Benchmarking Techniques

Michael Hope <michael.hope@linaro.org>

About Linaro

Compilers, debuggers, emulation, profiling, trace

Upstream first, then backport

GCC, GDB, QEMU,
perf, valgrind, ltrace

A selection of improvements

Vectoriser
Thumb-2 support in libraries
Cross-debug support
Cortex-A9 model in QEMU

Methods

Open, accessible group

Bug process with timing
Open planning
Auto builder
Merge requests
Verification
Performance testing

Why benchmark?

Issues are

Relevance

Accuracy

Repeatability

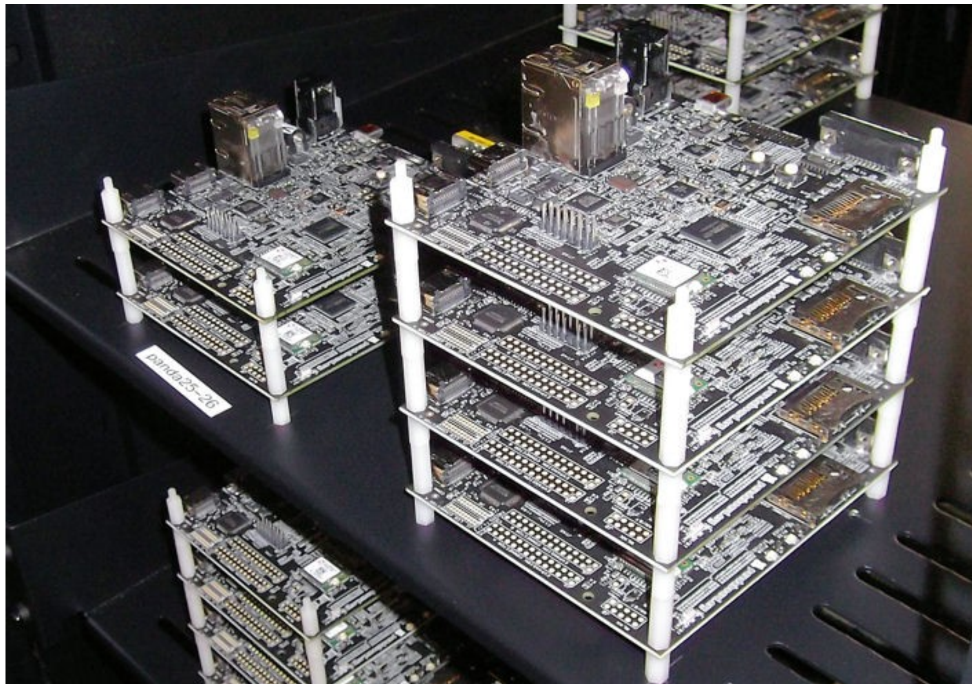
Picking relevant benchmarks:

Profile
Workload
Features

We use SPEC 2000 and EEMBC

We'd like shareable benchmarks

Test Platform



Build hosts

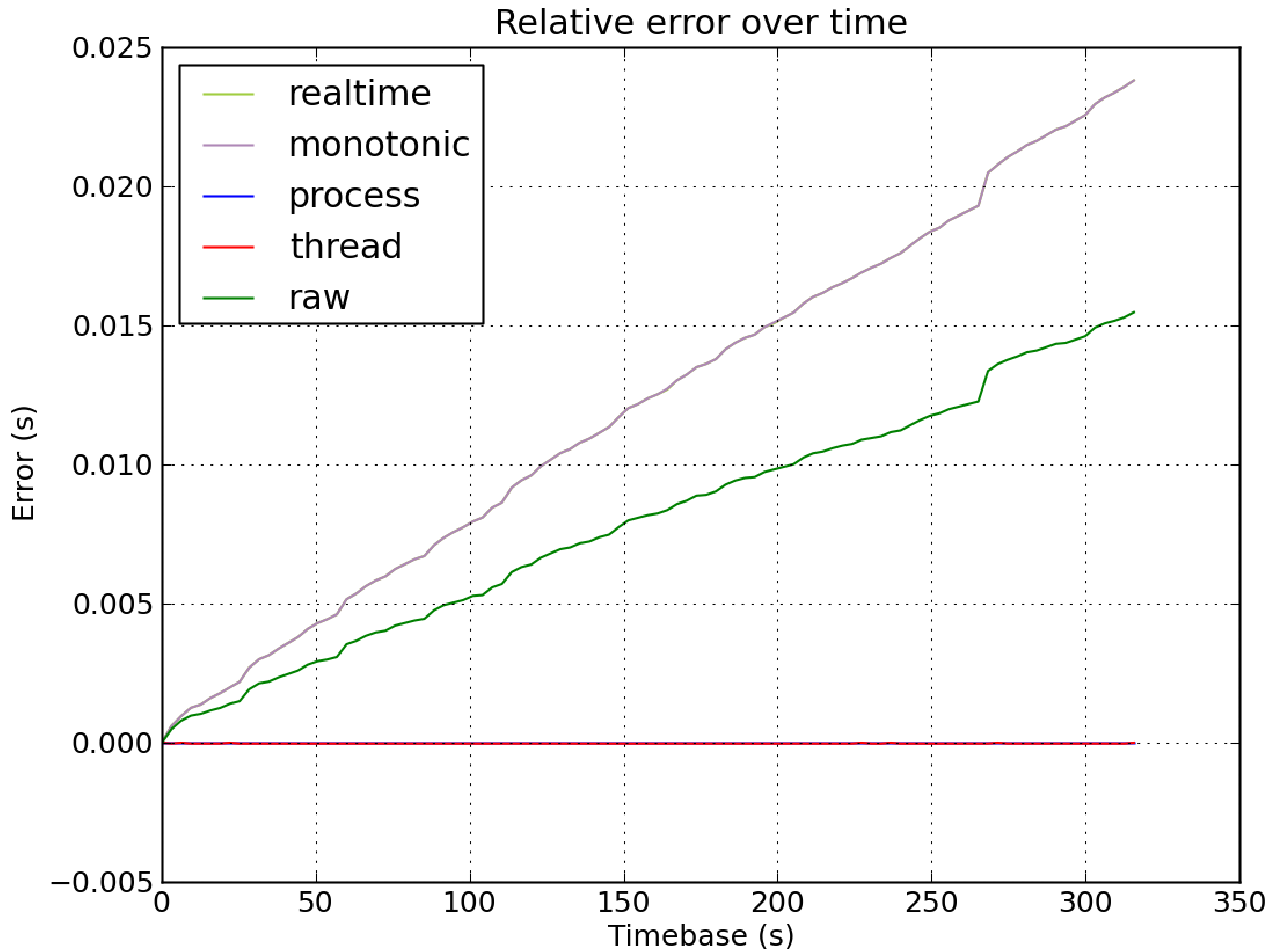
Name	State	Details	Last seen
leo1	idle *load 0.04 users cbuild*		1.2 minutes
leo2	idle *load 0.04 users cbuild*		1.5 minutes
oort1	lurking *load 1.16 users cbuild*		1.7 minutes
oort2	lurking *load 0.42 users cbuild*		0.5 minutes
pavo1	idle *load 0.0 users cbuild*		1.5 minutes
tcpanda01	idle *load 0.0 users cbuild*		1.0 minutes
tcpanda02	idle *load 0.05 users cbuild*		1.1 minutes
tcpanda03	updating *load 0.0 users cbuild*		0.3 minutes
tcpanda04	idle *load 0.03 users cbuild*		2.2 minutes
tcpanda05	updating *load 0.0 users cbuild*		0.3 minutes
tcpanda06	updating *load 0.0 users cbuild*		0.2 minutes
ursa1	reserved *load 0.05 users michaelh*	michaelh1	2.7 days
ursa2	running *load 2.32 users cbuild,michaelh*	denbench-o3-neon-cortexa9-build	5.2 minutes
ursa3	running *load 1.73 users cbuild,michaelh*	spec2000-o3-neon-cortexa8-run	25 minutes
ursa4	running *load 0.0 users cbuild*	denbench-o3-neon-publish	4.8 days

