

# Exploring an existing code base: measurements and instrumentation

by *Gerrit Muller* Embedded Systems Institute

e-mail: `gerrit.muller@embeddedsystems.nl`

`www.gaudisite.nl`

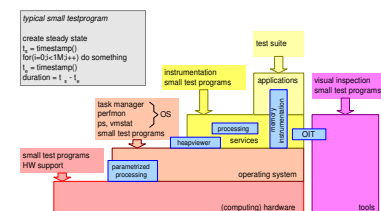
## Abstract

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.

February 10, 2011  
status: draft  
version: 0.4



# Problem Statement

*wanted:*

*new functions and interfaces, higher performance levels, improvements, et cetera*

*given:*

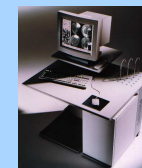
document  
repository

> 100 klines  
> 1k docs

code  
repository

> 1Mloc  
> 1k files

complex  
system

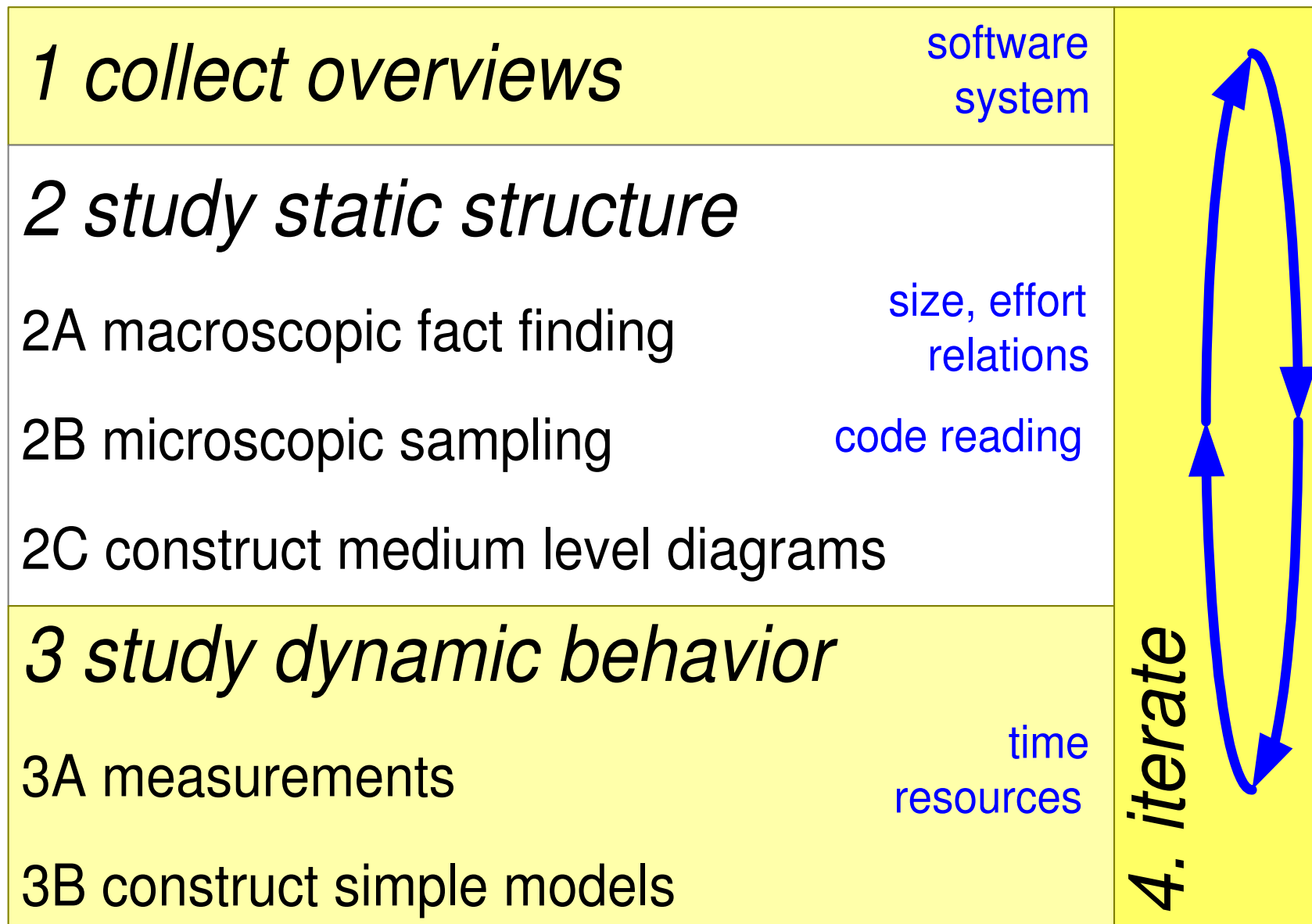


created by  
>100 people

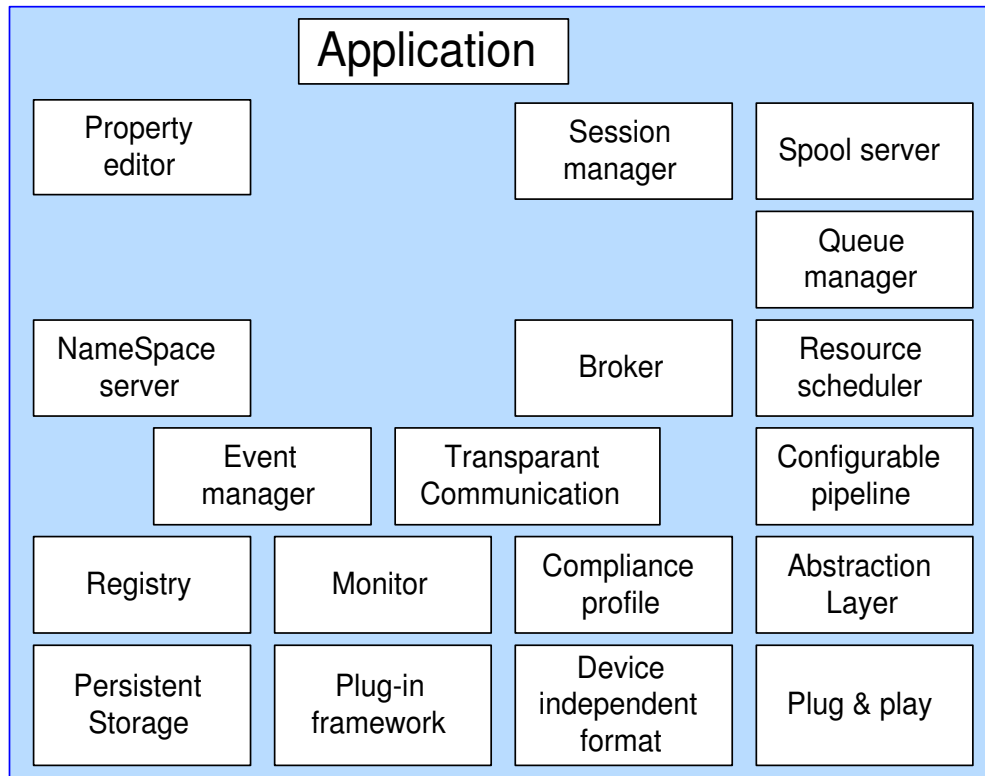
>100 people  
left



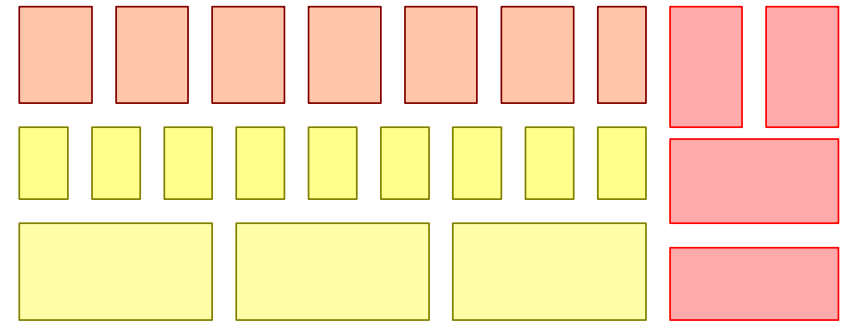
# Overview of Approach and Presentation Agenda



