Module Modeling and Analysis: Inputs and Uncertainties

Gerrit Muller
Embedded Systems Institute
Den Dolech 2 (Laplace Building 0.10) P.O. Box 513, 5600 MB Eindhoven The Netherlands
gerrit.muller@embeddedsystems.nl

Abstract

This module addresses Modeling and Analysis: Inputs and Uncertainties. The input for models comes from different sources: facts obtained from market and technology research, data from measurements, and assumptions. All these sources have uncertainties and may hide unknowns, or may even be wrong. We zoom in on commonly used technology.

The complete course MA 611™ is owned by Embedded Systems Institute. To teach this course a license from Embedded Systems Institute is required. This material is preliminary course material. The final material and course information can be found at: www.esi.nl/cursus.

All Gaudí documents are available at:
http://www.gaudisite.nl/

version: 0.6 status: planned February 11, 2012

Contents

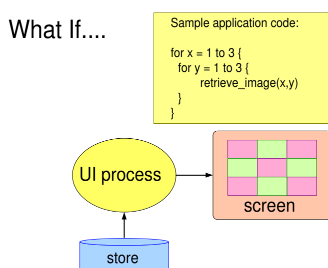
1 Introduction to System Performance Design **1**
1.1 Introduction 1
1.2 What if 1
1.3 Problem Statement 4
1.4 Summary 5
1.5 Acknowledgements 5

2 Modeling and Analysis Fundamentals of Technology **7**
2.1 Introduction 7
2.2 Computing Technology Figures of Merit 8
2.3 Caching in Web Shop Example 11
2.4 Summary 16

3 Modeling and Analysis: Measuring **17**
3.1 introduction 17
3.2 Measuring Approach 19
3.2.1 What do we need? 20
3.2.2 Define quantity to be measured. 21
3.2.3 Define required accuracy 22
3.2.4 Define the measurement circumstances 23
3.2.5 Determine expectation 23
3.2.6 Define measurement set-up 26
3.2.7 Expectation revisited 27
3.2.8 Determine actual accuracy 27
3.2.9 Start measuring 29
3.2.10 Perform sanity check 32
3.2.11 Summary of measuring Context Switch time on ARM9 . . 32
3.3 Summary 33
3.4 Acknowledgements 34

Chapter 1

Introduction to System Performance Design



1.1 Introduction

This article discusses a typical example of a performance problem during the creation of an additional function in an existing system context. We will use this example to formulate a problem statement. The problem statement is then used to identify ingredients to address the problem.

1.2 What if ...

Let's assume that the application asks for the display of $3 \cdot 3$ images to be displayed "instanteneously". The author of the requirements specification wants to sharpen this specification and asks for the expected performance of feasible solutions. For this purpose we assume a solution, for instance an image retrieval function with code that looks like the code in Figure 1.1. How do we predict or estimate the expected performance based on this code fragment?

If we want to estimate the performance we have to know what happens in the system in the retrieve_image function. We may have a simple system, as shown in

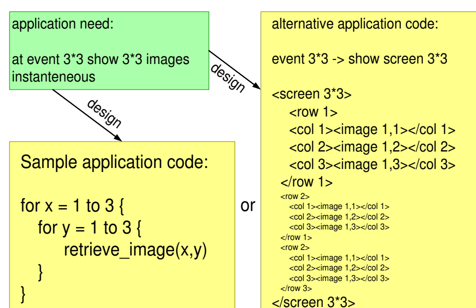


Figure 1.1: Image Retrieval Performance

Figure 1.2, where the retrieve_image function is part of a *user interface* process. This process reads image data directly form the hard disk based store and renders the image directly to the screen. Based on these assumptions we can estimate the performance. This estimation will be based on the disk transfer rate and the rendering rate.

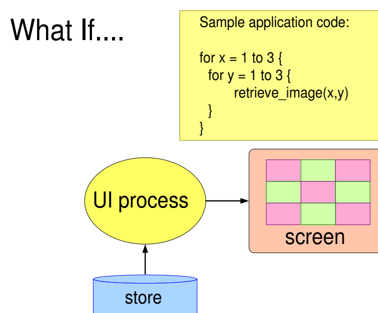


Figure 1.2: Straight Forward Read and Display

However, the system might be slightly more complex, as shown in Figure 1.3. Instead of one process we now have multiple processes involved: database, user interface process and screen server. Process communication becomes an additional contribution to the time needed for the image retrieval. If the process communication is image based (every call to retrieve_image triggers a database access and a transfer to the screen server) then $2 \cdot 9$ process communications takes place. Every process communication costs time due to overhead as well as due to copying image data from one process context to another process context. Also the database access will contribute to the total time. Database queries cost a significant amount of time.

The actual performance might be further negatively impacted by the overhead costs of the meta-information. Meta-information is the describing information of the image, typically tens to hundreds of attributes. The amount of data of meta-information, measured in bytes, is normally orders of magnitude smaller than the

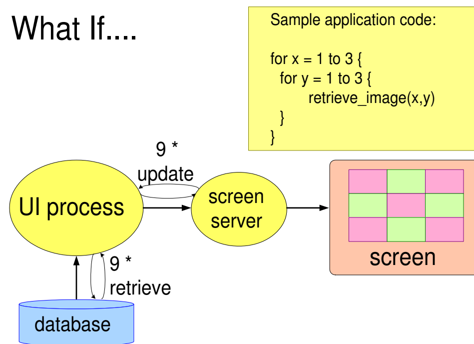


Figure 1.3: More Process Communication

amount of pixel data. The initial estimation ignores the cost of meta-information, because the of amount of data is insignificant. However, the chosen implementation does have a significant impact on the cost of meta-information handling. Figure 1.4 shows an example where the attributes of the meta-information are internally mapped on COM objects. The implementation causes a complete "factory" construction for every attribute that is retrieved. The cost of such a construction is $80\mu sec$. With 100 attributes per image we get a total construction overhead of $9 \cdot 100\ cdot 80\mu s = 72ms$. This cost is significant, because it is in the same order of magnitude as image transfer and rendering operations.

