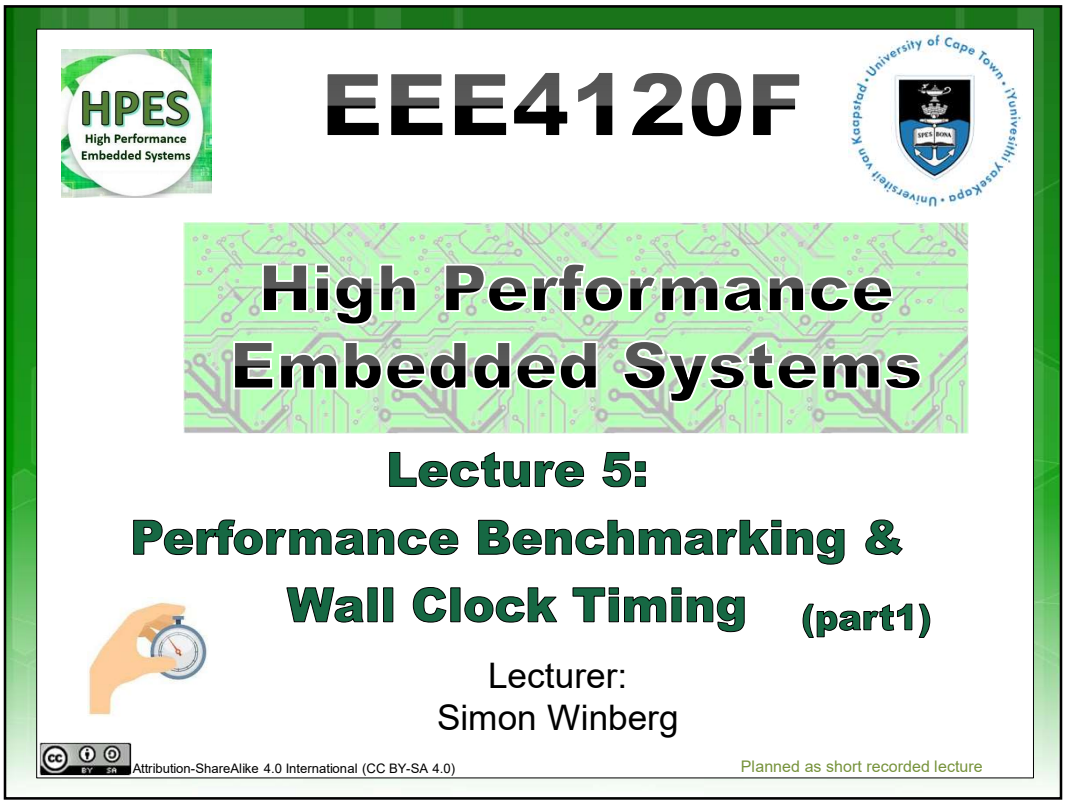


These slides have notes and explanations added in the comments. You want to view this as either handout notes pages or using presenter view if viewing the pptx in slideshow mode.

Slide 1 notes:

In this lecture, we're going to cover essential methods of performance benchmarking, profiling and testing of HPES systems. The aspects presented in these slides are largely aligned to software and multiprocessor systems; although the more general principles can be adapted to hardware accelerators or special purpose processor design and measurement of their performance.

--- Note on Voice Annotations---
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.
Image source: https://www.vectorportal.com

Outline for Lecture

- Towards performance benchmarking
- Simple benchmarking techniques

First covering broader topics of benchmarking and then going into simpler software benchmarking techniques.

Performance Benchmarking

EEE4120F HPES

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…

It is certainly desirable for a system to have a fast and efficient setup, so that you can start using it quickly or it can start doing what it needs to do without having to wait for a long while for it to start working. Parallelism is not always the best approach for marking a system that has a fast startup.

Important questions

- Should you bother to design a parallel algorithm?
- Is your parallel solution better than a simpler approach, especially if that approach is easier to read and share?
- Major telling factor is:

Real-time performance measure

Or “wall clock time”

One of the more common measures is the real-time performance measure (not just simulation-based and modelled performance). Where you determine your systems speed performance based on the start and stop times. The start and stop times could be of multiple different aspects, start-stop of the entire system (does it take ages to shutdown?), start-stop of processing jobs, start-stop of interrupt responses, etc.

