

# Characterization of COTS Microkernel-based Systems using MAFALDA

*Jean-Charles Fabre*

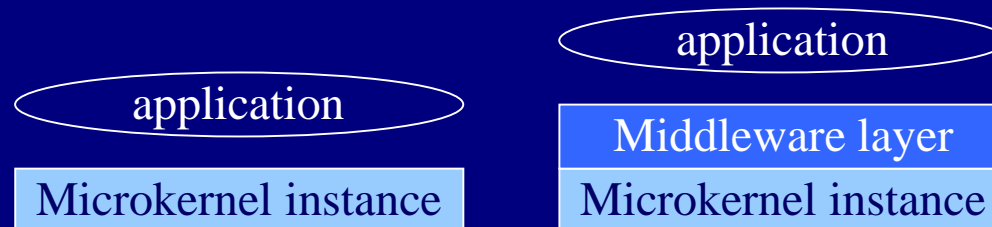


LAAS-CNRS  
Toulouse, France

*IFIP Working Group 10.4 Meeting — Paraty, Brazil — March 1-3, 2001*

# Problem statement

- **Building executive supports for dependable systems, two options:**
  - Development from scratch is complex & expensive
  - Use of commercial components is questionable
- **Main tendency for embedded systems**
  - Use of COTS componentized microkernels
  - Define a specific instance for the application
  - System development : two options



# Outline

- **The objectives of MAFALDA**
- **MAFALDA in action**
- **Experimental results**
- **Lessons learnt**

# Objectives of MAFALDA

- **Characterization by SWIFI**

*(S/W Implemented Fault Injection)*

- Identification of failure modes
- Evaluation of error detection coverage
- Identification of propagation channels
- Assessment of interface robustness

- **Wrapping framework**

- Definition of formal wrappers
- Definition of a reflective implementation framework
- Application to both white-box & black-box candidates

- **Evaluation of the wrapped microkernel instance**

*MAFALDA  
Microkernel  
Assessment by  
Fault injection  
AnaLysis and  
Design  
Aid*

