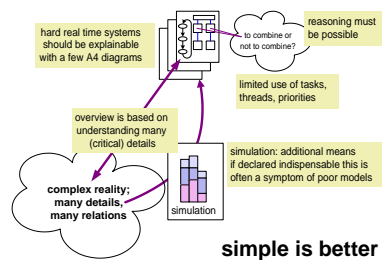


# Execution architecture concepts



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

The execution architecture determines largely the realtime and performance behavior of a system. Hard real time is characterized as "missing a deadline" will result in system failure, while soft real time will result "only" in dissatisfaction. An incremental design approach is described. Concepts such as latency, response time and throughput are illustrated. Design considerations and recommendations are given such as separation of concerns, understandability and granularity. The use of budgets for design and feedback is discussed.

### 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: 1.1

status: preliminary draft

February 10, 2011

# 1 Introduction

The execution architecture is only a small part of the total system architecture. The zero order approach is that execution architecture is the mapping of functionality via software building blocks on hardware resources by means of processes or tasks, priorities and synchronized by means of interrupts, see figure 1. Most effort is spend in creating (defining, building, testing) these building blocks, while a limited amount of time should be spend on the run time structure.

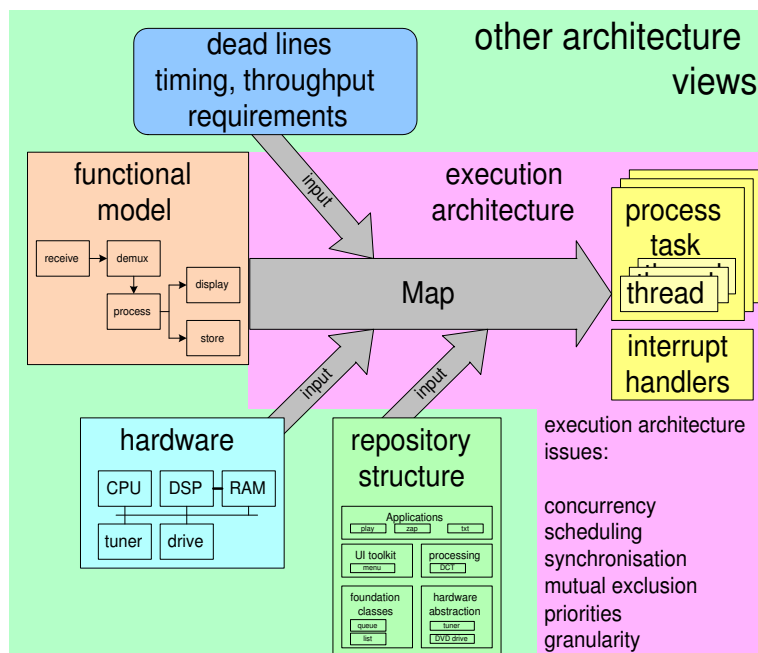


Figure 1: Execution Architecture

The more qualified view shows in first order approach that execution architecture assumptions and know how are used in hardware, software and functional design. The best approach is an highly incremental and iterative approach.

In real life the iteration is limited, amongst others due to different development lifecycles 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.

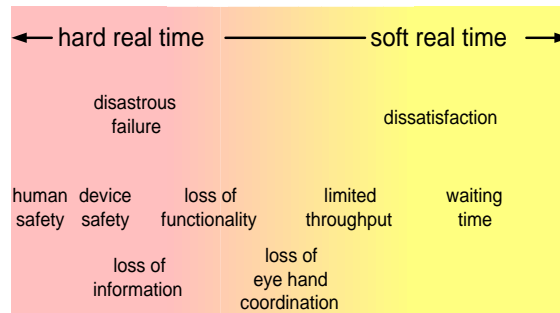


Figure 2: Fuzzy customer view on real time

One of the starting questions is what are the most critical timing issues. Sufficient understanding of the domain is required to answer this question. The reasoning that the fastest response is the most critical is way too simplistic, this approach takes only (part of) the solution domain into account. Both the problem domain as well as the solution domain must be taken into account.

