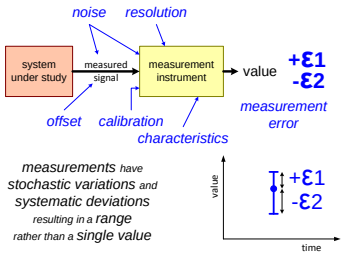
Modeling and Analysis: Measuring

-



Gerrit Muller
University of South-Eastern Norway-NISE
Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

This presentation addresses the fundamentals of measuring: What and how to measure, impact of context and experiment on measurement, measurement errors, validation of the result against expectations, and analysis of variation and credibility.

version: 1.2 status: preliminary draft September 9, 2018

1 introduction

Measurements are used to calibrate and to validate models. Measuring is a specific knowledge area and skill set. Some educations, such as Physics, extensively teach experimentation. Unfortunately, the curriculum of studies such as software engineering and computer sciences has abstracted away from this aspect. In this paper we will address the fundamentals of modeling.

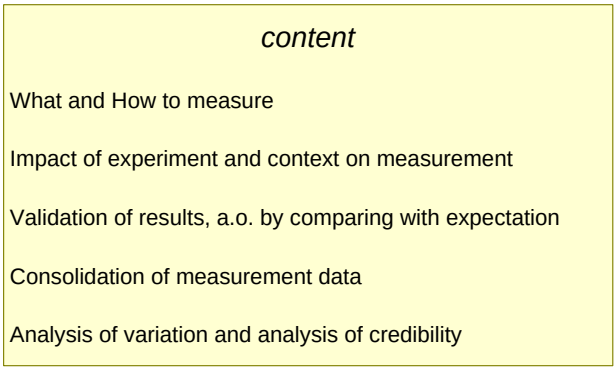


Figure 1: Presentation Content

Figure 1 shows the content of this paper. The crucial aspects of measuring are integrated into a measuring approach, see the next section.

2 Measuring Approach

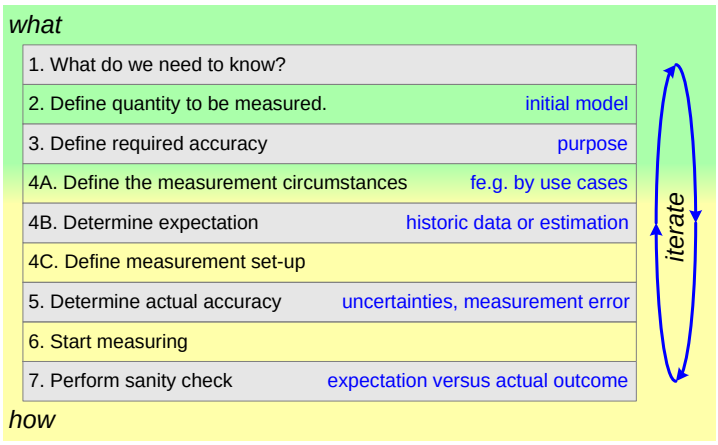


Figure 2: Measuring Approach: What and How

The measurement approach starts with *preparation and fact finding* and ends with *measurement and sanity check*. Figure 2 shows all steps and emphasizes the need for iteration over these steps.

1. What do we need? What is the problem to be addressed, so what do we need to know?

2. Define quantity to be measured Articulate as sharp as possible what quantity needs to be measured. Often we need to create a mental model to define this quantity.

3. Define required accuracy The required accuracy is based on the problem to be addressed and the purpose of the measurement.

4A. Define the measurement circumstances The system context, for instance the amount of concurrent jobs, has a big impact on the result. This is a further elaboration of step 1 *What do we need?*.

4B. Determine expectation The experimentator needs to have an expectation of the quantity to be emasured to design the experiment and to be able to assess the outcome.

4C. Define measurement set-up The actual design of the experiment, from input stimuli, measurement equipment to outputs.

Note that the steps 4A, 4B and 4C mutually influence each other.

5. **Determine actual accuracy** When the set-up is known, then the potential measurement errors and uncertainties can be analyzed and accumulated into a total actual accuracy.

6. **Start measuring** Perform the experiment. In practice this step has to be repated many times to "debug" the experiment.

7. **Perform sanity check** Does the measurement result makes sense? Is the result close to the expectation?

In the next subsections we will elaborate this approach further and illustrate the approach by measuring a typical embedded controller platform: ARM9 and VxWorks.

