

Tutorial Measuring and Modeling System Performance



Gerrit Muller

University of South-Eastern Norway-NISE

Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway

gaudisite@gmail.com

Abstract

Performance is a key system level property. Performance, too, is affected by the myriad of decisions and choices in the development and configuration of both hardware and software components.

This tutorial focuses on measuring and modeling system performance. We will discuss computer hardware architectures and execution architectures, the software design concepts for the dynamic behavior of the system.

Distribution

This article or presentation is written as part of the Gaudí project. The Gaudí project philosophy is to improve by obtaining frequent feedback. Frequent feedback is pursued by an open creation process. This document is published as intermediate or nearly mature version to get feedback. Further distribution is allowed as long as the document remains complete and unchanged.

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

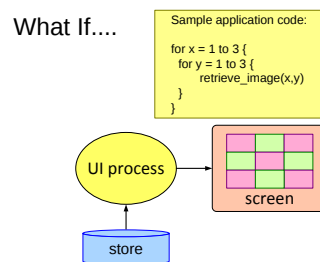
Contents

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	Performance Method Fundamentals	7
2.1	Introduction	7
2.2	Incremental approach	8
2.3	Multiple views needed to understand system performance	12
2.3.1	Construction Decomposition	12
2.3.2	Functional Decomposition	13
2.3.3	Execution Architecture	14
2.4	Benchmarking	16
2.5	Acknowledgements	18
3	Modeling and Analysis Fundamentals of Technology	19
3.1	Introduction	19
3.2	Computing Technology Figures of Merit	20
3.3	Caching in Web Shop Example	23
3.4	Summary	28
4	Modeling and Analysis: Measuring	29
4.1	introduction	29
4.2	Measuring Approach	31
4.2.1	What do we need?	32
4.2.2	Define quantity to be measured.	33
4.2.3	Define required accuracy	34
4.2.4	Define the measurement circumstances	35
4.2.5	Determine expectation	35
4.2.6	Define measurement set-up	38

4.2.7	Expectation revisited	39
4.2.8	Determine actual accuracy	39
4.2.9	Start measuring	41
4.2.10	Perform sanity check	44
4.2.11	Summary of measuring Context Switch time on ARM9 . .	44
4.3	Summary	45
4.4	Acknowledgements	46
5	Modeling and Analysis: Budgeting	47
5.1	Introduction	47
5.2	Budget-Based Design method	48
5.2.1	Goal of the method	48
5.2.2	Decomposition into smaller steps	49
5.2.3	Possible order of steps	49
5.2.4	Visualization	50
5.2.5	Guidelines	50
5.2.6	Example of overlay budget for wafersteppers	51
5.2.7	Example of memory budget for Medical Imaging Worksta- tion	52
5.2.8	Example of power budget visualizations in document han- dling	53
5.2.9	Evolution of budget over time	53
5.2.10	Potential applications of budget method	56
5.3	Summary	56
5.4	Acknowledgements	56
6	Formula Based Performance Design	58
6.1	Introduction	58
6.2	Using n-order formulas	58
6.3	Example of n-order formulas in MR reconstruction	59
6.4	Summary	62
6.5	Acknowledgements	63

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 "instantaneously". 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

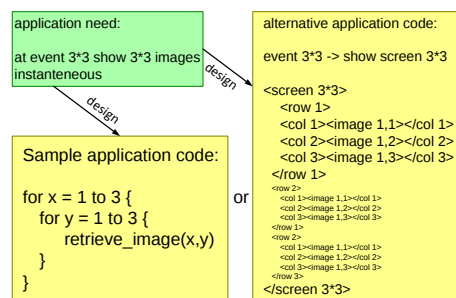


Figure 1.1: Image Retrieval Performance

Figure 1.2, where the `retrieve_image` function is part of a *user interface* process. This process reads image data directly from 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.

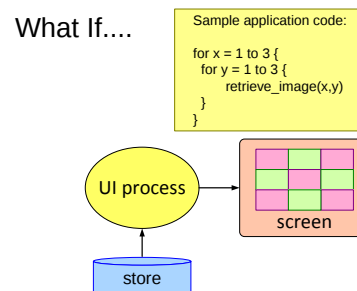


Figure 1.2: Straight Forward Read and Display

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

What If....

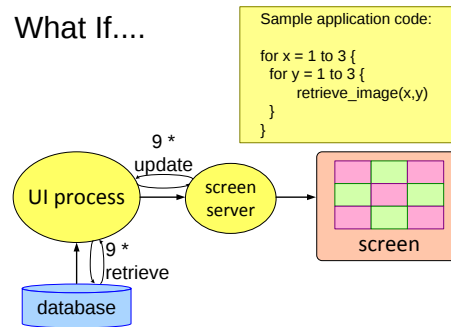


Figure 1.3: More Process Communication

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 $80\mu\text{sec}$. With 100 attributes per image we get a total construction overhead of $9 \cdot 100 \cdot 80\mu\text{s} = 72\text{ms}$. This cost is significant, because it is in the same order of magnitude as image transfer and rendering operations.

What If....

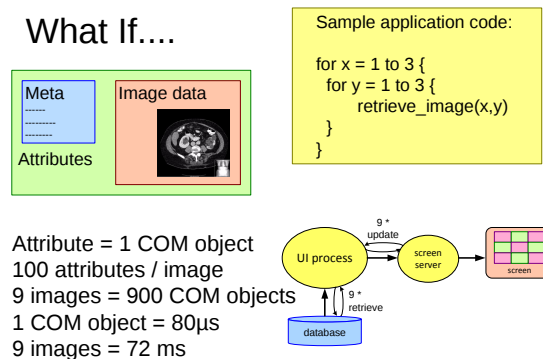


Figure 1.4: Meta Information Realization Overhead

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 $t_{I/O} = 1\text{ms}$, then the total overhead becomes $9 \cdot 512 \cdot 1\text{ms} \approx 4.5\text{s}$!

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)

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

$$t_{I/O} \approx 1ms$$

- . . .

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.

Figure 1.8 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).

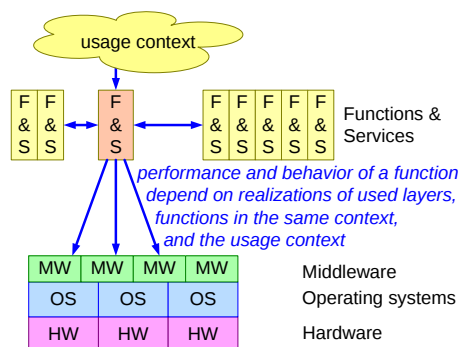
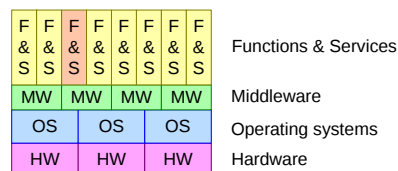


Figure 1.7: Function in System Context



Performance = Function (F&S, other F&S, MW, OS, HW)
MW, OS, HW >> 100 Manyyear : very complex

Challenge: How to understand MW, OS, HW
with only a few parameters

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 Kosteljik and Gerrit Muller.

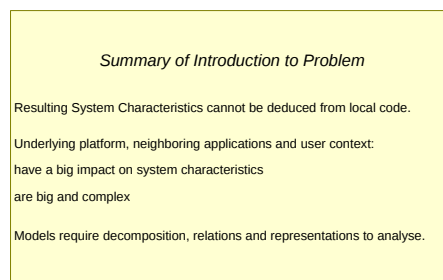
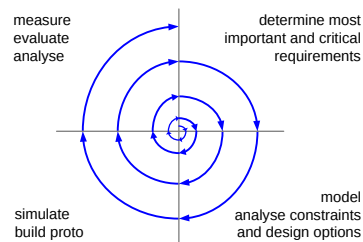


Figure 1.9: Summary of Problem Introduction

Chapter 2

Performance Method Fundamentals



2.1 Introduction

The performance of a system is determined by the hardware design, the software design and the mapping of the software design on the hardware, the so-called execution architecture. The execution architecture itself is the design step from the conceptual view to the realization view. The justification for design decisions has its roots in the customer objectives view and the application view, based on often ill articulated needs, concerns and expectations of the customer. A good understanding of mostly performance and timing related needs and expectations is needed and used to get a specific and measurable product definition with respect to performance and timing requirements. This definition is not a pure top down approach, a priori know how of the possible solutions is used to converge more quickly on relevant specification issues.

Figure 2.1 visualizes these relations in the CAFCR model. The top-down and bottom-up iteration is shown as modeling and analyzing top down and simulating and measuring bottom up.

We will discuss an incremental approach to ensure the link between the CAFCR views. Then we discuss shortly the representations needed to understand system

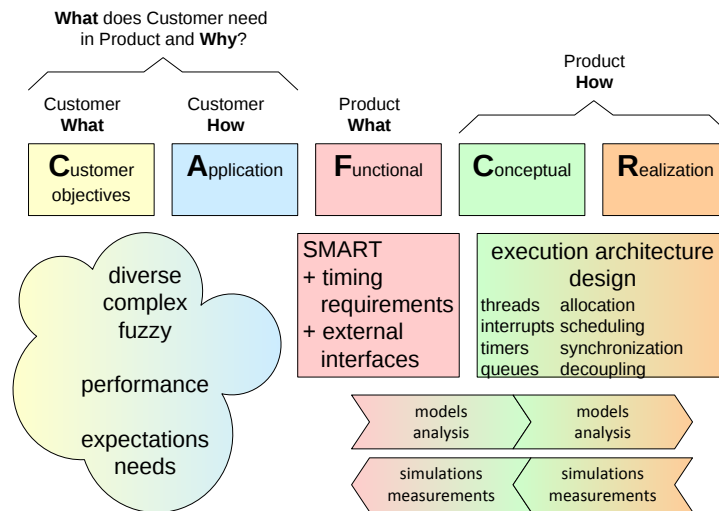


Figure 2.1: Positioning in CAFCR

performance. Finally, we discuss benchmarking as a way to get quantified insight for performance models.

2.2 Incremental approach

1A Collect most critical performance and timing requirements	
1B Find system level diagrams	HW block diagram, SW diagram, functional model(s) concurrency model, resource model, time-line
2A Measure performance at 3 levels	application, functions and micro benchmarks
2B Create Performance Model	
3 Evaluate performance, identify potential problems	
4 Performance analysis and design	granularity, synchronization, prioritization, allocation, resource management
Re-iterate all steps	are the right requirements addressed, refine diagrams, measurements, models, and improve design

Figure 2.2: Top-level Performance Design Method

Figure 2.2 shows a stepwise approach for performance design. Step 1 is the identification of the most critical timing and performance requirements, parallel with the search for system level diagrams. During step 2 the performance of the system is measured at multiple levels, and a performance model is created. Step 3

is the evaluation of the performance and the identification of potential problems. Step 4 is the actual performance analysis and design. All these steps are not purely sequential, iteration is crucial.

