

# Measuring Performance in Real-Time Linux

### Frederick M. Proctor Group Leader, Control Systems Group National Institute of Standards and Technology









## **Performance Measures**

- *Performance measures* are figures of merit that indicate how well a system behaves
- *Benchmarks* can provide performance measures for specific areas of interest, e.g.,
  - SPEC CPU2000 measures performance of processor, memory, compiler
  - SPEC WEB99 measures performance of web servers
  - x11perf measures performance of X servers

	<b>LPublish</b> Edit Vier			b <mark>99 Res</mark> nunicator	ults - Ne Help	etscape			-		×	
▶ 100000		eZka				D	14	00				
Ρι	idiist	iea	SPI	LCW	еруу	Resu	its (1	00)	1:			
Te	ster Na	me	Syster	n Nam	e HTT	I <mark>P Versio</mark>	n CPU	# F	lesu	ılt İ		
	vanced M vices	licro	Tyan Ti K7/Ath 1800+			IIS	5.0	2	1.	510		
Cor	npaq		Alphas	erver DS	10	Zeus 3.	1.9	1		484		
Cor	npaq		Alphas 6/667	erver DS	20	Zeus 3.	3.5	2	10	050		
Co		olishe				Netscape					_ 🗆	x
Col	_ <u>F</u> ile _Edit ▶	<u>v</u> iew /k		ommunicato	or <u>H</u> elp							
	CINT	2000	(212):	:							-	Ê
Co	Compa	ny N	ame	System	n Name			#C	PUE	Sase ]	Peak.	
Co	Advance	ed Mi	ro Devic	ASUS A	7V Mothe	rboard 1.2GH	Iz Athlon p	)	1	409	458	
Co	Advance	ed Mi	ro Devic	Gigabyte	GA-7DX	Motherboar	d 1.2GHz A		1	443	496	
~	Advance	ed Mi	ro Devic	ASUS A	7V Mothe	rboard, 1.3G	Hz Athlon		1	438	491	
Co	Advance	ed Mi	ro Devic	Gigabyte	GA-7DX	Motherboar	d, 1.33GHz		1	482	539	
Co	Advance	ed Mi	ro Devic	Gigabyte	GA-7DX	Motherboar	d, 1.4GHz		1	495	554	
• 	Advance	ed Mi	ro Devic	Tyan Th	under K7	Motherboard	<b>1</b> , 1.2GHz		1	495	522	
	Advance	ed Mi	ro Devic	Epox 8KI	HA+Moti	herboard, AN	/ID Athlon	(	1	633	656	
	Advance	ed Mi	ro Devic	Epox 8KI	HA+Moti	herboard, AN	/ID Athlon	(	1	648	671	
	Advance	ed Mi	ro Devic	Epox 8KI	HA+Moti	herboard, AN	/ID Athlon	(	1	677	701	
	Advance	ed Mi	ro Devic	Gigabyte	GA-7DX	Motherboar	d, AMD At	th	1	556	577	<b>–</b> 1
	•			-					J		₽	-
	<b>-</b>			Docum	ent: Done		🔆 🖉	6	P (	<u>a</u> .	<u>/</u>	11



😰 –🖂 kpm@rebozo <u>W</u>indow

cpu 🛛

15

0

0 0 0 0

0 0 0

0

0 0 0 0

0 0

0

1

9

18730 proctor

1 root

2 root

3 root

4 root

5 root

326 bin

340 root

705 proctor\* 709 proctor

Process Signal View Options

0.63620.63620

0 17772 16896

0 3212 2380

88

68

0

0

0 120

18729 proctor 18 0 4340 4340 3192 R

0 0 0

0 0 0

- Monitors show general resource use of program in a system, e.g.,
  - ps, top and its graphical front ends
  - Windows Task Manager

Help

1.38 3.39

1.18 49.67

6.74

1.85 0.20

0.59

0.00 0.05

0.00 0.00

0.00 0.00

0.00 0.00

0.00 0.00

0.00 0.00

0.00 0.00

🔳 user 🔲 nice 🗌 sys 🦳 idle 📕 used 🔲 buff 🗌 cach 🗌 free

PID USER PRINCE SIZE RSS SHARE STAT %CPU %MEM TIME CMDLINE

44852

1352 S

1712 S

48 S

0 SW

0 SW

0 SW

0 SW

0 SW

0 SW

swap

0:10 kpm

0:10 kwm

3.70s init [3]

