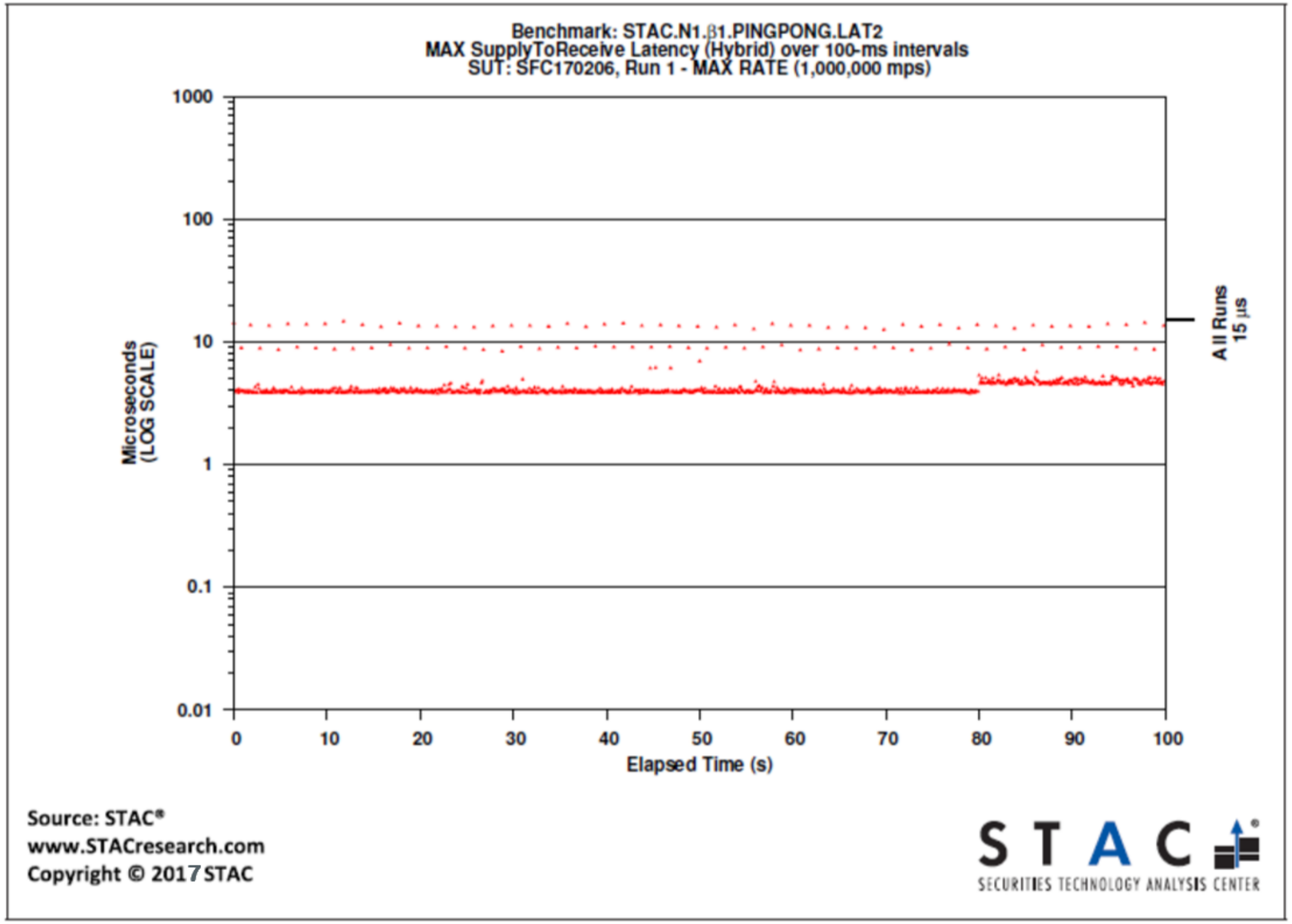
A BETTER WAY TO ANALYZE LATENCY OUTLIERS IN SOFTWARE

Authors: Stanislav Bratanov, Artyom Shatalin, Vasily Starikov, Ilia Kurakin,
Sergey Vinogradov

AGENDA

- Background on latency outliers and other performance anomalies
- Point of view: The best approach to anomaly explanation (hardware and software requirements)
- Illustrate the approach through a case study using Intel® Processor Trace (PT) and Intel® VTune™ Profiler
  - Using ITT API for marking performance-critical code region
  - Analyzing Latency histogram
  - Analyzing context switch induced anomalies
  - Analyzing kernel induced anomalies
  - Analyzing control flow deviations
  - Analyzing CPU frequency

THE BIG CONCERN IN TRADING: LATENCY OUTLIERS



Source: STAC®
www.STACresearch.com
Copyright © 2017 STAC

LATENCY OUTLIERS $\in$ PERFORMANCE ANOMALIES:

Any short-lived sporadic issue that causes unrecoverable consequences

- UX glitch – slow/skipped video frames, failed image tracking
- Unexpectedly long financial transaction
- Long network packet processing/lost packets

Those issues are not visible to traditional sampling-based methods, but

- Cost money and reputation

TYPICAL CAUSES OF PERFORMANCE ANOMALIES

- Control flow change
  - Different amount of work done by different instances of the same task
  - Expensive handling of errors or other rare-happening situations, like memory/storage reallocation
- Context switches – synchronization or preemption
- Unexpected kernel activity – interrupts, page faults, etc.
- Micro-architectural issues – cache misses, branch misprediction, etc.
- Frequency drops– low CPU utilization, cooling issues, AVX instructions, etc.

WHAT'S REQUIRED TO ANALYZE ANOMALIES?

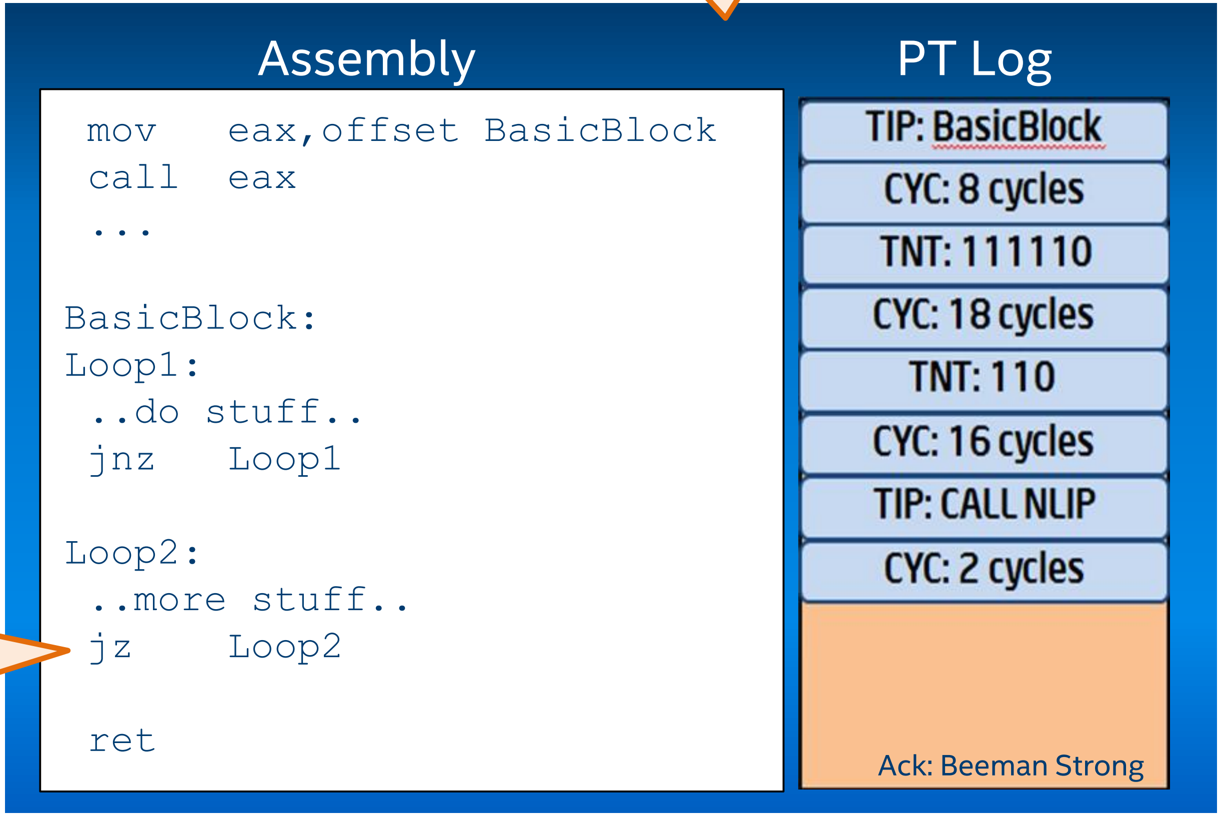
- Extremely granular information from the processor
  - Branching, timing, and frequency info logged at nanosecond level
    - Make sure your CPU supports this!
- A way to analyze the resulting information
  - Locate and explain performance deviations in critical code regions

INTEL CPUS NOW PROVIDE THE DATA

Intel® Processor Trace, *production quality since* ***Skylake***

- HW means to trace branching, transaction, and timing info in a highly-compressed, low-overhead manner
  - To be extended in the future with more info to enrich the picture of SW behavior

PT differentiates between processes, but not SW threads. SW *tools need to take care of that*



Assembly

```
mov    eax,offset BasicBlock
call   eax
...

BasicBlock:
Loop1:
 ..do stuff..
 jnz    Loop1

Loop2:
 ..more stuff..
 jz     Loop2

 ret
```

PT Log

```
TIP: BasicBlock
CYC: 8 cycles
TNT: 111110
CYC: 18 cycles
TNT: 110
CYC: 16 cycles
TIP: CALL NLIP
CYC: 2 cycles
```

Ack: Beeman Strong

PT saves information on conditional and indirect branches only.
The rest to be found by static analysis of disassembly to decode PT data stream correctly

WHAT ABOUT ANALYZING THE DATA?

- Intel PT provides an overwhelming amount of data
- Tools like Linux perf collect control flow and timing for a given time interval, but…
  - Users must dig through GBs of data to figure out what may be going wrong

A BETTER WAY WOULD BE (1/2):

- Collect PT data only for a specified process or set of processes
  - To minimize amount of data traced and data loss
- Analyze (or even collect) data at a thread level
  - Intel PT only differentiates by process, we need OS scheduling info
- Incorporate performance monitoring data
  - To get clues as to what’s happening at the microarchitecture level

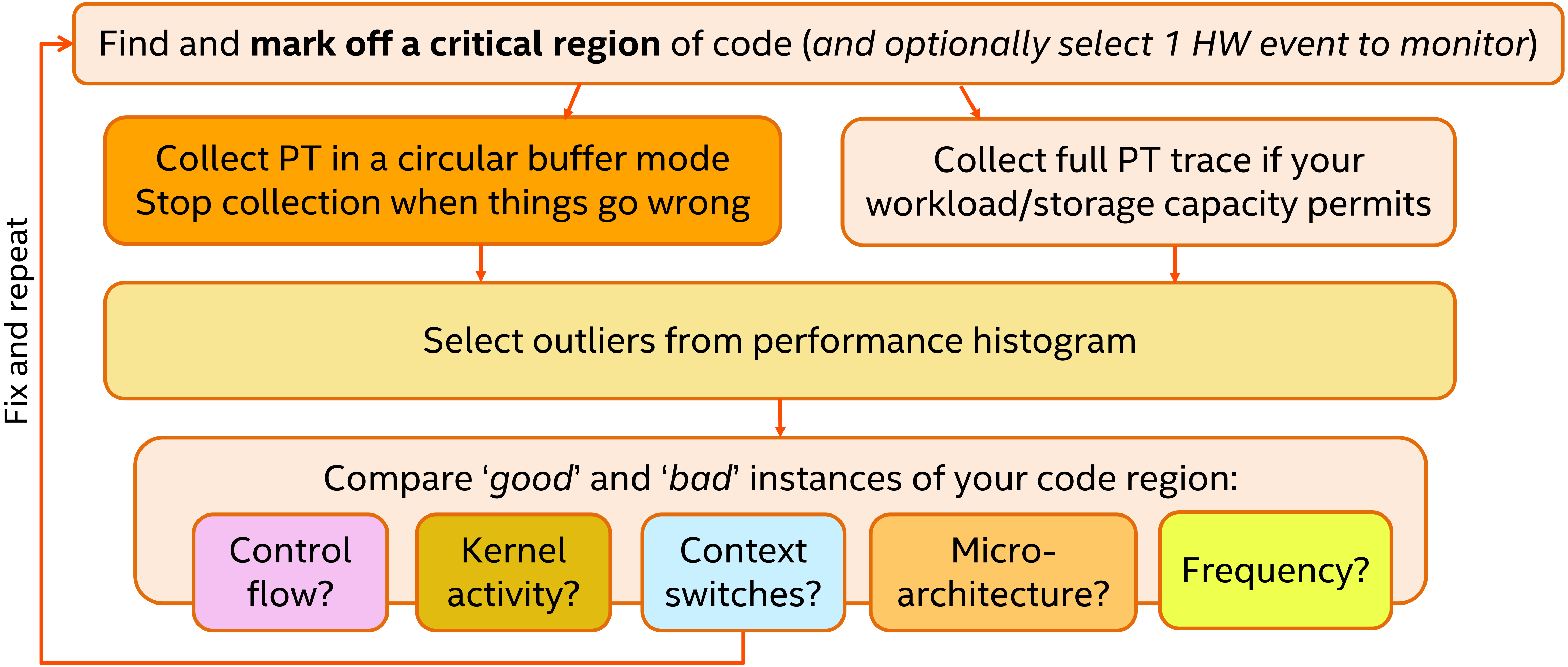
A BETTER WAY WOULD BE (2/2):