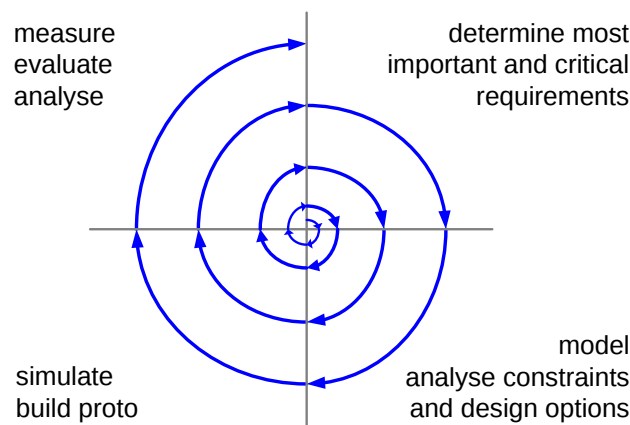


Figure 2.3: Incremental approach

An incremental approach is strongly recommended. The problem and solution domain is often so complex that no human being can understand and oversee it entirely. The understanding and overview is build up in steps or passes, where all aspects are touched in one pass. The next pass deepens and enriches the insights. The reason that incremental approaches work is that it enables the humans to learn, based on the short feedback cycles. Typical cycle times are days or weeks, not months.

Figure 2.3 shows the spiral approach. First the **what** (requirements) and **how** (design) are studied, than the implementation, verification and evaluation is done, which closes the feedback cycle.

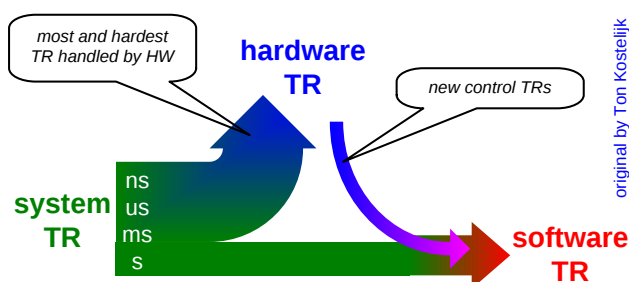


Figure 2.4: Decomposition of system TR in HW and SW

Most timing requirements are handled by the hardware, especially the very

short response times are implemented by means of dedicated hardware. However this dedicated hardware itself needs some control, with more relaxed timing constraints. The hardware design imposes also timing requirements on the software design. Figure 2.4 visualizes this transformation of severe system timing requirements in somewhat more relaxed software timing requirements.

The architect is continuously trying to improve his understanding of problem and solution[4]. This understanding is based on many different interacting insights, such as functionality, behavior, relationships et cetera. An important factor in understanding is the **quantification**. Quantification helps to get grip on the many vague aspects of problem and solution. Many aspects can be quantified, much more than most designers are willing to quantify.

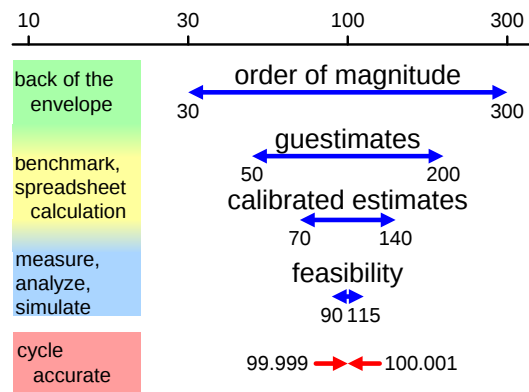


Figure 2.5: Successive quantification refined

The precision of the quantification increases during the project. Figure 2.5 shows the stepwise refinement of the quantification. In first instance it is important to get a feeling for the problem by quantifying orders of magnitude. For example:

- How fast should the system respond, for instance zap?
- What is the affordable cost, how much is the customer willing and able to spend?
- How many pictures/movies do they want to watch, transfer, store concurrently?
- How much storage and bandwidth is needed?

The order of magnitude numbers can be refined by making back of the envelop calculations, making simple models and making assumptions and estimates. From this work it becomes clear where the major uncertainties are and which measurements or other data acquisitions will help to refine the numbers further.

At the bottom of figure 2.5 the other extreme of the spectrum of quantification is shown, in this example cycle accurate simulation of video frame processing results in very accurate numbers. It is a challenge for an architect to bridge these worlds.

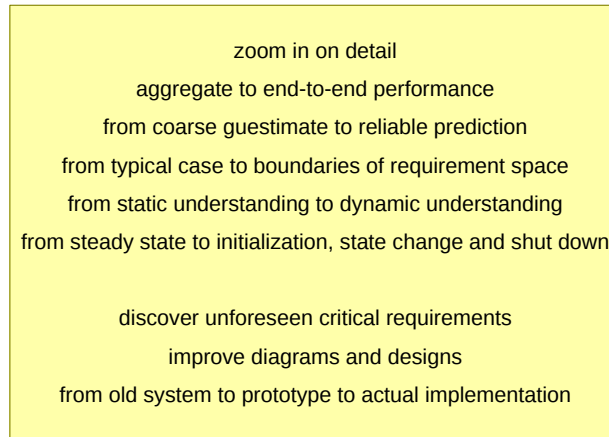


Figure 2.6: Directions of iterations

Figure 2.6 shows the many directions of potential iterations:

zoom in on detail Drill down to the essential detail, often based on historic data and experience.

aggregate to end-to-end performance Add all numbers to estimate the end-to-end time

from coarse guestimate to reliable prediction Work from coarse estimates, which provide guidance and insight, towards more accurate numbers that are sufficiently accurate and robust to be usable as prediction.

from typical case to boundaries of requirement space Start to understand the “typical” use case that is frequently happening and then look at the more complicated cases, such as the boundaries of the requirement space.

from static understanding to dynamic understanding Start by creating simple insight by ignoring many dynamic aspects. Add dynamics step by step, when the impact is significant.

from steady state to initialization, state change and shut down Start with the steady state situation, where the application is continuously is repeating the same operations. Later the singular moments are added, such as start-up, shut down and state changes.

discover unforeseen critical requirements Modeling of the system itself and exploring its performance often triggers the discovery of requirements that were not yet foreseen or that are more critical than foreseen.

improve diagrams and designs The increasing insight should be captured in the diagrams and designs.

from old system to prototype to actual implementation The earlier fact finding start the better the models are grounded in facts. Older, existing systems are a gold-mine of factual information. In order to get facts about the impact of design changes prototypes are needed. Finally the actual implementation should be used for verification of the performance requirements and the underlying designs, such as budgets.

2.3 Multiple views needed to understand system performance

The decomposition can be done along different axes. Subsection 2.3.1 shows *construction* as axis, and Subsection 2.3.2 shows the *functional* decomposition. The decomposition into concurrent activities and the mapping on processes, threads and processors is called the execution architecture, which is described in Subsection 2.3.3.

2.3.1 Construction Decomposition

The construction decomposition views the system from the construction point of view, see Figure 2.7 for an example. In this example the decomposition is structured to show layers and the degree of domain know-how. The vertical layering defines the dependencies: components in the higher layers depend on components in the lower layers. Components are not dependent on components at the same or higher layer. The amount of domain know how provides an indication of the added value of the components. More generic components are more likely to be shared in a broader application area, and are more likely to be purchased instead of being developed.

The construction decomposition is mostly used for the design management. It defines units of design, as these are created and stored in repositories and later updated. The atomic units are aggregated into compound design units. In software the compound design units are often called *packages*, in hardware they are called *modules*. The blocks in Figure 2.7 are at the level of these packages and modules. *Packages and modules* are used as unit for testing and release and they often coincide with organizational ownership and responsibility.

In hardware this is quite often a very natural decomposition, for instance into cabinets, racks, boards and finally integrated circuits, Intellectual property (IP)

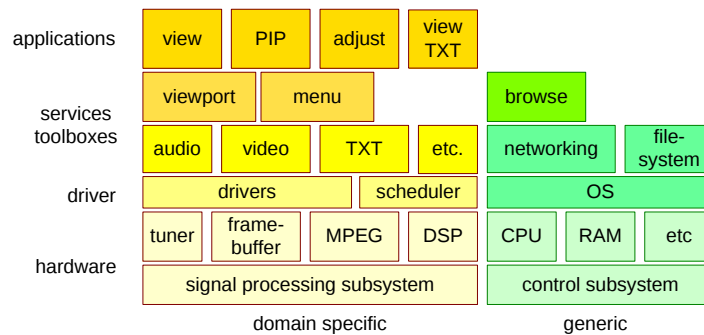


Figure 2.7: Example of a construction decomposition of a simple TV. The vertical axis is used for layers, where higher layers depend on lower layers, but not vice versa. In horizontal direction the left hand side shows the domain specific components, the right hand side shows the more generic components.

cores and cells. The components in the hardware are very tangible. The relationship with a number of other decompositions is reasonably one to one, for instance with the work breakdown for project management purposes.

The construction decomposition in software is more ambiguous. The structure of the code repository and the supporting build environment comes close to the hardware equivalent. Here files and packages are the aggregating construction levels. This decomposition is less tangible than the hardware decomposition and the relationship with other decompositions is sometimes more complex.

2.3.2 Functional Decomposition

The functional decomposition decomposes end user functions into more elementary functions. The elementary functions are internal, the decomposition in elementary functions is not easily observable from outside the system. In other words, the **what** is worked out in **how**. Be aware of the fact that the word *function* in system design is heavily overloaded. No attempt is made to define the functional decomposition more sharply, because a sharper definition does not provide more guidance to architects. Main criterium for a good functional decomposition is its useability for design. A functional decomposition provides insight how the system will accomplish its job.

Figure 2.8 shows an example of (part of) a functional decomposition for a camera type device. It shows a data flow with communication, processing, and storage functions and their relations. This functional decomposition is **not** addressing the control aspects, which might be designed by means of a second functional decomposition, this time taken from the control point of view.

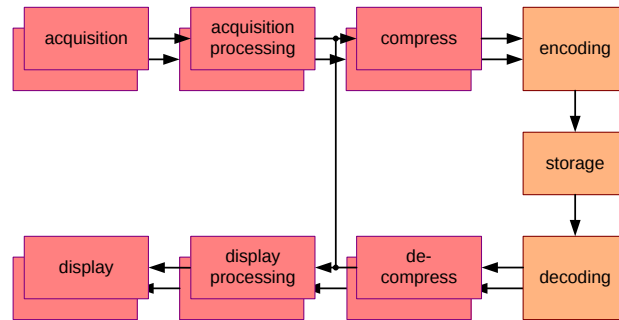


Figure 2.8: Example functional decomposition camera type device

2.3.3 Execution Architecture

The execution architecture is the run-time architecture of a system. The process¹ decomposition plays an important role in the execution architecture. Figure 2.9 shows an example of a process decomposition.

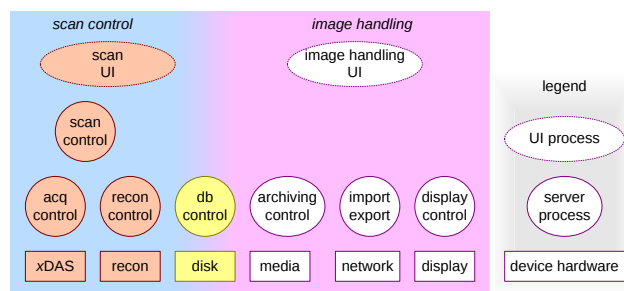


Figure 2.9: An example of a process decomposition of a MRI scanner.

One of the main concerns for process decomposition is concurrency: which concurrent activities are needed or running, and how do we synchronize these activities? Two techniques to support asynchronous functionality are widely used in operating systems: processes and threads. Processes are self sustained, which own their own resources, especially memory. Threads have less overhead than processes. Threads share resources, which makes them more mutually dependent. In other words processes provide better means for separation of concerns.