0.46s (kflushd)

0.00s (kpiod)

1.51s (kswapd)

0.00s (portmap)

0.00s (lockd)

0.23s (kupdate)

up 3 davs, 0:06

0.24 /opt/staroffice6.0/p

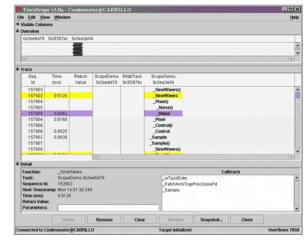
8:21 /etc/X11/X :0 - auth

	Eile Options View	<u>H</u> elp						
				Windows NT T	ask Manager			
	- CPU Usage	CPU Usage	History <u>F</u>	jile <u>O</u> ptions ⊻iew	<u>H</u> elp			
				Applications Proc	······································	<u> </u>		
	19%	A-A-A	A J LL March	Image Name	PID	CPU		Mem Usage 🔺
a	- MEM Usage	⊢ ⊢Memory Usa	ae Historu	System Idle Pro System	cess 0 2	99 00	16:27:41 0:00:46	16 K 220 K
S	MEM Osage	Memory 030	gernstory	SMSS.EXE	20	00	0:00:00	396 K
				CSRSS.EXE	30	00	0:00:00	1768 K
				SERVICES.EXE		00 00	0:00:00 0:00:01	1712 K 3164 K
	I 14952K			LSASS.EXE	40	00	0:00:01	2800 K
				SPOOLSS.EXE	69	00	0:00:01	2500 K
	Totals		- Physical M	NETDDE.EXE	76	00	0:00:00	1496 K
	Handles	2531	Total	defwatch.exe rtvscan.exe	89 97	00 00	0:00:00 0:10:19	1236 K 8892 K
	Threads Processes	221 31	Available File Cache	LOCATOR.EXE	108	00	0:00:00	1484 K
				RPCSS.EXE	113	00	0:00:00	1252 K
	Commit Charge (K)		- Kernel Mer	PSTORES.EXE mstask.exe	121 124	00 00	0:00:00 0:00:00	120 K 2728 K
	Total	114952	Total	WinMgmt.exe	124	00	0:00:00	664 K
	Limit Peak	516704 161604	Paged Nonpaged	systray.exe	174	00	0:00:00	1236 K
	- Cax	101004	Wonpagea	NDDEAGNT.E>		00	0:00:00	1092 K
				explorer.exe	183	00	0:00:08	2092 K 💌
_	Processes: 31	J Usage: 19%	Mem Usa					End Process
. ×   ¤ xte	m		-		CPU Usage: 0%	I	Mem Usage: 10	1156K / 516704K
3:59pm	up 2 days, 6:49, 3 us sses: 63 sleeping, 2 runn	rs, load averag	e: 0.03, 0.10, 0.					
66 proce CPU stat	sses: 63 sleeping, 2 runn es: 0.7% user, 0.5% sys 28092K av, 124300K used,	ing, 1 zombie, 0 cem, 0.0% nice,	stopped 98.6% idle					
Hen: 1 Swan: 1	28092K av, 124300K used, 36512K av, 14860K used,	3792K free, 121652K free	38340K shrd, 2	0716K buff 0720K cached				
PID US 17831 pr	octor 10 0 1012 1012	780 R 0	0.7 0.7 0:00 1					
709 pm 1 no		1408 S 0 48 S 0	0.5 1.5 0:06					
2 ro 3 ro	ot 0 0 0 0	OSW O OSW O	0.0 0.0 0:00 1	(flushd (update				
4 ro	ot 0 0 0 0	0 SW 0	0.0 0.0 0:00 1	piod				
5 ro 326 bi	n 0 0 228 196	0 SW 0 168 S 0	0.0 0.1 0:00	kswapd portmap				
340 ro 341 ro		0 SW 0 0 SW 0	0.0 0.0 0:00					
350 ro 365 ro	ot 0 0 316 288	240 S 0 16 S 0		pc.statd				
370 ro	ot 0 0 504 496	428 S 0	0.0 0.3 0:00	pbind				
431 ro 440 ro		256 S 0 140 S 0	0.0 0.2 0:00	syslogd (logd				
440 ro 454 no		24 S 0						