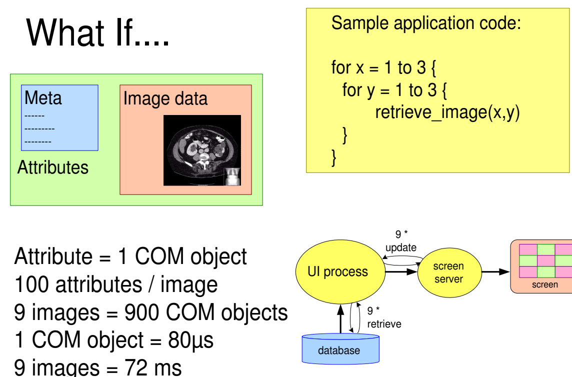


Figure 1.4: Meta Information Realization Overhead

Figure 1.5 shows I/O overhead as a last example of potential hidden costs. If the granularity of I/O transfers is rather fine, for instance based on image lines, then the I/O overhead becomes very significant. If we assume that images are 512^2, and if we assume $t_{I/O} = 1ms$, then the total overhead becomes $9 \cdot 512 \cdot 1ms \approx 4.5s$!

What If....

Sample application code:

```
for x = 1 to 3 {
  for y = 1 to 3 {
       retrieve_image(x,y)
  }
}
```

- I/O on line basis (512^2 image)

$9 * 512 * t_{I/O}$

$t_{I/O}$ ~= 1ms

- . . .

Figure 1.5: I/O overhead

1.3 Problem Statement

Sample application code:

```
for x = 1 to 3 {
  for y = 1 to 3 {
       retrieve_image(x,y)
  }
}
```

can be:
- fast, but very local
- slow, but very generic
- slow, but very robust
- fast and robust
- ...

The emerging properties (behavior, performance) cannot be seen from the code itself!

Underlying platform and neighbouring functions determine emerging properties mostly.

Figure 1.6: Non Functional Requirements Require System View

In the previous section we have shown that the performance of a new function cannot directly be derived from the code fragment belonging to this function. The performance depends on many design and implementation choices in the SW layers that are used. Figure 1.6 shows the conclusions based on the previous *What if* examples.

Figure 1.7 shows the factors outside our new function that have impact on the overall performance. All the layers used directly or indirectly by the function have impact, ranging from the hardware itself, up to middleware providing services. But also the neighboring functions that have no direct relation with our new function have impact on our function. Finally the environment including the user have impact on the performance.

Figure1.8 formulates a problem statement in terms of a challenge: How to understand the performance of a function as a function of underlying layers and surrounding functions expressed in a manageable number of parameters? Where the size and complexity of underlying layers and neighboring functions is large (tens, hundreds or even thousands man-years of software).

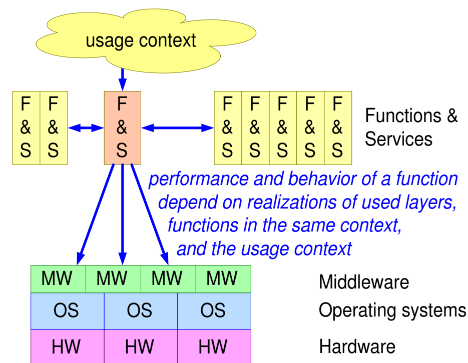


Figure 1.7: Function in System Context

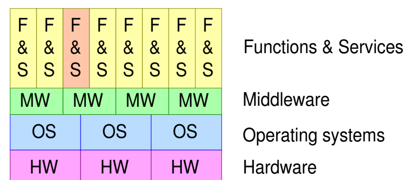


Figure 1.8: Challenge

1.4 Summary

We have worked through a simple example of a new application level function. The performance of this function cannot be predicted by looking at the code of the function itself. The underlying platform, neighboring applications and user context all have impact on the performance of this new function. The underlying platform, neighboring applications and user context are often large and very complex. We propose to use models to cope with this complexity.

1.5 Acknowledgements

The diagrams are a joined effort of Roland Mathijssen, Teun Hendriks and Gerrit Muller. Most of the material is based on material from the EXARCH course created by Ton Kostelijk and Gerrit Muller.

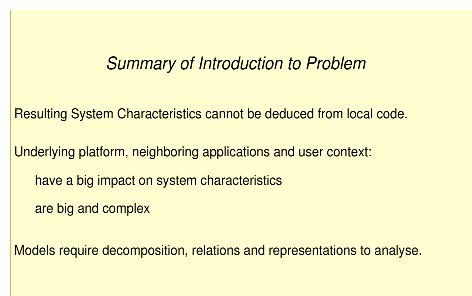
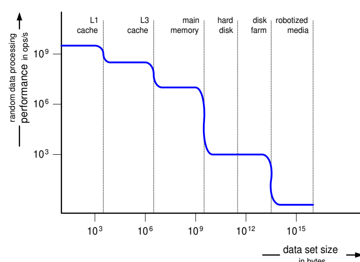


Figure 1.9: Summary of Problem Introduction

Chapter 2

Modeling and Analysis Fundamentals of Technology



2.1 Introduction

Figure 2.1 provides an overview of the content. In this article we discuss generic know how of computing technology. We will start with a commonly used decomposition and layering. We provide *figures of merit* for several generic computing functions, such as storage and communication. Finally we discuss caching as example of a technology that is related to storage figures of merit. We will apply the caching in a web shop example, and discuss design considerations.

content of this presentation

generic layering and block diagrams

typical characteristics and concerns

figures of merit

example of picture caching in web shop application

Figure 2.1: Overview Content *Fundamentals of Technology*

When we model technology oriented design questions we often need feasibility answers that are assessed at the level of non functional system requirements. Figure 2.2 shows a set of potential technology questions and the required answers at system level.

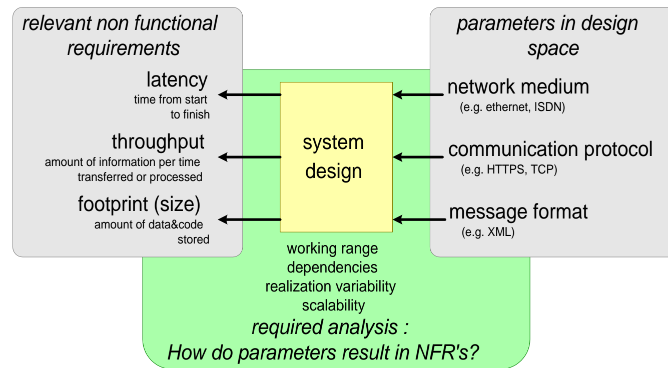


Figure 2.2: What do We Need to Analyze?

From design point of view we need, for example, information about the working range, dependencies, variability of the actual realization, or scalability.