- Mark off code regions of interest via a lightweight instrumentation API
  - Ideally, have HW instrumentation support, not to trace outside of regions
- Analyze control flow and timings only for the marked off code regions
  - To minimize data post-processing times and guide users to issue areas
- Categorize types of issues users may encounter and provide further guidance
  - Ideally, automate comparison between known ‘good’ and ‘bad’ code instances

VTUNE™ STRIVES TO FOLLOW THAT APPROACH:

Requirement	VTUNE	Comment
Collect PT only for specified processes	✓	
Analyze data at the thread level	✓	Turns collection on/off when a thread is scheduled on/off CPU
Mark off code regions of interest	Via SW API, up to 128 regions	~10 cycles per API call
Incorporate perfmon data	1 perfmon event per region	~300 cycles per region, need HW support to do better
Analyze data only for the marked off regions	✓	
Categorize issues and automate comparison	Side-by-side comparison	Working on enriched timeline, and disassembly

ANOMALY DETECTION METHODOLOGY IN VTUNE™



Find and **mark off a critical region** of code (*and optionally select 1 HW event to monitor*)

Collect PT in a circular buffer mode
Stop collection when things go wrong

Collect full PT trace if your workload/storage capacity permits

Select outliers from performance histogram

Compare ‘*good*’ and ‘*bad*’ instances of your code region:

- Control flow?
- Kernel activity?
- Context switches?
- Micro-architecture?
- Frequency?

Fix and repeat

CASE STUDY – ANALYZING PERFORMANCE ANOMALIES

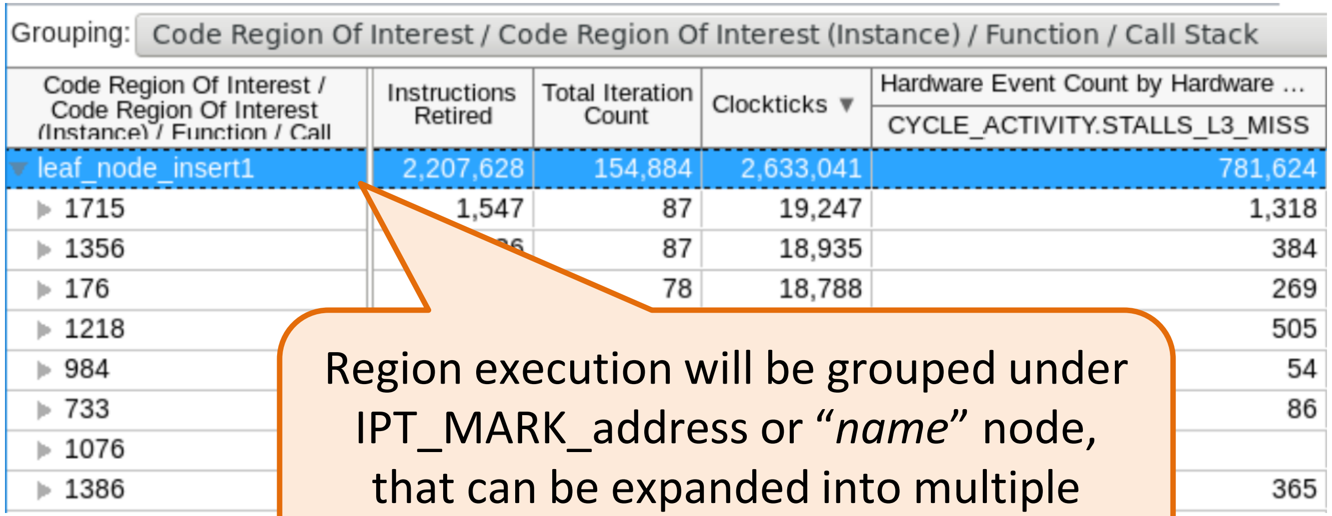
- We are going to analyze performance anomalies on Pelikan – unified cache backend by Twitter https://github.com/twitter/pelikan
- We run a client-server benchmark, which sends put/get requests over network, and analyze full request handling flow on the server side
  - Receiving/decoding a request
  - Processing a request
  - Sending a response
- We'll find outliers among ~600 000 requests and investigate the reasons of performance anomalies using Anomaly Detection methodology in VTune™

ITT API TO MARK OFF PERFORMANCE-CRITICAL TASK

Using ITT PTMARK API:

```
__itt_pt_region region = __itt_pt_region_create("name");
for(…;…;…)
{
 __itt_mark_pt_region_begin(region);
 … code, processing your task …
 __itt_mark_pt_region_end(region);
}
```

Begin/end API is directly registered by Intel PT HW, w/o intermediate trace files, time-based correlation hassle, etc.



Grouping: Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack

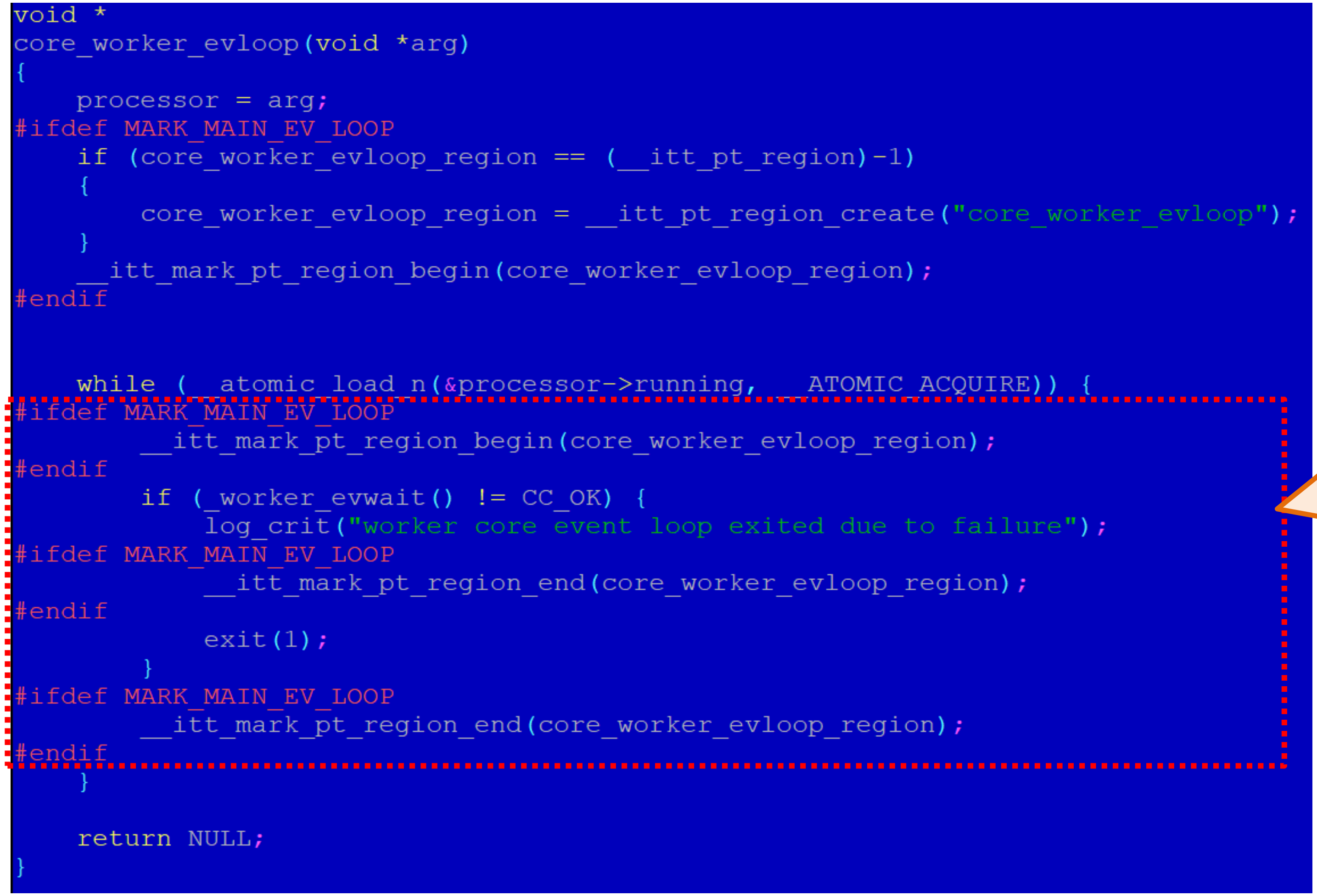
Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call	Instructions Retired	Total Iteration Count	Clockticks ▼	Hardware Event Count by Hardware …
				CYCLE_ACTIVITY.STALLS_L3_MISS
▼ leaf_node_insert1	2,207,628	154,884	2,633,041	781,624
▶ 1715	1,547	87	19,247	1,318
▶ 1356		87	18,935	384
▶ 176		78	18,788	269
▶ 1218				505
▶ 984				54
▶ 733				86
▶ 1076				
▶ 1386				365

Region execution will be grouped under IPT_MARK_address or "*name*" node, that can be expanded into multiple invocations/iterations (if any)

```
__itt_detach();
```

Call detach API to stop collection and get a snapshot of PT data

ITT API TO MARK OFF PERFORMANCE-CRITICAL TASK

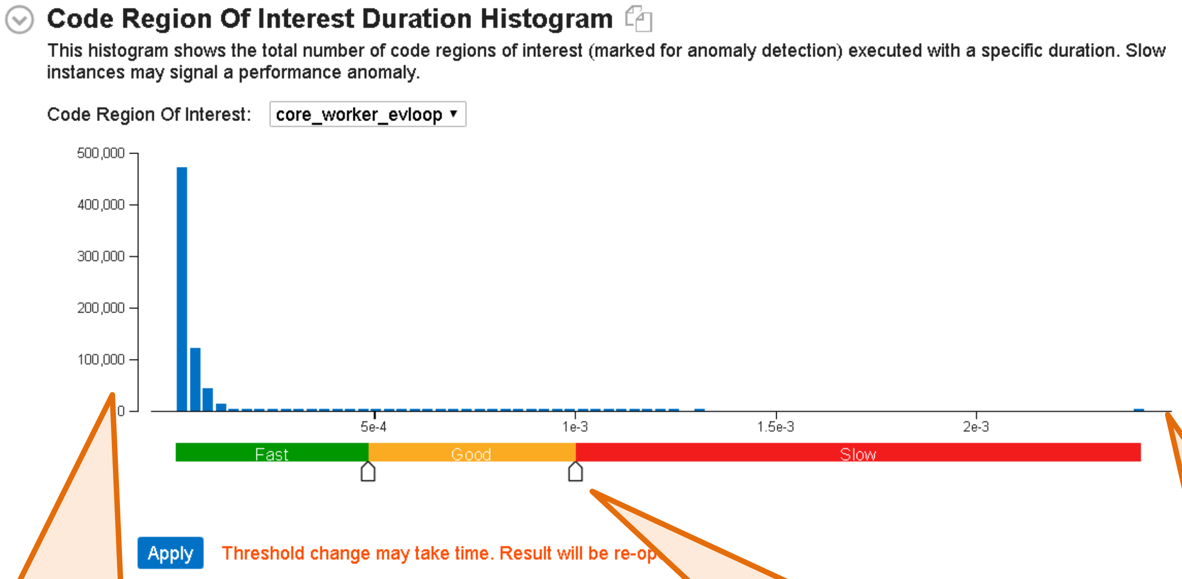


```
void *
core_worker_evloop(void *arg)
{
    processor = arg;
#ifdef MARK_MAIN_EV_LOOP
    if (core_worker_evloop_region == (__itt_pt_region)-1)
    {
        core_worker_evloop_region = __itt_pt_region_create("core_worker_evloop");
    }
    __itt_mark_pt_region_begin(core_worker_evloop_region);
#endif


    while (__atomic_load_n(&processor->running, __ATOMIC_ACQUIRE)) {
#ifdef MARK_MAIN_EV_LOOP
        __itt_mark_pt_region_begin(core_worker_evloop_region);
#endif
        if (_worker_evwait() != CC_OK) {
            log_crit("worker core event loop exited due to failure");
#ifdef MARK_MAIN_EV_LOOP
            __itt_mark_pt_region_end(core_worker_evloop_region);
#endif
            exit(1);
        }
#ifdef MARK_MAIN_EV_LOOP
        __itt_mark_pt_region_end(core_worker_evloop_region);
#endif
    }

    return NULL;
}
```

In our example, a **single iteration** of the request processing loop **is a performance critical task**.

