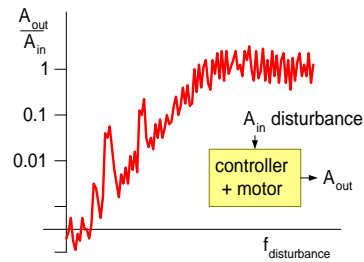


# How to Characterize SW and HW to Facilitate Predictable Design?

-



Gerrit Muller

Embedded Systems Institute

Den Dolech 2 (Laplace Building 0.10) P.O. Box 513, 5600 MB Eindhoven The Netherlands

[gerrit.muller@embeddedsystems.nl](mailto:gerrit.muller@embeddedsystems.nl)

## Abstract

SW engineering is quite different from conventional engineering disciplines. Major difference is the lack of quantification and the related analysis techniques. We will shortly explore an example from control engineering: How are control elements characterized and analyzed? We propose a similar approach for performance characterization and analysis of digital hardware and software platforms.

### 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/>

version: 1.0

status: preliminary draft

February 10, 2011

# 1 Introduction

What happens if you ask a software engineer to quantify the product under construction? Well, then in most cases this software engineer will quantify the project rather than the *system of interest*. Frequently used quantifications of software projects are:

- man-years
- lines-of-code
- problem reports
- code-complexity
- fault density
- release schedule

We formulate the following problem statement for this paper:

Qualities of SW intensive systems, such as performance, are *emerging* i.s.o. *predictable* properties. The SW engineering discipline today is *process* oriented, quantities are process metrics. The System Of Interest (SOI) is designed from *behavioral* point of view. Conventional Engineering disciplines design the SOI with *quantitative* techniques.

In this paper we will look at another discipline, control design, to see how quantitative engineering is done here. We use the same kind of approach for performance of software. The question arises what other aspects of software can be approached more quantitatively. Many aspects need more research to reach the level of practical applicability.

## 2 Example of quantified engineering in control

The discipline of control design is well established. In this discipline formalisms, techniques and methods are available to design feedback based controllers for electrical motors.

Figure 1 shows a typical block diagram from this domain. The goal is to design a controller for a given motor such that the requested set point is reached despite the presence of disturbances. To achieve this goal the position of the motor is measured and used for feedback by the controller. The comparison of the actual motor position with the required set point gives the deviation or error in positioning.

The combination of the controller and motor is *characterized* (or *identified*) by measuring the disturbance transfer and the tracking response. The *disturbance transfer* is measured by scanning through the disturbance frequencies. For every

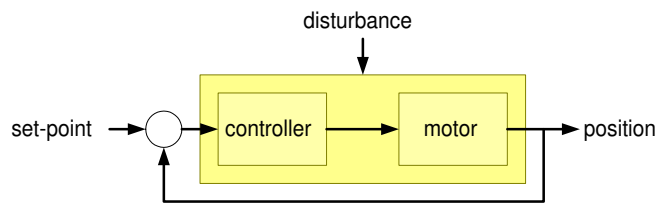


Figure 1: Block Diagram Control Measurement

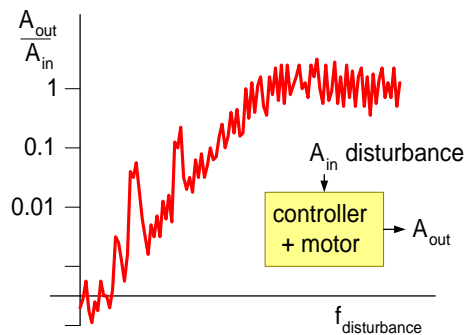


Figure 2: Measuring Disturbance Transfer

disturbance frequency the relative impact on the output is measured, as shown in Figure 2.

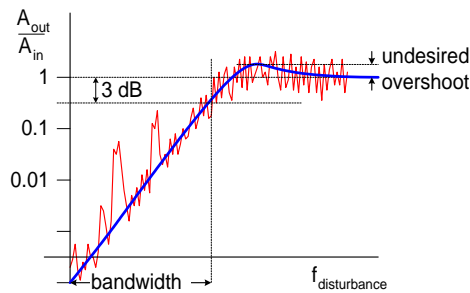


Figure 3: Idealized Disturbance Transfer

Based on available control design know how a curve is fitted through the measured points. This curve is parameterized by the *bandwidth* (where is the damping less than 3 dB) and by the (undesired) *overshoot* (how much is the disturbance amplified at higher frequencies), see Figure 3.

The *tracking response* is measured by scanning through the frequencies for all set point frequencies. For every frequency the relative impact on the output is measured.