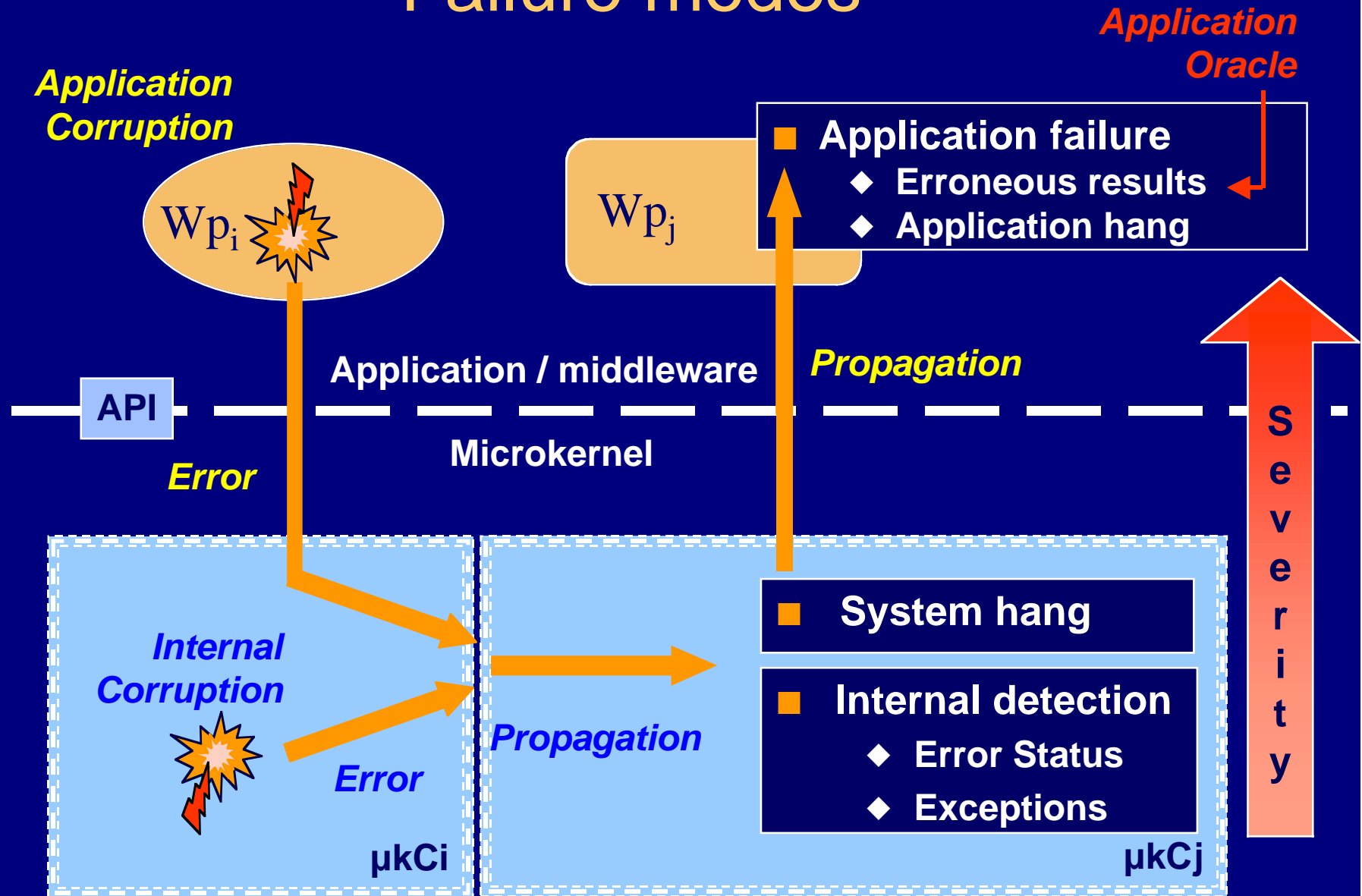


*Rack of target machines*



*Host Machine  
controlling the  
experiments*

# Failure modes



# Fault injection experiment

The screenshot displays a software interface for a fault injection experiment. At the top, there are several tabs: Error Propagation, Latency, Exception, Error Status, Confidence Interval, Global, Oracle, Fault Injection Experiments, Results, Failure Modes, and Wrapping. The 'Fault Injection Experiments' tab is active.

Under 'Fault Injection Experiments', the 'Experiment #' is set to 750. To the right, there are three progress bars: 'Rebooting' (red), 'Launching Processes' (white), and 'Monitoring' (white). The 'Monitoring' bar is labeled with '#3000', '60s', and '40s'.

A red information icon is followed by a blue bar containing the text 'Fault impact: APPLICATION FAILURE'.

