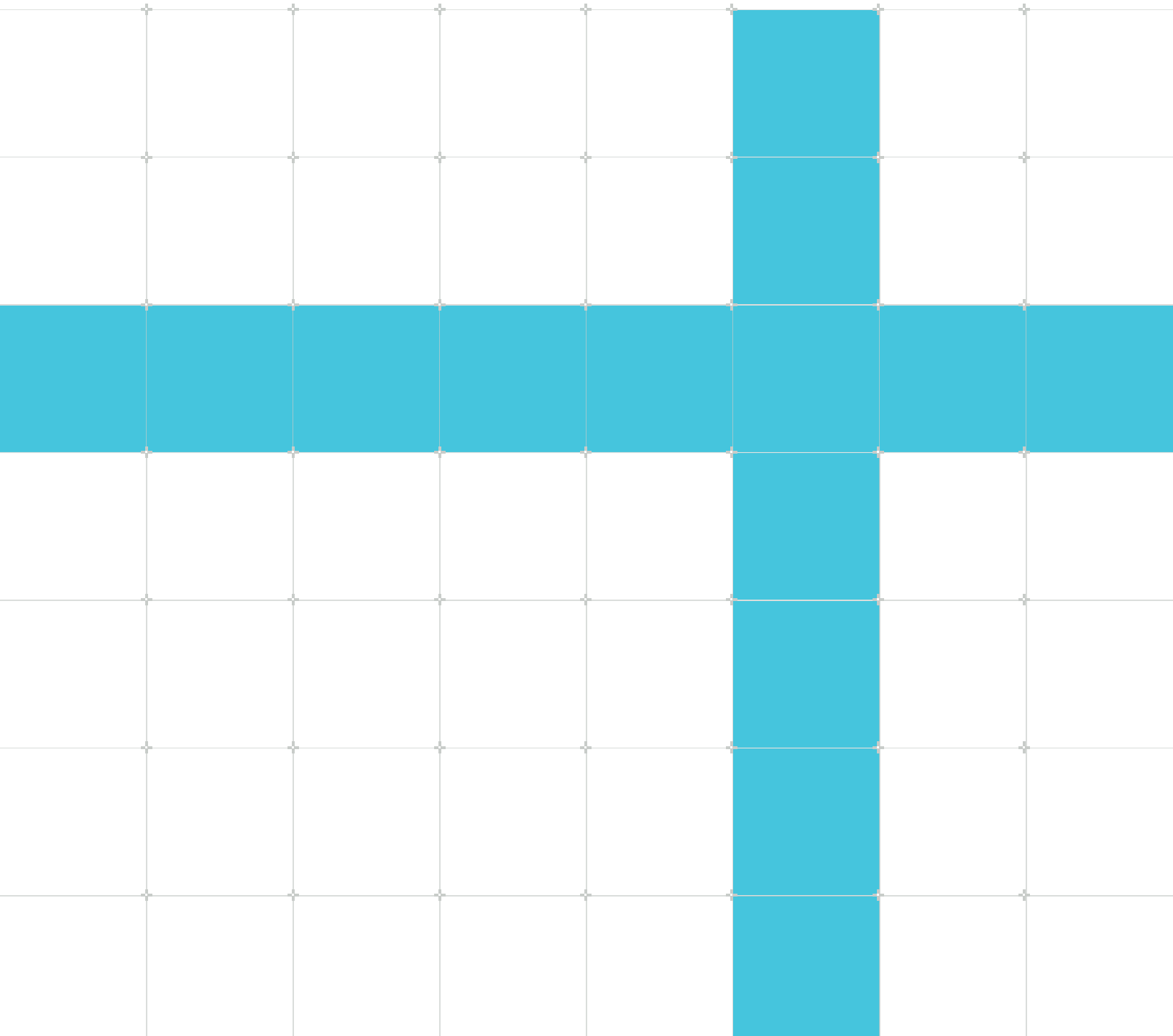
Arm® CPU Telemetry Solution

Version 1.0

Topdown Methodology Specification

Non-Confidential

Issue 01
109542_0100_01_en



Arm® CPU Telemetry Solution

Topdown Methodology Specification

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	26 January 2024	Non-Confidential	Initial release

Proprietary Notice

and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Caution

Recommendations. Not following these recommendations might lead to system failure or damage.

Warning

Requirements for the system. Not following these requirements might result in system failure or damage.

Requirements for the system. Not following these requirements will result in system failure or damage.

An important piece of information that needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

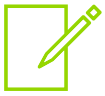
Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Telemetry on Arm Developer	–	Non-Confidential
Arm® Telemetry Solution GitLab repository	–	Non-Confidential

Arm architecture and specifications	Document ID	Confidentiality
Arm® Architecture Reference Manual for A-profile architecture	DDI 0487	Non-Confidential

Non-Arm resources	Document ID	Organization
A. Yasin, IEEE Xplore, "A Top-Down method for performance analysis and counters architecture"	–	A. Yasin, "A Top-Down method for performance analysis and counters architecture," 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459.
Linux kptr-restrict	–	Linux kptr-restrict
Linux perf-annotate	–	Linux perf-annotate
Linux perf-record	–	Linux perf-record
Linux perf-report	–	Linux perf-report
Linux perf-stat	–	Linux perf-stat
Linux perf_event and tool security, Unprivileged users	–	Linux perf_event and tool security, Unprivileged users
Linux perf_event_paranoid	–	Linux perf_event_paranoid
WindowsPerf GitLab repository	–	Linaro WindowsPerf GitLab repository


Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. About Arm CPU Telemetry Solution

Arm CPU Telemetry Solution comprises a set of components, that is, a top-down performance analysis methodology, a standardized telemetry framework, and profiling tool. It is designed to use a CPU's telemetry data to help identify performance bottlenecks and improve execution efficiency by using these components.

System-on-Chip (SoC) telemetry enables collection and analysis of hardware execution information from a platform to gain insights about a system's performance. This information can be used to identify performance issues and improve execution efficiency.

A modern CPU contains a hardware performance monitoring unit (PMU) which provides a set of functional events that describe the internal state of the microarchitecture during execution. Multiple events can be used to derive metrics, which provide more meaningful insights by abstracting the hardware details.

Performance analysis is an investigative and diagnostic process. Using a systematic methodology is important to narrow down the bottleneck for root cause analysis and code execution improvements. A well-known example of a systematic methodology is the top-down performance analysis methododology. For more information, see *A. Yasin, IEEE Xplore, "A Top-Down method for performance analysis and counters architecture"*.

Arm CPU Telemetry Solution is designed to collect, represent, and analyze CPU telemetry data on Arm CPU based platforms. A supported Arm CPU implementation provides a telemetry specification that defines the hardware PMU events, derived metrics, and Arm Topdown methodology supported by the CPU. Arm Topdown methodology is Arm's implementation of the top-down performance analysis methododology. It is a approach to consume the telemetry events and metrics in a hierarchical decision tree format for hotspot analysis. See Arm Topdown methodology.

The methodology, metrics, and events are represented in a standardized telemetry framework, Arm Telemetry framework that is designed to support the collection and processing of large amounts of CPU telemetry data. In this framework, events are categorized into function groups and metric groups that can be collected in stages, with the help of profiling tools. This framework also helps to manage and represent the telemetry specification of supported CPUs in a standardized machine-readable format, which can be harnessed by profiling tools.

Arm CPU Telemetry Solution also provides the Arm Topdown tool. It is a simple command line tool to profile applications. The tool parses the telemetry machine-readable specification (MRS), which is a JSON file, to collect and process the telemetry data that is supported by the CPU to provide performance insights. Arm Topdown tool is enabled for both Linux and Windows platforms.

For more information about Arm CPU Telemetry Solution, see *Arm® Telemetry on Arm Developer*.

3. Arm Topdown methodology

Arm Topdown methodology supports performance analysis, workload characterization, and microarchitecture exploration on Arm A-profile CPUs that implement version 3 of the Performance Monitors Extension, `FEAT_PMUv3`.

For more information about the Performance Monitors Extension, see *Arm® Architecture Reference Manual for A-profile architecture*.