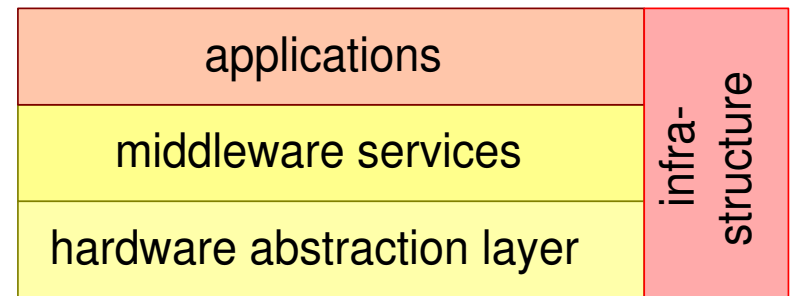
# SW Overview(s)



*mechanism centric*



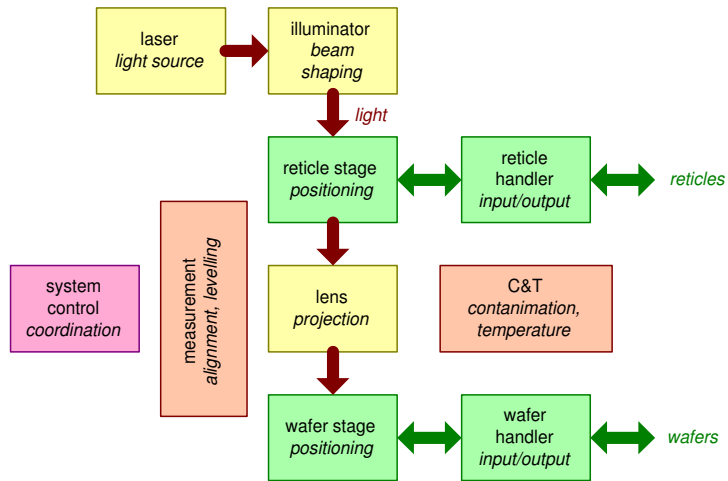
*delivery centric*



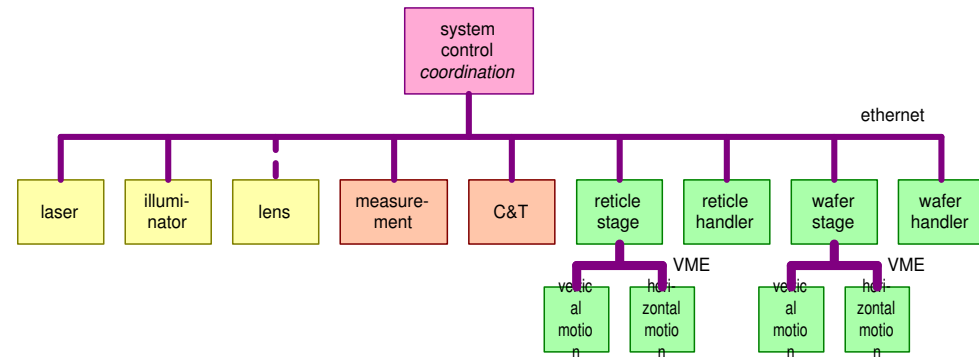
*(over)simplistic*

# System Overviews

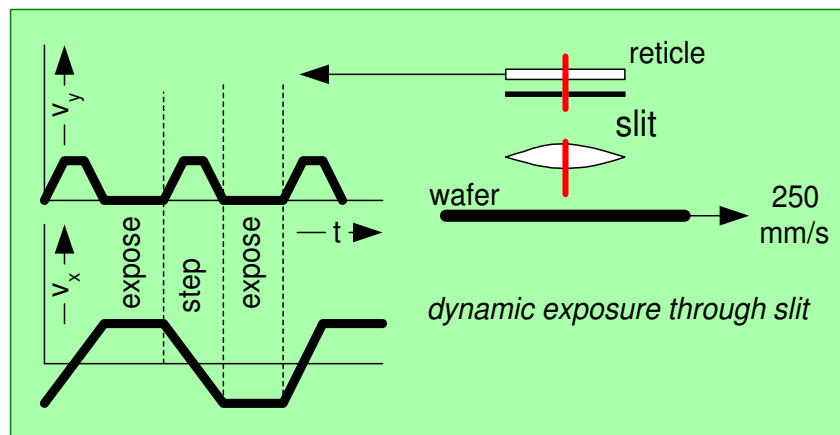
## subsystems



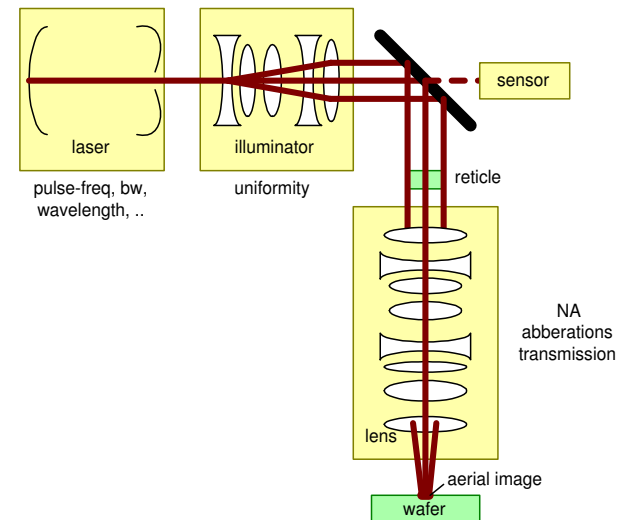
## control hierarchy



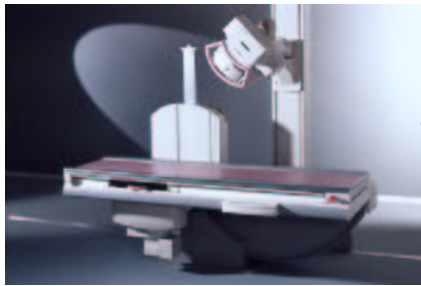
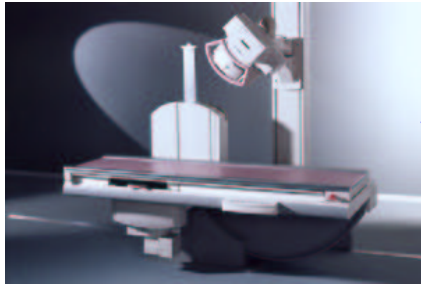
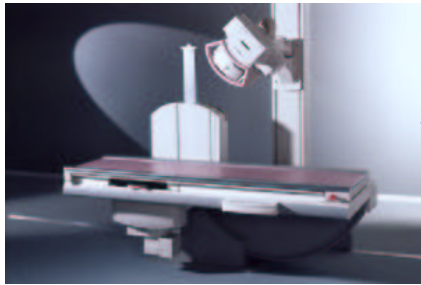
## kinematic



## physics/optics



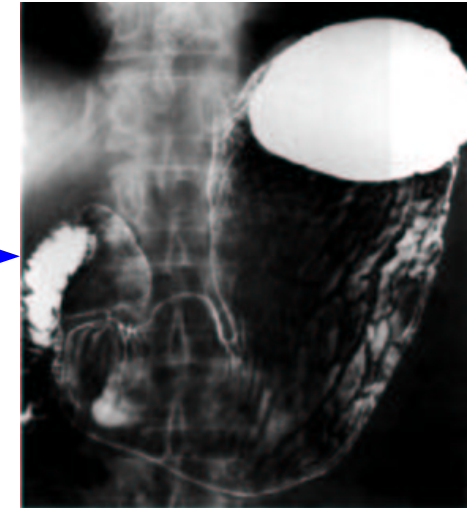
# Case 1: EasyVision (1992)



URF-systems



EasyVision: Medical Imaging Workstation



typical clinical  
image (intestines)