(PS: I left this audio annotation in, although it sounds rather incomplete in explaining this slide)

Benchmarking

- Process of measuring performance of a digital system
- A program (or systematic approach) that Quantitatively evaluates performance, cost and computer hardware and software resources (among other things) of a computing solution
- **Benchmark suites** – sets of benchmark programs designed to get a comprehensive view of the performance of a computer system for executing a variety of representative processing operations.
- Suitable benchmark
  - A meaningful representation of what the system can do
  - Helps select of an effective system
  - Indicates a measure of what one system can do compared to other options

Benchmarking is the process of measuring the performance of a digital system by quantitatively evaluating measures such as: cost, speed, computer and software resources.

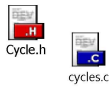
For doing benchmarking, “benchmark suites” are often used. These are programs, or a collection of programs, that are designed to get a comprehensive view of the performance of the computer system for executing a variety of representative processing operations.

PS: These are useful tips to consider for the YODA course project, as well as the pracs leading to that.

Wall clock time

- Generally the most accurate: use a built-in timer, which is directly related to real time (e.g., if the timer measures 1s, then 1s elapsed in the real world)
- Technique:

See file:
Cycle.h
Cycles.c

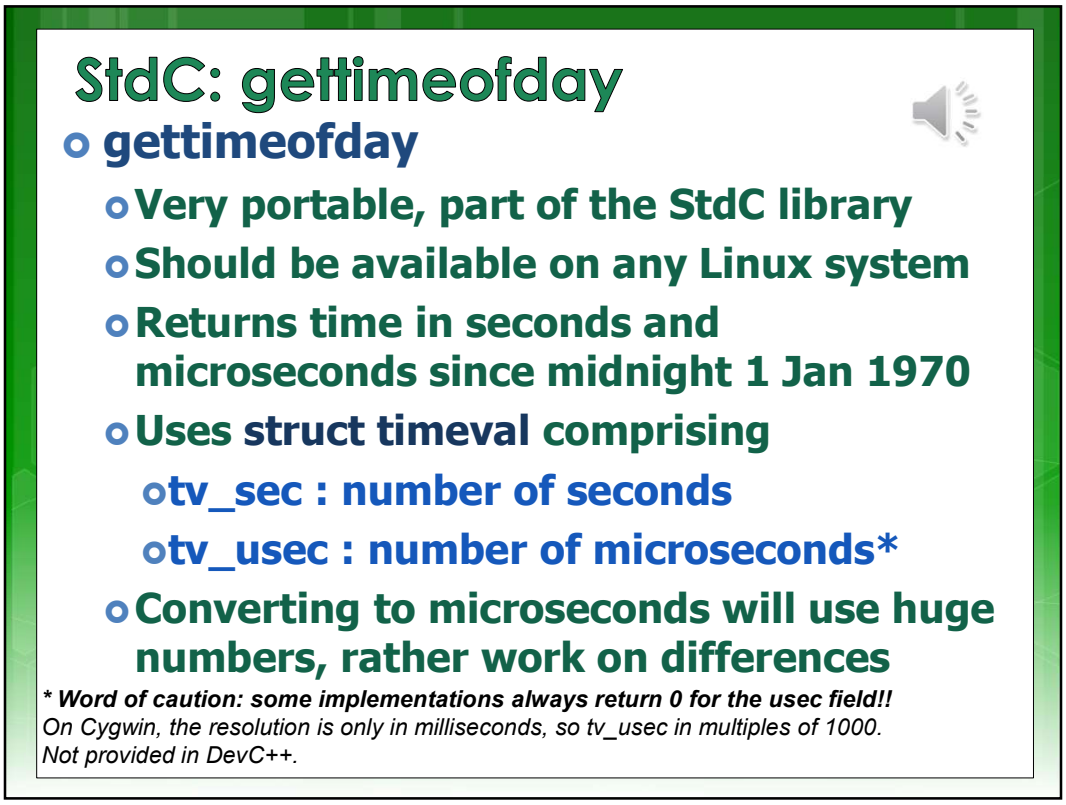



```
unsigned long long start; // store start time
unsigned long long end;   // store end time
start = read_the_timer(); // start timer / tic
 DO PROCESSNG
end = read_the_timer();   // end timer / toc
.. Output the time measurement (end-start), or save
it to an array if printing will interfere with the
times. Note: to avoid overflow, used unsigned vars.
```

An accurate approach to determine a system's speed performance is to utilize a built-in timer that is directly related to the real time.