This performance analysis methodology provides a systematic and top-down hierarchical approach to use hardware performance monitoring events for analysis use cases. The methodology is formulated with the help of performance metrics derived from the hardware monitoring events, making the hardware monitors accessible for an average software user without extensive microarchitectural knowledge. Computer architects and system designers can also use it for resource characterization and platform tuning by using the specified set of performance metrics.

To identify the application bottleneck for a CPU, the first step is to characterize the execution efficiency of the cycles spent. This workload characterization can be measured by the distribution of execution cycles broken down between:

- Those cycles that worked efficiently.
- Those cycles that are wasted by pipeline stalls and redirections.

The process of breaking down the cycles can be done hierarchically to narrow down the bottleneck in the CPU pipeline. It also helps to identify and ignore the CPU components that are not causing the performance issue.

After identifying the CPU component bottleneck in the hardware, the next step is to measure the microarchitectural metrics of that component for further root cause analysis.

To aid the investigative and diagnosis process, Arm Topdown methodology is conducted in two stages:

Stage 1: Topdown analysis

The first stage is to perform Topdown analysis. It uses hierarchical pipeline stall-related metrics to detect and identify the performance bottleneck in the CPU. For more information, see Stage 1: Topdown analysis.

Stage 2: Microarchitecture exploration

The second stage is to conduct microarchitecture exploration to further analyze bottlenecked CPU resources. It uses a set of CPU resource effectiveness metrics. For more information, see Stage 2: Microarchitecture exploration.

After completing stage 1, the Arm Topdown methodology provides recommended stage 2 metrics to further analyze the identified the bottleneck. Stage 2 metrics can be used directly for a targeted analysis of a CPU resource.

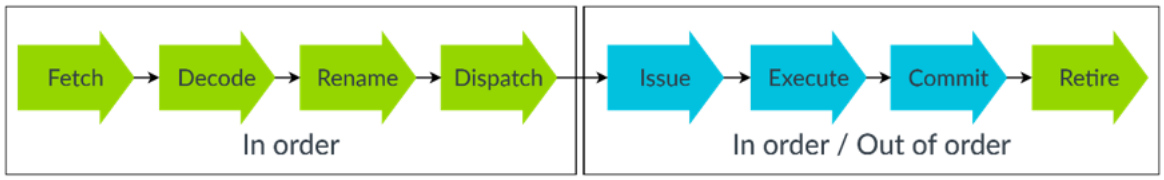
3.1 Stage 1: Topdown analysis

Topdown analysis is the first stage of the methodology to support hotspot detection. A set of pipeline efficiency metrics specify the PMU events to measure, which helps to characterize the distribution of cycles spent by the CPU.

Topdown analysis metrics are formulated as a decision tree of metrics that need to be hierarchically traversed, in a predefined path, to help locate the bottleneck.

An Arm CPU implementation consists of a frontend that fetches instructions from memory, decodes them, and dispatches them to a backend. The backend executes those instructions, including reading and writing data from and to memory. As shown in the pipeline model in the following figure, the frontend is often in order, whereas the backend can be in order or out of order depending on the CPU's microarchitecture.

Figure 3-1: An Arm CPU implementation pipeline model



In a simple scalar CPU implementation, the frontend dispatches a maximum of one instruction for each cycle. In this case, a cycle is classified as follows:

- Useful cycle, whereby an instruction were dispatched
- Stalled cycle, whereby no instruction was dispatched

In a superscalar CPU implementation, the frontend can dispatch many instructions for each cycle using instruction level parallelism, thereby achieving a higher Instructions Per Cycle (IPC). Each instruction takes one of an implementation-defined maximum number of dispatch slots on each cycle. Therefore, the utilization and efficiency of cycles spent is more accurately calculated by counting the number of dispatch slots in which instructions were dispatched and those slots that did not dispatch instructions.

A modern CPU implementation can also break instructions into micro-operations for execution. The frontend of the CPU decodes and decomposes instructions to micro-operations that can be executed by the backend execution units. In this case, the utilization and efficiency of a pipeline is measured by examining cycles or slots on which no operations are dispatched, referred to as stalled cycles or stalled slots.

To keep the CPU backend fed with instructions, the frontend also predicts the future instructions to be executed. The success of future instructions is highly dependent on the control flow of the code that is determined by branched instructions. When these predictions are wrong, the instructions dispatched to the backend are canceled and can cause pipeline bubbles.

On superscalar CPU implementations, the total execution bandwidth of the CPU can be measured in terms of the number of execution slots for operations. The total number of slots supported by the core determines the execution bandwidth of the CPU for top-down accounting and is defined as a microarchitectural parameter. The value of the parameter is used to derive the execution bandwidth metrics for a CPU.

These key characteristics determine efficient pipeline execution. Thus, the Arm Topdown analysis starts by calculating the following measurements to detect inefficiencies. Each measurement is a percentage of the total execution bandwidth of the CPU:

- The percentage of execution bandwidth used by operations that are retired.
- The percentage of execution bandwidth lost to mis-speculation.
- The percentage of execution bandwidth lost to stalls in the frontend.
- The percentage of execution bandwidth lost to stalls in the backend.

These measurements that form the metrics for the Topdown level 1 analysis are defined as follows:

`retiring`
: This metric is the percentage of total slots that retired operations. It indicates the proportion of cycles that were used and efficient.

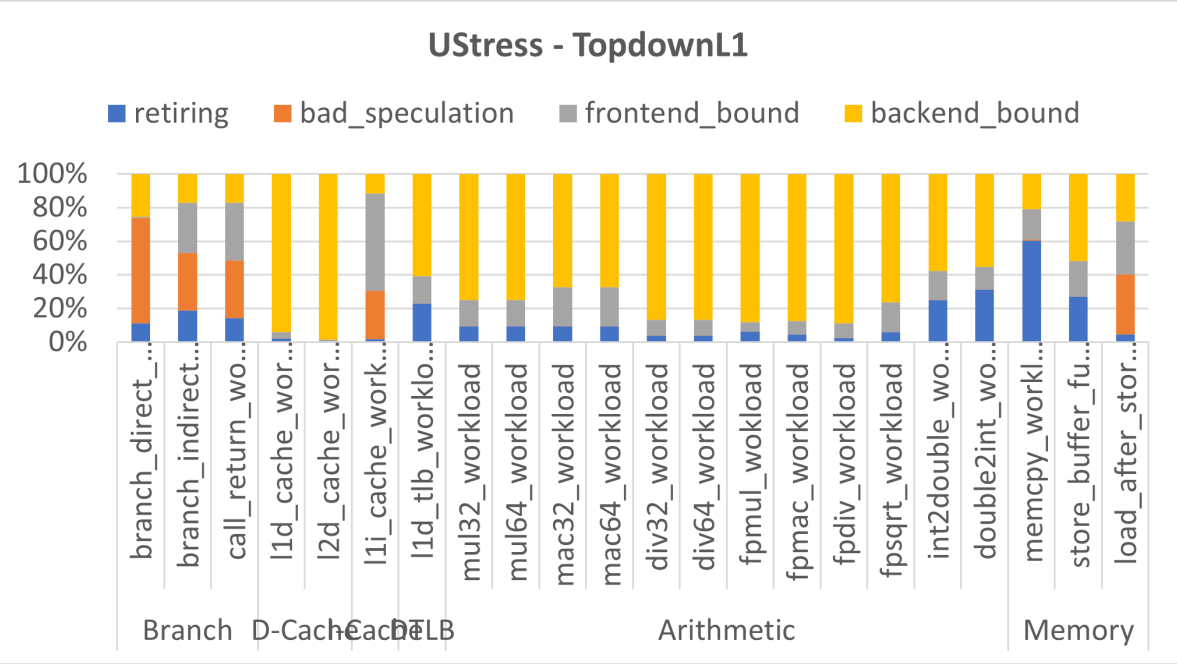
`bad_speculation`
: This metric is the percentage of total slots that executed operations but did not retire due to a pipeline flush caused by mis-speculation. It indicates the cycles that were used but were inefficient executing the wrong instructions. It also includes cycles spent recovering from the pipeline flush, which requires an instruction pipeline refill from the correct instruction location.

`frontend_bound`
: This metric is the percentage of total slots that were stalled due to resource constraints in the frontend unit of the CPU.