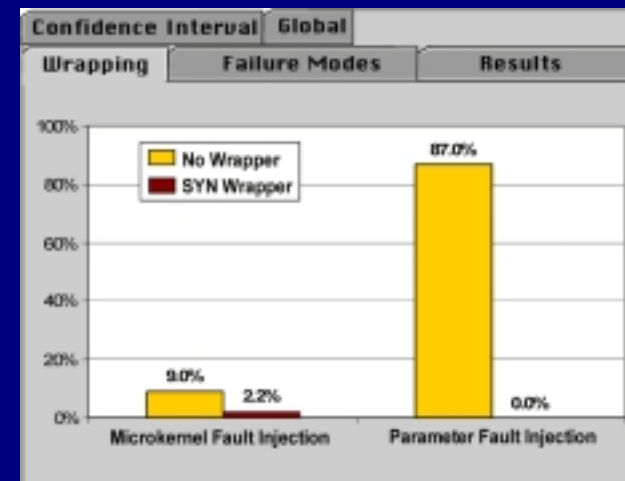
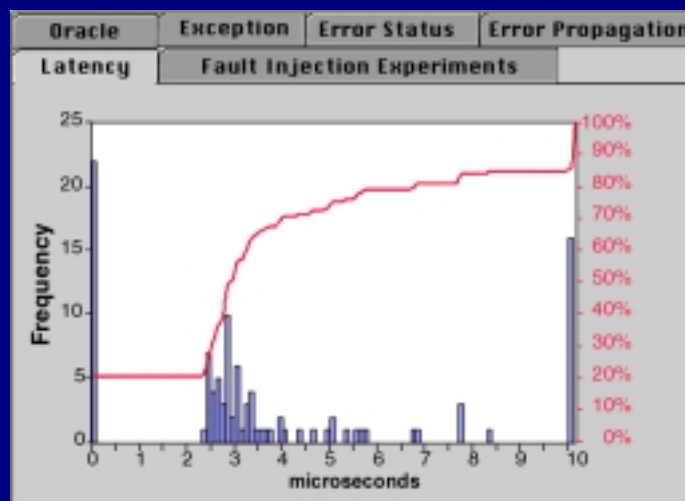
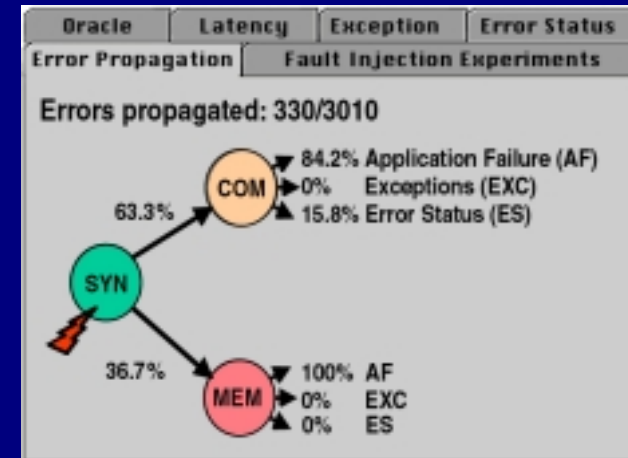
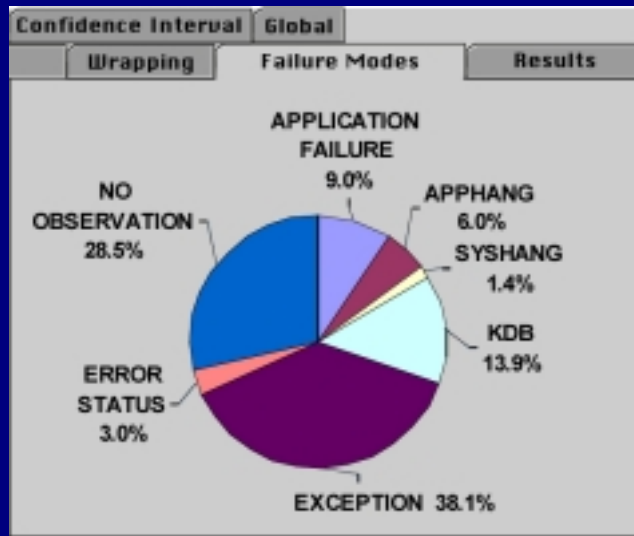
The 'Console' section shows the following text:  
[c042d32e:c042d32e:506:1978:\_uSemUnblock:TEXT]: 3e XOR 1 -> 3f  
Time trigger has occurred  
The fault has been activated (breakpoint0)  
Fault removed (Single Step Called)

The 'Outputs' section contains a list of system events:  
th2\_exit(0) SENDER\_STEP3\_OK RECEIVER\_STEP3\_OK th1\_create(th2) th2\_threadScheduler(prio-->) MEM\_BLOCK\_1\_...  
SENDER\_STEP1\_OK MEM\_BLOCK\_2\_OK th1\_doWork(A) th1\_delay(20) th3\_doWork(B) th3\_semV(s1)  
th3\_semP(s2) th4\_doWork(A) th4\_semP(s1) th2\_doWork(B) RECEIVER\_STEP1\_OK SENDER\_STEP2\_OK  
RECEIVER\_STEP2\_OK th1\_threadScheduler(prio-->) th2\_delay(30) SENDER\_STEP3\_OK RECEIVER\_STEP3\_OK th4\_dov...  
th4\_semV(s2) th4\_semV(s1) th4\_threadDelay(0) th1\_doWork(C) th2\_doWork(D) th2\_exit(0)  
MEM\_BLOCK\_3\_OK