# Examples of Macroscopic Fact Finding

```
> 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

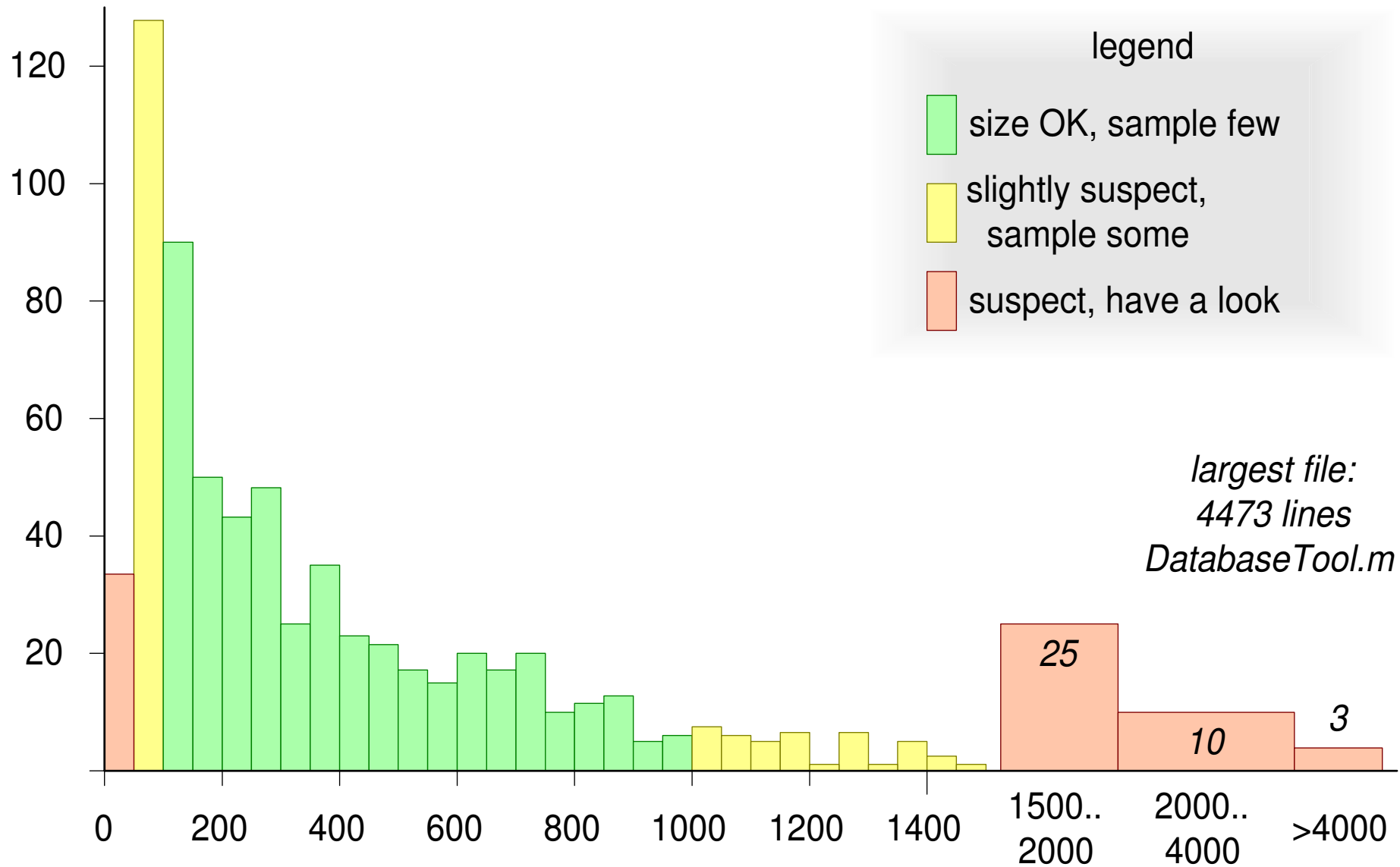
metrics:

QAC type information

# methods

# globals

# Histogram of File Sizes EV R1.0



# Microscopic Sampling (Code Reading)

*Example of small classes due to database design;*

*These classes are only supporting constructs*

- 13 IndexBtree.m
- 12 IndexInteriorNode.m
- 13 IndexLeafNode.m
- 13 ObjectStoreBtree.m
- 12 ObjectStoreInteriorNode.m
- 13 ObjectStoreLeafNode.m

*Example of large classes due to large amount of UI details*

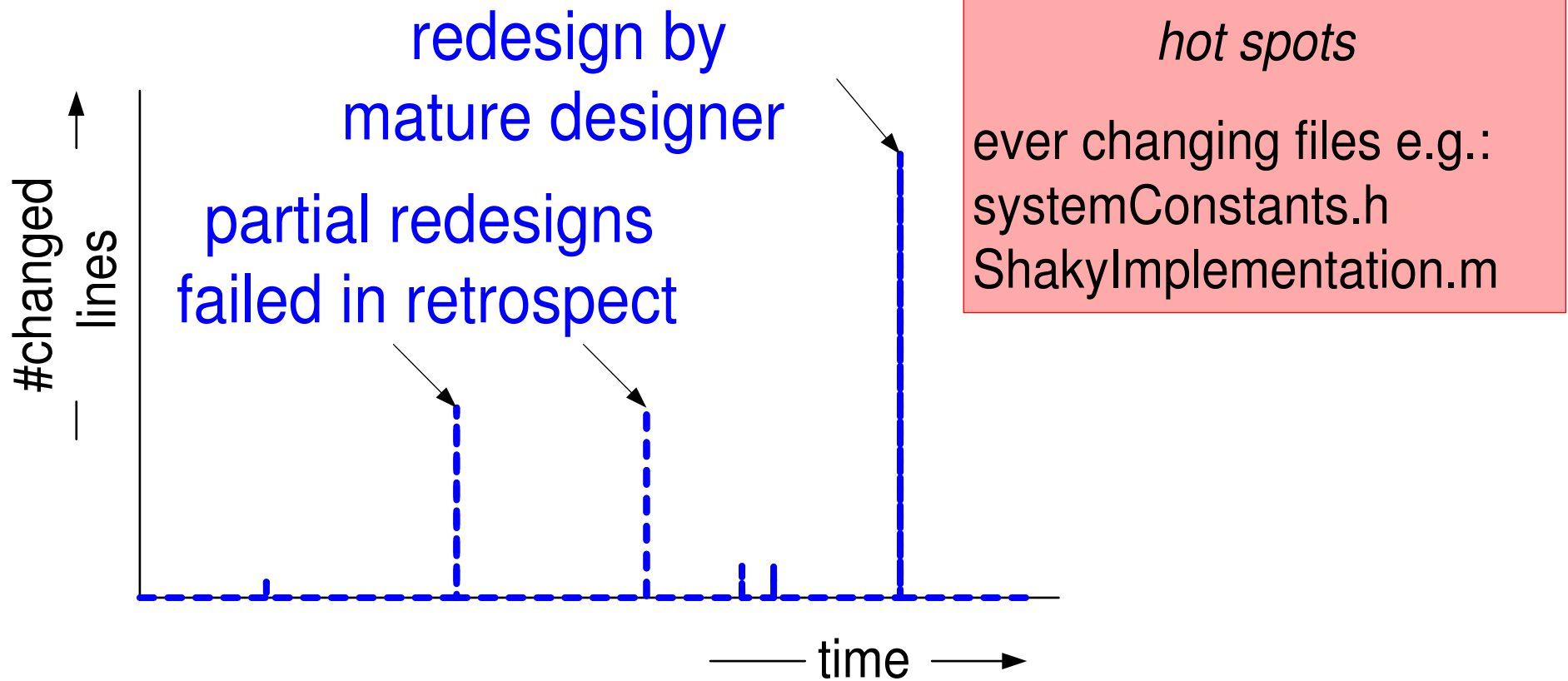
- 4473 DatabaseTool.m
- 1291 EnhancementTool.m
- 1106 AnnotateTool.m
- 1291 EnhancementTool.m
