

Module Design Side

logo
TBD

Gerrit Muller

Embedded Systems Institute

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

`gerrit.muller@embeddedsystems.nl`

Abstract

This module addresses the Conceptual and Realization Views.

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.

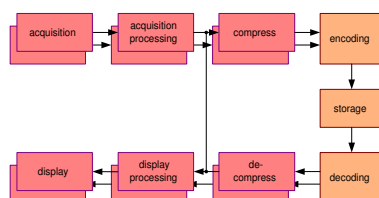
All Gaudí documents are available at:
<http://www.gaudisite.nl/>

Contents

1	The conceptual view	1
1.1	Introduction	1
1.2	Construction decomposition	2
1.3	Functional decomposition	4
1.4	Designing with multiple decompositions	5
1.5	Internal Information Model	8
1.6	Execution architecture	8
1.7	Performance	11
1.8	Safety, Reliability and Security concepts	11
1.9	Start up and shutdown	13
1.10	Work breakdown	13
1.11	Acknowledgements	17
2	The realization view	18
2.1	Budgets	18
2.2	Logarithmic views	20
2.3	Micro Benchmarking	22
2.4	Performance evaluation	23
2.5	Assessment of added value	24
2.6	Safety, Reliability and Security Analysis	28
2.7	Acknowledgements	28

Chapter 1

The conceptual view



1.1 Introduction

The conceptual view is used to understand how the product is achieving the specification. The methods and models used in the conceptual view should discuss the *how* of the product in conceptual terms. The lifetime of the concepts is longer than the specific implementation described in the *Realization* view. The conceptual view is more stable and reusable than the *realization* view.

The dominant principle in design is decomposition, often immediately coupled to interface management of the interfaces of the resulting components. It is important to realize that any system can be decomposed in many relevant ways. The most common ones are discussed here briefly: construction decomposition, section 1.2, functional decomposition, section 1.3, class or object decomposition, other decompositions (power, resources, recycling, maintenance, project management, cost, execution architecture...), and related models (performance, behavior, cost, ...).

If multiple decompositions are used then the relationships between decompositions are important. One of the methods to work with these relationships is via allocation. Within a decomposition and between decompositions the dependency structure is important.

From development management point of view it is useful to identify the infrastructure (factoring out shareable implementations), and to classify the technology in *core*, *key* and *base* technology.

The complement of decomposition is integration. Articulating the integrating concepts (start up, shutdown, safety, exception handling, persistency, resource management,...) provides guidance to the developers and helps to get a consistently behaving system.

1.2 Construction decomposition

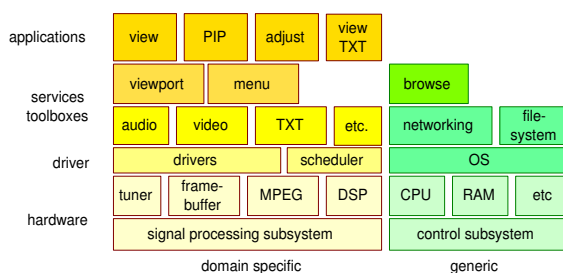


Figure 1.1: Example of a construction decomposition of a simple TV

The construction decomposition views the system from the construction point of view, see figure 1.1 for an example and figure 1.2 for the characterization of the construction decomposition.

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 in compound design units, which are used as unit for testing and release and this often coincides with organizational ownership and responsibility.

management of design	SW example	HW example
unit of creation storage update	file	PCB IP cells IP core
unit of aggregation for organisation test release	package module	box IP core IC

Figure 1.2: Characterization of the construction decomposition

In hardware this is quite often a very natural decomposition, for instance in cabinets, racks, boards and finally IC's, IP cores and cells. The components in the hardware components 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.

1.3 Functional decomposition

The functions as described in the functional view have to be performed by the design. These functions often are an aggregation of more elementary functions in the design. The functional decomposition decomposes end user functions in more elementary functions.

Be aware of the fact that the word *function* in system design is heavily overloaded. It does not help to define sharp boundaries with respect to the functional decomposition. Main criterium for a good functional decomposition is its useability for design. A functional decomposition provides insight how the system will accomplish its job.

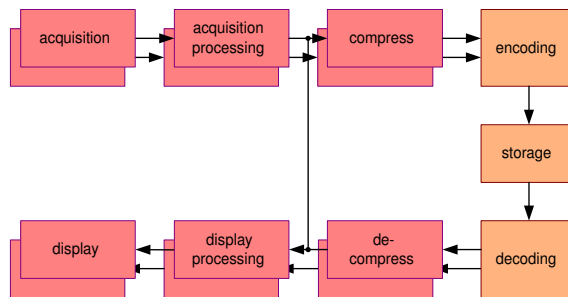


Figure 1.3: Example functional decomposition camera type device

Figure 1.3 shows an example of (part of) a functional decomposition for a camera type device. It shows 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, but from control point of view.