Let’s mark it off and **see what is happening inside individual iterations** which run longer than expected.

ANALYZING LATENCY HISTOGRAM



Code Region Of Interest Duration Histogram

This histogram shows the total number of code regions of interest (marked for anomaly detection) executed with a specific duration. Slow instances may signal a performance anomaly.

Code Region Of Interest: core_worker_evloop ▾

Threshold change may take time. Result will be re-op

The histogram shows how many instances of a performance-critical task had which duration (which we also call ‘latency’)

Y axis shows the number of instances within a given latency bin

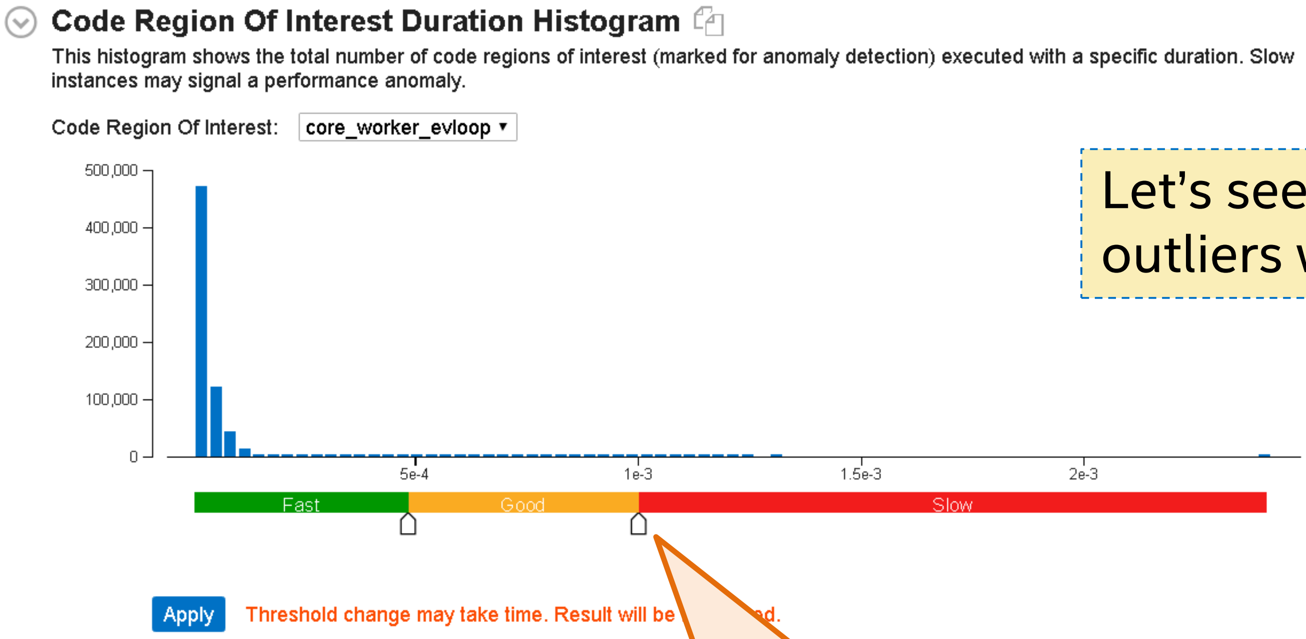
Fast/Good/Slow boundaries are adjustable

X axis shows task instance latency in seconds

ANALYZING LATENCY HISTOGRAM

Most of the requests take less than 100 µs to process, but there are outliers up to 2400 µs

Let's see what's happening inside outliers with >1000 µs latency



Code Region Of Interest Duration Histogram

This histogram shows the total number of code regions of interest (marked for anomaly detection) executed with a specific duration. Slow instances may signal a performance anomaly.

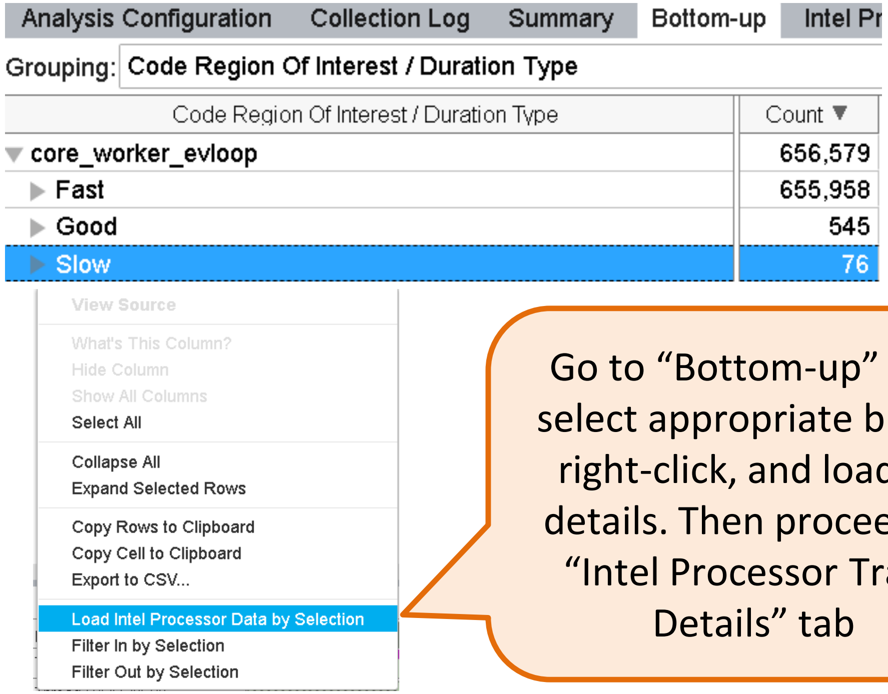
Code Region Of Interest: core_worker_evloop

Threshold change may take time. Result will be

d.

Boundary adjustments need to be applied

We Set slow boundary to ~1000 µs



Grouping: Code Region Of Interest / Duration Type

Code Region Of Interest / Duration Type	Count ▼
core_worker_evloop	656,579
Fast	655,958
Good	545
Slow	76

Go to "Bottom-up" tab, select appropriate bin(s), right-click, and load PT details. Then proceed to "Intel Processor Trace Details" tab

PROCESSOR TRACE DETAILS – MAIN VIEW

Allows **side-by-side comparison of individual instances** of marked code regions annotated with metrics, which helps to detect different types of anomalies

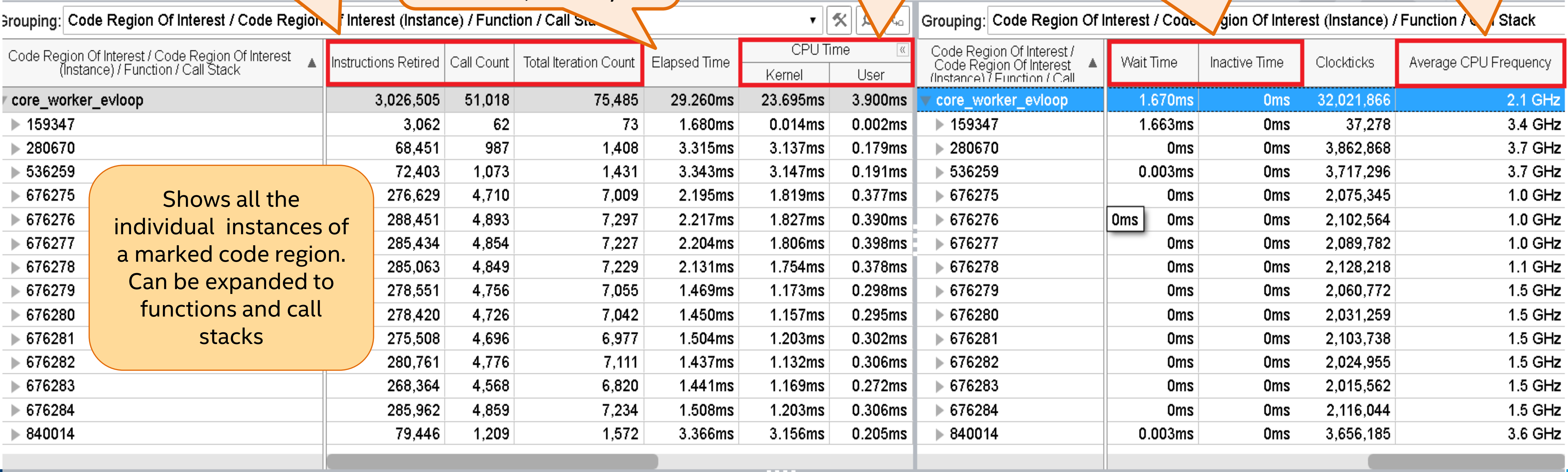
Control flow metrics

Wall-clock time of the code region execution, that is, Latency

Active time on CPU split into Kernel and User

Time a thread was idle because of synchronization or preemption

Average CPU frequency the code region was executed at



Grouping: Code Region Of Interest / Code Region f Interest (Instance) / Function / Call Sta

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time Kernel	CPU Time User
core_worker_evloop	3,026,505	51,018	75,485	29.260ms	23.695ms	3.900ms
159347	3,062	62	73	1.680ms	0.014ms	0.002ms
280670	68,451	987	1,408	3.315ms	3.137ms	0.179ms
536259	72,403	1,073	1,431	3.343ms	3.147ms	0.191ms
676275	276,629	4,710	7,009	2.195ms	1.819ms	0.377ms
676276	288,451	4,893	7,297	2.217ms	1.827ms	0.390ms
676277	285,434	4,854	7,227	2.204ms	1.806ms	0.398ms
676278	285,063	4,849	7,229	2.131ms	1.754ms	0.378ms
676279	278,551	4,756	7,055	1.469ms	1.173ms	0.298ms
676280	278,420	4,726	7,042	1.450ms	1.157ms	0.295ms
676281	275,508	4,696	6,977	1.504ms	1.203ms	0.302ms
676282	280,761	4,776	7,111	1.437ms	1.132ms	0.306ms
676283	268,364	4,568	6,820	1.441ms	1.169ms	0.272ms
676284	285,962	4,859	7,234	1.508ms	1.203ms	0.306ms
840014	79,446	1,209	1,572	3.366ms	3.156ms	0.205ms

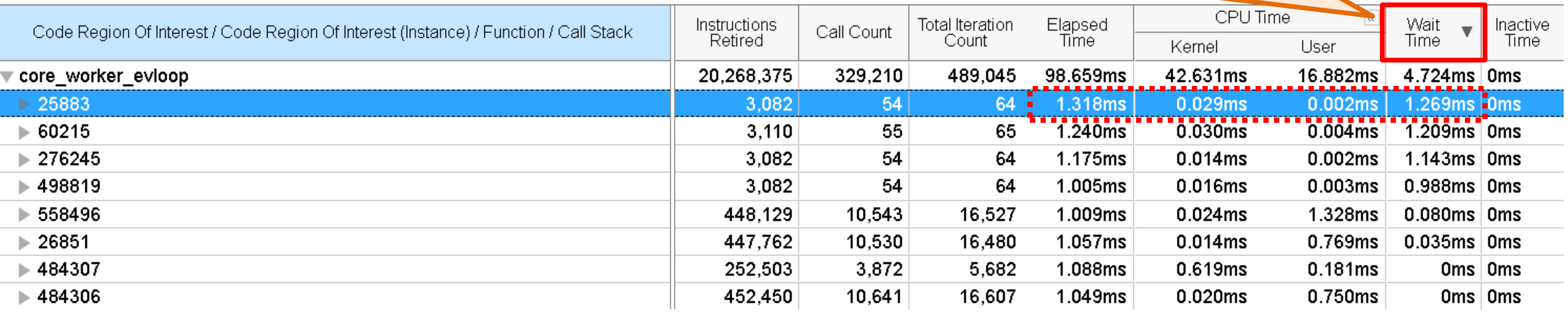
Grouping: Code Region Of Interest / Code gion Of Interest (Instance) / Function / Stack

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call	Wait Time	Inactive Time	Clockticks	Average CPU Frequency
core_worker_evloop	1.670ms	0ms	32,021,866	2.1 GHz
159347	1.663ms	0ms	37,278	3.4 GHz
280670	0ms	0ms	3,862,868	3.7 GHz
536259	0.003ms	0ms	3,717,296	3.7 GHz
676275	0ms	0ms	2,075,345	1.0 GHz
676276	0ms 0ms	0ms	2,102,564	1.0 GHz
676277	0ms	0ms	2,089,782	1.0 GHz
676278	0ms	0ms	2,128,218	1.1 GHz
676279	0ms	0ms	2,060,772	1.5 GHz
676280	0ms	0ms	2,031,259	1.5 GHz
676281	0ms	0ms	2,103,738	1.5 GHz
676282	0ms	0ms	2,024,955	1.5 GHz
676283	0ms	0ms	2,015,562	1.5 GHz
676284	0ms	0ms	2,116,044	1.5 GHz
840014	0.003ms	0ms	3,656,185	3.6 GHz