`backend_bound`
: This metric is the percentage of total slots that were stalled due to resource constraints in the backend unit of the CPU.

The following figure shows an example of Topdown level 1 analysis that was conducted for a microbenchmark designed to stress key CPU blocks.

Figure 3-2: Ustress Topdown level 1 metrics (TopdownL1)



After the first level of accounting, Arm Topdown methodology can then support lower levels in the hierarchy, to further break down the stalling events.

A relatively high `frontend_bound` metric shows that execution cycles are being wasted due to pipeline stalls in the in-order frontend of the CPU. Many causes can exist for these stalls, such as:

- Inefficiency in the branch prediction unit
- Fetch latency due to instruction cache misses
- Translation delays caused by instruction Translation Lookaside Buffer (TLB) walks

For example, if the frontend is stalled and there is an instruction cache miss in progress, then the stall might be due to the cache miss and hence attributed to it.

A relatively high `backend_bound` metric shows that execution cycles are wasted due to pipeline stalls in the backend of the CPU. Many causes can exist for these stalls such as:

- Inefficiency in the backend execution units
- Data cache misses
- Translation delays caused by data TLB walks

A relatively high `bad_speculation` metric shows that the pipeline stalls can be due pipeline flushes or machine clears that break the pipeline requiring a control flow change. The major causes for these stalls are branch mis-predictions and exceptions.

A relatively high `retiring` metric shows that the pipelines were utilized. However, this metric can indicate inefficiency due to underutilization of the microarchitectural capabilities. For example,

scalar execution of a code that should have been performed more efficiently using vector operations.

After the level 1 metrics, second level analysis mainly determines whether a pipeline stall is due to memory or processor effects, for example:

- An example of a memory effect is a cache miss. If memory effects dominate, then the program is referred to as memory bound.
- An example of a processor effect is when a backend is full of operations waiting for execution units or results of the previous operations to become available. If processor effects dominate, then the program is referred to as CPU bound.

These stalls might also break down into third, fourth level metrics, such as, the specific cases summarized in the previous memory and processor effects examples. Support for the lower levels can also be microarchitecture dependent.


Note

Refer to Topdown analysis tree of the Arm CPU implementation to understand what the CPU supports, for example, in an Arm core telemetry specification. For more information about available Arm core telemetry specifications, see *Arm® Telemetry on Arm Developer*.

Topdown event implementations for Arm CPU microarchitecture

It is not always practical or feasible to continue down the levels of top-down accounting, especially to track slot level. The lower the level, the higher the number of events that occur simultaneously.

The lower the level, the smaller it becomes, and any signal causing the bottleneck can get lost in the noise. Thus, to precisely attribute the causes to effects becomes more difficult. Tracking cause to effect can involve a lot of hardware which is:

- Difficult to design
- Difficult to validate
- Consumes area and power that can be otherwise spent on additional processing resources

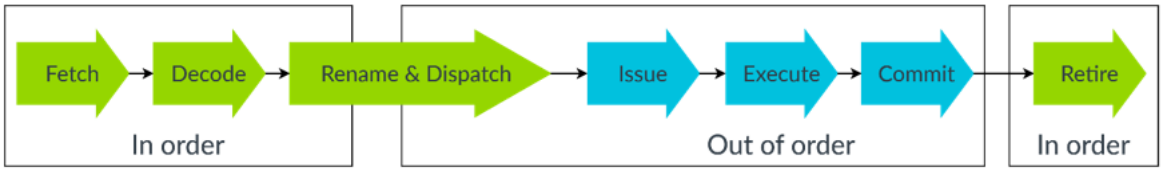
As a result, the accounting of events takes the form of: the counter counts each cycle by `STALL_<other event>` where `<a condition applies>`. That is, these events count the coincidence of a stall with the applied condition.

Although described as being coincidental, CPU implementations might choose to delay some event signals while deciding whether the event occurs. Because of pipelining, a stall that results from a condition might not be immediately possible once the condition becomes active. To improve the quality of the event data, the condition might be delayed to determine whether it is coincidental with a stall. That is, `<a condition applies>` becomes `<a condition applied after a fixed, implementation-specific, number of cycles>`.

For the purpose of stall accounting in Arm A-profile “big” CPUs, the boundary between frontend and backend is at the point of rename and dispatch as shown in the following figure. It is the point in the pipeline where the core switches from an in-order pipeline to an out-of-order pipeline. It

also marks the point where resource contention that causes a stall condition is associated with the throughput in the out-of-order division of the Arm A-profile "big" CPU. Examples of resource contention are execution bandwidth and physical register renames.

Figure 3-3: Frontend and backend division in an Arm "big" CPU with an out-of-order backend



Arm architecture defines some key Topdown events that support the Topdown metrics. The following table show some example events used in the Topdown level 1 metric definitions of Arm processor IP that have adopted Arm CPU Telemetry Solution, referred to as Arm cores in this specification.

Table 3-1: Topdown analysis event support in Arm cores

Topdown event category	Stall events	Supported Arm cores
Overall Stalls	STALL	Neoverse™ N2, V1, V2
	STALL_SLOT	Neoverse N2, V1, V2
Frontend Stall Events	STALL_FRONTEND	Neoverse N1, N2, V1, V2
	STALL_SLOT_FRONTEND	Neoverse N2, V1, V2
Backend Stall Events	STALL_BACKEND	Neoverse N1, N2, V1, V2
	STALL_SLOT_BACKEND	Neoverse N2, V1, V2

Note

Refer to the Topdown methodology tree in each supported Arm core for the complete list of events, hierarchy of metrics derived from these events, and event relationships. A Topdown methodology may include common architectural events across Arm processor families and a set of events which can be microarchitecture dependent. For more information about available Arm core telemetry specifications, see *Arm® Telemetry on Arm Developer*.

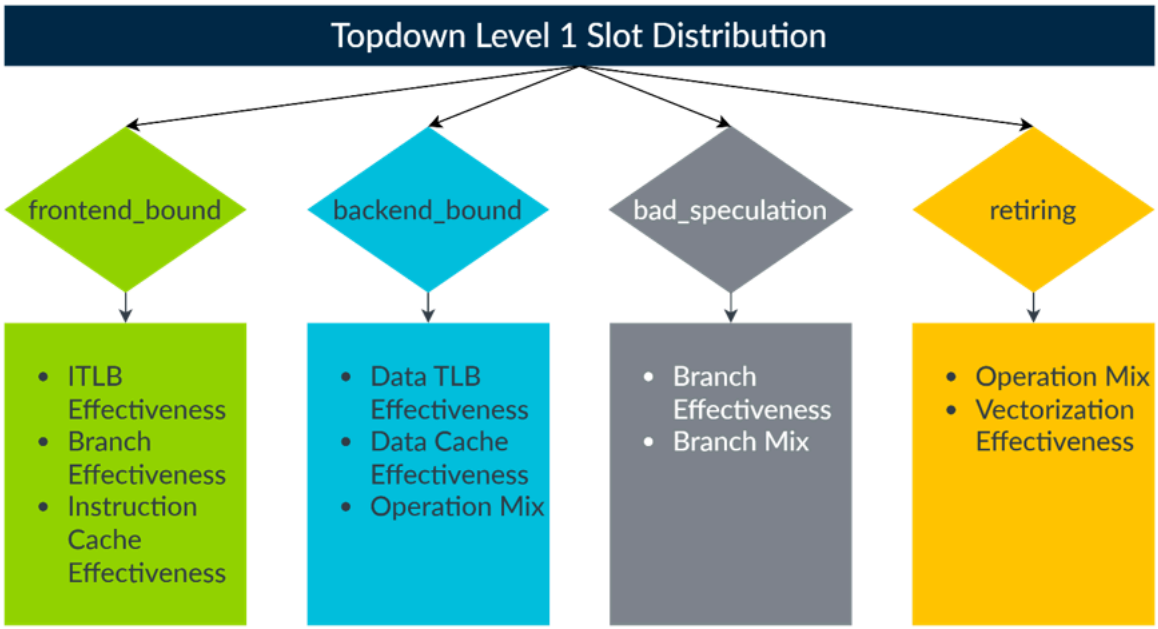
3.2 Stage 2: Microarchitecture exploration

After the potential hotspot in the CPU pipeline is identified in stage 1, the next stage is to conduct a microarchitectural analysis of the CPU pipeline resource causing the bottleneck.