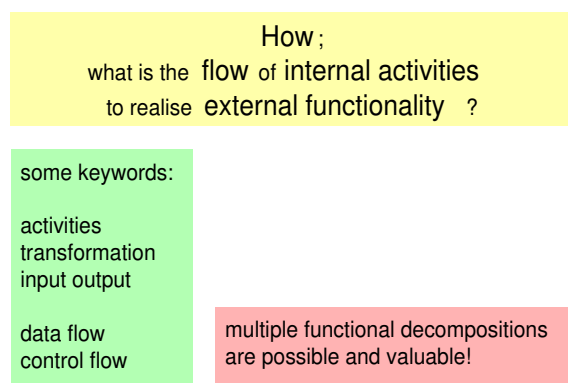


Figure 1.4: Characterization of the functional decomposition

1.4 Designing with multiple decompositions

The design of complex systems always requires multiple decompositions, for instance a construction and a functional decomposition. Many designers in the design team need support to cope with this multiplicity.

Most designers don't anticipate cross system design issues, for instance when asked in preparation of design team meetings. This limited anticipation is caused by the locality of the viewpoint, implicitly chosen by the designers.

How about the **<characteristic>**
of the **<component>**
when performing **<function>**?

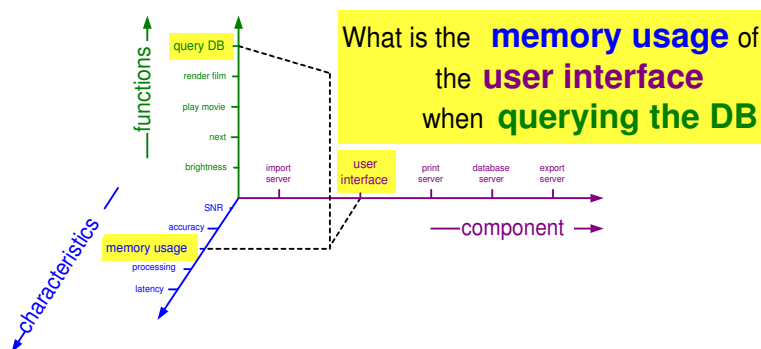


Figure 1.5: Question generator for multiple decompositions

Figure 1.5 shows a method to help designers to find system design issues. A three dimensional space is shown. Two dimensions are the decomposition dimension (component and functional), the last dimension is the design characteristic dimension.

For every point in this 3D space a question can be generated in the following way:

How about the *<characteristic>* of the *<component>* when performing *<function>*?

Which will result in questions like:

How about the *memory usage* of the *user interface* when *querying the database*?

The designers will not be able to answer most of these questions. Simply asking these questions helps the designer to change the viewpoint and discover many potential issues. Luckily most of the not answered questions will not be relevant. The answer to the memory usage question above might be *insignificant* or *small*.

The architect has to apply a priori know how to select the most relevant questions in the 3D space. Figure 1.6 shows a set of selection factors that can be used to determine the most relevant questions.

Critical for system performance

Risk planning wise

Least robust part of the design

Suspect part of the design

- experience based

- person based

Figure 1.6: Selection factors to improve the question generator

Critical for system performance Every question that is directly related to critical aspects of the system performance is relevant. For example *What is the CPU load of the motion compensation function in the streaming subsystem?* will be relevant for resource constrained systems.

Risk planning wise Questions regarding critical planning issues are also relevant. For example *Will all concurrent streaming operations fit within the designed resources?* will greatly influence the planning if resources have to be added.

Least robust part of the design Some parts of the design are known to be rather sensitive, for instance the priority settings of threads. Satisfactory answers should be available, where a satisfactory answer might also be *we scheduled a priority tuning phase, with the following approach.*

Suspect part of the design Other parts of the design might be suspect for several reasons. For instance experience learns that response times and throughput do not get the required attention of software designers (experience based suspicion). Or for instance we allocated an engineer to the job with insufficient competence (person based suspicion).

Figure 1.7 shows another potential optimization, to address a line or a plane in the multi dimensional space. The figure shows an example of a memory budget for the system, which is addressing all memory aspects for both functions and components in one budget. The other example is the design specification of a database query, where the design addresses the allocation to components as well as all relevant design characteristics.

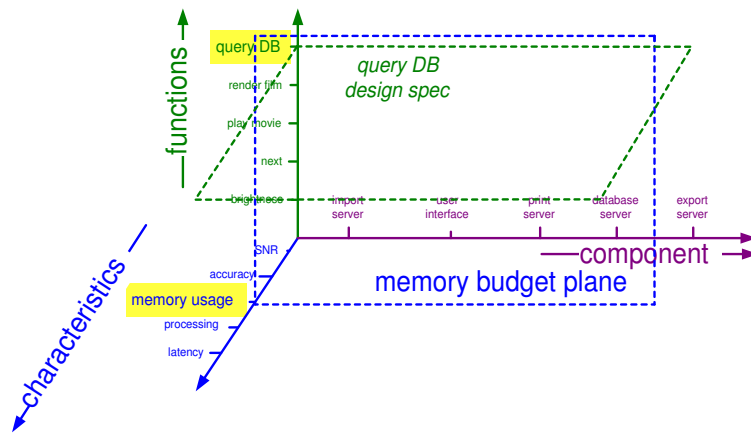


Figure 1.7: Addressing lines or planes at once in the multiple dimensions

1.5 Internal Information Model

The information model as seen from the outside from the system, part of the functional view, is extended into an internal information model. The internal information model is extended with design choices, for instance derived data information is cached to achieve the desired performance. The internal data model might also be chosen to be more generic (for reasons of future extendibility), or less generic (where program code is used to translate the specific internal models in the desired external models).