Below the console, there are four panels for filtering outputs: SCH, SYN, MEM, and COM. Each panel has a list of events with a vertical scrollbar. The 'SYN' panel has 'th3\_semP(s1)' highlighted in red.

SCH	SYN	MEM	COM
th2_doWork(B)	th4_doWork(A)	MEM_BLOCK_2_OK	SENDER_STEP2_OK
th1_threadScheduler(prio-->)	th4_semP(s1)	MEM_BLOCK_3_OK	RECEIVER_STEP2_OK
th2_delay(30)	th3_doWork(B)	MEM_BLOCK_1_OK	SENDER_STEP3_OK
th1_doWork(C)	th3_semV(s1)	MEM_BLOCK_2_OK	RECEIVER_STEP3_OK
th2_doWork(D)	th3_semP(s2)	MEM_BLOCK_3_OK	SENDER_STEP1_OK
th2_exit(0)	th3_threadDelay(0)		RECEIVER_STEP1_OK
			SENDER_STEP2_OK

# Sample of measures



# Campaigns

- **Microkernels**

- Candidates:
  - ◆ Chorus Classix r3.1 (Kernel API),
  - ◆ Lynx OS v 3.0.1 (Kernel/Posix API)
- Components:
  - ◆ Synchronisation (*semaphores*)
  - ◆ Memory (*protected regions*)
  - ◆ Communication (*message passing*)
  - ◆ Scheduling (*preemptive FIFO*)

- **Campaign parameters**

- Same workload mapped on two different APIs
- Running on the same Pentium-based platform
- Between 1000 to 3000 experiments for each component
- All components targeted
- Both microkernel and parameters fault injection experiments

