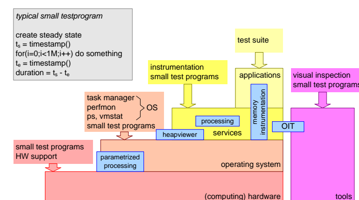
Exploring an existing code base: measurements and instrumentation

-



Gerrit Muller

University of South-Eastern Norway-NISE

Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway

gaudisite@gmail.com

Abstract

Many architects struggle with a given large code-base, where a lot of knowledge about the code is in the head of people or worse where the knowledge has disappeared. One of the means to recover insight from a code base is by measuring and instrumenting the code-base. This presentation addresses measurements of the static aspects of the code, as well as instrumentation to obtain insight in the dynamic aspects of the code.

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: 0.4 status: draft June 21, 2020

1 Introduction

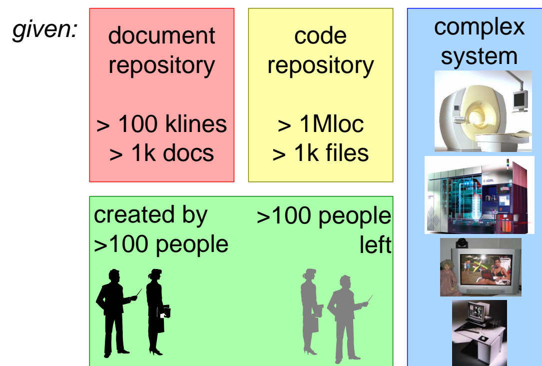


Figure 1: Problem Statement

Architects are frequently confronted with the request to extend an existing system, while most knowledge about the internals of the system are not readily available. Figure 1 shows that the requested extension may address new functions and interfaces, higher performance levels and other improvements. The system itself is documented in a large pile of documentation, typically more than 100,000 lines of documentation in more than 1000 documents. The code inside the system is typically beyond 1 million lines in more than 1000 files. Documentation and code are typically created by hundreds of people in several years, which means that also hundreds of people have left. In summary, the staring point is one big mass of electronic information resulting in some very complex system with hundreds of people involved.

The main questions we will address are:

- How do we get sufficient insight in the internals of the system, in order to extend the system with an acceptable risk level?
- What is a workable approach to attack this big mass of electronic information?

We will address the first question by following the steps as shown in Figure 2. The first step is to look for available overview information. Next step is to analyze the static structure of the software, followed by observing, measuring and analyzing the dynamic system behavior. The last step is to iterate over the first three steps and to recreate an architectural description.

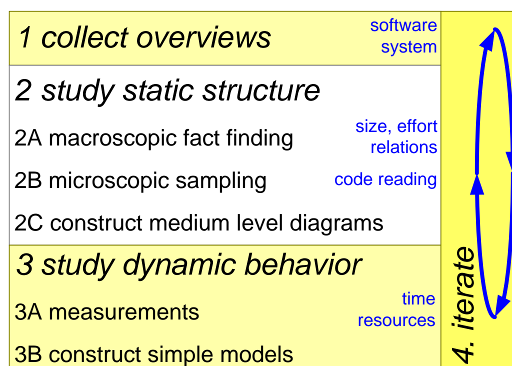


Figure 2: Overview of Approach and Article Structure

2 Collect Overviews

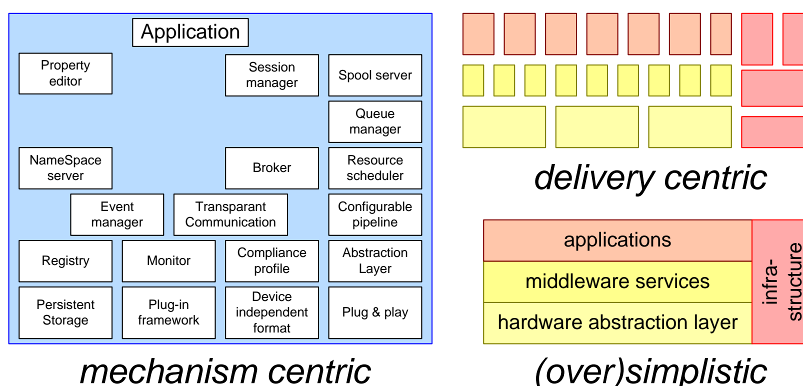


Figure 3: SW Overview(s)

Overview documentation can be partially found in the formal documentation system, however the remaining development crew is also an important source of information. So the first step is to browse through the formal documentation and to interview people. When you do this in the software department you will quickly get a few figures as shown in Figure 3. The question "please show me the software architecture" will often result in a diagram showing the mechanisms being used, the layer structure, and the package structure of the repository.