Figure 2 shows one of the dimensions of criticality: what is the importance of meeting the requirement, or what is the consequence of not meeting the requirement. *Hard* real time requirements are requirements which must be met, because the consequence of not meeting it are very severe. For instance human safety is very important and may not be endangered. Note that the time scale depends entirely on the domain, many operations in an airplane are real time, but in the range of seconds.

The figure shows that the notion of hard real time is not as hard as it seems. Device safety and loss of functionality are also very important. However some functionality loss is less severe than other functionality loss. For instance missing one video frame while watching TV is not disastrous, losing one frame while saving video on DVD+RW is already much more severe.

Human system interaction time is again much less severe. But here also constraints exist which might result in severe timing requirements. For instance the eye-hand coordination when using a mouse is rather sensitive for the response time. Typical the response time must be good, but occasional hickups are gracefully handled by our brains.

The main recommendation is to strive for a limited set of well defined timing requirements, where the understanding is shared between the main stakeholders.

## 2 Concepts

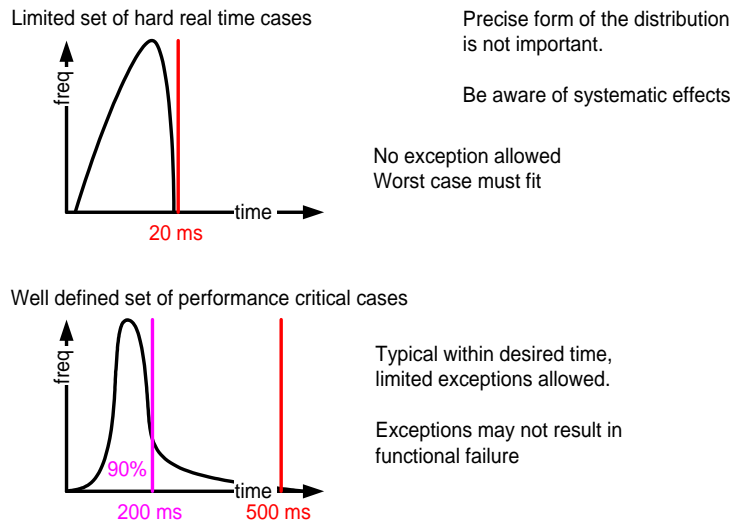


Figure 3: Smartening requirements

One of the aspects of making requirements sufficiently specific and measurable is the meaning of the number. Hard real time means the requirement must be met, no exception allowed, because missing the requirement once will have disastrous results.

In most systems the timing is a complex function of state, context, parameters et cetera, which results in a distribution of actual timings. See figure 3 which shows two types of distributions: the hard real time case needs a guaranteed finite distribution, while the soft real time case defines a typical response (90% of the events are handled within 200 ms), while all responses will be handled within 500 ms.

This second (soft real time) requirement needs more clarification for worst case incidents<sup>1</sup>. Typical the exceptional behavior will be defined (if the frame processing is not finished in time, the previous frame will be displayed twice).

The specification should be kept simple, don't try to describe the entire form of the distribution, only describe the user relevant characteristics. From design point of view systematic effects (for instance an exception, which triggers an avalanche of follow up activities) threaten the nice well defined distributions.

<sup>1</sup>Operating systems such as VMS, and windows NT, show incidental non reactive periods of many seconds. Even the real time modes of these systems can hit incidental slow responses. VMS used to be extremely unresponsive when a new process is activated. This does not happen often, but if it can happen, it will happen. What do we reflect in our specification?

