EEE4120F

High Performance Embedded Systems

Lecture 6:

Performance benchmarking, Metrics, Profiling & Testing

benchmarking

Lecturer:
Simon Winberg

Planned as short recorded lecture

Audio narrations have been left in some of the slides, these are provided as a means to just supplement some of the information, you do not need to listen to them (press on the speaker icon if you want to listen to the annotation).
The lecture sessions will present the slides and provide an opportunity for questions and discussion of the topics.

Outline for Lecture

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

* Reading R2 concerning the Valgrind profiling tool is an assigned reading; you would be expected to have some awareness of profiling tools and have an understanding of R2 which can be considered examinable.

We focus on the metrics of performance, ACPI and SWAP in this lecture.

The Valgrind profiling tool (Reading R2) is an assigned reading, which will be covered in the seminar for this week. Note that some awareness of profiling tools is expected (for the exam you should be able to respond to questions related to topics covered in Reading R2, especially in parts R2b and R2c).

Reminder: Performance Benchmarking

(recap from Lecture 4)

- "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…

(recap from Lecture 3)

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)

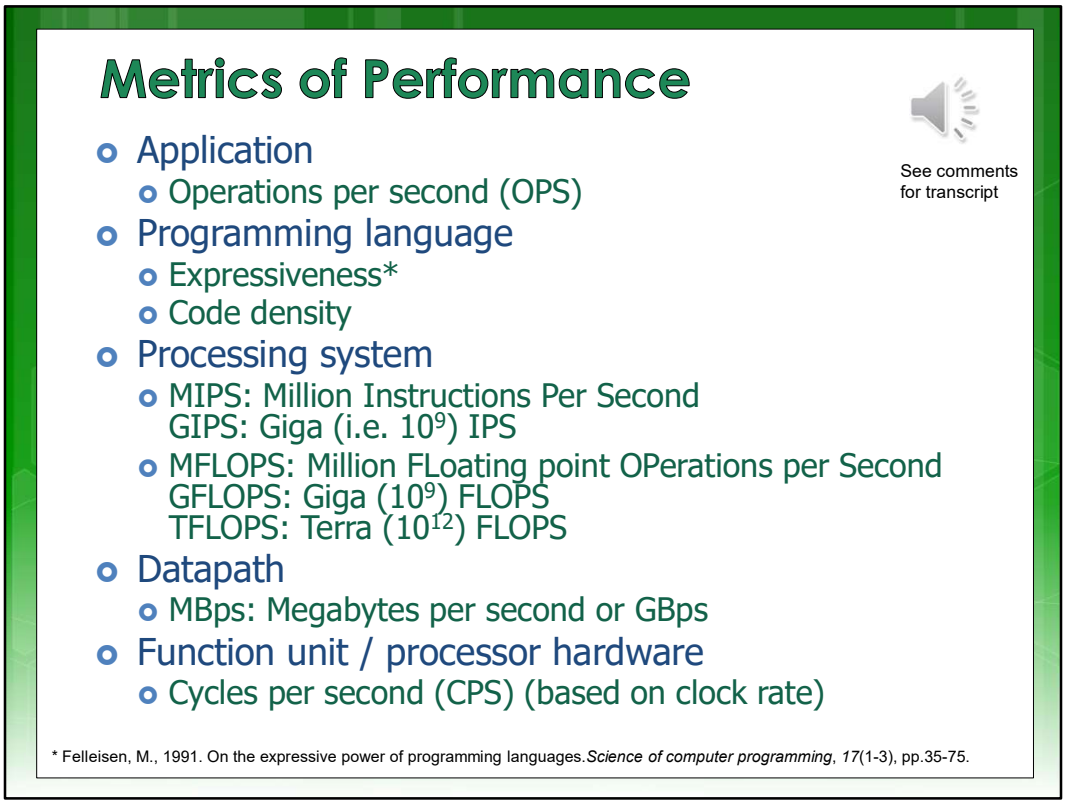
Benchmarking: what is usually measured

(recap from Lecture 3)

- 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



Audio Transcript: System Benchmarking Metrics

There are various metrics for benchmarking a computer system. For applications, the concept of operations per second, or ops, are often used. Here an operation is not a machine instruction; rather think of it as a function, a sequence of instructions excluding input and output time. I know, it's a bit vague and would take a while to explain; rather don't consider it a thread or a process either, these may be worked on intermittently.

Programming Language Metrics

For programming languages major metrics are:

Expressiveness of the language

Its code density

For example, C has high expressive power; you can write almost any computer program in C. SQL, on the other hand, is a lot less expressive; you would struggle writing a point of sale program using just SQL.

But SQL, like domain specific languages generally, can be very dense. It can represent complex processing needs in a few lines. C is different; it is less dense, taking a lot of coding to complete a similarly complex operation.

Processing and Hardware Metrics

For benchmarking processing there is an abundance of metrics. MIPS, GIPS and FLOPS are the most popular. Data path is typically measured in megabits per second. Function units, for instance, an application accelerator that completes an operation in one cycle, may be rated in cycles per second.

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!

CPU Performance metrics can be viewed from multiple perspectives.

Thinking just about how many instructions a processor can get through in a given time duration is not necessarily a meaningful measure, depending on the type of processor and application concerned.

For example, you could be comparing a CISC processor to a RISC processor. The RISC processor may complete an instruction at twice the speed it takes for the CISC processor. But, the RISC might still end up taking ten times longer to add up a sequence of floating point numbers compared to the CISC.

But, the CISC may take ten times longer than the RISC to forward data from one port to another.