The execution architecture must map the functional decomposition on the process decomposition. This mapping must ensure that the timing behavior of the system is within specification. The most critical timing behavior is defined by the dead lines. Missing a dead line may result in loss of throughput or functionality. The

¹Process in terms of the operating system

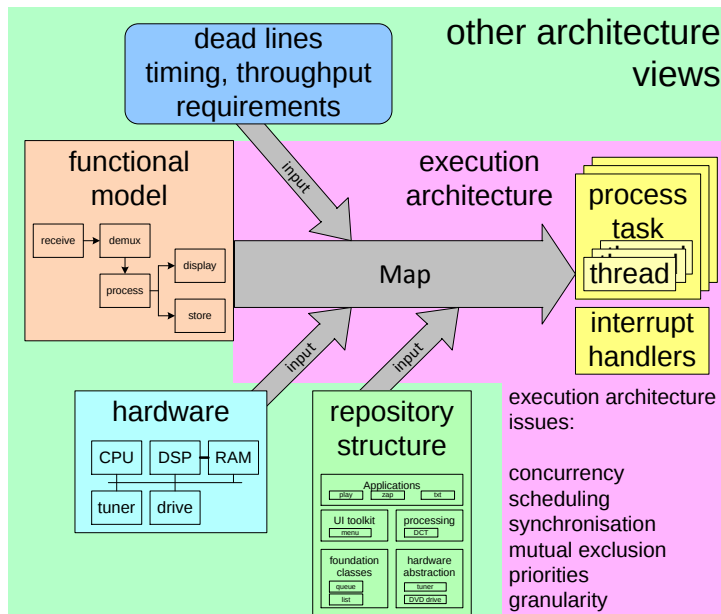


Figure 2.10: Execution Architecture

timing behavior is also determined by the choice of the synchronization methods, by the granularity of synchronization and by the scheduling behavior. The most common technique to control the scheduling behavior is by means of priorities. This requires, of course, that priorities are assigned. Subsystems with limited concurrency complexity may not even need multiple threads, but these subsystems can use a single thread that keeps repeating the same actions all the time. The mapping is further influenced by hardware software allocation choices, and by the construction decomposition. Figure 2.10 shows what views are combined to create the execution architecture.

A well known method in the hard real time domain is DARTS (Design Approach for Real Time Systems) [1]. This methods provides guidelines to identify hard real time requirements, translate them in activities and to map activities on tasks. DARTS then describes how to design the scheduling priorities.

In practice many components from the construction decomposition are used in multiple functions, and are mapped on multiple processes. These shared components are aggregated in shared or dynamic-link libraries (dll's). Sharing the program code run-time is advantageous from memory consumption point of view.

We promote iteration over hardware, software and functional design. In practice this iteration is limited, amongst others due to different development life-cycles of hardware, software and system. Often most hardware design choices are made long before the software design is known. In other words the hardware is a fact,

where only minor changes are possible. Another reality is that large amounts of software are inherited from existing systems, which also severely limits the degrees of freedom of the software design.

The remaining degrees of freedom for the execution architecture are limited to:

- allocation to tasks, processes or threads
- allocation of hardware resources
- priorities, scheduling strategy (limited by the operating system facilities)
- granularity

The art of designing a good execution architecture is to simplify the problems sufficiently, by focusing on the real critical timing issues.

2.4 Benchmarking

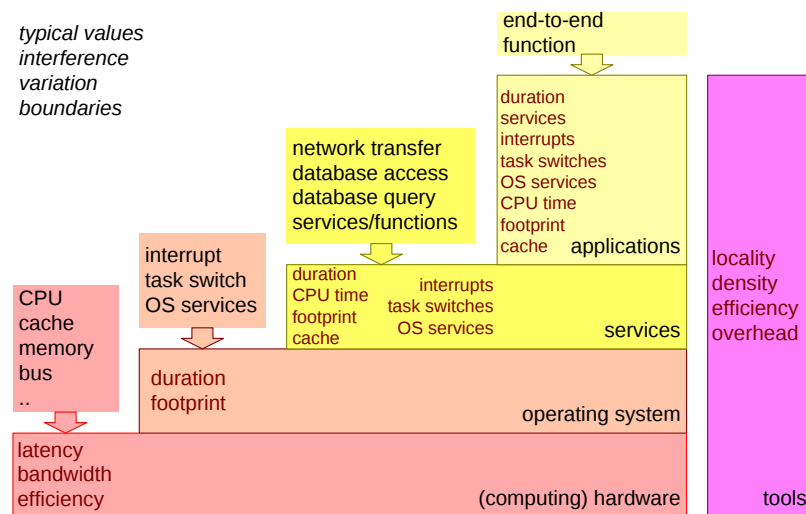


Figure 2.11: Layered Benchmarking

We propose to tackle the dynamic analysis by measuring and analyzing the system at several levels, as shown in Figure 2.11. The purpose of this approach is to understand the system performance throughout the entire system. Unfortunately the entire system is way too complex to understand in one single pass. Therefore we look for natural layers or subsystems. For the medical imaging workstation a reasonably generic four layer model is helpful:

Hardware CPU, memory, bus, cache, disk, network, et cetera. At this level latencies, bandwidth and resource efficiency are valuable data points.

Operating System (OS) Interrupt handling, task switching, process communication, resource management, and other OS services. At this level duration and footprint data needs to be known.

Services (or Middleware) Interoperability services based on networks or storage devices, database functionality, and other higher level services. At this level lots of performance data is needed: throughput, duration, CPU time, footprint, cache impact, number of generated interrupts and context switches, and number of invoked OS services.

Applications The end-to-end performance of functions, as perceived by the user of the system. The same performance data is needed here as on the services level, plus the amount of service invocations.

Tools Compilers, linkers, high level generators, configurators. These tools generally influence most other layers. Typical data to be known is locality and density of code, efficiency of generated output, run-time overhead induced by the tools.

We will start simple by determining typical values for the mentioned parameters. However, a lot of additional insight can be obtained by looking at the variation in these numbers, and by thinking in terms of range boundaries. Special attention is needed for interference aspects. For example sharing of computing resources often results in degraded cache performance when functions run concurrently.

The actual characteristics of the technology being used must be measured and understood in order to make a good (reliable, cost effective) design. The basic understanding of the technology is created by performing micro-benchmarks: measuring the elementary functions of the technology in isolation. Figure 2.12 lists a typical set of micro-benchmarks to be performed. The list shows infrequent and often slow operations and frequently applied operations that are often much faster. This classification implies already a design rule: slow operations should not be performed often².

The results of micro-benchmarks should be used with great care. The measurements show the performance in totally unrealistic circumstances, in other words it is the best case performance. This best case performance is a good baseline to understand performance, but when using the numbers the real life interference (cache disturbance for instance) should be taken into account. Sometimes additional

²This really sounds as an open door. However, I have seen many violations of this entirely trivial rule, such as setting up a connection for every message, performing I/O byte by byte et cetera. Sometimes such a violation is offset by other benefits, especially when a slow operation is in fact not very slow and when the brute force approach is both affordable as well as extremely simple.

	<i>infrequent operations, often time-intensive</i>	<i>often repeated operations</i>
<i>database</i>	start session finish session	perform transaction query
<i>network, I/O</i>	open connection close connection	transfer data
<i>high level construction</i>	component creation component destruction	method invocation same scope other context
<i>low level construction</i>	object creation object destruction	method invocation
<i>basic programming</i>	memory allocation memory free	function call loop overhead basic operations (add, mul, load, store)
<i>OS</i>	task, thread creation	task switch interrupt response
<i>HW</i>	power up, power down boot	cache flush low level data transfer

Figure 2.12: Typical micro-benchmarks for timing aspects

measurements are needed at a slightly higher level to calibrate the performance estimates.

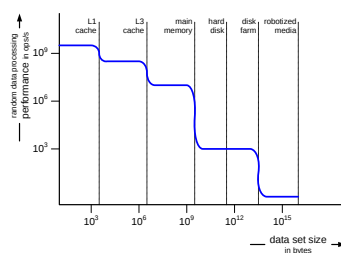
The standard work about performance issues in computer architectures is the book by Hennessy and Patterson [2]. Here modelling and measurement methods can be found that can serve as inspiration for performance analysis of embedded systems.

2.5 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. Reinder Bril gave feedback which was used to improve the sheets.

Chapter 3

Modeling and Analysis Fundamentals of Technology



3.1 Introduction

Figure 3.1 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.

<i>content of this presentation</i>
generic layering and block diagrams
typical characteristics and concerns
figures of merit
example of picture caching in web shop application

Figure 3.1: 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 3.2 shows a set of potential technology questions and the required answers at system level.

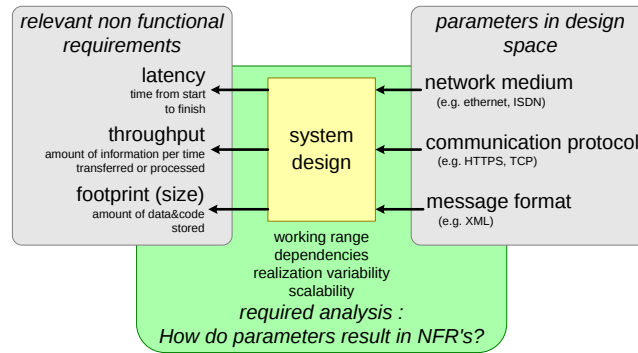


Figure 3.2: What do We Need to Analyze?

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

3.2 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 3.3 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

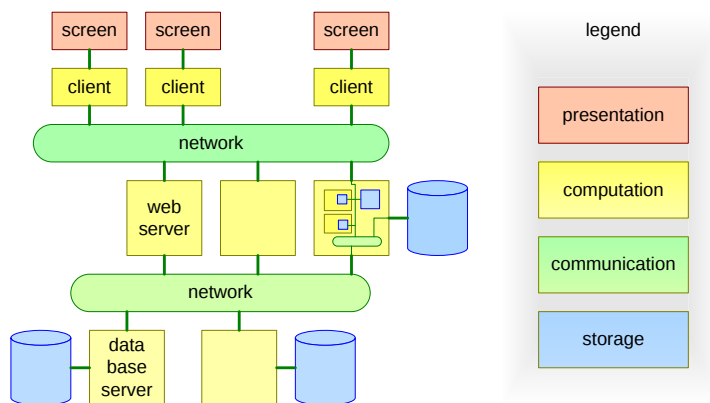


Figure 3.3: Typical Block Diagram and Typical Resources

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

Figure 3.4: Hierarchy of Storage Technology *Figures of Merit*

For every generic technology we can provide *figures of merit* for several characteristics. Figure 3.4 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.

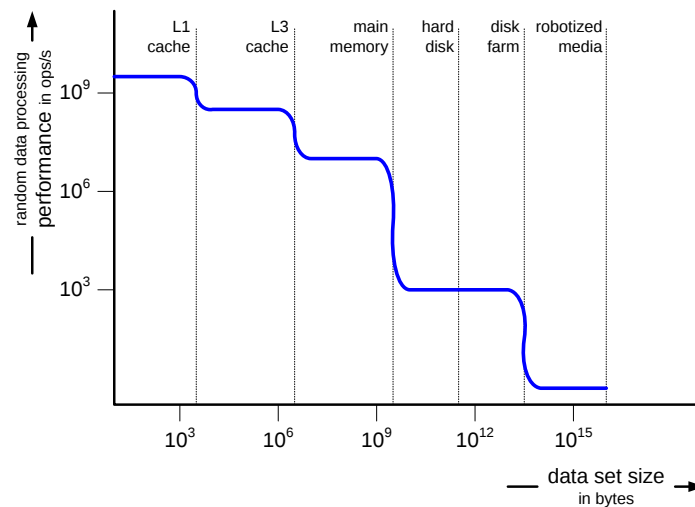


Figure 3.5: Performance as Function of Data Set Size

Figure 3.5 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 access 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 3.6. 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 high frequencies are much more difficult and costly. When we go to the long-

		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 3.6: Communication Technology Figures of Merit

distance networks optical technologies are being used, with very high frequencies.