### NIST • Manufacturing Engineering Laboratory • Intelligent Systems Division



- *Profilers* show details of program execution, e.g.,
  - the profil() function, gprof, strace
  - ParaSoft insure++, inuse
  - Rational Quantify, WindRiver WindView, RTI ScopeTools
  - the Linux Trace Toolkit
- None of these specifically address performance measures for real-time systems







For us, performance measures answer the question:

*How can I tell that a real-time operating system is able to satisfy my application's timing requirements?* 



## **RT Performance Measures**

- Real-time software must execute *on time* to be correct
- On time can mean:
  - any time between now and a deadline
  - within some interval around a target time
- For RT operating systems, performance measures should indicate how well the RTOS satisfies on-time demands
  - what is the shortest deadline by which the RTOS can guarantee a task's execution?
  - what is the smallest interval around a target time within which the RTOS can guarantee a task's execution?
  - how do these scale with task loading?

## **Classic RTOS Performance Measures**

- The shortest deadline measure applies to instances where an event initiates code that must run before a deadline
  - Typically the event is an interrupt, and the code is the interrupt service routine (ISR)
  - *Worst-case ISR latency* is the classic performance measure
- The smallest interval measure applies to instances where code must execute as close as possible to a target time
  - Typically the target time is one of a series of periodic timer expirations
  - *Scheduling jitter* is the classic performance measure



## **Types of Testing**

- *External testing* uses instrumentation not normally part of the RT system to stimulate and measure RT response
  - e.g., digital storage scopes, data acquisition systems
  - advantages: equipment is part of experiment's control; entire RT system is tested; can include arbitrary features, storage capacity, timing precision



- disadvantages: additional cost
- Internal testing uses native resources of the RT system
  - e.g., processor time stamp counters
  - advantages: no additional cost; tests can be incorporated into RT application for continuous monitoring or performance improvement
  - disadvantages: as with students grading their tests,
    "cheating" is possible; some effects will be invisible (e.g., clock chip jitter)





## **Testing Environment**

- If test results from different systems are to be compared, the testing environment must be adequately specified
  - what components must be present, e.g., network and video cards
  - what processes must be running; single v. multiuser mode
  - what optimizations are allowed or disallowed, e.g., disabling floating point support
- Hardware effects can be substantial, especially for generalpurpose processors
  - optimizations like the cache introducing timing uncertainty
  - hardware reference platforms are one answer to this problem, e.g., WinCE HARP



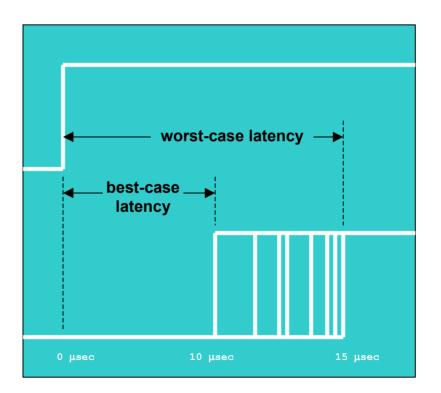
## **ISR Latency**

- *ISR latency* is the time between the occurrence of an interrupt and the execution of its service routine
  - "execution" is vague: time the ISR begins? completes?
  - maximum ISR latency is a system performance measure
- Latency contributors include:
  - hardware effects: processor must finish current instruction, and instruction lengths vary
  - software effects: interrupt masking and priority