At least as important is to get an overview of what the system does and how the system actually works, information that is entirely missing in the earlier diagrams. Often this information is distributed more and can be found outside the software department. Figure 4 shows a number of typical overview diagrams: subsystem

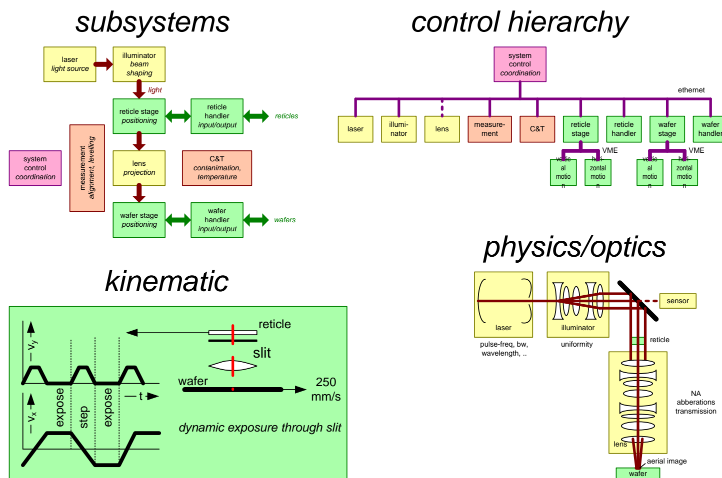


Figure 4: System Overviews

decomposition, control hierarchy of physical units, kinematic behavior, and the physics or optics structure.

Recovering diagrams as shown here can be done in a few days. Don't spend more time during this first step. In the next steps you will often discover that reality differs from this first design snapshot.

3 Static Measurements and Analysis

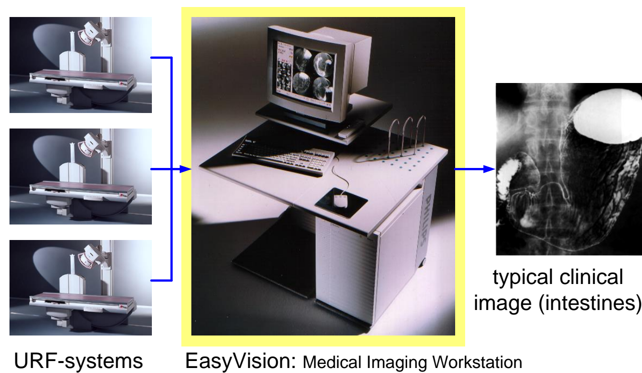


Figure 5: Case 1, a medical imaging workstation

We will use the medical imaging workstation[2] as case for the static analysis. The medical imaging workstation is an add-on product to existing X-ray systems,

introduced in the market in 1992. X-ray systems used to print the imaging results directly on film, by means of a so called CRT-copy, an exact copy of the monitor display on film. The workstation is positioned between X-ray system and printer and adds formatting and layout capabilities. One workstation can serve multiple examination rooms, see figure 5.

```
> wc -l  *.m
72 Acquisition.m
13 AcquisitionFacility.m
330 ActiveDataCollection.m
132 ActiveDataObject.m
304 Activity.m
281 ActivityList.m
551 AnnotateParser.m
1106 AnnotateTool.m
624 AnyOfList.m
466 AsyncBulkDataIO.m
264 AsyncDeviceIO.m
261 AsyncLocalDbIO.m
334 AsyncRemoteDbIO.m
205 AsyncSocketIO.m
```

version control information:
#new files
#deleted files
#changes per file since ...

package information:
files

metrics:
QAC type information
methods
globals

Figure 6: Examples of Macroscopic Fact Finding

A quick way of diving into the big mass of software code is by looking at file sizes. This product used a flat directory in the code repository, enabling a simple $wc - l * .m$[1] to count the number of lines per file. Figure 6 shows at the left hand side some of the result of the word-count command. The right hand side of this figure shows other analysis data that easily can be obtained, such as version control data, package data, and data from metrics tools such as QAC.

A yardstick is needed to assess the numbers. Figure 7 shows a histogram of the sizes of files. This histogram is color coded:

0..50 or more than 1500 lines red: suspect, should be analyzed further

50..100 or 1000..1500 yellow: slightly suspect, sample a some of these files

100..1000 green: size is OK, sample a few of these files

The idea behind this yardstick is that big files are difficult to understand and that these files are likely candidates for further modularization. Very small files are unwanted, because of fragmentation of design and understanding. These files might be the result of too much modularization. Very big and very small files are called suspect, there might be a good reason for particular files to be large without the need for further modularization.

[1]This system was programmed mostly in Objective-C with the extension $.m$

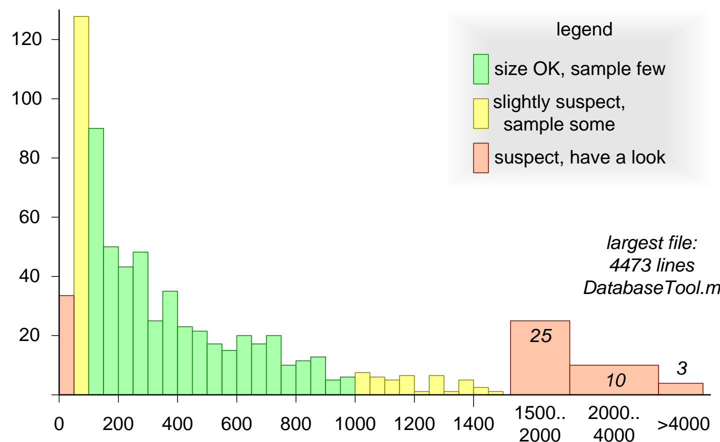


Figure 7: Histogram of File Sizes EV R1.0

The next step is to dive in, by looking into suspect files and by sampling slightly suspect or non-suspect files. The justification of the file-size can be understood by reading the code in these files. Figure 8 shows the outcome of code reading of suspect and sampled files. A number of the small files can be explained by a chosen design pattern of the database, this justification changes the size annotation to OK (green). Many top level user interface modules turn out be quite large, mostly with panel definitions, buttons, call-back assignments and other definitions; further modularization would be rather artificial and fragmenting. These user interface files also change to OK (green). Another set of read files address something complex and might need further refactoring.

