



Streamline CLI Tools User Guide

Version 9.2.2

Non-Confidential

Copyright © 2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

109847_9.2.2_01_en



Streamline CLI Tools User Guide

This document is Non-Confidential.

Copyright © 2024 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (109847_9.2.2_01_en) was issued on 2024-07-25. There might be a later issue at <http://developer.arm.com/documentation/109847>

The product version is 9.2.2.

See also: [Proprietary notice](#) | [Product and document information](#) | [Useful resources](#)

Start reading

If you prefer, you can skip to [the start of the content](#).

Intended audience

This document is written for application developers who want to profile and optimize software for Arm Neoverse-based systems. It assumes that you are familiar with general computer architecture concepts, but does not assume any detailed knowledge of Arm Neoverse CPUs.

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.

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>.

Contents

1. Overview of Streamline CLI Tools.....	5
2. Performance analysis concepts.....	6
2.1 Abstract CPU model.....	6
2.2 Profiling goals.....	7
2.3 Performance counters.....	9
2.4 Arm Statistical Profiling Extension.....	11
3. Using the Streamline CLI tools.....	12
3.1 Checking system compatibility.....	12
3.2 Installing the tools.....	13
3.3 Apply the kernel patch.....	13
3.4 Building your application.....	15
3.5 Capturing a profile.....	16
3.6 Analyzing a profile.....	17
3.7 Formatting a function profile.....	18
3.8 Using a formatted function profile.....	18
3.9 Data caveats.....	19
4. Custom sl-format.py reports.....	21
4.1 Configuration syntax.....	21
4.2 Column highlight styles.....	22
5. Troubleshooting.....	24
5.1 Capture data size is very large.....	24
5.2 Capture reports file descriptor exhaustion.....	24
5.3 Capture impacts application performance.....	24
6. Additional Resources.....	25
Proprietary notice.....	26
Product and document information.....	28
Product status.....	28

Revision history.....28

Conventions.....29

Useful resources.....31

1. Overview of Streamline CLI Tools

Arm Neoverse™ CPUs provide cloud and server workloads with an energy efficient computing platform. These systems give high application performance and an excellent operational price-performance ratio. To maximize performance, you can tune your software for the underlying hardware. To do this effectively, you need high quality performance data from the hardware, and performance analysis tooling to capture and interpret it.

The Streamline CLI tools are native command-line tools that are designed to run directly on an Arm server running Linux. The tools provide a software profiling methodology that gives you clear and actionable performance data. You can use this data to guide the optimization of the heavily used functions in your software. Profiling with these tools is a three-step process:

1. Capture the raw sampled data for the profile.
2. Analyze the raw sampled data to create a set of pre-processed function-attributed performance counters and metrics.
3. Format the pre-processed metrics into a pretty-printed human-readable form.

Refer to the following sections in this guide.

- Section 2 [Performance analysis concepts](#) introduces the fundamental concepts that you need to successfully use the top-down methodology that the tools provide.
- Section 3 [Using the Streamline CLI tools](#) explains how you can use the tools to capture, analyze, and format profiling data.
- Section 4 [Custom sl-format.py reports](#) explains how you create custom format definitions that are used for custom pretty-printed data visualization.
- Section 5 [Troubleshooting](#) provides troubleshooting advice for commonly encountered deployment issues.
- Section 6 [Additional Resources](#) provides links to further reading for Neoverse performance analysis, performance counters, and software optimization guides.

Audience

This document is written for application developers who want to profile and optimize software for Arm Neoverse-based systems. It assumes that you are familiar with general computer architecture concepts, but does not assume any detailed knowledge of Arm Neoverse CPUs.

Detailed profiling and counter guides exist for low-level developers, such as compiler engineers, who are interested in tuning code generation for a specific product microarchitecture.

2. Performance analysis concepts

This section explains the essential concepts that you need to understand to optimize software for a Neoverse CPU, and some useful background on the analysis approach used by the Streamline CLI Tools.

Performance analysis

A simple formula for understanding the performance of a software application is:

`Delivered performance = Utilization × Efficiency × Effectiveness`

Utilization measures the proportion of the total processor execution capacity that is spent processing instructions. This is a measure of the hardware performance.

Efficiency measures the proportion of the used processor execution capacity that is spent processing useful instructions, and not instructions that are speculatively executed and then discarded. This is a measure of the hardware performance.

Effectiveness measures the implementation efficiency of the software algorithm, compared to a hypothetical optimal implementation. This is a measure of the software performance.