## **External Latency Measurement**

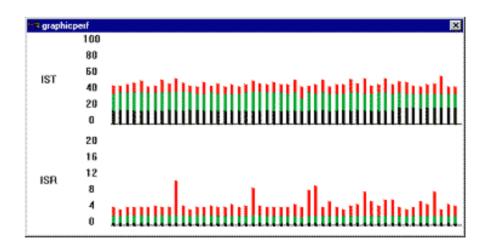
- An ISR is written that generates a measurable output, e.g., setting a parallel port bit high
- The interrupt is triggered repeatedly and the output is recorded on a digital storage oscilloscope in persistent display mode
- Pick latency off the display





## **Internal Latency Measurement**

- Use the programmable timer to down-count to zero from a start count and generate an interrupt
- The timer automatically reloads the start count and continues the down-counting
- The ISR is invoked and reads the timer
- The latency is the start count minus the reading
- WinCE "iltiming" tool does this





## **Scheduling Jitter**

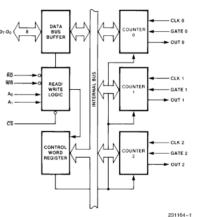
- *Scheduling jitter* is the variation in actual timing for a periodic task
- Jitter contributors include:
  - hardware effects: the cache
  - software effects: variation in branch instruction lengths in the scheduler
- External measurement technique:
  - a periodic task is written that generates a measurable output
  - the output timing can be analyzed with a hardware timing analyzer, e.g., LeCroy





# Internal Jitter Measurement

- See Phil Wilshire's 2nd RTLW paper, "Real-Time Linux: Testing and Evaluation"
- A single RT task is scheduled, which reads the Pentium Time Stamp Counter (TSC) and logs readings into RAM
  - the TSC is a 64-bit integer, incrementing once per clock cycle (2.5 nanosec resolution sector for a 400 megahertz clock)
- Pure periodic scheduling: 8254 Programmable Interval Timer (PIT) chip generates an interrupt, the RT scheduler is the interrupt service routine
- The TSC log is later analyzed for jitter
  - logged values should be exactly one interrupt time apart
  - variations in combined execution time of scheduler and task code will show up as deviations from the nominal







## **Interpreting Jitter**

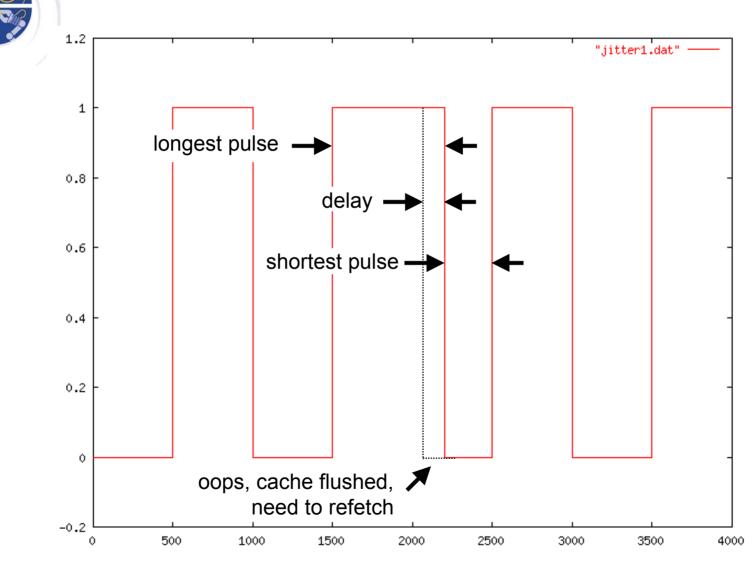
- If the TSC logging task were a square-wave pulse generator, then jitter would appear as variations in the pulse widths
- Two estimates of maximum jitter can be made
  - *cycle-to-cycle jitter*: difference between longest and shortest pulse
  - *period jitter*: largest difference between actual start/end of pulse and nominal expected
  - for the same TSC log, cycle-to-cycle jitter will be about twice the period jitter