2.2 Computing Technology Figures of Merit

In information and communication systems we can distinguish the following generic technology functions:

storage ranging from short term volatile storage to long term persistent storage. Storage technologies range from solid state static memories to optical disks or tapes.

communication between components, subsystems and systems. Technologies range from local interconnects and busses to distributed networks.

processing of data, ranging from simple control, to presentation to compute intensive operations such as 3D rendering or data mining. Technologies range from general purpose CPUs to dedicated I/O or graphics processors.

presentation to human beings, the final interaction point with the human users. Technologies range from small mobile display devices to large “cockpit” like control rooms with many flat panel displays.

Figure 2.3 shows these four generic technologies in the typical layering of a *Service Oriented Architecture* (SOA). In such an architecture the repositories, the bottom-tier of this figure, are decoupled from the business logic that is being

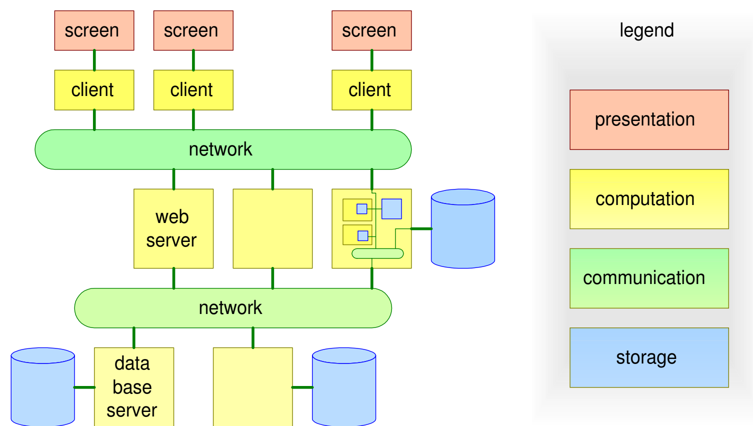


Figure 2.3: Typical Block Diagram and Typical Resources

handled in the middle layer, called *web server*. The client tier is the access and interaction layer, which can be highly distributed and heterogeneous.

The four generic technologies are recursively present: within a web-server, for example, communication, storage and processing are present. If we would zoom in further on the CPU itself, then we would again see the same technologies.

		latency	capacity
processor cache	*L1 cache* *L2 cache* *L3 cache*	sub ns ns	n kB n MB
fast volatile	*main memory*	tens ns	n GB
persistent	*disks* *disk arrays* *disk farms*	ms	n*100 GB n*10 TB
archival	*robotized* *optical media* *tape*	>s	n PB

Figure 2.4: Hierarchy of Storage Technology *Figures of Merit*

For every generic technology we can provide *figures of merit* for several characteristics. Figure 2.4 shows a table with different storage technologies. The table provides typical data for latency and storage capacity. Very fast storage technologies tend to have a small capacity. For example, L1 caches, static memory as part of

the CPU chip, run typically at processor speeds of several GHz, but their capacity is limited to several kilobytes. The much higher capacity main memory, solid state dynamic RAM, is much slower, but provides Gigabytes of memory. Non solid state memories use block access: data is transferred in chunks of many kilobytes. The consequence is that the access time for a single byte of information gets much longer, milliseconds for hard disks. When mechanical constructions are needed to transport physical media, such as robot arms for optical media, then the access time gets dominated by the physical transport times.

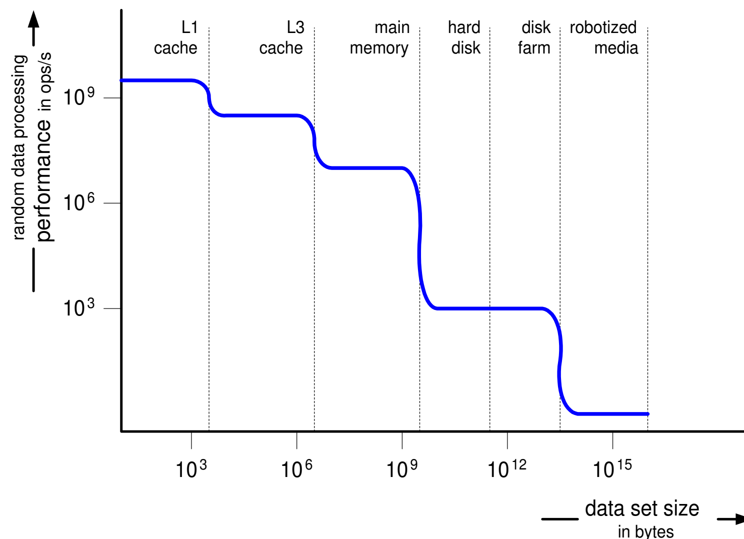


Figure 2.5: Performance as Function of Data Set Size

Figure 2.5 shows the same storage figures of merit in a 2-dimensional graph. The horizontal axis shows the capacity or the maximum data set size that we can store. The vertical axis shows the latency if we axis a single byte of information in the data set in a random order. Note that both axes are shown as a logarithmic scale, both axes cover a dynamic range of many orders of magnitude! The resulting graph shows a rather non-linear behavior with step-like transitions. We can access data very fast up to several kilobytes; the access time increases significantly when we exceed the L1 cache capacity. This effect repeats itself for every technology transition.

The communication figures of merit are shown in the same way in Figure 2.6. In this table we show *latency*, *frequency* and *distance* as critical characteristics. The latency and the distance have a similar relationship as latency and capacity for storage: longer distance capabilities result in longer latencies. The frequency behavior, which relates directly to the transfer capacity, is different. On chip very high frequencies can be realized. Off chip and on the printed circuit board these

		latency	frequency	distance
on chip	*connection*	sub ns	n GHz	n mm
	network	n ns	n GHz	n mm
PCB level		tens ns	n 100MHz	n cm
Serial I/O		n ms	n 100MHz	n m
network	*LAN*	n ms	100MHz	n km
	WAN	n 10ms	n GHz	global

Figure 2.6: Communication Technology Figures of Merit

high frequencies are much more difficult and costly. When we go to the long-distance networks optical technologies are being used, with very high frequencies.