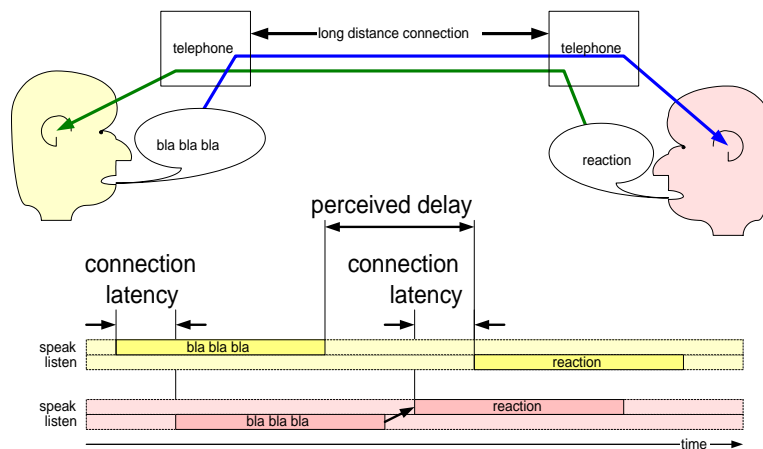


Figure 4: Latency

For a set of requirements the latency (the more or less constant shift in time) is critical. Especially audio latency is quite sensitive. Figure 4 shows an example of the audio latencies involved in a telephone connection. The latency determines the usability. Above a certain latency humans have to adopt their way of working, for instance by adding their own protocol layer (*over, close*).

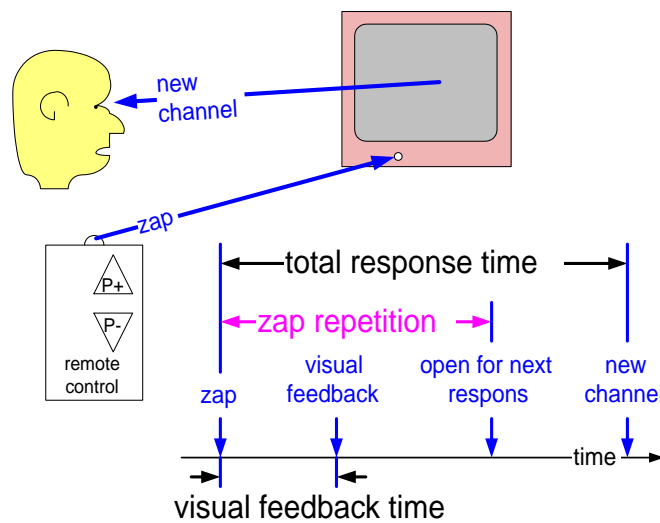
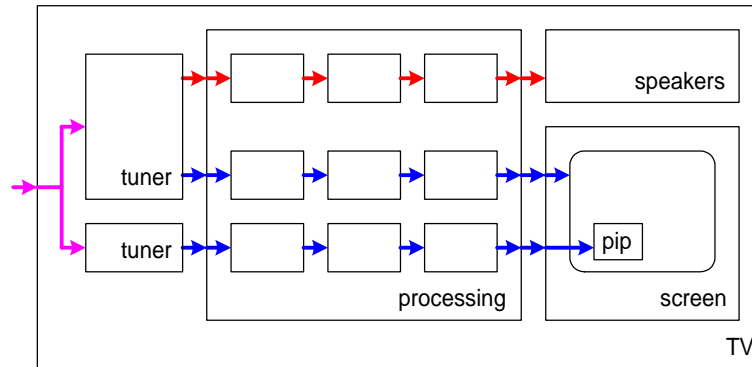


Figure 5: Response Time

A related set of requirements defines *response time*. Figure 5 shows an example of zapping response time. This example shows that a response time can involve multiple timing requirements. In this example the television provides early visual

feedback, which satisfies most response requirements from the user. However for zapping through it is also relevant to be able to press again, while the TV "sees" this additional zap. And of course the real TV channel should become visible within a reasonable amount of time.



throughput:  
 + processing steps/frame  
 + frames/second  
 + concurrent streams

Figure 6: Throughput

The amount of data processed and transferred in the system, *the throughput*, is limited by the available resources. The throughput or the load of a system can be a primary requirement, for instance in production systems (a waferstepper exposes 100 wafers/hour). For other systems the response times are leading, but the boundary condition with respect to throughput or load must be specified. Figure 6 shows an example of the throughput.