To get the best performance you must implement an effective software algorithm, and then achieve high processor utilization and execution efficiency when running it.

2.1 Abstract CPU model

The processing core of a modern Arm CPU is represented in this methodology as an abstract model consisting of 3 major phases.

Figure 2-1: An abstract CPU block diagram



We define the available performance of the core using the maximum number of micro-operations (micro-ops) that can be issued to the backend each clock cycle. This execution width is known as the issue slot count of the processor.

This simple model does not include a lot of detail. For the purposes of optimizing software, most of the low-level microarchitecture is not that important because software has little control over code execution at that level.

Frontend

The frontend phase represents instruction fetch, decode, and dispatch. This phase handles fetching instructions from the instruction cache, decoding those instructions, and adding the resulting micro-ops to the backend execution queues.

Each CPU frontend microarchitecture exposes a fixed number of decode slots that can decode instructions into micro-ops each cycle. The main goal of the frontend is to keep these decode slots busy decoding instructions, unless there is back-pressure from the backend queues because the backend is unable to accept new micro-ops.

The frontend also implements support for branch prediction and speculative execution. Predicting where program control flow goes next allows the frontend to keep the backend queues filled with work when execution is uncertain. However, incorrect predictions cause the cancellation of issued micro-ops on the wrong path, and the pipeline might take time to refill with new micro-ops.

Backend

The backend phase represents the execution of micro-ops by the processing pipelines inside the core. There are multiple pipeline types, each of which can process a subset of the instruction set, and be fed by their own issue queues.

An application will have uneven loading on the backend queues. Queue load depends on the instruction mix in the part of the code that is currently running. When optimizing your application, try to prioritize changes that will relieve pressure on the most heavily loaded queue.

Retire

The retire phase represents the resolution of micro-ops that are architecturally complete. Measuring the number of retired instructions gives a metric showing the amount of useful work completed by the processor.

Not all issued instructions will retire. Speculatively issued micro-ops will be cancelled if they are shown to be on the wrong code path and are therefore not required.

2.2 Profiling goals

Using the abstract CPU model defined in the previous section, we can define some optimization goals, and associate these with behaviors in the three pipeline stages.

The main goal of our performance analysis methodology is to attribute unused or unneeded slot issues to specific causes, to give you feedback about what is causing your software to run slowly.

Retiring performance

The *Retiring* metric defines the percentage of the slot capacity that is doing useful work. This metric provides the hardware-centric “Utilization × Efficiency” measure that we proposed earlier, representing two of the three aspects of software performance. Your goal when optimizing for the hardware is to make this number as high as possible, which indicates the best usage of the available processing resources.

A high retiring metric means only that you are using the available hardware efficiently. There might still be optimization opportunities in the software that improve the effectiveness of your algorithms, and you can use other hardware metrics to guide this work.

Optimizations to consider for software that has a high retiring rate include:

- Reducing redundant processing in the algorithm.
- Reducing redundant data movement in the algorithm.
- Vectorizing heavily used functions that have not been vectorized.

Frontend performance

The role of the frontend is to issue micro-ops fast enough to keep the backend queues filled with work. Software is described as frontend bound when the frontend cannot issue a micro-op when there is free space in the backend queue to accept one. The *Frontend bound* metric defines the percentage of slot capacity lost to frontend stalls.

Consider the following optimizations for software that is frontend bound:

- Reducing code size.
- Improving the memory locality of instruction accesses.

Bad speculation

In addition to instruction decode stalls, some percentage of the available issue capacity is wasted on cycles that are used either recovering from mispredicted branches, or executing speculative micro-ops that were subsequently cancelled. The *Bad speculation* metric defines the percentage of slot capacity lost to these effects.

Consider the following optimizations for software that is suffering from bad speculation:

- Improving predictability of branches.
- Converting unpredictable branches into conditional select instructions.

Backend performance

Backend pipelines can stall, making the issue queue unable to accept a new micro-op. This occurs due to the presence of a slow multi-cycle operation, or a stalling effect such as a cache miss. Software is described as backend bound when the backend queue cannot accept a micro-op when the frontend has one ready to issue. The *Backend bound* metric defines the percentage of slot capacity lost to these effects.

Consider the following optimizations for software that is backend bound:

- Reducing the size of application data structures and data types.
- Improving the memory locality of data accesses.
- Reducing use of slow multi-cycle instructions.
- Swapping instructions to move work away from issue queues that are under the most load.