- 3471 GreyLevelTool.m
- 1639 HCConfigurationTool.m
- 1007 HCQueueViewingTool.m
- 1590 HardcopyTool.m

*Example of large classes due to inherent complexity;*

*some of these classes are really suspect*

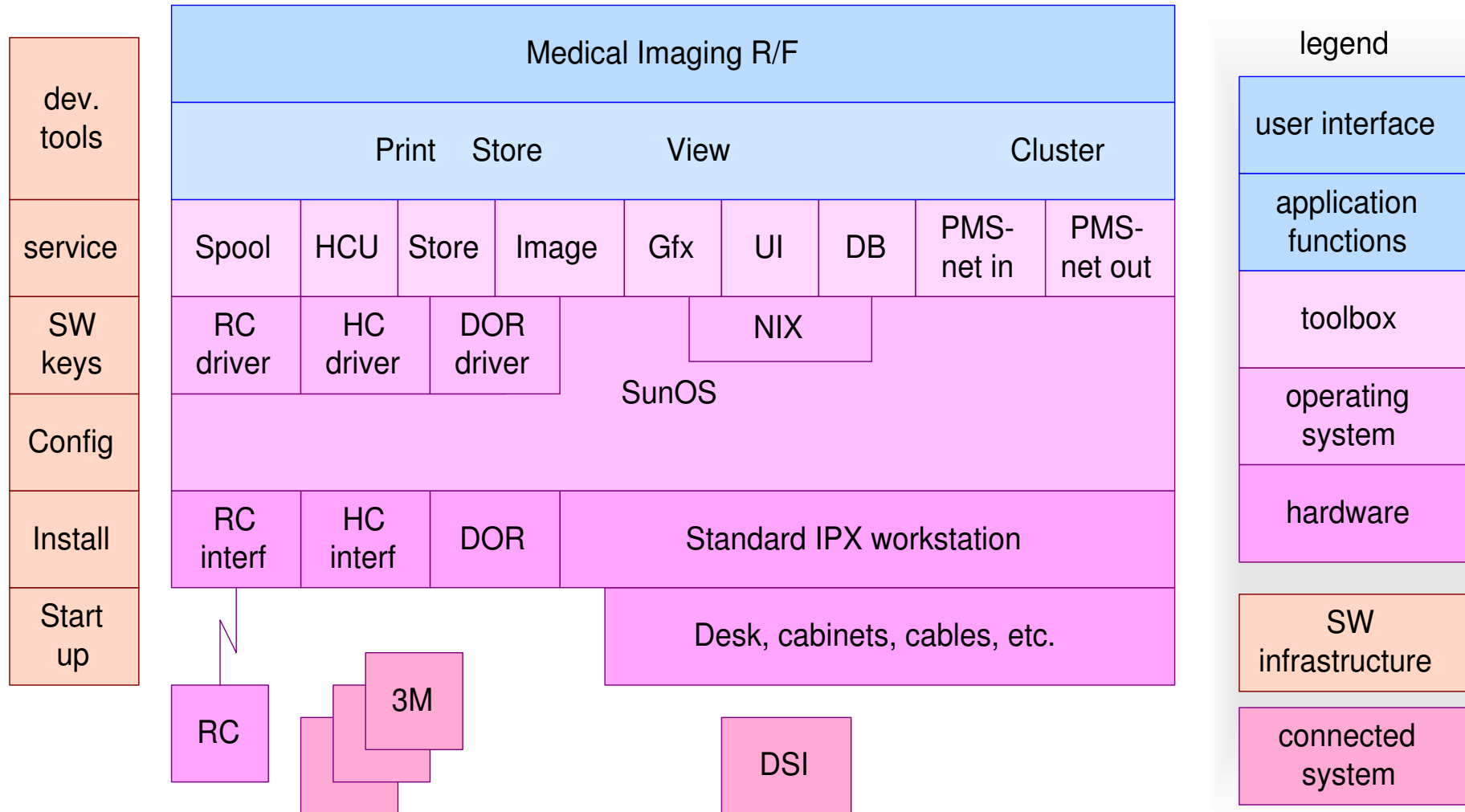
- 1541 GenericRegion.m
- 1415 GfxArea.m
- 1697 GfxFreeContour.m
- 4095 GfxObject.m
- 1714 GfxText.m
- 1374 CVObject.m
- 1080 ChartStack.m
- 1127 Collection.m
- 1651 Composite.m
- 1725 CompositeProjectionImage.m
- 1373 Connection1.m
- 1181 Database1.m
- 3707 DatabaseClient.m
- 3240 Image.m
- 1861 ImageSet.m

# Changes Over Time



# Simplified Medium Level Diagram

*The real layering diagram did have >15 layers*



# Conclusions Static Exploration

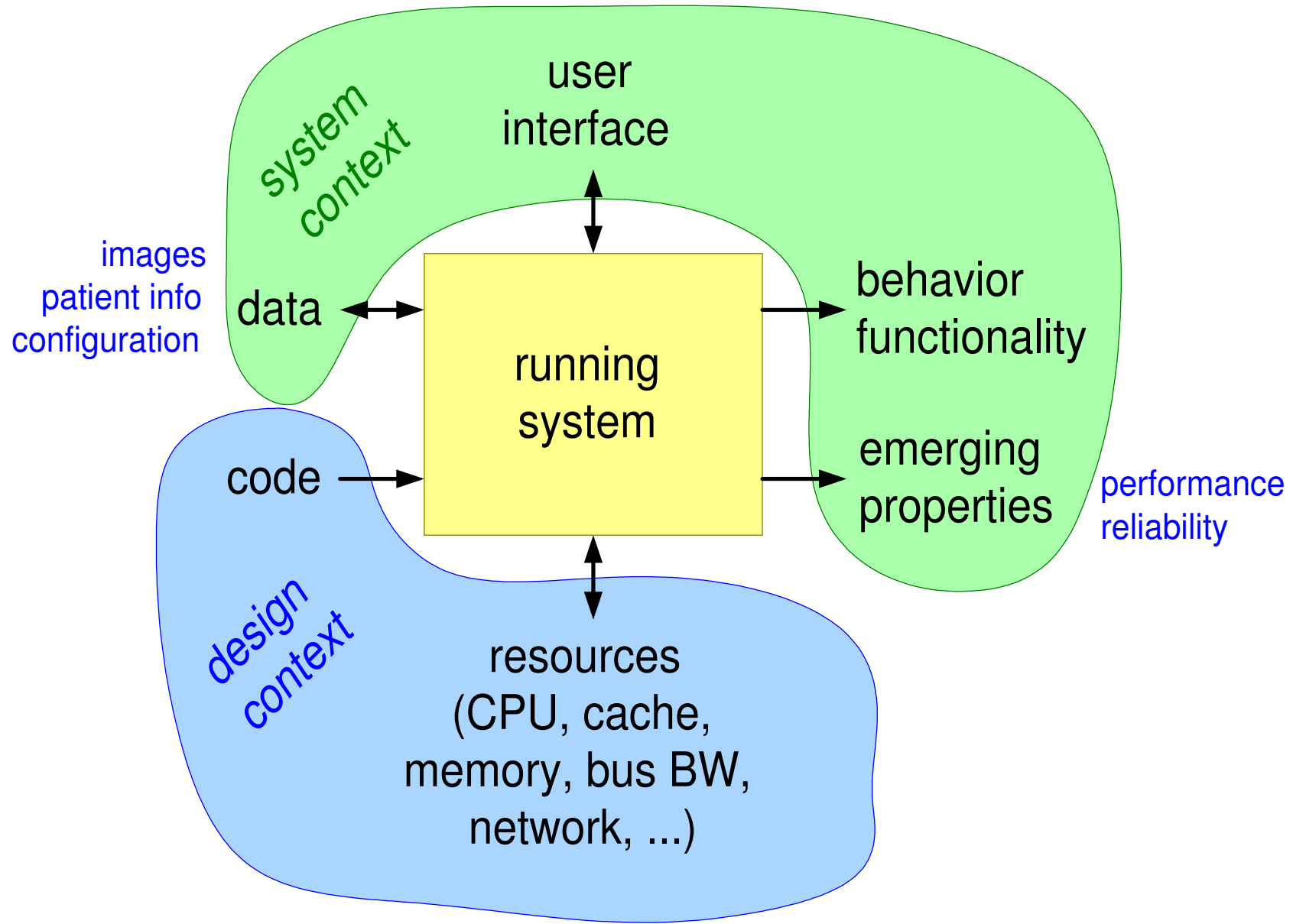