Sometimes you may want to get the timing as accurate as possible (in terms of what speed the computer can run at), and a means to do so, if the processor supports that, is to just use the processor clock. Or, more correctly, a counter that is added for each clock period (i.e. the counter is clocked by either a positive or negative edge of the processor clock). Thus, if you have a 3GHz clock driving your processors, you can get extremely accurate timing at that frequency (i.e. at an accuracy of 1/3 ns). That is exceedingly accurate timing. It is probably quite excessive for most tasks. Like does it really matter if the delay between different video frames is e.g. 50ms or 50.001ms? Not all architectures have a cycles counter linked to the processor clock. Intel processors generally do… hence the example provided here, which can be used for most intel processors (at least seems to work for i3 – i7).

Generally, you would surely want to use a more portable code for reading a timer. (You hopefully know that portable implies: using the same code usable on multiple platforms without needing to change the code (much)). See next slide.



The gettimeofday is a highly portable command, it is part of the STDC library, so should be available for any C / C++ compiler.

Although…. it doesn't always work entirely consistently. For instance, it seems Cygwin doesn't bother to give anything more accurate than milliseconds (it just leaves 0 for us and ns).

You may want to use this routine in your OpenCL and/or MPI applications.

See **timing.c** code file

gettimeofday example

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
struct timeval start_time, end_time; // variables to hold start and end time

int main()  {
   int tot_usecs;
   int i,j,sum=0;
   gettimeofday(&start_time, (struct timezone*)0); // starting timestamp

   /*   do some work */
   for (i=0; i<10000; i++)
      for (j=0; j<i; j++) sum += i*j;