Stage 2 is defined as the microarchitecture exploration stage for which a set of CPU resource effectiveness metrics are defined under metric groups for each resource. Industry-standard metrics such as Misses Per Kilo Instructions (MPKI) and Miss Ratios are metric groups that are also defined in this stage.

A relatively high frontend stall rate indicates that cycles are wasted due to pipeline stalls in the in-order frontend of the CPU. A relatively high backend stall rate indicates that cycles are wasted due to pipeline stalls in the backend. This breakdown helps to narrow down the dominating CPU blocks that can be further analyzed to identify performance bottlenecks as shown in the following figure.

Figure 3-4: Stage 2 metrics for microarchitecture exploration



The list of major CPU resource effectiveness metrics to further analyze a `frontend_bound` workload are as follows:

- Instruction TLB (ITLB) Effectiveness metrics
- Instruction Cache Effectiveness metrics
- Branch Effectiveness metrics

The list of major CPU resource effectiveness metrics to further analyze a `backend_bound` workload is as follows:

- Data TLB Effectiveness metrics
- Data Cache Effectiveness metrics
- Operation Mix metrics

The list of major CPU resource effectiveness metrics to further analyze a `bad_speculation` workload is as follows:

- Branch Effectiveness metrics
- Branch Mix metrics

The list of major CPU resource effectiveness metrics to further analyze a `retiring` workload is as follows:

- Operation Mix metrics
- Vectorization Effectiveness metrics

Performance analysis metrics supported by each Arm core depend on the hardware events implemented by the core. For more information about the metric groups and the list of metrics within each group, see the relevant Arm core telemetry specification. For more information about available Arm core telemetry specifications, see *Arm® Telemetry on Arm Developer*.

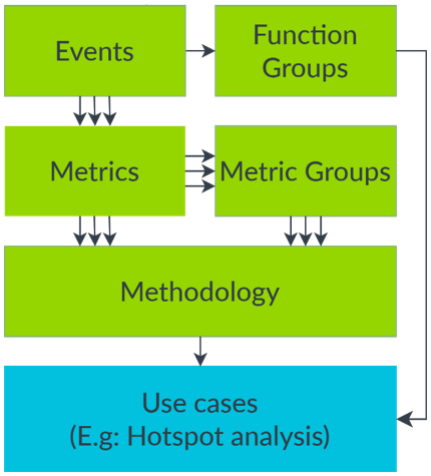
4. Arm Telemetry framework for CPUs

Arm Telemetry framework for CPUs has key elements that follow a standardized data model to produce and consume the CPU's telemetry data. The data model helps specify the telemetry features for the end users and profiling tools.

This framework supports all collaterals for telemetry, that is, written specifications for end users, for example, software analysts or system designers, and machine-readable specifications for profiling tools.

The main elements of the framework are events, metrics, metric groups, and methodology. The following figure shows those elements as a high-level data model.

Figure 4-1: Data model for Arm Telemetry framework for CPUs



PMU events implemented by an Arm core are either:

- Common, that is, the events are defined by the Arm architecture.
- **IMPLEMENTATION DEFINED**, that is, the events are specific to an Arm CPU.

Events are also either:

- Architectural - the events give the same result for the same workload on all Arm cores implementing the Arm architecture, to within a reasonable degree of accuracy, as described by the Arm architecture.
- Microarchitectural - the events give different results for the same workload on different Arm cores.

All events supported by an Arm core are grouped by function for each CPU resource in which they are counted. Metrics are derived from events that support the Topdown methodology and are grouped into metric groups for analysis. Key metrics can be standardized for an Arm processor product family. Event implementations that form a metric can vary for each microarchitectural

requirement. This design approach standardizes and abstracts metrics such that an analyst with software expertise can consume the events directly without knowing the hardware design in depth.

Each definition of the framework element contains an example from the Arm Machine Readable Specification schema. All Arm cores that support the Arm Telemetry framework publish the associated Arm core telemetry specification with the MRS source data to the Arm CPU Telemetry Solution data repository in GitLab: *Arm® Telemetry Solution GitLab repository*

Events

Hardware performance monitoring events implemented by the CPU that contain raw data read from the registers or memory buffers.

Data fields

- Event Code
- Event Mnemonic
- Event Title
- Description
- Functional Category

Example event

```
"L1I_CACHE_REFILL": {
  "code": "0x0001",
  "title": "Level 1 instruction cache refill",
  "description": "Counts cache line refills in the level 1 instruction
cache caused by a missed instruction fetch. Instruction fetches may include
accessing multiple instructions, but the single cache line allocation is
counted once.",
  "common": true,
  "accesses": [
   "PMU",
   "ETE"
  ],
  "architectural": false,
  "impdef": false
}
```

Metrics

Derived mathematical relationships between events that provide insight into the system behavior. They are developed to abstract hardware details of the events from consumers of the telemetry data.

Data fields

- Metric Name
- Metric Title
- Metric Description
- Metric Formula
- Metric Unit
- Metric Events

Example metric

```
"l1i_cache_mpki": {
   "title": "L1I Cache MPKI",
   "formula": "((L1I_CACHE_REFILL / INST_RETIRED) * 1000)",
   "description": "This metric measures the number of level 1 instruction cache
 accesses missed per thousand instructions executed.",
   "units": "MPKI",
   "events": [
     "INST_RETIRED",
     "L1I_CACHE_REFILL"
    ]
 }
```

Metric groups

Groups of metrics that are analyzed together for performance correlation during a specific investigation. A group can be dependent on a usage model. Thus, some metrics can belong to multiple metric groups.

Data fields

Metric Group Name

Metric Group Title

Metric Group Description

Metric Group Metrics

Example metric group

```
"Topdown_L1": {
  "title": "Topdown Level 1",
  "description": "This metric group contains the first set of metrics to begin
 topdown analysis of application performance, which provide the percentage
 distribution of processor pipeline utilization.",
  "metrics": [
    "frontend_bound",
    "backend_bound",
    "retiring",
    "bad_speculation"
     ]
 }
```

Events required for the metrics can be obtained from the metric data schema.

Methodology

Methodology is actionable guidance to explain how to consume the different metrics and events for a specific usage model, for example, hotspot analysis. Methodology can be a metric group or collection of metric groups. It provides actions and guidance notes on how to consume the data to take steps or derive actionable insights from the data.

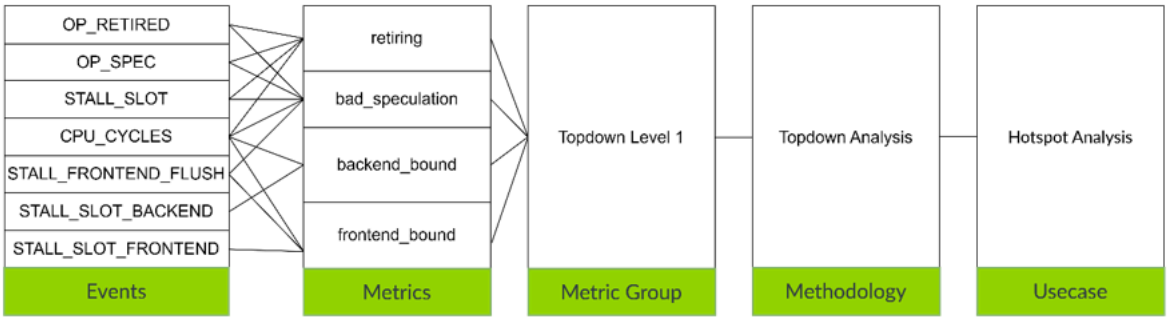
An example methodology is Arm Topdown methodology. It is a decision tree with root and leaf nodes of metrics that belong to specific metric groups. Metric relationships are key to represent the Topdown tree. Metric groups are required for the analysis process. They provide guidance for profiling data collection tools. Guidance includes how many hardware monitoring events to count, and in which order, because hardware resources are limited. Relationships between metrics and metric groups is specified for a methodology. Stage 1 and stage 2 metric relationships are captured in the methodology specification schema.

