

# seL4 microbenchmarking: designing predictable measurement context. Experimental project Kanata Bench.

by Nataliya Korovkina, volunteer for seL4  
malus.brandywine@gmail.com

## Background

Sel4bench suite (“Suite”, for future references) is a comprehensive tool which offers microbenchmarks for various seL4 mechanisms. For some benchmarks, a measured code path is considerably short and its execution time has such a magnitude that the effect of some microarchitectural events happening during the measurement period is considerable.

The work is focused on handling the influence of microarchitectural events on code execution. For purposes of this work, I would distinguish two types of influence:

- “designed”, or “permanent” influence, which is defined by benchmark design <sup>(\*)</sup> that controls how benchmark instrumental code and the code-under-test share microarchitectural resources. The designed influence manifests at every measurement across a sample, so it doesn’t impose any problem by itself. Since the influence of the instrumental code is inevitable, there can only be discussion on which design (influence scenario) suits better for a specific case.
- “random” influence, which manifests sporadically or periodically across a sample. It imposes a problem because it hinders analysis of code-under-test execution timing. Random influence can be caused by some missed or untraceable dependencies in the use of microarchitectural resources.

The case of initial caches warm-up is “designed” influence by its nature: its expected phenomena and its effect can be handled with a design decision.

The purpose of this experimental work is to *explore the ways to minimize or remove random influence* of benchmark instrumental code on execution times of code-under-test and *evaluate the suggested solution*.

The work offers (a) a specific measurement methodology, (b) a design of benchmark execution flow and (c) seL4 application implemented (a) and (b). (Reference name of the work is “Kanata Bench”, or short “KBench”).

This work uses a case of a microbenchmark supported in the Suite: delivery of Notification from Low Priority thread to High priority one (“Low prio” and “High prio”, for future use).

<sup>(\*)</sup> “benchmark design” includes both tools and methodology of metrics evaluation.

Background	1
1. The Suite measurement methodology	3
2. Kanata Bench	5
2.1 Measuring Circle	6
2.2 Benchmark Execution Flow	7
2.2.1 Analysis needs	7
2.2.2 Stability of a snapshot	8
2.2.3 Execution Flow Layout	9
3. Analysis of benchmark results	11
3.1 Analysis tools	11
3.2 L1 caches warm-up period	13
3.3 Primary latency Sample Mean and Sample Variance	16
3.4 Primary latency Mean and Variance	16
4. Kanata Bench implementation details	20
5. Concluding notes	21
5.1 Quality of the benchmark design	21
Appendix 1. Index of definitions	22
Appendix 2. Example of estimating Mean and Variance of Primary latency	23
References	26

## 1. The Suite measurement methodology

In the Suite, the measurement methodology for Notification delivery is organized the following way.

High prio thread is blocked on *Wait* system call, Low prio thread sends *Signal* system call to the proper Notification object to unblock High prio thread. The metric of interest is the time elapsed between the moment of sending signal and the moment of unblocking the waiting thread, measured in clock cycles.

The first clock reading is taken by Low prio thread right before invocation of *Signal*, the second clock reading is taken by High prio thread upon its unblocking; the difference of the 2nd and 1st readings is the latency of interest. The Suite microbenchmark conducts a required number of the latency measurements to get a sample of N measurements.

Conditions of measurement set up in the Suite microbenchmark:

- the benchmark measures the latency in the favourable circumstances: hot caches and minimum effect of user space code on measured values
- the benchmark measures and subtracts overhead associated with context switches
- the measurements are conducted with full compiler optimization (-O3)

For convenience, let's introduce the following definitions, that are applied to both the Suite and KBench methodologies:

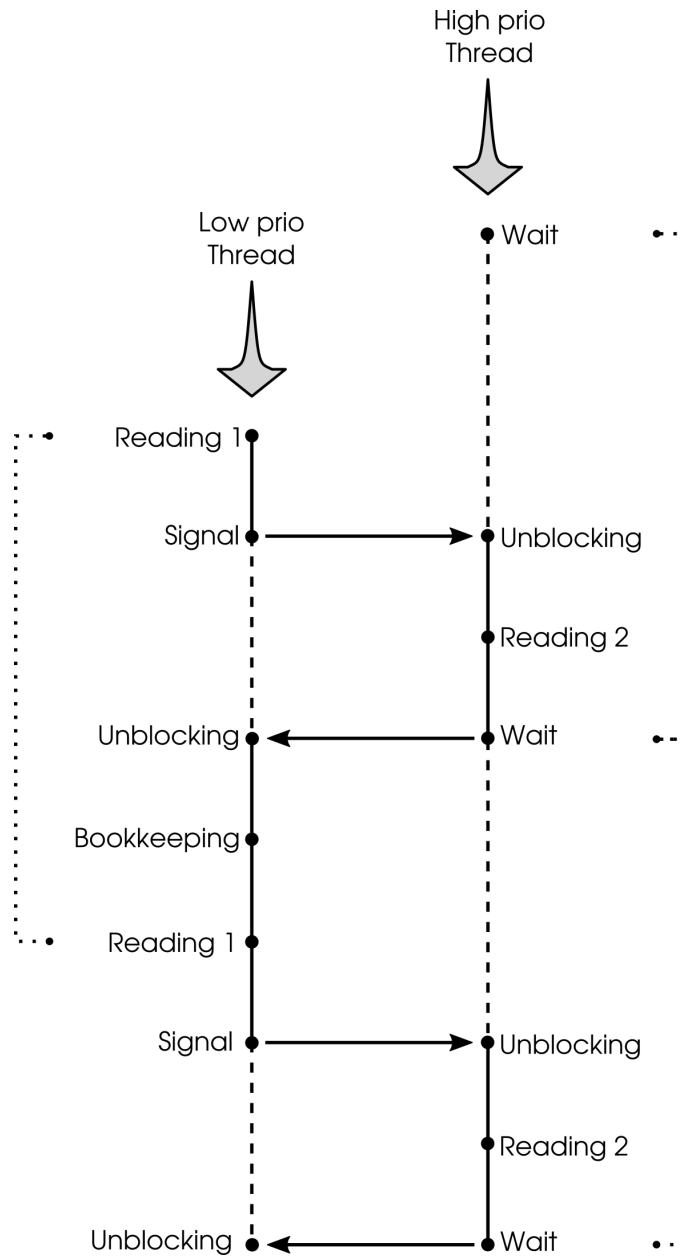
**“Measuring interval”** is the time interval between the two clock readings with which the latency of interest is measured.

