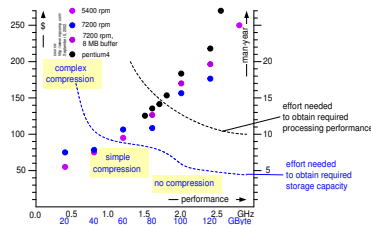


The realization view

-



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 realization view looks at the actual technologies used and the actual implementation. Methods used here are logarithmic views, micro-benchmarks and budgets. Analysis methods with respect to safety, reliability and security provide a link back to the functional and conceptual 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/>

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

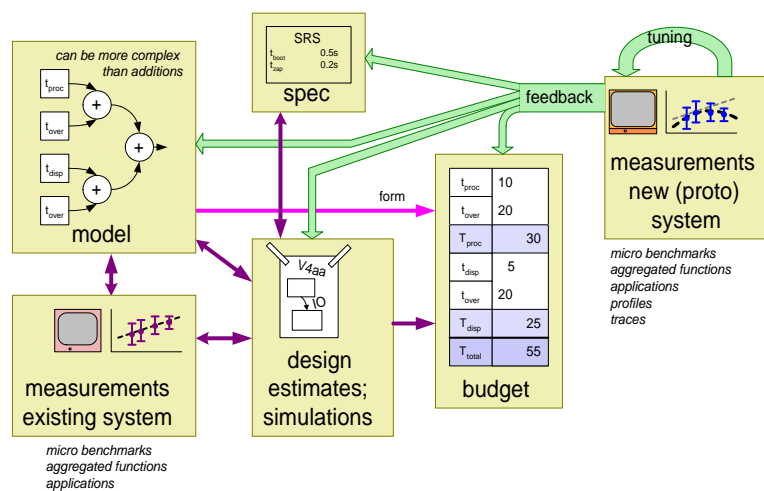


Figure 1: Budget based design flow

Figure 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 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: Example of a memory budget

Figure 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 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 3 shows an example of requirements and technologies on a logarithmic time axis.

”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

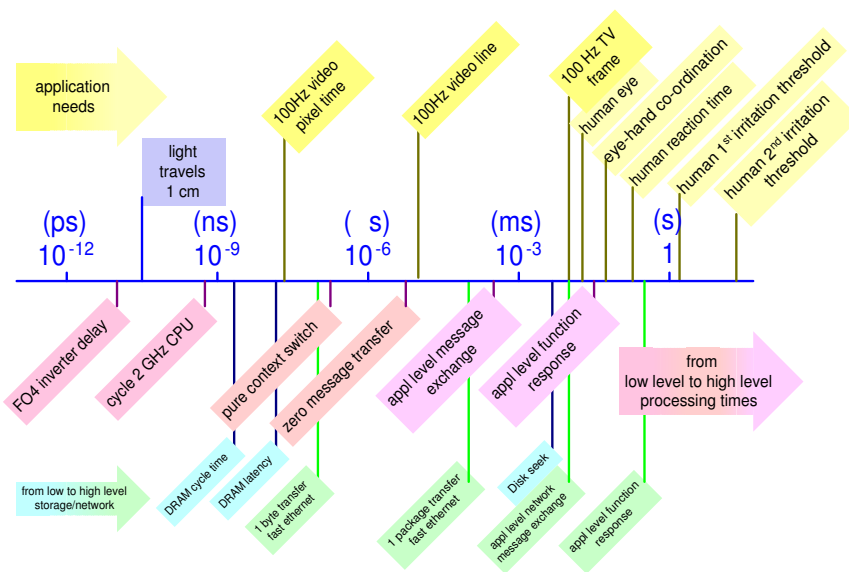


Figure 3: Actual timing represented on a logarithmic scale

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.

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 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 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 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.

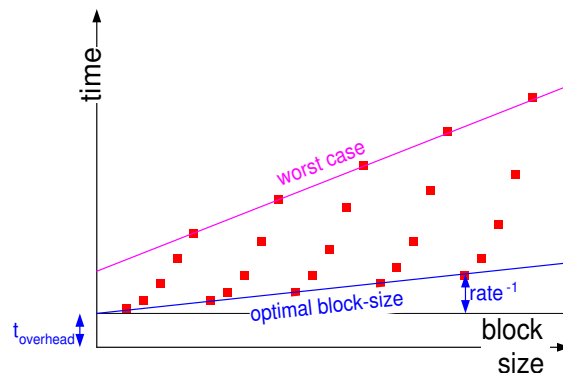


Figure 5: The transfer time as function of block size

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.

