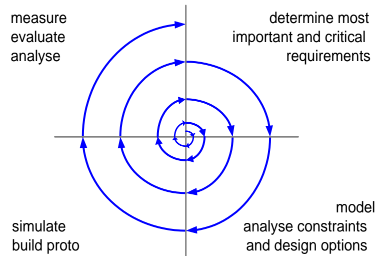
Performance Method Fundamentals

-



Gerrit Muller

Embedded Systems Institute

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

gerrit.muller@embeddedsystems.nl

Abstract

The Performance Design Methods described in this article are based on a multi-view approach. The needs are covered by a requirements view. The system design consists of a HW block diagram, a SW decomposition, a functional design and other models dependent on the type of system. The system design is used to create a performance model. Measurements provide a way to get a quantified characterization of the system. Different measurement methods and levels are required to obtain a usable characterized system. The performance model and the characterizations are used for the performance design. The system design decisions with great performance impact are: granularity, synchronization, priorization, allocation and resource management. Performance and resource budgets are used as tool.

The complete course ASP$^{\text{TM}}$ is owned by Embedded Systems Institute. To teach this course a license from Embedded Systems Institute is required. This material is preliminary course material. The final material and course information can be found at: www.esi.nl/cursus.

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

version: 0.2 status: draft July 1, 2011

1 Introduction

The performance of a system is determined by the hardware deign, 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.

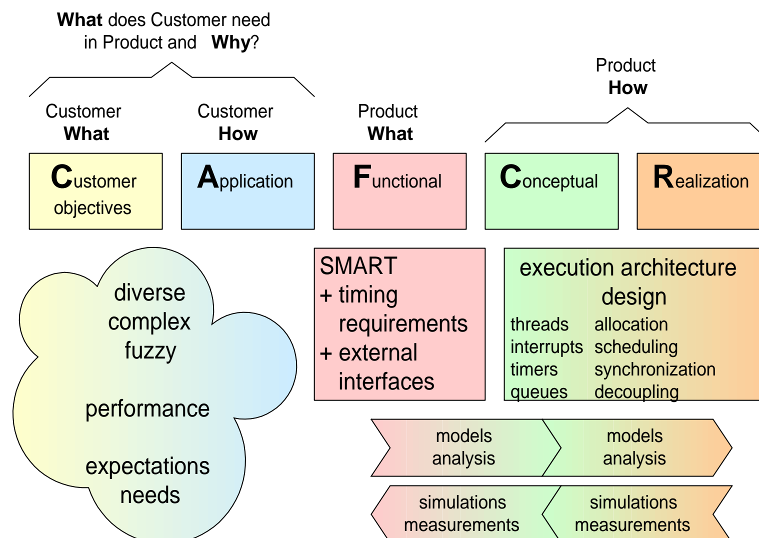


Figure 1: Positioning in CAFCR

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

Step	Notes
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, priorization, allocation, resource management
Re-iterate all steps	are the right requirements addressed, refine diagrams, measurements, models, and improve design

Figure 2: Top-level Performance Design Method

2 Incremental approach

Figure 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.

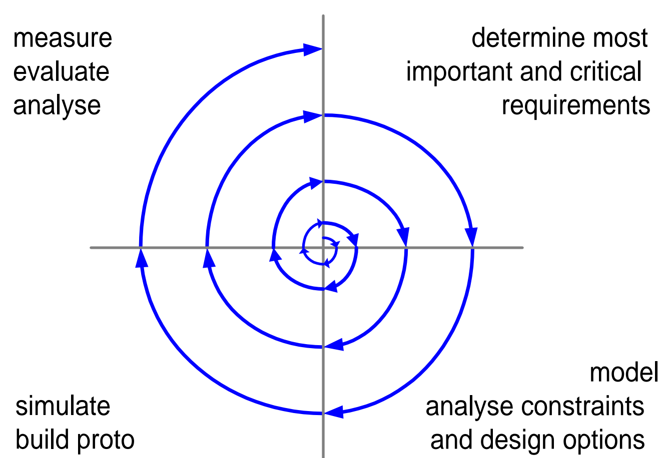


Figure 3: Incremental approach

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 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.

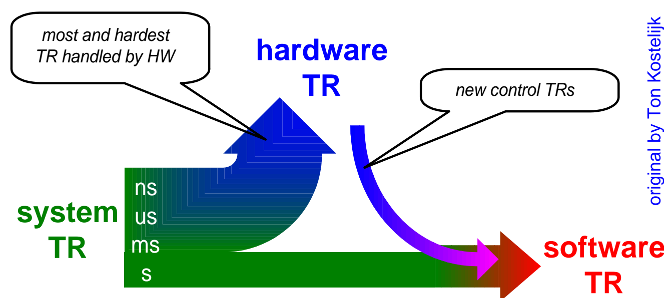


Figure 4: Decomposition of system TR in HW and SW

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 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.