2.3 Caching in Web Shop Example

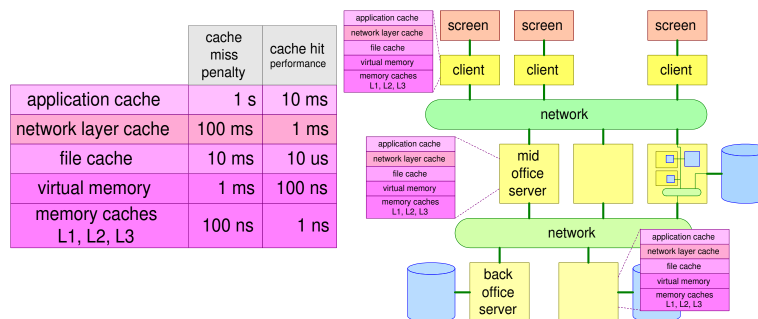


Figure 2.7: Multiple Layers of Caching

The speed differences in storage and communication often result in the use of a cache design pattern. The cache is a local fast storage, where frequently used data is stored to prevent repeated slow accesses to slow storage media. Figure 2.7 shows that this caching pattern is applied at many levels within a system, for example:

network layer cache to avoid network latencies for distributed data. Many communication protocol stacks, such as http, have local caches.

file cache as part of the operating system. The file cache caches the stored data

itself as well as directory information in main memory to speed up many file operations.

application cache application programs have dedicated caches based on application know how.

L1, L2, L3 memory caches A multi-level cache to bridge the speed gap between on-chip speed and off chip dynamic memory.

virtual memory where the physical main memory is cache for the much slower virtual memory that resides mostly on the hard disk.

Note that in the 3-tier SLA approach these caches are present in most of the tiers.

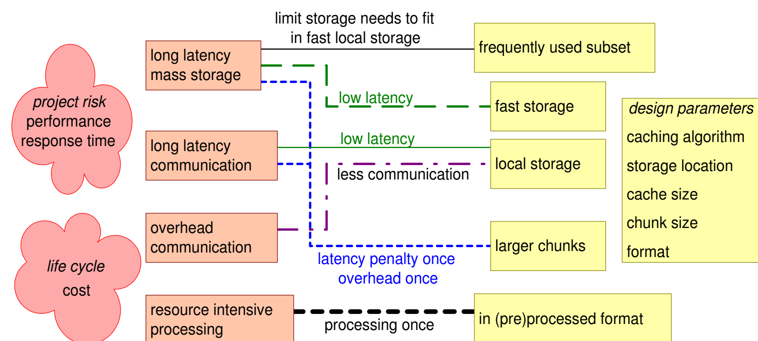


Figure 2.8: Why Caching?

In Figure 2.8 we analyze the introduction of caches somewhat more. At the left hand side we show that *long latencies of storage and communication*, *communication overhead*, and *resource intensive processing* are the main reasons to introduce caching. In the background the project needs for performance and cost are seen as driving factors. Potential performance problems could also be solved by overdimensioning, however this might conflict with the cost constraints on the project.

The design translates these performance reasons into a number of design choices:

frequently used subset enable the implementation to store this subset in the low capacity, but faster type of memory.

fast storage relates immediately to low latency of the storage itself

local storage gives low latency for the communication with the storage (sub)system

larger chunks reduces the number of times that storage or communication latency occurs and reduces the overhead.

cache in (pre)processed format to reduce processing latency and overhead

These design choices again translate in a number of design parameters:

- caching algorithm
- storage location
- cache size
- chunk size
- format

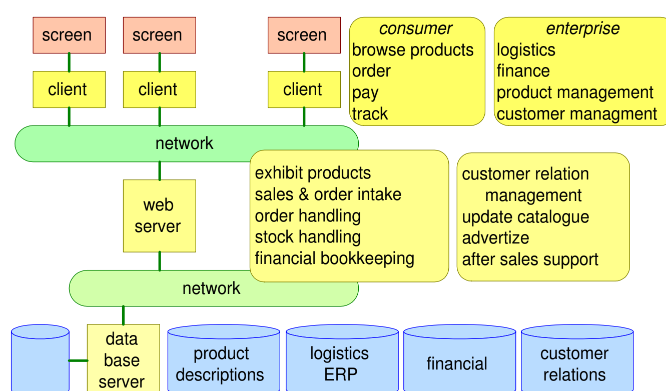


Figure 2.9: Example Web Shop

As an example of caching we look at a web shop, as shown in Figure 2.9. Customers at client level should be able to browse the product catalogue, to order products, to pay, and to track the progress of the order. Other stakeholders at client level have logistics functions, financial functions, and can do product and customer management. The web server layer provides the logic for the exhibition of products, the sales and order intake, the order handling, the stock handling, and the financial bookkeeping. Also at the web server layer is the logic for customer relation management, the update of the product catalogue, the advertisements, and the after sales support. The data base layer has repositories for product descriptions, logistics and resource planning, customer relations, and financial information.

We will zoom in on the product browsing by the customers. During this browsing customers can see pictures of the products in the catalogue. The originals of these pictures reside in the product catalogue repository in the data base layer. The web server determines when and how to show products for customers. The actual pictures are shown to many customers, who are distributed widely over the country.

The customers expect a fast response when browsing. Slow response may result in loss of customer attention and hence may cause a reduced sales. A picture

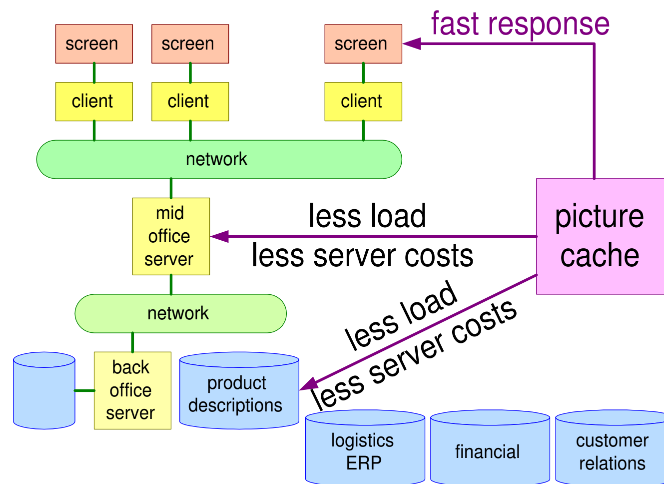


Figure 2.10: Impact of Picture Cache

cache at the web server level decreases the load at web server level, and at the same time improves the response time for customer browsing. It also reduces the server load of the data base.