2.1 What do we need?

The first question is: "What is the problem to be addressed, so what do we need to know?" Figure 3 provides an example. The *problem* is the need for guidance for *concurrency design* and *task granularity*. Based on experience the designers know that these aspects tend to go wrong. The effect of poor *concurrency design* and *task granularity* is poor performance or outrageous resource consumption.

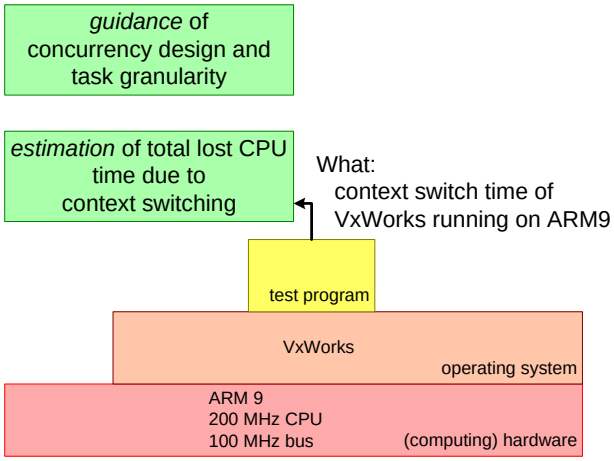


Figure 3: What do We Need? Example Context Switching

The designers know, also based on experience, that *context switching* is costly and critical. They have a need to estimate the total amount of CPU time lost due to context switching. One of the inputs needed for this estimation is the cost in CPU time of a single context switch. This cost is a function of the hardware platform, the operating system and the circumstances. The example in Figure 3 is based on

the following hardware: ARM9 CPU running internally at 200 MHz and externally at 100 MHz. The operating system is VxWorks. VxWorks is a real-time executive frequently used in embedded systems.

2.2 Define quantity to be measured.

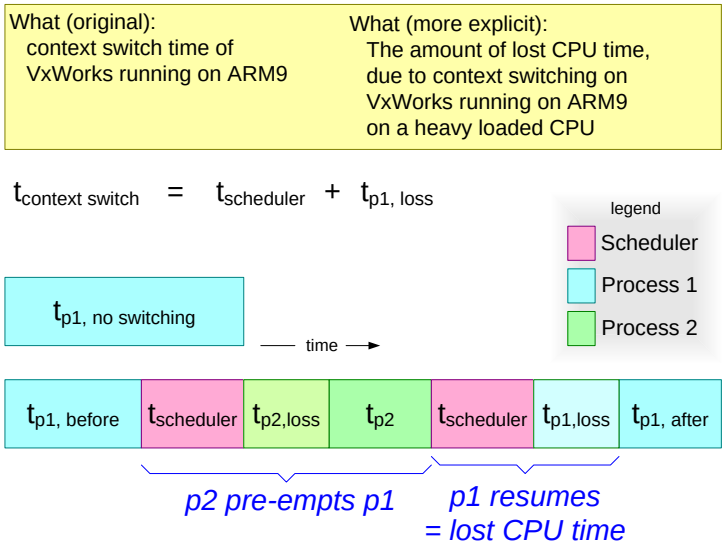


Figure 4: Define Quantity by Initial Model

As need we have defined the CPU cost of context switching. Before setting up measurements we have to explore the required quantity some more so that we can define the quantity more explicit. In the previous subsection we already mentioned shortly that the context switching time depends on the circumstances. The a priori knowledge of the designer is that context switching is especially significant in busy systems. Lots of activities are running concurrently, with different periods and priorities.

Figure 4 defines the quantity to be measured as the total cost of context switching. This total cost is not only the overhead cost of the context switch itself and the related administration, but also the negative impact on the cache performance. In this case the a priori knowledge of the designer is that a context switch causes additional cache loads (and hence also cache pollution). This cache effect is the term $t_{p1,loss}$ in Figure 4. Note that these effects are not present in a lightly loaded system that may completely run from cache.

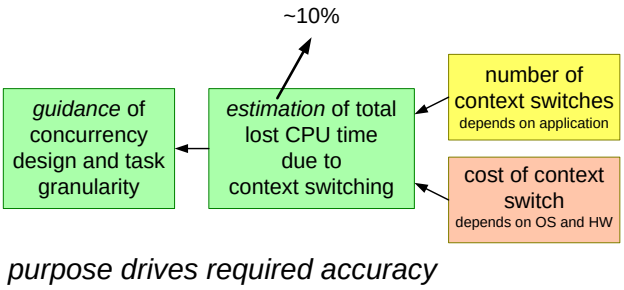


Figure 5: Define Required Accuracy

2.3 Define required accuracy

The required accuracy of the measurement is determined by the need we originally formulated. In this example the need is the ability to *estimate* the total lost CPU time due to context switching. The key word here is *estimate*. Estimations don't require the highest accuracy, we are more interested in the order of magnitude. If we can estimate the CPU time with an accuracy of tens of percents, then we have useful facts for further analysis of for instance task granularity.