Shows all the individual instances of a marked code region. Can be expanded to functions and call stacks

ANALYZING CONTEXT SWITCH INDUCED ANOMALIES

Sort by **Wait Time** metric which is thread idle time due to synchronization

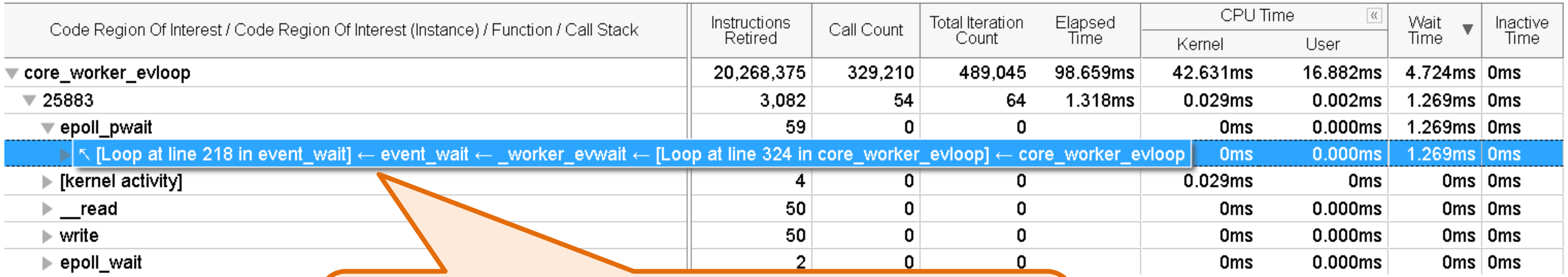


Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time ▼	Inactive Time
					Kernel	User		
core_worker_evloop	20,268,375	329,210	489,045	98.659ms	42.631ms	16.882ms	4.724ms	0ms
25883	3,082	54	64	1.318ms	0.029ms	0.002ms	1.269ms	0ms
60215	3,110	55	65	1.240ms	0.030ms	0.004ms	1.209ms	0ms
276245	3,082	54	64	1.175ms	0.014ms	0.002ms	1.143ms	0ms
498819	3,082	54	64	1.005ms	0.016ms	0.003ms	0.988ms	0ms
558496	448,129	10,543	16,527	1.009ms	0.024ms	1.328ms	0.080ms	0ms
26851	447,762	10,530	16,480	1.057ms	0.014ms	0.769ms	0.035ms	0ms
484307	252,503	3,872	5,682	1.088ms	0.619ms	0.181ms	0ms	0ms
484306	452,450	10,641	16,607	1.049ms	0.020ms	0.750ms	0ms	0ms

Significant time *(1.269 out of 1.318 ms)* spent in idle because of *synchronization context switches*

Thread moved to Idle from a *polling loop waiting for requests*

There are not enough requests in the queue!

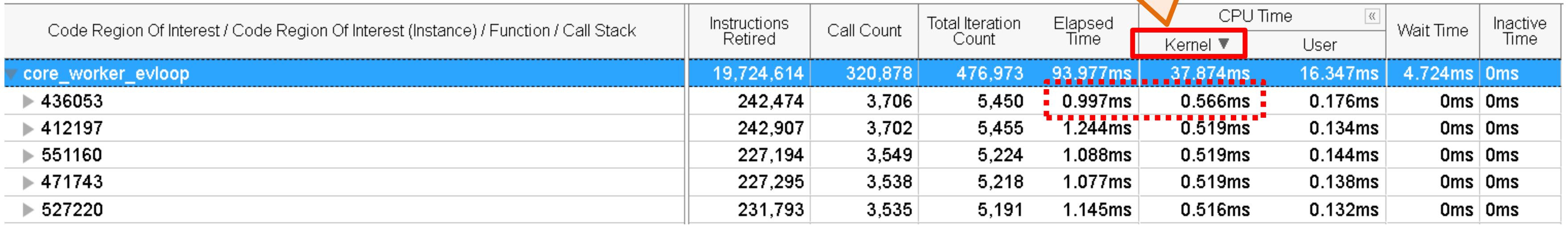


Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time ▼	Inactive Time
					Kernel	User		
core_worker_evloop	20,268,375	329,210	489,045	98.659ms	42.631ms	16.882ms	4.724ms	0ms
25883	3,082	54	64	1.318ms	0.029ms	0.002ms	1.269ms	0ms
epoll_pwait	59	0	0		0ms	0.000ms	1.269ms	0ms
↖ [Loop at line 218 in event_wait] ← event_wait ← _worker_evwait ← [Loop at line 324 in core_worker_evloop] ← core_worker_evloop					0ms	0.000ms	1.269ms	0ms
[kernel activity]	4	0	0		0.029ms	0ms	0ms	0ms
__read	50	0	0		0ms	0.000ms	0ms	0ms
write	50	0	0		0ms	0.000ms	0ms	0ms
epoll_wait	2	0	0		0ms	0.000ms	0ms	0ms

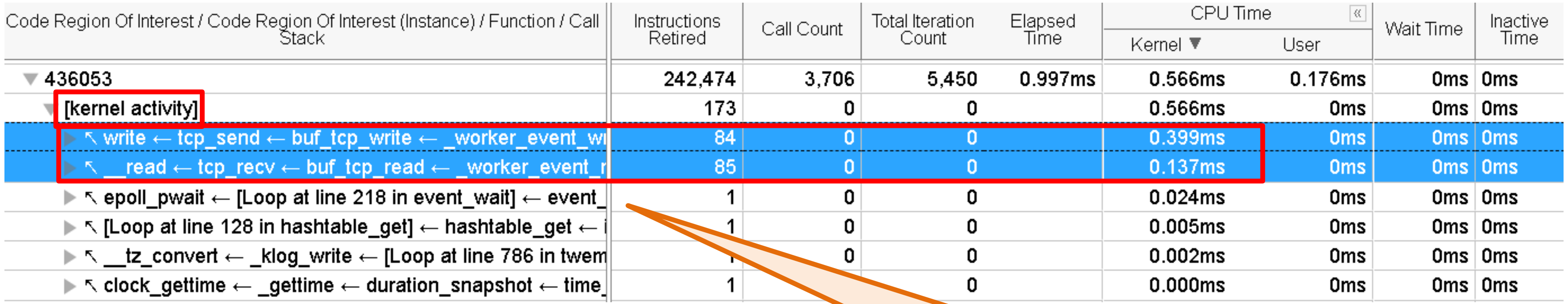
Expand instance with significant Wait Time metric to functions and stacks and see which stack(s) brought the thread to Idle

ANALYZING KERNEL INDUCED ANOMALIES

Sort by Kernel Time metric



Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel ▼	User		
core_worker_evloop	19,724,614	320,878	476,973	93.977ms	37.874ms	16.347ms	4.724ms	0ms
▶ 436053	242,474	3,706	5,450	0.997ms	0.566ms	0.176ms	0ms	0ms
▶ 412197	242,907	3,702	5,455	1.244ms	0.519ms	0.134ms	0ms	0ms
▶ 551160	227,194	3,549	5,224	1.088ms	0.519ms	0.144ms	0ms	0ms
▶ 471743	227,295	3,538	5,218	1.077ms	0.519ms	0.138ms	0ms	0ms
▶ 527220	231,793	3,535	5,191	1.145ms	0.516ms	0.132ms	0ms	0ms



Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel ▼	User		
▼ 436053	242,474	3,706	5,450	0.997ms	0.566ms	0.176ms	0ms	0ms
[kernel activity]	173	0	0		0.566ms	0ms	0ms	0ms
▶ ↖ write ← tcp_send ← buf_tcp_write ← _worker_event_wr	84	0	0		0.399ms	0ms	0ms	0ms
▶ ↖ __read ← tcp_recv ← buf_tcp_read ← _worker_event_r	85	0	0		0.137ms	0ms	0ms	0ms
▶ ↖ epoll_pwait ← [Loop at line 218 in event_wait] ← event_	1	0	0		0.024ms	0ms	0ms	0ms
▶ ↖ [Loop at line 128 in hashtable_get] ← hashtable_get ← i	1	0	0		0.005ms	0ms	0ms	0ms
▶ ↖ __tz_convert ← _klog_write ← [Loop at line 786 in twem	1	0	0		0.002ms	0ms	0ms	0ms
▶ ↖ clock_gettime ← _gettime ← duration_snapshot ← time_	1		0		0.000ms	0ms	0ms	0ms

Significant time (*566 out of 997 ms*) spent in the OS kernel

The control went to kernel while *receiving a request and sending a response* over the network

A likely cause of the slowdown is the network speed!

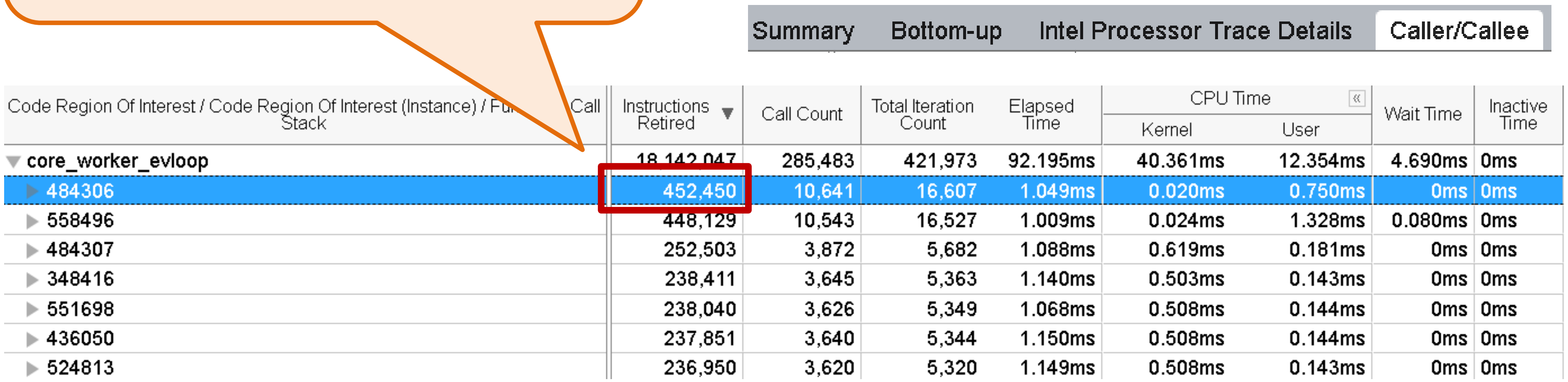
We don't show what is happening inside the kernel. We aggregate kernel time into an artificial [kernel activity] node

But in many cases stacks which led to the kernel give a clue

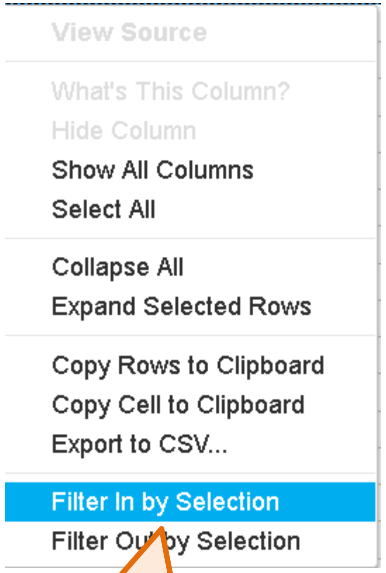
ANALYZING CONTROL FLOW DEVIATIONS (1/3)

Instead of expanding a specific instance, let's use another representation, which often works better for visualizing complex control flows - **Caller/Callee view**

Larger values of "Instructions Retired" often indicate Control Flow-related Anomalies.