2.3 Performance counters

Arm CPUs include a Performance Monitoring Unit (PMU) that measures instances of low-level execution events occurring in the hardware. These measurements are useful for multiple purposes:

- Counting instructions or cycles is useful for sizing a workload.
- Counting SIMD vector instructions is useful for identifying whether a workload is taking advantage of the available hardware acceleration.
- Counting branch mispredictions or cache misses is useful for identifying whether a workload is triggering specific performance pathologies.

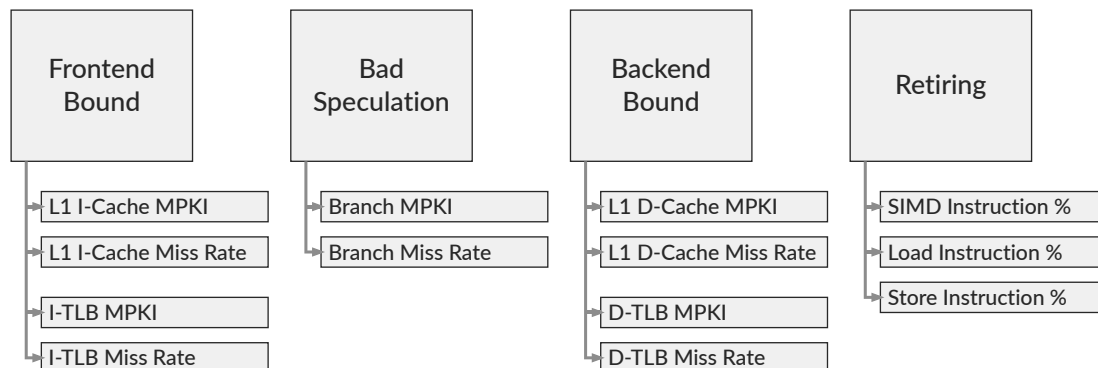
To make performance analysis easier, Arm has defined a standardized performance analysis methodology for the Neoverse CPUs. This methodology defines a common set of hardware performance counters, and how to use them to derive the higher-level metrics that enable you to optimize your applications.

The top-down methodology

The top-down methodology provides a systematic way to use performance counter data to identify performance problems in an application.

The methodology describes performance using a simple hierarchical tree of performance metrics. The basic metrics described for the abstract model provide the root nodes of the tree. Additional levels of hierarchy below each node provide a more detailed breakdown for causal analysis.

Figure 2-2: The major top-down metrics



This hierarchical approach, with clear causal metrics, provides an intuitive way to find and understand the microarchitecture-sensitive performance issues that your software is triggering. Using this information, you can target the problem with specific corrective actions to improve the performance.

One of the major usability benefits of the top-down methodology for software developers is that the first few levels of the top-down tree do not require any knowledge of the specific CPU you are running on. You can profile on any of the Neoverse CPUs and get the same metrics, despite

differences in the underlying hardware design. This lets you focus on your software and improving its performance, instead of worrying about which event to capture on a specific CPU.

The deeper levels of the tree become increasingly hardware specific, which is useful for developers who want to optimize very deeply for a specific microarchitecture. For most common software optimizations these levels are not necessary.

Stall metrics

The most common causes of stalls are cache misses and branch mispredictions. To make it easier to understand the impact of stalls, two forms of miss metrics are given:

- *Miss rate* metrics tell you the percentage of misses for that specific operation type. These metrics tell you how effectively a particular cache or prediction unit is performing.
- *Misses per thousand instruction* (MPKI) metrics tell you how many misses of that type occurred, on average, when running 1000 instructions of any type. These metrics tell you how significant the impact of a particular type of miss is, given the instruction makeup of the program.

For example, you measure a *Branch mispredict rate* of 45% when profiling, which tells you that 45% of branches are mispredicted. This is a clear sign that the branch predictor is struggling, so improving branches can be an optimization candidate. However, when you check the *Branch MPKI* metric you see that you only have 0.8 mispredictions for every 1000 instructions in the sample. Even though branches are not predicting well, optimizing will not bring significant improvements because branches are only a small proportion of the instruction mix.

Function attribution

The top-down metrics provide a systematic approach to identifying performance problems in your software, but this is only actionable feedback if the metrics are associated with a specific location in the running program.

The Streamline CLI Tools implement function-attributed metrics by measuring the performance counters over a small sample window of just a few hundred cycles. This allows the tool to see the useful function-frequency signals in the performance counter data that are lost with traditional 1ms periodic sampling.