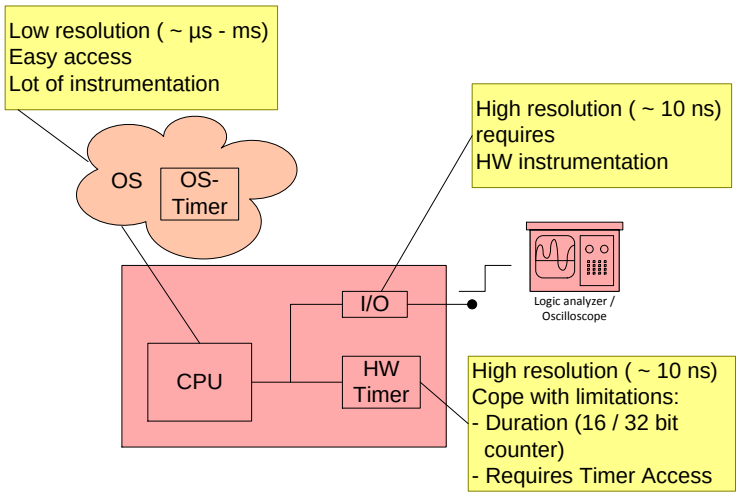


Figure 6: How to Measure CPU Time?

The relevance of the required accuracy is shown by looking at available measurement instruments. Figure 6 shows a few alternatives for measuring time on this type of platforms. The most easy variants use the instrumentation provided by the operating system. Unfortunately, the accuracy of the operating system timing is often very limited. Large operating systems, such as Windows and Linux, often

provide 50 to 100 Hz timers. The timing resolution is then 10 to 20 milliseconds. More dedicated OS-timer services may provide a resolution of several microseconds. Hardware assisted measurements make use of hardware timers or logic analyzers. This hardware support increases the resolution to tens of nanoseconds.

2.4 Define the measurement circumstances

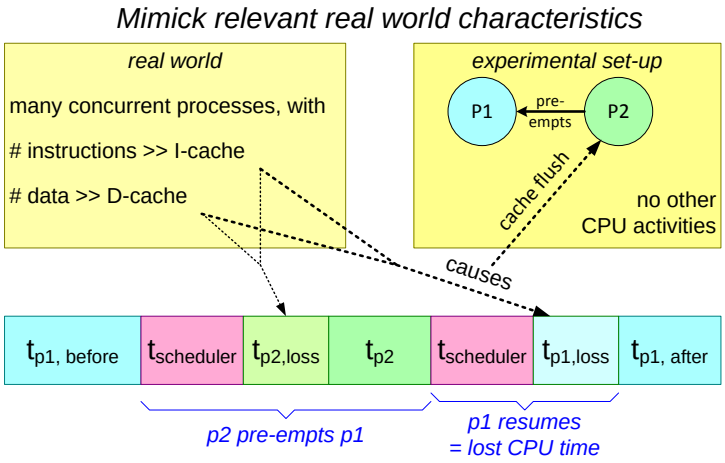


Figure 7: Define the Measurement Set-up

We have defined that we need to know the context switching time under *heavy load* conditions. In the final application *heavy load* means that we have lots of cache activity from both instruction and data activities. When a context switch occurs the most likely effect is that the process to be run is not in the cache. We lose time to get the process back in cache.

Figure 7 shows that we are going to mimick this cache behavior by flushing the cache in the small test processes. The overall set-up is that we create two small processes that alternate running: Process $P2$ pre-empts process $P1$ over and over.

2.5 Determine expectation

Determining the expected outcome of the measurement is rather challenging. We need to create a simple model of the context switch running on this platform. Figures 8 and 9 provide a simple hardware model. Figure 10 provides a simple software model. The hardware and software models are combined in Figure 11. After substitution with assumed numbers we get a number for the expected outcome, see Figure12.