The analysis so far was performed on the "current" snapshot of the repository. The change history of the repository is also a gold mine of information. Figure 9 gives an example of the amount of changes per check-in of one specific file. Within a few months two big changes have taken place in this file, a third even bigger change occurs again a few months later. Analysis in retrospect showed that this file was redesigned twice, without satisfactory results. Then a mature designer did the redesign as it should have been done in the first place. The latest redesign voided the earlier redesigns.

Other examples of insights obtained from the change history are suspect files that are changing all the time. Many systems have some kind of centralized system constants file. This type of file is related to many other files, making it very sensitive to changes by many different programmers. A shaky implementation can also cause a continuous stream of changes: this is a highly suspect type of file.

The macroscopic analysis by means of line counts and the microscopic analysis by code reading provides only local insights. Static analysis of dependencies provides a more global insight. Layering diagrams of packages (packages are

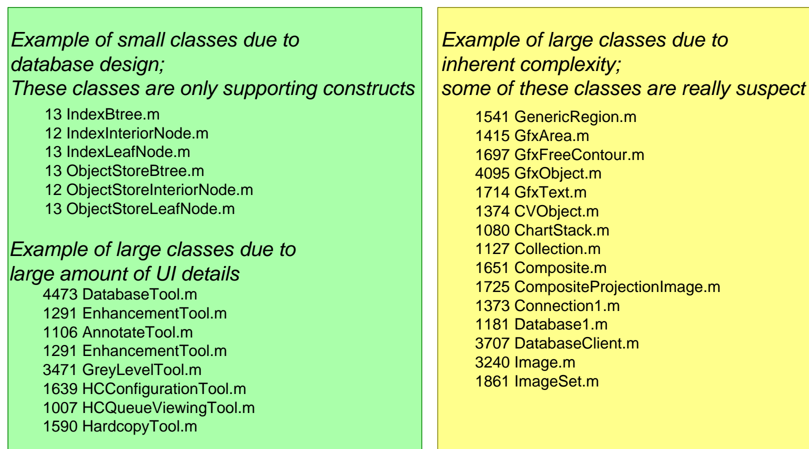


Figure 8: Microscopic Sampling (Code Reading)

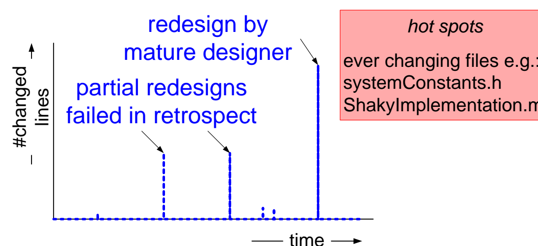


Figure 9: Changes Over Time

files that are grouped together for organizational and logistics purposes) are a good means to visualize dependencies. Figure 10 shows a simplified layering diagram of the medical imaging workstation. The actual static dependency diagram counted 15 layers[2].

We have shown that quantification of static code aspects helped to get insight in relative complexity and effort of components and functions. This activity calibrates the intuition of the architect. The numbers only get useful when macroscopic numbers are combined by microscopic detailed observations. The combination of macroscopic and microscopic data can be combined in medium level diagrams, such as layering diagrams. Layering diagrams are means to show dependencies and relations. As a side effect of the static analysis ”hot spots”, suspect parts of the code, are detected. This conclusion is summarized in Figure 11.

[2]Initially too many cross relations existed, preventing the creation of a useful layering diagram. However, the attempt to construct such a diagram triggered a refactoring step to remove the worst dependencies.

The real layering diagram did have >15 layers

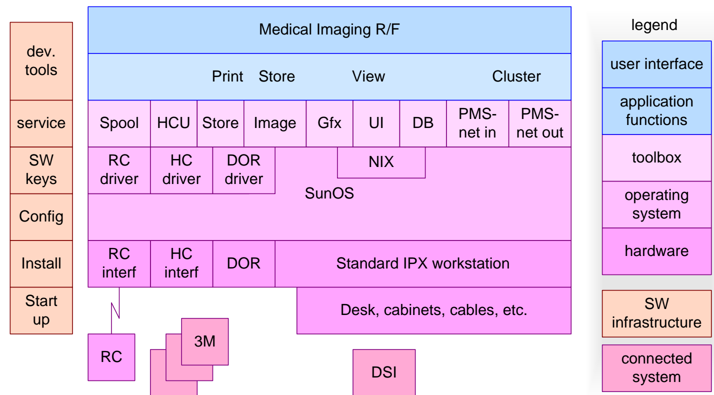


Figure 10: Simplified Layering Diagram

Quantification helps to *calibrate* the *intuition* of the architect

Macroscopic numbers related to *code level* understanding provides insight

+ relative complexity
+ relative effort
+ hot spots
+ (static) dependencies and relations