# Chorus vs. LynxOS

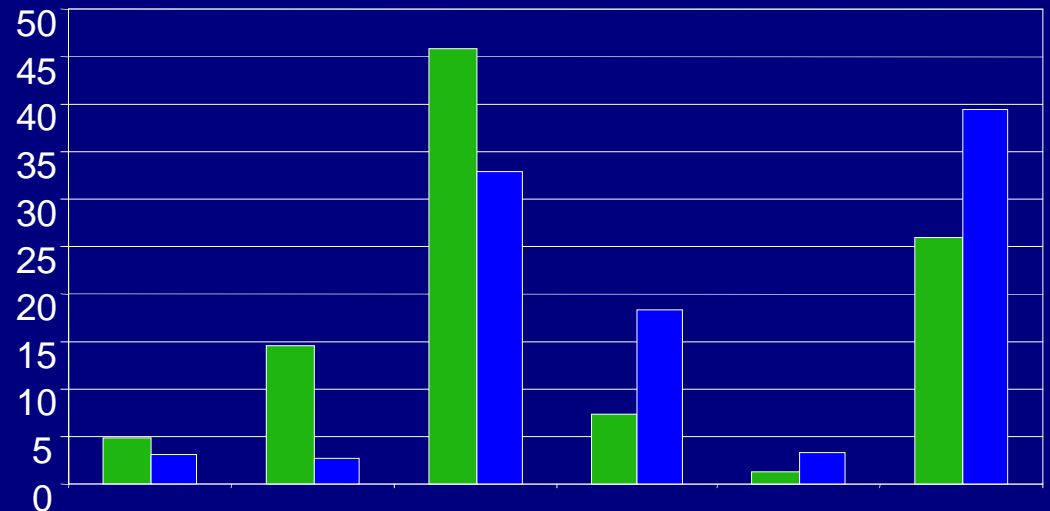
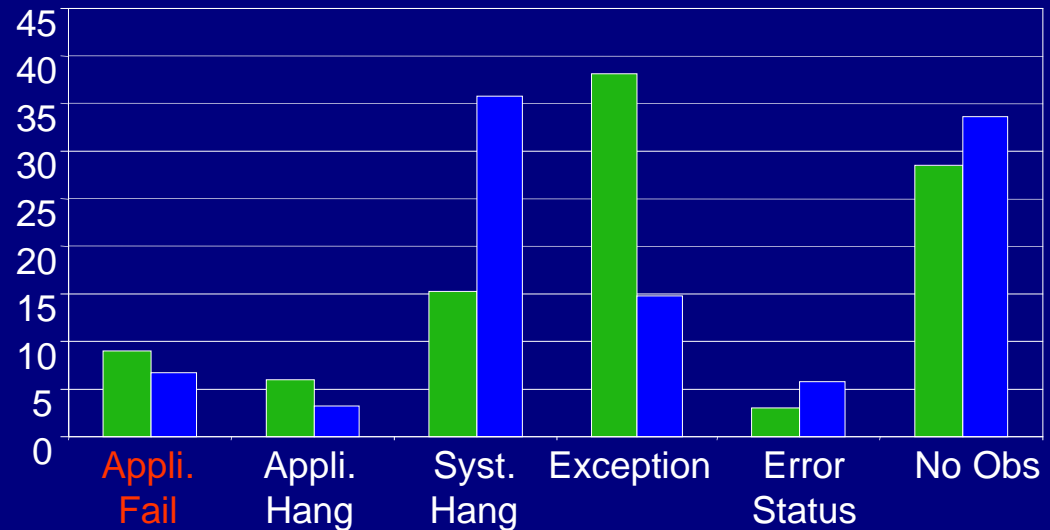
## *Code segment fault injection*

■ Chorus Classix r3.1  
■ LynxOS r 3.0.1