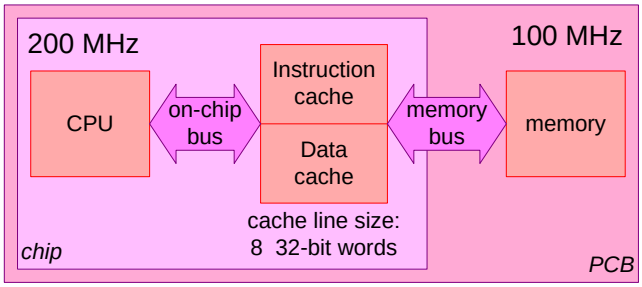


Figure 8: Case: ARM9 Hardware Block Diagram

Figure 8 shows the hardware block diagram of the ARM9. A typical chip based on the ARM9 architecture has anno 2006 a clock-speed of 200 MHz. The memory is off-chip standard DRAM. The CPU chip has on-chip cache memories for instruction and data, because of the long latencies of the off-chip memory access. The memory bus is often slower than the CPU speed, anno 2006 typically 100 MHz.

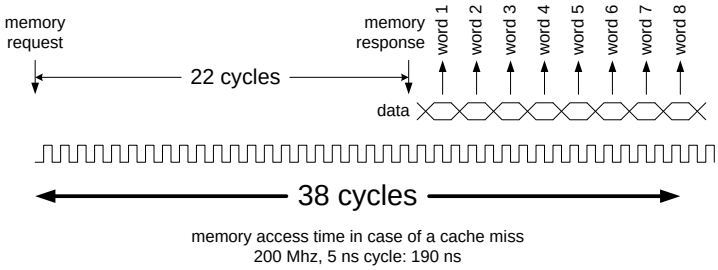


Figure 9: Key Hardware Performance Aspect

Figure 9 shows more detailed timing of the memory accesses. After 22 CPU cycles the memory responds with the first word of a memory read request. Normally an entire cache line is read, consisting of 8 32-bit words. Every word takes 2 CPU cycles = 1 bus cycle. So after $22 + 8 * 2 = 38$ cycles the cache-line is loaded in the CPU.

Figure 10 shows the fundamental scheduling concepts in operating systems. For context switching the most relevant process states are *ready*, *running* and *waiting*. A context switch results in state changes of two processes and hence in scheduling and administration overhead for these two processes.

Figure 11 elaborates the software part of context switching in five contributing activities:

- save state P1
- determine next runnable task

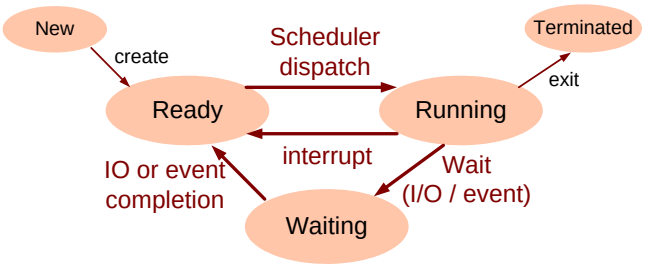


Figure 10: OS Process Scheduling Concepts

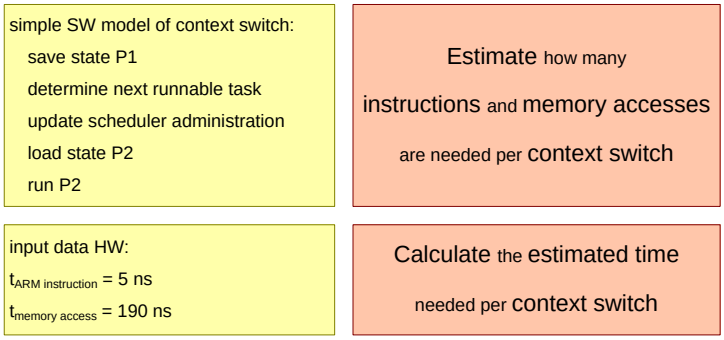


Figure 11: Determine Expectation

- update scheduler administration
- load state P2
- run P2

The cost of these 5 operations depend mostly on 2 hardware depending parameters: the numbers of instruction needed for each activity and the amount of memory accesses per activity. From the hardware models, Figure 9, we know that as simplest approximation gives us an instruction time of $5ns$ (= 1 cycle at 200 MHz) and memory accesses of $190ns$. Combining all this data together allows us to estimate the context switch time.