3.3 Caching in Web Shop Example

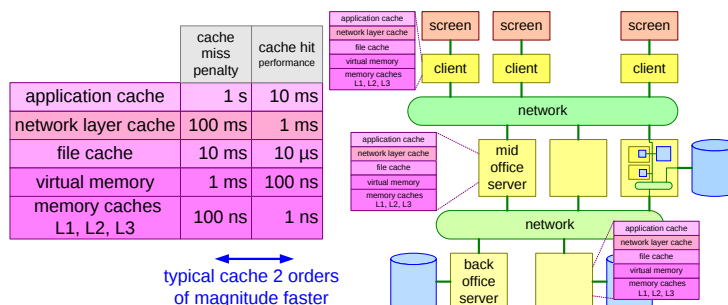


Figure 3.7: Multiple Layers of Caching

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 3.7 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.

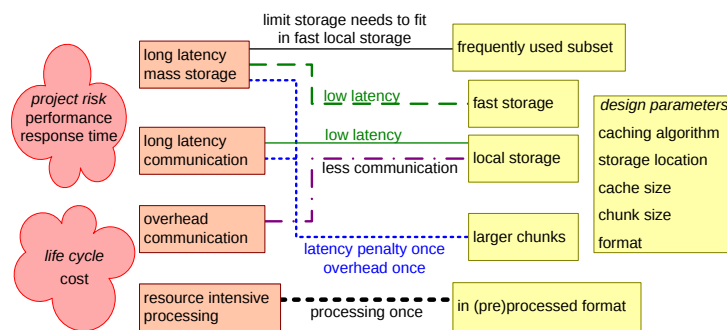


Figure 3.8: Why Caching?

In Figure 3.8 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

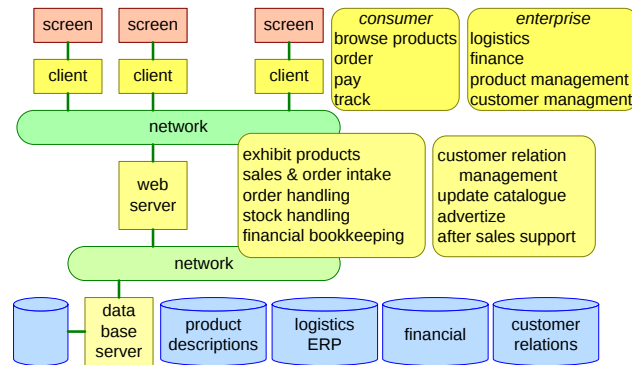


Figure 3.9: Example Web Shop

As an example of caching we look at a web shop, as shown in Figure 3.9. 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 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.

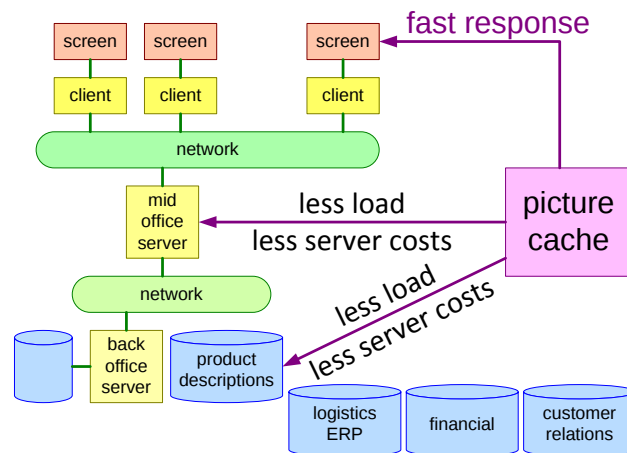


Figure 3.10: Impact of Picture Cache

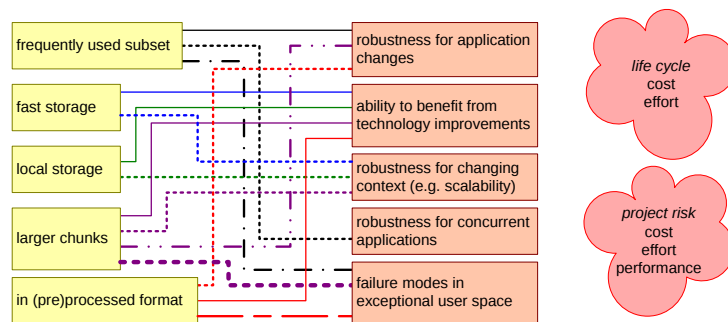


Figure 3.11: Risks of Caching

So far, the caching appears to be a no-brainer: improved response, reduces server loads, what more do we want? However, Figure 3.11 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.

3.4 Summary

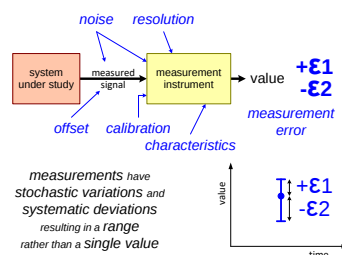
<i>Conclusions</i> Technology characteristics can be discontinuous Caches are an example to work around discontinuities Caches introduce complexity and decrease transparency
<i>Techniques, Models, Heuristics of this module</i> Generic block diagram: Presentation, Computation, Communication and Storage Figures of merit Local reasoning (e.g. cache example)

Figure 3.12: Summary

Figure 3.12 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 discontinuous 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 transparency 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 transparency.

Chapter 4

Modeling and Analysis: Measuring



4.1 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 4.1 shows the content of this paper. The crucial aspects of measuring are integrated into a measuring approach, see the next section.

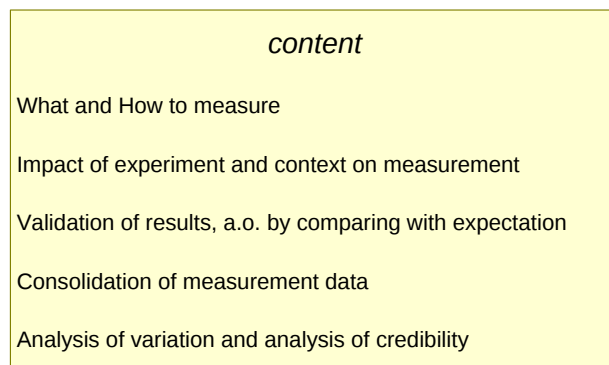


Figure 4.1: Presentation Content

4.2 Measuring Approach

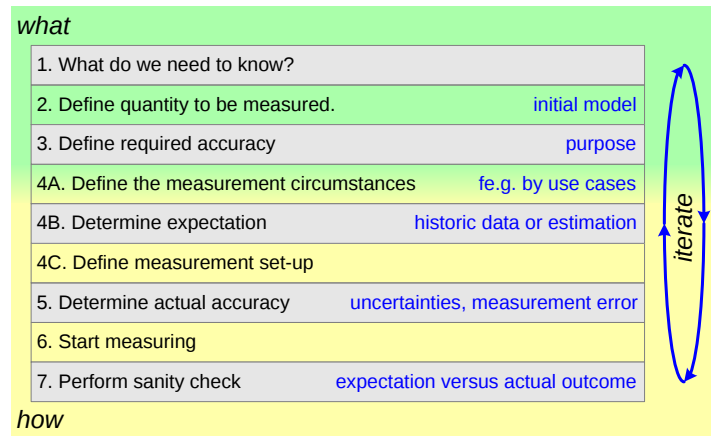


Figure 4.2: Measuring Approach: What and How

The measurement approach starts with *preparation and fact finding* and ends with *measurement and sanity check*. Figure 4.2 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 measured 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 repeated 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.

4.2.1 What do we need?

The first question is: “What is the problem to be addressed, so what do we need to know?” Figure 4.3 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.

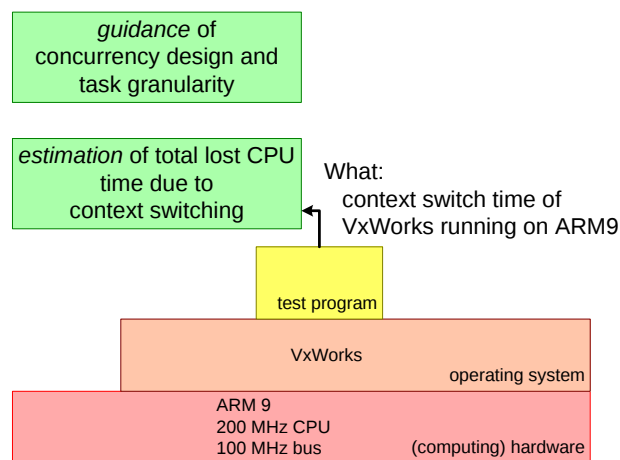


Figure 4.3: What do We Need? Example Context Switching

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 4.3 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.

4.2.2 Define quantity to be measured.

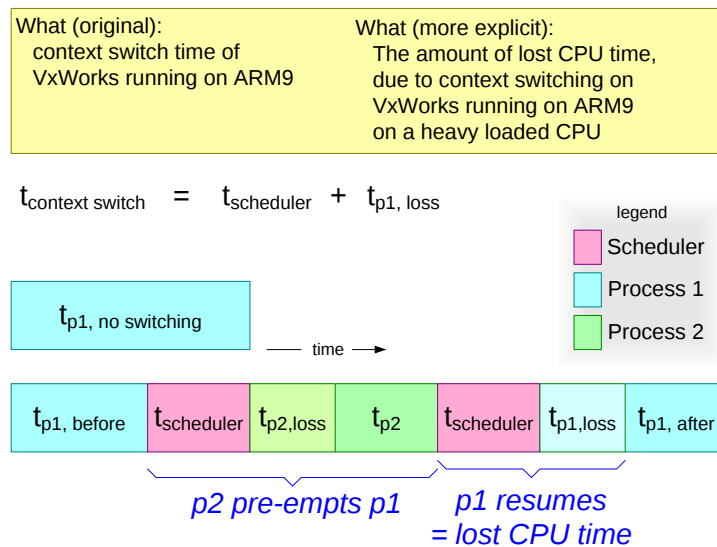


Figure 4.4: Define Quantity by Initial Model

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 4.4 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 $t_{p1, \text{loss}}$ in Figure 4.4. Note that these effects are not present in a lightly loaded system that may completely run from cache.

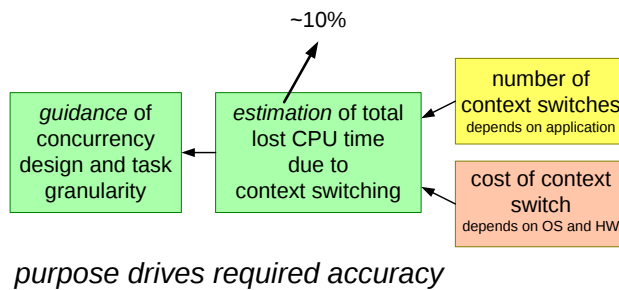


Figure 4.5: Define Required Accuracy

4.2.3 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.