---

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

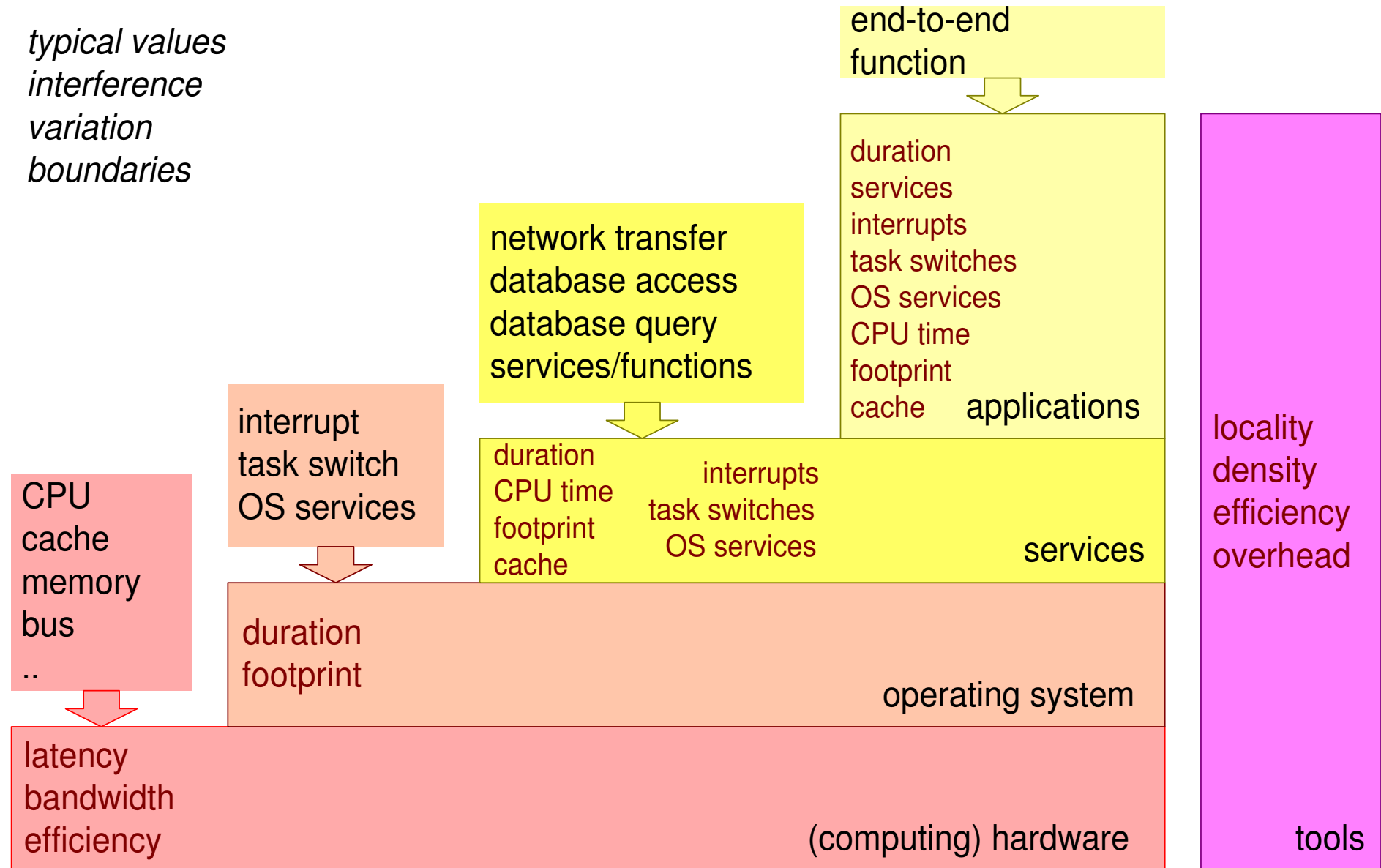
*Macroscopic* numbers related to *code level* understanding provides insight

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

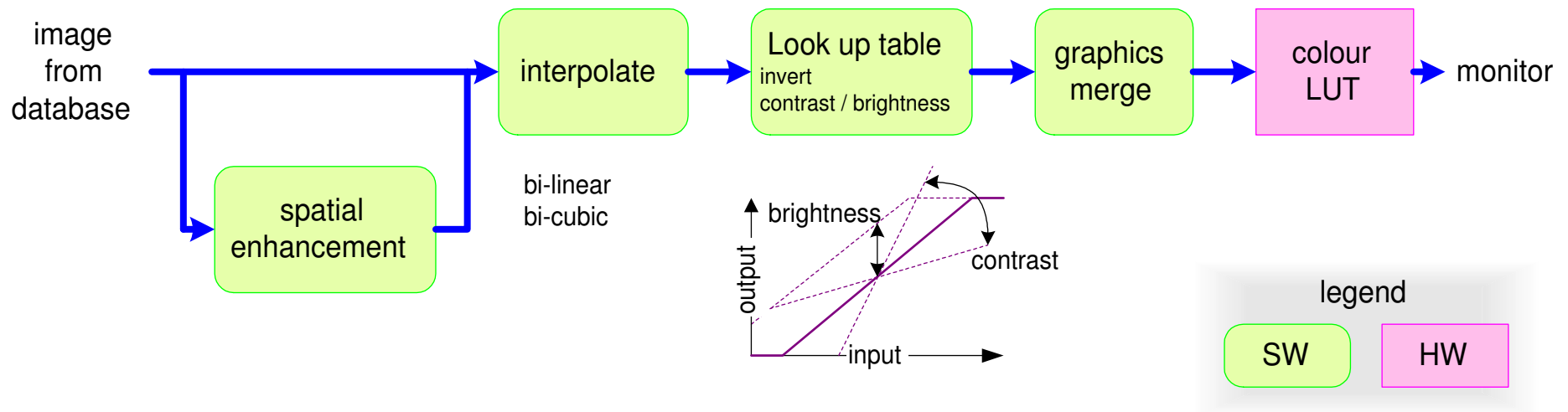
# Dynamics $\gg$ Static



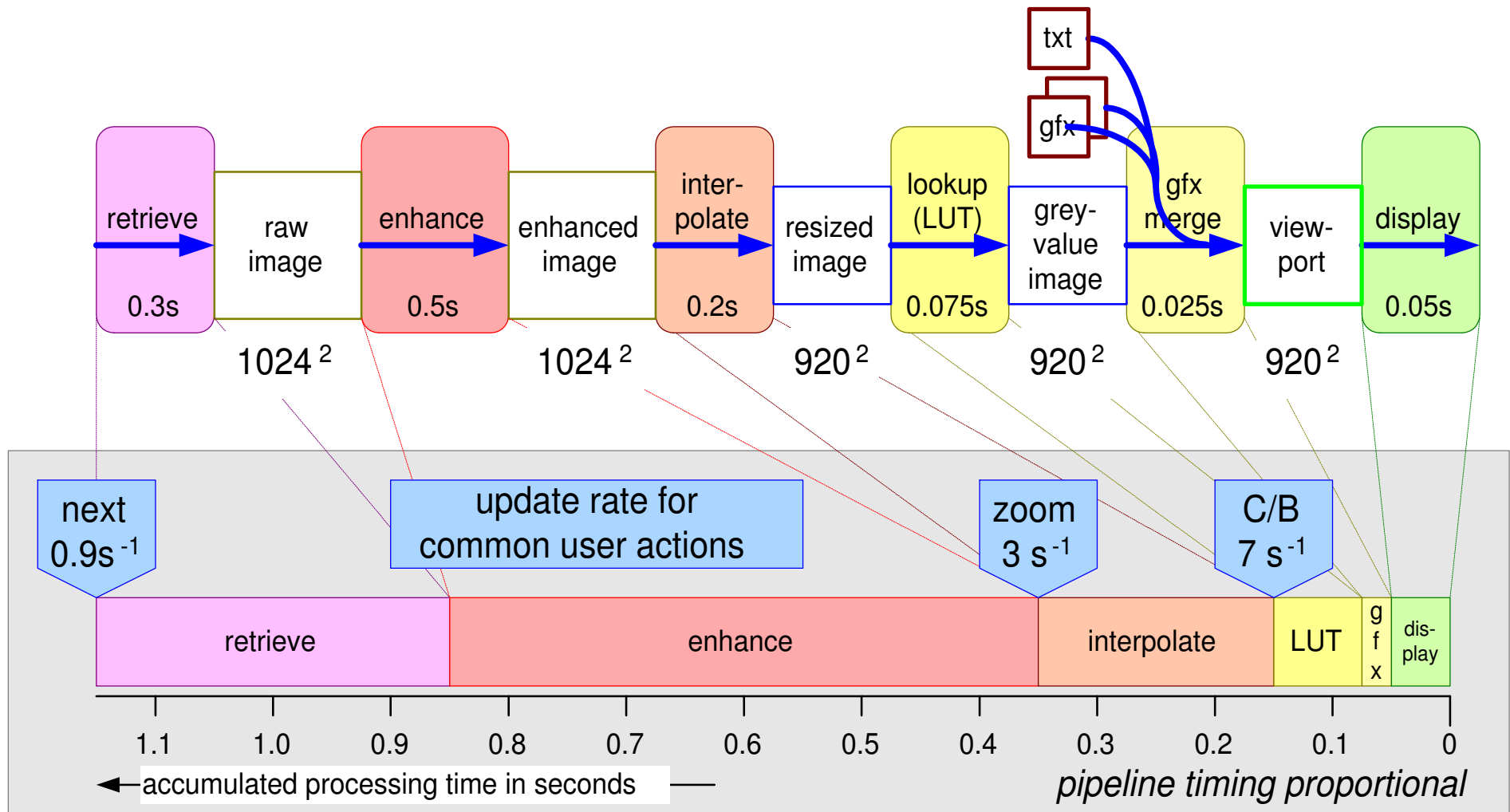
# Layered Benchmarking



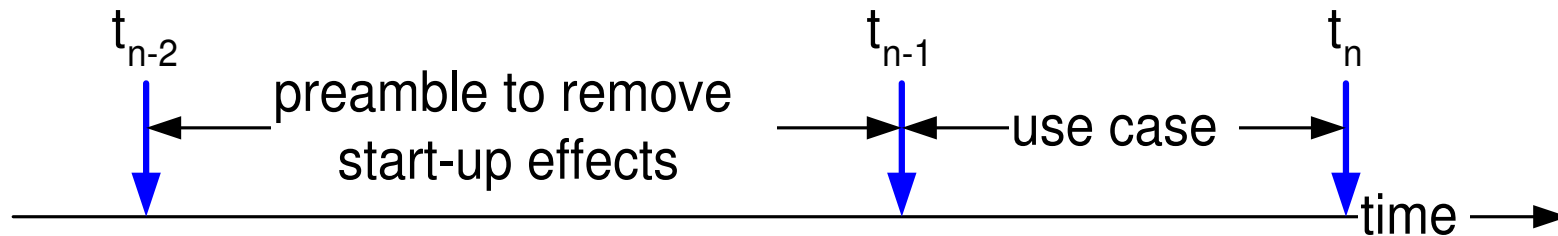
# Example: Processing HW and Service Performance



# Processing Performance



# Resource Measurement Tools



oit

△ object instantiations  
heap memory usage

ps  
vmstat  
kernel resource  
stats

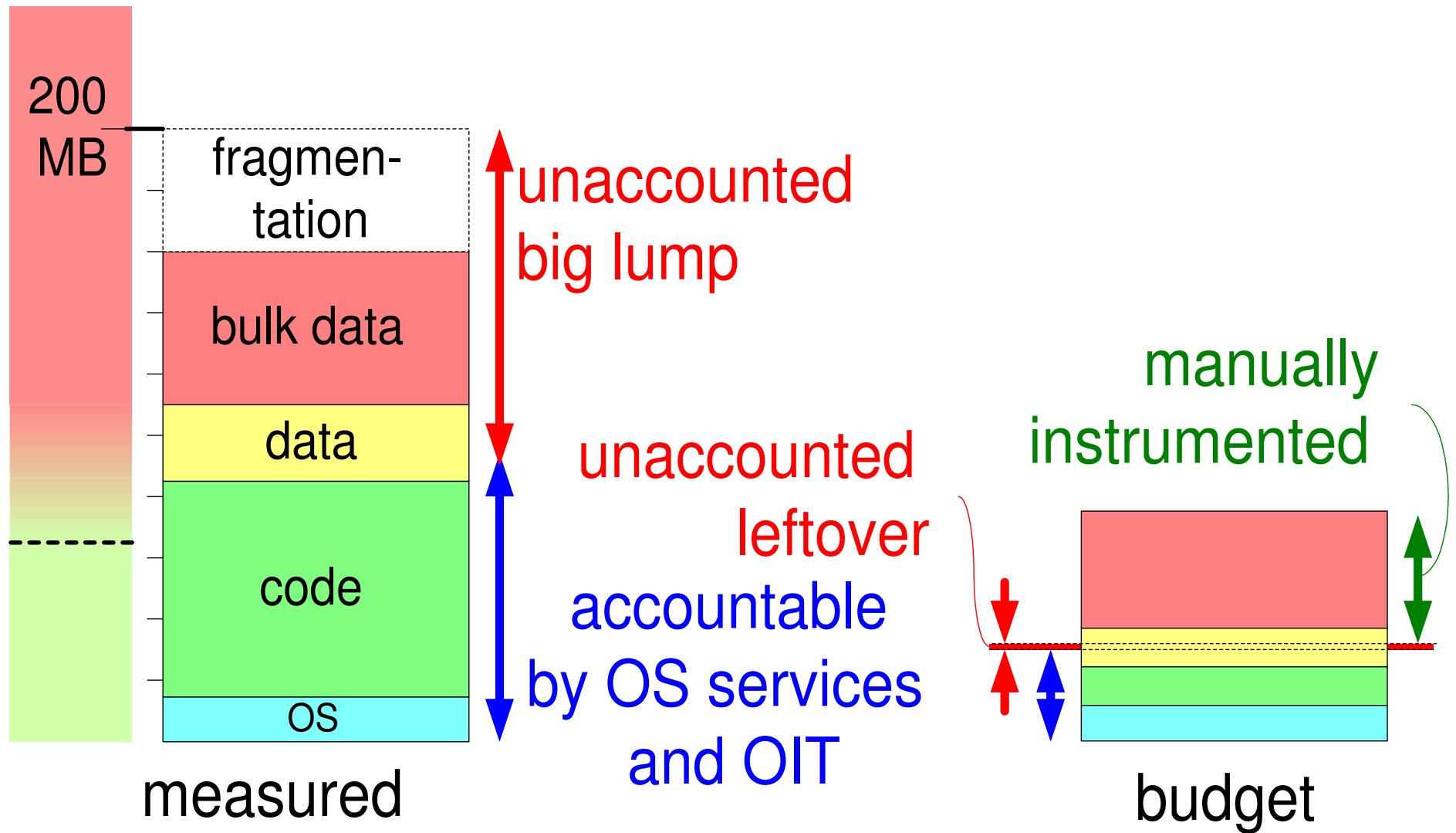
kernel CPU time  
user CPU time  
code memory  
virtual memory  
paging