In Figure 12 we have substituted estimated number of instructions and memory accesses for the 5 operations. The assumption is that very simple operations require 10 instructions, while the somewhat more complicated scheduling operation requires scanning some data structure, assumed to take 50 cycles here. The estimation is now reduced to a simple set of multipications and additions: $(10 + 50 + 20 + 10 + 10)instructions \cdot 5ns + (1 + 2 + 1 + 1 + 1)memoryaccesses \cdot 190ns = 500ns(instructions) + 1140ns(memoryaccesses) = 1640ns$ To add some

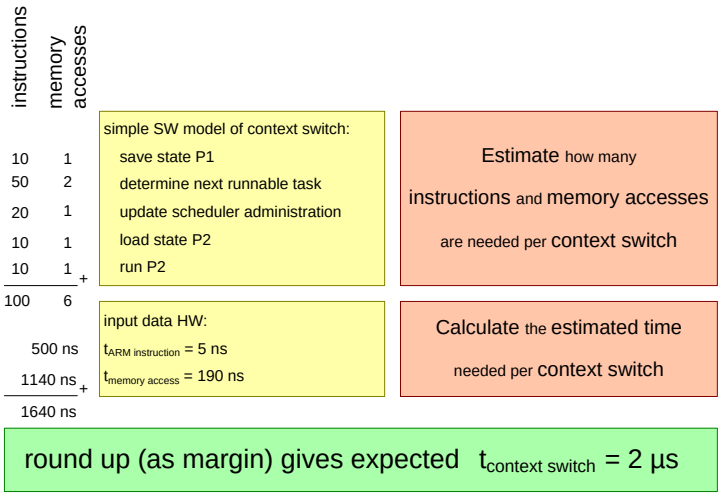


Figure 12: Determine Expectation Quantified

margin for unknown activities we round this value to $2\mu s$.

2.6 Define measurement set-up

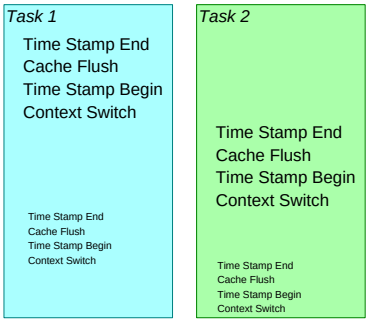


Figure 13: Code to Measure Context Switch

Figure 13 shows pseudo code to create two alternating processes. In this code time stamps are generated just before and after the context switch. In the process itself a cache flush is forced to mimick the loaded situation.

Figure 14 shows the CPU use as function of time for the two processes and the scheduler.

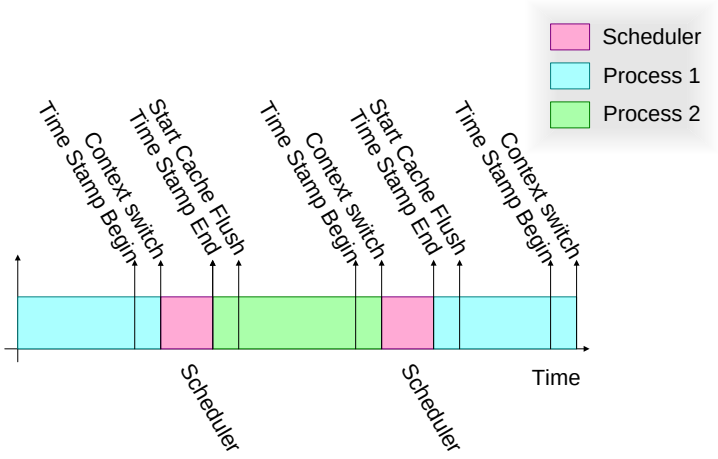


Figure 14: Measuring Context Switch Time

2.7 Expectation revisited

Once we have defined the measurement set-up we can again reason more about the expected outcome. Figure 15 is again the CPU activity as function of time. However, at the vertical axis the CPI (Clock cycles Per Instruction) is shown. The CPI is an indicator showing the effectiveness of the cache. If the CPI is close to 1, then the cache is rather effective. In this case little or no main memory acceses are needed, so the CPU does not have to wait for the memory. When the CPU has to wait for memory, then the CPI gets higher. This increase is caused by the waiting cycles necessary for the main memory accesses.

Figure 15 clearly shows that every change from the execution flow increases (worsens) the CPI. So the CPU is slowed down when entering the scheduler. The CPI decreases while the scheduler is executing, because code and data gets more and more from cache instead of main memory. When Process 2 is activitated the CPI again worsens and then starts to improve again. This pattern repeats itself for every discontinuity of the program flow. In other words we see this effect twice for one context switch. One interruption of $P1$ by $P2$ causes two context swicthes and hence four dips of the cache performance.

