Linaro Connect
MADRID 2024 | MAY 14-17 2024

Boost your performance on WOA with WindowsPerf

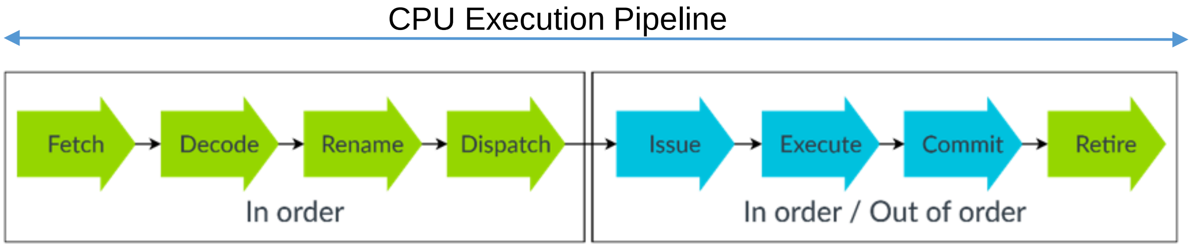
Jumana Mundichipparakkal, Principal System Architect, Arm
Przemyslaw Wirkus, Principal Engineer, Arm

Agenda

- Arm Topdown Methodology for Performance Analysis
- Introduction to Arm Telemetry Solution
- Introduction to WindowsPerf
- Performance Analysis using Arm Telemetry Solution & Windowsperf Tool
  - Demo 1: CPython
  - Demo 2: Synthetic Workloads

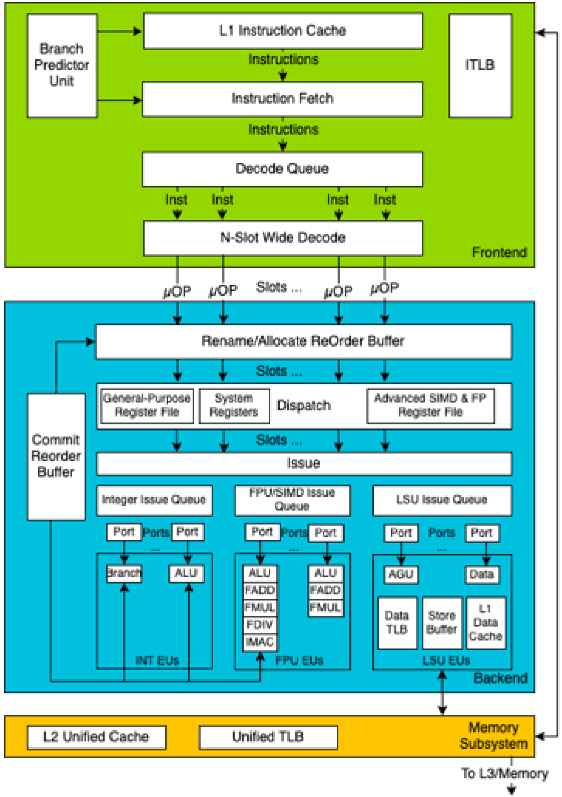
Arm Telemetry Solution for Performance Analysis

- **CPU Performance Analysis**:
  - Investigating and diagnosing performance inefficiencies during workload execution.
  - Iterative and complex process to pinpoint exact performance issues
- **Topdown Performance analysis methodology:**
  - Hierarchical metrics and guideline to characterize the distribution of cycles spent by the CPU to pinpoint efficient cycles (when instructions are executed) and wasted cycles (due to pipeline stalls and branch redirections), which helps identify bottlenecks.
  - To address inefficiencies observed from the CPU telemetry data, software developer can choose appropriate data structures and applying code tuning techniques in your software.



- **Arm Telemetry Solution: developer.arm.com/telemetry**
  - Provides Arm topdown performance analysis methodology, a standardized telemetry framework, a*nd Arm topdown tool*
  - Designed to use a CPU's telemetry data to help identify performance bottlenecks and improve execution efficiency by using these components