Topdown Methodology schema

```
"topdown_methodology": {
  "title": "Topdown Methodology",
  "description": "Topdown Performance Analysis Methodology",
  "metric_grouping": {
     "stage_1": [
     "Topdown_L1"
      ],
     "stage_2": [
       "Cycle_Accounting",
       "General",
       "MPKI",
       "Miss_Ratio",
       "Branch_Effectiveness",
       "ITLB_Effectiveness",
       "DTLB_Effectiveness",
       "L1I_Cache_Effectiveness",
       "L1D_Cache_Effectiveness",
       "L2_Cache_Effectiveness",
       "LL_Cache_Effectiveness",
       "Operation_Mix"
      ]
   },
  "decision_tree": {
     "root_nodes": [
       "frontend_bound",
       "backend_bound",
       "retiring",
       "bad_speculation"
       ],
    },
  "metrics": {
     "name": "frontend_bound",
     "group": "Topdown_L1",
     "next_items": [
       "Branch_Effectiveness",
       "ITLB_Effectiveness",
       "L1I_Cache_Effectiveness",
       "L2_Cache_Effectiveness",
       "LL_Cache_Effectiveness"
      ],
     "sample_events": [
       "STALL_SLOT_FRONTEND",
       "STALL_FRONTEND"
      ]
    }
 }
```

The following figure shows how the different elements of the Arm Telemetry framework are used to construct the level 1 metrics of the Topdown methodology, grouped as the Topdown level 1 metric group.

Figure 4-2: An example demonstration of Telemetry framework relationship



Because hardware counters are limited, it is important that the profiling data collection tool has sufficient counters to collect the data required for an analysis stage or a metric group. For the Topdown analysis level 1 metrics, as shown in the previous figure, a total of seven events are required for full coverage. The number of counters required will vary for each CPU, depending on the events required to derive the metrics. Software uses multiplexing when collecting metrics in a group that requires more events than the number of available hardware counters. This discrepancy can result in accuracy issues. Thus, hardware requirements for supported counters must be derived for each methodology requirement to achieve a reliable solution.

4.1 Data model standardization

All components of the Arm CPU Telemetry Solution are designed for scalability to enable an Arm A-profile CPU implementation to adopt this solution. The elements of the Arm Telemetry framework can be used across individual Arm cores or Arm processor families.

The elements also provide the flexibility for the characteristics of an Arm processor family or an individual Arm core, for example, a core's events implementation. Thus, the metric formulae can vary as required. An Arm processor family refers to a family of Arm processor IP that implement features of the A-profile architecture. Each family supports different markets and their requirements, for example, Cortex®-A, Cortex-X, or Neoverse.

For example, Topdown analysis defines a collection of metric groups that form a decision tree for a hotspot analysis use case. The nodes can be designed as a breadth first search tree, such that the traversal direction is decided by the metric weights that are defined for tree nodes at a single level. Metric groups, metric formulae, and relations can be updated for each A-profile CPU implementation. The elements of the Arm Telemetry framework let Arm processor IP teams easily define their microarchitectural events and metric formulae to derive the Topdown analysis relationships. However, the interfaces remain standardized for tooling and collateral generation.

Arm Telemetry framework can be easily adopted by any Arm A-profile CPU implementation, thereby achieving a standard telemetry solution for the Arm ecosystem. For more information about how the solution enables the profiling tools in both Linux and Windows environments, see Arm Telemetry Solution for software profiling: tools.

5. Arm Telemetry Solution for software profiling: tools

Performance monitoring features in a system have two main usage models, profiling and software optimization and run-time monitoring. Both models are enabled by established profiling tools.

Each usage model uses counting and sampling to read the hardware telemetry data supported by the platform hardware and the software telemetry data supported by system software.

Profiling and software optimization

Profiling tools that are used in software optimization get the telemetry data from system software and hardware performance monitors, which identify bottlenecks or areas of improvement. The main goal of optimization is primarily to reduce the elapsed time required to execute the program. While reducing power consumption can be a goal, it is usually a side effect of reducing the elapsed time.

Run-time monitoring

Monitoring tools that are used in system operations get the telemetry data from system software and hardware performance monitors across the system. The goal of performance monitoring is to provide a system operator with answers that can improve the efficiency of the system. Run-time monitoring helps to:

- Understand the current performance of the system.
- Evaluate whether the system executes efficiently or requires any further tuning or added capacity to meet the load demands to meet the Service Level Agreement (SLA). An SLA in a system meets sustained throughput or latency requirements of the deployed application.

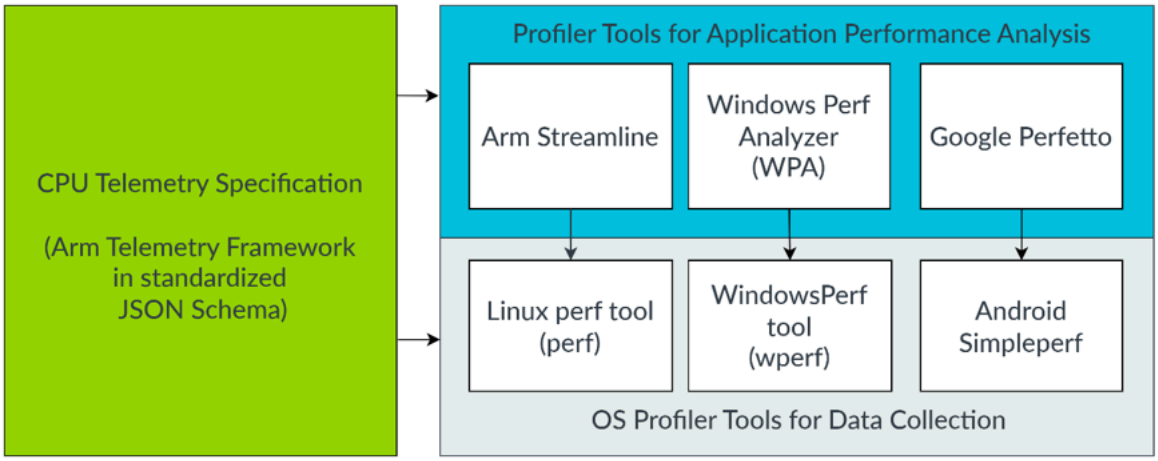
In a data center environment, observability and continuous run-time monitoring can cause the operator to do effective capacity planning and, for example, adjust capacity or rearrange load on the available machines. The job can be automated by software to monitor the system and take the required actions to guarantee the application SLA. Telemetry data from the SoC hardware is heavily used in this run-time monitoring, along with system software and application software provided metrics.

5.1 Arm telemetry specification and profiling tools

The Arm cores that support Arm CPU Telemetry Solution provide a telemetry specification that contains the methodology, metric groups, metrics, and hardware performance monitoring events which are supported by the core.

This specification is also provided in a standardized machine-readable format, that is, an Arm Telemetry JSON file that follows the schemas for the elements of the Arm Telemetry framework. See Arm Telemetry framework for CPUs. These files can be consumed by any profiling or monitoring tool as shown in in the following figure.

Figure 5-1: Arm Telemetry Solution and profiling tool enablement



Common profiling tools are Google Perfetto which is heavily used in the Android ecosystem and Microsoft Windows Performance Analyzer (WPA) which is heavily used in the Windows ecosystem. These tools either rely on the performance data collection capabilities provided by the underlying operating system or might support the tools' drivers for data collection. They can directly use the Arm Telemetry JSON files for telemetry data collection from the hardware.

Operating systems support performance data collection using utilities, for example, the Linux perf tool on Linux OS, WindowsPerf tool on Windows OS, and simpleperf tool on Android. These utilities also have support to collect performance metrics from the hardware monitoring units. They can use the Arm Telemetry specification to get the hardware information for event collection which can be post processed to derive metrics.