   gettimeofday(&end_time, (struct timezone*)0);  // ending timestamp

   tot_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
               (end_time.tv_usec-start_time.tv_usec);
   printf("Total time: %d usec.\n", tot_usecs);
}
```

Here is an example of the get time of day function being used in a speed test.

What is wrong about using only wall clock time?

- It can provide a false impression of how effective your solution is – at least doesn't give a 'full picture' …
  - Typically do tests after the system has 'warmed up'* (cache loaded) by running the same data multiple times
  - May show the solution is quicker… but at what costs? e.g.:
    - Speed improved but accuracy sacrificed?
    - Development effort vs. execution speed improvement?
    - Resource costs for upgrading vs. costs saved by remaining with the old version?
    - Power usage? Does the new solution need more power (per execution, also on average including idle time)
    - Maintainability? (e.g. is the new version more complex?)
    - Environment impact? (Does the upgrade result in waste that could be environmentally detrimental)

* But this can be a very false impression too, e.g. cache etc pre-set with needed data.

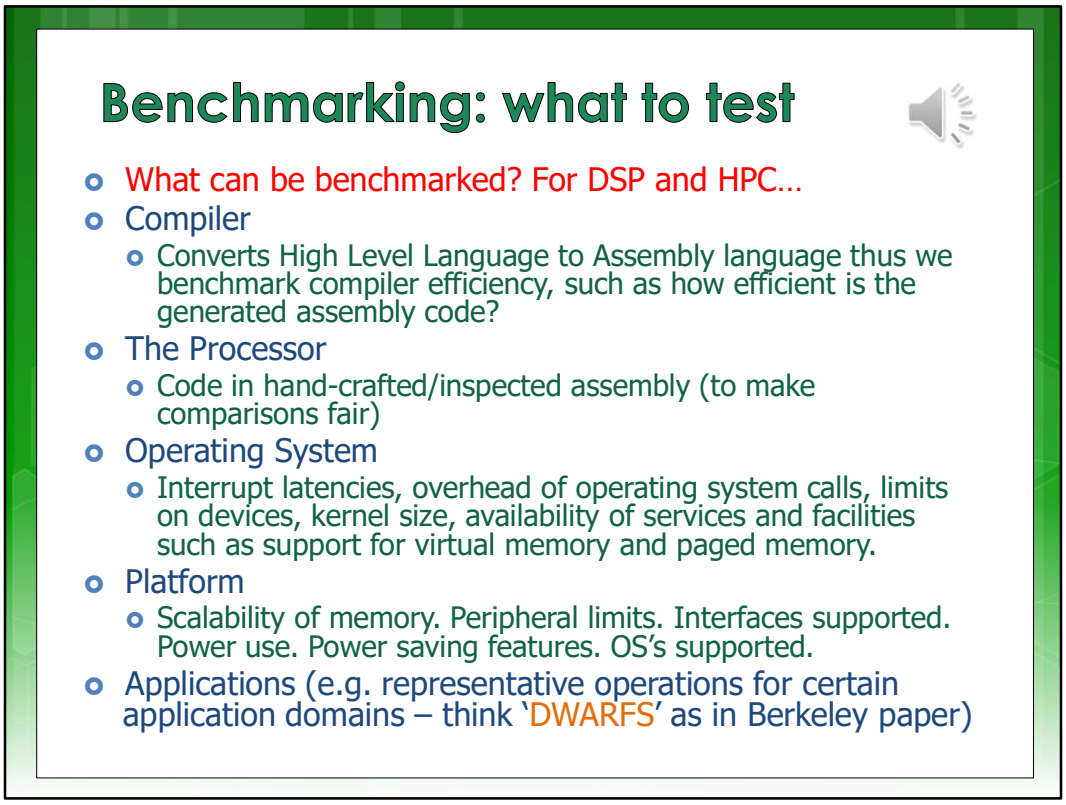
The issue with the wall clock time approach is that it can be inaccurate. It may provide a false impression of how effective the solution is. Typically you want to test your system only once the system has warmed up. This refers to only testing after the program has been run a couple of times (i.e. has been 'warmed-up').

Note that sometimes you do want to test the system only on a cold start. Before the cache and memories get warmed up. This is more accurate for transaction or real-time stream data processing. Otherwise, it gives a false impression of how the system would perform if the same data keeps getting sent to it.

You should thus say if your testing is based on a cold

start or a warmed-up case.

Furthermore, this test does not evaluate the accuracy, development effort, cost, power usage, maintainability and environmental impact of the solution. Therefore. A better wall clock time test does not mean a better system.



So the question is: What should be benchmarked?

And that depends on your system. And what you have control of, and what you are doing with your system.

For a compiler, the speed of the compiler is less important. But the efficiency of the generated assembly code is *very* important. Therefore, in such a case, your benchmarking should focus on the efficiency of the generated assembly code.

Another good example is an operating system. It would probably have numerous tasks, that might need to respond differently in different cases. Here, there may be many aspects that should be benchmarked. For instance. Where speed is important, it may be looking at interrupt latencies. But where memory efficiency is desired, it may be looking at how data management methods are avoiding problems of wasted space (e.g. reducing heap fragmentation).

From this it is clear to see that the benchmarking is very dependent on the system, and its purpose.

Typical things to benchmark

- Of course there's much that can be benchmarked, but for this course we can mainly consider:
  - Cycle count,
  - Memory usage,
  - Execution time, and
  - Power consumption

For digital signal processing and high-performance embedded systems, things typically benchmarked include: cycle count, memory usage, execution time, and power consumption.

You might want to choose some of those to look into for your YODA project ;-)

Next Lecture …

- We get more into depth of
  - Metrics for performance
  - Some specialized concepts (e.g. the 'ACPI' measure for a processor core)
  - Methods to summarising performance (commonly seen in performance reports)
  - SWAP
  - Profiling code designs*

* Only a brief flavour of profiling techniques, would need to be a course on its own to do properly.

This lecture was aimed to provide you an initial flavour of benchmarking. In the next lecture, the aim is to get into more depth of performance analysis and benchmarking. Accordingly, we will look at: metrics of performance. Some specialized concepts. Such as the A C P I. Methods to summarize performance (such as average performance). The S W A P concept. And a view on code profiling methods.

closing
remarks & reminders…

Have a look at:

Valgrind About Page

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

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