Synchronisation

Kernel

Scheduling



# Chorus vs. LynxOS

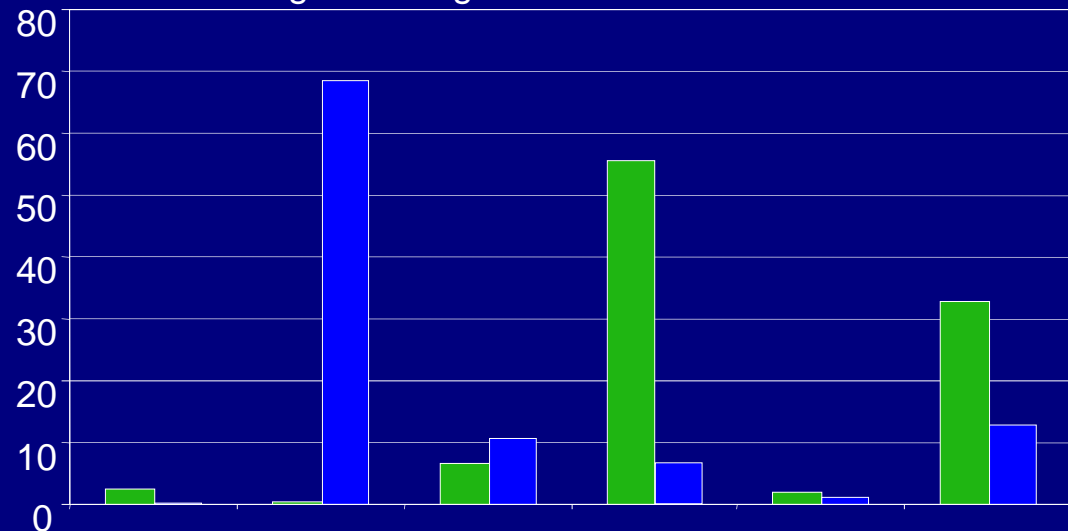
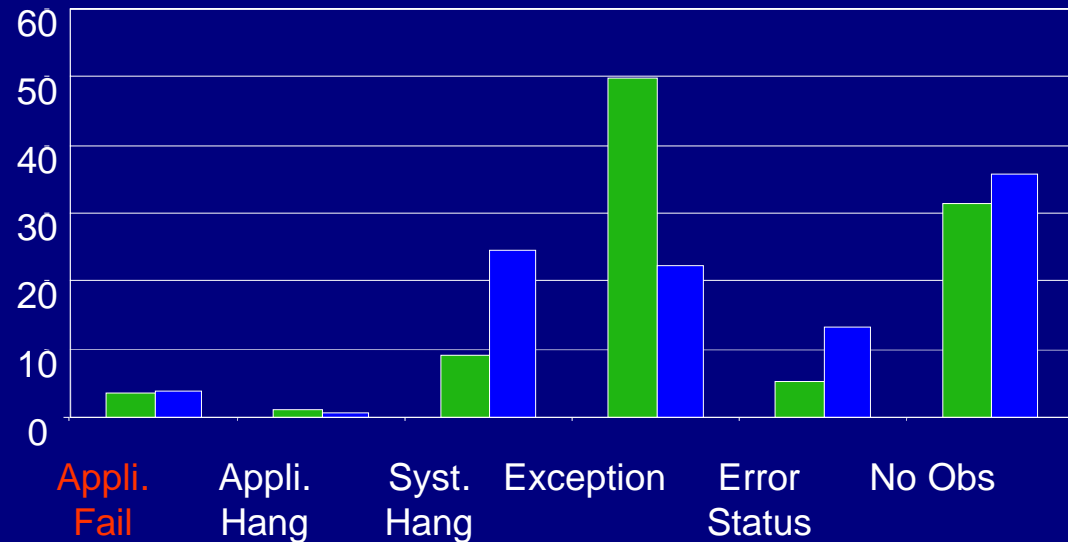
## *Code segment fault injection*