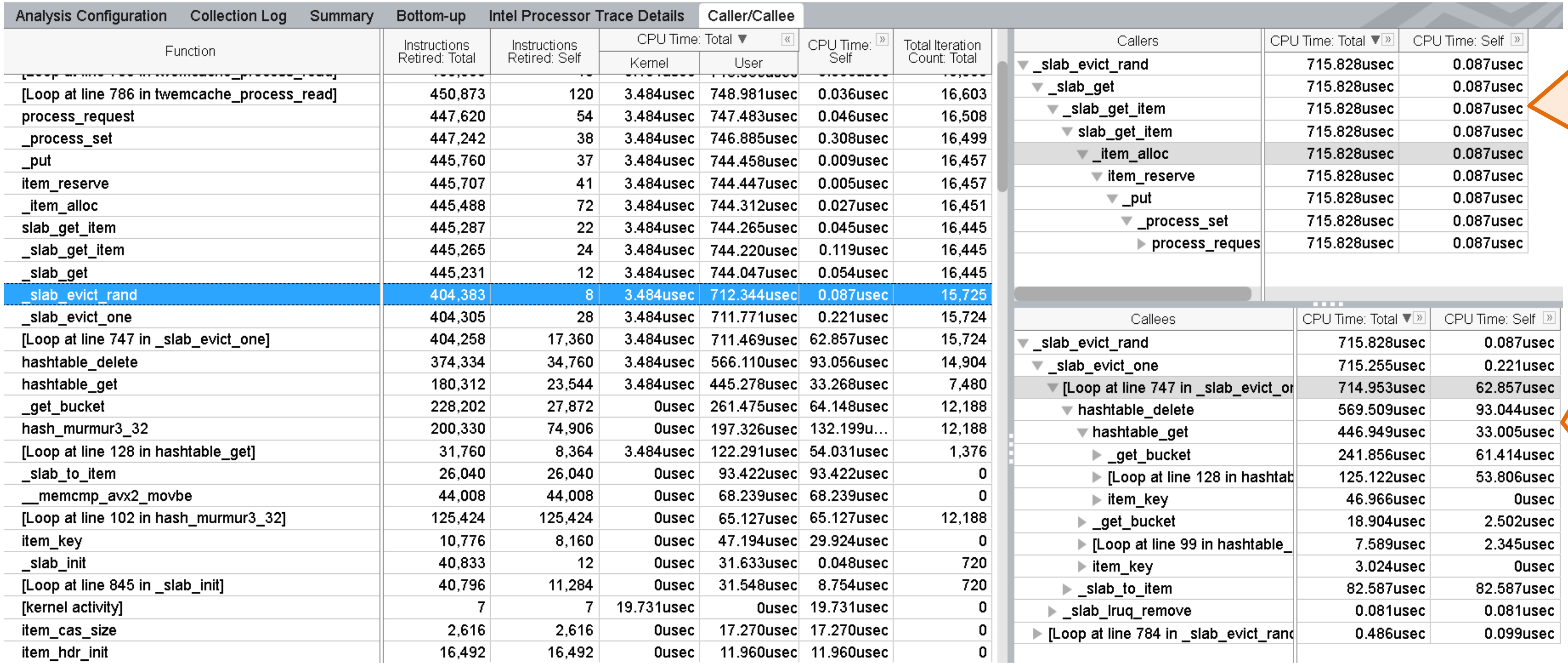
Code Region Of Interest / Code Region Of Interest (Instance) / Fu Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel	User		
core_worker_evloop	18,142,047	285,483	421,973	92.195ms	40.361ms	12.354ms	4.690ms	0ms
484306	452,450	10,641	16,607	1.049ms	0.020ms	0.750ms	0ms	0ms
558496	448,129	10,543	16,527	1.009ms	0.024ms	1.328ms	0.080ms	0ms
484307	252,503	3,872	5,682	1.088ms	0.619ms	0.181ms	0ms	0ms
348416	238,411	3,645	5,363	1.140ms	0.503ms	0.143ms	0ms	0ms
551698	238,040	3,626	5,349	1.068ms	0.508ms	0.144ms	0ms	0ms
436050	237,851	3,640	5,344	1.150ms	0.508ms	0.144ms	0ms	0ms
524813	236,950	3,620	5,320	1.149ms	0.508ms	0.143ms	0ms	0ms



Select a specific instance and choose "Filter In" from context menu. Then switch to Caller/Callee tab

ANALYZING CONTROL FLOW DEVIATIONS (2/3)

Flat profile view shows a function list annotated with self/total metrics



Function	Instructions Retired: Total	Instructions Retired: Self	CPU Time: Total ▼		CPU Time: Self	Total Iteration Count: Total
			Kernel	User		
[Loop at line 786 in twemcache_process_read]	450,873	120	3.484usec	748.981usec	0.036usec	16,603
process_request	447,620	54	3.484usec	747.483usec	0.046usec	16,508
_process_set	447,242	38	3.484usec	746.885usec	0.308usec	16,499
_put	445,760	37	3.484usec	744.458usec	0.009usec	16,457
item_reserve	445,707	41	3.484usec	744.447usec	0.005usec	16,457
_item_alloc	445,488	72	3.484usec	744.312usec	0.027usec	16,451
slab_get_item	445,287	22	3.484usec	744.265usec	0.045usec	16,445
_slab_get_item	445,265	24	3.484usec	744.220usec	0.119usec	16,445
_slab_get	445,231	12	3.484usec	744.047usec	0.054usec	16,445
_slab_evict_rand	404,383	8	3.484usec	712.344usec	0.087usec	15,725
_slab_evict_one	404,305	28	3.484usec	711.771usec	0.221usec	15,724
[Loop at line 747 in _slab_evict_one]	404,258	17,360	3.484usec	711.469usec	62.857usec	15,724
hashtable_delete	374,334	34,760	3.484usec	566.110usec	93.056usec	14,904
hashtable_get	180,312	23,544	3.484usec	445.278usec	33.268usec	7,480
_get_bucket	228,202	27,872	0usec	261.475usec	64.148usec	12,188
hash_murmur3_32	200,330	74,906	0usec	197.326usec	132.199u…	12,188
[Loop at line 128 in hashtable_get]	31,760	8,364	3.484usec	122.291usec	54.031usec	1,376
_slab_to_item	26,040	26,040	0usec	93.422usec	93.422usec	0
__memcmp_avx2_movbe	44,008	44,008	0usec	68.239usec	68.239usec	0
[Loop at line 102 in hash_murmur3_32]	125,424	125,424	0usec	65.127usec	65.127usec	12,188
item_key	10,776	8,160	0usec	47.194usec	29.924usec	0
_slab_init	40,833	12	0usec	31.633usec	0.048usec	720
[Loop at line 845 in _slab_init]	40,796	11,284	0usec	31.548usec	8.754usec	720
[kernel activity]	7	7	19.731usec	0usec	19.731usec	0
item_cas_size	2,616	2,616	0usec	17.270usec	17.270usec	0
item_hdr_init	16,492	16,492	0usec	11.960usec	11.960usec	0

Callers	CPU Time: Total ▼	CPU Time: Self
▼ _slab_evict_rand	715.828usec	0.087usec
▼ _slab_get	715.828usec	0.087usec
▼ _slab_get_item	715.828usec	0.087usec
▼ slab_get_item	715.828usec	0.087usec
▼ _item_alloc	715.828usec	0.087usec
▼ item_reserve	715.828usec	0.087usec
▼ _put	715.828usec	0.087usec
▼ _process_set	715.828usec	0.087usec
▶ process_reques	715.828usec	0.087usec

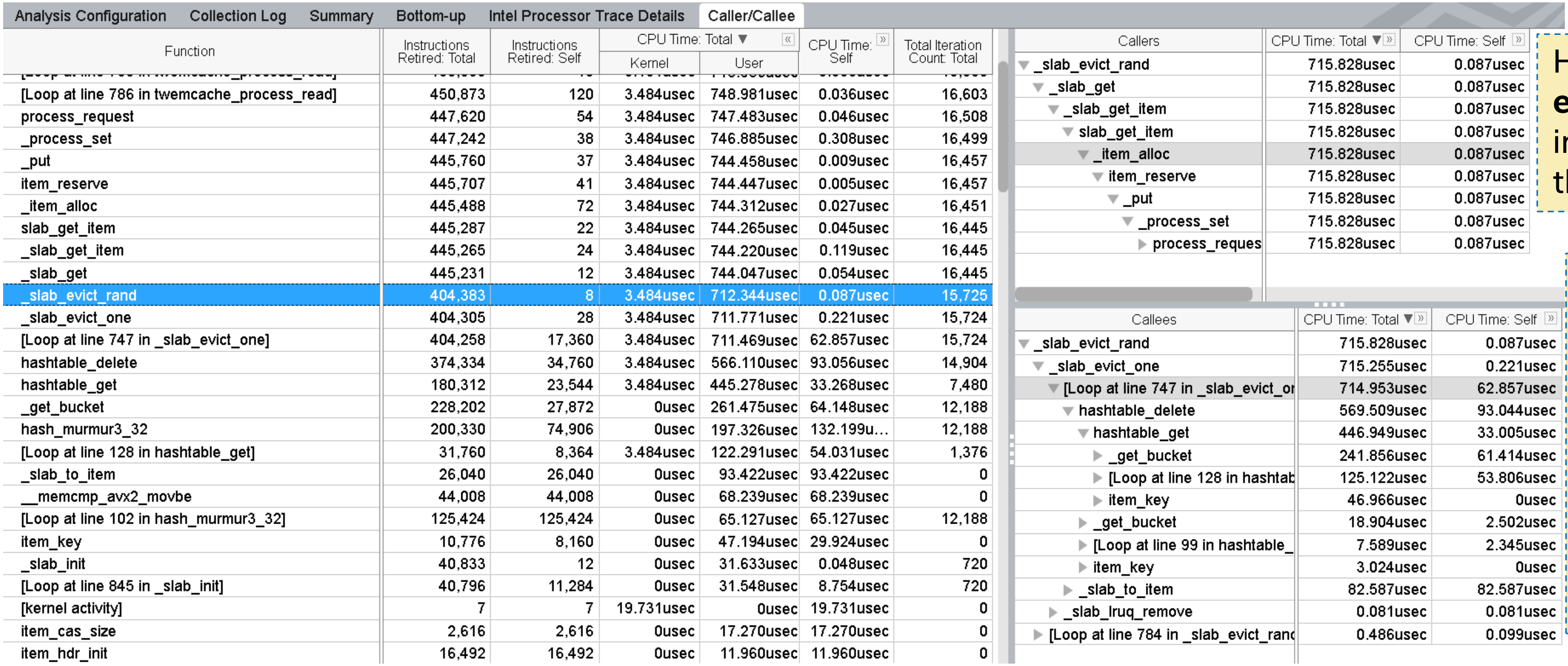
Callees	CPU Time: Total ▼	CPU Time: Self
▼ _slab_evict_rand	715.828usec	0.087usec
▼ _slab_evict_one	715.255usec	0.221usec
▼ [Loop at line 747 in _slab_evict_o	714.953usec	62.857usec
▼ hashtable_delete	569.509usec	93.044usec
▼ hashtable_get	446.949usec	33.005usec
▶ _get_bucket	241.856usec	61.414usec
▶ [Loop at line 128 in hashtab	125.122usec	53.806usec
▶ item_key	46.966usec	0usec
▶ _get_bucket	18.904usec	2.502usec
▶ [Loop at line 99 in hashtable_	7.589usec	2.345usec
▶ item_key	3.024usec	0usec
▶ _slab_to_item	82.587usec	82.587usec
▶ _slab_lruq_remove	0.081usec	0.081usec
▶ [Loop at line 784 in _slab_evict_ran	0.486usec	0.099usec

Caller view shows **callers of the selected function** in a bottom-up representation

Callee view shows a **call tree from selected function** in a top-down representation

ANALYZING CONTROL FLOW DEVIATIONS (3/3)

Call to **_slab_evict_one** causes slowdown, as that function and its callees take up most time



Function	Instructions Retired: Total	Instructions Retired: Self	CPU Time: Total Kernel	CPU Time: Total User	CPU Time: Self	Total Iteration Count: Total
[Loop at line 786 in twemcache_process_read]	450,873	120	3.484usec	748.981usec	0.036usec	16,603
process_request	447,620	54	3.484usec	747.483usec	0.046usec	16,508
_process_set	447,242	38	3.484usec	746.885usec	0.308usec	16,499
_put	445,760	37	3.484usec	744.458usec	0.009usec	16,457
item_reserve	445,707	41	3.484usec	744.447usec	0.005usec	16,457
_item_alloc	445,488	72	3.484usec	744.312usec	0.027usec	16,451
slab_get_item	445,287	22	3.484usec	744.265usec	0.045usec	16,445
_slab_get_item	445,265	24	3.484usec	744.220usec	0.119usec	16,445
_slab_get	445,231	12	3.484usec	744.047usec	0.054usec	16,445
_slab_evict_rand	404,383	8	3.484usec	712.344usec	0.087usec	15,725
_slab_evict_one	404,305	28	3.484usec	711.771usec	0.221usec	15,724
[Loop at line 747 in _slab_evict_one]	404,258	17,360	3.484usec	711.469usec	62.857usec	15,724
hashtable_delete	374,334	34,760	3.484usec	566.110usec	93.056usec	14,904
hashtable_get	180,312	23,544	3.484usec	445.278usec	33.268usec	7,480
_get_bucket	228,202	27,872	0usec	261.475usec	64.148usec	12,188
hash_murmur3_32	200,330	74,906	0usec	197.326usec	132.199u…	12,188
[Loop at line 128 in hashtable_get]	31,760	8,364	3.484usec	122.291usec	54.031usec	1,376
_slab_to_item	26,040	26,040	0usec	93.422usec	93.422usec	0
__memcmp_avx2_movbe	44,008	44,008	0usec	68.239usec	68.239usec	0
[Loop at line 102 in hash_murmur3_32]	125,424	125,424	0usec	65.127usec	65.127usec	12,188
item_key	10,776	8,160	0usec	47.194usec	29.924usec	0
_slab_init	40,833	12	0usec	31.633usec	0.048usec	720
[Loop at line 845 in _slab_init]	40,796	11,284	0usec	31.548usec	8.754usec	720
[kernel activity]	7	7	19.731usec	0usec	19.731usec	0
item_cas_size	2,616	2,616	0usec	17.270usec	17.270usec	0
item_hdr_init	16,492	16,492	0usec	11.960usec	11.960usec	0