**“Code-under-test”** is the code executed during Measuring interval.

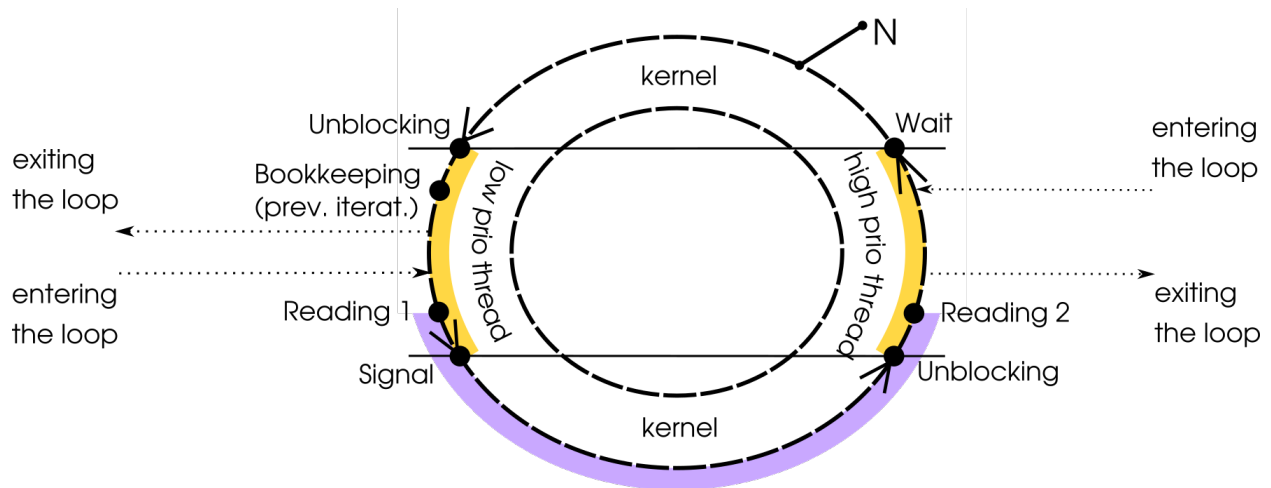
**“Measuring loop”** is the code loop (literally, *for () {}*; , *do {} while ()*; , etc), maintained by each thread to organize repeating measurements. Illustrated in Pic. 1.1

**“Measuring circle”** is the scope of the code of **user space** and **the kernel** which runs as continuous execution flow that establishes *complete repeating* measurement procedure. Illustrated in Pic 1.2.

**“Bookkeeping”** code calculates and processes latency of interest.



Pic. 1.1 Execution of measuring loops in Low Prio and High Prio threads along the timeline (top to bottom). Solid and dash lines mark "Running" and "Suspended" state of the threads respectively. Dotted line marks the boundaries of measuring loops execution.



Pic. 1.2 Organization of measuring procedure in the Suite

On the Pic. 1.2, the round wide band depicts a measuring circle that runs counterclockwise N times:

“Low prio measuring loop” (left yellow arc) → “kernel code” → “High prio measuring loop” (right yellow arc) → “kernel code”. The purple arc is a measuring interval.

Execution of every sector triggers microarchitectural events happening while the next sector along the way is executed.

In the Suite, Code-under-test is the code that is executed between the two readings: the kernel path that delivers a signal and unblocks a High prio thread and two context switches.

**Note.** In the Suite benchmark, one run along the measuring circle produces a single value of the metric of interest.

Putting aside the processes of initial D- and I-cache warm-up, we have bookkeeping code that is a prospective source of instability of measurements. When it gets even slightly complex, compiler and microarchitectural optimizations affect the following sectors of the measuring circle.

## 2. Kanata Bench

The benchmark offers two-step solution to minimize or avoid microarchitectural events randomly occurring during measuring interval.

The first step is the designing a proper measuring circle; the second one is the designing a specific execution flow.

Like the Suite, KBench sets up the following conditions:

- the benchmark measures the latency in the favourable circumstances: hot caches and minimum effect of user space code

However, KBench doesn't manage overhead associated with context switches, following the nature of its specific measuring procedure.

## 2.1 Measuring Circle

KBench introduces the measurement methodology that reduces amount of instrumental code in the measuring circle.

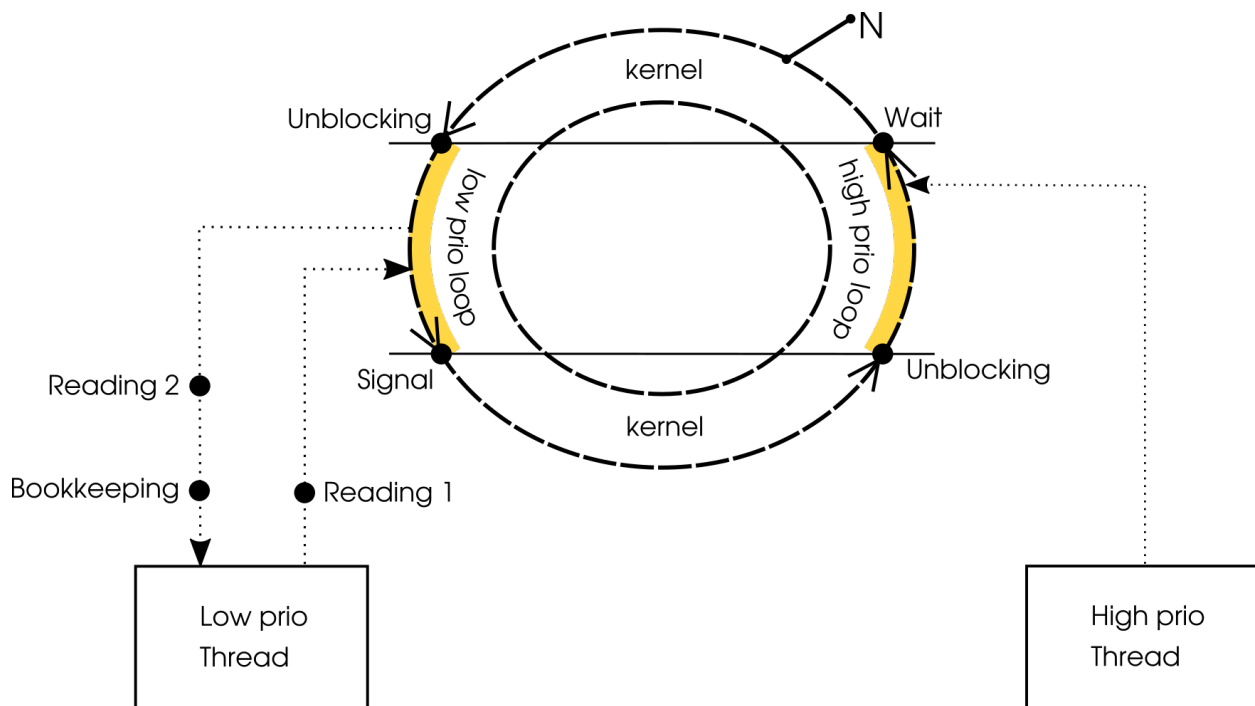
The methodology:

- sets the whole measuring circle a Code-under-test
- conducts indirect measurements — it measures accumulated latency of N measuring circle runs; such way of measurement allows to move bookkeeping code and clock counter readings outside of a single measuring circle

The illustration of the methodology is on the Pic. 2.1.