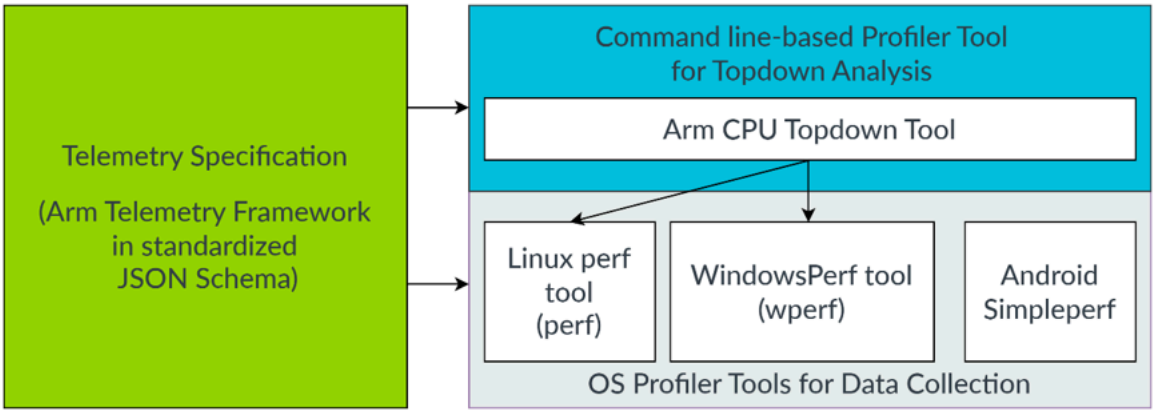
Arm recommends collecting all metrics that are in stage 1 and stage 2 Topdown analysis for workload characterization. For further analysis, an Arm core telemetry specification provides a recommended set of microarchitecture exploration metric groups against some hotspots detected in stage 1. All stage 2 metrics can be used to derive further insights into the overall microarchitecture behavior during the execution of the application under investigation. These metrics can be used independently of stage 1.

5.1.1 Arm Topdown tool

Arm CPU Telemetry Solution is implemented by the Arm Topdown tool. This tool supports collecting the recommended set of the telemetry data and processing it to derive the relevant set of metrics for application Topdown analysis.

It is a command line tool that supports using Linux perf and WindowsPerf to profile applications on Linux and Windows platforms respectively. The tool parses the machine-readable telemetry specifications provided as JSON files. It collects the data that is required to help the user with the Arm Topdown methodology stages, as shown in the following figure.

Figure 5-2: Arm Topdown tool



The tool is available to download from the Arm CPU Telemetry Solution: *Arm® Telemetry Solution GitLab repository*

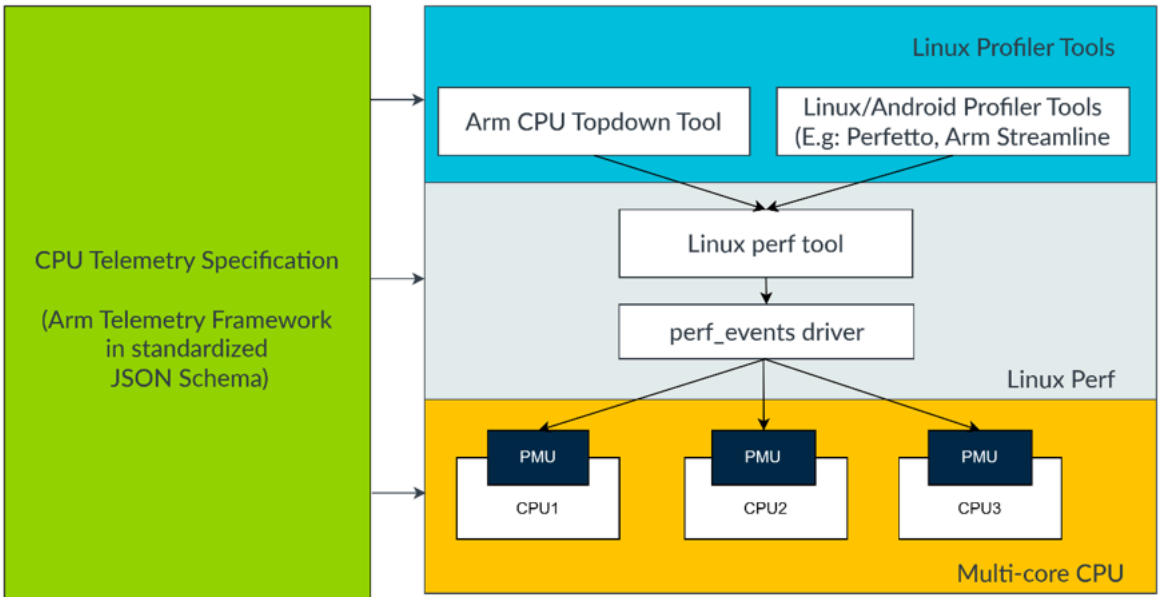
For introductory guidance about how to use the tool, see Arm Topdown tool example.

5.1.2 Performance analysis using Linux perf tool

The Linux perf tool is a widely used open source preformance analysis tool for collecting hardware and software performance events from different sources in the hardware and system software.

The kernel employs a `perf_event` subsystem to collect measurable events including the hardware PMU events from the CPU. As shown in the following figure, each CPU has its dedicated PMU hardware and the kernel perf driver collects events from each CPU's PMU separately.

Figure 5-3: Performance monitoring architecture for CPUs in a Linux based platform



The Linux `perf_event` subsystem provides an interface between the Linux kernel and user space performance monitoring tools to collect the raw hardware events as needed. As an open source tool, Linux perf can be used for performance monitoring, which supports the following types of performance measurement techniques:

Counting

Counting method collects overall statistics of an event during a workload's execution, where the counter assigned with each event produces an aggregate of the overall event count. These event statistics help to characterize the overall workload execution behavior, without providing any details on where a particular event occurred in the program. This method is the best approach for an initial workload characterization exercise to identify performance limitations of the workload.

Event sampling

Event sampling is a profiling method where each event is sampled, by configuring the PMU counter to overflow after a preset number of events. This overflow interrupt records the event count and also the program counter address and register information. Such sampled data is used to construct profiling information about the application, including stack trace and function level annotations. With this data, it is easy to locate the libraries and code portions that contribute to the large portion of the sampled event.

These measurement techniques or modes are available through the following perf commands:

`perf-stat`

Provide performance counter statistics for overall execution of the program. For more information, see *Linux perf-stat*.

`perf-record`

Record the execution performance with the percentage of samples for each event for all libraries and functions. For more information, see *Linux perf-record*.

`perf-report`

Generate a report of the recorded sample using record. For more information, see *Linux perf-report*.

`perf-annotate`

Annotate a report with the samples' percentage on the disassembly of the code. For more information, see *Linux perf-annotate*.

When high accuracy is needed, for example, when profiling hot loops or significant portions of code, the Counting mode should be preferred for its accuracy. It might require multiple profiling iterations when many different events must be logged.

When the number of events is greater than the total number of available counters, the counter is time multiplexed between events, and the final count is scaled for the total time period. This multiplexed counting can cause accuracy issues, but it is sufficient unless a precise measurement is needed.

The event sampling mode is extremely useful for hotspot analysis which relies on a statistical approach to sample different events over a large portion of time or code. This method has limitations that cause accuracy issues, such as:

- Sampling delay, that is, between the counter overflow and interrupt handler, which causes skid in the data obtained, that is, data stored during the sampling process and may not be the exact point where the event occurred.
- Speculative execution style of the CPU, whereby some instructions that executed and triggered events might not be valid if they were on the wrong code path.

While the event sampling mode has accuracy limitations, it is the best way to advance identification of hotspots in code execution. Linux perf allows tuning the sampling frequency, which helps to study variations in the event counts if the data shows large inconsistencies across runs.

For introductory guidance about how to use Linux perf to collect hardware counters on Arm cores in both counting and sampling modes, see Linux perf data collection.

5.1.3 Performance analysis using WindowsPerf Tool

WindowsPerf is a Linux perf inspired, lightweight Windows on Arm (WoA) performance profiling tool. Profiling using WindowsPerf is based on the A64 Performance Monitor Unit and its hardware counters.