heapviewer (visualise fragmentation)

# Object Instantiation Tracing

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]

# Memory Instrumentation



# Overview of Benchmarks and Other Measurement Tools

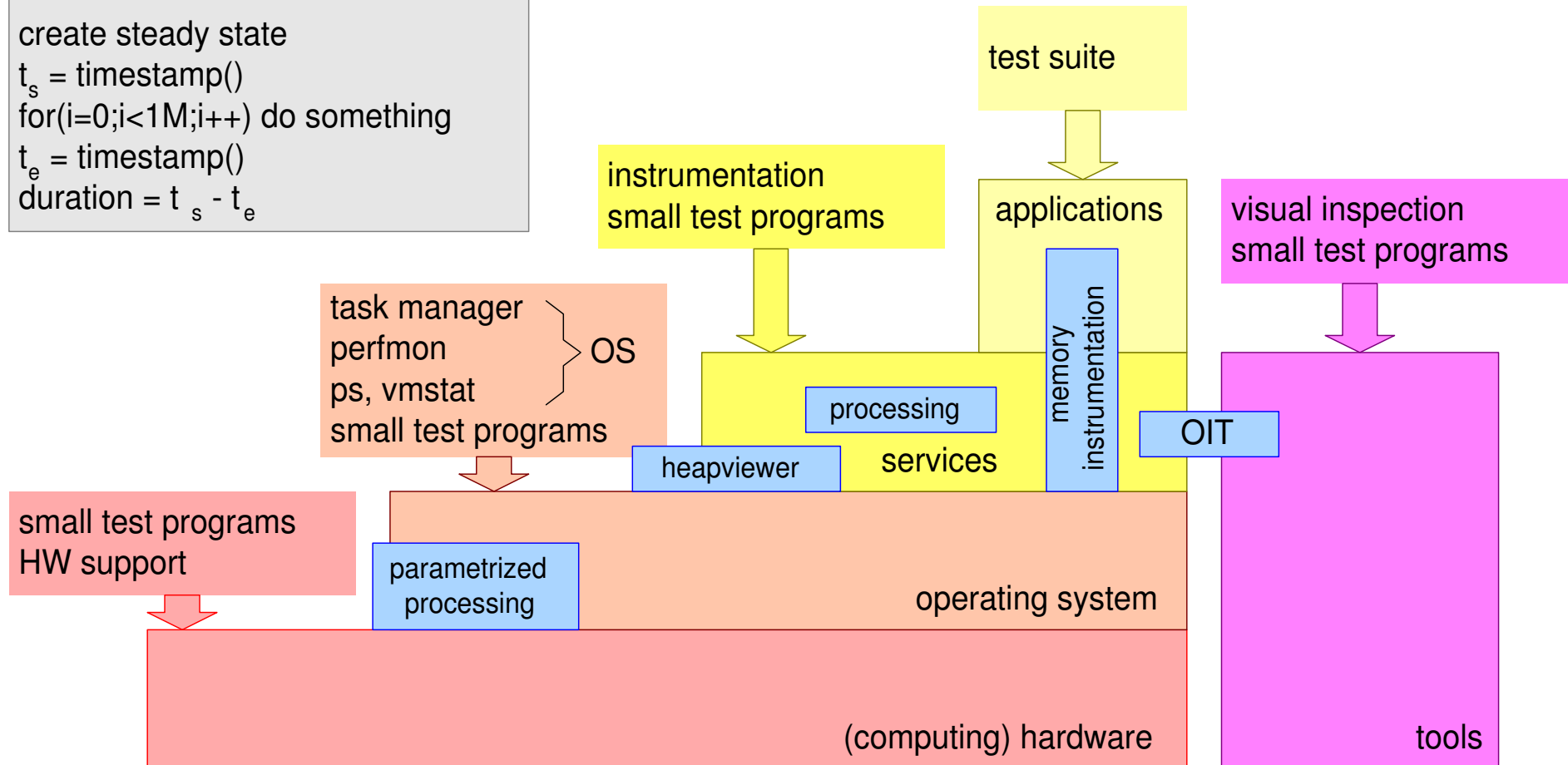
	test / benchmark	what, why	accuracy	when
<i>public</i>	SpecInt (by suppliers)	CPU integer	coarse	new hardware
	Byte benchmark	computer platform performance OS, shell, file I/O	coarse	new hardware new OS release
<i>self made</i>	file I/O	file I/O throughput	medium	new hardware
	image processing	CPU, cache, 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

# Tools and Instruments Positioned in the Stack

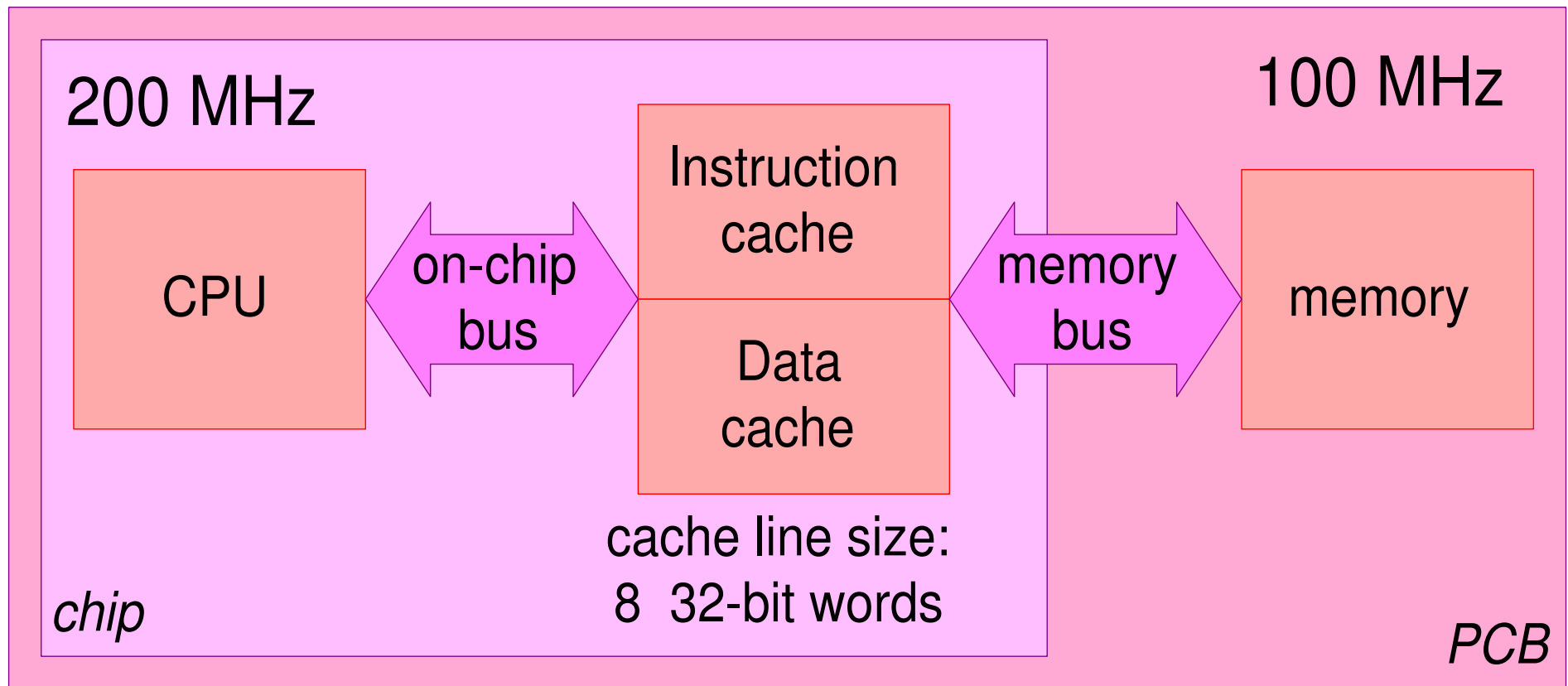
*typical small testprogram*