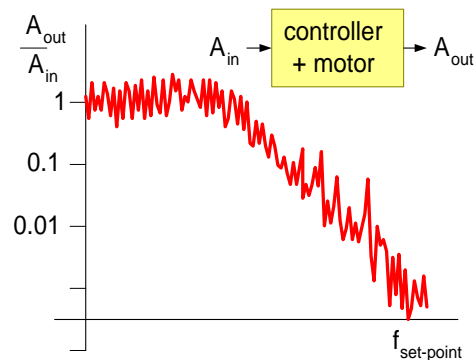


Figure 4: Measuring Tracking Response

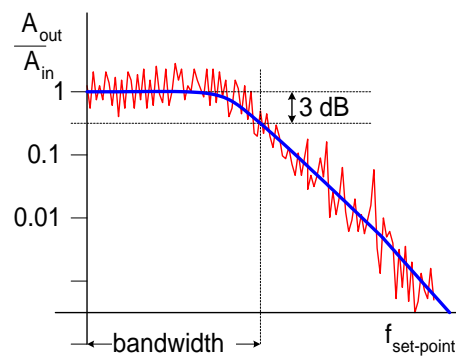


Figure 5: Idealized Tracking Response

The available know how provides a parameterized curve for the tracking response. The parameter of this curve is the bandwidth, up to which frequency does the control track the set point within 3 dB? Figure 5 shows a typical *tracking transfer* curve.

Note that for a good controller the tracking bandwidth and the disturbance response bandwidth should approximately be the same. In the working range of the controller the tracking response should be good as well as the damping of the disturbances.

The know how of control design also provides black box models of controllers, as shown in Figure 6. Using parameters such as disturbance transfer bandwidth, tracking bandwidth, overshoot, and controller parameters, such as the order of the controller, the performance of the controller can be analyzed: How well will the response follow the stimulus. The black box model is a set of simple mathematical formulas with a physical interpretation of the parameters and variables.

An essential part of the control design know how is what is hidden by the black box model. Control designers know much more about the controller-motor combi-

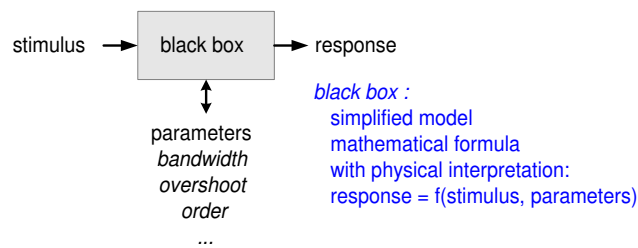
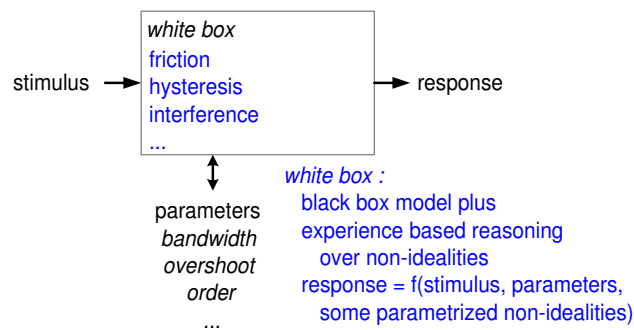


Figure 6: Black Box Model



challenge: to know what non-idealities to ignore and to ignore as much as possible

Figure 7: White Box Model

nation than is represented in the (highly) simplified black box model. Figure 7 shows the white box model of the controller-motor combination, where many non-idealities, such as friction, hysteresis, and interference become visible. The challenge for the control design discipline is to know what non-idealities to ignore and to be able to ignore as many non-idealities as possible.

Figure 8 shows the different parts of design control know how that have been discussed, plus know how that has been used implicit:

**typical controllers** What are typical controllers and their parameterization, e.g. differential, proportional, integrating.

**typical motors** What are typical motors and their parameterizations

**non-idealities of motors and controllers** What are typical non-idealities, such as friction, hysteresis, and interference.

**measurements and representations** What should be measured and how to present the results? What stimuli should be used, when to use logarithmic axes, et cetera.

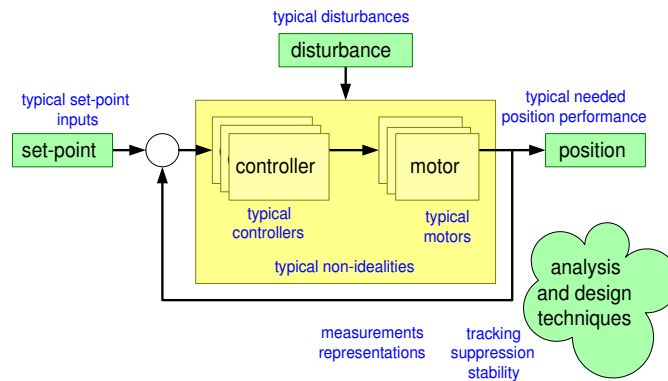


Figure 8: Control Engineering Knowledge

**set-point inputsperformance** What performance is typically required from the controller-motor combination given the typical set-point inputs?

**analysis and design techniques** What should be measured or calculated? How to determine the type of controller and its parameterization? How to analyze tracking performance, suppression of disturbances and stability of the combination?