Figure 11: Conclusions Static Exploration

4 Dynamic measurements and Analysis

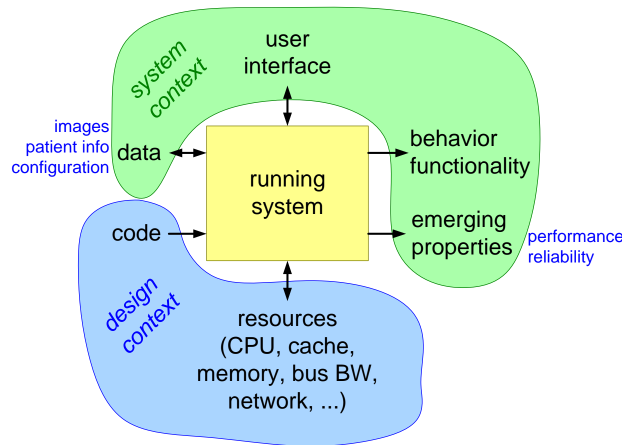


Figure 12: Dynamic behavior reveals much more than static analysis

We started the more in depth analysis by looking at static structures. Static structures can easily be analyzed, in other words static analysis has a low threshold. However, most of the system complexity is in the dynamic behavior. Most system decisions are related to the dynamics of the system. Figure 12 shows the running system in its context. The system runs the statically defined code on computing resources. The dynamic behavior of the system in influenced by the data (configuration data, patient information, images) and by the user interacting with the system. These complex systems show emerging properties, such as performance and reliability. Of course the product creation process is set up in such a way that these properties do not emerge completely random, but within desired margins.

We propose to tackle the dynamic analysis by measuring and analyzing the system at several levels, as shown in Figure 13. 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:

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

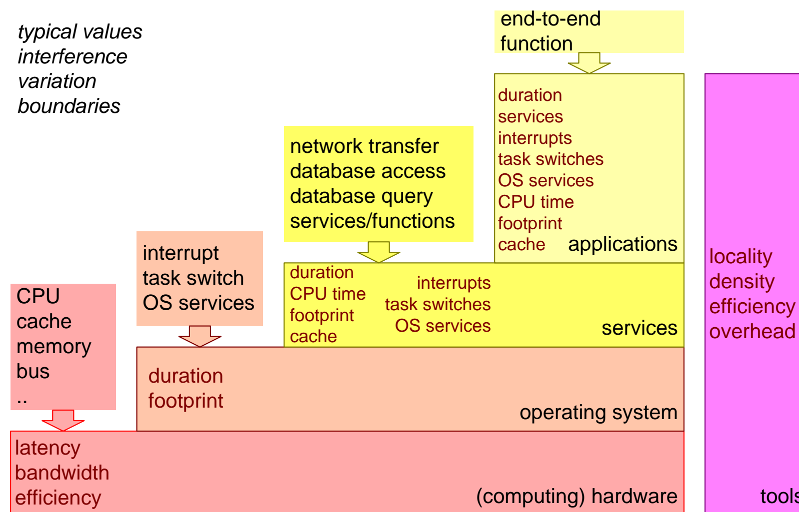


Figure 13: Layered Benchmarking

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.

Figure 14 shows the rendering pipeline as used in the medical imaging workstation. Enhancement is a filter operation. The coefficients of the enhancement kernel are predefined in the acquisition system. The interpolation is used to resize the image from acquisition resolution to the desired view-port (or film-port) size. The grey-levels for display are determined by means of a lookup table. A lookup table (LUT) is a fast and flexible implementation of a mapping function. Normally the mapping is linear: the slope determines the contrast and the vertical offset the brightness of

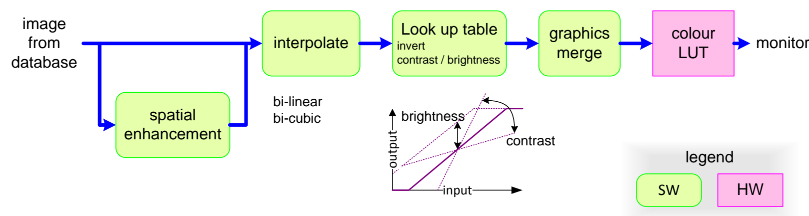


Figure 14: Example: Processing HW and Service Performance

the image. Finally graphics and text are superimposed on the image, for instance for image identification and for annotations by the user.

The CPU is a limited resource for the Medical Imaging Workstation. The performance and throughput of the system depend strongly on the available processing power and the efficiency of using the processing power. CPU time and memory can be exchanged partially, for instance by using caches to store intermediate results.

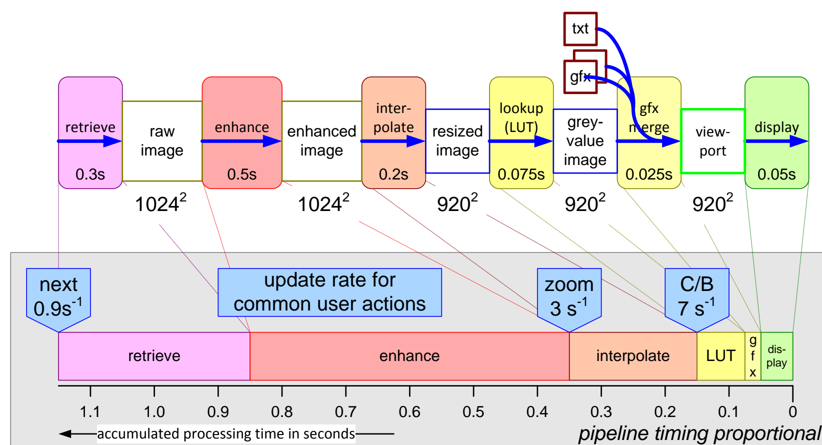


Figure 15: The CPU processing times are shown per step in the processing pipeline. The processing times are mapped on a proportional time line to visualize the viewing responsiveness