Build / test / benchmark via Linux / web / commodity hardware

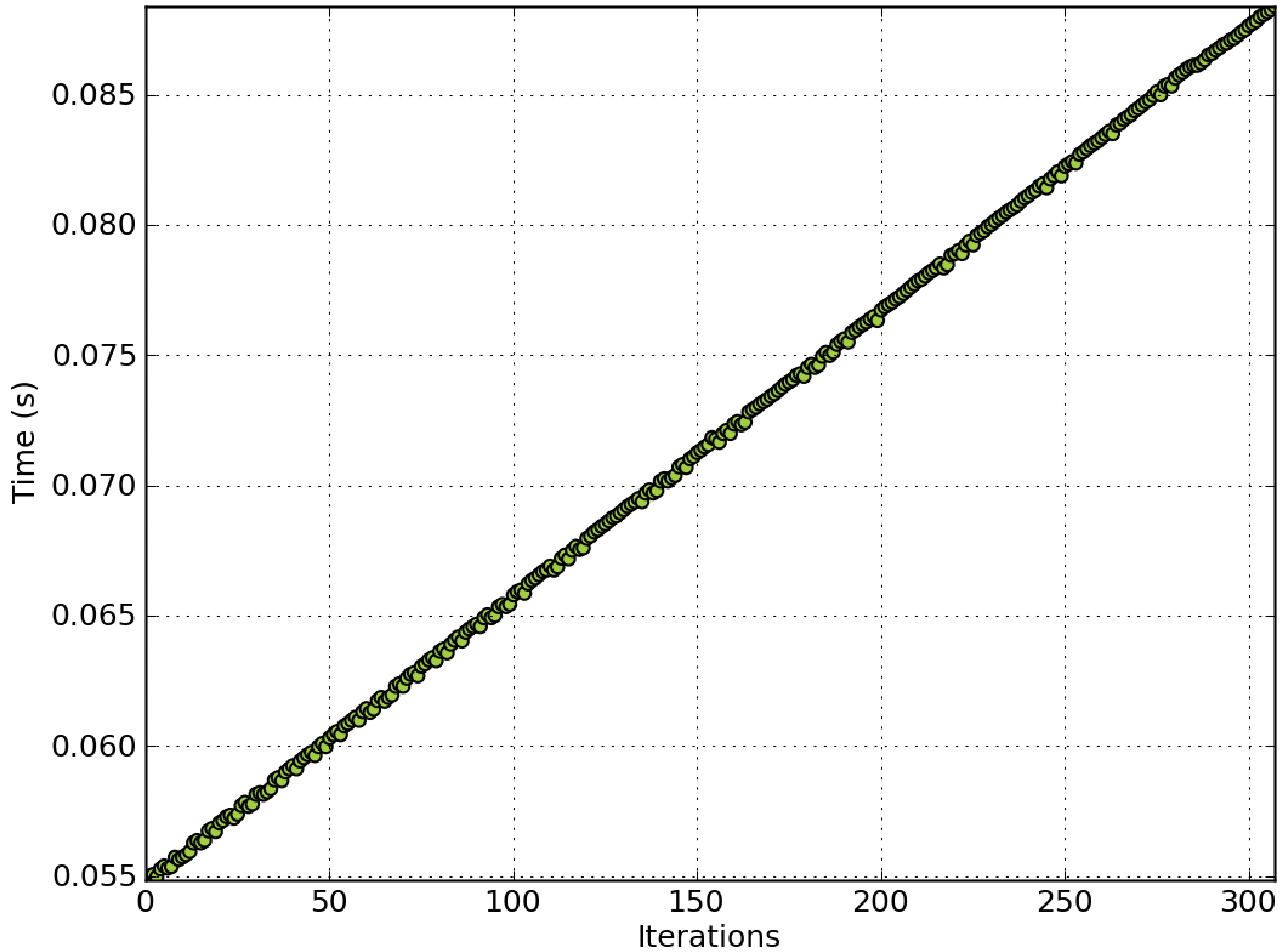
Measuring

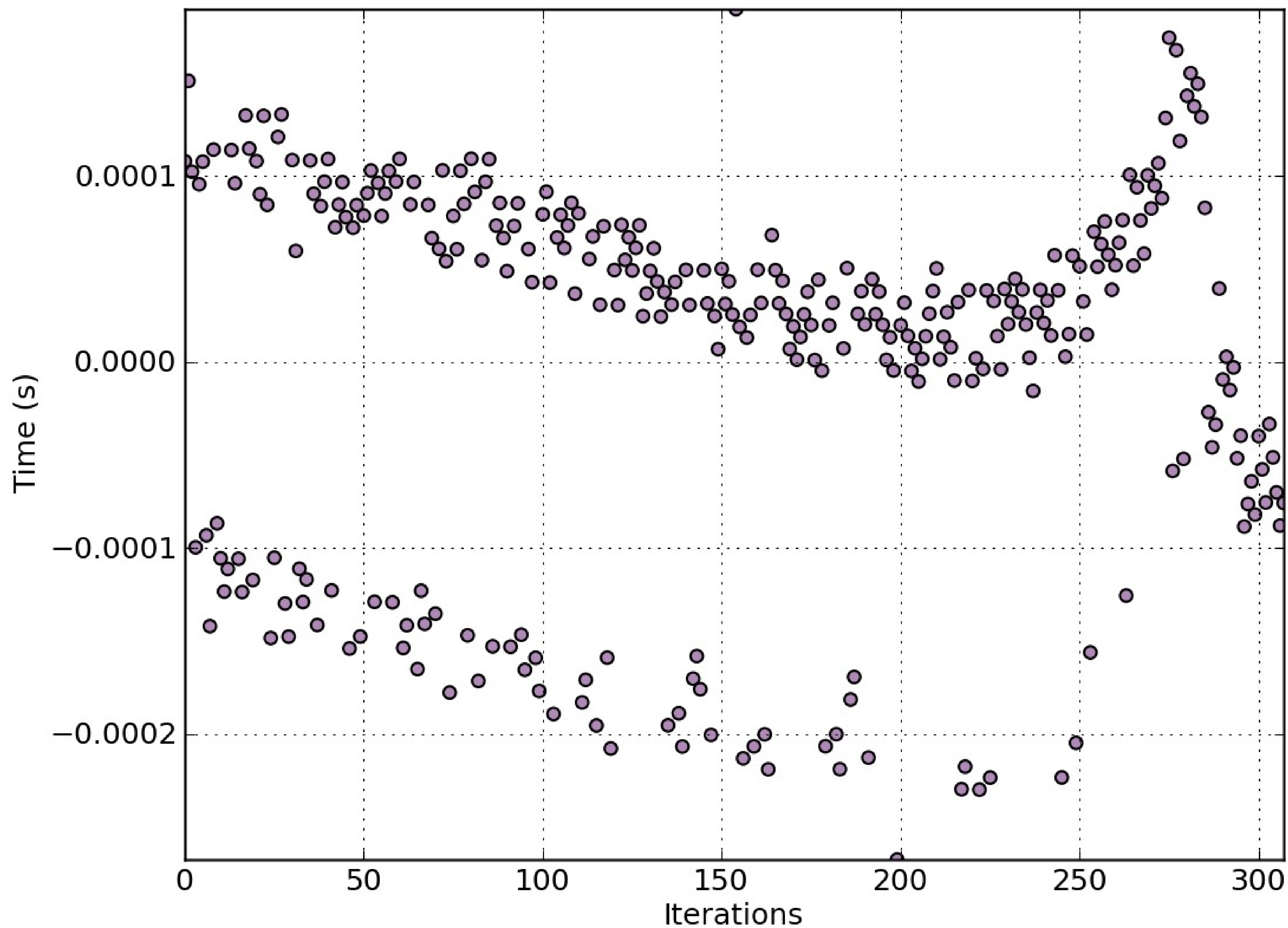
Realtime timers

See '`man clock_gettime`'



See https://wiki.linaro.org/WorkingGroups/ToolChain/Benchmarks/TimerAccuracy


Time to perform a number of iterations
Time (s)
0.055
0.060
0.065
0.070
0.075
0.080
0.085
0
50
100
150
200
250
300
Iterations



Detrended time to perform a number of iterations
Time (s)
Iterations
0.0001
0.0000
−0.0001
−0.0002
0
50
100
150
200
250
300

External influences

Idle

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	4.993	4.993	0.000353		
High	4.993	4.993	0.000190	100.001%	53.875%
Nice	4.993	4.993	0.000255	99.999%	72.225%

CPU load

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	14.966	4.992	0.007007		
High	5.096	4.992	0.004316	99.989%	61.589%
Nice	65.311	4.991	0.005056	99.971%	72.149%

Network load

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	5.134	5.099	0.005952		
High	5.112	5.099	0.005798	100.009%	97.397%
Nice	5.134	5.099	0.005952	100.000%	100.000%

Other features to be wary of

scheduler

governor

cpuidle, power management, thermal limiting

SMP

bugs, like core lockdown

NEON startup

See “Understanding the Linux Kernel” ch10:

http://oreilly.com/catalog/linuxkernel/chapter/ch10.html

Dispersion across boards
...and across tests

Putting things together and running

We use:

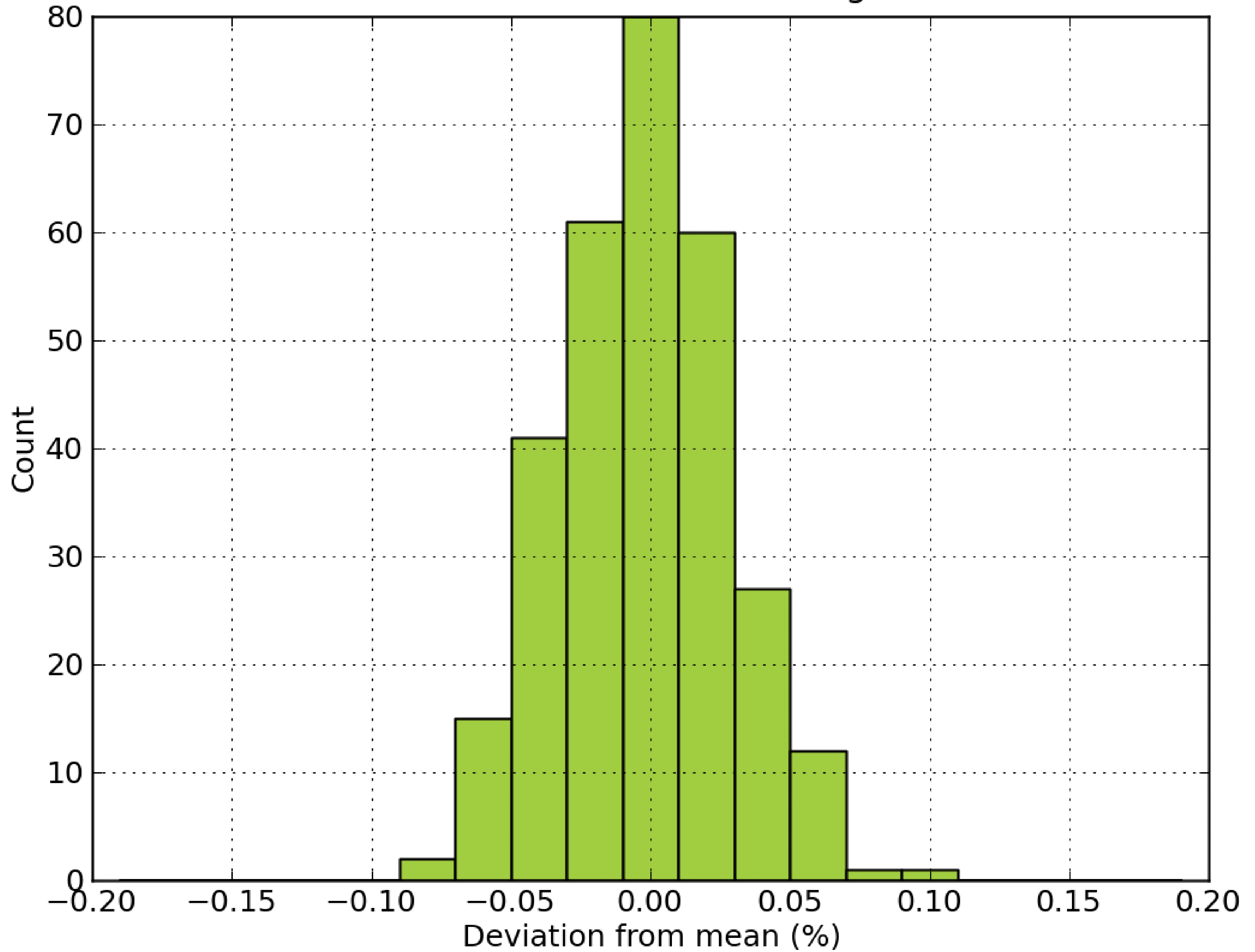
timer built into the app

run five times

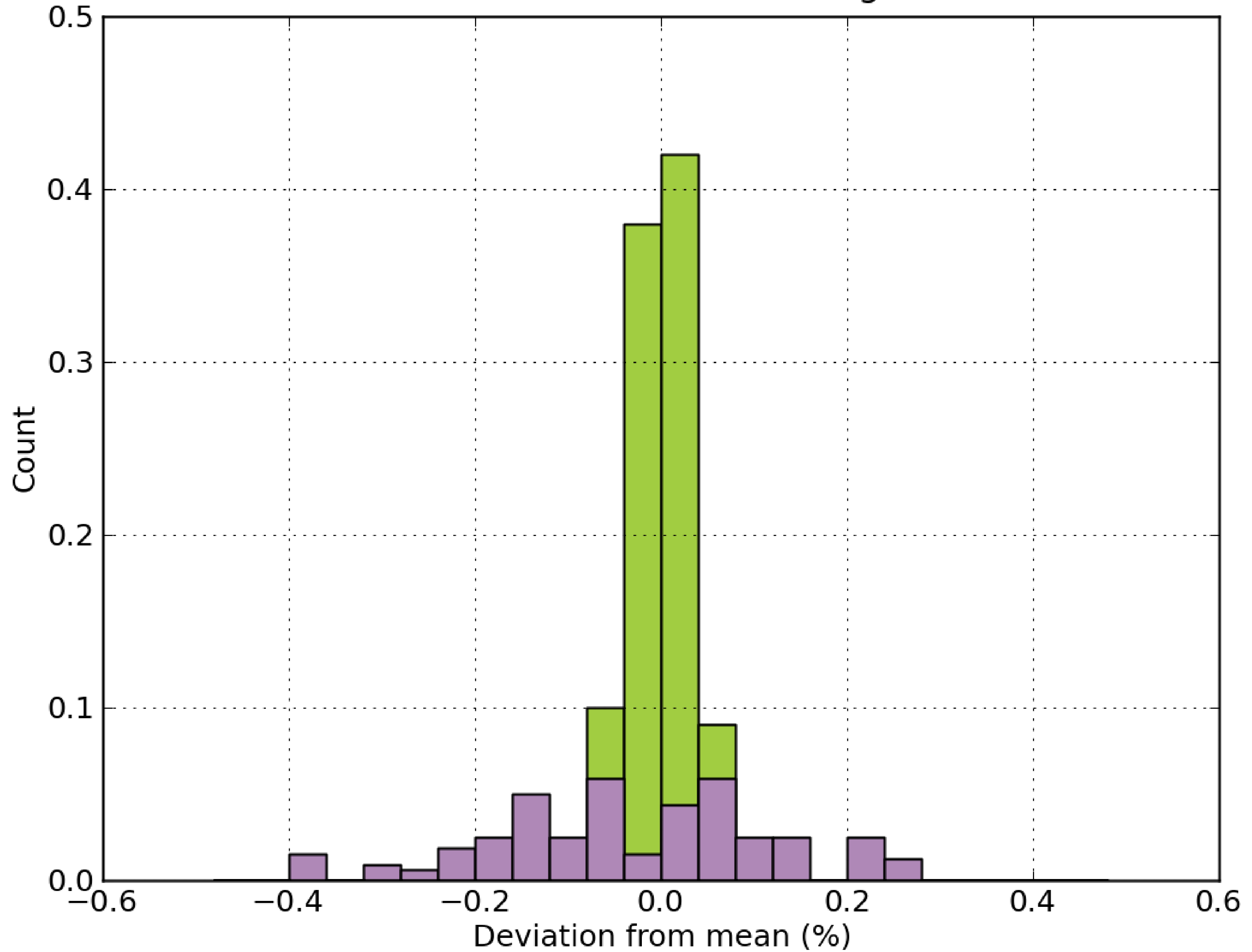
collect everything

post process

Statistics



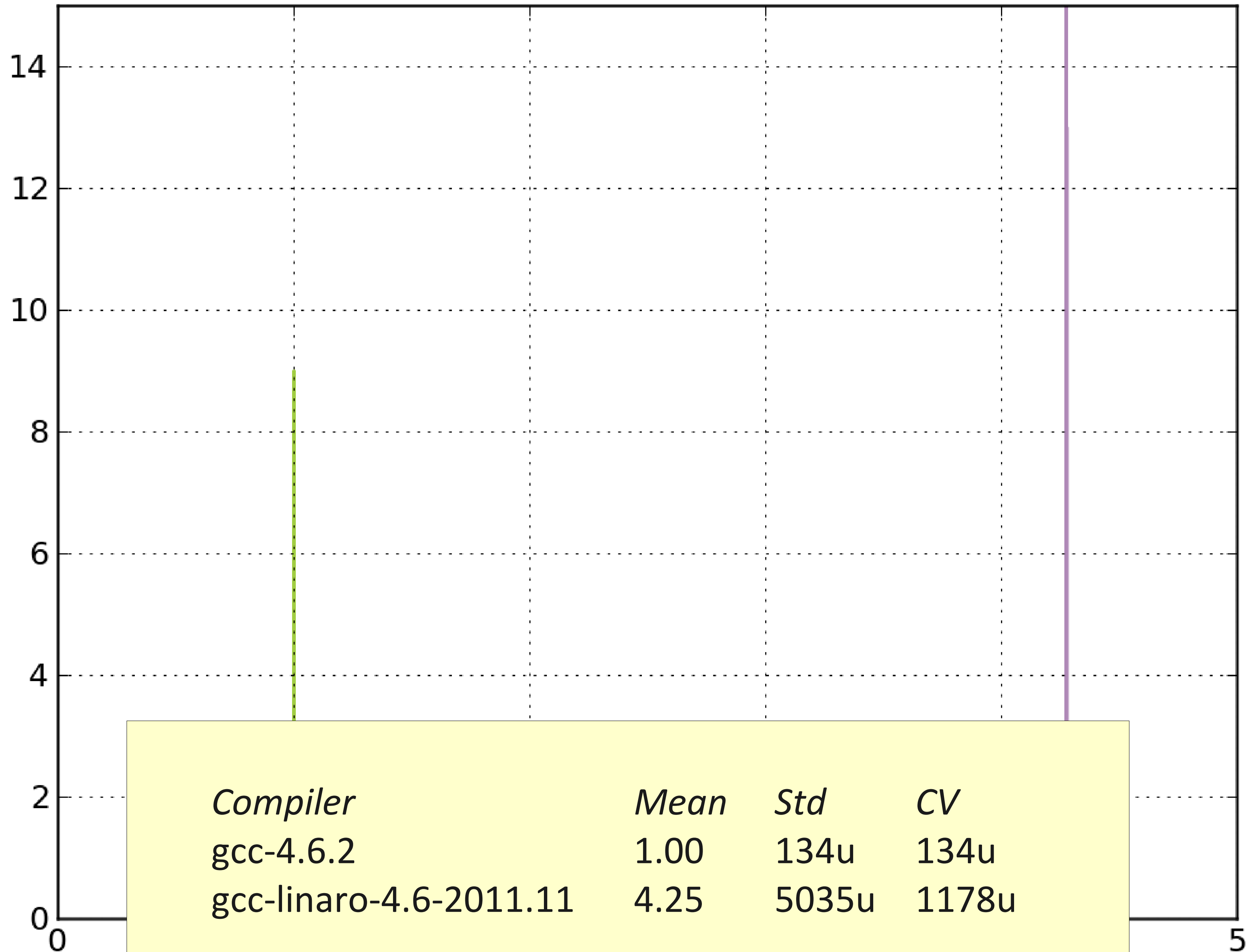
coremark normalised histogram
Count
Deviation from mean (%)
0
10
20
30
40
50
60
70
80
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20


cacheb01 normalised histogram
Count
Deviation from mean (%)
0.5
0.4
0.3
0.2
0.1
0.0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6

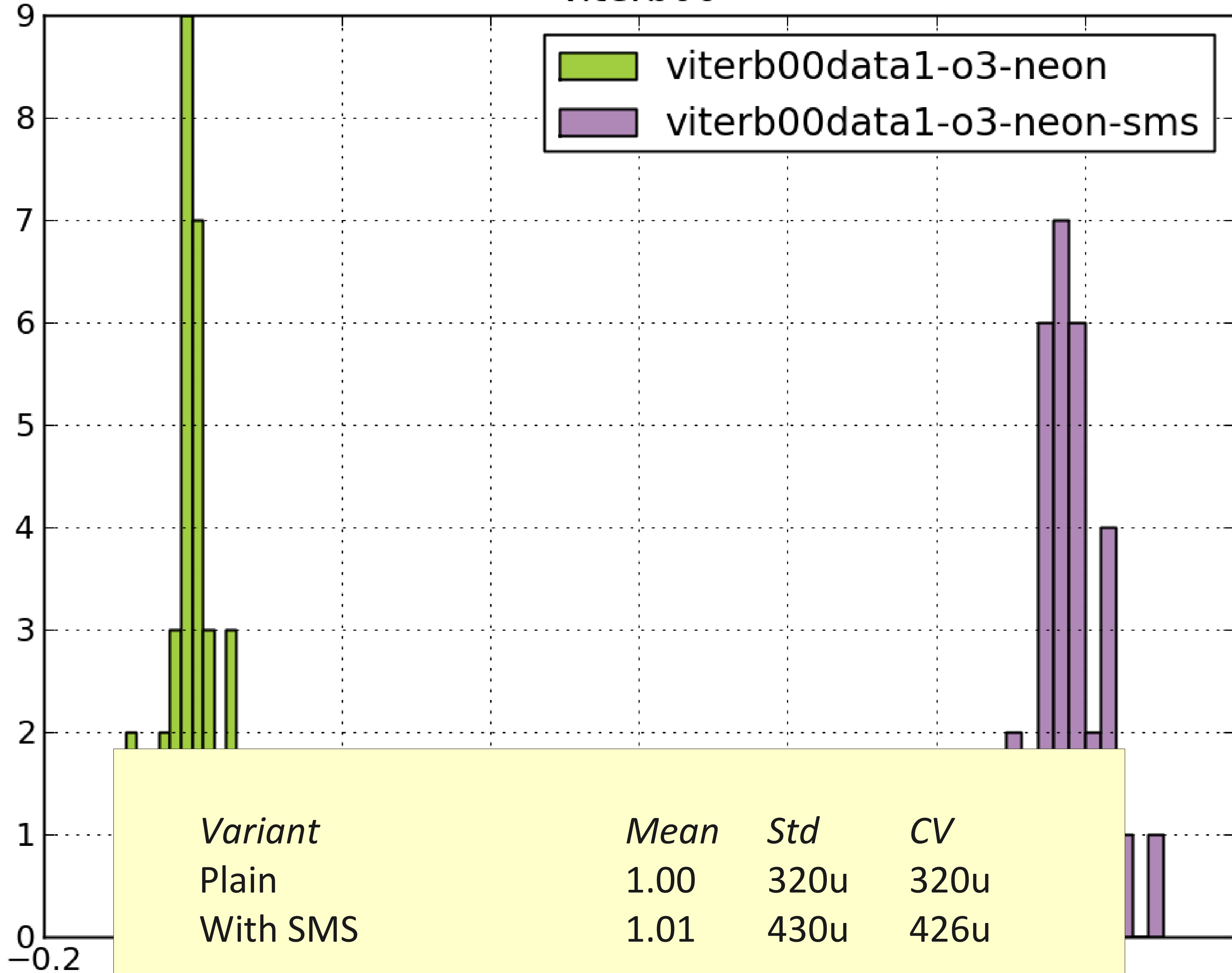
Standard deviation
Dispersion / coefficient of variance
t-scores
t-test