To reduce the volume of data produced, our approach uses a strobing sampling pattern with an uneven mark-space ratio. For example, we capture data for a 200 cycle window, but only do so once every 2 million cycles. This approach gives us the high frequency data visibility that we need for function-attribution, while keeping a low probe-effect on the running application and a manageable profile data size.



Support for strobing counter sample windows is a new capability for the Linux Perf kernel driver, which is not yet available upstream. A kernel patch is provided to implement this functionality.

2.4 Arm Statistical Profiling Extension

Arm CPUs can support the Statistical Profiling Extension (SPE), which adds support for hardware-based instruction sampling.

When using SPE, the hardware triggers a sample after a configurable number of micro-ops. It writes the sample data directly into a memory buffer without any software involvement. This sampling is not invasive to the running program, until software is needed to process a full memory buffer.

Each sample contains the program counter (PC) of the sampled operation, and additional operation-specific event data. This event data provides additional feedback about the execution of that operation. For example:

- For branch samples, the event data indicates if the branch was mispredicted.
- For load samples, the event data indicates which cache returned the data.

SPE provides a complementary technology to the traditional performance counters, and the best results can be achieved by using both together.

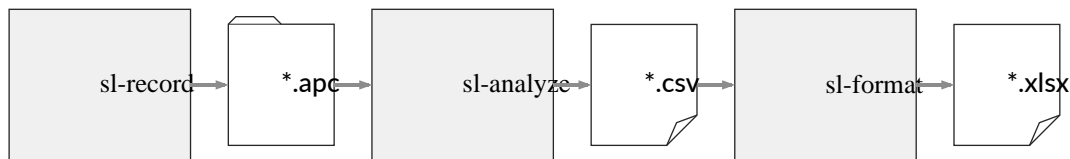
3. Using the Streamline CLI tools

The Streamline CLI tools are native command-line tools that are designed to run directly on an Arm server running Linux.

Profiling with these tools is a three-step process:

1. Use `sl-record` to capture the raw sampled data for the profile.
2. Use `sl-analyze` to pre-process the raw sampled data to create a set of function-attributed counters and metrics.
3. Use `sl-format.py` to pretty-print the function-attributed metrics in a more human-readable form.

Figure 3-1: Streamline CLI tools workflow



3.1 Checking system compatibility

Before you begin, you can use the the Arm Sysreport utility to determine whether your system configuration supports hardware-assisted profiling.

Follow the instructions in this [Learning Path tutorial](#) to discover how to download and run this utility.

The `perf counters` entry in the generated report will indicate how many CPU counters are available. The `perf sampling` entry will indicate if SPE is available.

You will achieve the best profiles in systems with at least 6 available CPU counters and SPE.

The Streamline CLI tools can be used in systems with no CPU counters, but will only be able to return a basic hot-spot profile based on time-based sampling. No top-down methodology metrics will be available.

The Streamline CLI tools can give top-down metrics in systems with as few as 3 available CPU counters. The effective sample rate for each metrics will be lower, because we will need to time-slice the counters to capture all of the requested metrics, so you will need to run your application for longer to get the same number of samples for each metric. Metrics that require more input counters than are available cannot be captured.

The Streamline CLI tools can be used without SPE. Load operation data source metrics will not be available, and branch mispredict metrics may be less accurate.

3.2 Installing the tools

The Streamline CLI tools are available as a standalone download to enable easier integration in to server workflows.

```
wget https://artifacts.tools.arm.com/arm-performance-studio/2024.3/
Arm_Streamline_CLI_Tools_9.2.2_linux_arm64.tgz

tar -xzf Arm_Streamline_CLI_Tools_9.2.2_linux_arm64.tgz
```

The `sl-format.py` Python script requires Python 3.8 or later, and depends on several third-party modules. We recommend creating a Python virtual environment containing these modules to run the tools. For example:

```
# From Bash
python3 -m venv sl-venv
source ./sl-venv/bin/activate

# From inside the virtual environment
python3 -m pip install -r ./<install>/bin/requirements.txt
```



The instructions below assume you have added the `<install>/bin/` directory to your `PATH` environment variable, and that you run all Python commands from inside the virtual environment.

3.3 Apply the kernel patch

For best results we provide a Linux kernel patch that modifies the behavior of Linux perf to improve support for capturing function-attributed top-down metrics on Arm systems. This patch provides two new capabilities:

- It allows a new thread to inherit the perf counter group configuration of its parent.
- It decouples the perf event-based sampling window size from the overall sample rate. This allows strobed mark-space sampling patterns where the tool can capture a small window without using a high sample rate.