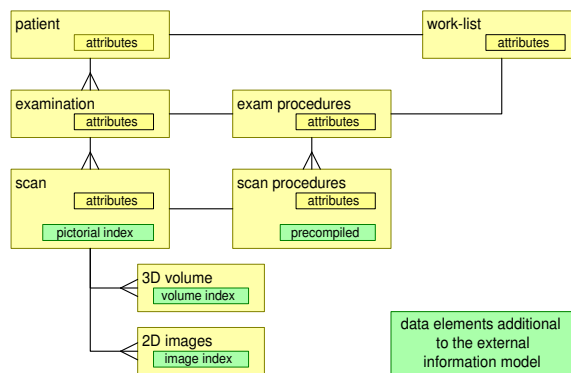


Figure 1.8: Example of a partial internal information model

The internal information model is an important means to decouple parts of the design. The functional behavior of the system is predictable as long as components in the system adhere to the internal information model.

Figure 1.8 shows an example of a part of an information model. In this example several information elements which are derived from the primary data are stored explicitly to improve the response time. The pictorial index, existing of reduced size images, is an example of derived information, which takes some time to calculate. This index is build in the background during import, so that the navigation can use it, which makes the navigation very responsive.

All considerations described in section ??, such as the layering hold also for the internal information model.

1.6 Execution architecture

The execution architecture is the run time architecture of a system. The process decomposition plays an important role in the execution architecture. Figure 1.9 shows an example of a process decomposition.

One of the main concerns for process decomposition is concurrency: which concurrent activities are needed or running, how to synchronize these activities. A

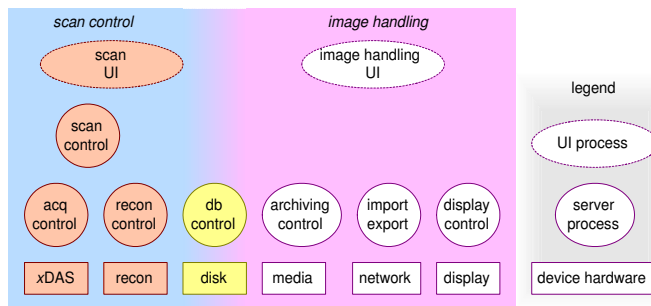


Figure 1.9: Example process decomposition

process or a task of an operating system is a concept which supports asynchronous functionality as well as separation of concerns by providing process specific resources, such as memory. A thread is a lighter construction providing support for asynchronous activities, without the separation of concerns.

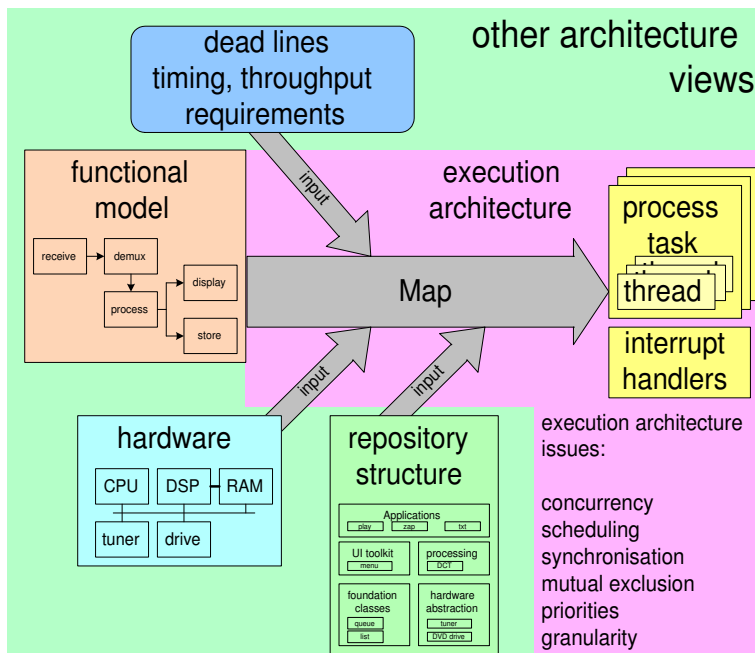


Figure 1.10: Execution architecture

The execution architecture must map the functional decomposition on the process decomposition, taking into account the construction decomposition. In practice many building blocks from the construction decomposition are used in multiple functions mapped on multiple processes. These shared building blocks are aggre-

gated in shared (or dynamic link) libraries. Sharing is advantageous from memory consumption point of view, some attention is required for the configuration management side¹.

Figure 1.10 shows the role of the execution architecture. The main inputs are the real time and performance requirements at the one hand and the hardware design at the other hand. The functions need to be mapped on processes, threads and interrupt handlers, synchronization method and granularity need to be defined and the scheduling behavior (for instance priority based, which requires priorities to be defined).

¹The *dll-hell* is not an windows-only problem. Multiple pieces of software sharing the same library can easily lead to version problems, module 1 requires version 1.13, while module 2 requires version 2.11. Despite all compatibility claims it often does not work.

1.7 Performance

The performance of a system can be modeled by complementing models. In figure 1.11 the performance is modelled by a flow model at the top and an analytical model below. The analytical model is entirely parameterized, making it a generic model which describes the performance ratio over the full potential range.