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

See http://en.wikipedia.org/wiki/Welch%27s_t_test

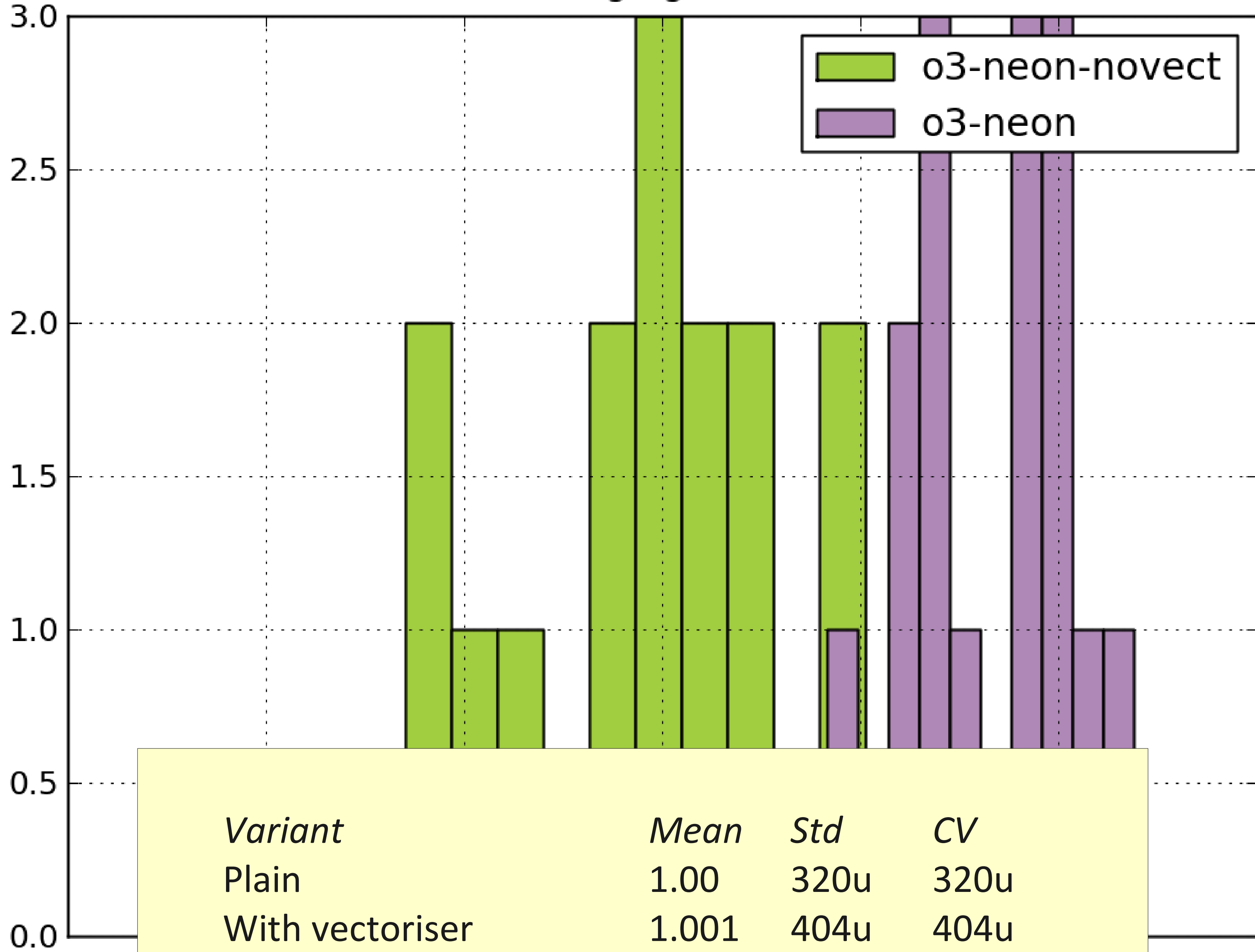

rgbcmy01
14
12
10
8
6
4
2
0
0
5
Compiler Mean Std CV
gcc-4.6.2 1.00 134u 134u
gcc-linaro-4.6-2011.11 4.25 5035u 1178u
t = 30,300

Closer example:
Effect of modulo scheduling (SMS)



Variant	*Mean*	*Std*	*CV*
Plain	1.00	320u	320u
With SMS	1.01	430u	426u

t = 118



Variant	*Mean*	*Std*	*CV*
Plain	1.00	320u	320u
With vectoriser	1.001	404u	404u

t = 8.27 - significant

Our tools

perf
difftest
betters
tabulate
Python
LibreOffice!

Other statistical tools like scipy.stat, R, Judge, and ministat

results.csv - LibreOffice Calc

File Edit View Insert Format Tools Data Window Help

Arial 10

B149 gcc-4.6.1

	A	B	C	D	E	F	G
1	testnam	version	klas	variant	subname	nsample	min
140	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4042
141	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.42
142	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4442
143	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4462
144	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4
145	spec2000	gcc-4.6.1			176_gcc	5	5.5171
146	spec2000	gcc-4.6.1			176_gcc	5	5.5227
147	spec2000	gcc-4.6.1			176_gcc	5	5.5346
148	spec2000	gcc-4.6.1			176_gcc	5	5.5491
149	spec2000	gcc-4.6.1		o3-neon	176_gcc	5	5.6
594	spec2000	gcc-linaro-4.6-2011.11		o2	254_gap	5	10.419
611	spec2000	gcc-linaro-4.6-2011.11		o3-neon	254_gap	5	10.4
647	spec2000	gcc-4.6.1		o2	254_gap	5	10.639
661	spec2000	gcc-4.6.1		o3-neon	254_gap	5	10.7
706	spec2000	gcc-linaro-4.6-2011.11		o3-neon-novect	254_gap	5	11.128
731	spec2000	gcc-linaro-4.6-2011.11		o3-vfpv3	254_gap	5	11.281
732	spec2000	gcc-4.6.1		o3-neon-novect	254_gap	5	11.281
733	spec2000	gcc-linaro-4.6-2011.11		o3	254_gap	5	11.285

AutoFilter dropdown (variant):
- All
- Top 10
- Standard Filter...
- - empty -
- - not empty -
- o2
- o3
- o3-neon
- o3-neon-novect
- o3-vfpv3