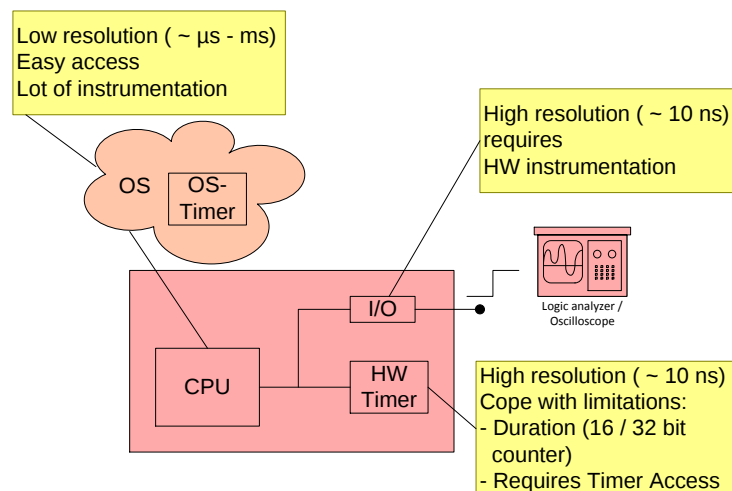


Figure 4.6: How to Measure CPU Time?

The relevance of the required accuracy is shown by looking at available measurement instruments. Figure 4.6 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.

4.2.4 Define the measurement circumstances

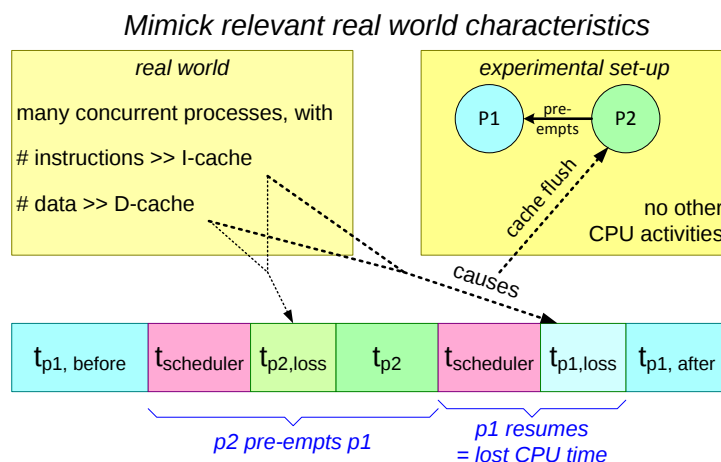


Figure 4.7: Define the Measurement Set-up

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 4.7 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.

4.2.5 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 4.8 and 4.9 provide a simple hardware model. Figure 4.10 provides a simple software model. The hardware and software models are combined in Figure 4.11. After substitution with assumed numbers we get a number for the expected outcome, see Figure 4.12.

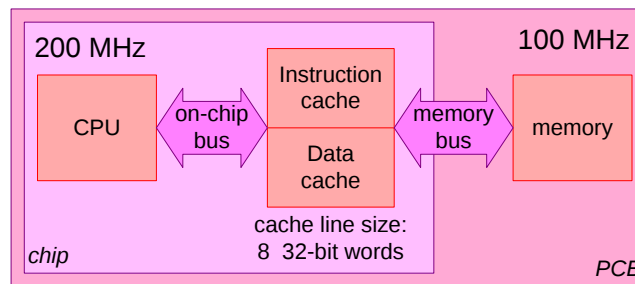


Figure 4.8: Case: ARM9 Hardware Block Diagram

Figure 4.8 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.

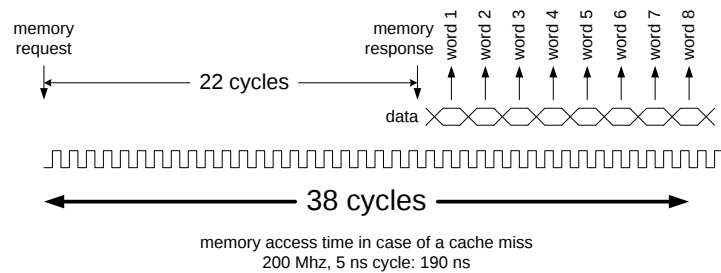


Figure 4.9: Key Hardware Performance Aspect

Figure 4.9 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 4.10 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 4.11 elaborates the software part of context switching in five contributing activities:

- save state P1

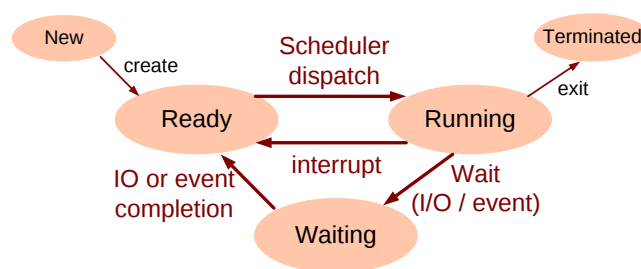


Figure 4.10: OS Process Scheduling Concepts

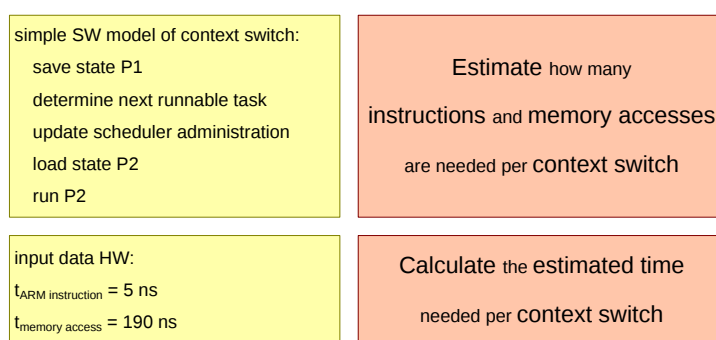


Figure 4.11: Determine Expectation

- 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 4.9, 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 4.12 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 multiplications and additions: $(10 + 50 + 20 +$

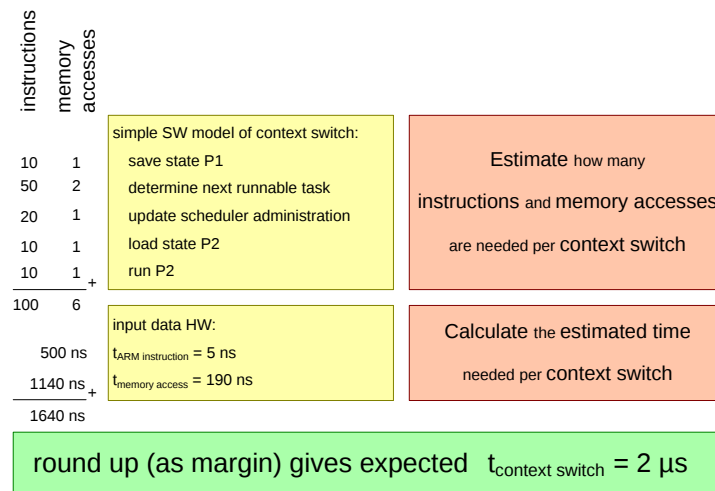


Figure 4.12: Determine Expectation Quantified

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

4.2.6 Define measurement set-up

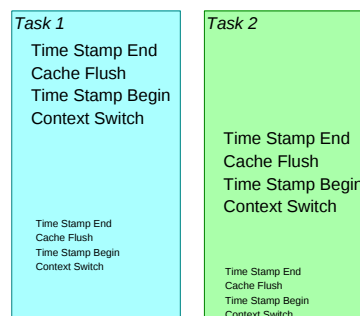


Figure 4.13: Code to Measure Context Switch

Figure 4.13 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 4.14 shows the CPU use as function of time for the two processes and the scheduler.

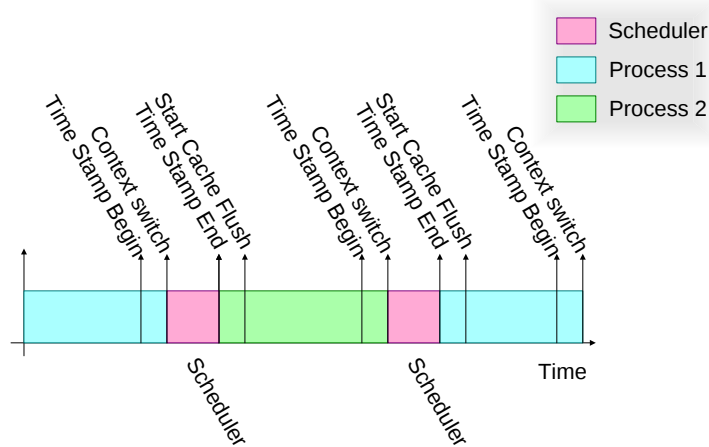


Figure 4.14: Measuring Context Switch Time

4.2.7 Expectation revisited

Once we have defined the measurement set-up we can again reason more about the expected outcome. Figure 4.15 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 accesses 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 4.15 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 activated 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 switches and hence four dips of the cache performance.

4.2.8 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.

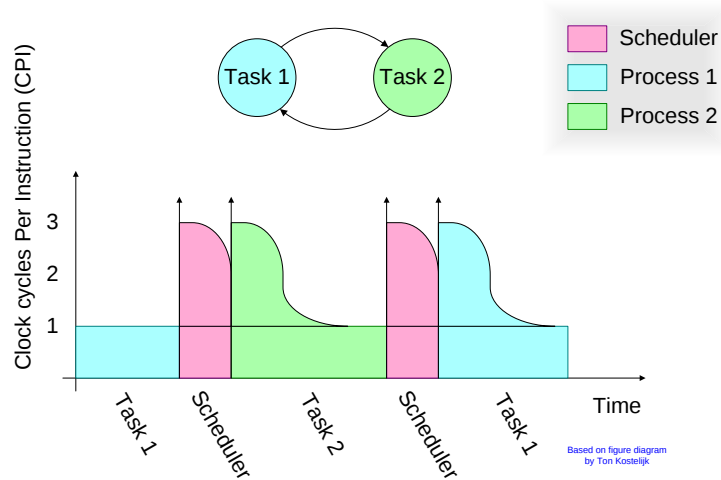


Figure 4.15: Understanding: Impact of Context Switch

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 stochastic, such as noise, or systematic, such as offsets. Error accumulation works differently for stochastic or systematic contributions: stochastic 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 4.17 shows the effect of error propagation. Special attention should be paid to subtraction of measurement results, because the values are subtracted while the errors are added. If we do a single measurement, as shown earlier in Figure 4.13, then we get both a start and end value with a measurement error. Subtracting these values adds the errors. In Figure 4.17 the provided values result in $t_{duration} = 4 + / - 4\mu s$. In other words when subtracted 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 4.18.