**disturbances** What are typical disturbances, what is their impact?

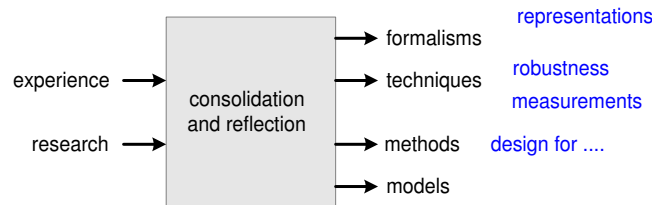


Figure 9: Summary of Control Engineering

Figure 9 summarizes what we can learn from a more mature discipline. Based on experience and research the discipline consolidated this know how by reflection into *formalisms* (and representations), *techniques* (such as robustness analysis and characterization measurements), *methods* (how to design a controller that . . . ), and models capturing the know how in mathematical models with physical interpretations.

### 3 Quantified engineering of software performance

We will discuss the performance of software as an example of a property that lends itself well for quantified engineering techniques. Despite the quantitative nature of performance, many projects where a lot of software is involved, suffer from performance problems before and after formal release.

#### 3.1 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 10. How do we predict or estimate the expected performance based on this code fragment?

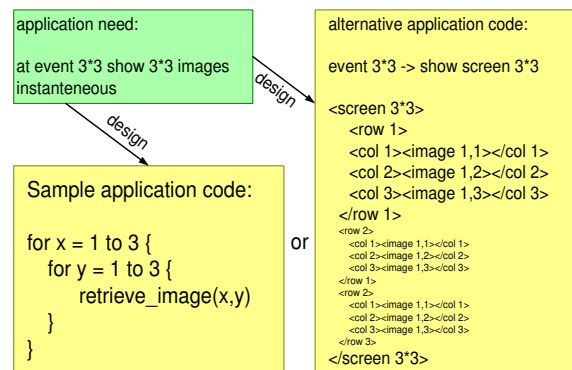


Figure 10: Image Retrieval Performance

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 11, 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.

However, the system might be slightly more complex, as shown in Figure 12. 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

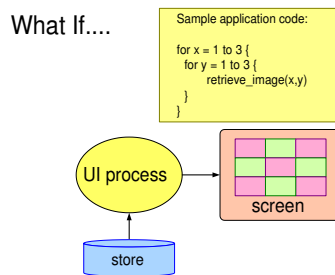


Figure 11: Straight Forward Read and Display

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.

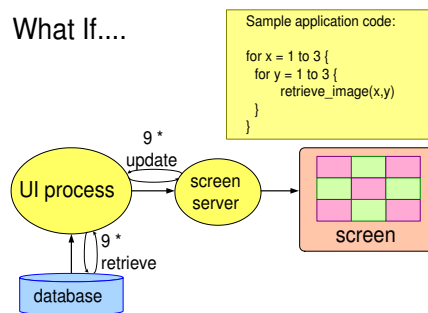


Figure 12: More Process Communication

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 amount of pixel data. The initial estimation ignores the cost of meta-information, because the amount of data is insignificant. However, the chosen implementation does have a significant impact on the cost of meta-information handling. Figure 13 shows an example where the attributes of the meta-information are internally mapped on COM objects. The implementation causes a complete “factory” construction for every attribute that is retrieved. The cost of such a construction is  $80\mu sec$ . With 100 attributes per image we get a total construction overhead of  $9 \cdot 100 \cdot 80\mu s = 72ms$ . This cost is significant, because it is in the same order of magnitude as image transfer and rendering operations.

Figure 14 shows I/O overhead as a last example of potential hidden costs. If the granularity of I/O transfers is rather fine, for instance based on image lines, then the I/O overhead becomes very significant. If we assume that images are  $512^2$ , and if we assume  $t_{I/O} = 1ms$ , then the total overhead becomes  $9 \cdot 512 \cdot 1ms \approx 4.5s$ !

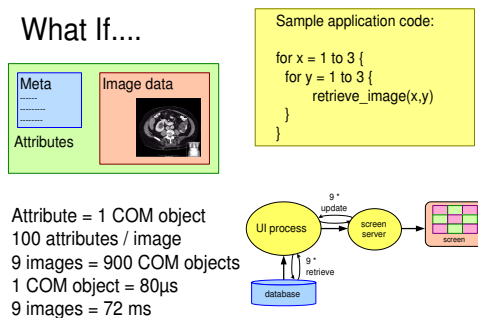


Figure 13: Meta Information Realization Overhead

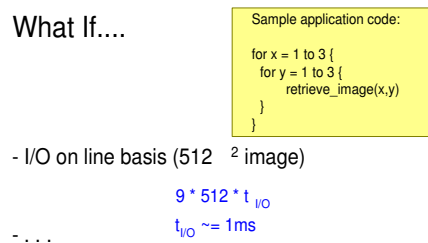


Figure 14: I/O overhead

### 3.2 Problem Statement

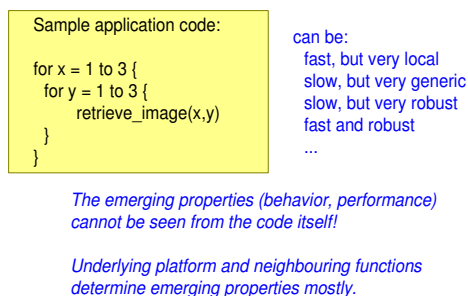


Figure 15: Non Functional Requirements Require System View

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 15 shows the conclusions based on the previous *What if* examples.