Another aspect of the specific requirement is how much is included or excluded in the defined number, is it a gross or a nett number, see figure 7? Many specified numbers, also of components being used are gross numbers; the limit value in ideal circumstances. In reality circumstances are not ideal and the performance is lower due to all kinds of overheads and losses.

bus bandwidth, processor load [memory usage]  
useful macroscopic views, be aware of microscopic behavior

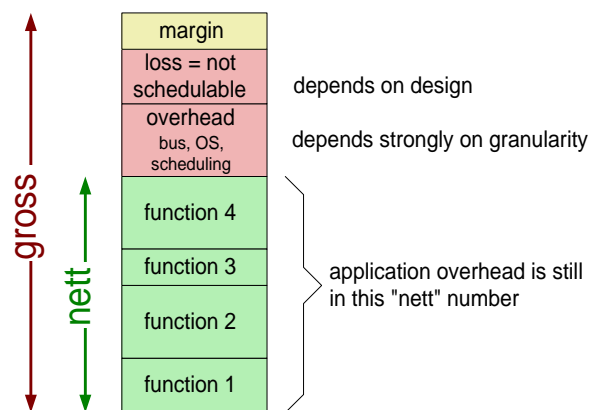


Figure 7: Gross versus Nett

### 3 Design recommendations and patterns

This section provides number of general recommendations and design patterns for execution architectures. A very generic recommendation is the separation of concerns, see figure 8. It is recommended to factor out the hard real time software and to minimize the influence of the soft real time software on the hard real time software. This is far from trivial, for instance locks on shared data structures can violate this decoupling with disastrous results.

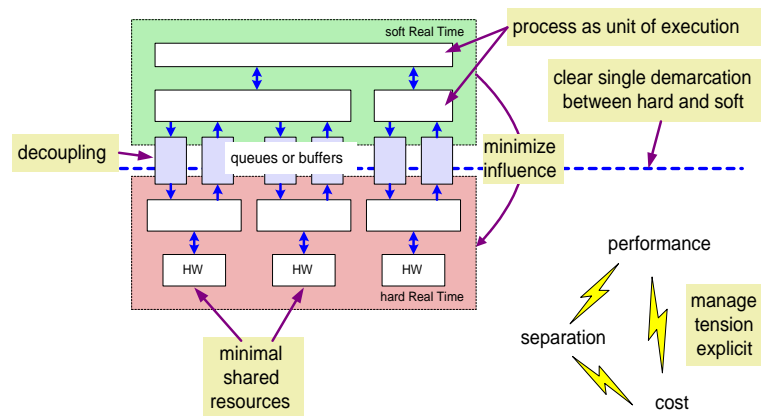


Figure 8: Design recommendations separation of concerns

In the hard real time part of the system all sharing of resources is a form of coupling complicating the real time design. Sharing of resources is often done for cost reasons, cost and real time behavior can be conflicting. Note that for coupled algorithms sharing is sometimes better than separation, due to the communication overhead.

For understandability reasons the design should be kept as simple as possible. True hard real time systems should be explainable in a few sheets of paper, otherwise the chance of failure due to missing insight is too large. **But** the fundamentals of these few sheets of paper are formed by an understanding of many critical details. Reality is quite complex, with many related details.

Figure 9 shows an number of understandability recommendations. The simplicity requirement for execution architectures can be translated in: limit the use of process, tasks, threads and priorities. It must be possible for the human designer to reason about the system. When designers can only discuss the system by means of simulations, this is often a symptom of uncontrolled complexity, it is a warning signal for the architect.