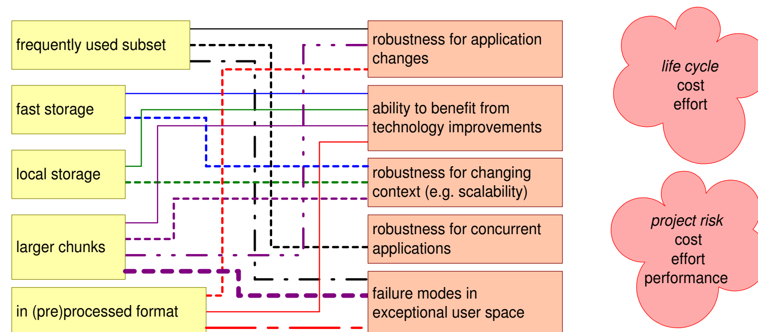


Figure 2.11: Risks of Caching

So far, the caching appears to be a no-brainer: improved response, reduces server loads, what more do we want? However, Figure 2.11 shows the potential risks of caching, caused mostly by increased complexity and decreased transparency. These risks are:

- The robustness for application changes may decrease, because the assumptions are not true anymore.
- The design becomes specific for this technology, impacting the ability to benefit from technology improvements.

- The robustness for changing context (e.g. scalability) is reduced
- The design is not robust for concurrent applications
- Failure modes in exceptional user space may occur

All of these technical risks translate in project risks in terms of cost, effort and performance.

2.4 Summary

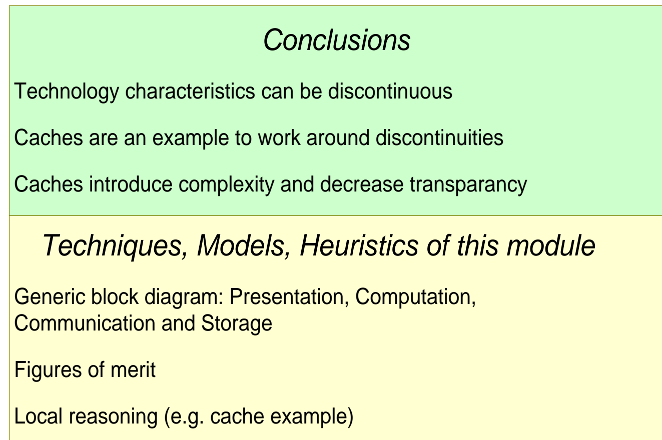
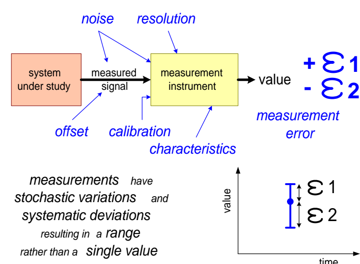


Figure 2.12: Summary

Figure 2.12 shows a summary of this paper. We showed a generic block diagram with *Presentation*, *Computation*, *Communication* and *Storage* as generic computing technologies. Technology characteristics of these generic technologies have discontiuous characteristics. At the transition from one type of technology to another type of technology a steep transition of characteristics takes place. We have provided *figures of merit* for several technologies. Caches are an example to work around these discontinuities. However, caches introduce complexity and decrease the transparancy of the design. We have applied local reasoning graphs to discuss the reasons of introduction of caches and the related design parameters. later we applied the same type of graph to discuss potential risks caused by the increased complexity and decreased transparancy.

Chapter 3

Modeling and Analysis: Measuring



3.1 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 3.1 shows the content of this paper. The crucial aspects of measuring are integrated into a measuring approach, see the next section.

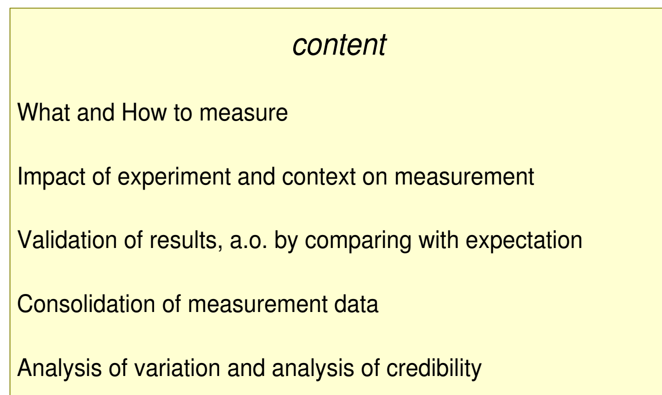


Figure 3.1: Presentation Content

3.2 Measuring Approach

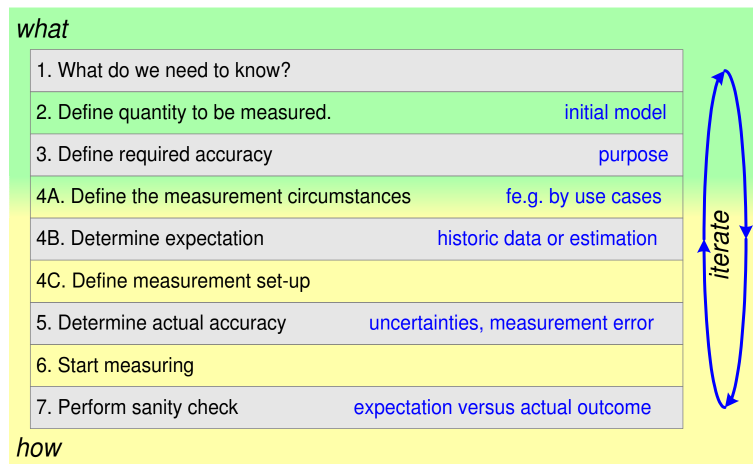


Figure 3.2: Measuring Approach: What and How

The measurement approach starts with *preparation and fact finding* and ends with *measurement and sanity check*. Figure 3.2 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.

3.2.1 What do we need?

The first question is: "What is the problem to be addressed, so what do we need to know?" Figure 3.3 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.

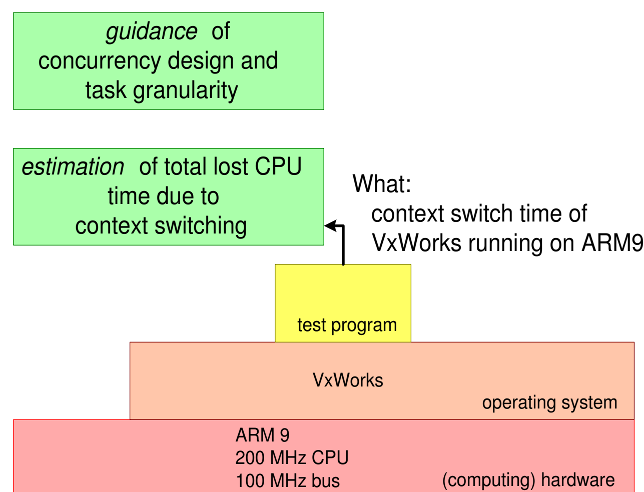


Figure 3.3: What do We Need? Example Context Switching

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 3.3 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.