Figure 15 shows typical update speeds and processing times for a single image user interface layout. Contrast brightness (C/B in the figure) changes must be fast, to give immediate visual feedback when turning a contrast or brightness wheel. Working on the cached resized image about 7 updates per second are possible, which is barely sufficient. The gain of the cached design relative to the non-cached design is about a factor 8 (7 updates per second versus 0.9 updates per second). Zooming and panning is done with an update rate of 3 updates per second. The performance gain for zooming and panning is from application viewpoint less

important, because these functions are used only exceptionally in the daily use. Retrieving the next image (also a very frequent user operation), requires somewhat more than a second, which was acceptable at that moment in time.

The processing functions are measured at two levels:

- hardware capability (CPU, cache, memory)
- service level performance (the complete processing pipeline as used by applications)

The hardware oriented micro-benchmarks measure the performance of individual processing steps as function of operation, image size and pixel size. In this way insight is obtained in the raw CPU speed (mostly operation dependent), memory behavior (for instance consequences of alignment), and cache performance (depends on image size and algorithm). The service level performance is more aggregated functionality including many higher level design decisions, such as caching and granularity choices.

The image processing is at the core of this system. Hardware characteristics and low level software design choices have a big impact on system performance. As a consequence many higher level design constructs are influenced by hardware and low level software constraints. Measuring these core performance numbers often results in insight in system design choices:

- Why are dedicated processing functions or pipelines being used?
- What is the logic behind the ordering?
- Why are these elements stored persistently?

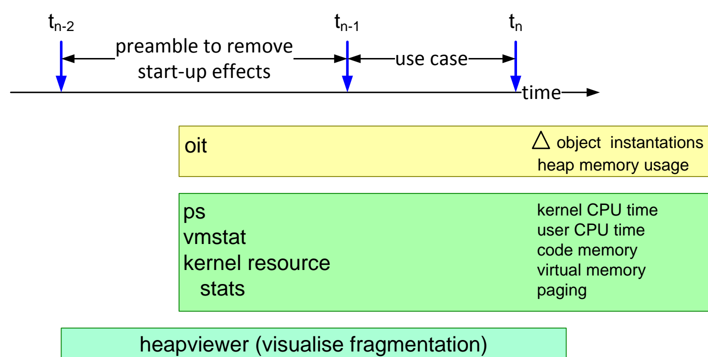


Figure 16: Resource Measurement Tools

The most important tools are: *Object Instantiation Tracing*, *standard Unix utilities* and a *heap viewer*. The resource usage is measured at well defined moments

in time, by means of events. The entire software is event-based. The event for resource measurement purposes can be fired by programming it at the desired point in the code, or by a user interface event, or by means of the Unix command line.

The resource usage is measured twice: before performing the use case under study and afterwards. The measurement results show both the changes in resource usage as well as the absolute numbers. The initialization often takes more time in the beginning, while in a steady running system no more initialization takes place. Normally the real measurement is preceded by a set of actions to bring the system in a kind of steady state.

Note that the budget definitions and the *Unix utilities* fit well together, by design. The types of memory budgeted are the same as the types of memory measured by the Unix utilities. The typically used Unix utilities are:

ps process status and resource usage per process

vmstat virtual memory statistics

kernel resource stats kernel specific resource usage

The *heap-viewer* shows the free and allocated memory blocks in different colors, comparable with the standard Windows disk defragmentation utilities.

class name	current nr of objects	deleted since t_{n-1}	created since t_{n-1}	heap memory usage
AsynchronousIO	0	-3	+3	
AttributeEntry	237	-1	+5	
BitMap	21	-4	+8	
BoundedFloatingPoint	1034	-3	+22	
BoundedInteger	684	-1	+9	
BtreeNode1	200	-3	+3	[819200]
BulkData	25	0	1	[8388608]
ButtonGadget	34	0	2	
ButtonStack	12	0	1	
ByteArray	156	-4	+12	[13252]

Figure 17: Example output of OIT (Object Instantiation Tracing) tool

The *Object Instantiation Tracing* (OIT) keeps track of all object instantiations and disposals. It provides an absolute count of all the objects and the change in the number of objectives relative to the previous measurement. The system is programmed with Objective-C. This language makes use of run-time environment, controlling the creation and deletion of objects and the associated housekeeping. The creation and deletion operations of this run-time environment were rerouted via a small piece of code that maintained the statistics per class of object instantiations and destructions. At the moment of a trigger this administration was saved in readable form. The few lines of code (and the little run time penalty) have paid

many many times. The instantiation information gives an incredible insight in the internal working of the system.