! The methodology is applicable to the cases where it is rational to evaluate the execution of the code along the whole measuring circle, not only part of it. The discussion on relevance of measured data is placed in section *Relevance/Usefulness of the data*.

! Since the methodology measures latency of interest indirectly — when individual observations are not available, the benchmark offers tools to infer estimations of important statistics and distribution parameters of measured variable using statistical methods.



Pic 2.1 KBench measuring circle

Takeaways from Pic 2.1: (a) measuring loops are “thinner” than the Suite ones; (b) clock Readings and Bookkeeping are removed outside of the circle; (c) the measuring interval is enclosed between the two clock readings and comprises N runs of measuring circle (Code-under-test) — a round wide band in the center.

KBench Low and High prio threads look in detail the following.

Low prio thread:

- reads clock counter for “Reading 1” value
- enters the loop of N exchanges with High Prio thread
- upon exiting the loop it reads clock counter for “Reading 2” value and saves difference of the readings into a global variable

High Prio thread:

- runs infinite loop of exchanges with Low Prio thread until it's stopped by the init thread

Let's make two introductions:

**“Primary latency”** is latency of execution of a Code-under-test — a latency of single run of a measuring circle — which the method doesn't measure directly.

**“N-sized Accumulated latency”** is latency of N measuring circle runs obtained with direct measurements, a sum of N Primary latency values.

## 2.2 Benchmark Execution Flow

Before drawing a diagram of the benchmark execution flow let's consider the following things: *Analysis needs* and *Stability of a snapshot*.

### 2.2.1 Analysis needs

Since the goal of the benchmark is to derive Primary latency statistics, KBench provides tools to analyze execution qualities of a Code-under-test.

KBench introduces the following concepts and parameters, the use of which will be explained in detail in section 3. “Analysis of benchmark results”.

**“N-sized Test”** is an execution of N measuring circles to get a single value of N-sized Accumulated latency.

N-sized Test can be viewed as the one that produces a sample of N size, of Primary latency, however individual values are not observable.

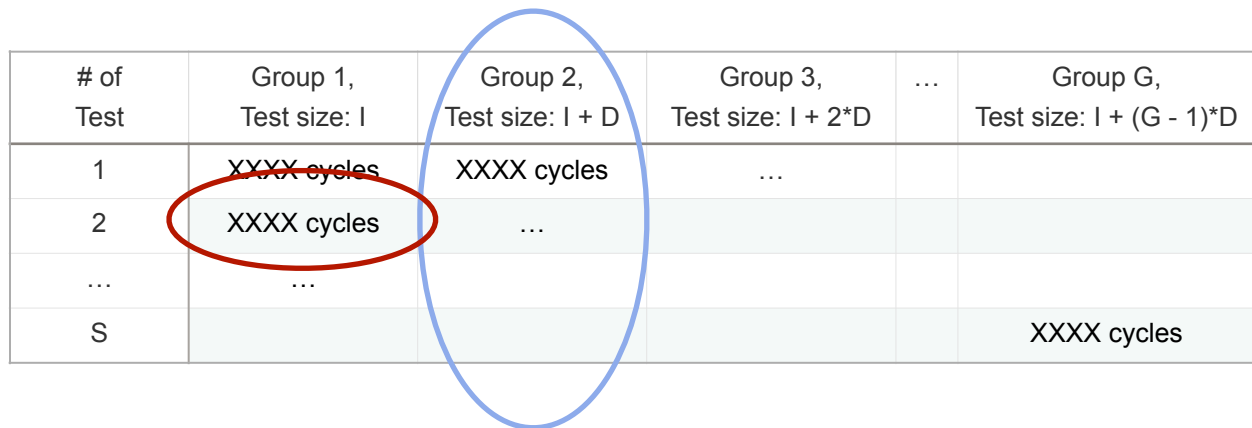
**“Group of N-sized Tests”** is a sequence of Tests that produces a sample of N-sized Accumulated latency. Let’s use “S” to refer to Group size in current configuration of the benchmark application.

KBench allows to run a series of Groups each of which has increased Test size comparing to a previous Group. Let’s use “G” to refer to number of Groups in current configuration of the benchmark application.

**“Initial Test size”** is the test size of the first Group in a sequence (“I”).

**“Delta”** is a constant that defines an increase of Test size for the next Group.

One run of KBench application produces G S-sized samples of N-sized Accumulated latencies, where N changes from I to  $I+(G-1)*D$  with step D. The output data are shown in the table 2.2



# of Test	Group 1, Test size: I	Group 2, Test size: I + D	Group 3, Test size: I + 2*D	...	Group G, Test size: I + (G - 1)*D
1	XXXX cycles	XXXX cycles	...		
2	XXXX cycles	...			
...	...				
S					XXXX cycles

Table 2.2. Data produced by a run of KBench application and representations of samples in KBench output table.

In the table 2.2:

- “I” is the Initial Test size; “D” is Delta; “G” is number of Groups.
- Red ellipse: Value XXXX is a value of I-sized Accumulated latency, represents a sample of Primary latency of size I.
- Blue ellipse circles a sample of Accumulated latency, sample size is S.

The usage of the data will be explained in section 3. “Analysis of benchmark results”.

## 2.2.2 Stability of a snapshot

Another solution to avoid random instability is to assure having the same “microarchitectural snapshot” (\*) right before the execution flow enters the measuring interval, at every measuring



iteration. To achieve this, the application should run the same (or almost the same) code at some distance before the interval. How to determine such a distance?

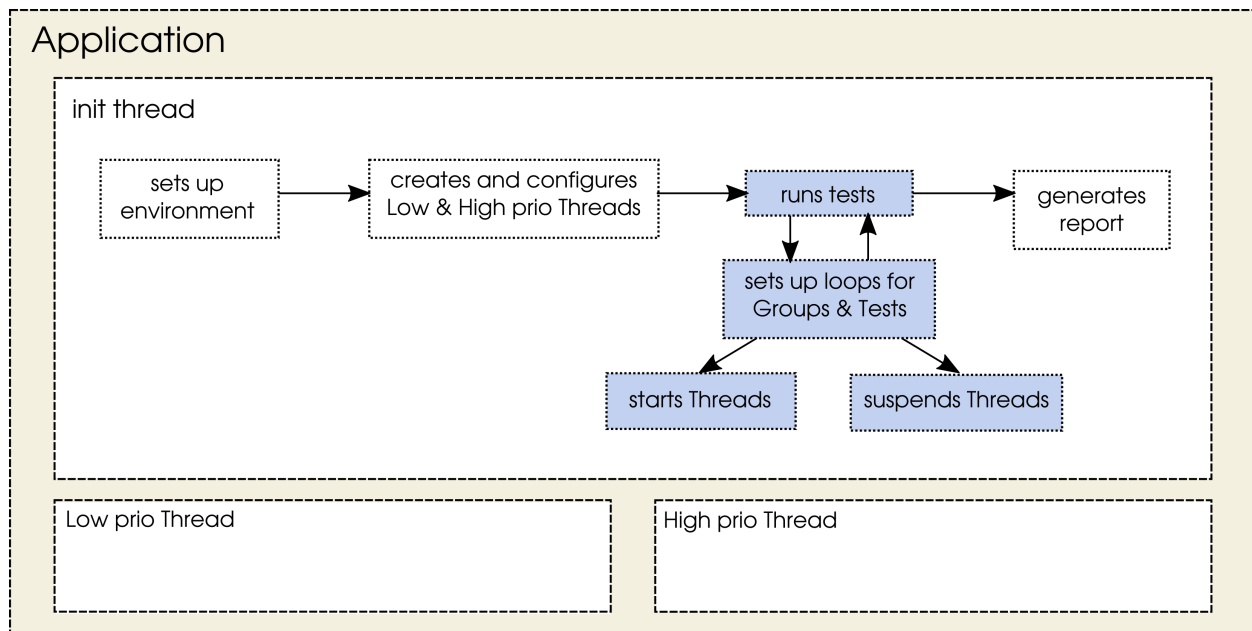
Since the benchmark runs a number of N-sized Tests in the scope of a Test Group and a number of Test Groups, the natural boundary of the distance would be the code that repeatedly runs between the Tests.