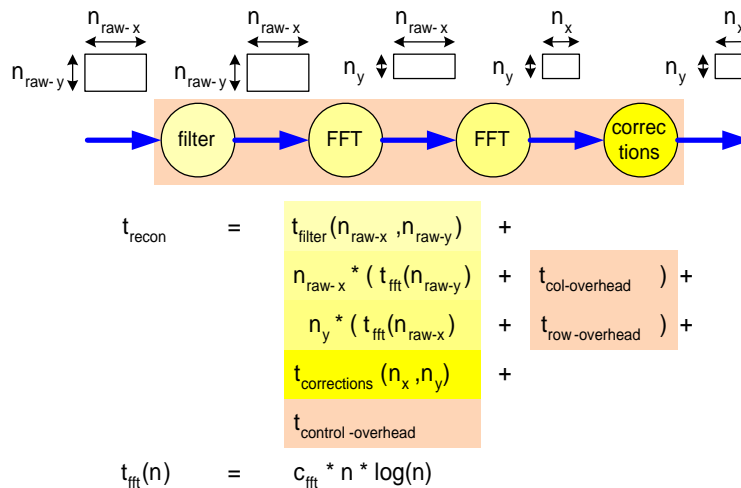


Figure 1.11: Performance Model

Later in the realization view it will be shown that this model is too simplistic, because it focuses too much on the processing and does not take the overheads sufficiently in account.

1.8 Safety, Reliability and Security concepts

The qualities *safety*, *reliability* and *security* share a number of concepts, such as:

- containment (limit failure consequences to well defined scope)
- graceful degradation (system parts not affected by failure continue operation)
- dead man switch (human activity required for operation)
- interlock (operation only if hardware conditions are fulfilled)
- detection and tracing of failures
- black box (log) for post mortem analysis
- redundancy

A common guideline in applying any of these concepts is that the more critical a function is, the higher the understandability should be, or in other words the simpler the applied concepts should be. Many elementary safety functions are implemented in hardware, avoiding large stacks of complex software.

1.9 Start up and shutdown

In practice insufficient attention is paid to the start up and shutdown of a system, since these are relatively exceptional operations. However the design of this aspect has an impact on nearly all components and functions in the system. It is really an integrating concept. The trend is that these operations become even more entangled with the normal run-time functionality, for instance by run-time downloading, stand-by and other power saving functionality.

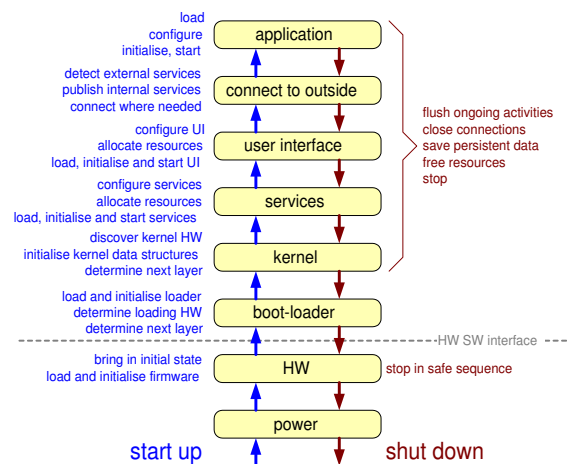


Figure 1.12: Simplified start up sequence

Figure 1.12 shows a typical start up shutdown pattern. The system is brought step by step to higher operational levels. Higher levels benefit from more available support functions, lower levels are less dependent on support functions.

One of the considerations in the design of this system aspect is the impact of failures. The right granularity of operational levels enable coping with exceptions (for example network not available). For shutdown the main question is how power failures or glitches are handled.

1.10 Work breakdown

Project leaders expect a work breakdown to be made by the architect. In fact a work breakdown is again another decomposition, with a more organizational point of view. The work in the different work packages should be cohesive internally, and should have low coupling with other work-packages.

Figure 1.13 shows an example of a work breakdown. The entire project is broken down in a hierarchical fashion: project, segment, work-package. In this example color coding is applied to show the technology involved and to show

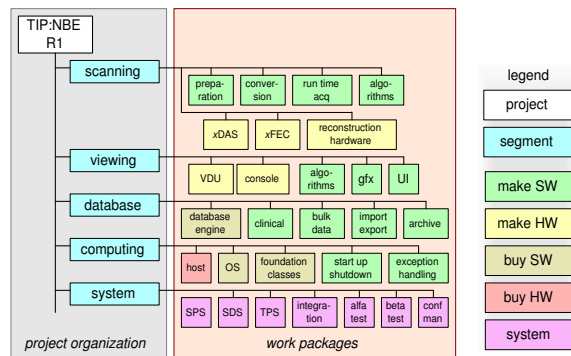


Figure 1.13: Example work breakdown

development work or purchasing work. Both types of work require domain know how, but different skills to do the job.