Figure 16 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

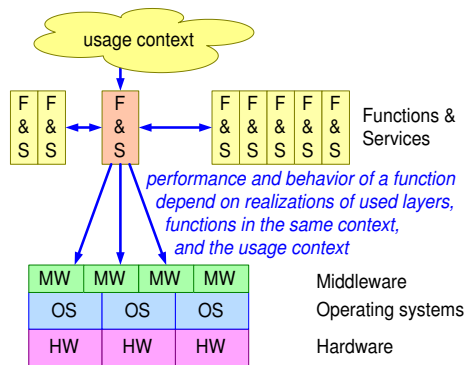
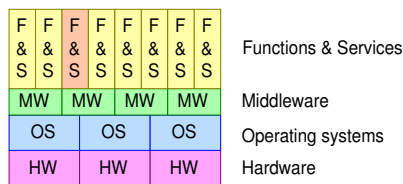


Figure 16: Function in System Context

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.



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

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

Figure 17: Challenge

Figure17 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).

### 3.3 Layered benchmarking approach

We propose to tackle the performance analysis by measuring and analyzing the system at several levels, as shown in Figure 18. 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

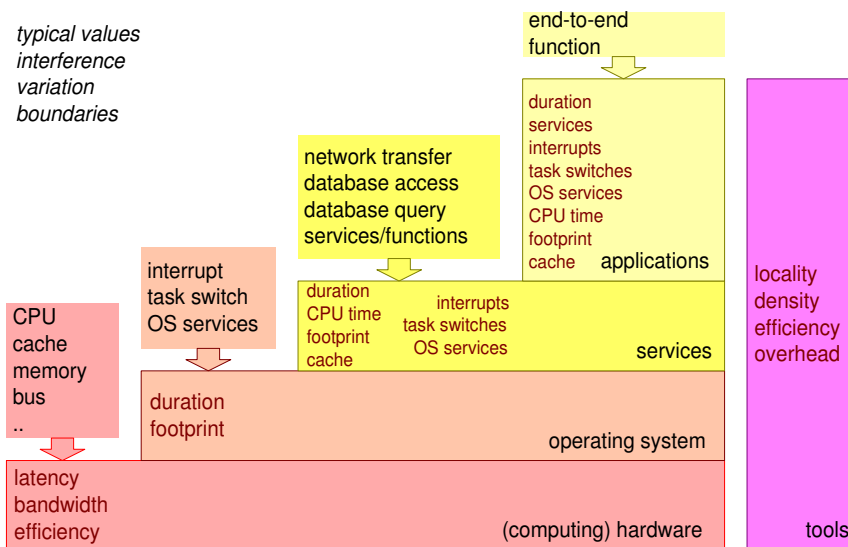


Figure 18: Layered Benchmarking

we look for natural layers or subsystems. 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.

### 3.4 Micro-benchmarks of hardware and OS

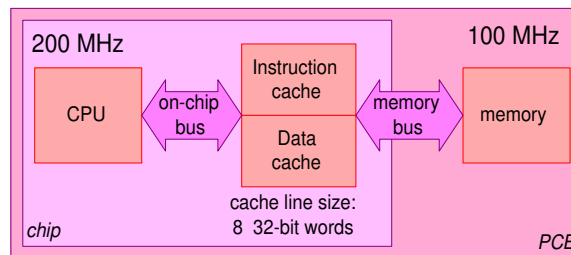


Figure 19: Case 2: ARM9 Cache Performance

An up to date example of micro-benchmarking uses the ARM9 as case, see Figure 19. 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.

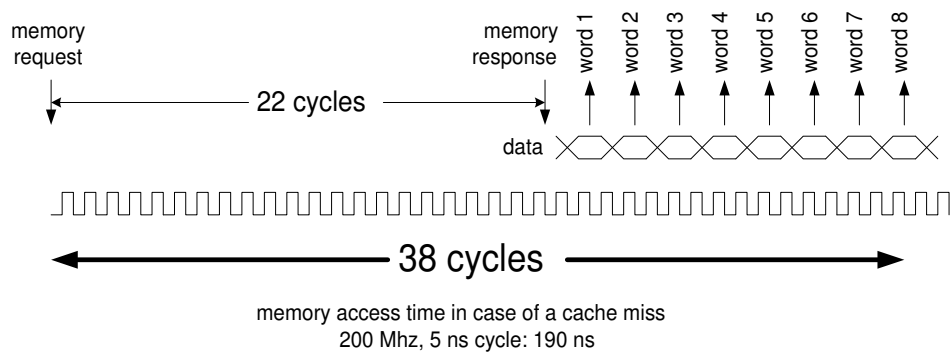


Figure 20: Example Hardware Performance

Figure 20 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.