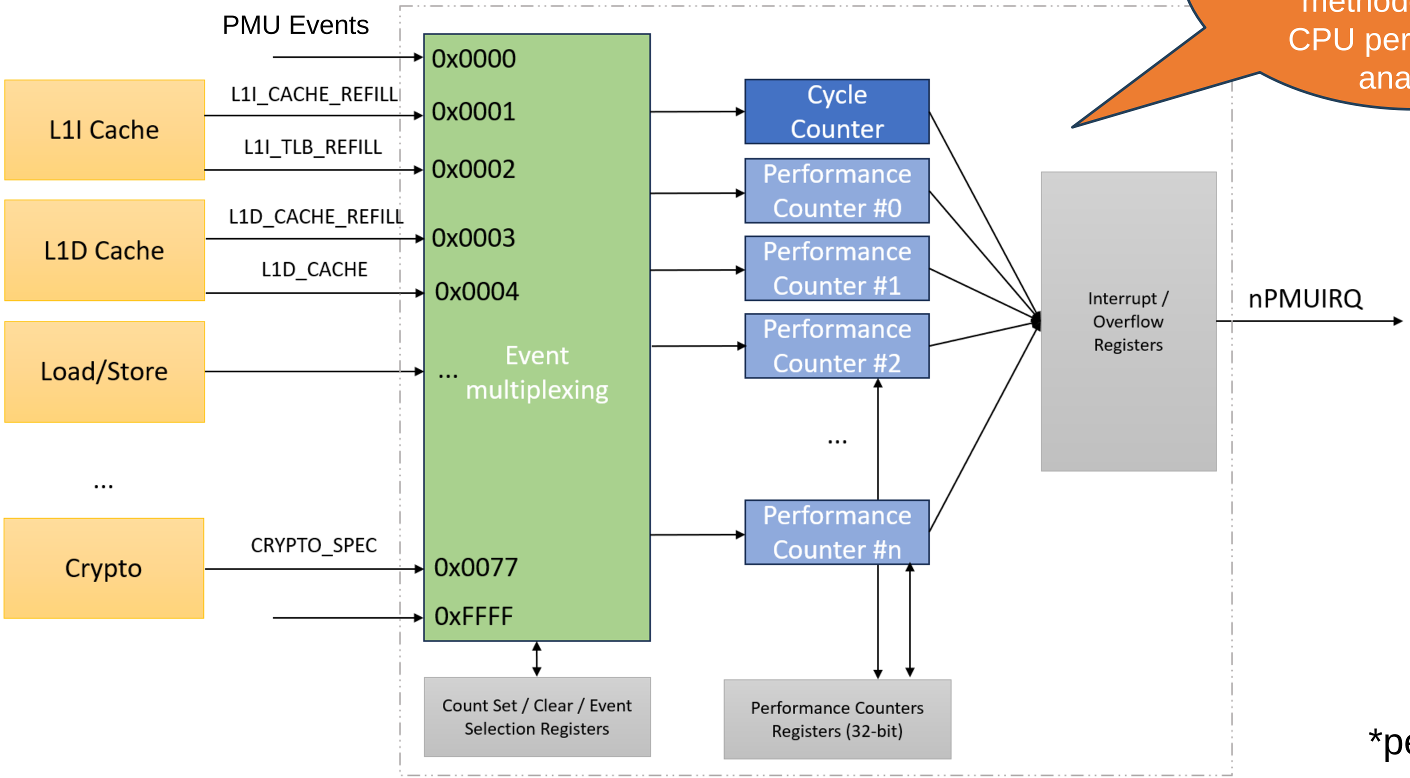
Neoverse CPU Microarchitecture



- In order Frontend
  - Major Blocks:
    - L1I Cache
    - ITLB
    - Branch Predictors
  - **Instruction Fetch, Decode and Rename**
- Out of Order Backend
  - L1 Data Cache
  - Execution Units
    - Branch, Arithmetic, FP/SIMD, Load Store
  - **Operation Rename, Dispatch, Issue, Execute, Retire (Instructions)**
- Shared Memory Subsystem
  - L2 Cache, L2 TLB
  - System bus interface for uncore memory transactions

Neoverse server systems support a last level cache in the system interconnects which is a platform configuration option