http://help.libreoffice.org/Calc/Applying_AutoFilter

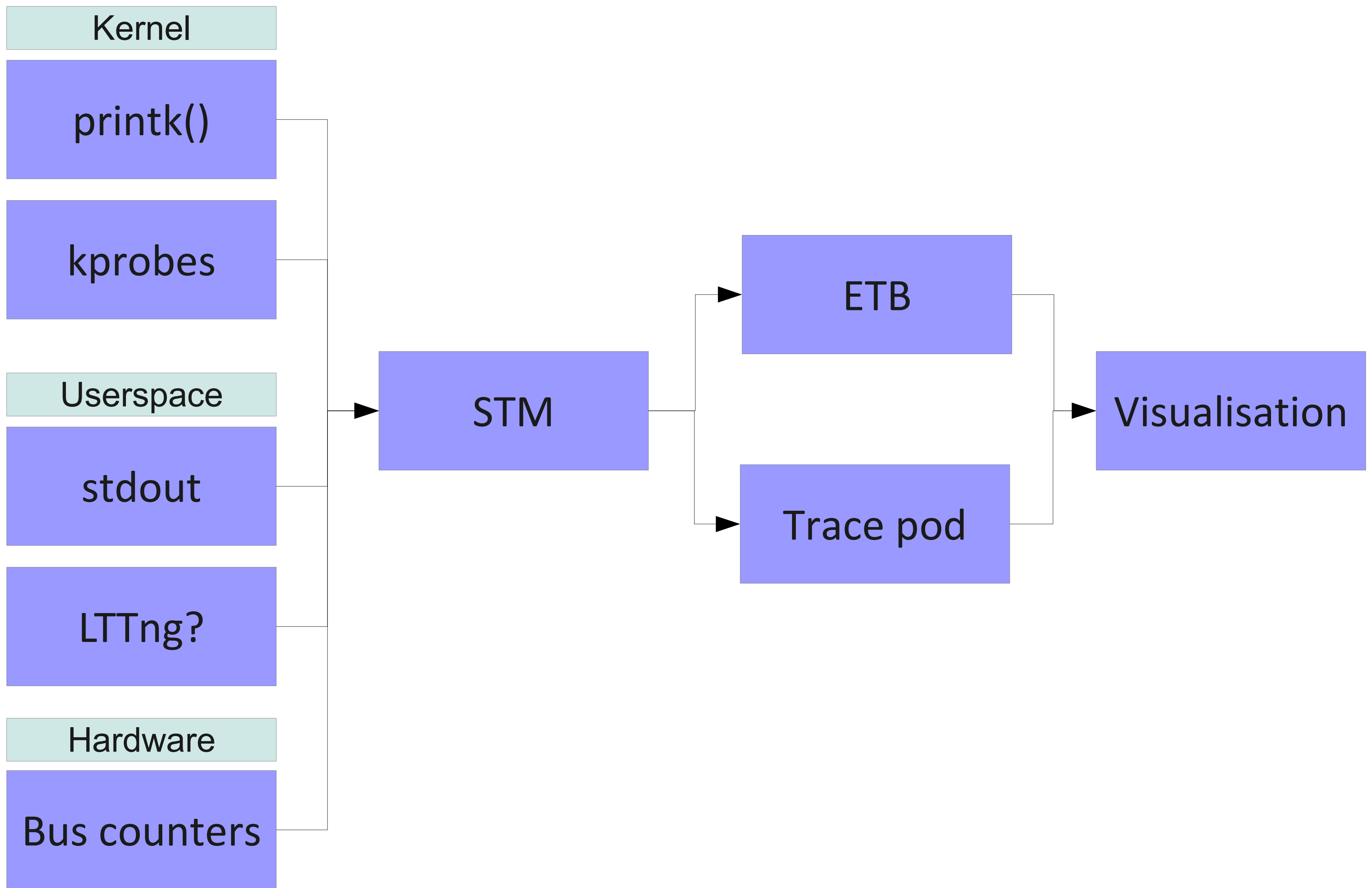
Digging deeper: using perf

```
michaelh@leo1:~/coremark_v1.0$ perf report -n -d coremark.exe --stdio
# dso: coremark.exe
# Events: 15K cycles
#
# Overhead  Samples        Command                 Symbol
# ........ ..........  ............  ......................
#
    37.96%      5980  coremark.exe  [.] core_state_transition
    27.21%      4197  coremark.exe  [.] core_bench_list
    16.25%      2522  coremark.exe  [.] matrix_test
     7.04%      1110  coremark.exe  [.] crcu32
     6.50%       931  coremark.exe  [.] crc16
     2.42%       396  coremark.exe  [.] core_bench_state
     1.70%       279  coremark.exe  [.] crcu16
     0.87%       143  coremark.exe  [.] calc_func
     0.04%         6  coremark.exe  [.] core_bench_matrix
     0.02%         3  coremark.exe  [.] main


#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

Future

Open benchmarks
Scientific benchmarks
Locking down the environment

Future: System level trace


Kernel
printk()
kprobes
Userspace
stdout
LTTng?
Hardware
Bus counters
STM
ETB
Trace pod
Visualisation

www.linaro.org / wiki.linaro.org

people.linaro.org/~michaelh/presentations

Benchmarking Techniques

Michael Hope <michael.hope@linaro.org>

bzr branch lp:~michaelh1/+junk/benchmarking-techniques r6

Hi, I am...

This presentation is a technical talk on the techniques and engineering side of benchmarking and how we benchmark inside the toolchain group at Linaro.

About Linaro

2

Dave Russling talked about Linaro earlier today
Engineering organisation
Members pool together their engineers to work on common goals so that those are better and they can focus on the points of differentiation

Works pretty well, well over a year old, over 100 people

Compilers, debuggers, emulation, profiling, trace

Upstream first, then backport

GCC, GDB, QEMU,
perf, valgrind, ltrace

In my working group we work on everything toolchain related including compilers, debuggers, emulation, profiling, trace, and emulation. We work upstream to make the next version better, but also backport these improvements to our Linaro products, like gcc-linaro, so that you can pick up these changes early

A selection of improvements

Vectoriser
Thumb-2 support in libraries
Cross-debug support
Cortex-A9 model in QEMU

Methods

Open, accessible group

Bug process with timing
Open planning
Auto builder
Merge requests
Verification
Performance testing

Why benchmark?

6

You benchmark because you want to compare things, be it different boards, mass storage, compilers, or even individual improvements.

A benchmark is a focused set of tests that together represent the real workloads of the end user.

We're focused on performance, so we benchmark to track how a change improves things and catch any unexpected regressions.

Issues are

Relevance
Accuracy
Repeatability

7

A benchmark must be

Relevant to the profile, such as a mobile device.

Needs to be accurate. Can't vary wildly across runs as you can't compare small changes and, as part of that, needs to be repeatable.

Repeatable

Picking relevant benchmarks:

Profile
Workload
Features

We use SPEC 2000 and EEMBC

We'd like shareable benchmarks

8

We use a heirachy to justify the benchmarks.

Start out with a profile – currently mobile devices, but there's more and more on the server and virtualisation side coming down

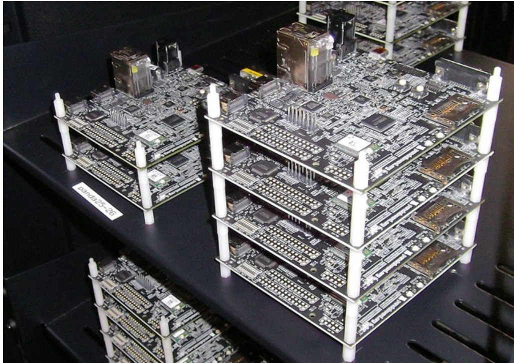
Then take that profile and figure out the workloads the machine will be doing. For mobile this is things like video decoding, audio encoding, web browsing, and to a lesser extent cryptography.