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

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

Figure 21: Actual ARM Figures

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 21. 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 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.

Figure 22 shows the impact of context switches on system performance for different context switch rates. Both parameters  $t_{\text{contextswitch}}$  and  $n_{\text{contextswitch}}$  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  $t_{\text{contextswitch}} = 10\mu\text{s}$  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  $t_{\text{contextswitch}} = 2\mu\text{s}$  would assess 50000 context switches as significant, but not yet problematic.

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

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

Figure 22: Context Switch Overhead

### 3.5 From micro-benchmarks to system performance

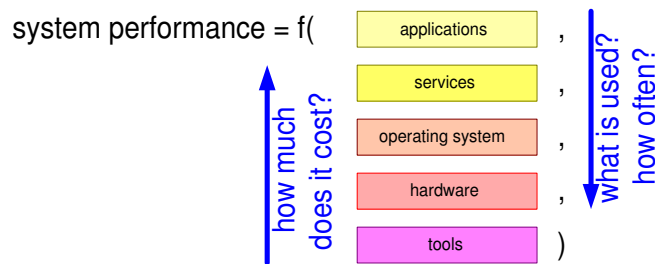


Figure 23: Performance as Function of all Layers

All data gathering activities must be processed in an intelligent way into a set of higher level diagrams and models. For example the micro-benchmarks generate a lot of data points that should be turned into a layered performance model, visualized in Figure 23. This performance model is **not** one single formula, but a more a set of related formula's. For instance the interrupt handling and task switching duration can be expressed in the lower layers as function of hardware parameters:

$$t_{\text{interrupthandling}} = f(\text{CPU}_{\text{speed}}, \text{cache}_{\text{size}}, \text{OS})$$

At the higher service layer a typical value for the interrupt handling time is used, without the complicating dependencies on hardware and operating system:

$$t_{\text{service}} = n_{\text{interrupts}} * 10\mu\text{s}(t_{\text{interrupthandling}}) + g_{\text{service}}(\text{inputdata}, t_{\text{transaction}}, t_{\text{network}})$$

We recommend to work bottom-up and top-down concurrently. Bottom-up means start to measure the bottom layer and work upwards. Try to understand the higher layer numbers in terms of the lower layer data, during this bottom-up process. Top-down starts at the end user side, by measuring end-to-end performance. The end-to-end performance can be decomposed in contributions of the subsystems or functions involved in this operation. The top-down approach requires a lot of *reasoning*:

- what is happening and what *should* be happening?
- how much time is contributed by the different functions?
- what are the main lower level parameters that determine this amount of time

At last the end-to-end performance should be explainable in terms of the lower level micro-benchmark results. By working concurrently bottom-up and top-down both activities can be limited to *relevant* measurements. In a system that does only have a few interrupts, the interrupt handling time might be ignored.

### 3.6 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 24 shows such classification.

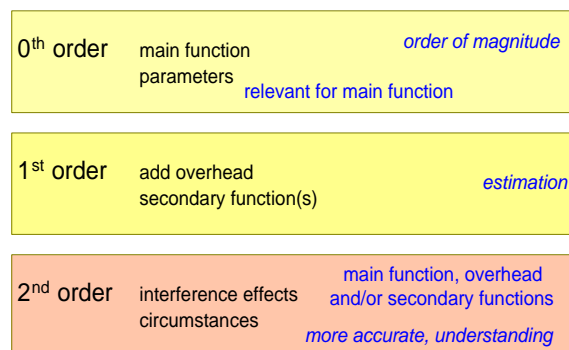


Figure 24: Theory Block 1: n Order Formulas

Figure 25 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.

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

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

Figure 25: CPU Time Formula Zero Order

It could be that the 0-order formula does not work well enough, for example because overhead is significant. In Figure 26 the biggest overhead contribution is added to the formula, in this example the context switch overhead.

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

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

Figure 26: CPU Time Formula First Order

However, in a heavily loaded system may suffer additional loads due to the context switches, the so-called second order effects. In Figure 27 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.