Arm CPU PMU* overview

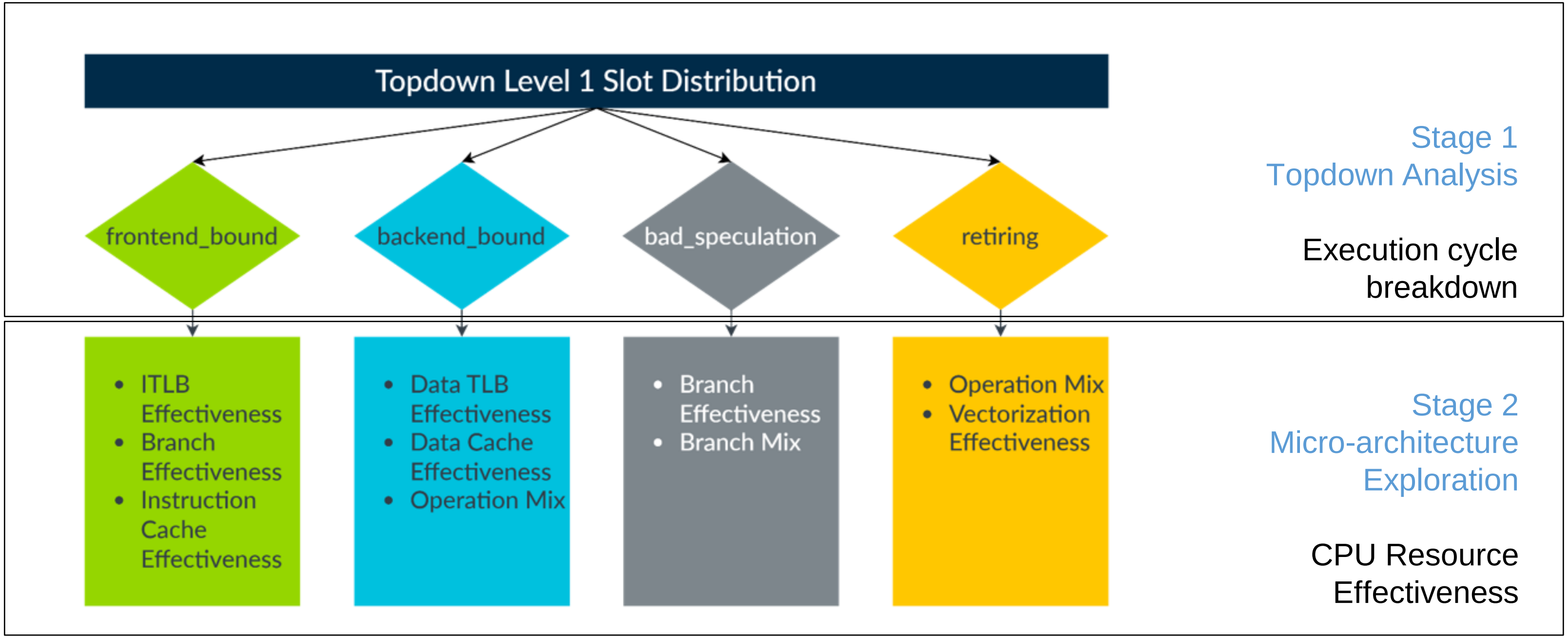


PMU events provided by hardware help to create metrics and methodology for CPU performance analysis

*per CPU core

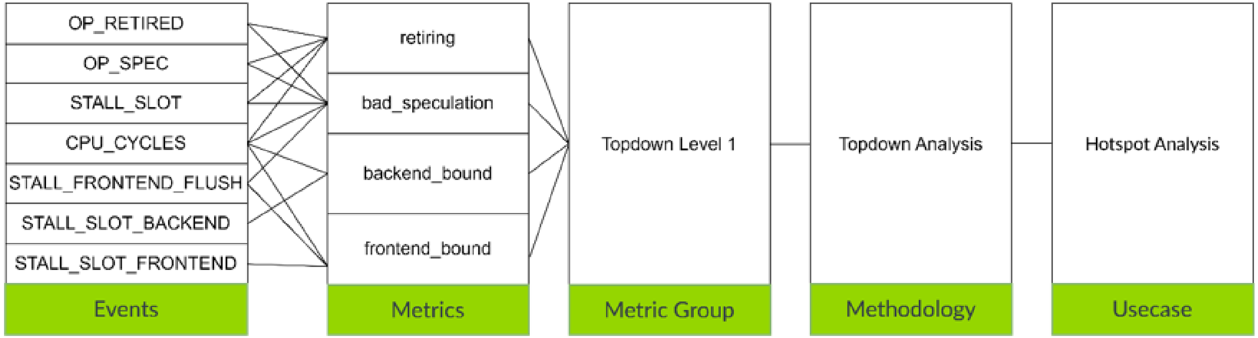
Arm Topdown Methodology for Hotspot Analysis

- Provides hierarchical metrics to evaluate cycle utilization on a modern CPU for performance bottleneck identification and hotspot analysis in an application