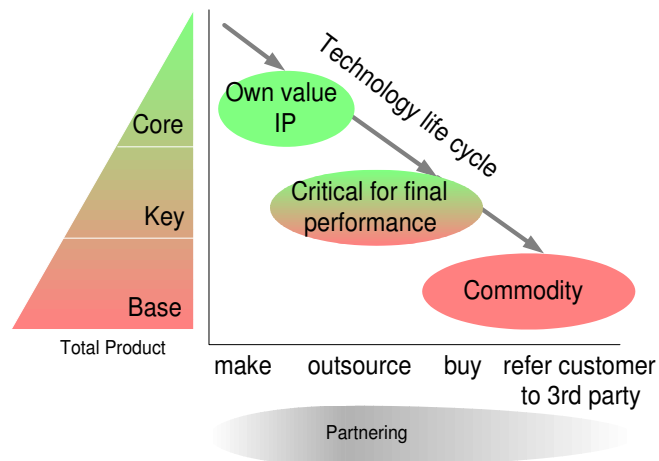


Figure 1.14: Core, Key or Base technology

Make versus Buy is a limited subset of an entire spectrum of approaches. The decision how to obtain the needed technology should be based on where the company intends to add value. A simple reference model to help in making these decisions is based on *core*, *key*, and *base* technology, see figure 1.14.

Core technology is technology where the company is adding value. In order to be able to add value, this technology should be developed by the company itself.

Key technology is technology which is critical for the final system performance. If the system performance can not be reached by means of third party technology

than the company must develop it themselves. Otherwise outsourcing or buying is attractive, in order to focus as much as possible on *core* technology added value. However when outsourcing or buying an intimate partnership is recommended to ensure the proper performance level.

Base technology is technology which is available on the market and where the development is driven by other systems or applications. Care should be taken that these external developments can be followed. Own developments here are de-focusing the attention from the company's core technology.

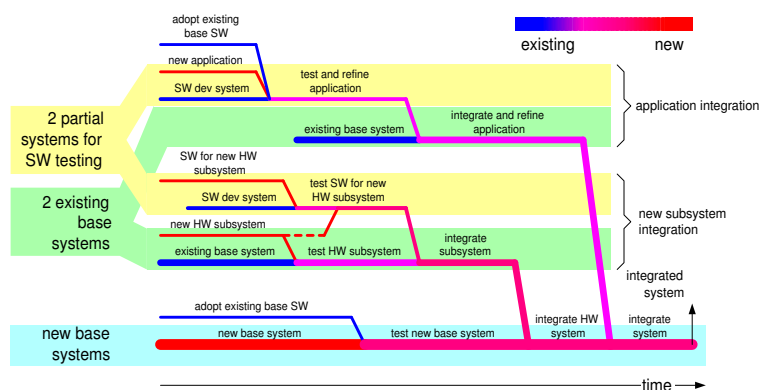


Figure 1.15: Example integration plan, with 3 tiers of development models

Schedules, work breakdown and many technical decompositions are heavily influenced by the integration plan. Integration is time, effort and risk determining part of the entire product creation process. The integration viewpoint must be used regular because of its time, effort and risk impact.

Figure 1.15 shows an example integration plan. This plan is centered around 3 tiers of development vehicles:

- SW development systems
- existing HW system
- new HW system

The SW development system, existing from standard clients and servers, is very flexible and accessible from software point of view, but far from realistic from hardware point of view. The existing and new HW systems are much less accessible and more rigid, but close to the final product reality. The new HW system will be available late and hides many risks and uncertainties. The overall strategy is to move for software development from an accessible system to a stable HW system to the more real final system. In general integration plans try to avoid stacking

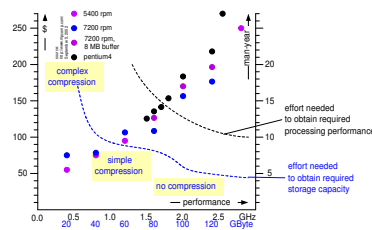
too many uncertainties by looking for ways to test new modules in a stable known environment, before confronting new modules with each other.

1.11 Acknowledgements

Constructive remarks from Peter Bingley, Peter van den Hamer, Ton Kostelijk, William van der Sterren and Berry van der Wijst have been integrated in this document.

Chapter 2

The realization view



2.1 Budgets

The implementation can be guided by making budgets for the most important resource constraints, such as memory size, response time, or positioning accuracy. The budget serves multiple purposes:

- 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 explicit

Figure 2.1 shows a budget based design flow. The starting point of a budget is a model of the system, from the conceptual view. 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

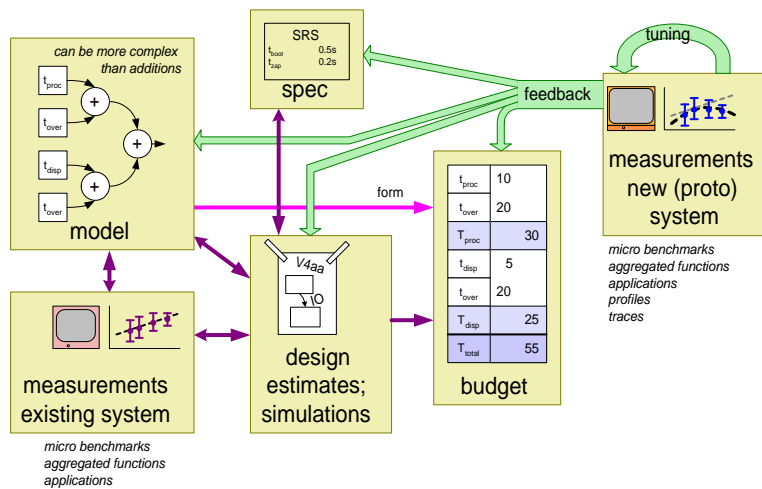


Figure 2.1: Budget based design flow

new system, sufficient improvements must be designed to achieve the required improvement.