■ Chorus Classix r3.1  
■ LynxOS r 3.0.1

Communication

Kernel

Memory

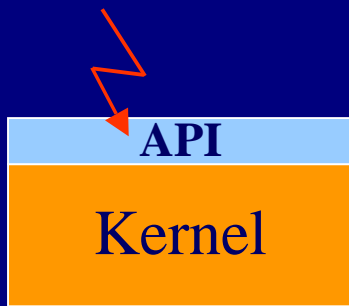


# Chorus vs. LynxOS

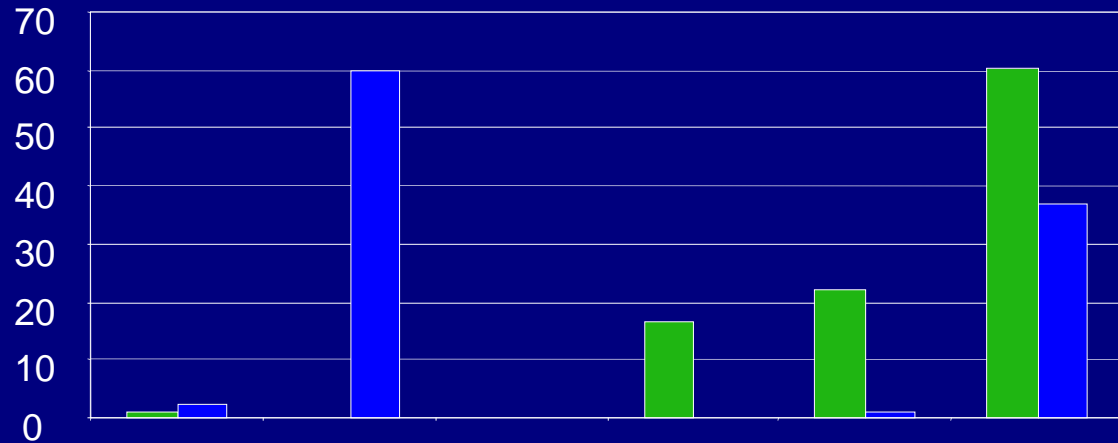
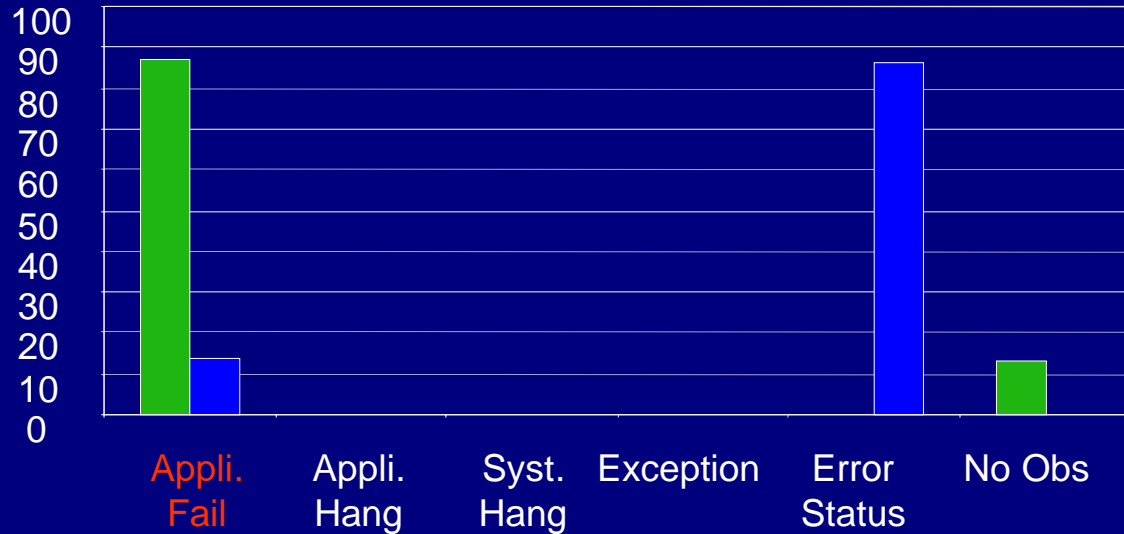
## *Parameter fault injection*

■ Chorus Classix r3.1  
■ LynxOS r 3.0.1

Synchronisation



Memory

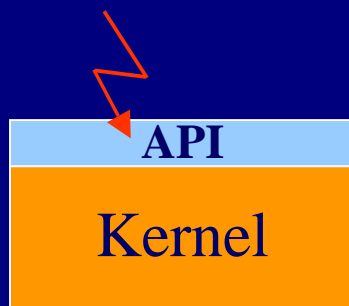


# Chorus vs. LynxOS

*Parameter fault injection*

■ Chorus Classix r3.1  
■ LynxOS r 3.0.1

Communication



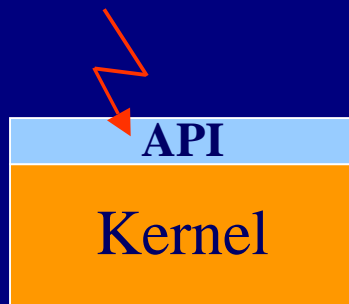
*Similar behavior, except that a system call with given parameters can hang the application or even the kernel*

# Chorus vs. LynxOS

## *Parameter fault injection*

■ Chorus Classix r3.1  
■ LynxOS r 3.0.1

Communication



- int portMigrate (options, **srcactorcap**, portli, dstactorcap, seqnum)
- int portDelete (**actorcap**, portli)