Arm Telemetry Framework & Specification

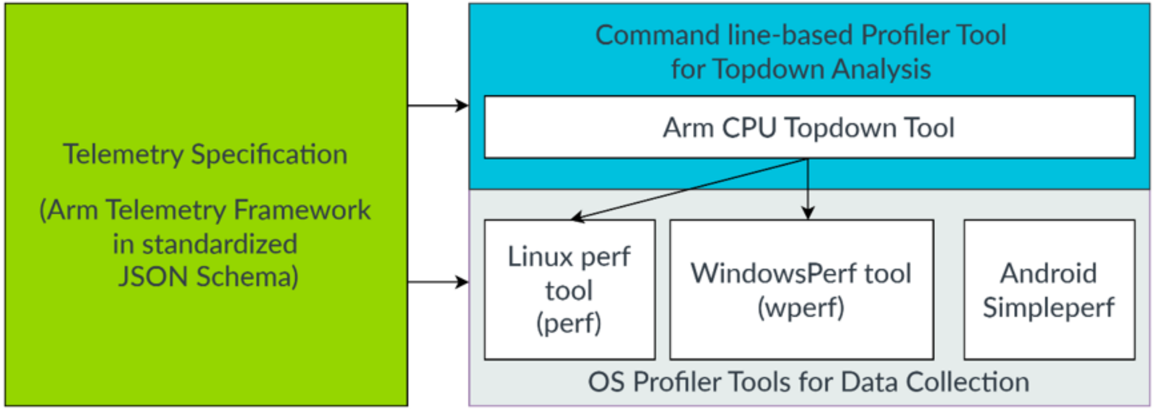
- Arm Telemetry Framework provides a standardized data model to capture the PMU event supported by the hardware and the derived metrics needed for performance analysis.
- Key elements of the framework:
    - Events, Metrics, Metric Groups and Methodology



- All Arm CPUs that support telemetry solution implements this data framework to produce the telemetry data
    - JSON schema and per-CPU JSON data available at: gitlab.arm.com/telemetry-solution/data

Arm CPU Topdown Tool

- Command line tool to collect the telemetry data and process it to produce the telemetry metrics for application topdown analysis
- Supports data collection on Linux and Windows platforms
- Parses Arm CPU Telemetry JSON as the profiler input



- Arm topdown tool: gitlab.arm.com/telemetry-solution/tools

Performance Analysis using WindowsPerf

Topdown µArchitecture analysis with WindowsPerf

- PMU Event based performance analysis on Windows On Arm.

- WindowsPerf Tool:
  - Provide Arm PMU based performance monitoring tool inspired by Linux perf.
  - Static performance analysis tool.
  - Support core PMU, DSU DMC, and more.
  - Open-sourced with permissive BSD License.

- Data Collection Approaches
  - the **counting model**, for obtaining aggregate counts of occurrences of PMU events, and
  - the **sampling model**, for determining the frequencies of event occurrences produced by program locations at the function, basic block, and/or instruction levels.

WindowsPerf and Arm Telemetry Solution Integration

- Arm Topdown Methodology for µarch performance analysis:
    - Arm Telemetry Solution specification - per Arm CPU µarch.
    - PMU events, metrics, groups of metrics.
- WindowsPerf Tool Features:
    - Platform µarchitecture detection (Neoverse-N1 / V1 / N2) for uarch telemetry JSON
    - µarch events, metrics, metric groups
- Arm Topdown Tool
    - Supports data collection on Windows on Arm CPUs using Windowsperf

- Arm Telemetry Solution and *Windowsperf* enable Windows on Arm performance analysis

WindowsPerf in Action: Demo Examples

1) CPython performance analysis demo
    1) Manual analysis using *WindowsPerf*
    2) Automated Analysis using *Arm Topdown tool*
2) Synthetic workload performance analysis demo
    1) Analysis using *Windowsperf* GUI

CPython: Performance Analysis Demo

- CPython is the reference implementation of the Python programming language written in C.
- Explore CPython computation of a simple arithmetic operation.
  - Compute googolplex which is 10^(10^100).
- Prerequisites:
  - CPython debug-mode ARM64 build with executables, libraries and corresponding PDB files.
- Analysis demo include:
  - Manual top-down analysis and exploration.
  - Use of Arm Telemetry Solution metrics.
  - Narrow search with events deduced from top-down analysis.
  - Arm Telemetry Solution topdown-tool exploration.

CPython example – Cycle_Accounting counting + metrics

Arm Telemetry Solution Events & Metrics:

```
>wperf stat -m Cycle_Accounting -c 7 --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
counting ... done

Performance counter stats for core 7, no multiplexing, kernel mode excluded, on Arm Limited core implementation:
note: 'e' - normal event, 'gN' - grouped event with group number N, metric name will be appended if 'e' or 'g' comes
from it
         counter value  event name      event idx  event note
         =============  ==========      =========  ==========
        31,032,167,389  cycle           fixed      e
        31,032,167,389  cpu_cycles      0x11       g0,frontend_stalled_cycles
           536,087,085  stall_frontend  0x23       g0,frontend_stalled_cycles
        31,032,167,389  cpu_cycles      0x11       g1,backend_stalled_cycles
         4,773,104,887  stall_backend   0x24       g1,backend_stalled_cycles

Telemetry Solution Metrics:
        core  product_name  metric_name               value  unit
        ====  ============  ===========               =====  ====
           7  neoverse-n1   backend_stalled_cycles   15.381  percent of cycles
           7  neoverse-n1   frontend_stalled_cycles   1.728  percent of cycles

              10.916 seconds time elapsed
```