KBench reduces the amount of this code by moving as many actions needed to set up the measurement procedure as possible to the beginning of execution flow; only necessary minimum has been left to execute between the Tests.

(\*) “Snapshot” includes statuses of a pipeline (we consider one core for this use case) and all available on a platform caches in use.

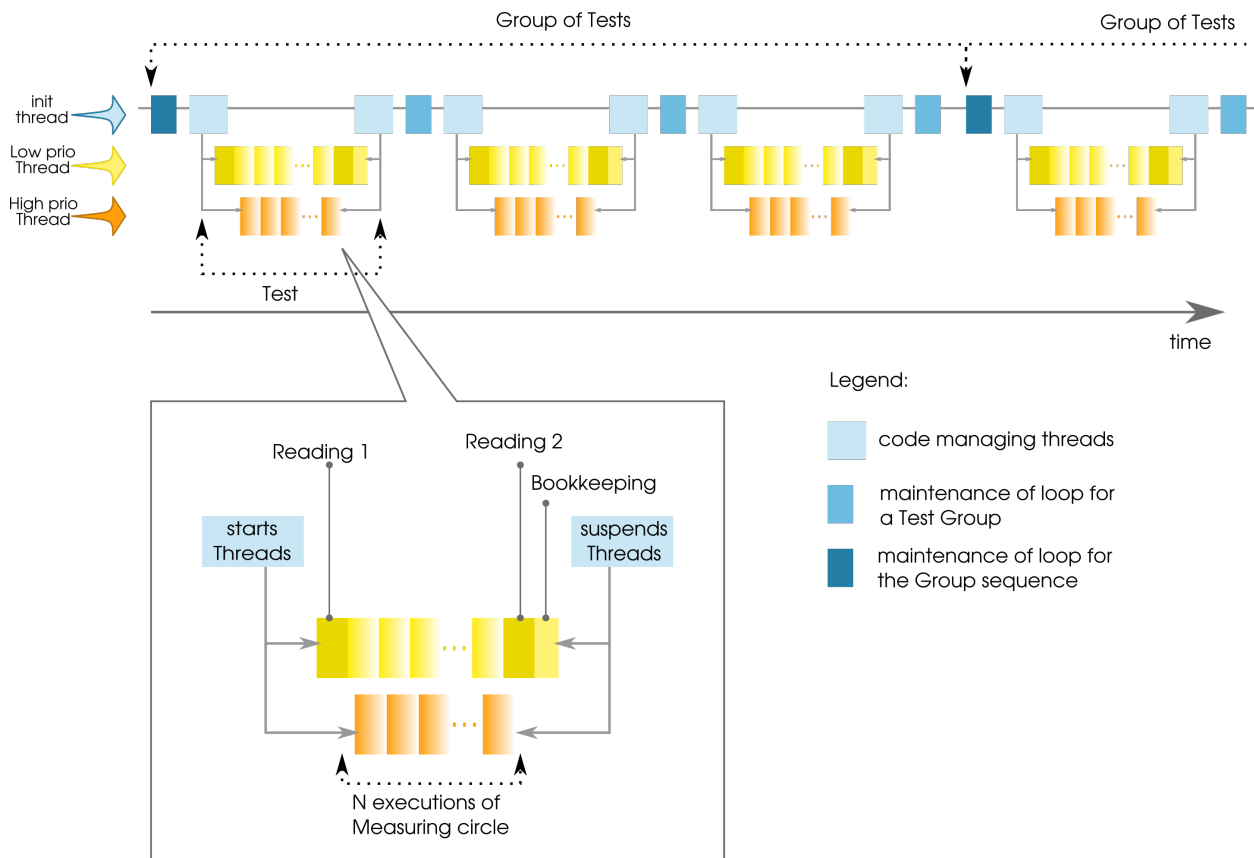
### 2.2.3 Execution Flow Layout

*Analysis needs and Stability of a snapshot* requirements define the following benchmark execution flow layout, Pic. 2.3.



Pic 2.3 KBench Execution Flow layout

The code that runs between the individual Tests is represented by lower three of the blue blocks.



Pic 2.4 Execution flow of the measuring procedure

On Pic 2.4, blue blocks belong to Initial Thread, they are executed between the Tests, yellow blocks represent Low Prio thread, Orange ones - High Prio thread.

A single run of Low and High prio threads makes an N-sized Test.

Compiler optimization brings two points into the influence stability:

- optimization regards to execution speed makes some microarchitectural events happen during a measuring interval
- optimization generates different code for measuring loops when number of Groups or Group size changes; it means the measurement tool changes from one benchmark configuration to another.

By those two reasons, compiler optimization was switched off for the parts of code that are executed immediately before Low and High prio threads start running. Four blue blocks on Pic 2.3 mark not optimized code. Low and High prio threads are fully optimized according to the requirements. Not optimized code was manually improved to reduce the use of external memory.