For server this might be serving web pages, or running a certain app under a JVM.

Then make sure it covers the features you want to improve. We improve core bound code instead of other areas like I/O or memory bound.

In the end this shakes sown to the SPEC CPU 2000 benchmarks and the EEMBC range, especially

Test Platform



Build hosts

Name	State	Details	Last seen
leo1	idle *load 0.04 users cbuild*		1.2 minutes
leo2	idle *load 0.04 users cbuild*		1.5 minutes
oort1	lurking *load 1.16 users cbuild*		1.7 minutes
oort2	lurking *load 0.42 users cbuild*		0.5 minutes
pavo1	idle *load 0.0 users cbuild*		1.5 minutes
tcpanda01	idle *load 0.0 users cbuild*		1.0 minutes
tcpanda02	idle *load 0.05 users cbuild*		1.1 minutes
tcpanda03	updating *load 0.0 users cbuild*		0.3 minutes
tcpanda04	idle *load 0.03 users cbuild*		2.2 minutes
tcpanda05	updating *load 0.0 users cbuild*		0.3 minutes
tcpanda06	updating *load 0.0 users cbuild*		0.2 minutes
ursa1	reserved *load 0.05 users michaelh*	michaelh1	2.7 days
ursa2	running *load 2.32 users cbuild,michaelh*	denbench-o3-neon-cortexa9-build	5.2 minutes
ursa3	running *load 1.73 users cbuild,michaelh*	spec2000-o3-neon-cortexa8-run	25 minutes
ursa4	running *load 0.0 users cbuild*	denbench-o3-neon-publish	4.8 days

Build / test / benchmark via Linux / web / commodity hardware

We use auto builders for the build, test and benchmark. Here's a nice stack of PandaBoards in our validation lab.

The same scheduler system is used for all jobs. Scales lineary, easy to install, just nice in general.

For consistency we use Linux everywhere. This introduces noise which we'll quantify and check using statistics later in this presentation.

We're looking into alternatives like running a u-boot app, a kernel module, or using STM tracing to reduce the noise.

More on STM?

Measuring

10

Let's measure first

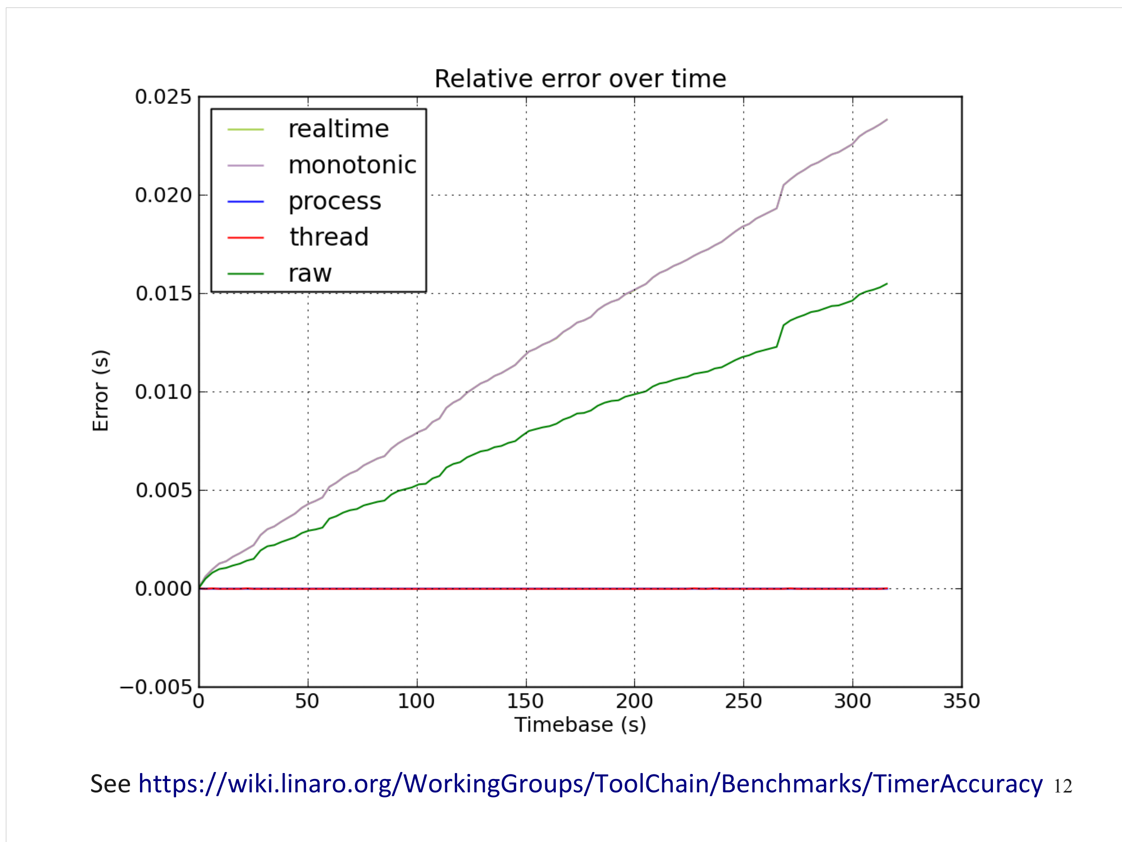
Realtime timers

See 'man clock_gettime'

11

Throughput or speed wise, you want to measure execution time and the best timer to use is the POSIX per-process time.

There are others, such as wall time, monotonic time, and monotonic raw but these have different problems.

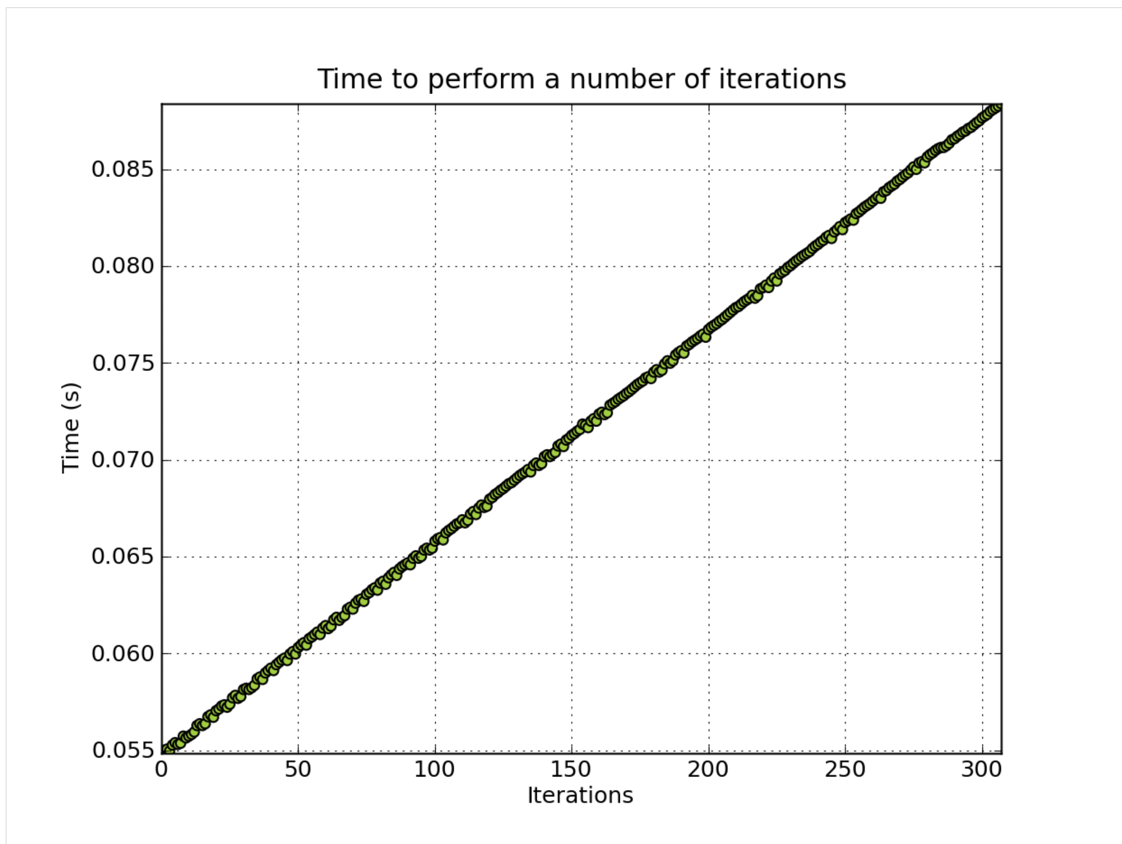


Here's the clocks side by side referred back to process time. They fall into three categories and you can see the difference.

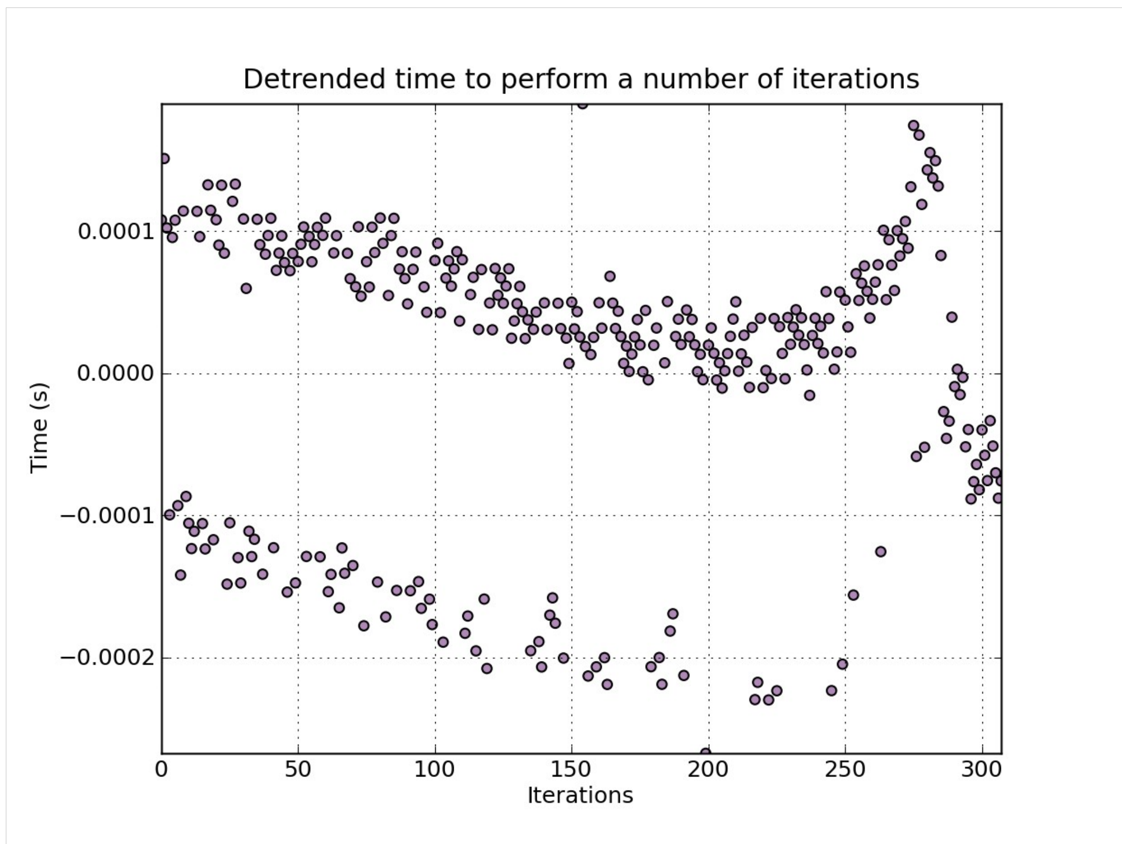
The top is monotonic time. Always increases, includes NTP corrections, and is sensitive to CPU load. Notice the glitch due to CPU load on the right.

Middle is raw monotonic which is the same minus NTP corrections.

The bottom is the control – process time and the identical thread time. Notice that there's no glitch either



Are they accurate? Here's a number of CPU loops versus process time. As you can see it's nice and linear and ticks up very well with increasing loops count. There's no staircasing that you'd see with a low resolution timer.



Here's the detrended version. There's no correlation here in the error. The left axis is in seconds so you can see we have much better than millisecond resolution.

External influences

15

Process time is good but you should still reduce external influences.

Idle

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	4.993	4.993	0.000353		
High	4.993	4.993	0.000190	100.001%	53.875%
Nice	4.993	4.993	0.000255	99.999%	72.225%

CPU load

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	14.966	4.992	0.007007		
High	5.096	4.992	0.004316	99.989%	61.589%
Nice	65.311	4.991	0.005056	99.971%	72.149%

Network load

	Wall mean	Mean	Stddev	Mean vs plain	Stddev vs plain
Plain	5.134	5.099	0.005952		