The precision of the quantification increases during the project. Figure 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?

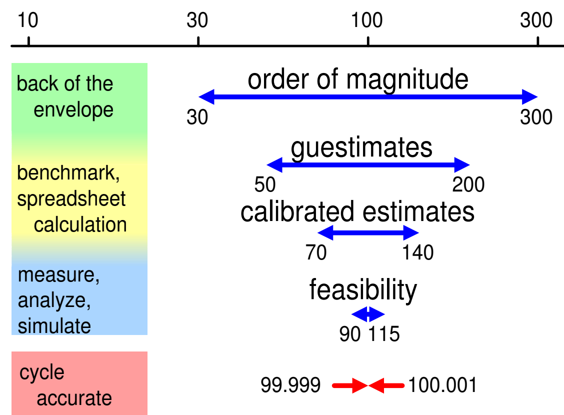


Figure 5: Successive quantification refined

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 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.

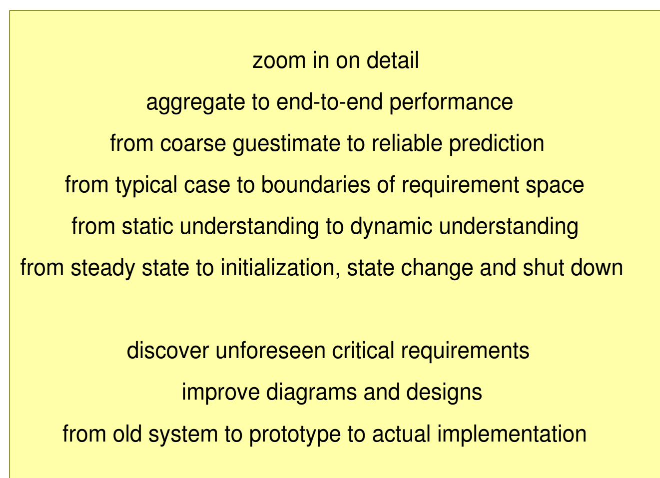


Figure 6: Directions of iterations

Figure 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.

3 Multiple views needed to understand system performance

The decomposition can be done along different axes. Subsection 3.1 shows *construction* as axis, and Subsection 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 3.3.

3.1 Construction Decomposition

The construction decomposition views the system from the construction point of view, see Figure 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.

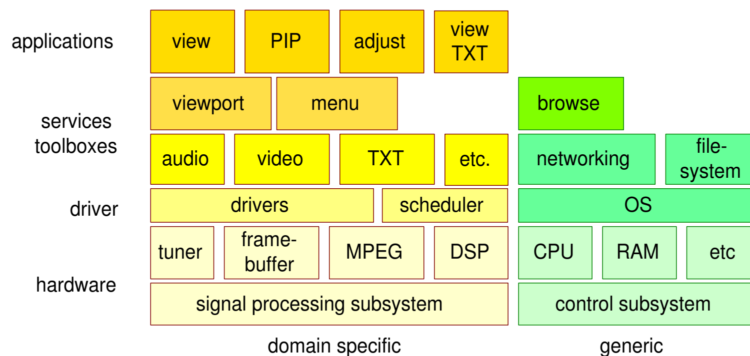


Figure 7: Example of a construction decomposition of a simple TV. The vertical axis is used for layers, where higher layers depend on lower layers, but not vice versa. In horizontal direction the left hand side shows the domain specific components, the right hand side shows the more generic components.

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 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) 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.

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.

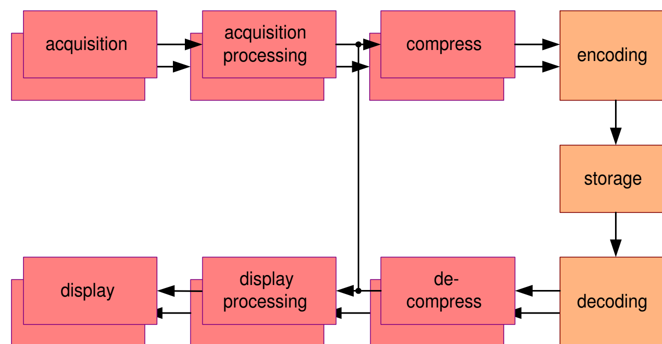


Figure 8: Example functional decomposition camera type device

Figure 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.

3.3 Execution Architecture

The execution architecture is the run-time architecture of a system. The process[1] decomposition plays an important role in the execution architecture. Figure 9 shows an example of a process decomposition.