```

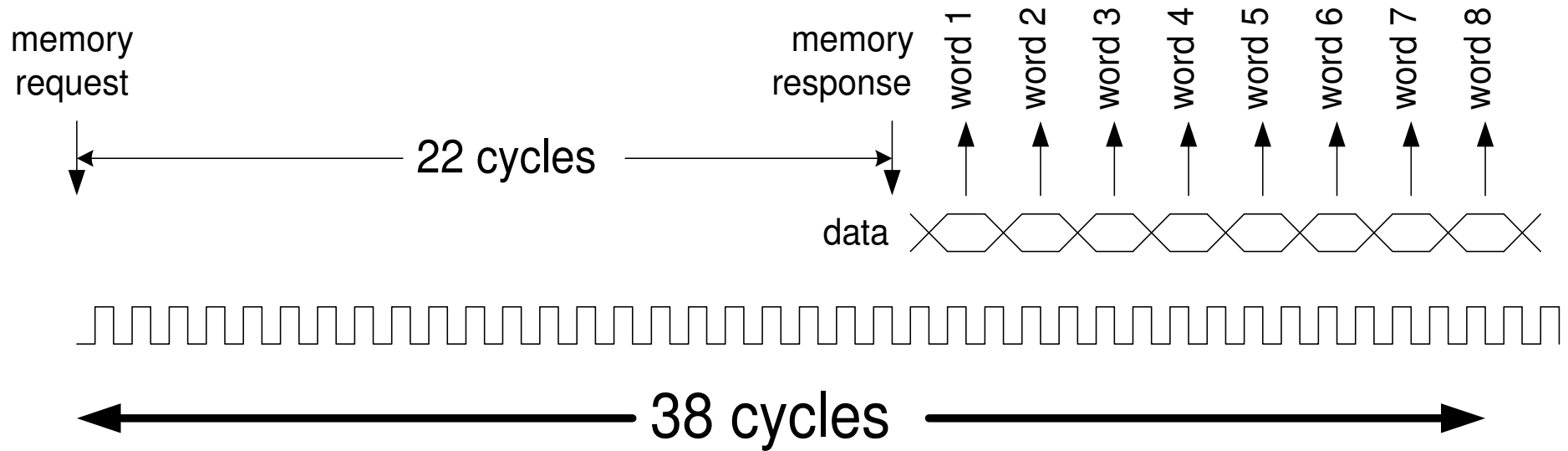
create steady state
ts = timestamp()
for(i=0;i<1M;i++) do something
te = timestamp()
duration = ts - te
    