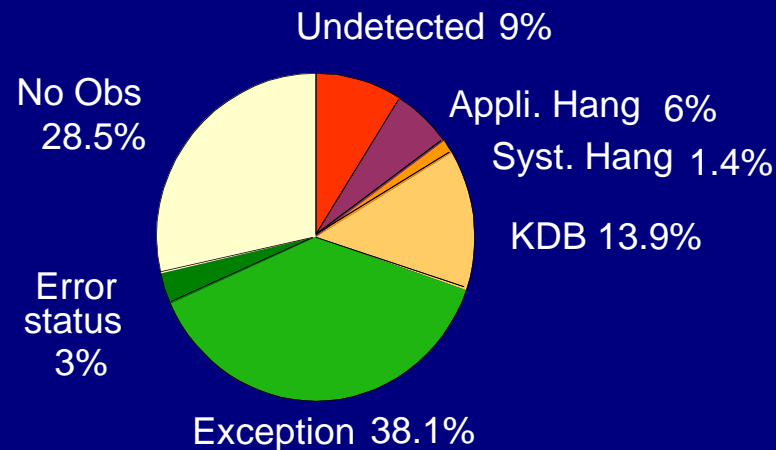
**KnCap**

# Running mode impact

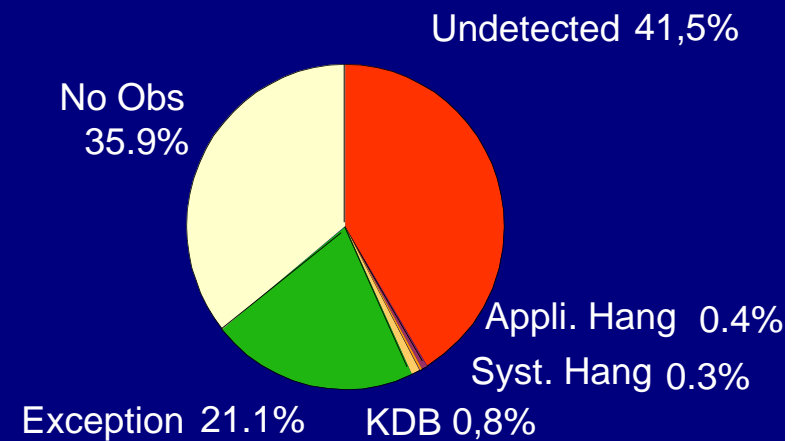
*Downloading application code into kernel space*

*(Synchronisation workload)*

User mode



Kernel mode



*Code segment fault injection  
experiments carried out on Chorus Classix r 3.1*

# Detailed system call analysis

Kernel call	GetPriority				SetPriority							
Parameter number	1		2		1		2		3		Total	
Type	int	which	int	pid	int	which	int	pid	int	prio		
Activated faults	214		213		138		180		116		861	
Application failure	0	0.0%	17	8.0%	0	0.0%	4	2.2%	15	12.9%	36	4.2%
Application hang	0	0.0%	0	0.0%	0	0.0%	3	1.7%	0	0.0%	3	0.3%
Exception	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
Error status	214	100%	64	30.0%	138	00%	144	80.0%	92	79.3%	652	75.7%
No observation	0	0.0%	132	62.0%	0	0.0%	29	16.1%	9	7.8%	170	19.7%

- Most of individual cases can be analysed – Examples:
  - ◆ Priority out-of-bounds (*Error status*)
  - ◆ Invalid priority (*Application failure*)
- Possible conclusions:
  - ◆ The corrupted input value can be detected (assertion missing)
  - ◆ The corrupted input is valid for the kernel and cannot be checked (to be checked at the application/middleware level)
- The input space is randomly corrupted (sometimes all bits)

# Some Lessons learnt

- **Interpretation of results**
  - ◆ One campaign : a microkernel instance + an activation profile
  - ◆ Variability of results:
    - stand-alone vs. Posix-based version
    - reactive vs. static application
- **Raw data analysis**
  - ◆ Analysis of logged data ⇒ precise analysis of faulty situations
  - ◆ User-defined semantics of the failure modes
- **Integrator's vs. supplier's viewpoint**
  - ◆ Integrator: weaknesses revealed ⇒ ED mechanisms (wrappers)
  - ◆ Supplier: bugs not yet revealed ⇒ product improvement
- **Target system evolution**
  - ◆ A slightly new instance ⇒ new campaign needed
  - ◆ Is the new release/version acceptable?