CPython example – manual top-down analysis

```
> wperf list
...
List of supported groups of metrics (to be used in -m)

Group                    Metrics
=====                    =======
Branch_Effectiveness     branch_mpki,branch_misprediction_ratio
Cycle_Accounting         frontend_stalled_cycles,backend_stalled_cycles
DTLB_Effectiveness       dtlb_mpki,l1d_tlb_mpki,l2_tlb_mpki,dtlb_walk_ratio,l1d_tlb_miss_ratio,l2_tlb_miss_ratio
General                  ipc
ITLB_Effectiveness       itlb_mpki,l1i_tlb_mpki,l2_tlb_mpki,itlb_walk_ratio,l1i_tlb_miss_ratio,l2_tlb_miss_ratio
L1D_Cache_Effectiveness  l1d_cache_mpki,l1d_cache_miss_ratio
L1I_Cache_Effectiveness  l1i_cache_mpki,l1i_cache_miss_ratio
L2_Cache_Effectiveness   l2_cache_mpki,l2_cache_miss_ratio
LL_Cache_Effectiveness   ll_cache_read_mpki,ll_cache_read_miss_ratio,ll_cache_read_hit_ratio
MPKI                     branch_mpki,itlb_mpki,l1i_tlb_mpki,dtlb_mpki,l1d_tlb_mpki,l2_tlb_mpki,
                         l1i_cache_mpki,l1d_cache_mpki,l2_cache_mpki,ll_cache_read_mpki
Miss_Ratio               branch_misprediction_ratio,itlb_walk_ratio,dtlb_walk_ratio,l1i_tlb_miss_ratio,
                         l1d_tlb_miss_ratio,l2_tlb_miss_ratio,l1i_cache_miss_ratio,l1d_cache_miss_ratio,
                         l2_cache_miss_ratio,ll_cache_read_miss_ratio
Operation_Mix            load_percentage,store_percentage,integer_dp_percentage,simd_percentage,
                         scalar_fp_percentage,branch_percentage,crypto_percentage
```

Cycle_Accounting & Operation_Mix Metric Groups

These two groups of metrics are candidates for top-level performance bottleneck analysis.

Cycle_Accounting

This metric group contains a set of metrics that measure the percentage of processor cycles stalled in either frontend or backend of the processor.

Operation_Mix

This metric group provides the distribution of micro-operation types executed for the program.

CPython example – Operation_Mix metrics collection

Arm Telemetry Solution Metrics:

```
>wperf stat -m Operation_Mix -c 7 --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
counting ... done

Performance counter stats for core 7, no multiplexing, kernel mode excluded, on Arm Limited core implementation:
note: 'e' - normal event, 'gN' - grouped event with group number N, metric name will be appended if 'e' or 'g' comes
from it

...

Telemetry Solution Metrics:
        core  product_name  metric_name            value  unit
        ====  ============  ===========            =====  ====
           7  neoverse-n1   branch_percentage      9.712  percent of operations
           7  neoverse-n1   crypto_percentage      0.000  percent of operations
           7  neoverse-n1   integer_dp_percentage 38.220  percent of operations
           7  neoverse-n1   load_percentage       37.766  percent of operations
           7  neoverse-n1   scalar_fp_percentage   0.000  percent of operations
           7  neoverse-n1   simd_percentage        0.021  percent of operations
           7  neoverse-n1   store_percentage      14.068  percent of operations

              10.953 seconds time elapsed
```