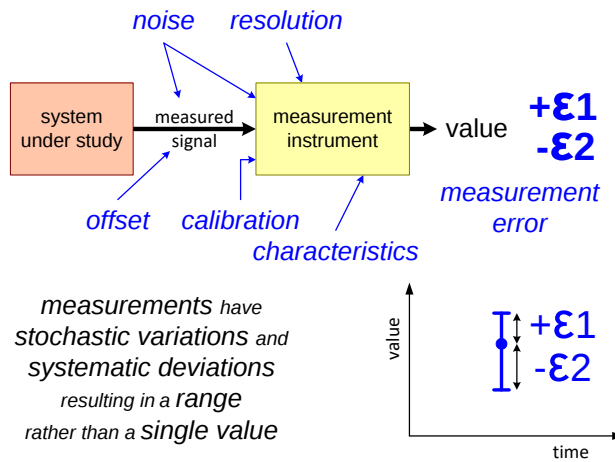


Figure 4.16: Accuracy: Measurement Error

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

$$t_{\text{start}} = 10 \pm 2 \mu\text{s}$$

$$t_{\text{end}} = 14 \pm 2 \mu\text{s}$$

$$t_{\text{duration}} = 4 \pm ? \mu\text{s}$$

systematic errors: add linear

stochastic errors: add quadratic

Figure 4.17: Accuracy 2: Be Aware of Error Propagation

4.2.9 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 4.19. 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 $2\mu\text{s}$ 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 $10\mu\text{s}$. For comparison the measurement is also repeated with a disabled cache, which decreases the context switch even more to $50\mu\text{s}$. These measurements show

Measurements have stochastic variations and systematic deviations resulting in a <i>range</i> rather than a <i>single value</i> .
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 4.18: Intermezzo Modeling Accuracy

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

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

Figure 4.19: 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 $2\mu\text{s}$ for the context switch time, however already taking into account negative cache effects. The expectation is a factor 5 more optimistic 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 4.20 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

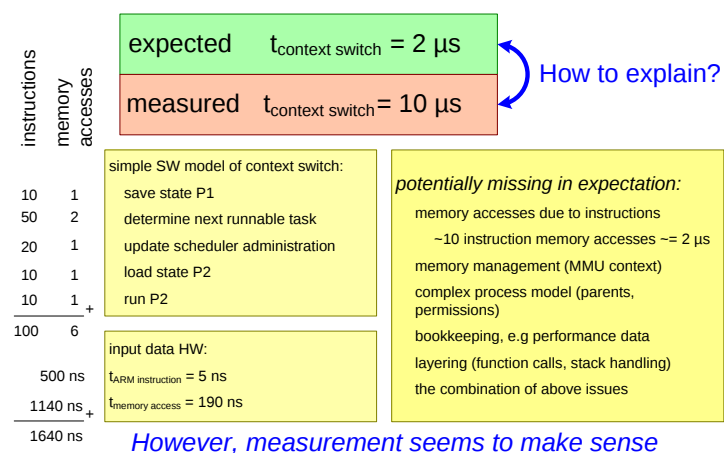


Figure 4.20: Expectation versus Measurement

Management Unit (MMU) might be part of the process context, causing more state information to be saved and restored. Often many small management activities take place in the kernel. For example, the process model might be more complex than assumed, with process hierarchy and permissions. Maybe hierarchy or permissions are accessed for some reasons, maybe 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 cumulates, but might also multiply.

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

$n_{\text{context switch}}$ (s^{-1})	$t_{\text{context switch}} = 10 \mu\text{s}$		$t_{\text{context switch}} = 2 \mu\text{s}$	
	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 4.21: Context Switch Overhead

Figure 4.21 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.

4.2.10 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 general 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 available 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 the overhead, but now averaged over a longer period.

4.2.11 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 $10\mu s$.

4.3 Summary

Figure 4.22 summarizes the measurement approach and insights.

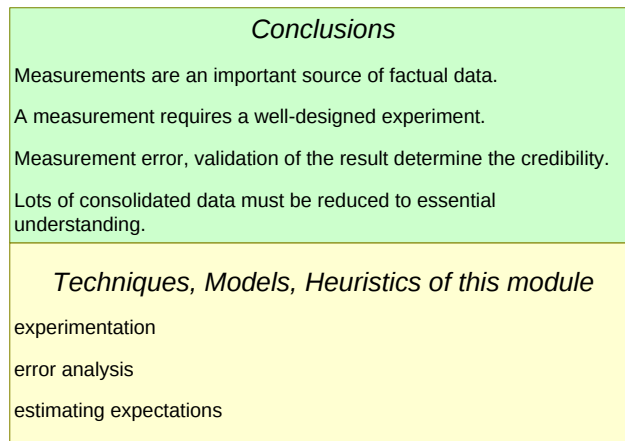


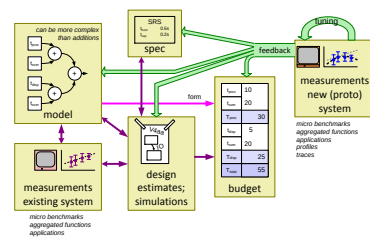
Figure 4.22: Summary Measuring Approach

4.4 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.

Chapter 5

Modeling and Analysis: Budgeting



5.1 Introduction

Budgets are well known from the financial world as a means to balance expenditures and income. The same mechanism can be used in the technical world to balance for instance resource use and system performance.

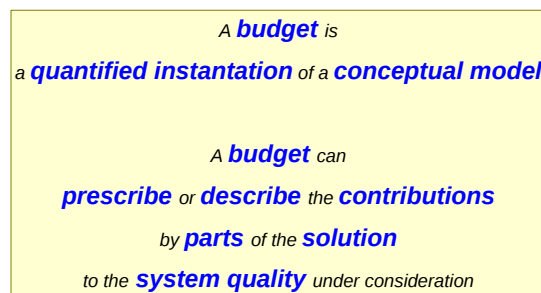


Figure 5.1: Definition of a budget in the technical world

Budgets are more than an arbitrary collection of numbers. The relationship

between the numbers is guided by an underlying model. Figure 5.1 shows *what* a budget is. Technical budgets can be used to provide guidance by prescribing allowed contributions per function or subsystem. Another use of budgets is as a means for understanding, where the budget describes these contributions.

We will provide and illustrate a budget method with the following attributes:

- a goal
- a decomposition in smaller steps
- possible orders of taking these steps
- visualization(s) or representation(s)
- guidelines

5.2 Budget-Based Design method

In this section we illustrate a *budget-based design* method applied at waferstepper, health care, and document handling systems, where it has been applied on different resources: overlay, memory, and power.

5.2.1 Goal of the method

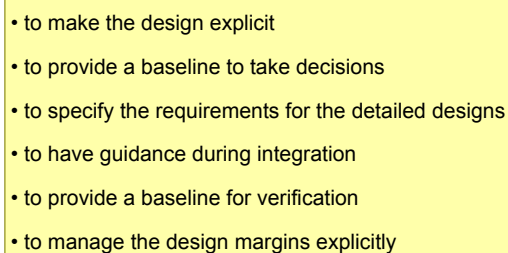
- 
- to make the design explicit
 - to provide a baseline to take decisions
 - to specify the requirements for the detailed designs
 - to have guidance during integration
 - to provide a baseline for verification
 - to manage the design margins explicitly

Figure 5.2: Goals of budget based design

The goal of the budget-based design method is to guide the implementation of a technical system in the use of the most important resource constraints, such as memory size, response time, or positioning accuracy. The budget serves multiple purposes, as shown in Figure 5.2.

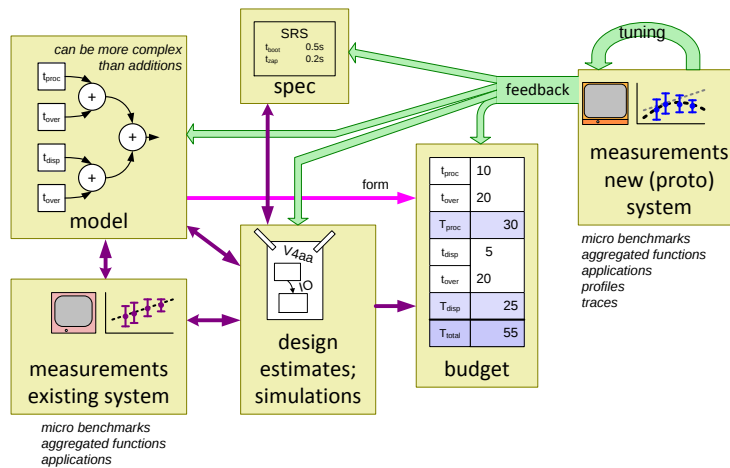


Figure 5.3: Visualization of Budget-Based Design Flow. This example shows a response time budget.

5.2.2 Decomposition into smaller steps

Figure 5.3 visualizes the budget-based design flow. This visualization makes it clear that although the budget plays a central role in this design flow, cooperation with other methods is essential. In this figure other cooperating methods are performance modeling, micro-benchmarking, measurement of aggregated functions, measurements at system level, design estimates, simulations, and requirements specification.

Measurements of all kinds are needed to provide substance to the budget. Micro-benchmarks are measurements of elementary component characteristics. The measured values of the micro-benchmarks can be used for a bottom-up budget. Measurements at functional level provide information at a higher aggregated level; many components have to cooperate actively to perform a function. The outcome of these function measurements can be used to verify a bottom-up budget or can be used as input for the system level budget. Measurements in the early phases of the system integration are required to obtain feedback once the budget has been made. This feedback will result in design changes and could even result in specification changes. The use of budgets can help to set up an integration plan. The measurement of budget contributions should be done as early as possible, because the measurements often trigger design changes.

5.2.3 Possible order of steps

Figure 5.4 shows a budget-based design flow (the *order* of the method). The starting point of a budget is a model of the system, from the conceptual view.

step	example
1A measure old systems	micro-benchmarks, aggregated functions, applications
1B model the performance starting with old systems	flow model and analytical model
1C determine requirements for new system	response time or throughput
2 make a design for the new system	explore design space, estimate and simulate
3 make a budget for the new system:	models provide the structure measurements and estimates provide initial numbers specification provides bottom line
4 measure prototypes and new system	micro-benchmarks, aggregated functions, applications profiles, traces
5 Iterate steps 1B to 4	

Figure 5.4: Budget-based design steps

An existing system is used to get a first guidance to fill the budget. In general the budget of a new system is equal to the budget of the old system, with a number of explicit improvements. The improvements must be substantiated with design estimates and simulations of the new design. Of course the new budget must fulfill the specification of the new system; sufficient improvements must be designed to achieve the required improvement.

5.2.4 Visualization

In the following three examples different actually used *visualizations* are shown. These three examples show that a multi-domain method does not have to provide a single solution, often several useful options exist. The method description should provide some guidance in choosing a visualization.

5.2.5 Guidelines

A *decomposition* is the foundation of a budget. No universal recipe exists for the decomposition direction. The construction decomposition and the functional decomposition are frequently used for this purpose. Budgets are often used as part of the design specification. From project management viewpoint a decomposition is preferred that maps easily on the organization.

The architect must ensure the *manageability* of the budgets. A good budget has tens of quantities described. The danger of having a more detailed budget is loss of overview.

The simplification of the design into budgets introduces design constraints. Simple budgets are entirely static. If such a simplification is too constraining or too

costly then a dynamic budget can be made. A dynamic budget uses situationally determined data to describe the budget in that situation. For instance, the amount of memory used in the system may vary widely depending on the function or the mode of the system. The budget in such a case can be made mode-dependent.