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 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).

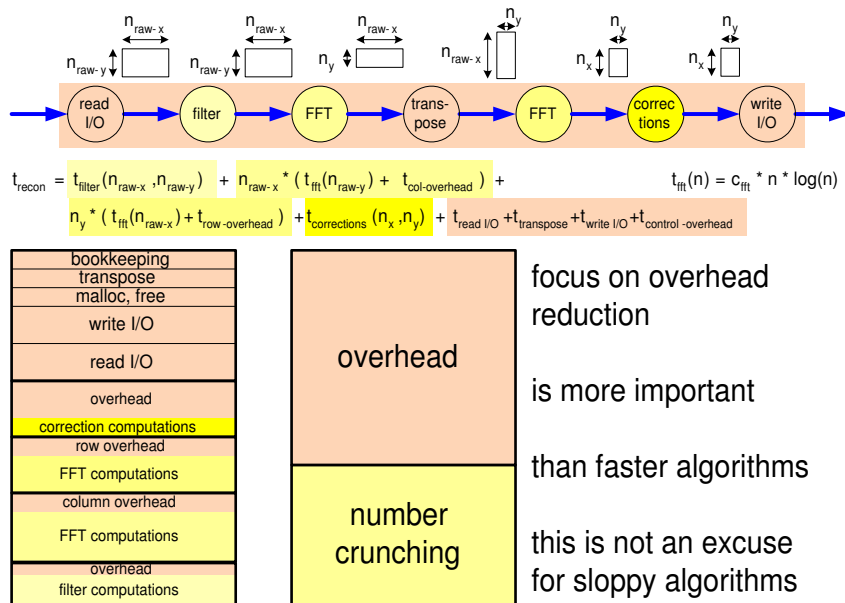


Figure 6: Example of performance analysis and evaluation

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 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 the job, then it is OK). This is shown in figure 8, with rather non-linear sales value

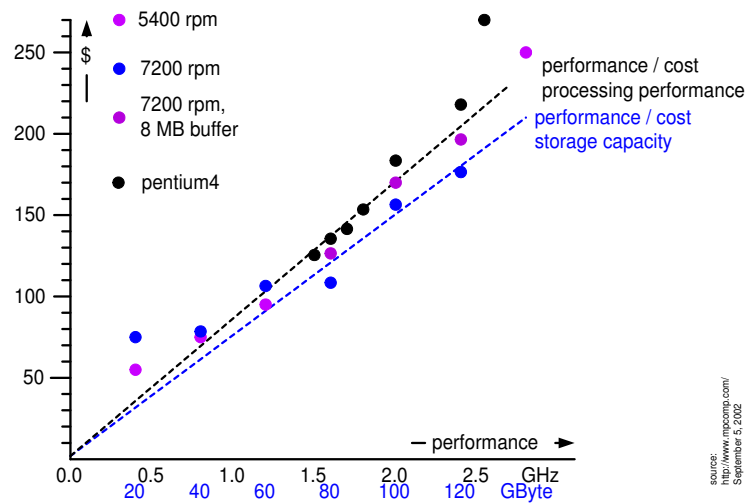


Figure 7: Performance Cost, input data

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 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 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).