The *Object Instantiation Tracing* also provided heap memory usage per class. This information could not be obtained automatically. At every place in the code where malloc and free was called some additional code was required to get this information. Figure 17 shows an example output of the OIT tool. Per class the current number of objects is shown, the number of deleted and created objects since the previous measurement and the amount of heap memory in use. The user of this tool knows the use case that is being measured. In this case, for example, the *next image* function. For this simple function 8 new BitMaps are allocated and 3 AsynchronousIO objects are created. The user of this tool compares this number with his expectation. This comparison provides more insight in design and implementation.

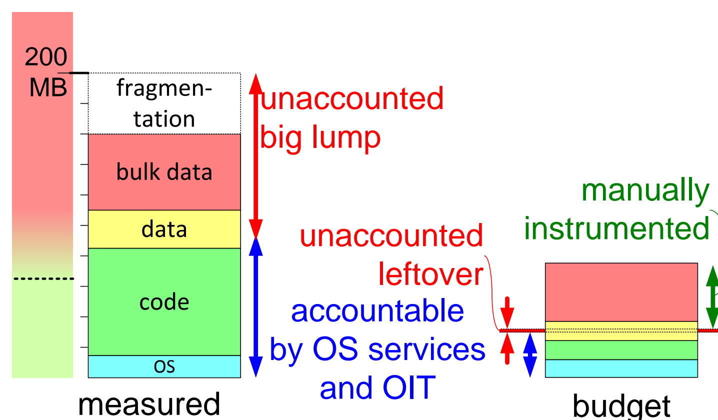


Figure 18: Memory Instrumentation

Early during the development of the medical imaging workstation we hit performance problems caused by the usage of too much memory. Initial system level measurements indicated a use of 200 MByte, while the available physical memory was only 64 MByte. The available measurement tools were useful to understand about half of the memory use, the other half was one big unaccounted lump of memory. Manual instrumentation of the code, by intercepting memory allocations and de-allocations helped to understand 80% of this unaccounted lump. This manual instrumentation activity was stopped at 80%, because a sufficient level of understanding was reached.

Figure 19 shows an overview of the benchmarking and other measurement tools used during the design. The overview shows per tool what is measured and why, and how accurate the result is. It also shows when the tool is being used.

The Objective-C overhead measurements, to measure the method call overhead

	test / benchmark	what, why	accuracy	when
public	SpecInt (by suppliers)	CPU integer	coarse	new hardware
	Byte benchmark	computer platform performance OS, shell, file I/O	coarse	new hardware new OS release
self made	file I/O	file I/O throughput	medium	new hardware
	image processing	CPU, cache, memory as function of image, pixel size	accurate	new hardware
	Objective-C overhead	method call overhead memory overhead	accurate	initial
	socket, network	throughput CPU overhead	accurate	ad hoc
	data base	transaction overhead query behaviour	accurate	ad hoc
	load test	throughput, CPU, memory	accurate	regression

Figure 19: Overview of benchmarks and other measurement tools

and the memory overhead caused by the underlying OO technology, is used only in the beginning. This data does not change significantly and scales reasonably with the hardware improvements.

A set of coarse benchmarking tools was used to characterize new hardware options, such as new workstations. These tools are publicly available and give a coarse indication of the hardware potential.

The application critical characterization is measured by more dedicated tools, such as the image processing benchmark, which runs all the algorithms with different image and pixel sizes. This tool is home made, because it uses the actual image processing library used in the product. The outcome of these measurements were used to make design optimizations, both in the library itself as well as in the use of the library.

Critical system functionality is measured by dedicated measurement tools, which isolate the desired functionality, such as file I/O, socket, networking and database.

The complete system is put under load conditions, by continuously importing and exporting data and storing and retrieving data. This load test was used as regression test, giving a good insight in the system throughput and in the memory and CPU usage.

Figure 20 positions the tools and instrumentation efforts in the benchmarking stack as originally shown in Figure 13. Many of the micro-benchmarks are small programs, where the benchmarks operation is repeated many times and the time is measured by time-stamps before and after the repetition. Some information is obtained by instrumentation. Test inputs are needed to obtain repeatable measurement experiments.

A more up to date example of micro-benchmarking uses the ARM9 as case, see Figure 21. A typical chip based on the ARM9 architecture has anno 2006 a