5.2.6 Example of overlay budget for wafersteppers

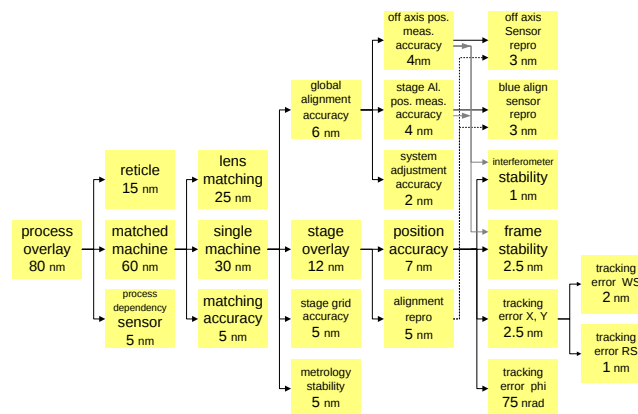


Figure 5.5: Example of a quantified understanding of overlay in a waferstepper

Figure 5.5 shows a graphical example of an “overlay” budget for a waferstepper. This figure is taken from the *System Design Specification* of the ASML TwinScan system, although for confidentiality reasons some minor modifications have been applied.

The *goal* of the overlay budget is:

- to provide requirements for subsystems and components.
- to enable measurements of the actual contributions to the overlay during the design and integration process, on functional models or prototypes.
- to get early feedback of the overlay design by measurements.

The *steps* taken in the creation, use and validation of the budget follow the description of Figure 5.4. This budget is based on a model of the overlay functionality in the waferstepper (step 1B). The system engineers made an explicit model of the overlay. This explicit model captures the way in which the contributions accumulate: quadratic summation for purely stochastic, linear addition for systematic effects and some weighted addition for mixed effects. The waferstepper budget is created by measuring the contributions in an existing system (step 1A). At the same time a top-down budget is made, because the new generation of machines needs

a much better overlay specification than the old generation (step 1C). In discussions with the subsystem engineers, design alternatives are discussed to achieve the required improvements (step 2 and 3). The system engineers also strive for measurable contributions. The measurability of contributions influences the subsystem specifications. If needed the budget or the design is changed on the basis of this feedback (step 4).

Two *visualizations* were used for the overlay budget: tables and graphs, as shown in Figure 5.5.

The overlay budget plays a crucial role in the development of wafersteppers. The interaction between the system and the customer environment is taken into account in the budget. However, many open issues remain at this interface level, because the customer environment is outside the scope of control and a lot of customer information is highly confidential. The translation of this system level budget into mono-disciplinary design decisions is still a completely human activity with lots of interaction between system engineers and mono-disciplinary engineers.

5.2.7 Example of memory budget for Medical Imaging Workstation

The *goal* of the memory budget for the medical imaging workstation is to obtain predictable and acceptable system performance within the resource constraints dictated by the cost requirements. The *steps* taken to create the budget follow the order as described in Figure 5.4. The *visualization* was table based.

<i>memory budget in Mbytes</i>	code	obj data	bulk data	total
shared code	11.0			11.0
User Interface process	0.3	3.0	12.0	15.3
database server	0.3	3.2	3.0	6.5
print server	0.3	1.2	9.0	10.5
optical storage server	0.3	2.0	1.0	3.3
communication server	0.3	2.0	4.0	6.3
UNIX commands	0.3	0.2	0	0.5
compute server	0.3	0.5	6.0	6.8
system monitor	0.3	0.5	0	0.8
application SW total	13.4	12.6	35.0	61.0
UNIX Solaris 2.x				10.0
file cache				3.0
total				74.0

Figure 5.6: Example of a memory budget

The rationale behind the budget can be used to derive *guidelines* for the creation of memory budgets. Figure 5.6 shows an example of an actual memory budget for a medical imaging workstation from Philips Medical Systems. This budget decomposes the memory into three different types of memory use: code ("read only" memory with the program), object data (all small data allocations for control and

bookkeeping purposes) and bulk data (large data sets, such as images, which is explicitly managed to fit the allocated amount and to prevent memory fragmentation). The difference in behavior of these three memory types is an important reason to separate into different budget entries. The operating system and the system infrastructure, at the other hand, provide means to measure these three types at any moment, which helps for the initial definition, for the integration, and for the verification.

The second decomposition direction is the *process*. The number of processes is manageable, since processes are related to specific development teams. Also in this case the operating system and system infrastructure support measurement at process level.

The memory budget played a crucial role in the development of this workstation. The translation of this system level budget into mono-disciplinary design decisions was, as in the case of overlay in wafersteppers, a purely human activity. The software discipline likes to abstract away from physical constraints, such as memory consumption and time. A lot of room for improvement exists at this interface between system level design and mono-disciplinary design.

5.2.8 Example of power budget visualizations in document handling

Visualizations of a budget can help to share the design issues with a large multi-disciplinary team. The tables and graphs, as shown in the previous subsections, and as used in actual practice, contain all the information about the resource use. However the *hot spots* are not emphasized. The visualization does not help to see the contributions in perspective. Some mental activity by the reader of the table or figure is needed to identify the design issues.

Figure 5.7 shows a visualization where at the top the physical layout is shown and at the bottom the same layout is used, however the size of all units is scaled with the allocated power contribution. The bottom visualization shows the *power foot print* of the document handler units.

Figure 5.8 shows an alternative power visualization. In this visualization the energy transformation is shown: incoming electrical power is in different ways transformed into heat. The width of the arrows is proportional to the amount of energy. This visualization shows two aspects at the same time: required electrical power and required heat disposition capacity, two sides of the same coin.

5.2.9 Evolution of budget over time

Figure 5.9 shows a classification for budget types. It will be clear that already with four different attributes the amount of different types of budgets is large. Every type of budget might have its own peculiarities that have to be covered by the method. For instance, worst case budgets need some kind of over-kill prevention.

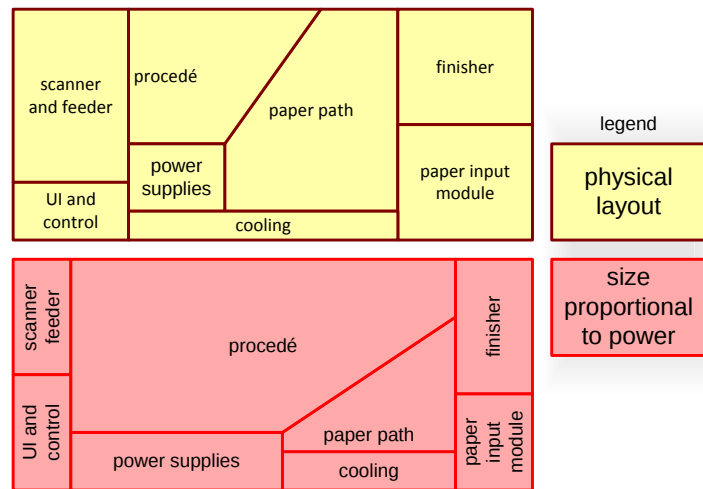


Figure 5.7: Power Budget Visualization for Document Handler

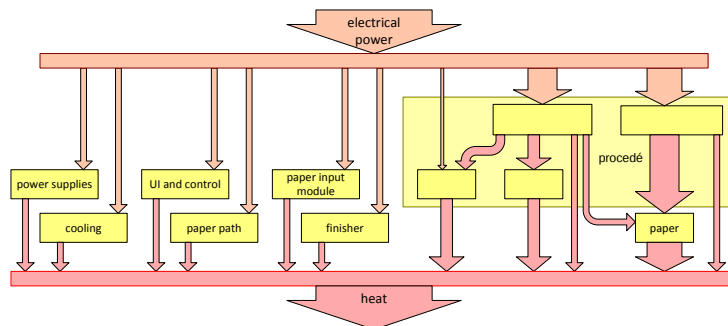


Figure 5.8: Alternative Power Visualization

Add to these different types the potential different purposes of the budget (design space exploration, design guidance, design verification, or quality assurance) and the amount of method variations explodes even more.

We recommend to start with a budget as simple as possible:

- coarse guesstimate values
- typical case
- static, steady state system conditions
- derived from existing systems

This is also shown in Figure 5.10. This figure adds the later evolutionary increments, such as increased accuracy, more attention for boundary conditions and

static	dynamic
typical case	worst case
global	detailed
approximate	accurate

is the budget based on
wish, empirical data, extrapolation,
educated guess, or expectation?

Figure 5.9: What kind of budget is required?

dynamic behavior.

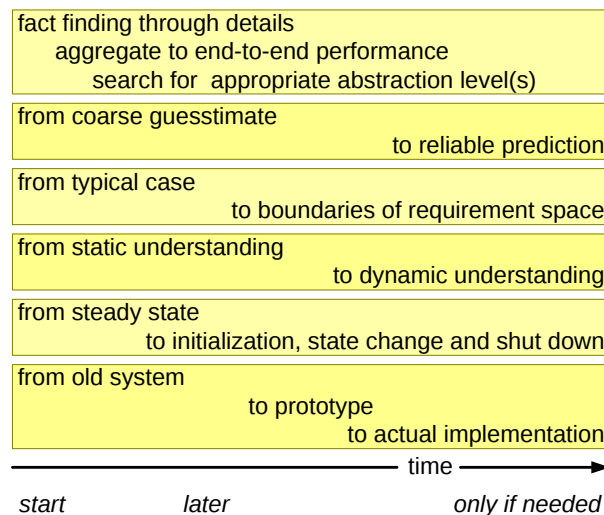


Figure 5.10: Evolution of Budget over Time

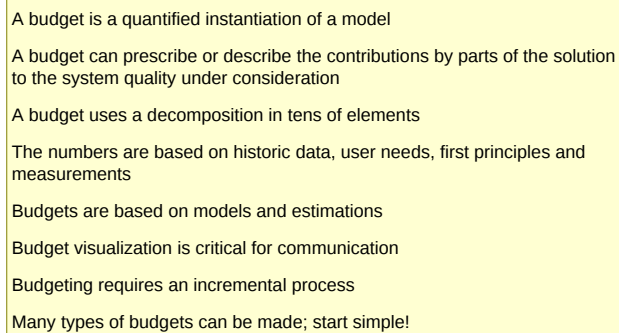
However, some fact finding has to take place before making the budget, where lots of details can not be avoided. Facts can be detailed technical data (memory access speed, context switch time) or at customer requirement level (response time for specific functions). The challenge is to mold these facts into information at the *appropriate* abstraction level. Too much detail causes lack of overview and understanding, too little detail may render the budget unusable.

5.2.10 Potential applications of budget method

For instance the following list shows potential applications, but this list can be extended much more. At the same time the question arises whether *budget-based design* is really the right submethod for these applications.

- resource use (CPU, memory, disk, bus, network)
- timing (response time, latency, start up, shutdown)
- productivity (throughput, reliability)
- image quality (contrast, signal to noise ratio, deformation, overlay, depth-of-focus)
- cost, space, time, effort (for instance expressed in lines of code)

5.3 Summary



A budget is a quantified instantiation of a model

A budget can prescribe or describe the contributions by parts of the solution to the system quality under consideration

A budget uses a decomposition in tens of elements

The numbers are based on historic data, user needs, first principles and measurements

Budgets are based on models and estimations

Budget visualization is critical for communication

Budgeting requires an incremental process

Many types of budgets can be made; start simple!

Figure 5.11: Summary of budget based design

5.4 Acknowledgements

The Boderc project contributed to the budget based design method. Figure 5.12 shows the main contributors.