Early measurements in the 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.

<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 2.2: Example of a memory budget

Figure 2.2 shows an example of an actual memory budget. This budget decomposes the memory in 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 fragmentation). The difference in behavior is an important reason to separate in different budget entries. At the other hand the operating system and the system infrastructure provide means to measure these 3 types at any moment, which helps for the initial definition, for the integration and the verification.

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

2.2 Logarithmic views

A logarithmic positioning of requirements and implementation alternatives helps to put these alternatives in perspective. In most designs we have to make design choices which cover a very large dynamic range, for instance from nanoseconds up to hours, days or even years. Figure 2.3 shows an example of requirements and technologies on a logarithmic time axis.

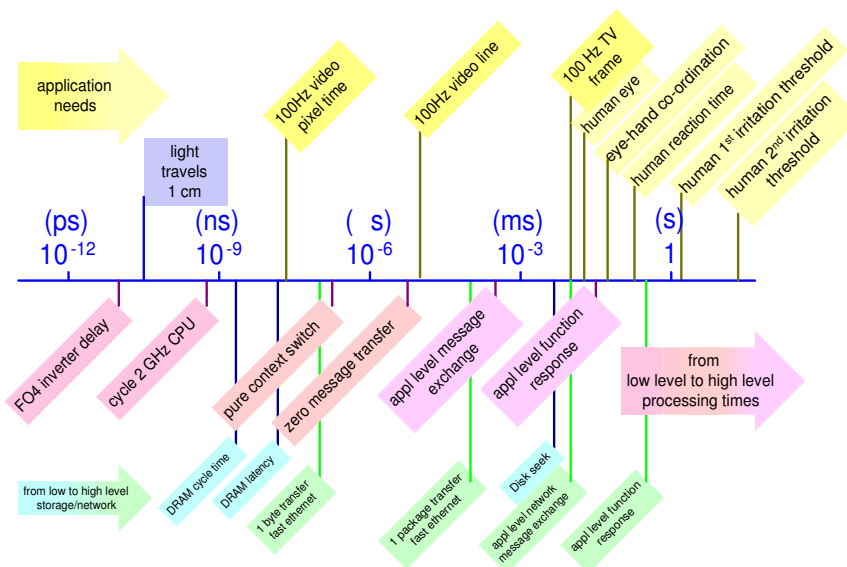


Figure 2.3: Actual timing represented on a logarithmic scale

”Fast” technologies can serve many slow requirements, but often slower technologies offer other benefits, which offset their slowness. ”Slow” technologies offer more flexibility and power, at the cost of performance. For instance real time executive interrupt response time are very short, while reacting in a user task is slower, but can access much more user level data and can interact more easy with other application level functions. Going from real time executive to a ”fat” operating

system slows down the interrupt response, with a wealth of other operating system functionality (networking, storage, et cetera) in return. Again at user process level the response needed is again bigger, with a large amount of application level functionality in return (distribution, data management, UI management, et cetera).

Requirements itself also span such a large dynamic range from very fast (video processing standards determining pixel rates) to much slower (select teletext page).

For every requirement a reasonable implementation choice is needed with respect to the speed. Faster is not always better, a balance between fast enough, cost and flexibility and power is needed.

2.3 Micro Benchmarking

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 2.4 lists a typical set of micro-benchmarks to be performed. The list shows infrequent and often slow operations and frequently applied operations, which are often much faster. This classification implies already a design rule: slow operations should not be performed often¹.

	<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 2.4: 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 performance measured in a micro benchmark is often dependent on a number of parameters, such as the length of a transfer. Micro benchmarks are applied with a variation of these parameters, to obtain understanding of the performance as a function of these parameters. Figure 2.5 shows an example of the transfer rate performance as a function of the block size.

For example measuring disk transfer rates will result in this kind of curves, due

¹This 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 if a slow operation is in fact not very slow and the brute force approach is both affordable as well as extremely straightforward (simple!) then this is better than over-optimizing for efficiency.

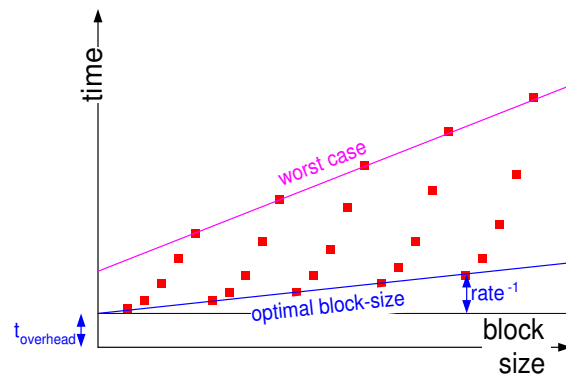


Figure 2.5: The transfer time as function of block size

to a combination of cycle time, seek time and peek transfer rate. This data can be used in different ways: the slowest speed can be used, a worst case design, or the buffer size can be tuned to obtain the maximum transfer rate. Both choices are defensible, the conservative choice is costly, but robust, the optimized choice is more competitive, but also more vulnerable.

2.4 Performance evaluation

The performance is conceptually modelled in the conceptual view, which is used to make budgets in the realization view. An essential question for the architect is: Is this design *good*? This question can only be answered if the criteria are known for a *good* design. Obvious criteria are meeting the need and fitting the constraints. However an architect will add some criteria himself, such as balanced and future-proof.