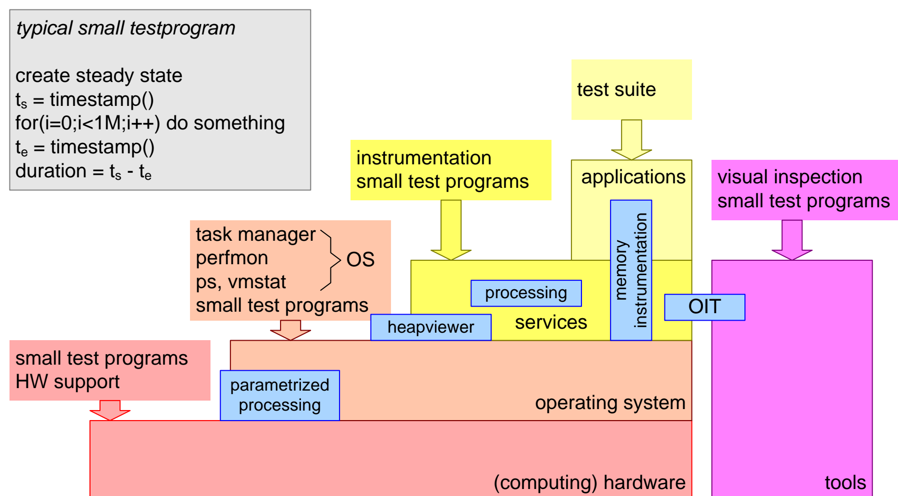


Figure 20: Tools and Instruments Positioned in the Stack

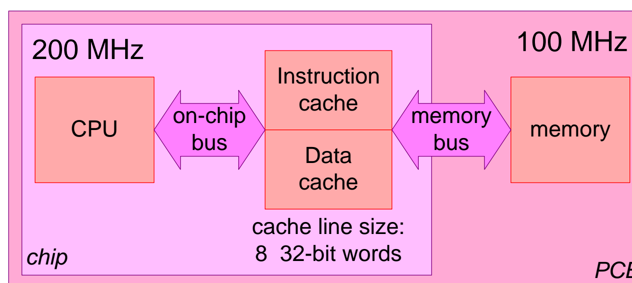


Figure 21: Case 2: ARM9 Cache Performance

clock-speed of 200 MHz. The memory is off-chip standard DRAM. The CPU chip has on-chip cache memories for instruction and data, because of the long latencies of the off-chip memory access. The memory bus is often slower than the CPU speed, anno 2006 typically 100 MHz.

Figure 22 shows more detailed timing of the memory accesses. After 22 CPU cycles the memory responds with the first word of a memory read request. Normally an entire cache line is read, consisting of 8 32-bit words. Every word takes 2 CPU cycles = 1 bus cycle. So after $22 + 8 * 2 = 38$ cycles the cache-line is loaded in the CPU.

At OS level a micro-benchmark was performed to determine the context switch time of a real-time executive on this hardware platform. The measurement results are shown in Figure 23. The measurements were done under different conditions. The most optimal time is obtained by simply triggering continuous context

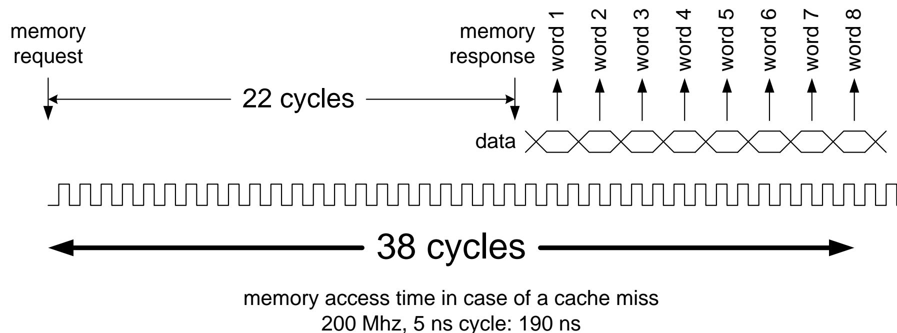


Figure 22: Example Hardware Performance

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

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

Figure 23: Actual ARM Figures

switches, without ant other activity taking place. The effect is that the context switch runs entirely from cache, resulting in a $2\mu s$ context switch time. Unfortunately, this is a highly misleading number, because in most real-world applications many activities are running on a CPU. The interrupting context switch pollutes the cache, which slows down the context switch itself, but it also slows down the interrupted activity. This effect can be simulated by forcing a cache flush in the context switch. The performance of the context switch with cache flush degrades to $10\mu s$. For comparison the measurement is also repeated with a disabled cache, which decreases the context switch even more to $50\mu s$. These measurements show the importance of the cache for the CPU load. In cache unfriendly situations (a cache flushed context switch) the CPU performance is still a factor 5 better than in the situation with a disabled cache. One reason of this improvement is the locality of instructions. For 8 consecutive instructions "only" 38 cycles are needed to load these 8 words. In case of a disabled cache $8 * (22 + 2 * 1) = 192$ cycles are needed to load the same 8 words.

Figure 24 shows the impact of context switches on system performance for

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

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

Figure 24: Context Switch Overhead

different context switch rates. Both parameters $t_{contextswitch}$ and $n_{contextswitch}$ can easily be measured and are quite indicative for system performance and overhead induced by design choices. The table shows that for the realistic number of $t_{contextswitch} = 10\mu s$ the number of context switches can be ignored with 500 context switches per second, it becomes significant for a rate of 5000 per second, while 50000 context switches per second consumes half of the available CPU power. A design based on the too optimistic $t_{contextswitch} = 2\mu s$ would assess 50000 context switches as significant, but not yet problematic.