CPython example – Operation_Mix event counting

Arm Telemetry Solution Events:

```
>wperf stat -m Operation_Mix -c 7 --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
counting ... done

 counter value   event name        event idx  event note                multiplexed    scaled value
        =============  ==========         =========  ==========                ===========    ============
       29,281,324,658  cycle              fixed      e                             109/109  29,281,324,658
       19,185,118,243  inst_spec          0x1b       g0,load_percentage             37/109  56,518,321,310
        7,245,526,302  ld_spec            0x70       g0,load_percentage             37/109  21,344,928,835
       19,185,118,243  inst_spec          0x1b       g1,store_percentage            37/109  56,518,321,310
        2,699,003,768  st_spec            0x71       g1,store_percentage            37/109   7,951,119,208
        7,332,463,151  dp_spec            0x73       g2,integer_dp_percentage       37/109  21,601,040,093
       19,185,118,243  inst_spec          0x1b       g2,integer_dp_percentage       37/109  56,518,321,310
            4,081,505  ase_spec           0x74       g3,simd_percentage             36/109      12,357,890
       19,283,621,140  inst_spec          0x1b       g3,simd_percentage             36/109  58,386,519,562
       19,283,621,140  inst_spec          0x1b       g4,scalar_fp_percentage        36/109  58,386,519,562
                    0  vfp_spec           0x75       g4,scalar_fp_percentage        36/109               0
        1,761,446,072  br_immed_spec      0x78       g5,branch_percentage           36/109   5,333,267,273
          136,013,914  br_indirect_spec   0x7a       g5,branch_percentage           36/109     411,819,906
       19,536,695,976  inst_spec          0x1b       g5,branch_percentage           36/109  59,152,773,927
                    0  crypto_spec        0x77       g6,crypto_percentage           36/109               0
       19,536,695,976  inst_spec          0x1b       g6,crypto_percentage           36/109  59,152,773,927
```

CPython example – Operation_Mix metrics breakdown

Arm Telemetry Solution Metrics:

```
integer_dp_percentage = ((dp_spec / inst_spec) * 100)   // Integer Operations Percentage
      load_percentage = ((ld_spec / inst_spec) * 100)   // Load Operations Percentage
     store_percentage = ((st_spec / inst_spec) * 100)   // Store Operations Percentage
```

Events:

- **dp_spec** – Operation speculatively executed, integer data processing. Counts speculatively executed logical or arithmetic instructions such as MOV/MVN operations.
- **ld_spec** – Operation speculatively executed, load. Counts speculatively executed load operations including Single Instruction Multiple Data (SIMD) load operations.
- **st_spec** – Operation speculatively executed, store. Counts speculatively executed store operations including Single Instruction Multiple Data (SIMD) store operations.
- **inst_spec** - Operation speculatively executed. Counts operations that have been speculatively executed.

CPython example – ld_spec sampling

```
>wperf record -e ld_spec -c 7 --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
base address of 'cpython\PCbuild\arm64\python_d.exe': 0x7ff692041288, runtime delta: 0x7ff552040000
sampling ............. done!
======================== sample source: ld_spec, top 50 hot functions ========================
        overhead  count  symbol
        ========  =====  ======
           78.24    266  x_mul:python312_d.dll
            6.76     23  v_isub:python312_d.dll
            4.41     15  _Py_atomic_load_32bit_impl:python312_d.dll
            2.65      9  PyErr_CheckSignals:python312_d.dll
            2.35      8  unknown
            2.06      7  v_iadd:python312_d.dll
            1.47      5  x_add:python312_d.dll
            0.59      2  _Py_atomic_load_64bit_impl:python312_d.dll
            0.29      1  _PyMem_DebugRawAlloc:python312_d.dll
            0.29      1  pymalloc_free:python312_d.dll
            0.29      1  pymalloc_alloc:python312_d.dll
            0.29      1  write_size_t:python312_d.dll
            0.29      1  _Py_ThreadCanHandleSignals:python312_d.dll
100.00%        340  top 13 in total
```

CPython example – ld_spec sampling + annotate

```
>wperf record -e ld_spec -c 7 --annotate --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
base address of 'cpython\PCbuild\arm64\python_d.exe': 0x7ff692041288, runtime delta: 0x7ff552040000
sampling ............. done!
======================== sample source: ld_spec, top 50 hot functions ========================
x_mul:python312_d.dll
        line_number  hits  filename
        ===========  ====  ========
        3,591        92    C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,590        54    C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,593        54    C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,594        50    C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,592        19    C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,595        3     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,569        2     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,571        2     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,588        1     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,600        1     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,601        1     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,602        1     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
        3,604        1     C:\Users\przemek\Desktop\wperf\merge-request\3.4.3\cpython\Objects\longobject.c
...
```

CPython example – ld_spec sampling + disassemble

```
>wperf record -e ld_spec -c 7 --disassemble --timeout 10 -- cpython\PCbuild\arm64\python_d.exe -c 10**10**100
base address of 'cpython\PCbuild\arm64\python_d.exe': 0x7ff692041288, runtime delta: 0x7ff552040000
sampling ............ done!
======================== sample source: ld_spec, top 50 hot functions ========================
x_mul:python312_d.dll
        line_number  hits  filename                      instruction_address  disassembled_line
        ===========  ====  ========                      ===================  =================
        3,591        99    cpython\Objects\longobject.c  3e8e2c                 address  instruction
                                                                                =======  ===========
                                                                                3e8e28   ldr x8, [sp, #0x10]
                                                                                3e8e2c   and x8, x8, #0x3fffffff
                                                                                3e8e30   mov w8, w8
                                                                                3e8e34   ldr x9, [sp, #0x20]
                                                                                3e8e38   str w8, [x9]
                                                                                3e8e3c   ldr x8, [sp, #0x20]
                                                                                3e8e40   add x8, x8, #0x4
                                                                                3e8e44   str x8, [sp, #0x20]
...
```