High	5.112	5.099	0.005798	100.009%	97.397%
Nice	5.134	5.099	0.005952	100.000%	100.000%

The main ones are non-benchmark CPU load and I/O load.

When the host is idle the results are really tight. Increasing the process priority does pull in the variation a bit but not enough for the added complexity.

CPU load greatly affects the wall time but not the process time. It again bumps the variance up.

Network or other I/O load is a problem and should be avoided. Make sure your test is pre-loaded into the cache and puts any temporary files into a tmpfs.

Other features to be wary of

scheduler
governor
cpuidle, power management, thermal limiting
SMP
bugs, like core lockdown
NEON startup

See “Understanding the Linux Kernel” ch10:
http://oreilly.com/catalog/linuxkernel/chapter/ch10.html

17

Keep an eye out for other kernel effects. Higher priorities help, as does real time scheduling but make sure you don't wind the priority so high that a errant benchmark locks out everything else.

Check the CPU performance governor else the kernel may switch the CPU clock during the run. Performance governor is best.

Check tickless versus not. Pick one and stick with it.

Check out O'Reilly's Understanding the Linux Kernel for more.

Dispersion across boards
...and across tests

Putting things together and running

We use:
timer built into the app
run five times
collect everything
post process

19

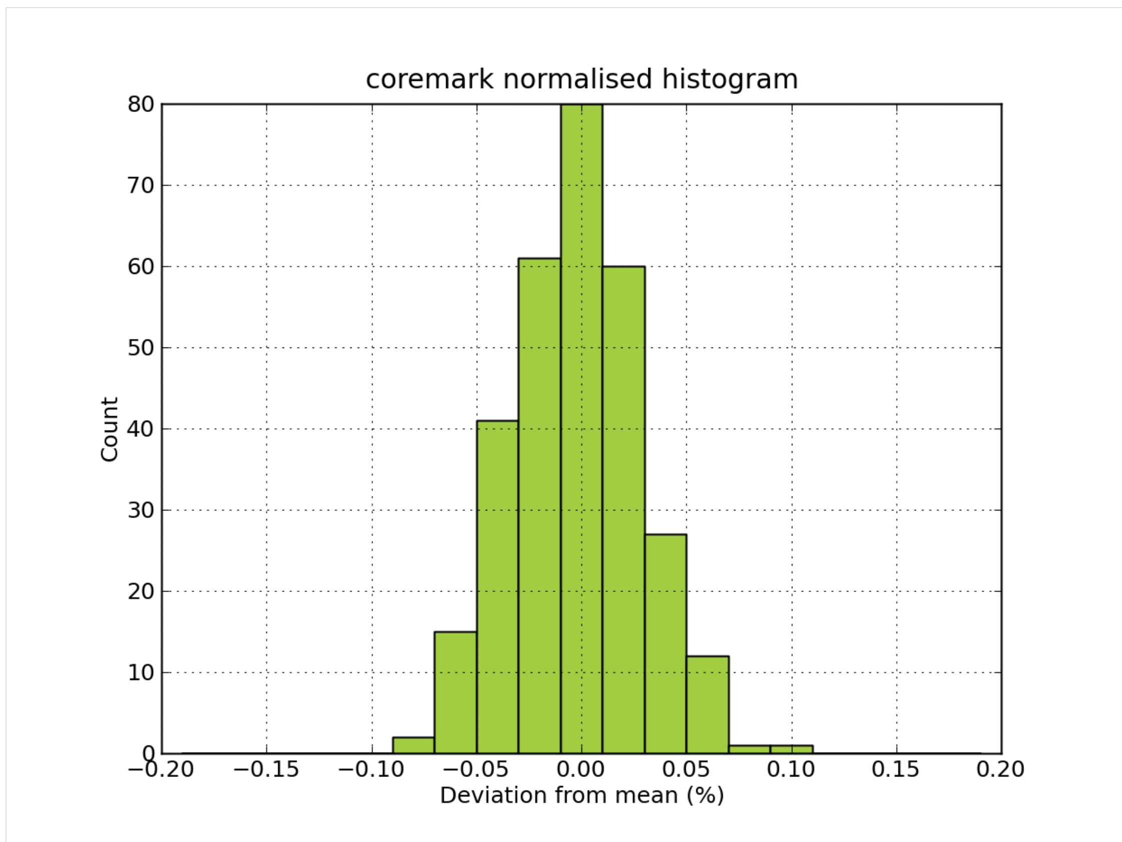
Timer in app: eliminate load, init time
Gives a median, bare minimum to calculate variance on
Never know what you'll want use in the future
Post process using easy languages and tools like Python

Statistics

20

Statistics is the most important part in reliable benchmarks. It tells you about

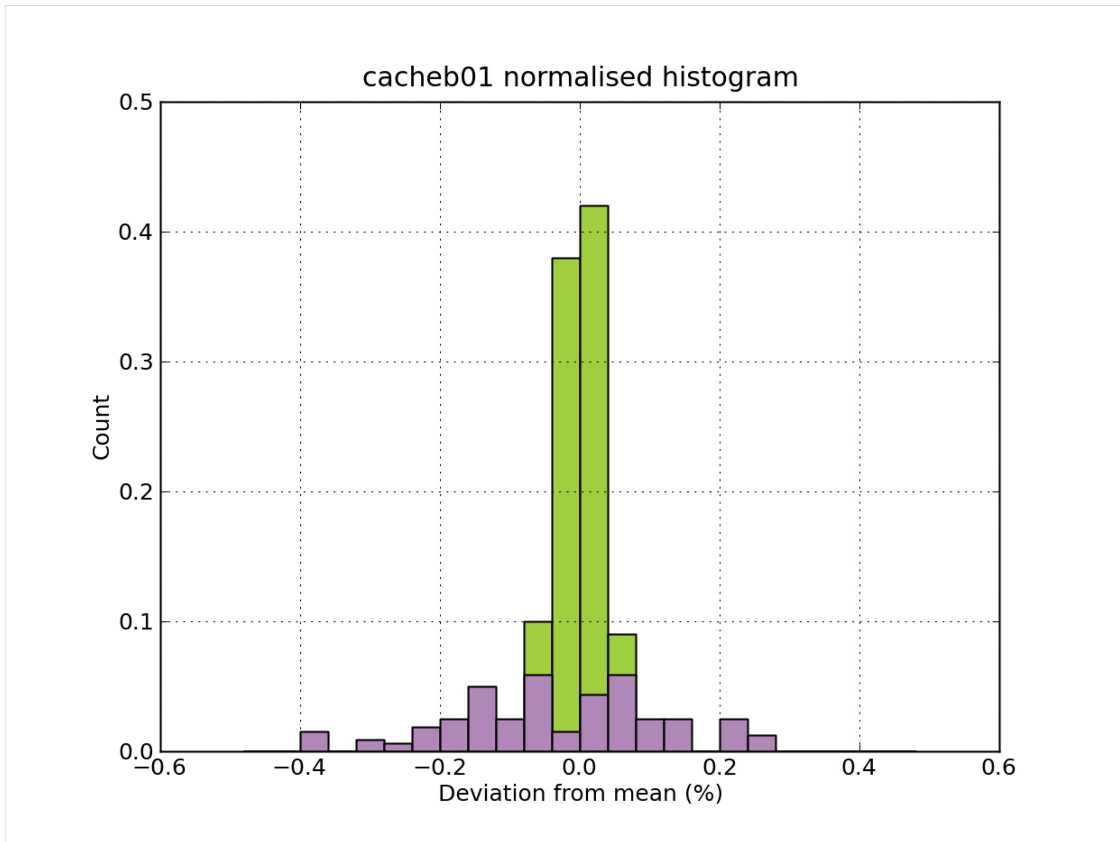
consistency in a run
consistency outside a run
if a result is significant
how tightly you can report a result



Coremark is...
Good work on timers
 No outliers
 Distribution is fairly symmetrical

Close enough to a normal distribution that we can use the tools you get with that type

Note that the best is always the fastest. Any interruption only lowers the score.



Do check per test

CPU bound like CoreMark have a nice tight ditribution

Others that hammer memory or I/O don't – here's the cache blaster benchmark from EEMBC. Much wider, more erratic.

Standard deviation
Dispersion / coefficient of variance
t-scores
t-test

Assumed a normal distribution
Can use standard deviation

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

See http://en.wikipedia.org/wiki/Welch%27s_t_test

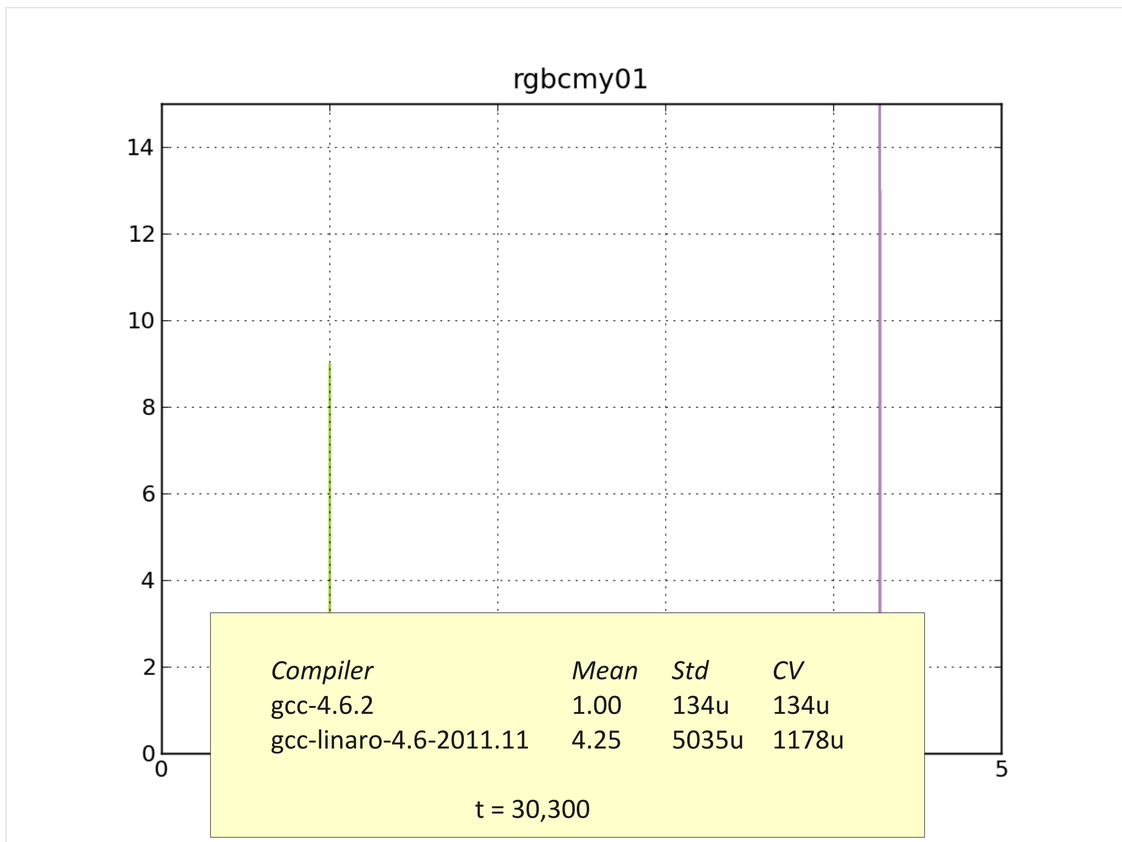
The most important euqation

Welch's t-test

Take two distributions, different variation, even a different sample size, calculate t-score

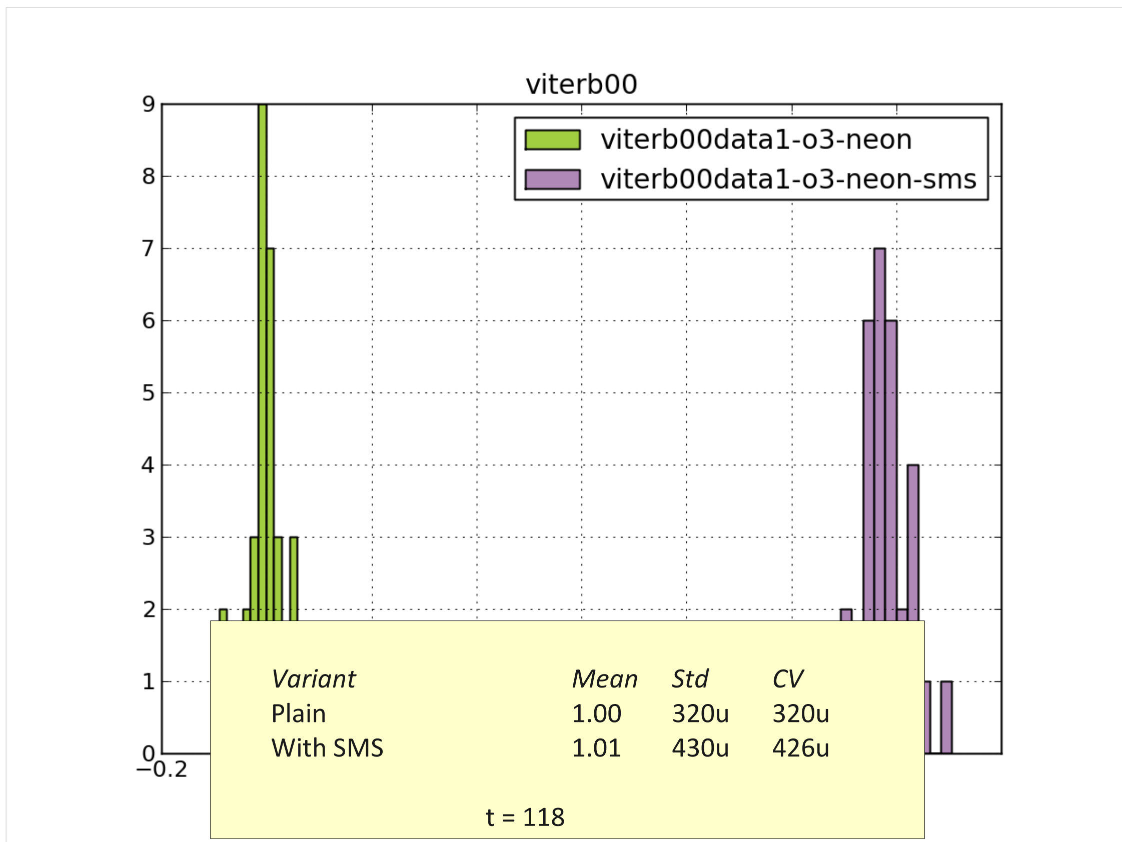
Then look that up in a t-distribution table or use Excel to get the p-number

Tells you how likely it is that the effect is real.



I couldn't resist this one
We're doing a lot of work on the vectoriser
Recognises data parallel code and converts to NEON instructions
Here's one we did on colour space conversion
<two peaks, miles apart>
<t score of +lots>
Visually and statistically the gain is significant

Closer example:
Effect of modulo scheduling (SMS)

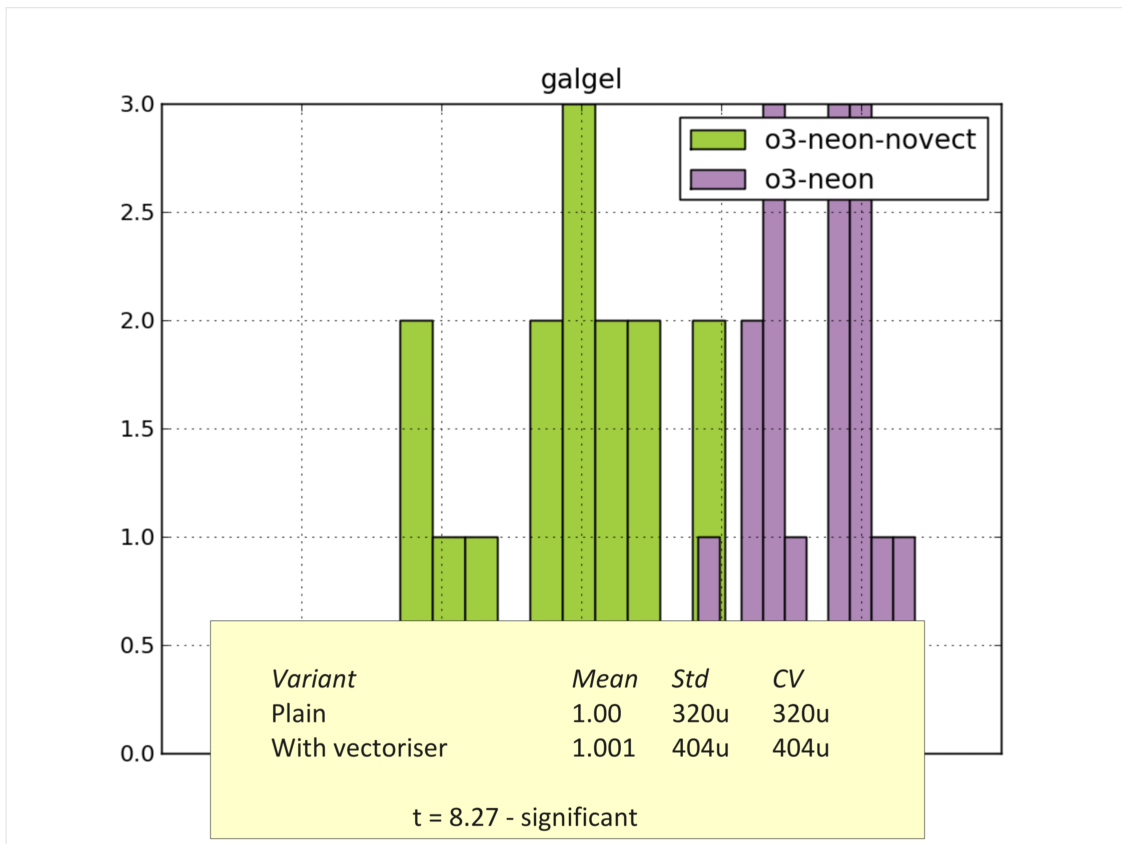


Here's a harder one.

We're working on swing-modulo scheduling which is a type of software pipelining.

Very good at hiding load/store latencies

I searched the results for a small improvement.



Here's one that's close to the noise.
Unexpected improvement due to the vectoriser
Not in a hot loop

Our tools

perf
difftest
betters