One of the crucial design choices is the granularity of operation. The granularity of operation may be different per operation, for instance I/O might be based on large chunk to minimize I/O overhead, while the processing might be based on

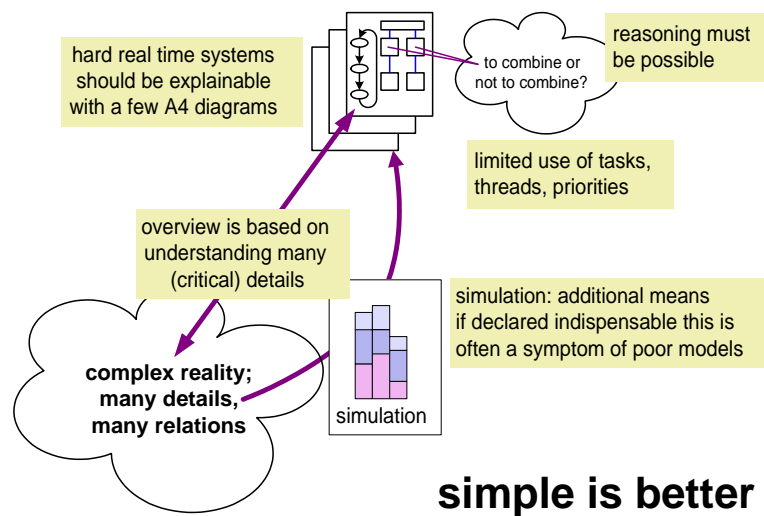


Figure 9: Design recommendations understandability

small chunks to minimize overall memory usage. Figure 10 shows some granularity considerations. Again a best solution does not exist, a balance must be found between issues such as memory use, overhead, latency, flexibility et cetera.

Several synchronization design patterns can be used. Figure 11 shows two main patterns: a complete synchronous design for hard real time subsystems, which can be nicely decoupled from the rest of the system and asynchronous design based on timers and interrupts as activating triggers for (asynchronous) threads.

Some designers declare a synchronous approach as inferior. However this type of design can be highly effective due to its simplicity and well defined behavior, see figure12. This approach breaks down when asynchronous events have to be handled, causing if-then-else structures in the code or worst case growing into a complete proprietary scheduler.