Callers	CPU Time: Total	CPU Time: Self
_slab_evict_rand	715.828usec	0.087usec
_slab_get	715.828usec	0.087usec
_slab_get_item	715.828usec	0.087usec
slab_get_item	715.828usec	0.087usec
_item_alloc	715.828usec	0.087usec
item_reserve	715.828usec	0.087usec
_put	715.828usec	0.087usec
_process_set	715.828usec	0.087usec
process_reques	715.828usec	0.087usec

Callees	CPU Time: Total	CPU Time: Self
_slab_evict_rand	715.828usec	0.087usec
_slab_evict_one	715.255usec	0.221usec
[Loop at line 747 in _slab_evict_or	714.953usec	62.857usec
hashtable_delete	569.509usec	93.044usec
hashtable_get	446.949usec	33.005usec
_get_bucket	241.856usec	61.414usec
[Loop at line 128 in hashtab	125.122usec	53.806usec
item_key	46.966usec	0usec
_get_bucket	18.904usec	2.502usec
[Loop at line 99 in hashtable_	7.589usec	2.345usec
item_key	3.024usec	0usec
_slab_to_item	82.587usec	82.587usec
_slab_lruq_remove	0.081usec	0.081usec
[Loop at line 784 in _slab_evict_ranc	0.486usec	0.099usec

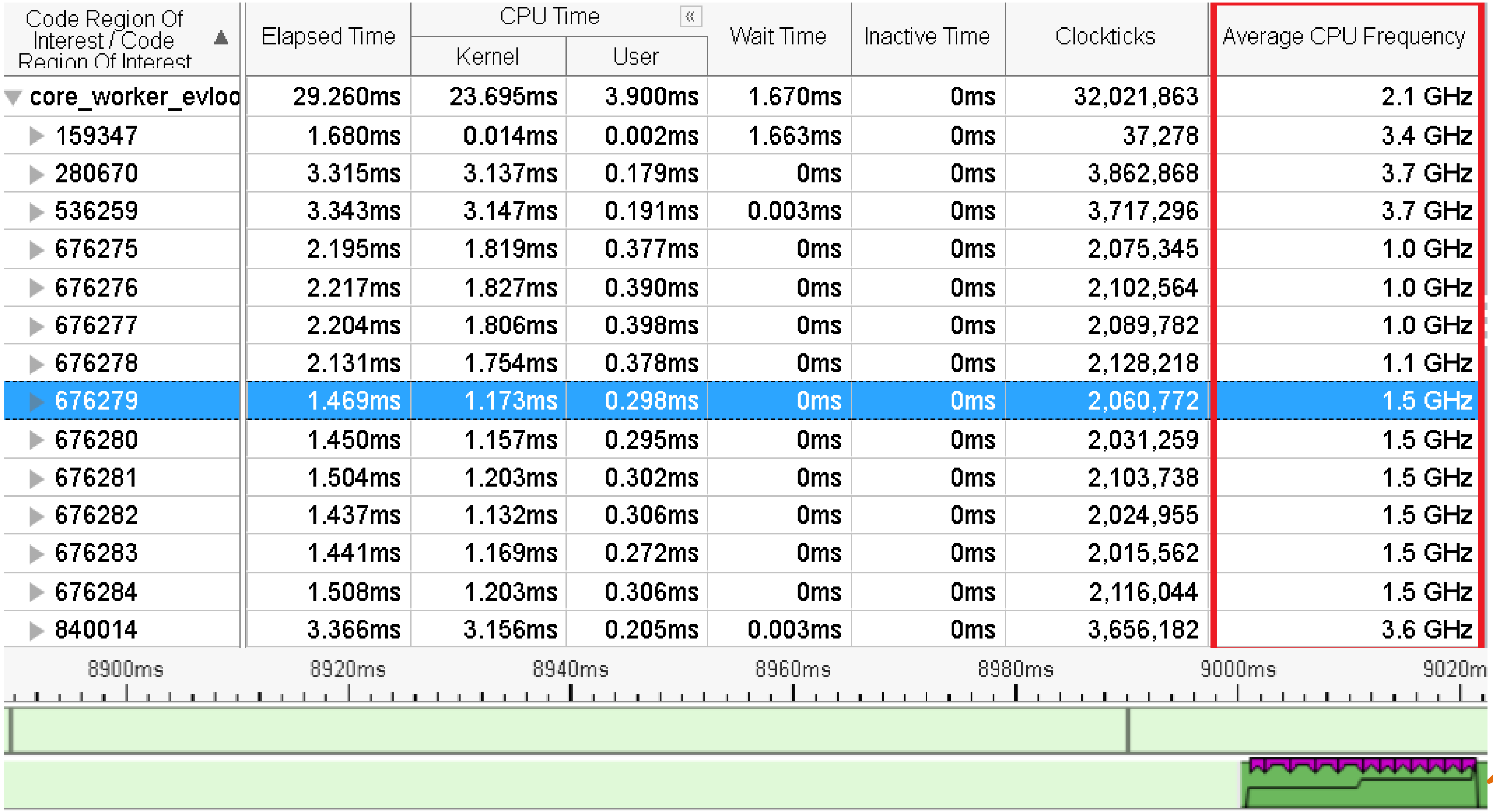
Here we have **cache eviction**. A rare operation in a normal flow, rather than an anomaly

We cannot eliminate evictions, but we can:

- **Make them less frequent** by increasing the cache size
- **Try to optimize eviction processing** with the help of Caller/Callee view

ANALYZING CPU FREQUENCY EFFICIENCY

CPU frequency drop/boost can affect total latency up to several times



Code Region Of Interest / Code Region Of Interest	Elapsed Time	CPU Time		Wait Time	Inactive Time	Clockticks	Average CPU Frequency
		Kernel	User				
core_worker_evloo	29.260ms	23.695ms	3.900ms	1.670ms	0ms	32,021,863	2.1 GHz
159347	1.680ms	0.014ms	0.002ms	1.663ms	0ms	37,278	3.4 GHz
280670	3.315ms	3.137ms	0.179ms	0ms	0ms	3,862,868	3.7 GHz
536259	3.343ms	3.147ms	0.191ms	0.003ms	0ms	3,717,296	3.7 GHz
676275	2.195ms	1.819ms	0.377ms	0ms	0ms	2,075,345	1.0 GHz
676276	2.217ms	1.827ms	0.390ms	0ms	0ms	2,102,564	1.0 GHz
676277	2.204ms	1.806ms	0.398ms	0ms	0ms	2,089,782	1.0 GHz
676278	2.131ms	1.754ms	0.378ms	0ms	0ms	2,128,218	1.1 GHz
676279	1.469ms	1.173ms	0.298ms	0ms	0ms	2,060,772	1.5 GHz
676280	1.450ms	1.157ms	0.295ms	0ms	0ms	2,031,259	1.5 GHz
676281	1.504ms	1.203ms	0.302ms	0ms	0ms	2,103,738	1.5 GHz
676282	1.437ms	1.132ms	0.306ms	0ms	0ms	2,024,955	1.5 GHz
676283	1.441ms	1.169ms	0.272ms	0ms	0ms	2,015,562	1.5 GHz
676284	1.508ms	1.203ms	0.306ms	0ms	0ms	2,116,044	1.5 GHz
840014	3.366ms	3.156ms	0.205ms	0.003ms	0ms	3,656,182	3.6 GHz

Looking at timeline, request handling activity is done in sparse bursts, there’s not enough overall CPU utilization, so *OS lowers frequency and then tries to catch up*

Try increasing the number of requests or *disable frequency changes*

Frequency graph for a burst of marked code regions

CASE STUDY SUMMARY

The real-life example above demonstrated many of the typical reasons for performance anomalies, and using VTune™ Anomaly Detection methodology we were able to give the following recommendations:

Type of performance anomaly	Reason	Recommendations
Context switches	*Request handler goes to idle while waiting for a request - not enough requests in a queue*	Clients should give enough work for the server
Kernel activity	*Receiving a request/sending a response over network takes up a significant part of the request handling process*	Check network conditions, a faster link between client and server might be required
Control flow	*Request processing takes significantly longer if cache eviction is required*	Adjust the cache size to avoid frequent evictions Try to optimize eviction handling using the data provided by VTune. Caller/Callee view gives enough information to analyze hot paths in eviction handling
CPU frequency	*Low CPU utilization may cause frequency drop*	If CPU utilization varies with the number of requests, disable frequency changes in the system to minimize performance deviations
Microarchitectural issues	*Other problems outweighed microarchitectural issues in this example*	Refer to backup slides for an example of microarchitectural anomalies

NEXT STEPS

- Give Intel® VTune™ Profiler a try and apply our anomaly detection methodology to your work:
  - https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html
  - See backup slides for setting up data collection
- Please get back to us with any feedback – questions/suggestions/complaints
  - It will help us streamline the analysis and prioritize future work on SW and HW features!

CONCLUSION

- Nanosecond-level processor tracing (PT) in CPUs is the best way to spot latency anomalies
- Effectively utilizing PT requires lightweight instrumentation and thread-specific data collection and analysis
- **Intel PT + VTune + ITT API** provide for fine-grain time and event measurements
  - Granularity of **microseconds** and **nanoseconds**
  - Indispensable for detecting sporadic latency ***anomalies*** that are ***hard to find*** with traditional tools

intel®

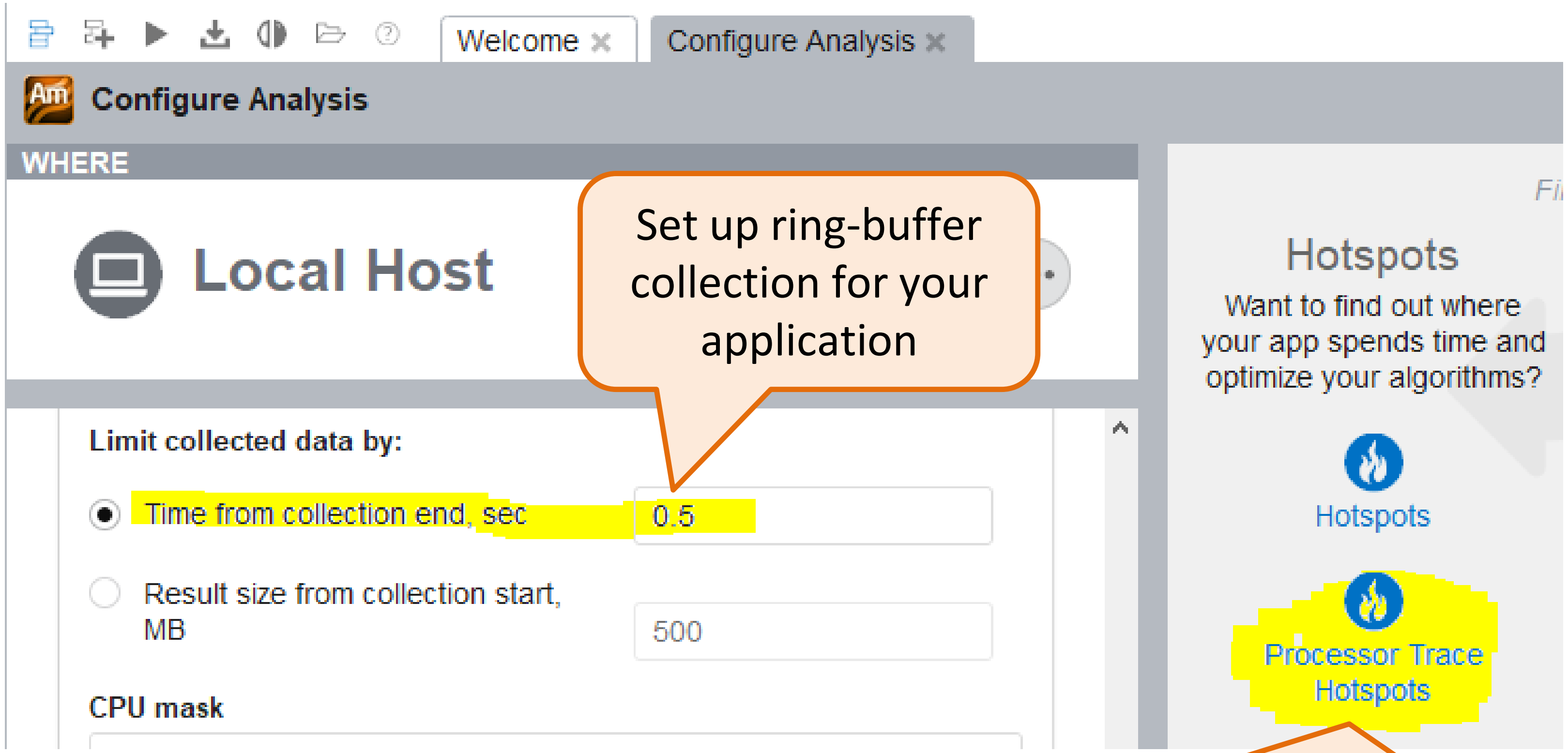
BACKUP SLIDES

INSTRUMENTING CODE WITH ITT API

- Include header file from VTune:
  - #include "ittnotify.h“
- Instrument your code:
  - __itt_pt_region region = __itt_pt_region_create(<region name>);