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

[1]Process in terms of the operating system

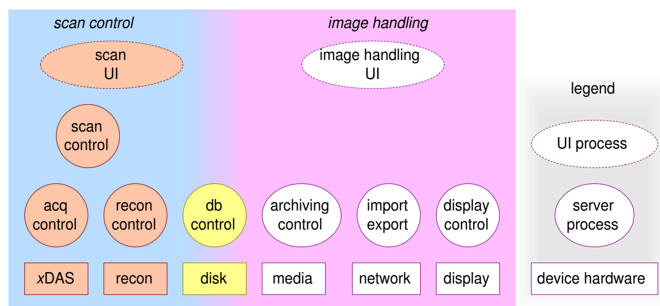


Figure 9: An example of a process decomposition of a MRI scanner.

processes. Threads share resources, which makes them more mutually dependent. In other words processes provide better means for separation of concerns.

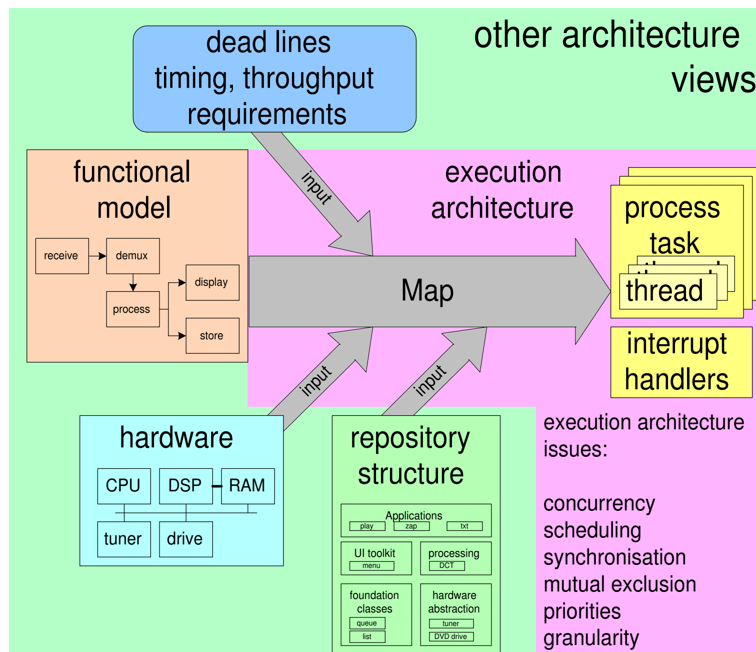


Figure 10: Execution Architecture

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 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 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.

4 Benchmarking

We propose to tackle the dynamic analysis by measuring and analyzing the system at several levels, as shown in Figure 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:

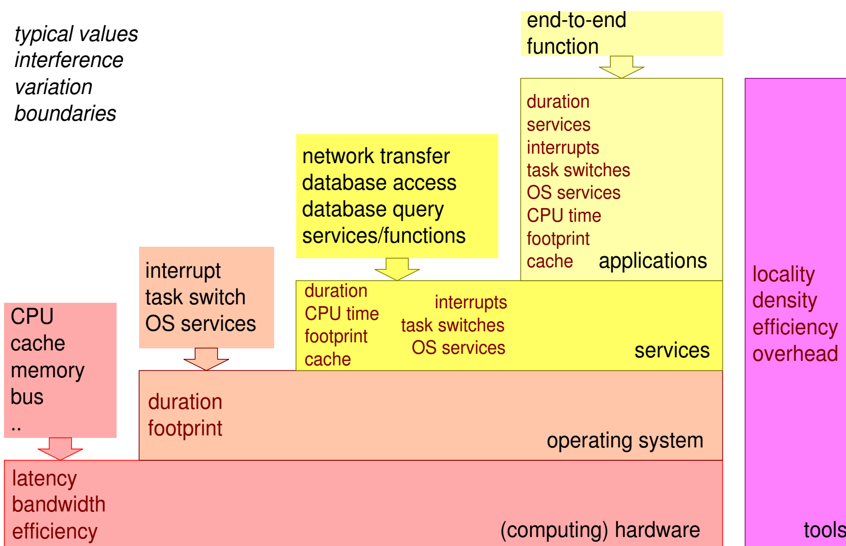


Figure 11: Layered Benchmarking

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 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 often[2].

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

Figure 12: Typical micro-benchmarks for timing aspects

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 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 Hennesey and Patterson [2]. Here modelling and measurement methods can be found that can serve as inspiration for performance analysis of embedded systems.

[2]This 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.

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.

References

[1] H Gomaa. *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/\discretionary{-}{}{}ArchitecturalReasoningBook.pdf`, 2002.

History

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

- added diagram of banchmark layers
- added process view and execution architecture
- cahnegd logo to EAAsprial
- changed status to draft
- created text

Version: 0.1, date: June 18, 2006 changed by: Gerrit Muller

- Relayout and reorder

Version: 0, date: January 10, 2006 changed by: Gerrit Muller

- Created, no changelog yet