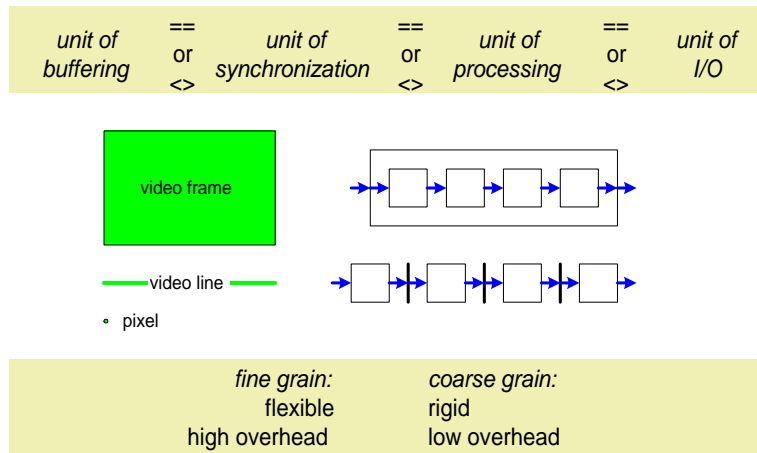


Figure 10: Granularity considerations

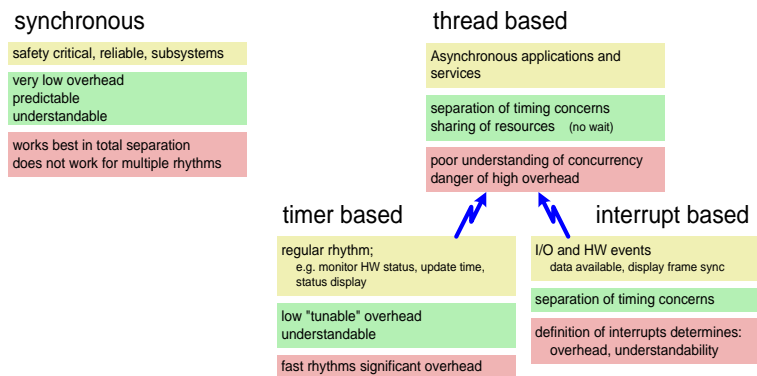


Figure 11: Design patterns

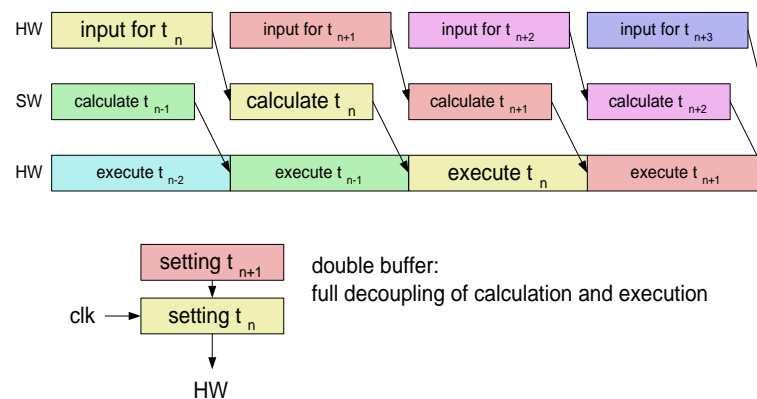


Figure 12: Synchronous design

## 4 Logarithmic time axis

Plotting application timing requirements and technology timing characteristics on a logarithmic time axis provides insight in the technology required for a given application requirement. Figure 13 shows an example of such a plot. Note the very large dynamic range of the time axis from picoseconds to seconds.

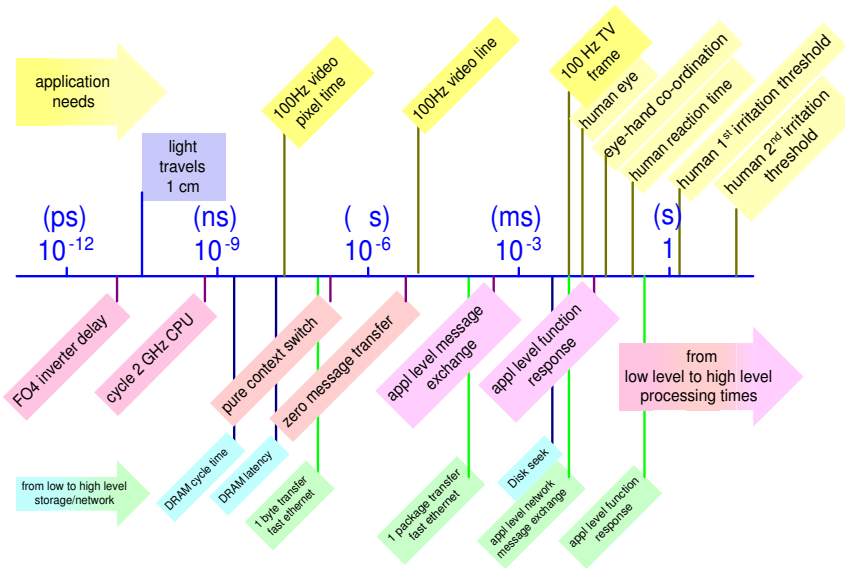


Figure 13: Actual timing on logarithmic scale

At the technology side very dedicated hardware solutions dominate. The *FO4* time is a process technology timing indication. It indicates the time needed to invert a signal and distribute it to 4 gates. It represents more or less the lower limit for digital solutions. The next item is the cycle time, which need to be multiplied with the needed number of cycles to do something useful. In most systems a real time executive facilitates the basic needs, however from timing point of view a pure context switch time is the minimum time to take into account for any action. At higher levels of abstractions even more overhead is involved: from zero message transfer to application level message transfer. Every abstraction step provides some additional software service and application freedom (independence of processor, process, service, system, ...) at the penalty of increased overhead.

For many systems the most powerful (righthand) technology should be used, maximizing the benefits of higher abstraction layers, **but** only as long as it fits in timing, power and cost requirements.

## 5 Measuring

The designer of the execution architecture has to build up understanding of the domain and the system. Measuring is crucial to build up this understanding. Measurements can be done at all aggregation levels. It is recommended to at least measure a number of elementary characteristics, micro benchmarks. Figure 14 shows a list of measurable elementary characteristics.