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

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

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

Figure 27: CPU Time Formula Second Order

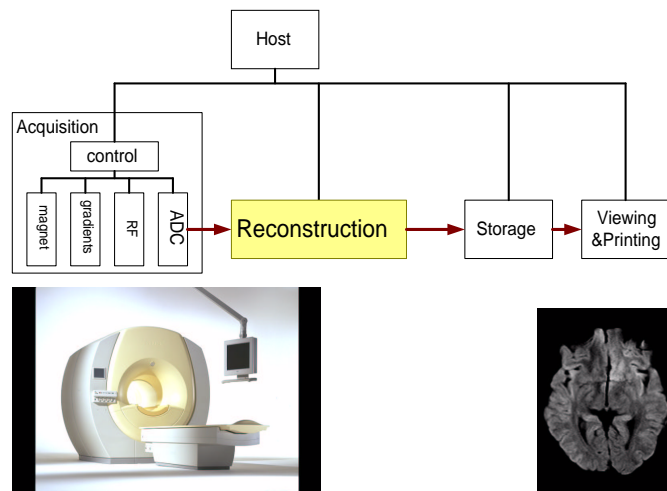


Figure 28: MR Reconstruction Context

### 3.7 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 for a number of performance critical applications. Figure 28 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.

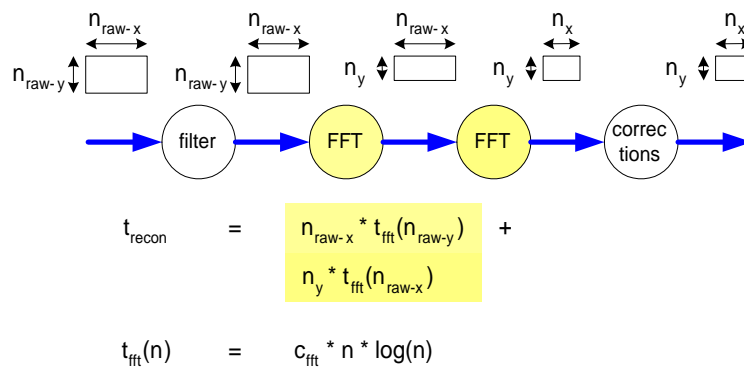


Figure 29: MR Reconstruction Performance Zero Order

In Figure 29 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, but the row and column overhead are already identified as significant and taken into account. 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  $c_{fft}$ .

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

using:

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

Figure 30: Zero Order Quantitative Example

Unfortunately the formulas don't tell us much without quantification. Figure 30 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. Also an assumption is made about the row and column overhead: both 0.2 msec. For this use case and assumptions we get 1.4 seconds.

Figure 32 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 32 provides the quantifications obtained by micro-benchmarking both operations: 2 msec to process 1k points. Using the same numbers as Figure 30 we get for filtering  $512 * 256 * 2/1024ms \approx 0.26s$  and for correction  $256 * 256 * 2/1024ms \approx 0.13s$ . Both processing steps can not be ignored compared to the FFT operation!

Finally we add bookkeeping and I/O type operations to the formula, see Figure 33. In practice both terms often ruin the performance of well designed processing kernels, mostly by a lack of attention.

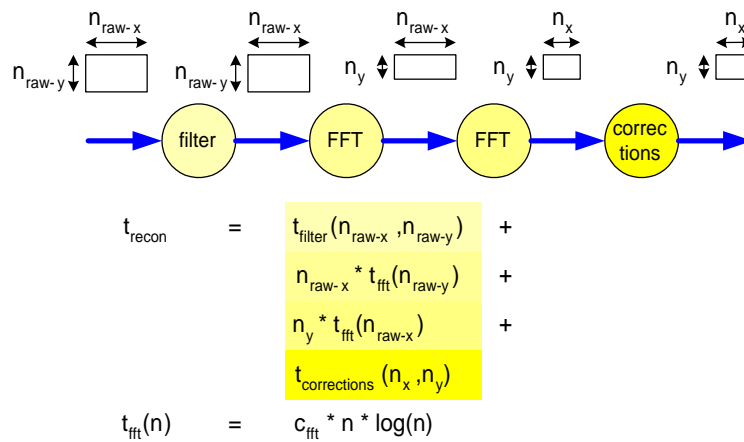


Figure 31: MR Reconstruction Performance First Order

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 32: First Order Quantitative Example