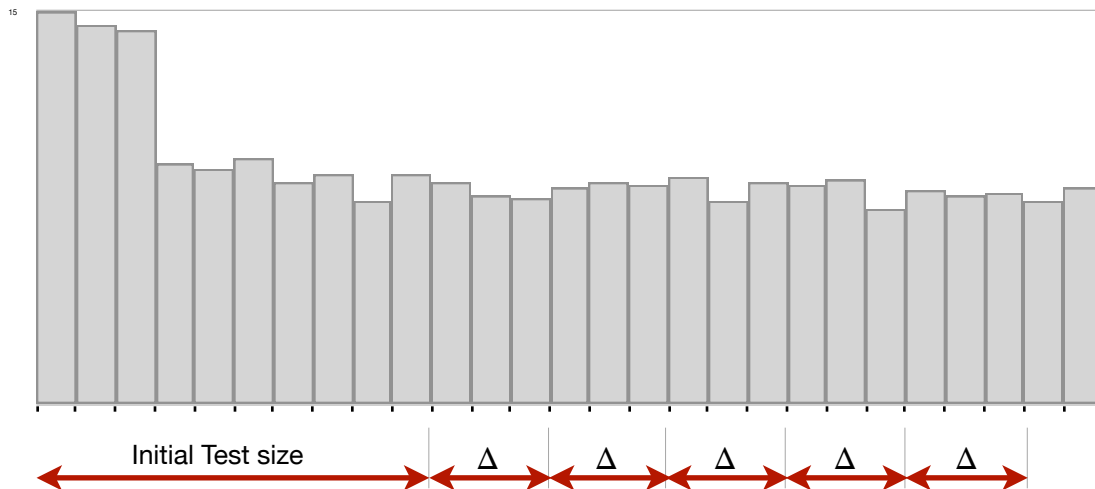
As an additional improvement, measuring loop of High prio thread becomes infinite and is stopped by the init thread. Removal of loop counter switches off in-advance loading of the instructions following the loop, into pipeline and cache.

### 3. Analysis of benchmark results

#### 3.1 Analysis tools

We can think of a sequence of Primary latency observations arranged in the order of their occurrence as a manifestation of a microarchitectural-level Process induced by execution of Code-under-test. Assuming that KBench design was targeted to make a minor influence on measured latencies, there is an opportunity to analyze the microarchitectural-level Process.

KBench provides the mechanisms and parameters to enable the analysis: Groups of Tests, sequence of Groups, Initial Test size and Delta. Relation between the parameters is shown in Pic 3.1:



Pic.3.1 Illustration of KBench parameters.

In the Pic.3.1, we see a Group of 6 Tests, Initial Test size is 10, Delta is 3, the following Tests are of sizes: 13, 16, 19, 22, 25.

Other illustration of how Primary and Accumulated latency measurements relate is presented in Table 3.2:

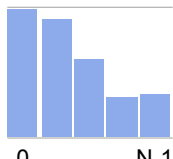
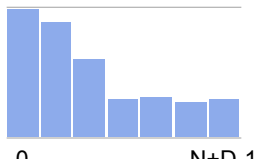
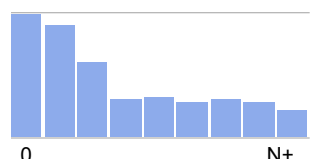
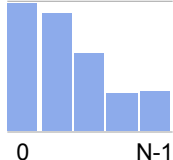
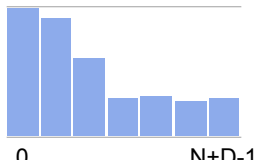
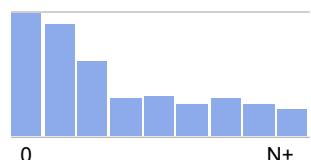
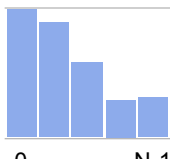
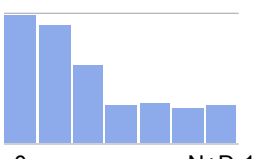
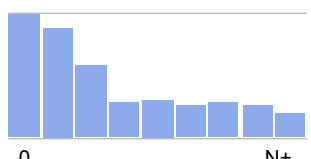
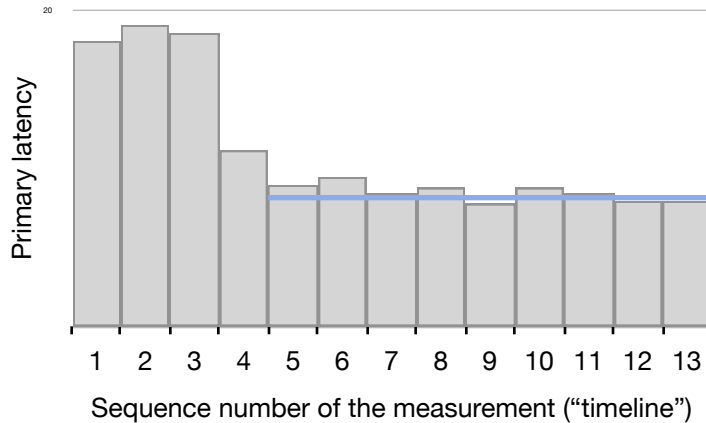
# of Test	Group 1, Test size: I	Group 2, Test size: I + D	...	Group G, Test size: I + (G - 1)*D
0	$\sum_{n=0}^{N-1} p_i$ 	$\sum_{n=0}^{N+D-1} p_i$ 	...	$\sum_{n=0}^{N+(G-1)*D-1} p_i$ 
1	$\sum_{n=0}^{N-1} p_i$ 	$\sum_{n=0}^{N+D-1} p_i$ 	...	$\sum_{n=0}^{N+(G-1)*D-1} p_i$ 
...	...	...	...	...
S-1	$\sum_{n=0}^{N-1} p_i$ 	$\sum_{n=0}^{N+D-1} p_i$ 	...	$\sum_{n=0}^{N+(G-1)*D-1} p_i$ 

Table 3.2 “Table of collected measurements”. (We will use this type of table later.)

Group of Tests (a column in the Table 3.2) generates a sample of Accumulated latency which in turn produces a sample of “Primary latency Sample Mean”.

### 3.2 L1 caches warm-up period

Let's assume a model of how Primary latencies change due to L1 caches warm-up: the first few values would be distinctively higher than succeeding ones, which would demonstrate acceptably small standard deviation. The model is illustrated in Pic.3.3.



Pic.3.3 Effect of L1 warm-up process.

Adjusting the four benchmark parameters, its possible empirically to find the point on the timeline where the Sample Mean of Primary latency stops significantly changing. Compare statistics of the measurements collected with two runs of KBench, specifically, Sample Mean of “Primary latency Sample Mean” in Tables 3.4 and 3.5 (\*).

Table 3.4 Results of KBench run (clock cycles) and some statistics.