CPython example – next steps

- Sample for all hot spot candidates based on identified metric(s) and its events: dp_spec, ld_spec, and st_spec.
- Wash, rinse, repeat:
  - Isoate hot spot: algorithm / module / source file / function
  - Application specific analysis of identified hot spots.
  - Improve code -> benchmark / performance analysis.
- Use topdown-tool to perform workload Telemetry Solution top-down methodology:

```
>topdown-tool cpython\PCbuild\arm64\python_d.exe -c 10**10**100
```

CPython example – Arm topdown-tool example output

Arm Telemetry Solution Methodology & Metrics

```
Stage 1 (Topdown metrics)
=========================
[Cycle Accounting]
Frontend Stalled Cycles............. 1.63% cycles
Backend Stalled Cycles.............. 15.37% cycles

Stage 2 (uarch metrics)
=======================
[Branch Effectiveness]
  (follows Frontend Stalled Cycles)
Branch Misprediction Ratio.......... 0.005 per branch
Branch MPKI......................... 0.515 misses per 1,000 instructions

[Data TLB Effectiveness]
  (follows Backend Stalled Cycles)
DTLB MPKI........................... 0.001 misses per 1,000 instructions
DTLB Walk Ratio..................... 0.000 per TLB access
L1 Data TLB Miss Ratio.............. 0.000 per TLB access
L1 Data TLB MPKI.................... 0.120 misses per 1,000 instructions
L2 Unified TLB Miss Ratio........... 0.002 per TLB access
L2 Unified TLB MPKI................. 0.001 misses per 1,000 instructions

[General]
Instructions Per Cycle.............. 1.923 per cycle

...
```

```
[Miss Ratio]
Branch Misprediction Ratio.......... 0.005 per branch
DTLB Walk Ratio..................... 0.000 per TLB access
ITLB Walk Ratio..................... 0.000 per TLB access
L1D Cache Miss Ratio................ 0.000 per cache access
L1 Data TLB Miss Ratio.............. 0.000 per TLB access
L1I Cache Miss Ratio................ 0.000 per cache access
L1 Instruction TLB Miss Ratio....... 0.003 per TLB access
L2 Cache Miss Ratio................. 0.010 per cache access
L2 Unified TLB Miss Ratio........... 0.002 per TLB access
LL Cache Read Miss Ratio............ 0.988 per cache access

[Speculative Operation Mix]
  (follows Backend Stalled Cycles)
Branch Operations Percentage........ 9.71% operations
Crypto Operations Percentage........ 0.00% operations
Integer Operations Percentage....... 38.23% operations
Load Operations Percentage.......... 37.77% operations
Floating Point Operations Percentage 0.00% operations
Advanced SIMD Operations Percentage. 0.02% operations
Store Operations Percentage......... 14.08% operations
```

