - Cycle count
  - Data and Program memory usage
  - Execution time
  - Power consumption (has recently become a common thing to report on)

Benchmarking Techniques

EEE4120F

Metrics of Performance

- Application
  - Operations per second (OPS)
- Programming language
  - Expressiveness*
  - Code density
- Processing system
  - MIPS: Million Instructions Per Second
    GIPS: Giga (i.e. 10^9) IPS
  - MFLOPS: Million FLoating point OPerations per Second
    GFLOPS: Giga (10^9) FLOPS
    TFLOPS: Terra (10^{12}) FLOPS
- Datapath
  - MBps: Megabytes per second or GBps
- Function unit / processor hardware
  - Cycles per second (CPS) (based on clock rate)

* Felleisen, M., 1991. On the expressive power of programming languages.*Science of computer programming*, *17*(1-3), pp.35-75.

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Property of system used to rating aspect on left

Aspect of processing rated	Instruction Count (IC)	Cycles Per Instruction (CPI)	Clock
Program Size	✓		
Computer	✓	✓	
Instruction Set	✓	✓ *	
Organization		✓	✓
Technology			✓

Note that this is only a few examples of aspects that may want to be rated in regards to what performance a CPU achieves. And there are also many more properties that may be used in doing this rating accurately (e.g. number registers may impact program size).

* Some instructions may of course take multiple cycles … see next slides for ACPI

Hint: Sorry to say, but something like this is hard to resist in setting test questions!

Average Cycles Per Instruction:
processors with varying clocks per instruction

- Average Cycles Per Instruction (ACPI)

CPI = Cycles / Instruction Count
= (CPU Time * Clock Rate) / Instruction Count

$$\text{CPU time} = \text{CycleTime} * \sum_{i=1}^{n} \text{CPI}_i * I_i$$

i.e. all instructions I_1 to I_n where CPI_i is the cycles needed to complete instruction I_i

"Instruction Frequency" (F_i) (how often particular instruction is invoked)

$$\text{ACPI} = \sum_{i=1}^{n} \text{CPI}_i * F_i \qquad \text{where} \quad F_i = \frac{I_i}{\text{Instruction Count}}$$

(You can read more on https://en.wikipedia.org/wiki/Cycles_per_instruction, but only the basics of this slide is examinable.)

Example: Calculating ACPI

Operation	Frequency (F_i)	CPI_i	$CPI_i * F_i$	Time %
Store	10%	2	0.2	**13%**
Load	20%	2	0.4	**27%**
Branch	20%	2	0.4	**27%**
ALU	50%	1	0.5	**33%**
ACPI	100%		1.5	**100%**

Summarizing Performance

- Arithmetic mean (weighted * arithmetic mean) tracks execution time (n = number runs):
  - **$\Sigma(T_i)/n$ or $\Sigma(W_i*T_i)$**
- Harmonic mean (weighted harmonic mean) of rates (e.g., R = MFLOPS) tracks execution time:
  - **$n/\ \Sigma(1/R_i)$ or $n/\ \Sigma(W_i/R_i)$**
- Normalized execution time useful for scaling performance
  - **e.g. X times faster than a Pentium4**
  - Arithmetic mean impacted by choice of reference machine (e.g. MIPS-1 processor)
- Use the geometric mean for comparison:
  - **$\Sigma(T_i)$^1/n**
  - Independent of chosen machine…
  - but not good metric for total execution time

* Weighting assigns particular value (or priority of importance to certain runs)
T_i = time of run i

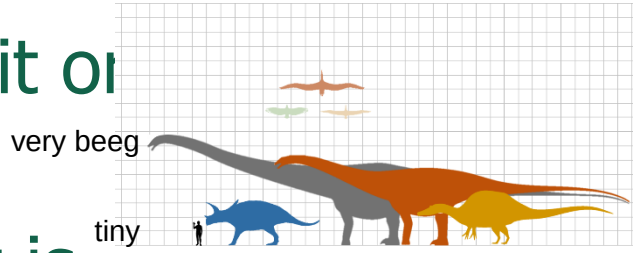
Cost/Performance:
The Relationship of Cost to Price?

- Recurring Costs
  - Component Costs
  - Direct Costs, recurring costs: labor, purchasing, warranty, scraping
- Non-Recurring Costs or Gross Margin
  - R&D, equipment maintenance, machine and test equipment rentals, marketing, sales, financing cost, pretax profits, taxes, etc. etc.
- Average Discount to get List Price
  - Allowing for volume discounts and/or retailer markup

Remember SWAP

Benchmarking between platforms and implementations can also be in terms of:

- **Size** – footprint/size the circuit or product takes up



- **Weight** – how heavy product is (also concerns for the connectedness/dependencies – like a friend who brings tons of baggage on the hiking trip and you end up having to carry it) … **And**

- **Power** – how much power the product uses, which implies its level of mobility and cost in keeping it running, etc.