Configuration: Initial Test size: <b>1</b> ; Delta: <b>1</b> ; Group size (S): <b>30</b> ; Number of Groups: <b>5</b>						
	S	Group 1, Test size: 1	Group 2, Test size: 2	Group 3, Test size: 3	Group 4, Test size: 4	Group 5, Test size: 5
	1	7152	9502	14610	19647	23302
	2	5567	9369	14525	19381	23537
	3	4837	9449	15093	19221	23272
	4	4768	9954	14301	18555	23374
	5	5152	9494	14011	18698	24324
	6	5483	9405	14479	19467	23043
	7	4784	9402	14705	18865	23834
	8	4923	9636	14527	18542	23926
	9	5071	9754	14606	19211	24204
	10	5212	9310	14278	20008	24012

	11	4874	9940	13845	19376	23324
	12	4686	9304	14786	18694	23701
	13	4828	10345	15269	20052	23717
	14	4916	9780	14668	18399	23876
	15	4674	9406	14622	19546	23672
	16	4724	9387	13959	19017	23171
	17	4897	9306	14302	19527	24243
	18	5245	9291	14451	18946	23288
	19	4865	9491	15186	19044	23158
	20	5438	9765	14624	18606	23324
	21	5244	9636	13860	18795	23736
	22	5197	9317	15143	18828	24093
	23	4744	9742	15043	18595	23097
	24	5065	9813	15104	19329	23529
	25	4981	9502	14708	18476	23039
	26	4892	9508	14030	18909	23094
	27	4949	9926	14289	19015	23742
	28	5346	10013	14287	18950	23335
	29	5401	9819	13916	19782	23001
	30	5114	9602	14014	18326	23516
Sample Mean		5100.97	9605.60	14508.03	19060.23	23549.47
Sample Std. Deviation		461.51	262.10	420.36	471.02	389.48
Coeff. of Variation, %		9.05	2.73	2.90	2.47	1.65
Test size		1	2	3	4	5
Sample Mean of "Primary latency Sample Mean": "Sample Mean" / "Test size" (the cells above)						
		5100.97	4802.80	4836.01	4765.06	4709.89

(\*) In this document, all the measurements were obtained with Kanata Bench application executed on the board SiFive Freedom U540 (RiscV64) in Trustworthy Systems machine pool.

Table 3.5 Results of KBench run (clock cycles) and some statistics.

Configuration: Initial Test size: <b>30</b> ; Delta: <b>1</b> ; Group size (S): <b>30</b> ; Number of Groups: <b>5</b>						
	S	Group 1, Test size: 30	Group 2, Test size: 31	Group 3, Test size: 32	Group 4, Test size: 33	Group 5, Test size: 34
	1	139790	142430	146244	151095	155805
	2	138378	141608	146696	150846	155196
	3	136312	141946	146279	150381	155252
	4	136542	141405	146581	152194	155183
	5	138943	141947	146549	151752	155279
	6	137720	141324	145543	150581	155472
	7	138556	141267	148178	151440	155607
	8	137577	141487	146528	152064	154795
	9	136392	140859	145958	151252	156663
	10	140090	143355	147103	150779	154379
	11	137438	141939	145977	151192	155122
	12	137174	141493	145910	150147	155606
	13	138110	143180	147017	151305	155831
	14	137875	142077	148121	150920	154968
	15	137454	143801	146830	150676	156012
	16	137041	142646	146784	151208	154580
	17	138010	141428	145840	151424	155961
	18	137023	142505	147955	150961	156228
	19	140616	142100	147942	151044	155330
	20	137618	141016	146698	152123	156443
	21	137097	142547	147050	150589	155966
	22	136605	141483	146658	150153	155783
	23	137401	141429	147219	153123	158056
	24	138575	142031	146191	151498	154940
	25	137328	141224	146809	151064	154679
	26	136881	143502	146101	150049	155814
	27	137264	141889	146780	151444	154162
	28	136998	141538	146480	150149	154572

	29	136442	142282	147649	152550	156569
	30	137381	142993	145825	150280	155374
Sample Mean		137687.70	142024.37	146716.50	151142.77	155520.90
Sample Std. Deviation		1074.86	765.63	708.86	748.74	799.66
Coeff. of Variation, %		0.78	0.54	0.48	0.50	0.51
Test size		30	31	32	33	34
Sample Mean of "Primary latency Sample Mean": "Sample Mean" / "Test size" (the cells above)						
		4589.59	4581.43	4584.89	4580.08	4574.14

**Note.** We can't remove the first few observations from calculations for Primary latency, but it's possible to do it for Accumulated latency. It suggests that we need to enlarge sample by the number of observations to be skipped.

### 3.3 Primary latency Sample Mean and Sample Variance

Sample Mean of Primary latency is computed by dividing a value of Accumulated latency by Test size. Group of Tests generates a sample of Accumulated latency which in turn produces a sample of Primary latency Sample Mean.

In KBench, it's impossible to compute Sample Variance of Primary latency directly because its individual observations are not available.

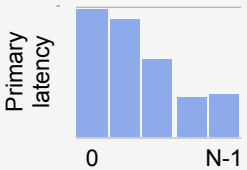
### 3.4 Primary latency Mean and Variance

KBench enables to estimate Mean and Variance of Primary latency with the following steps.

► Let's assume that random variable "Primary latency"  $P$  has Normal distribution parameters Mean  $\mu_P$  and Variance  $\sigma_P^2$ , the both are unknown.

► Value of N-sized Accumulated latency  $A^N$  represents a sample of Primary latency of size N. In the "Table of collected measurements", it occupies one cell (Pic. 3.6):



# of Test	Group 1, Test size: N	...
1	$A^N = \sum_{n=0}^{N-1} p_i$ 	
...		

Pic. 3.6

Sample Mean of this sample is  $\frac{A^N}{N}$ .

Let's introduce random variable "Primary latency Sample Mean for sample size N"  $Y$  and assume it's normally distributed with parameters Mean  $\mu_Y$  and Variance  $\sigma_Y^2$ :

$$Y = \frac{A^N}{N} \quad (3.1)$$

According to Central Limit Theorem, with large Test size N:

$$\mu_Y = \mu_P, \quad \sigma_Y^2 = \frac{\sigma_P^2}{N} \quad (3.2)$$

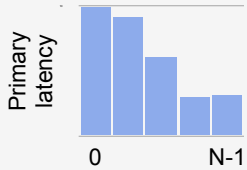
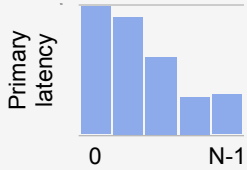
So, we can calculate  $\mu_P$  and  $\sigma_P^2$  using  $\mu_Y$  and  $\sigma_Y^2$ .

We can't determine how large N should be, because individual observations of Primary latency are unavailable. So, we can either find it empirically (like mentioned in [section 3.2](#)) or pick an a priori large number (order of hundreds, for example).

► To determine parameters  $\mu_Y$  and  $\sigma_Y^2$ , we need a sample of  $Y$  that provides desired confidence interval (CI) for  $\mu_Y$  with specific confidence level.

Let's generate a sample of N-sized Accumulated latency, of size  $S \geq 30$ . It occupies a whole column in the "Table of collected measurements" (left side of Pic 3.7).

Then, we derive a sample of  $Y$  (right side of Pic 3.7)

# of Test	Group 1, Test size: N	...
0	$a_0^N = \sum_{n=0}^{N-1} p_i$ 	
...	...	
S-1	$a_{S-1}^N = \sum_{n=0}^{N-1} p_i$ 	
	$\mu_{A,S}$	
	$\sigma_{A,S}^2$	

# of Test	Sample of Y	...
0	$y_0 = \frac{a_0^N}{N} = \frac{\sum_{n=0}^{N-1} p_i}{N}$	
...	...	
S-1	$y_{S-1} = \frac{a_{S-1}^N}{N} = \frac{\sum_{n=0}^{N-1} p_i}{N}$	
	$\mu_{Y,S}$	
	$\sigma_{Y,S}^2$	

Table 3.7

For both the samples we calculate sample means and sample variances:  $\mu_{A,S}$ ,  $\sigma_{A,S}^2$  and  $\mu_{Y,S}$ ,  $\sigma_{Y,S}^2$  (Table 3.7).