Without the patch it is possible to capture profiles. However, not all capture options are available and capturing top-down metrics will rely on high frequency sampling. The following options are available:

- System-wide profile with top-down metrics.
- Single threaded application profile with top-down metrics.

- Multi-process application profile without top-down metrics.

With the patch applied it is possible to collect the following profiles:

- System-wide profile with top-down metrics.
- Single threaded application profile with top-down metrics.
- Multi-process application profile with top-down metrics.

The following instructions show how to install the patch on Amazon Linux 2023. You may need to adapt slightly to other Linux distributions.

Manual application to the source tree

To apply the patch to the latest 6.7 kernel, you can use `git`:

```
git apply v6.7-combined.patch
```

Or patch:

```
patch -p 1 -i v6.7-combined.patch
```

Manual application to an RPM-based distribution

Follow these steps to integrate these patches into an RPM-based distribution's kernel:

1. Remove any existing `rpmbuild` directory (rename as appropriate):

```
rm -fr rpmbuild
```

2. Fetch the kernel sources:

```
yum download --source kernel
```

3. Install the sources binary:

```
rpm -i kernel-<VERSION>.src.rpm
```

4. Enter the `rpmbuild` directory that is created:

```
cd rpmbuild
```

5. Copy the patch into the correct location. Replace the 9999 patch number with the next available patch number in the sequence:

```
cp vX.Y-combined.patch SOURCES/9999-strobing-patch.patch
```

6. Open the specs file in your preferred editor:

```
nano SPECS/kernel.spec
```

7. Search for the list of patches starting with `Patch0001` and append the line for the new patch to the end of the list. Replace 9999 with the patch number used earlier:

```
Patch9999: 9999-strobing-patch.patch
```

8. Search for the list of patch apply steps starting with `ApplyPatch` and append the line for the new patch to the end of the list. Replace 9999 with the patch number used earlier:

```
ApplyPatch 9999-strobing-patch.patch
```

9. Save the changes and exit the editor.

10. Build the kernel and other rpms:

```
rpmbuild -ba SPECS/kernel.spec
```

11. Install the built packages:

```
sudo rpm -ivh --force RPMS/aarch64/*.rpm
```

12. Reboot the system:

```
sudo reboot
```

13. Validate that the patch applied correctly:

```
ls -l /sys/bus/event_source/devices/*/format/strobe_period
```

This should list at least one CPU PMU device supporting the strobing features, for example:

```
/sys/bus/event_source/devices/armv8_pmu3_0/format/strobe_period
```

3.4 Building your application

Before you can capture a software profile you must build your application with debug information. This enables the profiler to map instruction addresses back to specific functions in your source code. For C and C++ you do this by passing the `-g` option to the compiler.

Arm recommends that you profile an optimized release build of your application, as this ensures you are profiling a realistic code workload. For C and C++ you do this by passing the `-O2` or `-O3` option to the compiler. However, we also recommend that you disable invasive optimization techniques, such as link-time optimization (LTO), because they heavily restructure the code and make the profile difficult to understand.

3.5 Capturing a profile

Use `sl-record` to capture a raw profile of your application and save the data to a directory on the filesystem.

Arm recommends making a profile of at least 20 seconds in duration, which ensures that the profiler can capture a statistically significant number of samples for all of the metrics.

```
sl-record -C workflow_topdown_basic -o <profile.apc> -A <your app command-line>
```

This command uses the following options:

- The `-c` option provides a comma-separated list of counters and metrics to capture. The workflow-prefixed options in the counter list select a predefined group of counters and metrics, making it easier to select everything you need for a standard configuration. Using `workflow_topdown_basic` is a good baseline option to start with.

To list all of the available counters and metrics for the current machine, use the command `sl-record --print counters`.

- The `-o` option provides the output directory for the capture data. The directory must not already exist because it is created by the tool when profiling starts.
- The `-A` option provides the command-line for the user application. This option must be the last option provided to `sl-record` because all subsequent arguments are passed to the user application.

Optionally, to enable SPE add the `-x workflow_spe` option. Enabling SPE significantly increases the amount of data captured and the `sl-analyze` processing time.

Captures are highly customizable, with many different options that allow you to choose how to profile your application. Use the `--help` option to see the full list of options for customizing your captures.

Capturing a system-wide profile

To capture a system-wide profile, which captures all processes and threads, run with the `-s yes` option and omit the `-A ... application-specific` option and following arguments.