  …
  - __itt_mark_pt_region_begin(region);
  - <code region of interest here>
  - __itt_mark_pt_region_end(region);
- Link with ITT API library:
  - CC … -L$VTUNE_PROFILER_2020_DIR/lib64 -littnotify -I$VTUNE_PROFILER_2020_DIR/include

CONFIGURE COLLECTION: RING BUFFER



Limit collected data by:

● Time from collection end, sec: 0.5

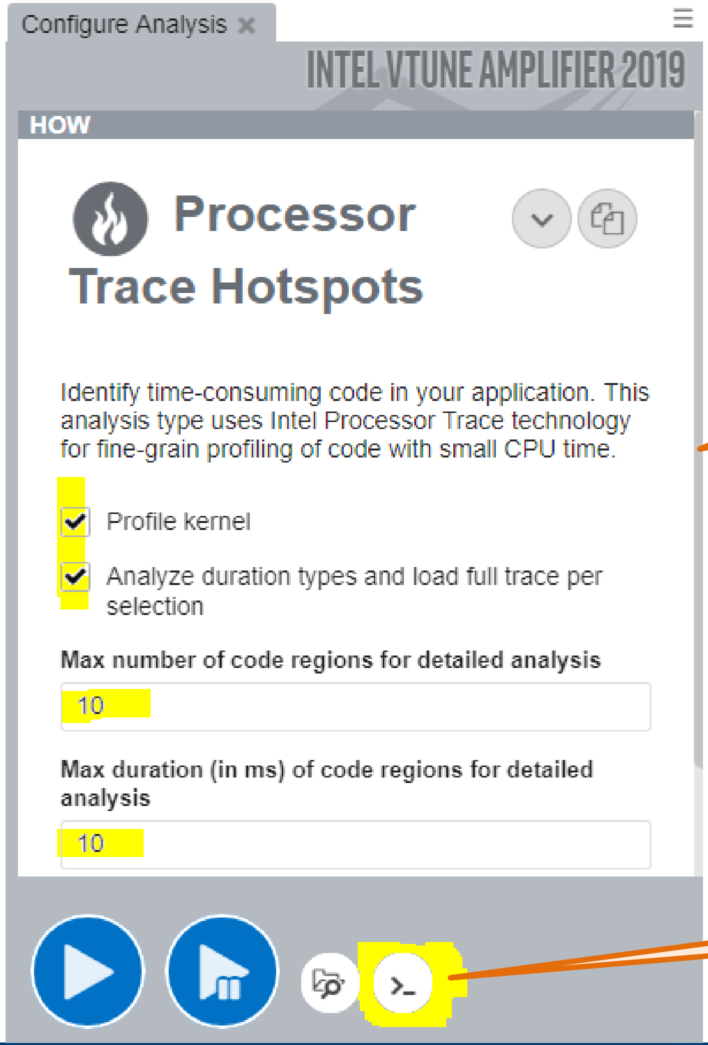
○ Result size from collection start, MB: 500

CPU mask

Set up ring-buffer collection for your application

Enabled by AMPLXE_EXPERIMENTAL=full-intel-pt

CONFIGURE COLLECTION: PT CONFIG + COMMAND LINE



Processor Trace Hotspots

Identify time-consuming code in your application. This analysis type uses Intel Processor Trace technology for fine-grain profiling of code with small CPU time.

- ☑ Profile kernel
- ☑ Analyze duration types and load full trace per selection

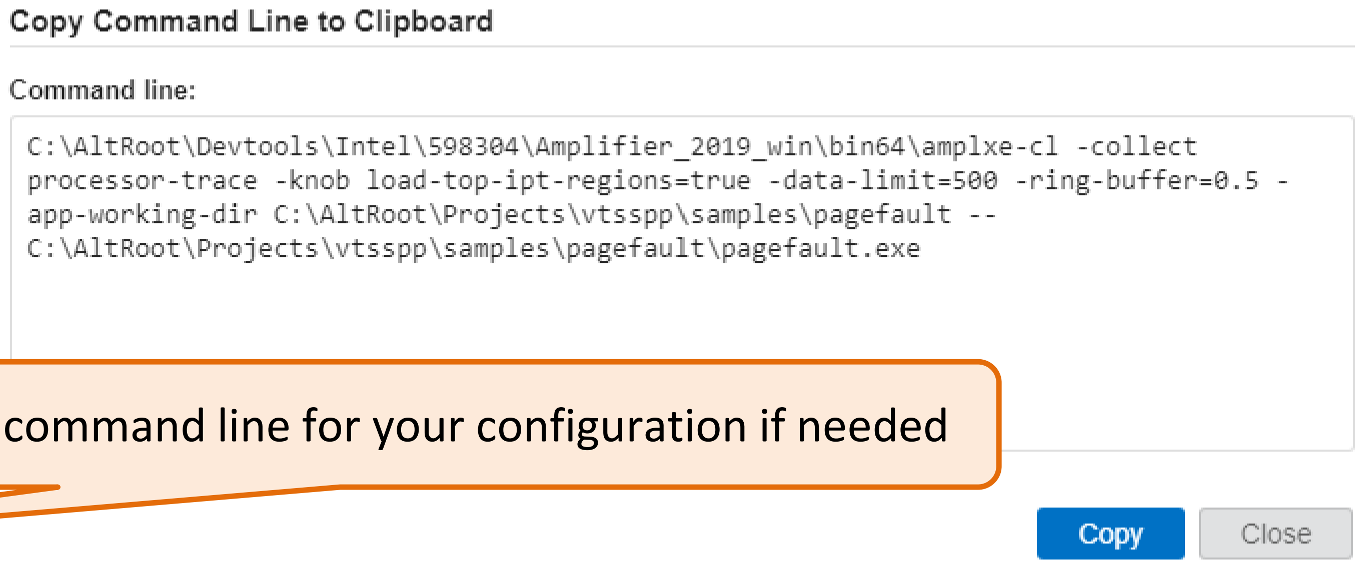
Max number of code regions for detailed analysis

10

Max duration (in ms) of code regions for detailed analysis

10

Enable kernel profiling and specify code region parameters: how many regions to load details for and what's the maximum expected duration of a code region *– regions outside the specified limits will be discarded*



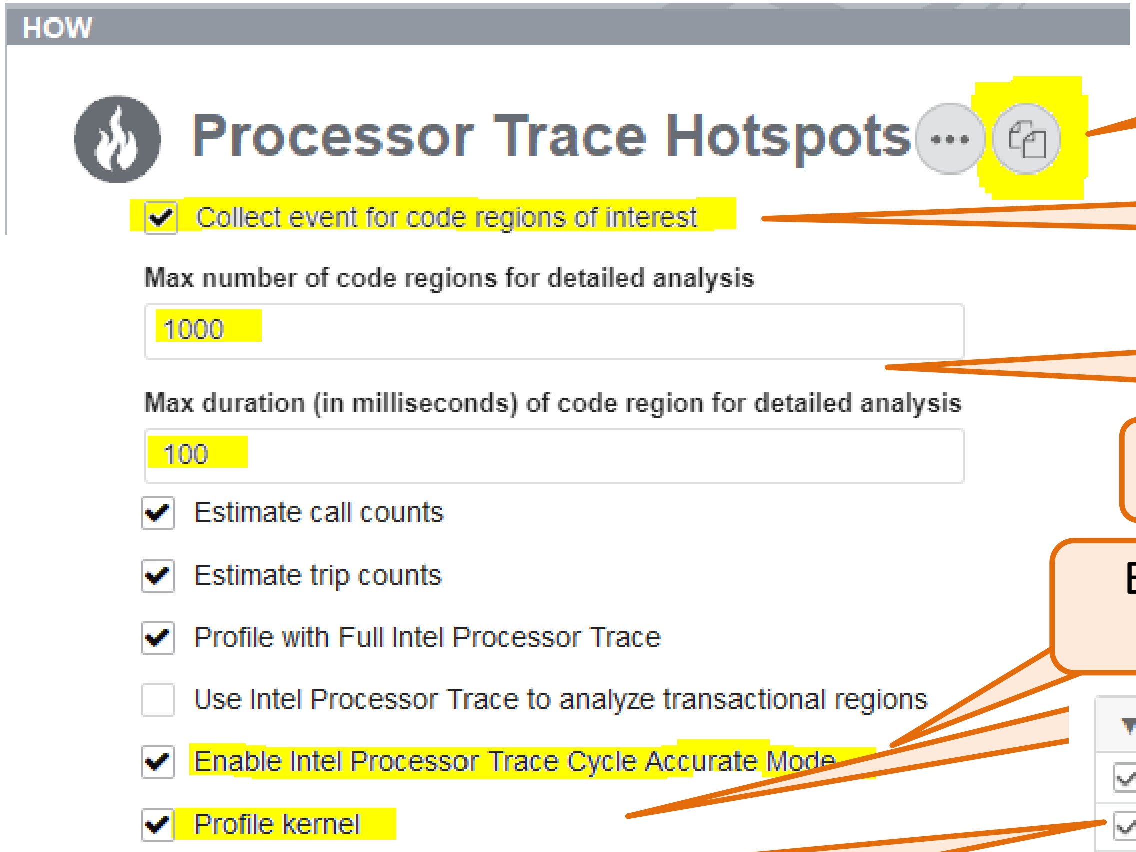
Copy Command Line to Clipboard

Command line:

```
C:\AltRoot\Devtools\Intel\598304\Amplifier_2019_win\bin64\amplxe-cl -collect
processor-trace -knob load-top-ipt-regions=true -data-limit=500 -ring-buffer=0.5 -
app-working-dir C:\AltRoot\Projects\vtsspp\samples\pagefault --
C:\AltRoot\Projects\vtsspp\samples\pagefault\pagefault.exe
```