Figure 2.6 shows an example of a performance analysis. The model is shown at the top of the figure, as discussed in the conceptual view. The measurement below the model shows that a number of significant costs have not been included in the original model, although these are added in the model here. The original model focuses on processing cost, including some processing related overhead. However in practice overhead plays a dominant role in the total system performance. Significant overhead costs are often present in initialization, I/O, synchronization, transfers, allocation and garbage collection (or freeing if explicitly managed).

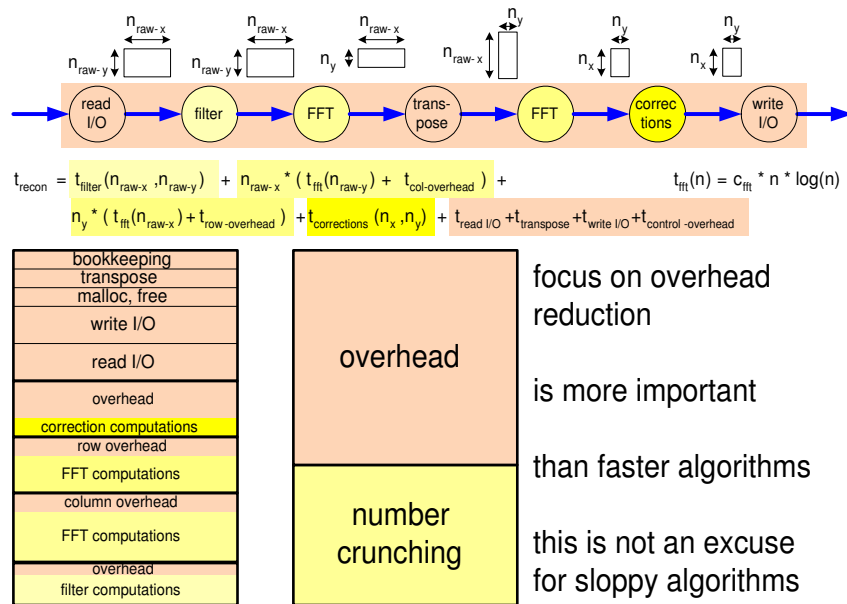


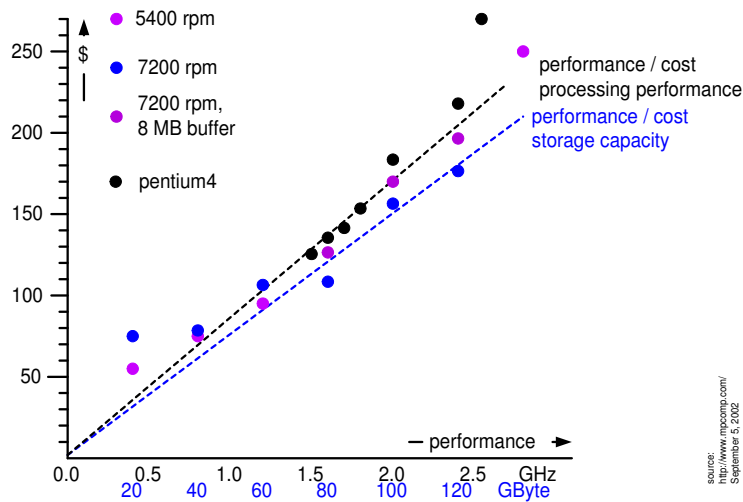
Figure 2.6: Example of performance analysis and evaluation

2.5 Assessment of added value

The implementation should be monitored with respect to its quality. The most common monitoring is problem reporting and fault analysis. The architect should maintain a quality assessment, based on the implementation itself. This is done by monitoring size and change frequency. In order to do something useful with these metrics some kind of value indicator is also needed. The architect must build up a reference of "value per size" metrics, which he can use for this a priori quality monitoring.

Figure 2.7 shows an example of a performance cost curve, in this example Pentium4 processors and hard disks. Performance and cost are roughly proportional. For higher performance the price rises faster than the performance, At the low performance side the products level out at a kind of bottom price, or that segment is not at all populated (minimum Pentium4 performance is 1.5 GHz, the lower segment is populated with Celerons, which again don't go down to any frequency).

The choice of a solution will be based on the needs of the customer. To get grip on these needs the performance need can be translated in the sales value. How much is the customer willing to pay for performance? In this example the customer is not willing to pay for a system with insufficient performance, neither is the customer willing to pay much for additional performance (if the system does



source: <http://www.embedded.com/>
September 8, 2002

Figure 2.7: Performance Cost, input data

the job, then it is OK). This is shown in figure 2.8, with rather non-linear sales value curves.

Another point of view is the development effort. Over-dimensioning of processing or storage capacity simplifies many design decisions resulting in less development effort. In figure 2.9 this is shown by the effort as function of the performance.

For example for the storage capacity three effort levels can be distinguished: with a low cost (small capacity) disk a lot of tricks are required to fit the application within the storage constraint, for instancing by applying complex compression techniques. The next level is for medium cost disks, which can be used with simple compression techniques, while the expensive disks don't need compression at all.