3.2.2 Define quantity to be measured.

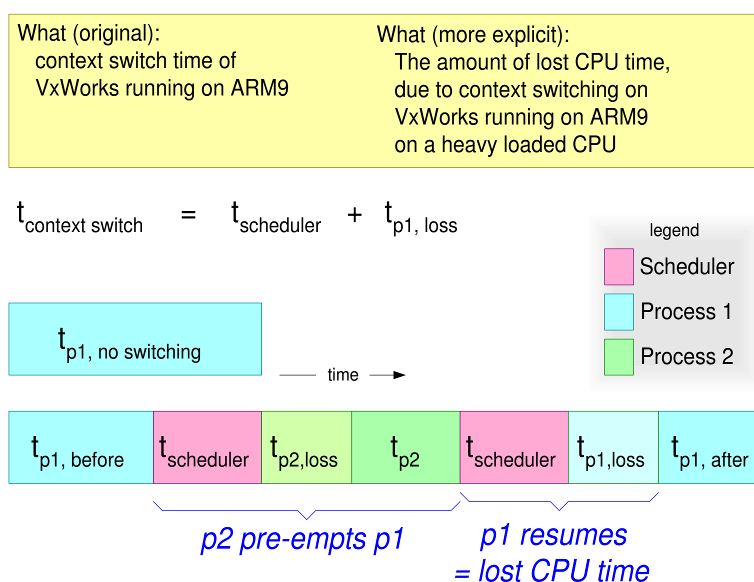


Figure 3.4: Define Quantity by Initial Model

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 3.4 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 3.4. Note that these effects are not present in a lightly loaded system that may completely run from cache.

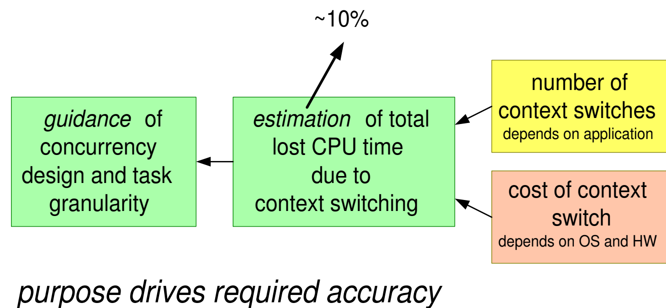


Figure 3.5: Define Required Accuracy

3.2.3 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.

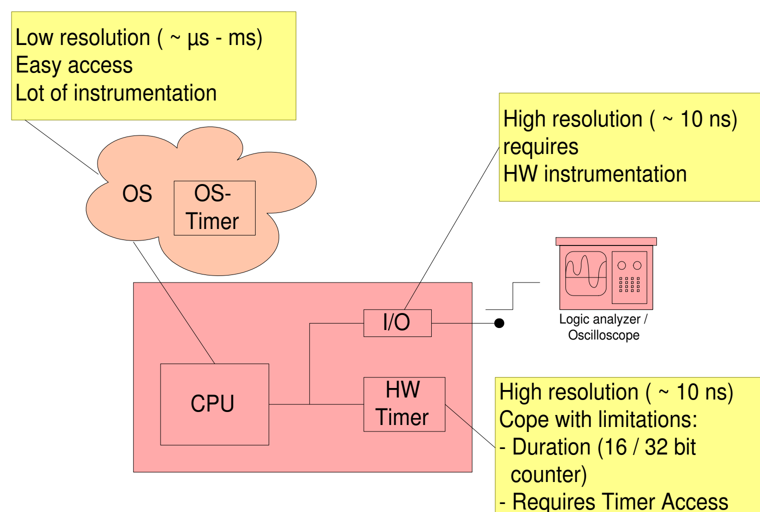


Figure 3.6: How to Measure CPU Time?

The relevance of the required accuracy is shown by looking at available measurement instruments. Figure 3.6 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.

3.2.4 Define the measurement circumstances

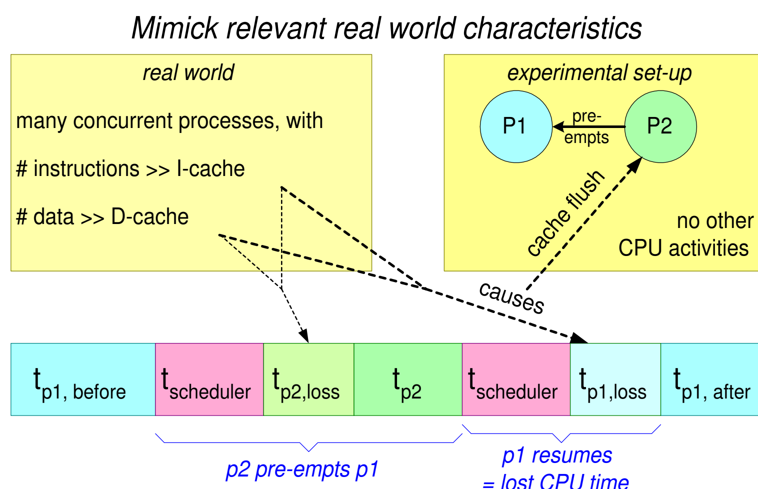


Figure 3.7: Define the Measurement Set-up

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 3.7 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.

3.2.5 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 3.8 and 3.9 provide a simple hardware model. Figure 3.10 provides a simple software model. The hardware and software models are combined in Figure 3.11. After substitution with assumed numbers we get a number for the expected outcome, see Figure3.12.

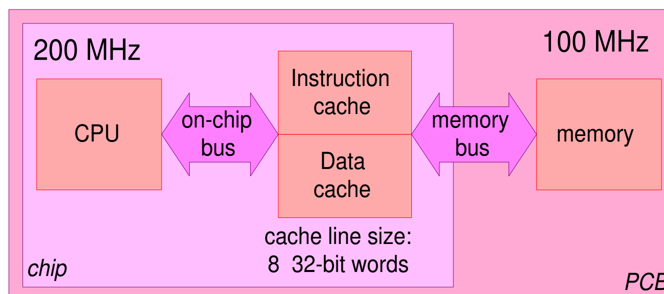


Figure 3.8: Case: ARM9 Hardware Block Diagram

Figure 3.8 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.