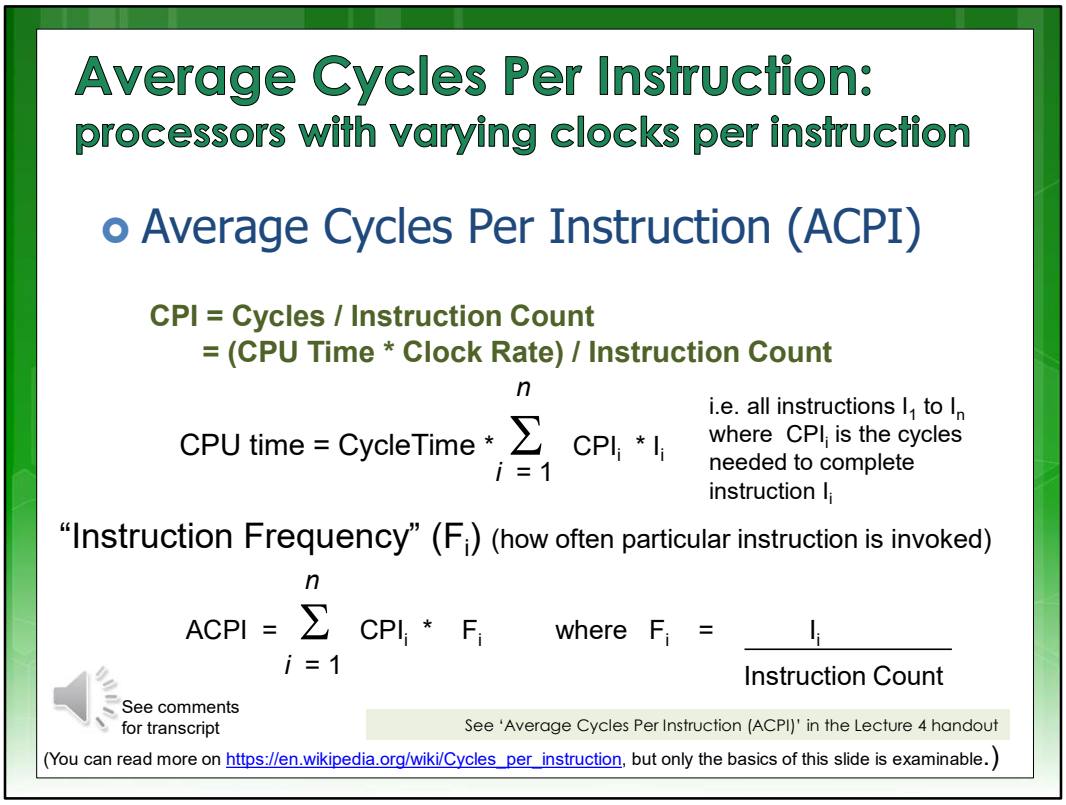
Which processor has better performance?

You probably see the issue… It depends on the aspect, or performance, of processing that you are interested in.

You may be more interested in how fast it runs a particular program.

Or how many instructions a program takes.

The table here indicates on the left the aspect that you may be interested in. The other columns indicates commonly available properties of a processor (e.g. found in the datasheet). The ticks indicate which aspect on the right is a default (or one or the first things) considered in rating that aspect.



Audio Transcript: Average Cycles Per Instruction (ACPI)

Average Cycles Per Instruction, or the **ACPI** (sometimes referred to as **CPI**) of a **CPU** is a term that has been used in many papers and data sheets. It is perhaps not used as extensively in scientific articles as it was in the past. That is because there is such a diversity of computing solutions out there for which this expression is not so relevant.

The Role of Modern Hardware

For example, an **FPGA** might grab some data, update a running average, and turn on some **LEDs** all in one clock cycle. **ACPI** is more useful for comparing processors that run similar types of instructions. For example, you might be comparing two **CPUs** to run a certain application:

Processor 1: No pipelining, with an **ACPI of 1**.

Processor 2: A classic **RISC** with a five-stage pipeline and an **ACPI** of, say, **4**.

Pipelining and Clock Speed

A lower **ACPI** may seem better at face value, but is it?

The first processor might be clocked at **10 MHz**, a slower clock to account for the maximum propagation delays in the **CPU** circuit.

The second processor, having its circuit split into pipeline sections that run concurrently, might support **100 MHz**.

Now it's not so clear which one would win a processing race. Probably the second processor, because **100 divided by the ACPI of 4** is **25 million instructions per second**, which is **2.5 times better** than the first. However, if you optimize your code, the second pipelined processor might run closer to **100 million instructions per second** if the pipeline is kept full and there are no stalls—meaning in each clock, one instruction is coming in and one instruction is completing. That is a speedup closer to **10**.

Calculating ACPI

The **ACPI** depends on a processor architecture, usually in relation to a program to benchmark. This typically concerns performance measurements for a pipelined architecture. It is pretty obvious that a non-pipelined program is going to have a **CPI of 1** regardless of the program it runs; its **ACPI** would be **1** as well.

To work out the **CPI** for a processor given a particular program, the essential formulas are used. The overall **ACPI** is the sum of the cycles per instruction multiplied by the frequency of that instruction for

each instruction that appears in the program concerned. This gives out a quantity of the **ACPI** for the given program.

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:
  - **$\Sigma(T_i)^{\wedge}1/n$**
  - 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

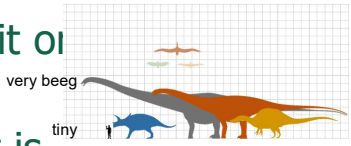
Cost/Performance:
The Relationship of Cost to Price?

- 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

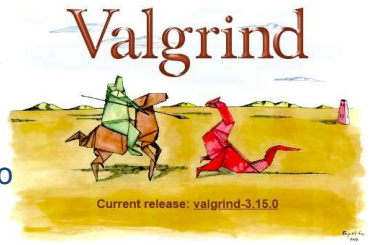
Prescribed: read on your own

- 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/

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)