```



# Case 2: ARM9 Cache Performance



# Example Hardware Performance



memory access time in case of a cache miss  
200 Mhz, 5 ns cycle: 190 ns

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

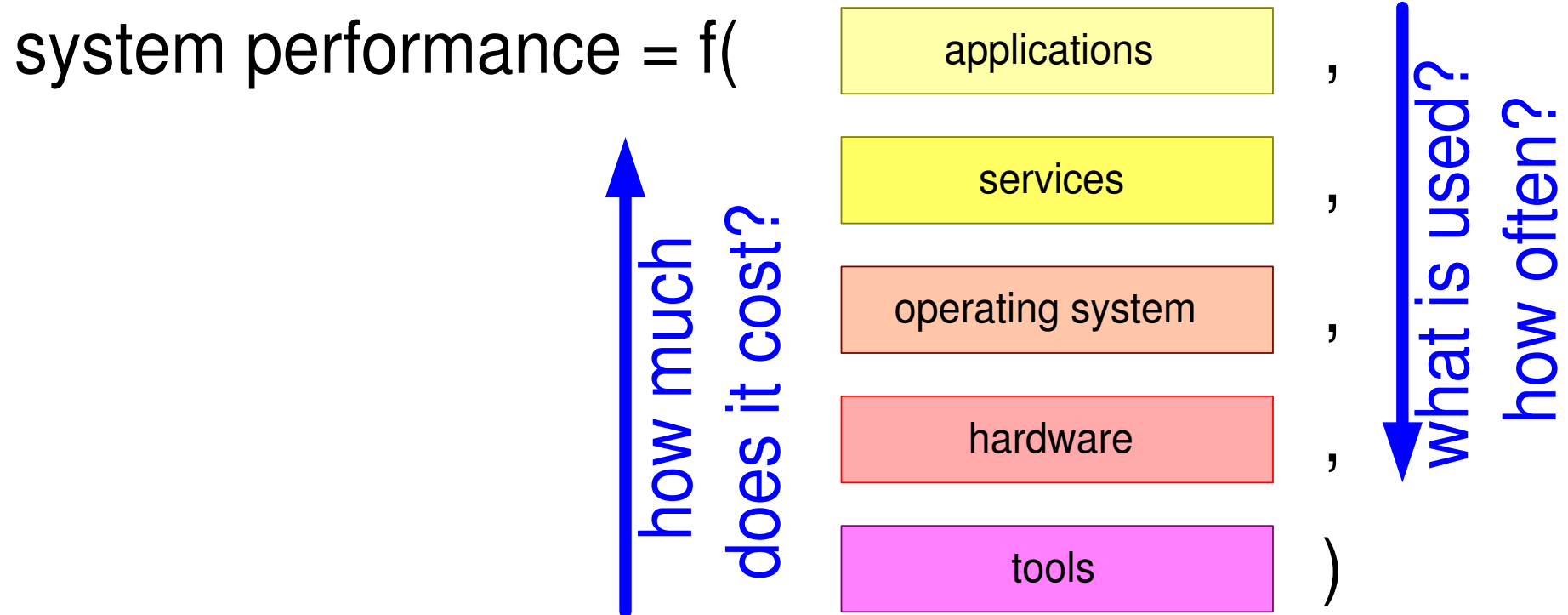
cache setting	$t_{\text{context switch}}$
From cache	2 $\mu\text{s}$
After cache flush	10 $\mu\text{s}$
Cache disabled	50 $\mu\text{s}$

# Context Switch Overhead

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

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

# Performance as Function of all Layers



# Annotated Performance Formule

system performance = f(

applications

hit-rate, miss-rate,  
#transactions  
interrupt-rate, task switch rate  
CPU-load

,

services

transaction overhead: 25 ms

,

operating system

interrupt latency: 10 us  
task-switch: 10 us  
(with cache flush)

,

hardware

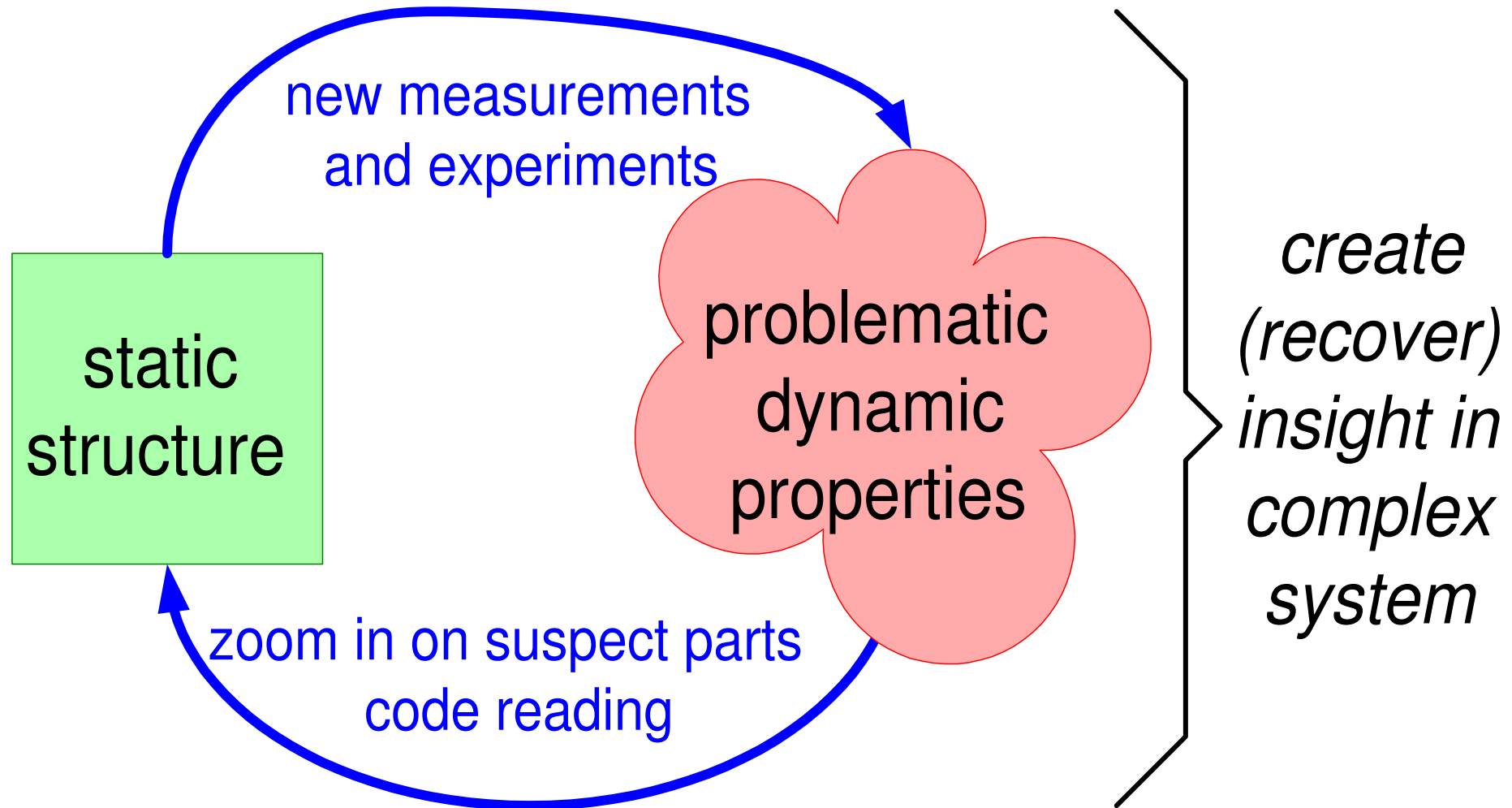
cache miss: 190ns

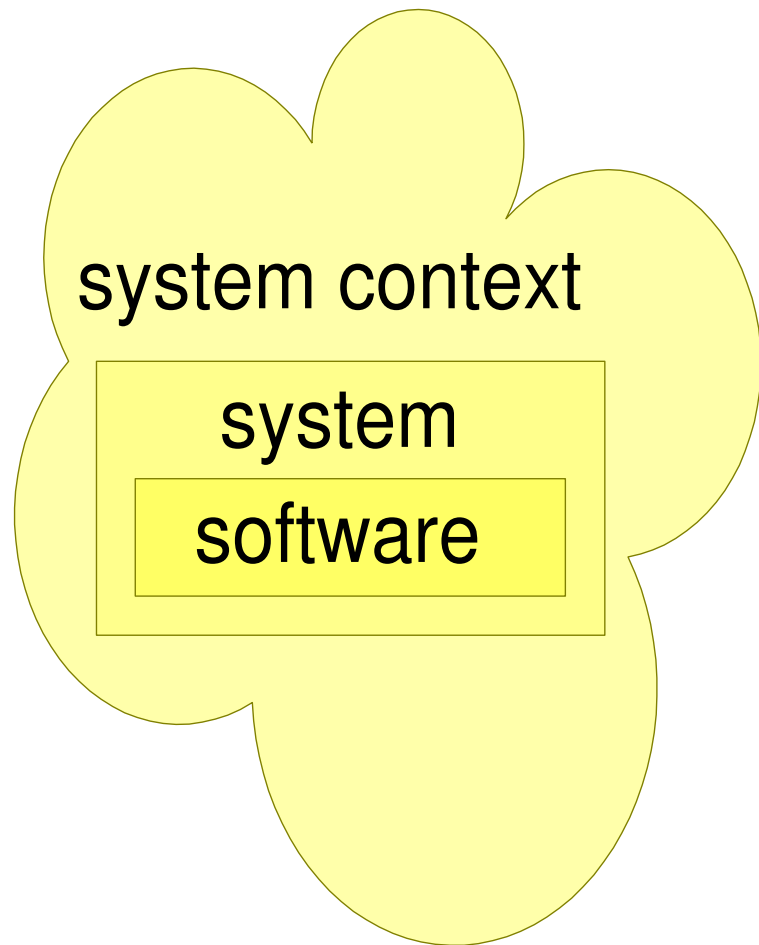
,

tools

)

# Keep iterating!





0. many design teams have lost the overview of the system
1. a good (sw) architect has a quantified understanding of system context, system and software
2. a good design facilitates measurements of critical aspects for a small realization effort