Synthetic workload example performance analysis

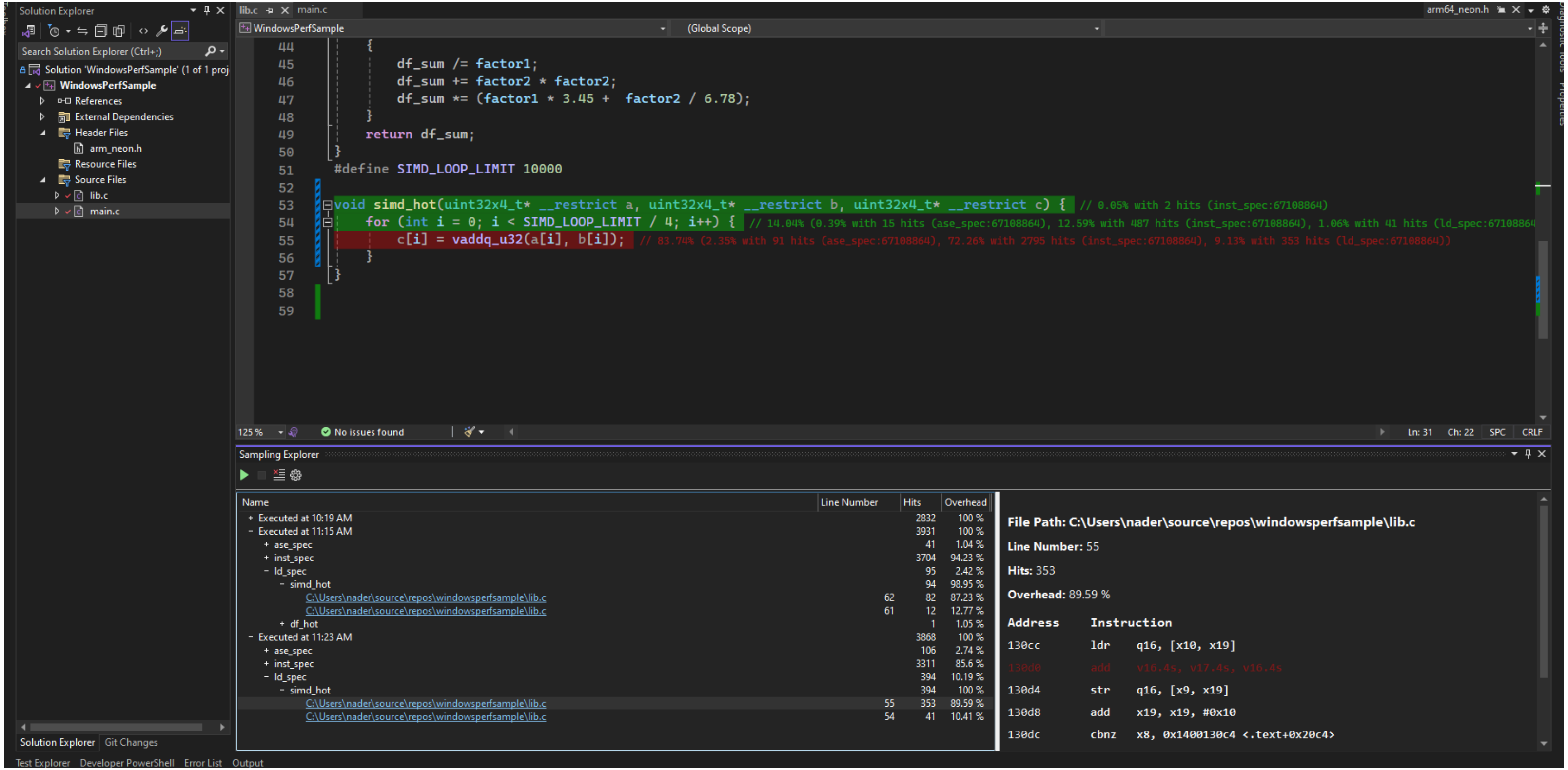
- Synthetic workload designed to exploit certain core PMU events.
- Integration with Visual Studio via Extension.
- Simple developer workflow for narrow scopes.
- Predetermined known hot spot / function / algorithm.

Synthetic workload example – WindowsPerf GUI

```
#define SIMD_LOOP_LIMIT 10000
void simd_hot(unsigned int * __restrict a,
              unsigned int * __restrict b, unsigned int * __restrict c)
{
    for (int i = 0; i < SIMD_LOOP_LIMIT; i++)
        a[i] = b[i] + c[i];
}


//Candidate for optimisation
void simd_hot4(uint32x4_t* __restrict a,
               uint32x4_t* __restrict b, uint32x4_t* __restrict c) {
    for (int i = 0; i < SIMD_LOOP_LIMIT/4; i++) {
        c[i] = vaddq_u32(a[i] , b[i]);
    }
}
```

Synthetic workload example – WindowsPerf GUI



WindowsPerf Reference

- Blog Posts
  - Introducing 1.0.0-beta release of WindowsPerf Visual Studio extension
  - Introducing the WindowsPerf GUI: the Visual Studio 2022 extension
  - Announcing WindowsPerf: Open-source performance analysis tool for Windows on Arm
  - WindowsPerf Release 2.4.0
  - WindowsPerf Release 2.5.1
  - WindowsPerf Release 3.0.0
  - WindowsPerf Release 3.3.0
- External Documentation
  - Perf for Windows on Arm (WindowsPerf)
  - Get started with WindowsPerf
  - Sampling CPython with WindowsPerf

Arm Telemetry Solution References

- Arm Telemetry Solution
  - Arm CPU Topdown Methodology Specification
  - Arm Neoverse CPU Telemetry Specifications
- Arm Telemetry Solution Gitlab
  - Arm Topdown Tool
- Arm Topdown-tool Install Guide

Thank you