WindowsPerf is a Linaro open-source project with a permissive BSD license which aims to build Arm PMU based WOA performance monitoring tool. The tool has similar features as Linux perf for counting and sampling PMU events from hardware IP blocks on the WOA platform. It also supports multiplexing and grouping similar to the Linux perf tool, which become part of Windows Kernel API in the future.

WindowsPerf is in the preliminary stages of development, though it already supports the counting model for obtaining aggregate counts of PMU events. It also uses a sampling model for determining

the frequencies of event occurrences produced by program locations at the function, basic block, and/or instruction levels. These are very similar to the approach described in Performance analysis using Linux perf tool.

WindowsPerf documentation and the tool can be accessed from: *WindowsPerf GitLab repository*

The WindowsPerf architecture comprises:

`wperf-driver`

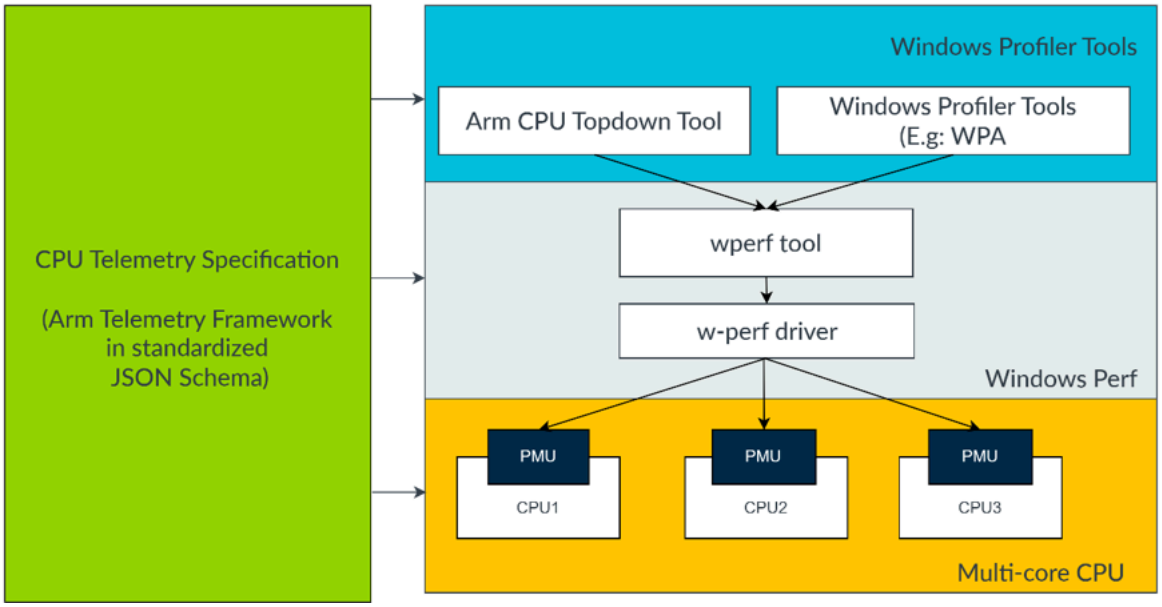
A signed Windows kernel mode driver.

WindowsPerf Kernel driver is signed only for Windows 11.

`wperf`

A command line interface like Linux perf as shown in the following figure.

Figure 5-4: Performance monitoring architecture for CPUs in a Windows based platform



WindowsPerf supports the counting and sampling modes through the following commands:

`wperf stat`

Provide performance counter statistics for overall execution of a WoA application. It supports multiplexing capabilities for events and can profile between 1 and n cores as specified. In counting mode, timeline mode is also supported, which refers to consecutive counting of events and a CSV file format output.

`wperf sample`

Provide PMU event sampling for core events, which records the execution performance using the percentage of samples for each event per libraries and functions in a WoA application. In sampling mode, users can sample WOA applications pinned to a specified core. WindowsPerf offers sampling output with hot function names annotated with:

- source code file name + line number
- optional disassembly. For an example of the output, see https://gitlab.com/Linaro/WindowsPerf/windowsperf/-/blob/main/wperf/README.md?ref_type=heads#counting-to-asses-which-events-are-popular

WindowsPerf offers human readable console output, standardized JSON output to file, console, or CSV output for timeline mode.

For more information about how to use WindowsPerf to collect hardware counters in both counting and sampling modes, see WindowsPerf tool data collection.

Appendix A Arm Topdown tool example

Use the following steps to conduct application hotspot analysis using the Arm Topdown tool.

Arm Topdown tool, `topdown-tool` provides rich options to fine tune the profiling process. Run `topdown-tool \--help` to list all options.

Step 1, Collect the stage 1 Topdown analysis metrics

In this step, we specify the Topdown methodology stage to profile, for example, `topdown` for stage 1, and `uarch` for stage 2.

```
topdown-tool -s topdown -- ./a.out
```

Results for Topdown level 1:

```
=========================
 [Topdown Level 1]
    Frontend Bound... 4.25% slots
    Backend Bound.... 91.34% slots
    Retiring......... 4.08% slots
    Bad Speculation.. 0.34% slots
```

Based on these results, this workload shows a very high backend bound metric.

Step 2, Collect the stage 2 microarchitecture exploration metrics

In this step, we can specify the metric groups of interest for further analysis. In the following example, the cache effectiveness is checked for a high backend bound workload.

```
topdown-tool -m L1D_Cache_Effectiveness, L2_Cache_Effectiveness -- ./a.out
```

Results for Stage 2:

```
Stage 2 (uarch metrics)
=======================
[L1 Data Cache Effectiveness]
  L1D Cache MPKI................... 331.529 misses per 1,000 instructions
  L1D Cache Miss Ratio............. 0.996 per cache access
[L2 Unified Cache Effectiveness]
  L2 Cache MPKI.................... 0.028 misses per 1,000 instructions
  L2 Cache Miss Ratio.............. 0.000 per cache access
```

Based on these results, there are very high L1D cache misses.

There are many metric groups to explore. For example, we can check the proportions of different kinds of instructions, for example, the Operation Mix metric.

```
./topdown-tool -m Operation_Mix -- ./a.out
```

Results for Speculative Operation Mix:

```
Stage 2 (uarch metrics)
=======================
[Speculative Operation Mix]
  Load Operations Percentage.......... 33.17% operations
  Store Operations Percentage......... 0.06% operations
  Integer Operations Percentage....... 33.48% operations
  Advanced SIMD Operations Percentage. 0.00% operations
  Floating Point Operations Percentage 0.00% operations
  Branch Operations Percentage........ 31.09% operations
  Crypto Operations Percentage........ 0.00% operations
```

Appendix B Linux perf data collection

To enable PMU event collection, the Linux Kernel must be built with `CONFIG_HW_ PERF_EVENTS` enabled in the kernel config.

Most of the production builds have this config option enabled. However, ensure that this option is enabled when building a custom kernel.

For more information about perf_event and unprivileged users, see *Linux perf_event and tool security, Unprivileged users*.

There are also two system settings which need to be configured as root user to obtain kernel symbols and add extra privileges as follows:

```
echo -1 > /proc/sys/kernel/perf_event_paranoid
echo 0 > /proc/sys/kernel/kptr_restrict
```

`perf_event_paranoid`

This setting affects privilege checks in the kernel. If set to `-1`, it permits enabling of events that might reveal sensitive information or can impact the stability of the system. For more information about this setting, see *Linux perf_event_paranoid*.

`kptr_restrict`

This setting affects whether kernel addresses are exposed, that is, through `/proc/kallsyms`. Some developers use this technique to get kernel symbol resolution when they do not have the `vmlinux` to hand, or where `KASLR` is in use. For more information about this setting, see *Linux kptr-restrict*.

In both cases, there are potential security implications, so it is advised to check the official kernel documentation and consult the system administrator before enabling them.

A quick test to verify that PMU events are being counted properly is to use the `perf stat` functionality of the Linux perf tool to count instructions and cycles. `perf stat` counts the total count of a specified event, provided as the hex register code. `0x8` is the hex code for instructions that are retired and `0x11` is the hex code for CPU cycles specified by the Arm architecture common across all Arm CPU implementations.

These events are provided to the `perf stat -e` option with a prefix `'r'` to it. For a code example, see Collect hardware PMU events using counting mode on Linux perf.