Profiling Code

EEE4120F HPES

Profiling

- Code profiling or
  Software profiling or
  Program profiling =
  - A process of dynamic program analysis (rather than 'static code analysis'*) that investigates the program's behavior by gathering data related to its execution (e.g. stack use, O/S calls, etc.) while the program executes.
- The purpose:
  - Usually profiling is done to analysis which sections of a program to optimize, in order to increase its overall speed, decrease its memory requirement, decrease its power consumption … or a optimize around a combination of these aspects.

* static code analysis is a code quality inspection and bug finding method. Typically done as a manual code reviews, not running the code but reading over the code.

Reasons for profiling code

Prescribed: read on your own

- Better understanding how your program runs on the architecture
- Reduced hardware & maintenance cost
- Learn how to be a better coder
- Develop better programs
- 80/20 rule in efficient code development
  - 80% of runtime uses only 20% of the code

Profiling pitfalls

Prescribed: read on your own

- Pre-optimization might be a costly waste of time (WoT)
- Optimizing the 80% of the code that runs 20% of the time may be a WoT
- Not adequately understanding the architecture or application workload
- Over optimizing code (e.g. making 4ms into 1ms when you don't need less than 5ms)
- Can overcomplicate code
- Not understanding the bottlenecks

Common software bottlenecks

Prescribed: read on your own

- Typical bottlenecks
  - CPU
  - Memory
  - Disk
  - Lock contention (processes flighting over devices)
  - Network
  - External resources / IO
  - Databases access, etc…

Command-line tools for simple profiling on Linux

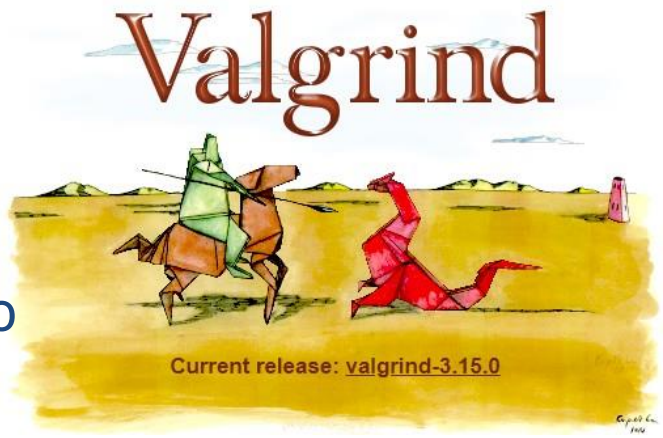
Prescribed: read on your own

- top, htop (useful for threaded apps)
- strace
  - Intercepts and records system calls, which are called by a process and the signals which are received by a process
- vmstat (virtual memory statistics)
  - reports info on processes, memory, paging, IO blocks, traps, disks and CPU activity
- dstat (enhancement of vmstat)
  - View (or report for a period) all of your system resources, you can e.g. compare disk usage in combination with interrupts from your IDE controller, or compare the network bandwidth numbers directly with the disk throughput. *
- time (very simple but useful sanity check!)
  - When command finishes, prints info on timing statistics about the program: (i) elapsed real time between invocation and termination, (ii) the user CPU time, and (iii) the system CPU time.

If you have not tried time already, you are seriously missing out… Quick! Find a Linux computer and try it out!

* May need to install, not always standard on Linux. Reference and further info at: https://linux.die.net/man/1/dstat

Valgrind : open-source profiler



- Valgrind: an instrumentation framework for building dynamic analysis tools.
- There are Valgrind tools that can automatically detect many memory management and threading bugs, and to profile your programs in detail.
- Can also use Valgrind to build new dynamic analysis / code profiling tools.
- The Valgrind distribution includes six production-quality tools:
  - memory error detector
  - 2x thread error detectors
  - cache and branch-prediction profiler
  - call-graph generator
  - branch-prediction profiler
  - heap profiler

Reading task:

read ‘About’ page for Valgrind (very useful): https://valgrind.org/info/about.html

Find out more and possibly try it out by visiting: https://valgrind.org/

Prac 1 - reasoning

- Prac1 is not just about testing
- Why correlation is used in the intro:
  - The purpose here is that for later pracs, or parts of the project, you may want to use correlation to compare the results of different implementations (i.e. to compare a dodgy prototype to a trusted 'golden measure', using e.g. Julia, OCTAVE or MATLAB to do correlations and other stats to check your prototype)
- Using Latex… I suggest either
  - TexStudio : https://www.texstudio.org/
  - Overleaf : https://www.overleaf.com/

Next lecture will say a bit more about prac reports, planning project report etc.

closing
remarks & reminders…

End of Lecture

Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons “Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)” license, and that is why I selected that license to apply to this presentation (it’s not because I particulate want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:

Wikipedia (open commons)
http://www.flickr.com
http://pixabay.com/
Forrest of trees: Wikipedia (open commons)