Get command line for your configuration if needed

CONFIGURE COLLECTION: HW EVENTS



Processor Trace Hotspots

☑ Collect event for code regions of interest

Max number of code regions for detailed analysis: 1000

Max duration (in milliseconds) of code region for detailed analysis: 100

☑ Estimate call counts

☑ Estimate trip counts

☑ Profile with Full Intel Processor Trace

☐ Use Intel Processor Trace to analyze transactional regions

☑ Enable Intel Processor Trace Cycle Accurate Mode

☑ Profile kernel

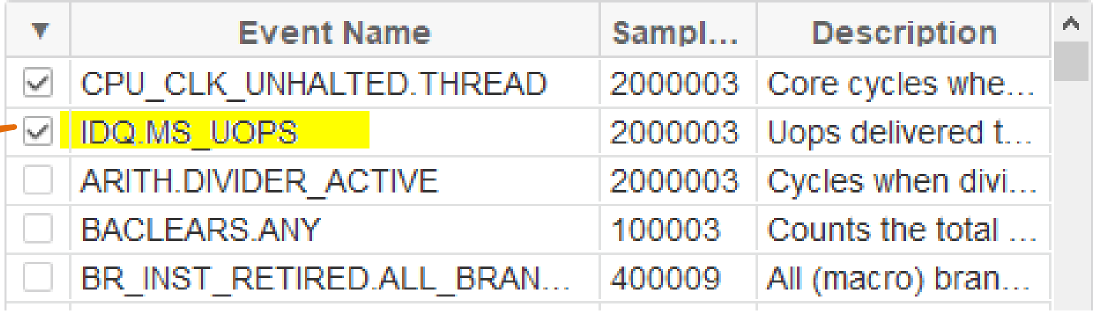
Clone Processor Trace Hotspots

Enable HW event collection

Configure number of code regions and their expected durations

Make sure Cycle-Accurate Mode is on

Enable kernel tracing to profile interrupts, exceptions and other OS activities

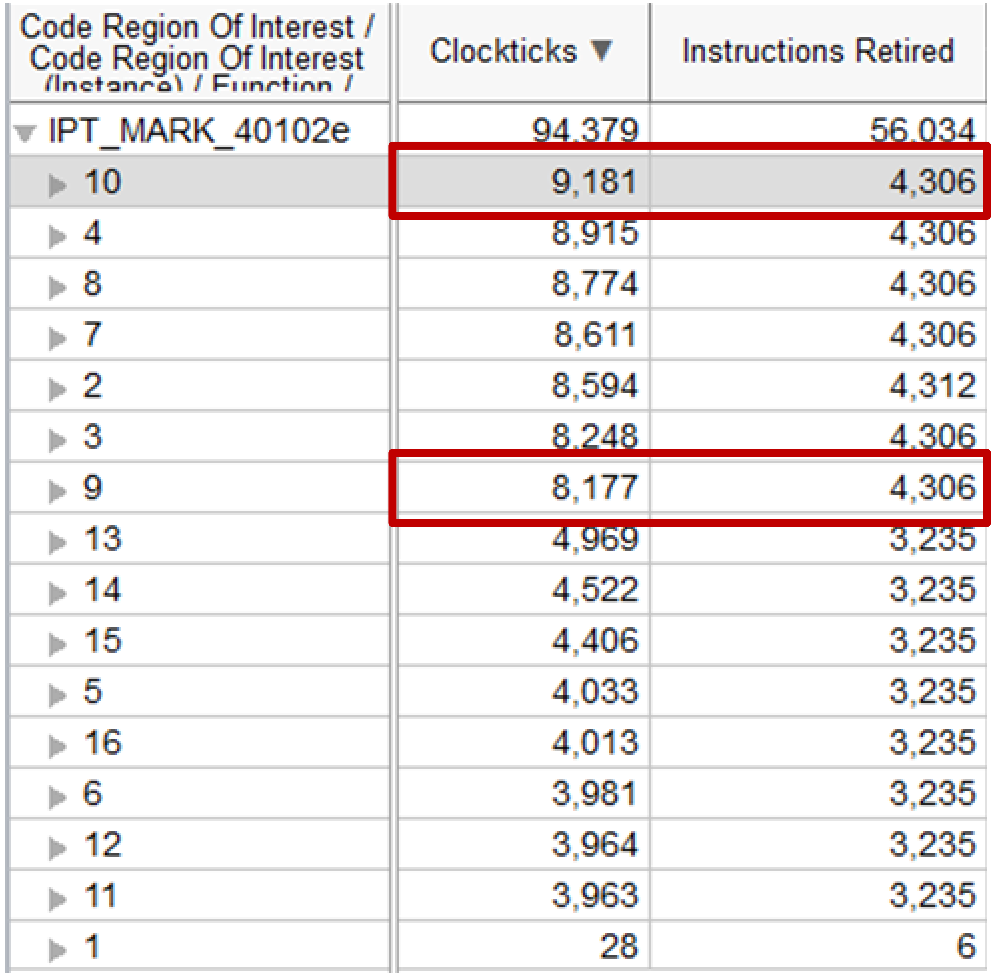


▼	Event Name	Sampl...	Description
☑	CPU_CLK_UNHALTED.THREAD	2000003	Core cycles whe...
☑	IDQ.MS_UOPS	2000003	Uops delivered t...
☐	ARITH.DIVIDER_ACTIVE	2000003	Cycles when divi...
☐	BACLEARS.ANY	100003	Counts the total ...
☐	BR_INST_RETIRED.ALL_BRAN...	400009	All (macro) bran...

Choose **one** event to profile, in addition to CPU_CLK_UNHALTED.THREAD

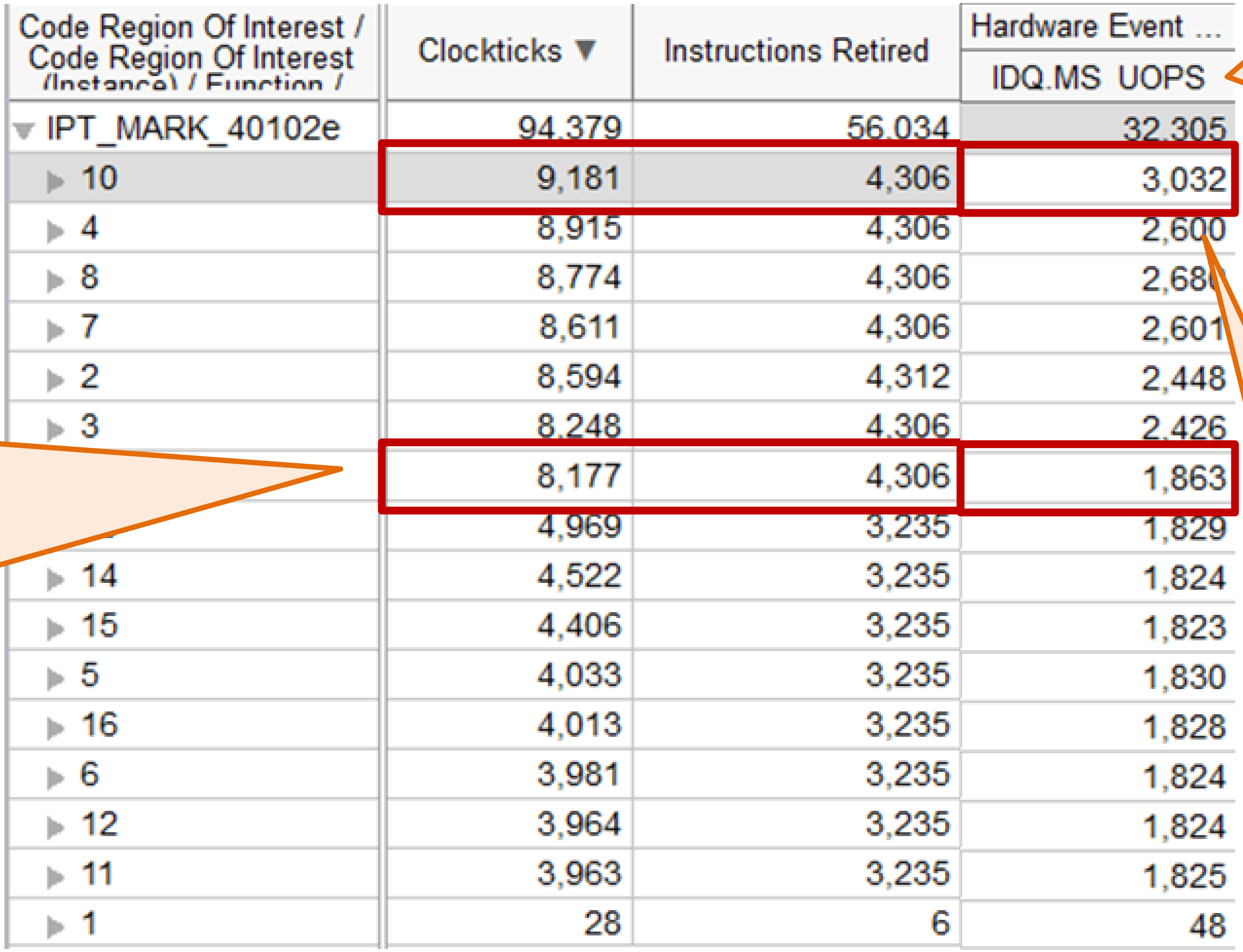
ANALYZING MICROARCHITECTURE-RELATED ANOMALIES

Code instances with *different Clockticks*/CPU Time, but the *same or close number of retired instructions*, plus no significant Idle time or kernel activity, often indicate **microarchitecture-related anomalies**



Code Region Of Interest / Code Region Of Interest (Instance) / Function /	Clockticks ▼	Instructions Retired
▼ IPT_MARK_40102e	94,379	56,034
▶ 10	9,181	4,306
▶ 4	8,915	4,306
▶ 8	8,774	4,306
▶ 7	8,611	4,306
▶ 2	8,594	4,312
▶ 3	8,248	4,306
▶ 9	8,177	4,306
▶ 13	4,969	3,235
▶ 14	4,522	3,235
▶ 15	4,406	3,235
▶ 5	4,033	3,235
▶ 16	4,013	3,235
▶ 6	3,981	3,235
▶ 12	3,964	3,235
▶ 11	3,963	3,235
▶ 1	28	6

ANALYZING MICROARCHITECTURE-RELATED ANOMALIES



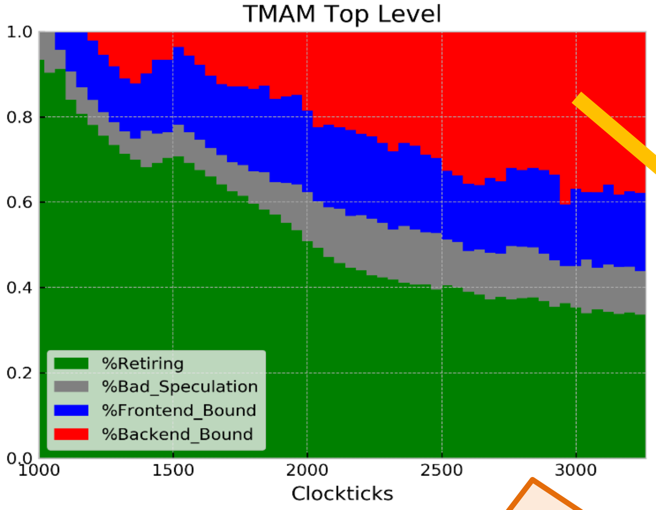
Code Region Of Interest / Code Region Of Interest (Instance) / Function /	Clockticks ▼	Instructions Retired	Hardware Event ... IDQ.MS_UOPS
IPT_MARK_40102e	94,379	56,034	32,305
10	9,181	4,306	3,032
4	8,915	4,306	2,600
8	8,774	4,306	2,68
7	8,611	4,306	2,601
2	8,594	4,312	2,448
3	8,248	4,306	2,426
	8,177	4,306	1,863
	4,969	3,235	1,829
14	4,522	3,235	1,824
15	4,406	3,235	1,823
5	4,033	3,235	1,830
16	4,013	3,235	1,828
6	3,981	3,235	1,824
12	3,964	3,235	1,824
11	3,963	3,235	1,825
1	28	6	48

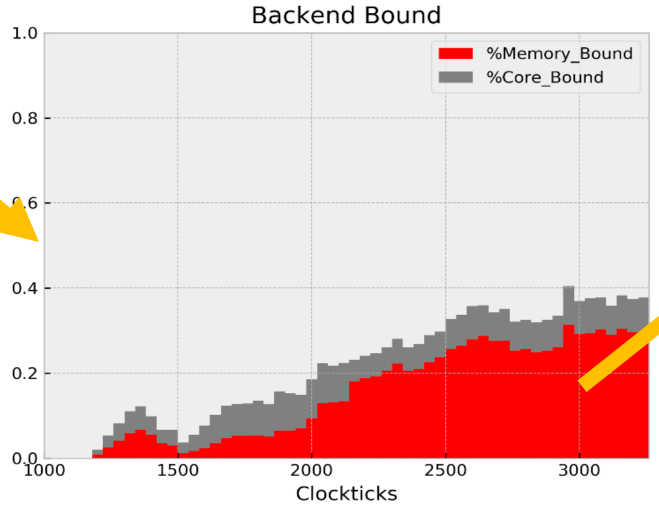
Let's collect a perfmon event together with PT

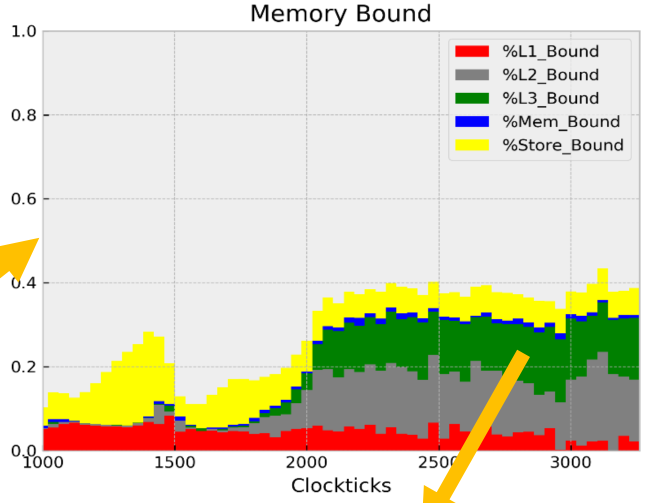
Code instances with *different Clockticks*/CPU Time, but the *same or close number of retired instructions*, plus no significant Idle time or kernel activity, often indicate **microarchitecture-related anomalies**

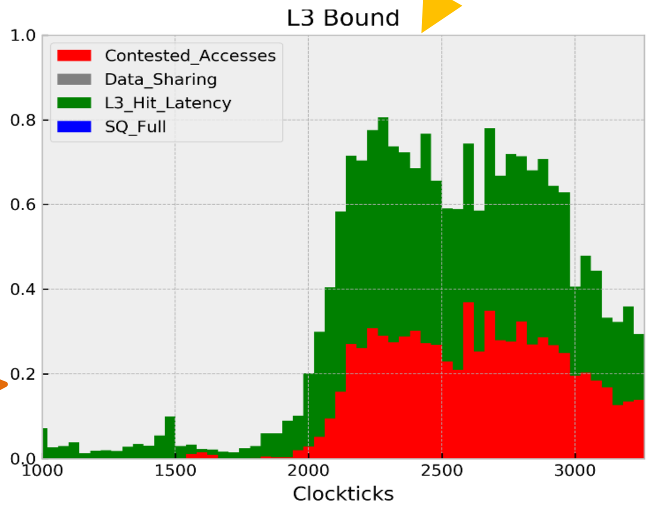
Slower code instances are **affected by** micro-operations from **Microcode Sequencer**

PROTOTYPE: TMAM METRICS PER REGION INVOCATION









In multiple runs, collect TMAM events, level by level, for each code region invocation.

Longer instances suffer from contested accesses to L3 data!