## Cycle-to-Cycle Jitter

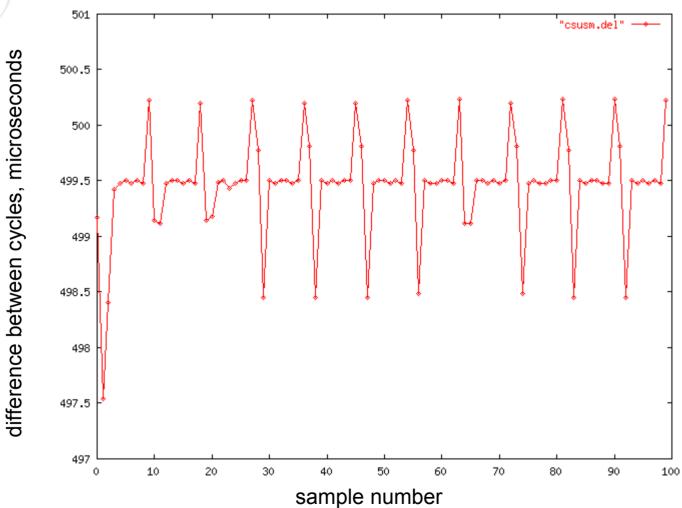
- Cycle-to-cycle jitter is calculated by differencing adjacent points in the TSC log to get the intervals, then taking the difference between the largest and smallest intervals
- With cycle-to-cycle jitter, a single late task invocation will lengthen one pulse, and shorten the following pulse
- This jitter value is effectively double the scheduling delay
- If relative task timing is important, as for a square wave pulse generator, the cycle-to-cycle jitter value is the most meaningful