2.8 Determine actual accuracy

Measurement results are in principle a range instead of a single value. The signal to be measured contains some noise and may have some offset. Also the measurement instrument may add some noise and offset. Note that this is not limited to the analog world. For instance concurrent background activities may cause noise as well as offsets, when using bigger operating systems such as Windows or Linux.

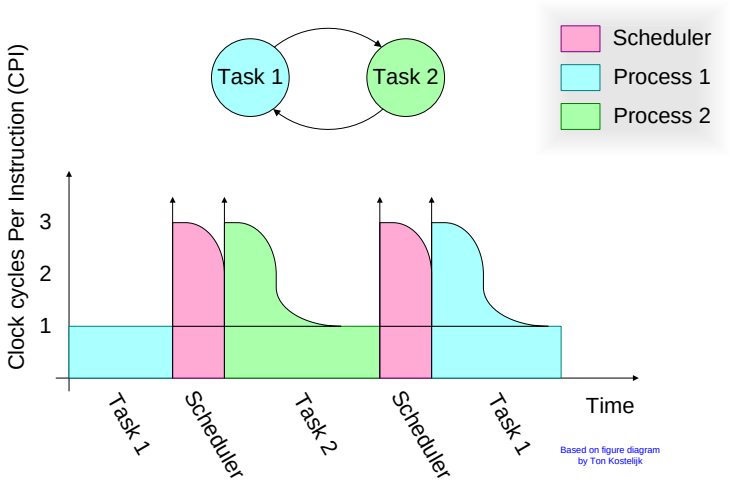


Figure 15: Understanding: Impact of Context Switch

The (limited) resolution of the instrument also causes a measurement error. Known systematic effects, such as a constant delay due to background processes, can be removed by calibration. Such a calibration itself causes a new, hopefully smaller, contribution to the measurement error.

Note that contributions to the measurement error can be stochatic, such as noise, or systematic, such as offsets. Error accumulation works differently for stochatic or systematic contributions: stochatic errors can be accumulated quadratic $\varepsilon_{total} = \sqrt{\varepsilon_1^2 + \varepsilon_2^2}$, while systematic errors are accumulated linear $\varepsilon_{total} = \varepsilon_1 + \varepsilon_2$.

Figure 17 shows the effect of error propagation. Special attention should be paid to substraction of measurement results, because the values are substracted while the errors are added. If we do a single measurement, as shown earlier in Figure 13, then we get both a start and end value with a measurement error. Substracting these values adds the errors. In Figure 17 the provided values result in $t_{duration} = 4 + / - 4\mu s$. In other words when substracted values are close to zero then the error can become very large in relative terms.

The whole notion of measurement values and error ranges is more general than the measurement sec. Especially models also work with ranges, rather than single values. Input values to the models have uncertainties, errors et cetera that propagate through the model. The way of propagation depends also on the nature of the error: stochastic or systematic. This insight is captured in Figure 18.