► Let's say, we want to express half-width of desired CI as a fraction  $e$  of Sample Mean  $\mu_{Y,S}$  :

$$e * \mu_{Y,S}$$

We calculate CI for  $\mu_{Y,S}$  and compare its width with the desired value.

(a) If the confidence interval has a satisfactory width, then we consider Sample Mean  $\mu_{Y,S}$  and Sample Variance  $\sigma_{Y,S}^2$  proper estimations of  $\mu_Y$  and  $\sigma_Y^2$  respectively, and move to the next step: computing  $\mu_P$  and  $\sigma_P^2$  (the next blue triangle section).

(b) If the confidence interval doesn't have the desired width, we can obtain a new one by generating a new sample of proper size  $S'$ . How to find that size?

The requirement on CI half-width was set for  $Y$ , but we firstly generate sample of  $A^N$ , so we need to translate the original requirement to requirement for sample of  $A^N$ .

Since  $Y$  is a linear combination of  $A^N$ : ( $Y = A^N / N$ ), then random variable "Accumulated latency Sample Mean for sample size  $N$ " is normally distributed with parameters:

$$\mu_A = \mu_Y * N, \quad \sigma_A^2 = \sigma_Y^2 * N^2$$

It means that widths of confidence intervals of two related samples of  $A^N$  and  $Y$  are proportional, with ratio  $N$ . So the requirement for CI half-width concerning  $Y$  translates to the same requirement for CI half-width concerning  $A^N$ : fraction  $e$  of sample mean.

The needed size of sample  $A^N$  is computed with the equation:

$$S' = \left( \frac{\sigma_{A,S} * z_{\alpha/2}}{\mu_{A,S} * e} \right)^2, \quad [1] \quad (3.3)$$

where:

$z_{\alpha/2}$  — critical value of standard Normal distribution,

$e$  — half-width of confidence interval expressed as a fraction of sample mean.

Sample Mean  $\mu_{Y,S}$  and Sample Variance  $\sigma_{Y,S}^2$  taken on a sample of size  $S'$  are the proper estimations of  $\mu_N$  and  $\sigma_N^2$ , so we can move to the next step: computing  $\mu_P$  and  $\sigma_P^2$ .

► We use equation (3.2) to compute parameters  $\mu_P$  and  $\sigma_P^2$  of "Primary latency"  $P$ :

$$\mu_P = \mu_Y, \quad \sigma_P^2 = \sigma_Y^2 * N$$

Example of such calculations is presented in Appendix 2.

## 4. Kanata Bench implementation details

Kanata Bench is a single-application (root server) project. This experimental ad-hoc application created only for the specific use case, with the resource usage limited to the necessities of the case.

It doesn't suggest any massive additions of other of benchmarks into existing code. Also, it doesn't provide any mechanisms to support a single execution flow for a number of benchmarks as a batch (like the Suite does).

The application mechanisms and configuration parameters enable varied processing scenarios of measured data. By that reason, the KBench prints out only a table of measured raw data, making no assumption on the following processing needs and, therefore, providing no additional statistics.

Description of the benchmark configuration parameters is included into the source files.

Application output is presented on Pic 4.1:

```
Jumping to kernel-image entry point...

Initial Test size: 30
Delta: 1
Number of Tests / Sample size of Accumulated latency: 30
Number of Groups: 5
Accumulated latencies (clock cycles):
138832 141131 145625 150524 155448
137993 140751 146566 151460 156124
137349 141389 146082 149993 155161
137726 142729 145533 150594 155252
137503 141790 148076 150456 155457
136981 141187 146438 150486 154729
136730 144473 145776 151365 154923
137736 141891 145586 150460 156442
138153 142825 145943 150022 156626
137220 141371 145310 150250 156503
136535 140634 145816 150574 154873
137225 141771 146375 150078 156224
138192 141929 145186 151161 155101
136805 141134 145938 150492 155967
137934 142097 146280 151305 155660
136667 142550 147028 150115 155485
137739 142260 146372 151241 154553
136260 141001 145276 151328 154527
136611 141571 146802 150768 154882
136871 140724 147264 152971 154627
136353 141440 146578 151037 155813
139577 141944 146116 150585 155671
137672 141404 146487 151744 159050
137572 142757 146552 151234 154690
136686 141324 146250 151277 155799
136520 141489 147422 152212 155257
136194 141462 145568 150911 155507
136079 142631 145578 150611 155709
136326 141427 145446 150353 157846
136727 142114 145785 151442 154971

Done!
```

## 5. Concluding notes

The solution uses indirect measurements, so it doesn't allow to compute some statistics that would be possible to accomplish if individual observations were available. However, some important statistics can be inferred within few steps, as described earlier.

Kanata Bench provides mechanisms to explore microarchitectural processes — sequence of Test Groups that are run within a single run of the application.

### 5.1 Quality of the benchmark design

[Kounev et al., 2020] lists the following desirable benchmark characteristics: Relevance, Reproducibility, Fairness, Verifiability, Usability.

*Relevance/Usefulness of the data.* In KBench methodology, relevance of obtained data depends on quality of a measuring circle. Measuring circle always includes the code of the most interest and some amount of code that maintains the circle. In well designed measuring circle, the code of the interest will make the bigger fraction. Ideally, the whole measuring circle comprises a code of interest.

Let's look at the specific case of Notification delivery. The code of the most interest is a delivery of Signal from one thread to another; KBench "adds" some code maintaining the circle — few instructions of High prio thread and some kernel code — composing the "Code-under-test".

"Microbenchmarks are often used to determine the maximum performance that would be possible if the overall system performance were limited by the performance of the respective part of the system under evaluation" [1]. In the case of KBench, we conduct measurements under almost-zero workload: caches are hot and user space code provides the same microarchitectural snapshot for every measurement iteration.

Some remarks:

- The nature of this use case (and similar microbenchmarks providing almost-zero workload) determines that the measured values can unlikely be used to infer/estimate real workload figures.
- Since there's no close analogue to seL4 API, there's probably no necessity to measure a precise code path.
- For the purpose of demonstration of a metric values magnitude, well designed measuring circle can make an affordable compromise.
- The absolute importance of the data measured with KBench lies in the fact that the values can be used to compare different versions of the same code paths.

Conclusion:

For some cases, KBench methodology can be a good compromise "closeness to the code path of interest" / "reliability of the data"

*Reproducibility.* The benchmark demonstrates run-to-run consistency for the same benchmark configuration

*Fairness.* Not applicable parameter. seL4 microbenchmarks evaluate only seL4.

*Verifiability of the results.* This document describing the methodology and KBench implementation details as well as published benchmark code, support verifiability of the results.