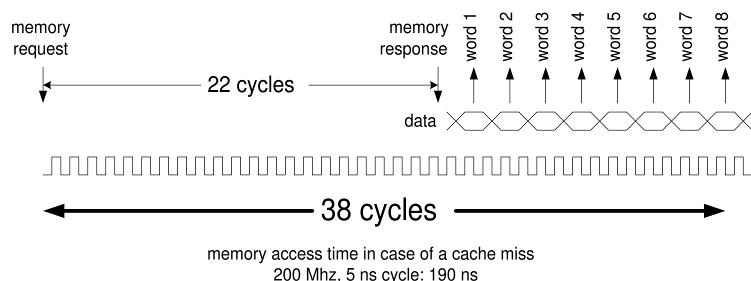


Figure 3.9: Key Hardware Performance Aspect

Figure 3.9 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 3.10 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 3.11 elaborates the software part of context switching in five contributing activities:

- save state P1

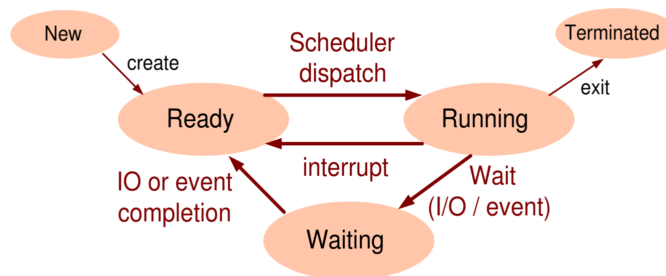


Figure 3.10: OS Process Scheduling Concepts

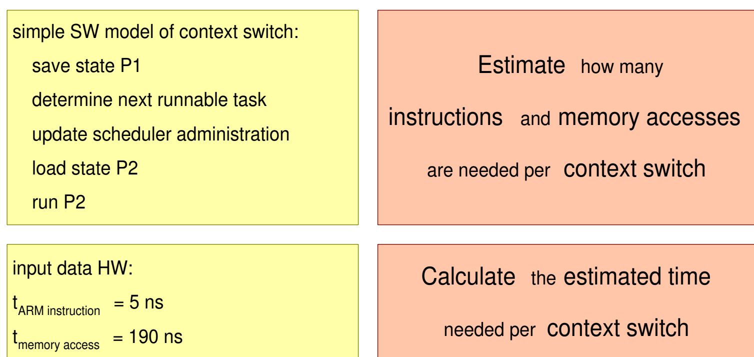


Figure 3.11: Determine Expectation

- determine next runnable task
- 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 3.9, 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 3.12 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: $(10 + 50 + 20 +$

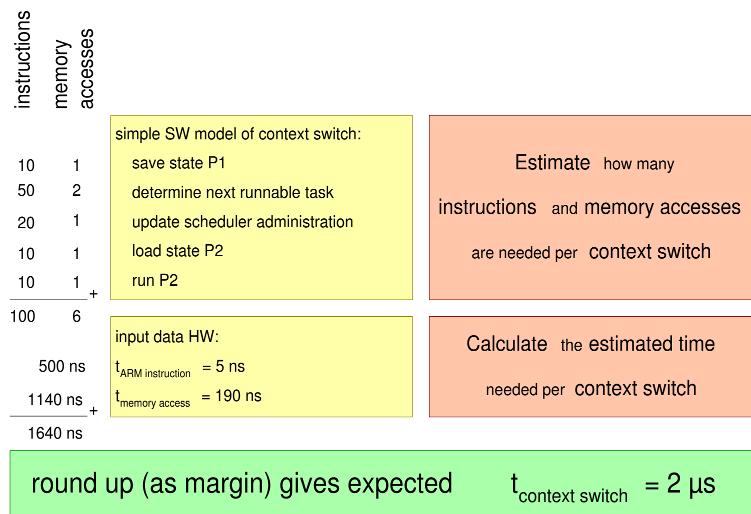


Figure 3.12: Determine Expectation Quantified

$10 + 10)instructions \cdot 5ns + (1 + 2 + 1 + 1 + 1)memoryaccesses \cdot 190ns = 500ns(instructions) + 1140ns(memoryaccesses) = 1640ns$ To add some margin for unknown activities we round this value to $2\mu s$.

3.2.6 Define measurement set-up

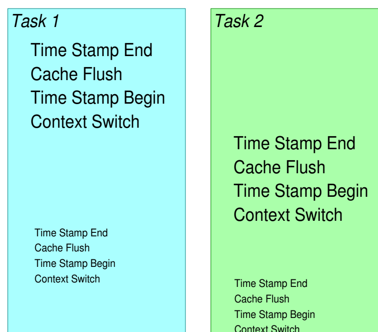


Figure 3.13: Code to Measure Context Switch

Figure 3.13 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 3.14 shows the CPU use as function of time for the two processes and the scheduler.

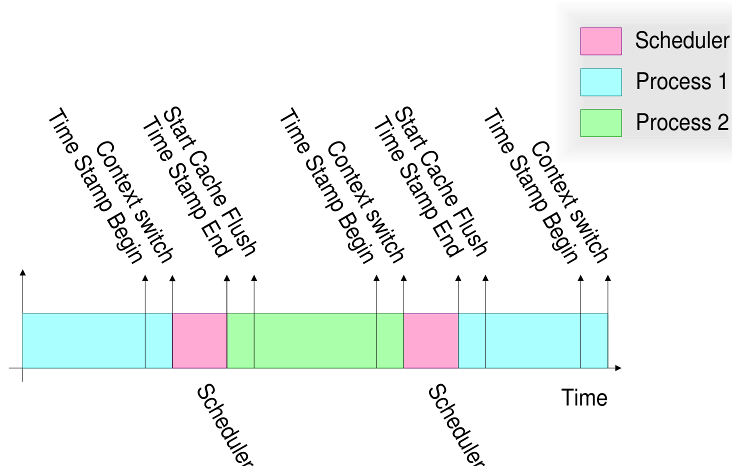


Figure 3.14: Measuring Context Switch Time

3.2.7 Expectation revisited

Once we have defined the measurement set-up we can again reason more about the expected outcome. Figure 3.15 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 3.15 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.

3.2.8 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.