Figure 2.10 show that many more issues determine the final choice for the "right" cost/performance choice: the capabilities of the rest of the system, the constraints and opportunities in the system context, trade-offs with the image quality. All of the considerations are changing over time, today we might need complex compression, next year this might be a no-brainer. The issue of effort turns out to be related with the risk of the development (large developments are more risky) and to time to market (large efforts often require more time).

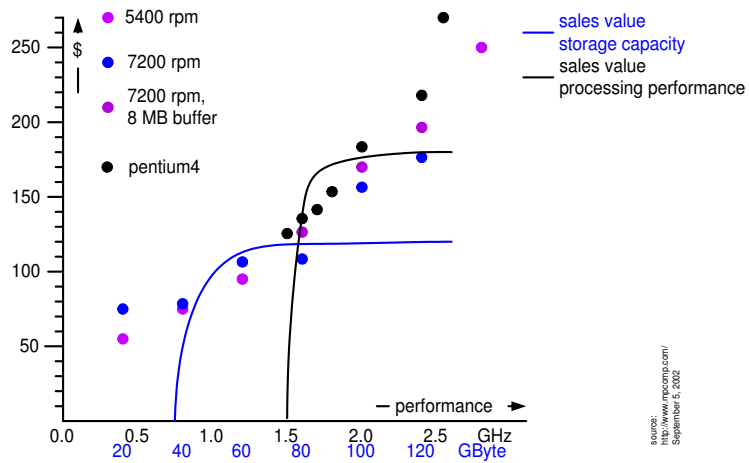


Figure 2.8: Performance Cost, choice based on sales value

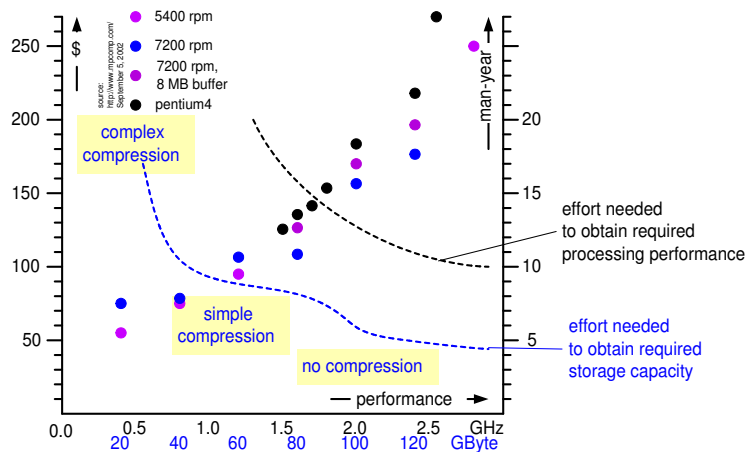


Figure 2.9: Performance Cost, effort consequences

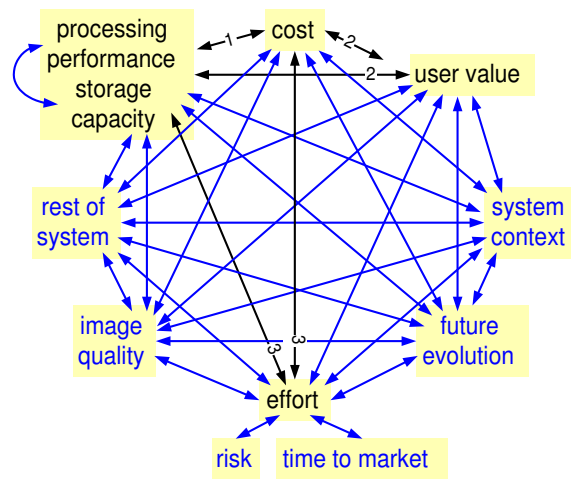


Figure 2.10: But many many other considerations

2.6 Safety, Reliability and Security Analysis

Qualities such as safety, reliability and security depend strongly on the actual implementation. Specialized engineering disciplines exist for these areas. These disciplines have developed their own methods. One class of methods relevant for system architects is the class of analysis methods, which start with a (systematic) brainstorm, see figure 2.11.

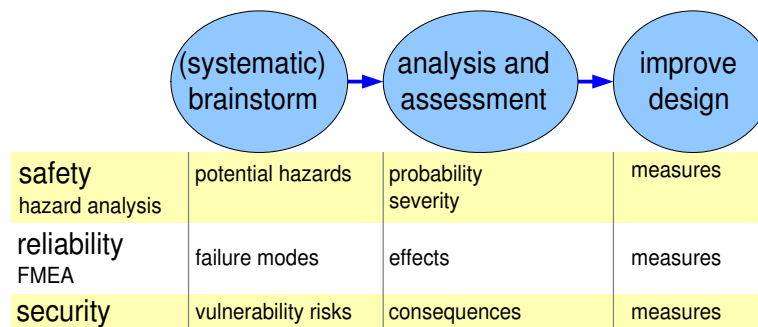


Figure 2.11: Analysis methods for safety, reliability and security

Walk-through is another effective assessment method. A few use cases are taken and together with the engineers the implementation behavior is followed for these cases. The architect will especially assess the understandability and simplicity of the implementation. An implementation which is difficult to follow with respect to safety, security or reliability is suspect and at least requires more analysis.

2.7 Acknowledgements

William van der Sterren and Peter van den Hamer invented the nice phrase micro benchmarking.

Bibliography

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

History

Version: 0, date: July 2, 2004 changed by: Gerrit Muller

- created module