The `perf stat` command counts the total count of instructions and cpu cycles on all CPUs for 10 seconds. Linux perf allows to count for a particular CPU, for each process, for each thread and so on.

`perf` can silently fail if an event is not supported or enabled on an Arm core. For more information about supported events, see the specific Arm core telemetry specification or its Technical Reference Manual.

Collect hardware PMU events using counting mode on Linux perf

To count all events for characterization, a typical solution is to capture events in batches of the total counter registers available in the platform.

One method to determine the number of counters available is to successively increase the number of counters that are requested in a single group until the last counter reads `<"not supported">`. If this method is performed through the Linux perf tool, it opens N+1 counters, because perf always opens the group leader event in disabled mode, and the kernel does not count this event towards the number of available events.

This example uses the `-e` option to pass the instructions and cycles events to `perf stat` by using their hex codes, that is, `'r8'` and `'r11'` respectively:

```
ubuntu@linux-1:-$ perf stat -e r8,r11 -- sleep 10
Performance counter stats for 'sleep 10':
    1242692    r8
    1000747    r11
.001721954 seconds time elapsed
0.000772000 seconds user
0.000000000 seconds sys
```

Collect hardware PMU events using sampling mode on Linux perf

For sampling events, use the `perf record` command from Linux perf tool.

For more information about how to conduct sampling and analyze the sampled data with these command lines, see the following Linux perf examples.

```
ubuntu@linux-1:-$ perf record -e instructions, cycles -- sleep
ubuntu@linux-1:-$ perf report
ubuntu@linux-1:-$ perf annotate
```

Appendix C WindowsPerf tool data collection

Use the following examples to collect PMU events using either counting or sample modes with the WindowsPerf tool.

Collect hardware PMU events using counting mode on WindowsPerf

The following example collects the L1D cache metric on a running `ustress` micro-benchmark workload:

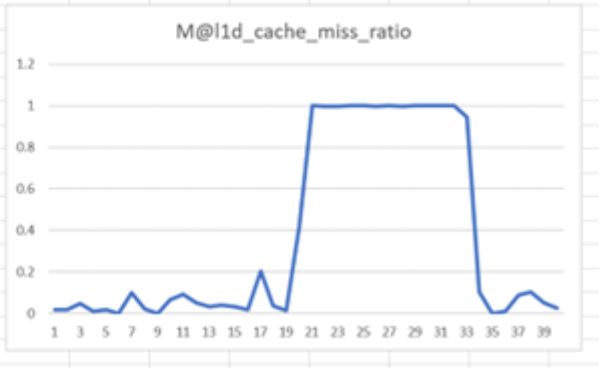
- Pin `ustress` micro-benchmark workload `l1d_cache_workload.exe` to core #1.
- Run WindowsPerf in counting mode with timeline mode `-t` to measure the `l1d_cache_miss_ratio` metric ustress `l1d_cache_workload.exe` workload built for a native WoA environment.

```
start /affinity 2 l1d_cache_workload.exe 99999
wperf stat -m l1d_cache_miss_ratio -t -i 0 \--timeout 0.5 -c 1
                  |
counting ... done
...
sleeping ... done
counting ... /Ctrl-C received, quit counting..done
sleeping ... done                                                |
```

This command creates a CSV text file that contains the `l1d_cache_miss_ratio` metric measurements by collecting the relevant hardware PMU events. To derive this metric, the tool collected the `L1D_CACHE` and `L1D_CACHE_REFILL` events and processed them. The data is collected in a CSV which can be used to plot chart for the metric as shown in the following figure.

Figure C-1: "l1d_cache_miss_ratio" metric as a chart

7	core 1	core 1	core 1	core 1
8	cycle	l1d_cache	l1d_cache_refill	M@l1d_cache_miss_ratio
9	994862	408943	7155	0.017
10	1317675	557162	9684	0.017
11	4618130	671553	31333	0.047
12	526023	210083	2268	0.011
13	7782630	2823894	50098	0.018
14	0	0	0	0
15	331350	64138	6314	0.098
16	1009404	411119	8394	0.02
17	0	0	0	0
18	175112	59919	3853	0.064
19	329680	52884	4940	0.093
20	462235	100923	5024	0.05
21	1661235	388539	12575	0.032
22	1372011	265643	10317	0.039



Collect hardware PMU events using sampling mode on WindowsPerf

WindowsPerf supports the `record` command to sample a process on a specific core.

The `record` command spawns the process and pins it to the core specified by the `-c` option. `-pe <file>` can be used as a command line option to spawn a specific process or provide the application to execute for profiling using the "`--`" double-dash option.

For example, the following command uses `wperf record` to profile an executable `python_d.exe`.

```
wperf record -e ld_spec:100000 -c 1 --timeout 30 -- python_d.exe -c 10**10**100
```

This example:

- `-c 1` option samples on core #1.
- `-e ld_spec` samples the PMU event, `LD_SPEC`.
- `ldspec:100000` sets a sampling frequency, `100000`.
- `--timeout 30` sets the time internal to sample for `30` seconds.
- `python_d.exe -c 10\*\*10\*\*100)` is a Cpython process to compute expression `"10\^10\^100"` (googolplex). In this example we sample CPython, built with the debug information.

To configure the sampling mode a user must provide additional debug information files, with the extension "`.pdb`", called the "Program DataBase", or simply "the PDB".


Note

You can specify the process PDB file name by using ``--pdb_file python_d.pdb` and `--`. In the previous example, `wperf` can deduce the PDB file name, with its path from PE file name, that is `python_d.exe -> python_d.pdb`.

WindowsPerf can correctly resolve symbols in executables, for example, `python_d.exe and DLLs` if the corresponding PDB files are provided.

An example of the expected output is as follows:

```
base address of 'python_d.exe': 0x7ff6e0a41270, runtime delta: 0x7ff5a0a40000
sampling ....ee.e.eCtrl-C received, quit counting... done!
============== sample source: ld_spec, top 50 hot functions =================
 35.42%       136  _PyEval_EvalFrameDefault:python312_d.dll
  9.38%        36  unicodekeys_lookup_unicode:python312_d.dll
  5.47%        21  _PyFrame_Stackbase:python312_d.dll
  3.91%        15  GETITEM:python312_d.dll
  3.65%        14  dictkeys_get_index:python312_d.dll
  3.39%        13  _Py_DECREF_SPECIALIZED:python312_d.dll
  3.12%        12  _PyFrame_ClearExceptCode:python312_d.dll
  2.86%        11  _PyFrame_Initialize:python312_d.dll
  2.60%        10  DK_UNICODE_ENTRIES:python312_d.dll
  2.60%        10  _Py_dict_lookup:python312_d.dll
  2.60%        10  unicode_get_hash:python312_d.dll
  2.34%         9  clear_thread_frame:python312_d.dll
  2.08%         8  _PyFrame_StackPush:python312_d.dll
  2.08%         8  PyDict_Contains:python312_d.dll
  1.82%         7  Py_INCREF:python312_d.dll
  1.82%         7  _PyThreadState_PopFrame:python312_d.dll
  1.82%         7  _PyErr_Occurred:python312_d.dll
  1.82%         7  medium_value:python312_d.dll
  1.56%         6  get_small_int:python312_d.dll
```

```
  1.30%          5  PyTuple_GET_SIZE:python312_d.dll
  1.30%          5  _PyLong_FromSTwoDigits:python312_d.dll
  1.04%          4  Py_XDECREF:python312_d.dll
  1.04%          4  _Py_atomic_load_64bit_impl:python312_d.dll
  0.78%          3  Py_IS_TYPE:python312_d.dll
  0.78%          3  _Py_EnterRecursivePy:python312_d.dll
  0.52%          2  _PyFrame_GetStackPointer:python312_d.dll
  0.52%          2  read_u16:python312_d.dll
  0.52%          2  _PyLong_Add:python312_d.dll
  0.52%          2  _PyFrame_PushUnchecked:python312_d.dll
  0.52%          2  Py_SIZE:python312_d.dll
  0.26%          1  _Py_IncRefTotal:python312_d.dll
  0.26%          1  _PyFrame_SetStackPointer:python312_d.dll
  0.26%          1  unknown
```