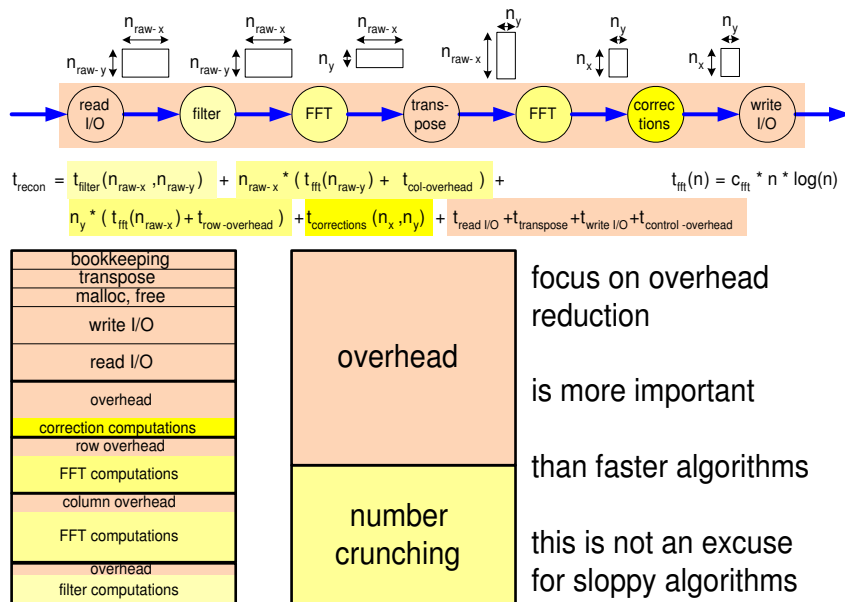


Figure 33: MR Reconstruction Performance Second Order

## 4 From behavioral to quantified software design

Current software engineering methods are only quantitative for project like data or in highly specialized niches such as algorithmic research. What type of research is needed to shift the software engineering discipline more towards quantified engineering? The starting point is to look for quantifiable aspects of software. Performance, for example response times or throughput, is one of the most easy quantifiable aspects. Performance at user level has impact on the design of the use of computing resources.

Another area of quantification is user interfaces: the number of user interactions and the amount screen real estate being used. The user interface quantification is also resource related, the user interface resources. The first research challenge is to identify the aspects to be quantified.

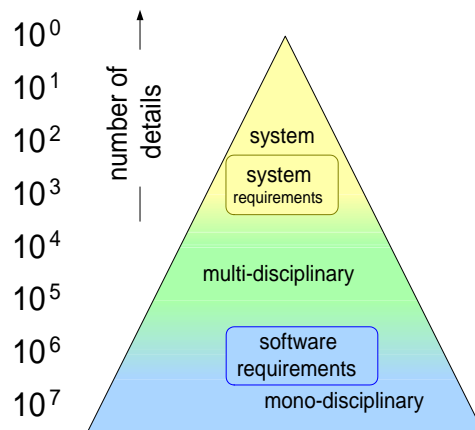


Figure 34: Software is Intimately Coupled with System

Both examples show that quantification of software as a product starts at system level. A system has desired properties that are captured as quantified system requirements. Then we apply a system and software design step to be able to create a software requirements specification. In this step from relatively few system requirements to software requirements the amount of details is increased with several orders of magnitude, see Figure 34. This increase is caused by the design step involved: we specify the software itself as components, classes and methods; a much finer granularity level than the software requirements. Most software requirements specifications are not quantified. The focus is most often on functionality or behavior.

Figure 35 shows a frequently occurring problem in system development: the disconnect of the customer world where systems are being used from the technical world where detailed design decisions are taken. The quantification of *system* requirements starts in a quantified understanding of the user world, the software

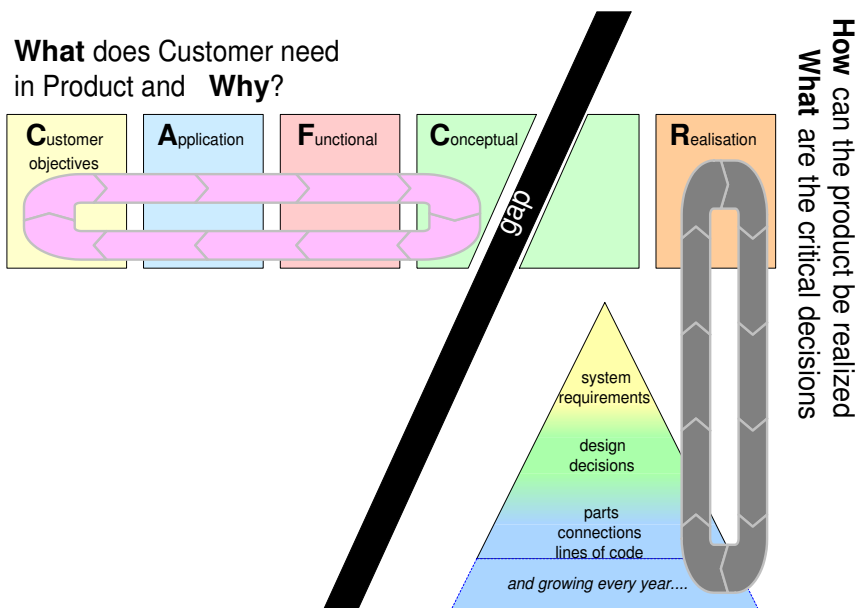


Figure 35: Most Quantifications Relate Context to Design!

realization has a big impact on the final system properties as experienced by customers.