tabulate
Python
LibreOffice!

Other statistical tools like scipy.stat, R, Judge, and ministat

29

Use a range of tools
Most are homegrown
Compare two runs and automatically validate results and check for significance
Used in a benchmark run with every commit – check correctness and speed
Recommend Python for throw-away, scientific jobs. Matplotlib, numpy, scipy great for numbers and statistics.

results.csv - LibreOffice Calc

File Edit View Insert Format Tools Data Window Help

Arial 10

B149 gcc-4.6.1

	A	B	C	D	E	F	G
1	testnam	version	klas	variant	subname	nsample	min
140	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4042
141	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.42
142	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4442
143	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4462
144	spec2000	gcc-linaro-4.6-2011.11			176_gcc	5	5.4
145	spec2000	gcc-4.6.1			176_gcc	5	5.5171
146	spec2000	gcc-4.6.1			176_gcc	5	5.5227
147	spec2000	gcc-4.6.1			176_gcc	5	5.5346
148	spec2000	gcc-4.6.1			176_gcc	5	5.5491
149	spec2000	gcc-4.6.1		o3-neon	176_gcc	5	5.6
594	spec2000	gcc-linaro-4.6-2011.11		o2	254_gap	5	10.419
611	spec2000	gcc-linaro-4.6-2011.11		o3-neon	254_gap	5	10.4
647	spec2000	gcc-4.6.1		o2	254_gap	5	10.639
661	spec2000	gcc-4.6.1		o3-neon	254_gap	5	10.7
706	spec2000	gcc-linaro-4.6-2011.11		o3-neon-novect	254_gap	5	11.128
731	spec2000	gcc-linaro-4.6-2011.11		o3-vfpv3	254_gap	5	11.281
732	spec2000	gcc-4.6.1		o3-neon-novect	254_gap	5	11.281
733	spec2000	gcc-linaro-4.6-2011.11		o3	254_gap	5	11.285

All
Top 10
Standard Filter...
- empty -
- not empty -
o2
o3
o3-neon
o3-neon-novect
o3-vfpv3

http://help.libreoffice.org/Calc/Applying_AutoFilter

Example of Libreoffice
Really good at diving into similar data
Auto-filter to burrow down
Useful for
- Compare versions on one test
- Variants
- One version, variant, show best

Digging deeper: using perf

Perf is
Hooks into performance counters
Qualify what a benchmark tests (memory, I/O, etc)
No instrumentation

Perf record
Perf
Perf diff

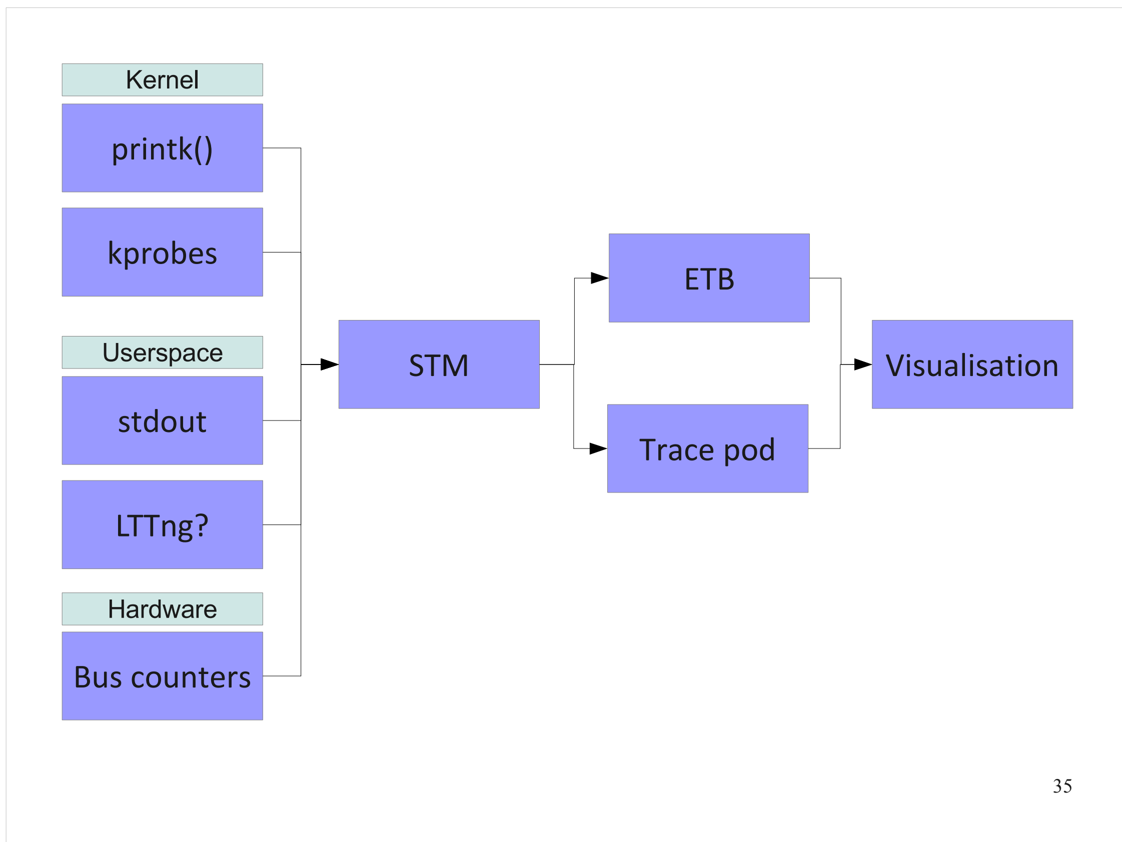
```
michaelh@leo1:~/coremark_v1.0$ perf report -n -d coremark.exe --stdio
# dso: coremark.exe
# Events: 15K cycles
#
# Overhead  Samples          Command                Symbol
# ........ ..........  ............  ......................
#
    37.96%       5980  coremark.exe  [.] core_state_transition
    27.21%       4197  coremark.exe  [.] core_bench_list
    16.25%       2522  coremark.exe  [.] matrix_test
     7.04%       1110  coremark.exe  [.] crcu32
     6.50%        931  coremark.exe  [.] crc16
     2.42%        396  coremark.exe  [.] core_bench_state
     1.70%        279  coremark.exe  [.] crcu16
     0.87%        143  coremark.exe  [.] calc_func
     0.04%          6  coremark.exe  [.] core_bench_matrix
     0.02%          3  coremark.exe  [.] main


#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

Dive into hot function

Future

Open benchmarks
Scientific benchmarks
Locking down the environment

Future: System level trace



New discussion
Started off by the system trace module included with many current cores
Low overhead, high bandwidth logging
Multiple masters, channels
Interleaves the data into a single stream

Means we can look at system level trace
Take events from...

Record them to ETB out to userspace
Or out through a pod

Then visualise.
Check relationship between code and bus load
See how the power state is related to a system level change
Visualise events and problems across cores, both ARM and others like DSP.
The design is fairly fixed.

www.linaro.org / wiki.linaro.org

people.linaro.org/~michaelh/presentations