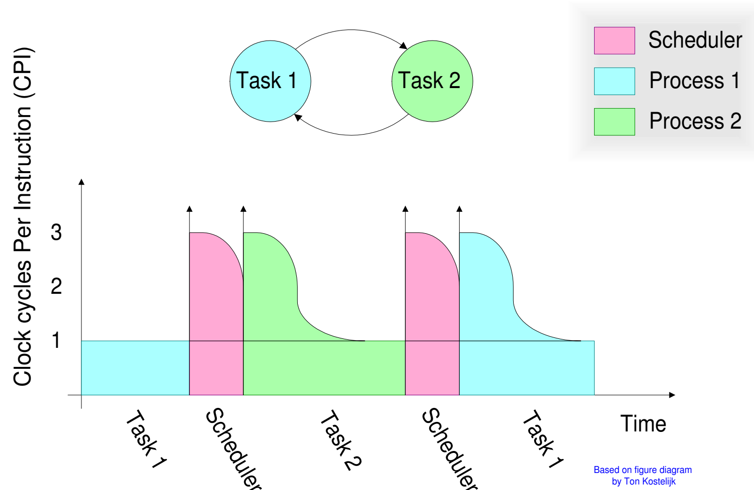


Figure 3.15: Understanding: Impact of Context Switch

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 3.17 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 3.13, then we get both a start and end value with a measurement error. Substracting these values adds the errors. In Figure 3.17 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 3.18.

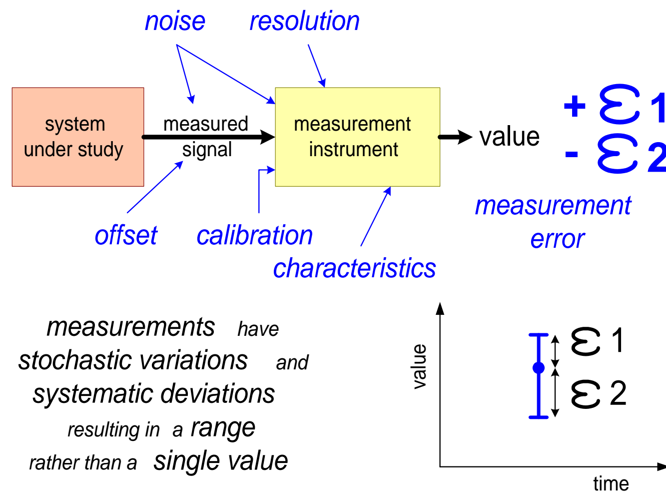


Figure 3.16: Accuracy: Measurement Error

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

$t_{start} = 10$ +/- 2 µs

$t_{end} = 14$ +/- 2 µs

$t_{duration} = 4$ +/- ? µs

systematic errors: add linear

stochastic errors: add quadratic

Figure 3.17: Accuracy 2: Be Aware of Error Propagation

3.2.9 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 3.19. 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

Measurements have
stochastic variations and systematic deviations
resulting in a range rather than a single value.

The inputs of modeling ,
"facts" , assumptions , and measurement results ,
also have stochastic variations and systematic deviations.

Stochastic variations and systematic deviations
propagate (add , amplify or cancel) through the model
resulting in an output range.

Figure 3.18: 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 3.19: 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 3.20 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

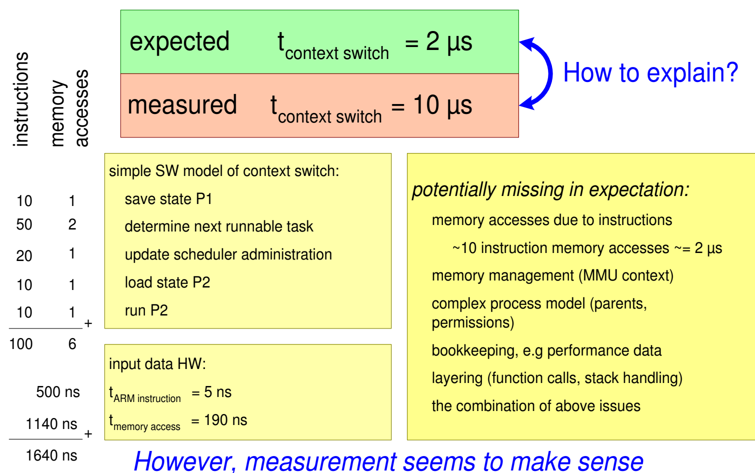


Figure 3.20: Expectation versus Measurement

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 3.21: Context Switch Overhead

Figure 3.21 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.

3.2.10 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.

3.2.11 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 $10\mu s$.

3.3 Summary

Figure 3.22 summarizes the measurement approach and insights.

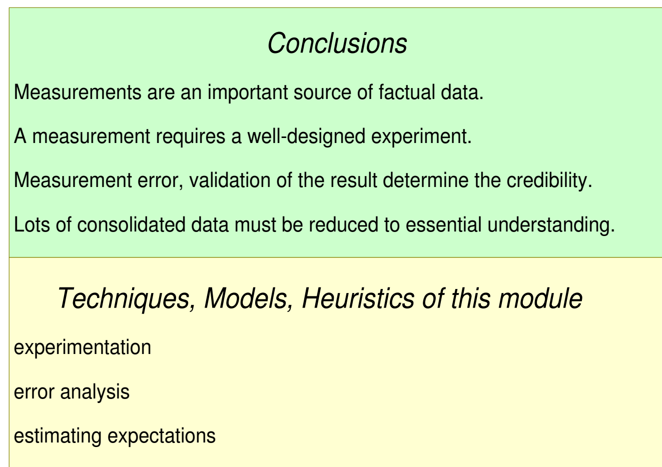


Figure 3.22: Summary Measuring Approach

3.4 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.

Bibliography

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

History

Version: 0.6, date: 6 March, 2007 changed by: Gerrit Muller

- added position slide

Version: 0.5, date: 23 February, 2007 changed by: Gerrit Muller

- added Fundamentals of Technology

Version: 0.4, date: 11 January, 2007 changed by: Gerrit Muller

- added intermezzo sheet why MA?

Version: 0.3, date: 5 January, 2007 changed by: Gerrit Muller

- removed budgeting

Version: 0.2, date: 4 January, 2007 changed by: Gerrit Muller

- changed the title from "Fundamentals of Technology" into "Inputs and Uncertainties"

Version: 0.1, date: 5 December, 2006 changed by: Gerrit Muller

- created reader

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

- created module