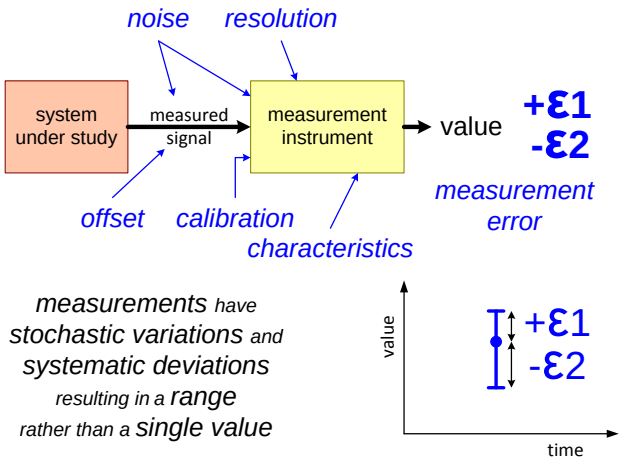


Figure 16: Accuracy: Measurement Error

$t_{duration} = t_{end} - t_{start}$

$t_{start} = 10 +/- 2\ \mu s$

$t_{end} = 14 +/- 2\ \mu s$

$t_{duration} = 4 +/- ?\ \mu s$

systematic errors: add linear

stochastic errors: add quadratic

Figure 17: Accuracy 2: Be Aware of Error Propagation

2.9 Start measuring

At OS level a micro-benchmark was performed to determine the context switch time of a real-time executive on this hardware platform. The measurement results are shown in Figure 19. The measurements were done under different conditions. The most optimal time is obtained by simply triggering continuous context switches, without any other activity taking place. The effect is that the context switch runs entirely from cache, resulting in a $2\mu s$ context switch time. Unfortunately, this is a highly misleading number, because in most real-world applications many activities are running on a CPU. The interrupting context switch pollutes the cache, which slows down the context switch itself, but it also slows down the interrupted activity. This effect can be simulated by forcing a cache flush in the context switch. The performance of the context switch with cache flush degrades to $10\mu s$. For comparison the measurement is also repeated with a disabled cache, which decreases the context switch even more to $50\mu s$. These measurements show

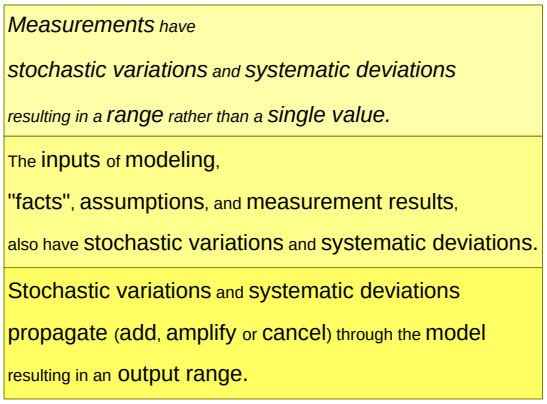


Figure 18: Intermezzo Modeling Accuracy

ARM9 200 MHz $t_{context\ switch}$
as function of cache use

cache setting	$t_{context\ switch}$
From cache	2 µs
After cache flush	10 µs
Cache disabled	50 µs

Figure 19: Actual ARM Figures

the importance of the cache for the CPU load. In cache unfriendly situations (a cache flushed context switch) the CPU performance is still a factor 5 better than in the situation with a disabled cache. One reason of this improvement is the locality of instructions. For 8 consecutive instructions ”only” 38 cycles are needed to load these 8 words. In case of a disabled cache $8 * (22 + 2 * 1) = 192$ cycles are needed to load the same 8 words.

We did estimate $2\mu s$ for the context switch time, however already taking into account negative cache effects. The expectation is a factor 5 more optimistics than the measurement. In practice expectations from scratch often deviate a factor from reality, depending on the degree of optimism or conservatism of the estimator. The challenging question is: Do we trust the measurement? If we can provide a credible explanation of the difference, then the credibility of the measurement increases.

In Figure 20 some potential missing contributions in the original estimate are presented. The original estimate assumes single cycle instruction fetches, which is not true if the instruction code is not in the instruction cache. The Memory

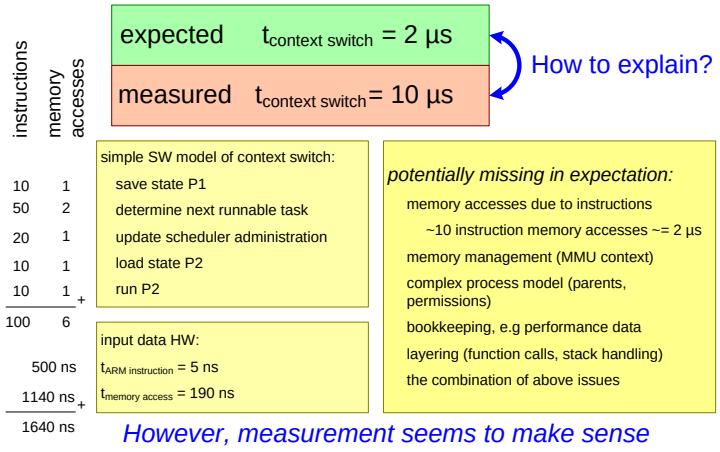


Figure 20: Expectation versus Measurement

Management Unit (MMU) might be part of the process context, causing more state information to be saved and restored. Often may small management activities take place in the kernel. For example, the process model might be more complex than assumed, with process hierarchy and permissions. May be hierarchy or permissions are accessed for some reasons, may be some additional state information is saved and restored. Bookkeeping information, for example performance counters, can be maintained. If these activities are decomposed in layers and components, then additional function calls and related stack handling for parameter transfers takes place. Note that all these activities can be present as combination. This combination not only cummulates, but might also multiply.