### Cycle-to-Cycle Jitter





## Cycle-to-Cycle Jitter



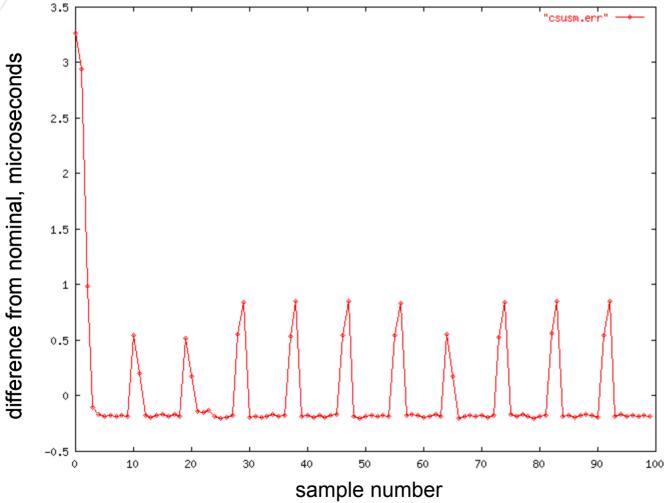


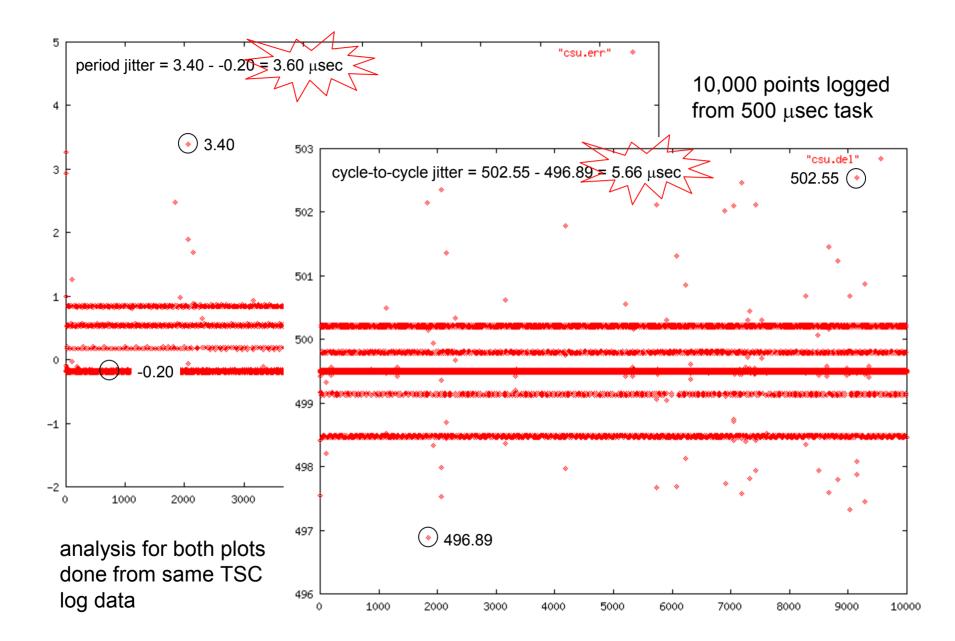
## **Period Jitter**

- Period jitter is calculated by computing best-fit line to TSC log values, then taking the difference between the maximum and minimum deviations from this line
- With period jitter, a single late task invocation will penalize only a single pulse; the following pulse will occur on schedule
- This jitter value is effectively equal to the scheduling delay, and is about half the cycle-to-cycle value
- If synchronization with external triggers is important, the period jitter value is the most meaningful



## **Period Jitter**

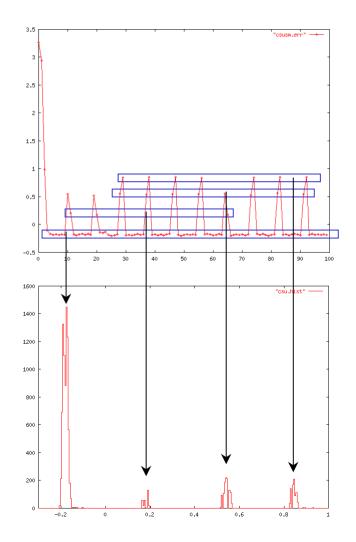


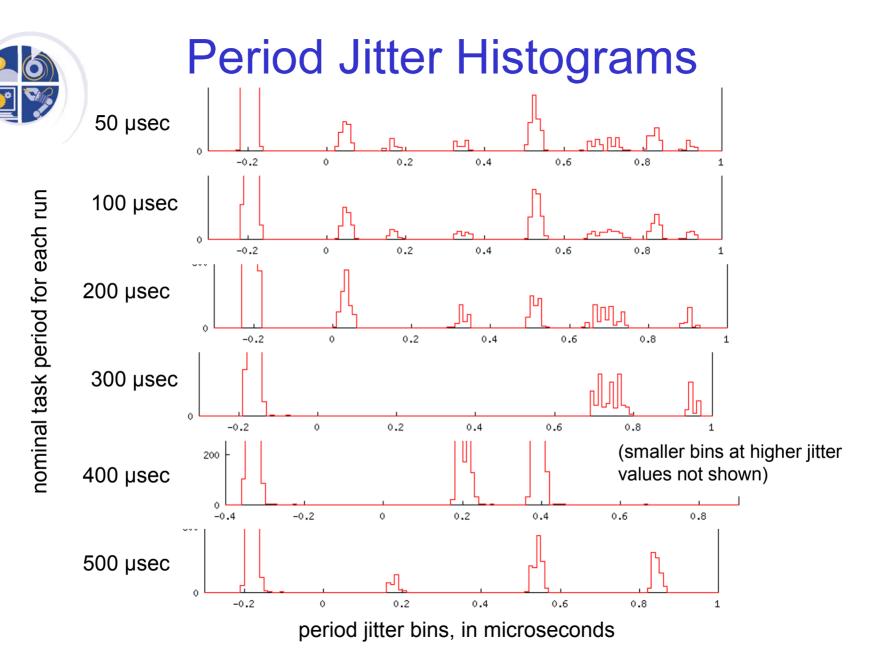




## **Jitter Bands**

- Bands in the jitter plots indicate a clustering of time stamp deviations
- Histograms of the period jitter values show this clustering more clearly
- Clusters are consistent across different tests, suggesting common origins