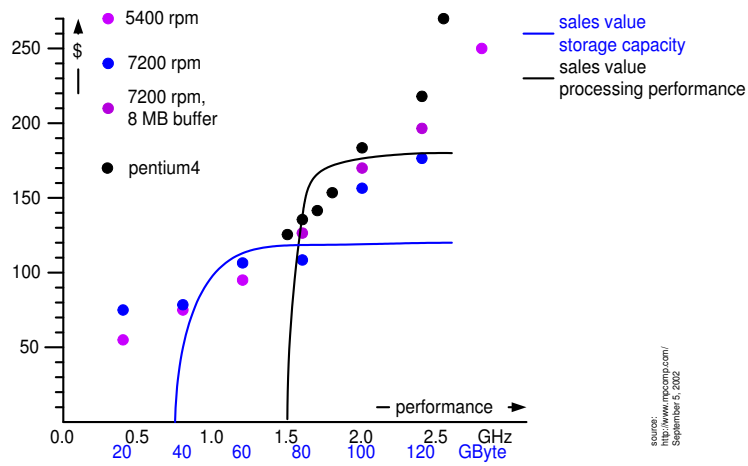


Figure 8: Performance Cost, choice based on sales value

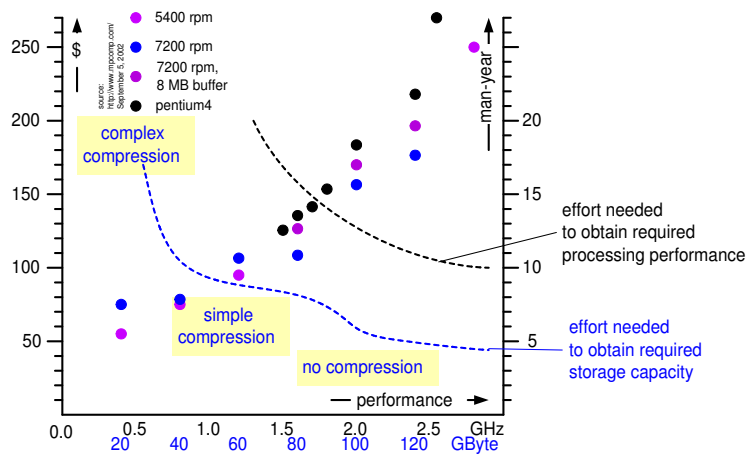


Figure 9: Performance Cost, effort consequences

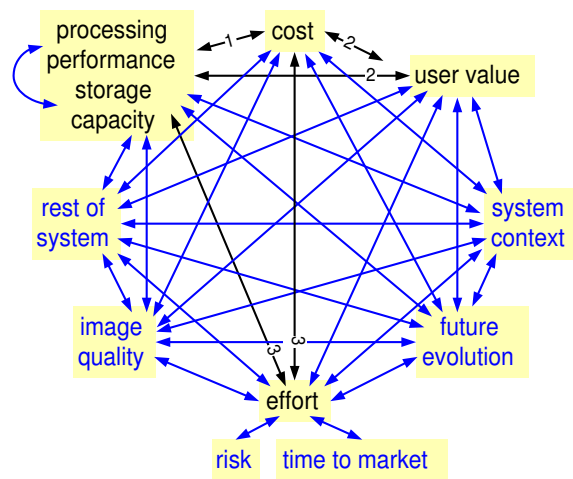


Figure 10: But many many other considerations

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 11.

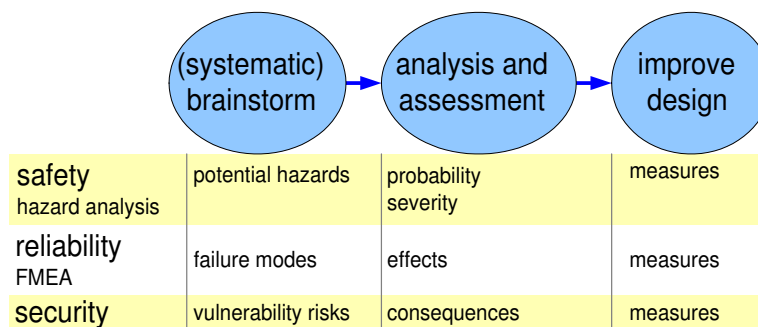


Figure 11: Analysis methods for safety, reliability and security

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.

7 Acknowledgements

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

References

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

History

Version: 0.2, date: September 3 2002 changed by: Gerrit Muller

- updated figure Time axis
- added budget based design flow
- added cost performance figures
- added a lot of text

Version: 0.1, date: July 9 2002 changed by: Gerrit Muller

- updated figure Time axis

Version: 0, date: June 21 2002 changed by: Gerrit Muller

- Created, no changelog yet