The Boderc project contributed to Budget Based Design. Especially the work of *Hennie Freriks, Peter van den Bosch (Océ), Heico Sandee and Maurice Heemels (TU/e, ESI)* has been valuable.

Figure 5.12: Colophon

Chapter 6

Formula Based Performance Design



6.1 Introduction

We recommend to model performance by using simple, secondary school, mathematical formulas. Frequently designers tend to start using more advanced techniques and formalisms, in an attempt to be accurate. In this paper we discuss an approach using simple mathematical formulas, starting with the most simple formulas and refining these formulas as far as needed.

6.2 Using n-order formulas

The basis for most performance models are simple mathematical formulas, using secondary school math. The challenge is to keep the models as simple as possible, as discussed in the section about control design. We can express the degree of detail in formulas by the order of the formula. Figure 6.1 shows such classification.

Figure 6.2 shows an example of a highly simplified model of the CPU load for image processing. This formula assumes that the CPU load is directly proportional to the number of pixels plus some time to perform the user interface tasks. We call such a formula, where only the main parameter is present, a zeroth order formula.

It could be that the 0-order formula does not work well enough, for example because overhead is significant. In Figure 6.3 the biggest overhead contribution is added to the formula, in this example the context switch overhead.

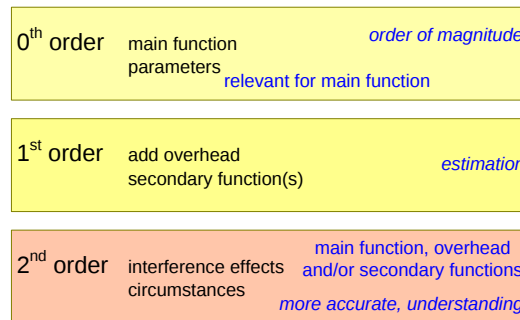


Figure 6.1: Theory Block 1: n Order Formulas

$$t_{\text{cpu total}} = t_{\text{cpu processing}} + t_{\text{UI}}$$

$$t_{\text{cpu processing}} = n_x * n_y * t_{\text{pixel}}$$

Figure 6.2: CPU Time Formula Zero Order

However, in a heavily loaded system may suffer additional loads due to the context switches, the so-called second order effects. In Figure 6.4 these second order effects are added to the formula. The second order impact may depend on the type of system load. The second order terms might be parameterized to express this relation. For example signal processing loads might cause low penalties, due to high cache efficiency, while control processing might be much more sensitive to these effects.

6.3 Example of n-order formulas in MR reconstruction

The reconstruction of MR images is a processing intensive operation. Fast reconstructions are beneficial for the throughput of MRI scanners and are prerequisite

$$t_{\text{cpu total}} = t_{\text{cpu processing}} + t_{\text{UI}}$$

$$+ t_{\text{context switch overhead}}$$

Figure 6.3: CPU Time Formula First Order

$$t_{\text{cpu total}} = t_{\text{cpu processing}} + t_{\text{UI}} + t_{\text{context switch overhead}} +$$

$t_{\text{stall time due to cache efficiency}} + t_{\text{stall time due to context switching}}$

signal processing: high efficiency
 control processing: low/medium efficiency

Figure 6.4: CPU Time Formula Second Order

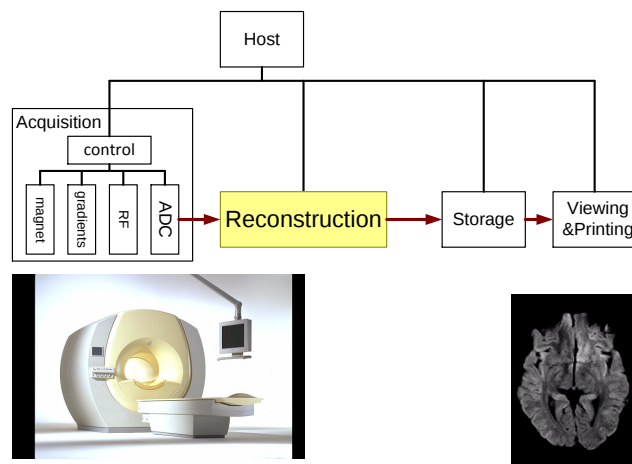


Figure 6.5: MR Reconstruction Context

for a number of performance critical applications. Figure 6.5 shows a simplified block diagram of an MRI scanner, the context of the MR reconstruction. The MR data is digitized in the acquisition subsystem and transferred to the reconstruction subsystem. The reconstructed images are stored in the data base and viewed at the operator or viewing console. All subsystems are controlled by a central host computer.

In Figure 6.6 a visualization and mathematical formulas are used in combination to model the performance of the MR reconstruction. The visualization shows the processing steps that are performed as reconstruction. Above the arrows it is shown what the size of the data matrices is at that phase.

This 0-order model uses the Fast Fourier Transform (FFT) as the dominating term contributing to the performance. Most operations are directly proportional to the matrix size shown above the formulas. The FFT itself is an order $n \log(n)$ term, parameterized with its corresponding load c_{fft} .

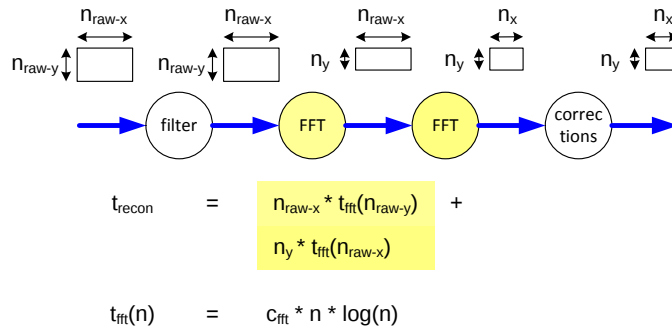


Figure 6.6: MR Reconstruction Performance Zero Order

Typical FFT, 1k points ~ 5 msec
(scales with $2 * n * \log(n)$)

using:

$$\begin{aligned}
 n_{\text{raw-x}} &= 512 & t_{\text{recon}} &= n_{\text{raw-x}} * t_{\text{fft}}(n_{\text{raw-y}}) + \\
 n_{\text{raw-y}} &= 256 & & n_y * t_{\text{fft}}(n_{\text{raw-x}}) + \\
 n_x &= 256 & & 512 * 1.2 + 256 * 2.4 \\
 n_y &= 256 & & \sim 1.2 \text{ s}
 \end{aligned}$$

Figure 6.7: Zero Order Quantitative Example

Unfortunately the formulas don't tell us much without quantification. Figure 6.7 provides us with some quantified input based on a FFT micro-benchmark: an FFT on thousand points executes in about 5 msecs (typical performance figures for processing hardware around 1990). The figure takes one typical use case, where a $512*256$ raw image is reconstructed on a $256*256$ image, to calculate the reconstruction performance. For this use case and assumptions we get 1.2 seconds.

Figure 6.9 extends the model to also take the non-FFT processing into account. These operations filter the raw data and perform some simple corrections on the image. Both operations are proportional to the number of pixels that is processed.

Figure 6.9 provides the quantifications obtained by micro-benchmarking both operations: 2 msec to process 1k points. Using the same numbers as Figure 6.7 we get for filtering $512 * 256 * 2 / 1024 \text{ms} \approx 0.26 \text{s}$ and for correction $256 * 256 * 2 / 1024 \text{ms} \approx 0.13 \text{s}$. Both processing steps can not be ignored compared to the FFT operation!

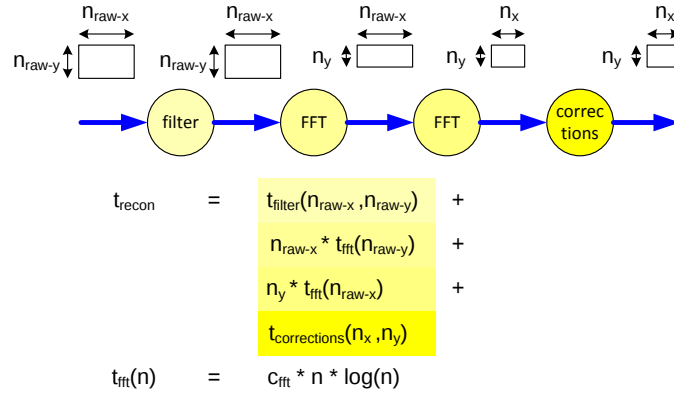


Figure 6.8: MR Reconstruction Performance First Order

Typical FFT, 1k points ~ 5 msec
(scales with $2 * n * \log(n)$)

Filter 1k points ~ 2 msec
(scales linearly with n)

Correction ~ 2 msec
(scales linearly with n)

Figure 6.9: First Order Quantitative Example

Finally we add bookkeeping and I/O type operations to the formula, see Figure 6.10. In practice both terms often ruin the performance of well designed processing kernels, mostly by a lack of attention.

6.4 Summary

We have shown that performance can be modeled by starting with the formula for the main function and its parameters. This formula is refined by adding factors that significantly contribute to the (non-)performance.

We used MRI reconstruction as an example of these types of formulas. The formulas are limited to multiplications and logarithms. In this example we have to add quite some factors outside of the main functionality to obtain a usable performance model: simple correction and filter functions, bookkeeping, data-restructuring, and input/output. We also showed that the formulas provide insight, especially the impact of the different parameters, but that actual quantifications also add insight in actual performance numbers and the relevance of the different terms.

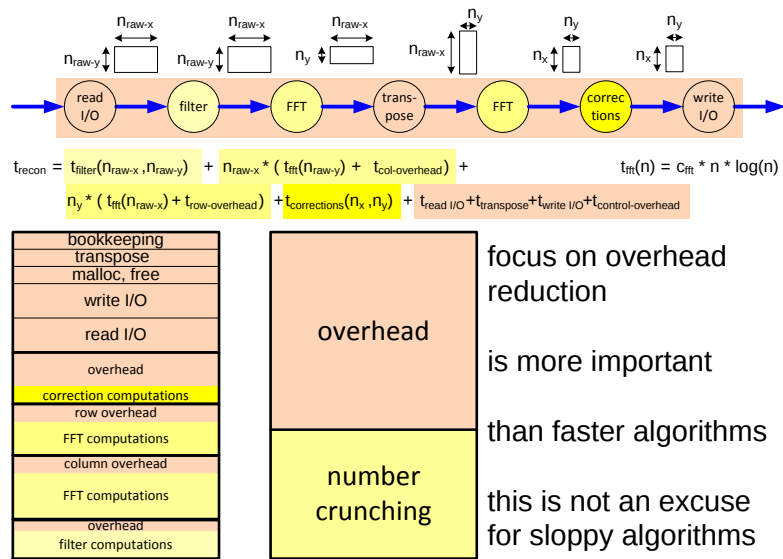


Figure 6.10: MR Reconstruction Performance Second Order

6.5 Acknowledgements

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

Bibliography

- [1] H Gomma. *Software Design Methods for Real-time Systems*. Addison-Wesley, 1993.
- [2] John L. Hennessy, David A. Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.
- [3] Gerrit Muller. The system architecture homepage. <http://www.gaudisite.nl/index.html>, 1999.
- [4] Gerrit Muller. Architectural reasoning explained. <http://www.gaudisite.nl/ArchitecturalReasoningBook.pdf>, 2002.

History

Version: 1.1, date: March 19, 2008 changed by: Gerrit Muller

- added Elevator Modeling

Version: 0, date: September 4, 2007 changed by: Gerrit Muller

- Created, no changelog yet