5 Integration, Discussion and Conclusions

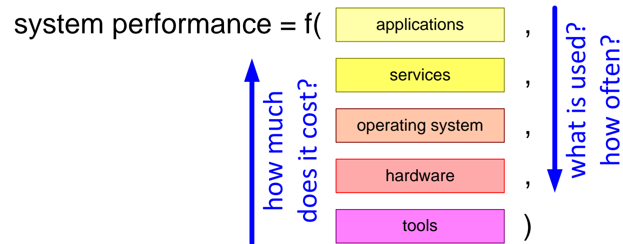


Figure 25: Performance as Function of all Layers

All data gathering activities must be processed in an intelligent way into a set of higher level diagrams and models. For example the micro-benchmarks generate a lot of data points that should be turned into a layered performance model, visualized in Figure 25. This performance model is **not** one single formula, but a more a set of related formula's. For instance the interrupt handling and task switching duration can be expressed in the lower layers as function of hardware parameters:

$$t_{interrupthandling} = f(CPU_{speed}, cache_{size}, OS)$$

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

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

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

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

At last the end-to-end performance should be explainable in terms of the lower level micro-benchmark results. By working concurrently bottom-up and top-down

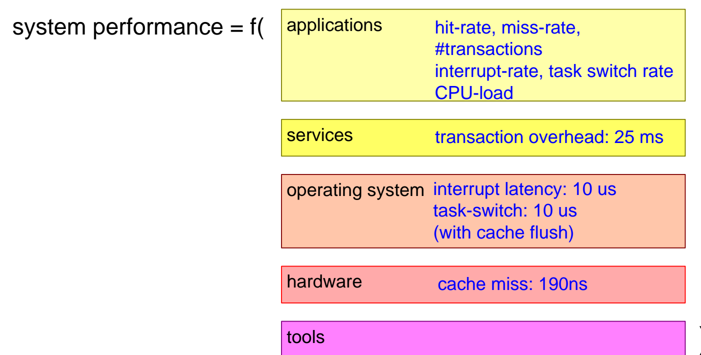


Figure 26: Annotated Performance Formula

both activities can be limited to *relevant* measurements. In a system that does only have a few interrupts, the interrupt handling time might be ignored.

Figure 26 annotates the previous figure with some typical issues and numbers from the 2 cases that have been discussed.

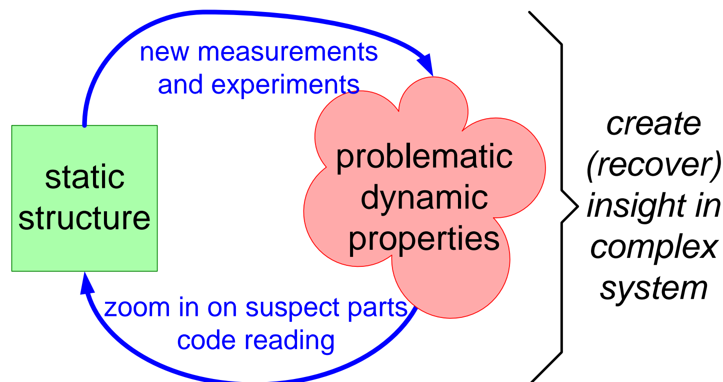


Figure 27: Keep iterating!

The dynamic measurements often trigger plenty of questions about the code structure. These questions are addressed by code reading and other static analysis, see Figure 27. The other way around code reading and static analysis results trigger a lot of questions about the dynamic behavior. These questions result in new experiments and measurements. During this integration the insight in the overall complex system increases. The insights obtained along this path should be captured in higher level diagrams. After some time a system design description is growing, allowing a more founded extension of the system as required originally.

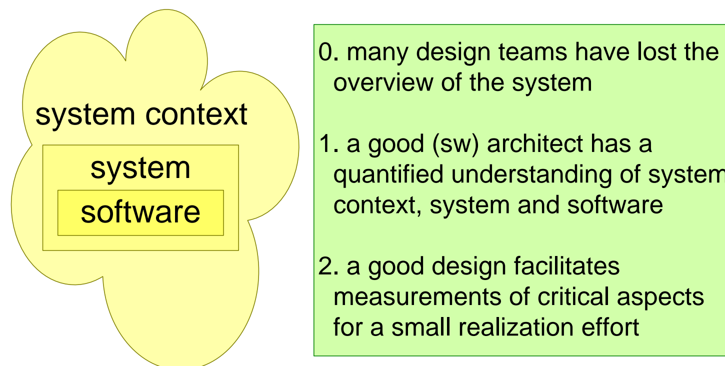


Figure 28: Discussion propositions

We close this paper with a number of discussion propositions as shown in Figure 28:

- Many design teams have lost the overview of the system. The consequence is that product extension becomes a trial and error proposition, where especially performance and reliability become unpredictable emerging properties.
- A good (SW) architect has a quantified understanding of system context, system and software, as shown at the left hand of the figure.
- A good design facilitates measurements of critical aspects for a small realization effort. Typical instrumentation functions have a size of tens to hundreds lines of code.

6 Acknowledgements

Peter van den Bosch, Marcel Verhoef and Evert de Waal reviewed the presentation and provided feedback. Ger Schoeber found a nasty missing factor of 10 in the text.

References

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

[2] Gerrit Muller. Case study: Medical imaging; from toolbox to product to platform. `http://www.gaudisite.nl/MedicalImagingPaper.pdf`, 2000.

History

Version: 0.4, date: October 26, 2006 changed by: Gerrit Muller

- corrected factor 10 mismatch of text with Figure

Version: 0.3, date: June 14, 2006 changed by: Gerrit Muller

- added text

Version: 0.2, date: June 2, 2006 changed by: Gerrit Muller

- made cases more explicit
- added examples to context of running system
- changed status to draft

Version: 0.1, date: May 19, 2006 changed by: Gerrit Muller

- minor improvements

Version: 0, date: May 8, 2006 changed by: Gerrit Muller

- Created, no changelog yet