$$t_{overhead} = n_{context\ switch} * t_{context\ switch}$$

$n_{context\ switch}$ (s^{-1})	$t_{context\ switch}$ = 10µs		$t_{context\ switch}$ = 2µs	
	$t_{overhead}$	CPU load overhead	$t_{overhead}$	CPU load overhead
500	5ms	0.5%	1ms	0.1%
5000	50ms	5%	10ms	1%
50000	500ms	50%	100ms	10%

Figure 21: Context Switch Overhead

Figure 21 integrates the amount of context switching time over time. This figure shows the impact of context switches on system performance for different context switch rates. Both parameters $t_{contextswitch}$ and $n_{contextswitch}$ can easily be measured and are quite indicative for system performance and overhead induced by design choices. The table shows that for the realistic number of $t_{contextswitch} = 10\mu s$ the number of context switches can be ignored with 500 context switches per second, it becomes significant for a rate of 5000 per second, while 50000 context switches per second consumes half of the available CPU power. A design based on the too optimistic $t_{contextswitch} = 2\mu s$ would assess 50000 context switches as significant, but not yet problematic.

2.10 Perform sanity check

In the previous subsection the actual measurement result of a single context switch including cache flush was $10\mu s$. Our expected result was in the order of magnitude of $2\mu s$. The difference is significant, but the order of magnitude is comparable. In geenral this means that we do not completely understand our system nor our measurement. The value is usable, but we should be alert on the fact that our measurement still introduces some additional systematic time. Or the operating system might do more than we are aware of.

One approach that can be taken is to do a completely different measurement and estimation. For instance by measuring the idle time, the remaining CPU time that is avaliable after we have done the real work plus the overhead activities. If we also can measure the time needed for the real work, then we have a different way to estimate th overhead, but now averaged over a longer period.

2.11 Summary of measuring Context Switch time on ARM9

We have shown in this example that the goal of measurement of the ARM9 VxWorks combination was to provide guidance for concurrency design and task granularity. For that purpose we need an estimation of context switching overhead.

We provided examples of measurement, where we needed context switch overhead of about 10% accuracy. For this measurement the instrumentation used toggling of a HW pin in combination with small SW test program. We also provided simple models of HW and SW layers to be able to determine an expectation. Finally we found as measurement results for context switching on ARM9 a value of $10\mu s$.

3 Summary

Figure 22 summarizes the measurement approach and insights.

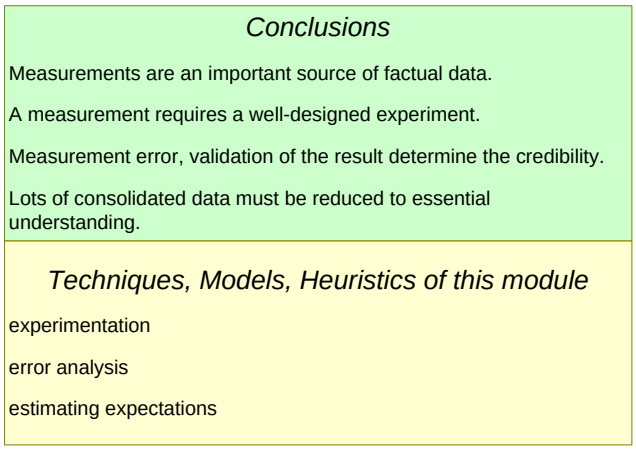


Figure 22: Summary Measuring Approach

4 Acknowledgements

This work is derived from the EXARCH course at CTT developed by Ton Kostelijk (Philips) and Gerrit Muller. The Boderc project contributed to the measurement approach. Especially the work of Peter van den Bosch (Océ), Oana Florescu (TU/e), and Marcel Verhoef (Chess) has been valuable. Teun Hendriks provided feedback, based on teaching the Architecting System Performance course.

References

[1] Gerrit Muller. The system architecture homepage. `http://www.gaudisite.nl/index.html`, 1999.

History

Version: 1.2, date: 29 October, 2007 changed by: Gerrit Muller

- added step numbers to the slides

Version: 1.1, date: 28 March, 2007 changed by: Gerrit Muller

- added discussion of measurement versus expectation

Version: 1.0, date: 28 November 2006 changed by: Gerrit Muller

- added text
- changed the order of slides
- added colophon

Version: 0.2, date: 17 November 2006 changed by: Gerrit Muller

- many minor improvements
- changed status to preliminary draft

Version: 0.1, date: 14 November 2006 changed by: Gerrit Muller

- Added more theory

Version: 0, date: 7 November 2006 changed by: Gerrit Muller

- Created, no changelog yet