	<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 14: Typical micro benchmarks for timing aspects

Often small code snippets can be used to do the measurements. When using these numbers the circumstances and assumptions behind the numbers should be taken into account. The most simple measurements often result in a more *gross*-like number. More intelligent measurement methods can provide also the more *nett*-like numbers. The designer should build up understanding for the sensitivity of these numbers, for instance by experimenting with different measurements and measurements conditions.

The design team must make conscious design decisions based on these numbers. Using gross numbers will backfire, because the product suffers in the real world from many performance leaks. At the other hand a complete worst case design is often way too pessimistic, resulting in a too expensive over dimensioned system.

Many functions depend on a number of parameters, for instance transfer size. Figure 15 shows an example of a disk transfer time as a function of the block size. The function has regular discontinuities, which are related to the disk cycle time. This type of disk can be used in the design in different ways.

Without any design constraints on the disk usage, the transfer rate has to be based on the worst case data rate. This design is decoupled from most disk characteristics, except for a minimum transfer rate assumption. The penalty is that only a fraction of the disk speed capability is used.

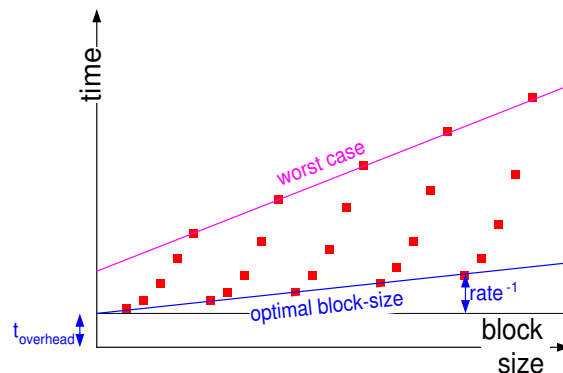


Figure 15: The transfer time as function of blocksize

The alternative is to use the known disk characteristics, for instance by prescribing optimal block sizes. The benefit is that the full capability of the disk can be used, but this software is much more coupled to this specific disk.

memory budget in Mbytes	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 16: Example of a memory budget

Figure 16 shows an example of a memory budget. This budget is entirely based on measurements on an older version of the system and adapted to the expected design changes. The structure of the budget is driven by the fact that it should be easily measurable. The decomposition is per process, because the operating facilitates memory measurements per process. The operating system also recognizes 2 types of memory: read-only (= program code) and heap (=run time used). In this table the second type of memory is further subdivided into object data (= all kind of housekeeping data) and bulk data (=image pixel matrices). The bulk data is managed explicitly to prevent fragmentation and to guarantee its limits. The bulk

data usage is configurable.

<i>complications</i>	<i>measures</i>
cache	considered margin
bus allocation	explicit behavior
memory management	architecture rules
garbage collection	monitoring, logging
memory (buffer, storage) fragmentation	pool management
non preemptable OS activities	feedback to architect
"hidden" dependencies (ie [dead]locks)	flipover simulation
systematic "coincidences", avalanche triggers	
instable response, performance	

Figure 17: Complicating factors and measures

Figure 17 gives a list of often occurring complications in execution architecture design and measurements and some counter measures which can be taken.

## 6 Acknowledgements

This course is a joint effort of Ton Kosteljik and myself. Ton creates some of the course modules. He proved to be an inspiring sparring partner.

## References

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

## History

**Version: 1.1, date: January 30, 2003 changed by: Gerrit Muller**

- repaired figure 1 in Article

**Version: 1.0, date: December 4, 2002 changed by: Gerrit Muller**

- changed design patterns diagram

**Version: 0.4, date: October 1, 2002 changed by: Gerrit Muller**

- changed execution architecture diagram
- changed separation of concerns diagram

**Version: 0.3, date: September 6, 2002 changed by: Gerrit Muller**

- added execution architecture diagram

**Version: 0, date: August 7, 2002 changed by: Gerrit Muller**

- Created, no changelog yet