*Usability.* A repo manifest of KBench code guarantees ease of use.

## Appendix 1. Index of definitions

**“Measuring interval”** is the time interval between the two clock readings with which the latency of interest is measured.

**“Code-under-test”** is the code executed during Measuring interval.

**“Measuring loop”** is the code loop (literally, *for () {}*; , *do {} while ()*; , etc), maintained by each thread to organize repeating measurements.

**“Measuring circle”** is the scope of the code of **user space** and **the kernel** which runs as continuous execution flow that establishes *complete repeating* measurement procedure.

**“Bookkeeping”** code calculates and processes latency of interest.

**“Primary latency”** is latency of execution of a Code-under-test — a latency of single run of a measuring circle — which the method doesn’t measure directly.

**“N-sized Accumulated latency”** is latency of N measuring circle runs obtained with direct measurements, a sum of N Primary latency values.

**“N-sized Test”** is an execution of N measuring circles to get a single value of N-sized Accumulated latency.

**“Group of N-sized Tests”** is a sequence of Tests that produces a sample of N-sized Accumulated latency. Let’s use “S” to refer to Group size in current configuration of the benchmark application.

**“Initial Test size”** is the test size of the first Group in a sequence (“1”).

**“Delta”** is a constant which defines an increase of Test size for the next Group.

## Appendix 2. Example of estimating Mean and Variance of Primary latency

**Step 1.** We start with estimating of  $\mu_Y$  and  $\sigma_Y^2$  of  $Y$  ("Primary latency Sample Mean for sample size N").

We generate experimental sample of Accumulated latency of size 30 ( $S = 30$ ) and Test size  $N = 300$ , and compute Sample Mean  $\mu_{A,S}$  and Sample Variance  $\sigma_{A,S}^2$ .

For illustration purposes, let's run KBench three times.

Table A.2.1 Three runs of the benchmark: Accumulated latency measurements and statistics.

Accumulated latency (clock cycles), Test size: 300, sample size: 30						
	S	Run 1		Run 2		Run 3
	1	1363962		1365676		1364709
	2	1366266		1362773		1362914
	3	1361963		1365105		1362488
	4	1364586		1364055		1366801
	5	1360804		1360190		1364767
	6	1362537		1358666		1360557
	7	1362877		1361368		1361472
	8	1360579		1365811		1360008
	9	1363375		1362925		1360045
	10	1366395		1364722		1363121
	11	1353482		1360439		1364351
	12	1361213		1360139		1361531
	13	1362684		1361834		1360347
	14	1361353		1361285		1365487
	15	1363735		1360507		1362036
	16	1360329		1365372		1361369
	17	1362563		1359406		1359720
	18	1362151		1363075		1359068
	19	1362097		1362774		1358835
	20	1360104		1359971		1364051
	21	1362359		1361222		1363167
	22	1361991		1366397		1363887

	23	1363161		1360466		1360291
	24	1361317		1364495		1363993
	25	1362731		1364785		1364412
	26	1364707		1367475		1362851
	27	1360280		1359687		1360267
	28	1356589		1362517		1353650
	29	1361992		1362510		1363893
	30	1361451		1354348		1359742
Mean, $\mu_{A,S}$		1361987.77		1362333.17		1361994.33
Variance, $\sigma_{A,S}^2$		6227560.94		7777320.14		6874037.68

**Step 2.** Let's derive a sample of  $Y$  from the sample received in the Step 1. To do so, we divide measurements in Table A.2.1 by Test size  $N$ .

Then we calculate Sample Mean  $\mu_{Y,S}$ , Sample Variance  $\sigma_{Y,S}^2$  and confidence interval.

For computing confidence interval we assume that required confidence level is 90% (for which  $z_{\alpha/2} = 1.645$ ), required maximum half-width of confidence interval is 2% of a sample mean ( $e = 0.02$ , for equation (3.3)).

Table A.2.2 Three runs of the benchmark: measurements of  $Y$  and statistics.

Primary latency Sample Mean (clock cycles), Test size: 300, sample size: 30						
	S	Run 1		Run 2		Run 3
	1	4546.54		4552.25		4549.03
	2	4554.22		4542.58		4543.05
	3	4539.88		4550.35		4541.63
	4	4548.62		4546.85		4556.00
	5	4536.01		4533.97		4549.22
	6	4541.79		4528.89		4535.19
	7	4542.92		4537.89		4538.24
	8	4535.26		4552.70		4533.36
	9	4544.58		4543.08		4533.48



	10	4554.65		4549.07		4543.74
	11	4511.61		4534.80		4547.84
	12	4537.38		4533.80		4538.44
	13	4542.28		4539.45		4534.49
	14	4537.84		4537.62		4551.62
	15	4545.78		4535.02		4540.12
	16	4534.43		4551.24		4537.90
	17	4541.88		4531.35		4532.40
	18	4540.50		4543.58		4530.23
	19	4540.32		4542.58		4529.45
	20	4533.68		4533.24		4546.84
	21	4541.20		4537.41		4543.89
	22	4539.97		4554.66		4546.29
	23	4543.87		4534.89		4534.30
	24	4537.72		4548.32		4546.64
	25	4542.44		4549.28		4548.04
	26	4549.02		4558.25		4542.84
	27	4534.27		4532.29		4534.22
	28	4521.96		4541.72		4512.17
	29	4539.97		4541.70		4546.31
	30	4538.17		4514.49		4532.47
<hr/>						
Mean, $\mu_{Y,S}$		4539.96		4541.11		4539.98
Variance, $\sigma_{Y,S}^2$		69.20		86.41		76.38
Std. Deviation, $\sigma_{Y,S}$		8.32		9.30		8.74
Coeff. of Variation, %		0.18		0.20		0.19
Confidence interval (c1, c2), Confidence level 90%, $z_{\alpha/2} = 1.645$ , N = 30						
c1		4537.46		4538.32		4537.36
c2		4542.46		4543.90		4542.61
CI half-width, % of sample mean		0.06		0.06		0.06

Half-width of the confidence interval is less than 2% for all three runs, so we don't need to generate larger sample to get better precision.

**Step 3.** We use  $\mu_{Y,S}$  and  $\sigma^2_{Y,S}$  to compute Primary latency statistics using equation (3.2).

Table A.2.3 Statistics of Primary latency, three benchmark runs

		Run 1		Run 2		Run 3
Mean, $\mu_P = \mu_Y$		4539.96		4541.11		4539.98
Variance, $\sigma_Y^2$		69.20		86.41		76.38
Variance, $\sigma_P^2 = \sigma_Y^2 * N$		20758.54		25924.40		22913.46
Std. Deviation, $\sigma_P$		144.08		161.01		151.37
Coeff. of Variation, %		3.17		3.55		3.33

## References

[1] Kounev, Samuel; Lange, Klaus-Dieter; von Kistowski, Jóakim. Systems Benchmarking For Scientists and Engineers. Springer International Publishing. Kindle Edition. (2020)

November 18, 2022