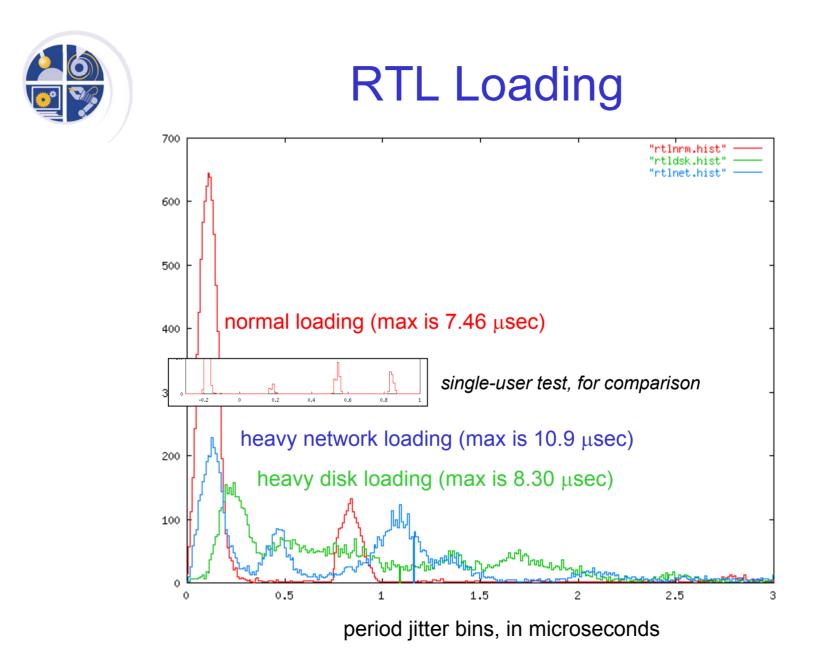


NIST • Manufacturing Engineering Laboratory • Intelligent Systems Division



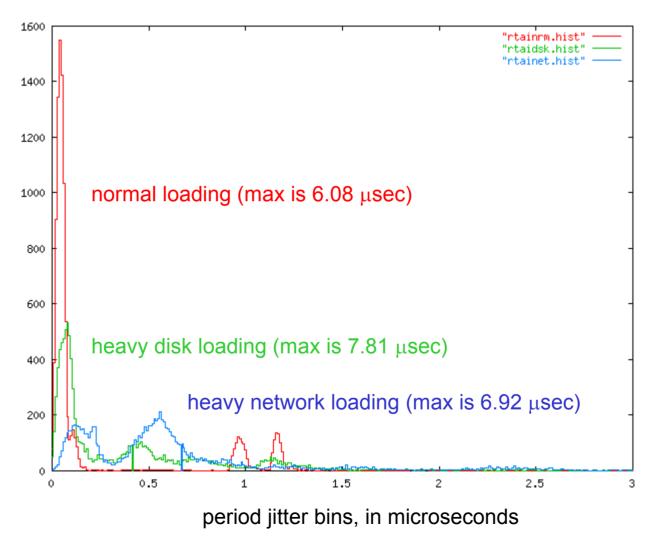
## Effects of Processor Load

- As the processor is more heavily loaded, real-time performance will suffer, if only due to cache displacement of RT code
  - the previous jitter measurements were done in single-user mode, with minimal processor loading
  - subsequent measurements of period jitter in loaded conditions shows increased variation
- Surprisingly, for a given task period, faster processors will show slower RT task times
  - more non-RT code runs between RT tasks and dirties up the cache
  - multiprocessor partitioning of RT, non-RT code helps





## **RTAI Loading**





## A Method to Reduce Jitter

- TSC can be used to reduce jitter, as proposed by Tomasz Motylewski of the University of Basel
- A series of subtasks polls the TSC for the precise instant that the time-critical code should execute
  - most subtasks return immediately, since target TSC is farther in future than the subtask period
  - the final subtask cycle polls the TSC until the target is reached
- CPU load depends on time to service subtasks, and time spent polling
  - more frequent subtasks incur too much overhead from null cycles
  - less frequent subtasks incur too much polling during final cycle



## **Optimal Subtask Scheduling**

Load analysis:

- 1. T/A subtask cycles
- 2. T/A-1 null cycles, 1 polling cycle
- 3. Time to service null cycles is (T/A-1) \* S
- 4. Worst case poll time is A
- 5. Load is

$$load = \frac{(T/A - 1)S + A}{T}$$

6. Minimizing with respect to A:

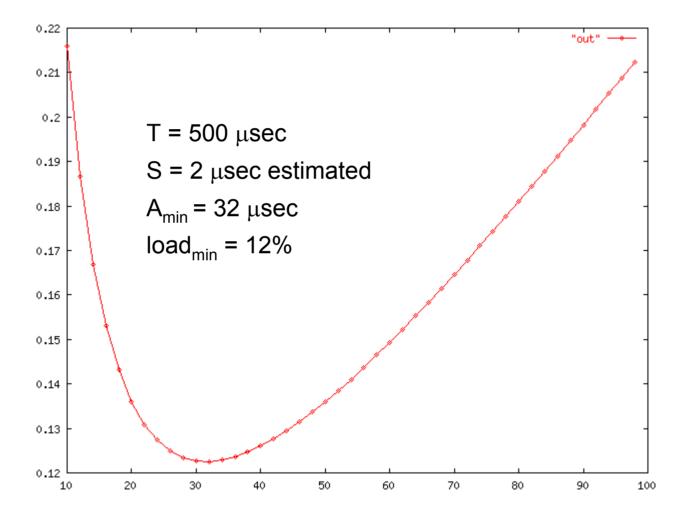
$$A_{\min} = \sqrt{ST}$$

 $load_{\min} = \frac{2\sqrt{ST - S}}{T}$ 

T/A subtask cycles

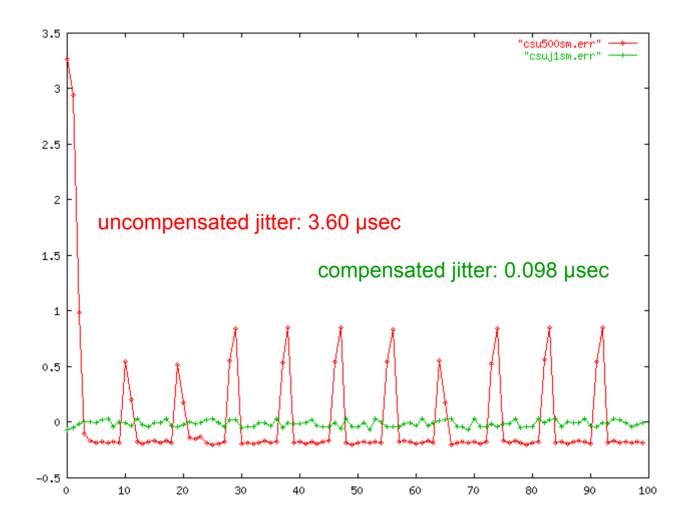


## **Optimal Example**



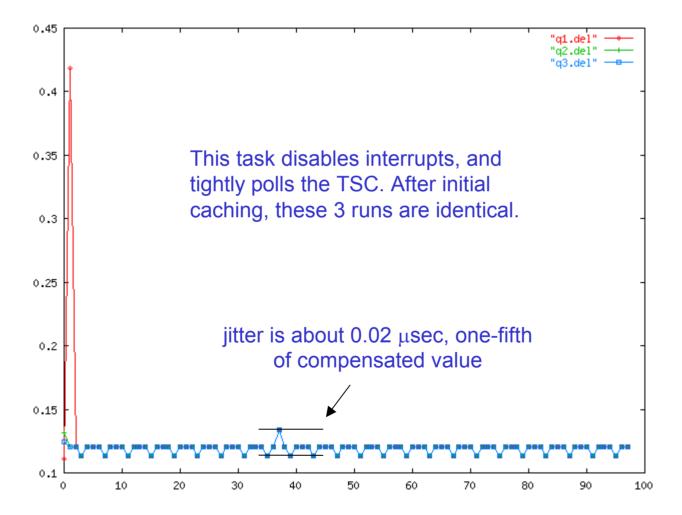


## **Compensated Jitter**











## Summary

- Performance measures answer the question, "How can I tell that a real-time operating system is able to satisfy my application's timing requirements?"
- Classic measures include interrupt service routine latency and scheduling jitter
- Both external and internal techniques can be used to measure these
- The testing environment is important if results are to be compared
- Internal techniques can be adapted to reduce scheduling jitter at the expense of processor time