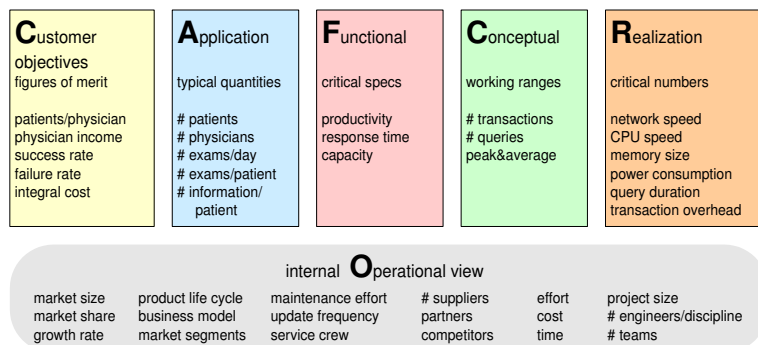


Figure 36: Example facts for “Electronic Patient Record”

Figure 36 shows quantifiable aspects in the CAFCR-model. It shows possible quantifications in terms of the *Customer Objectives*, what does the customer want to achieve, up to the *Realization*, what are the quantified properties of actual components.

## 5 Future Research to come closer to quantified software engineering

In the history of science different types of research have been applied, see Figure 37:

- Observational research
- Theory development
- Experimental research
- Fundamental research

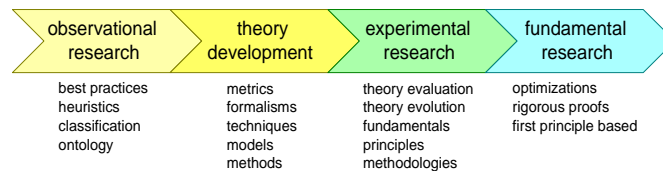


Figure 37: What Kind of Research is Needed?

In the early phases, when scientists did not yet have workable theories, *observational* research is the starting point. Observational research starts with describing the observations. For example case descriptions are valuable means. In system and software engineering we should capture best practices and heuristics. Research could also make a classification and ontology, based on the observations.

Theories are developed by trying to explain the observations. Numeric understanding is strived for by defining metrics. Theory development requires formalisms, techniques, models, and methods as means. These more abstract elements are often also the outcome of this type of research.

In the *experimental* phase the theories are tested. The more or less historical view of the early phases is replaced by an attempt to get a more objective validation of the theory. The main vehicle to achieve validation is experimentation. Theories are evaluated, often resulting in adaptation of the theory. The experimental researchers are searching for fundamentals, principles, and methodologies.

Once the field is well-defined, then more *fundamental* research becomes possible. For example the search for optimal solution, rigorous proofs, and first principle based derivations.

In practice all these types of research are concurrent and iterative.

Figure 38 summarizes what needs to be defined and researched in system and software engineering:

**entities** What are the most relevant entities used or created by the engineers?

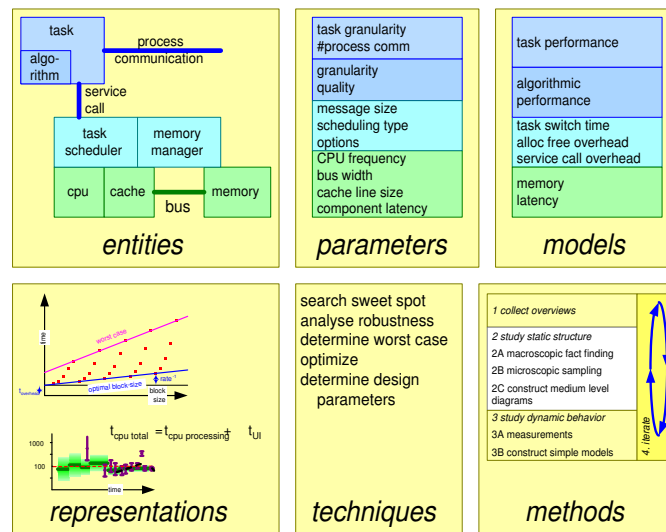


Figure 38: What needs to be defined and researched?

**parameters** How can these entities be parameterized?

**models** How can we model these entities and their properties in a useful way?

**representations and formalisms** What formalisms and representations do we need to capture the entities and parameters in order to explore them by means of models and techniques?

**techniques** What techniques help us to explore relevant properties?

**methods** What methods will help engineers to combine all of the above in a usable way?

## 6 Acknowledgements

Heico Sandee explained the Control Engineering approach. Peter van den Bosch, and Jaap van der Heijden provided feedback. The Boderc project ([www.esi.nl/Boderc](http://www.esi.nl/Boderc)) provided insight in the differences currently between software engineering as discipline and control engineering.

## References

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

## History

**Version: 1.0, date: September 3, 2007 changed by: Gerrit Muller**

- added section acknowledgements
- added text sections quantification and research
- increased status to draft

**Version: 0.2, date: August 31, 2007 changed by: Gerrit Muller**

- started to create a text version
- selected PENGcontrolDisturbanceMeasurement as logo
- decreased status to preliminary draft because of preliminary state of the text.

**Version: 0.1, date: September 8, 2006 changed by: Gerrit Muller**

- Added Research types
- System Software relation
- reordered last slides
- changed status to draft

**Version: 0, date: August 11, 2006 changed by: Gerrit Muller**

- Created, no changelog yet