In systems without the kernel patches, system-wide profiles can capture the top-down metrics. To keep the captures to a usable size, it may be necessary to limit the duration of the profiles to less than 5 minutes.

Capturing top-down metrics without the kernel patches

To capture top-down metrics in a system without the kernel patches there are three options available.

To capture a system-wide profile, which captures all processes and threads, run with the `-s yes` option and omit the `-A ... application-specific` option and following arguments. To keep the captures to a usable size, it may be necessary to limit the duration of the profiles to less than 5 minutes

To reliably capture single-threaded application profile, add the `--inherit no` option to the command line. However, in this mode metrics are only captured for the first thread in the application process and any child threads or processes are ignored.

For multi-threaded applications, the tool provides an experimental option, `--inherit poll`, which uses polling to spot new child threads and inject the instrumentation. This allows metrics to be captured for a multi-threaded application, but has some limitations:

- Short-lived threads may not be detected by the polling.
- Attaching perf to new threads without inherit support requires many new file descriptors to be created per thread. This can result the application failing to open files due to the process hitting its inode limit.

Minimizing profiling application impact

The `s1-record` application requires some portion of the available processor time to capture the data and prepare it for storage. When profiling a system with a high number of CPU cores, Arm recommends that you leave a small number of cores free so that the profiler can run in parallel without impacting the application. You can achieve this in two different ways:

- Running an application with fewer threads than the number of cores available.
- Running the application under `taskset` to limit the number of cores that the application can use. You must only `taskset` the application, not `s1-record`, for example:

```
s1-record -C ... -o ... -A taskset <core_mask> <your app command-line>
```



The number of samples made is independent of the number of counters and metrics that you enable. Enabling more counters reduces the effective sample rate per counter, and does not significantly increase the performance impact that capturing has on the running application.

3.6 Analyzing a profile

Use `s1-analyze` to process the raw profile of your application and save the analysis output as several CSV files on the filesystem.

```
s1-analyze --collect-images -o <output_dir> <profile.apc>
```

This command uses the following arguments:

- The `-collect-images` option instructs the tool to assemble all of the referenced binaries and split debug files required for analysis. The files are copied and stored inside the `.apc` directory, making them ready for analysis.
- The `-o` option provides the output directory for the generated CSV files.
- The positional argument is the raw profile directory created by `s1-record`.

Several CSV files are generated by this analysis:

- Files that start `functions-`: A flat list of functions, sorted by cost, showing per-function metrics.
- Files that start `callpaths-`: A hierarchical list of function call paths in the application, showing per-function metrics for each function per call path location.
- Files that end `-bt.csv`: Results from the analysis of the software-sampled performance counter data, which can include back-traces for each sample.
- Files that end `-spe.csv`: Results from the analysis of the hardware-sampled Statistical Profiling Extension (SPE) data. SPE data does not include call back-trace information.

3.7 Formatting a function profile

The function profile CSV files generated by `sl-analyze` contain all the enabled events and metrics, for all functions that were sampled in the profile.

Use `sl-format.py` to generate a simpler pretty-printed XLSX spreadsheet that is suitable for human consumption.

```
python3 sl-format.py -o <output.xlsx> <input.csv>
```

This command uses the following arguments:

- The `-o` option provides the output file path to save the XLSX file to.
- The positional argument is the `functions-*.csv` file created by `sl-analyze`.

This formatter has several basic capabilities:

- Selecting and ordering the desired metrics columns.
- Filtering out low-value function rows by absolute or relative significance.
- Formatting metrics columns using short names for compact presentation.
- Formatting metrics cell colors using threshold rules to spotlight bad values.
- Emitting the data as an XLSX data table, allowing interactive column sorting and row filtering when opened in OpenOffice or Microsoft Excel.

Section 4 [Custom `sl-format.py` reports](#) of this guide explains how you create and specify custom format definitions that are used to change the pretty-printed data visualization.

3.8 Using a formatted function profile

There is no right way to profile and optimize, but the top-down data presentation gives you a systematic way to find optimization opportunities.

Here is our optimization checklist:

Check the compiler did a good job.

- Disassemble your most significant functions.
- Verify that the generated code looks efficient.

Check the functions that are the most frontend bound:

- If you see high instruction cache miss rate, apply profile-guided optimization to reduce the code size of less important functions. This frees up more instruction cache space for the important hot-functions.
- If you see high instruction TLB misses, apply code layout optimization, using tools such as [Bolt](#). This improves locality of code accesses, reducing the number of TLB misses.

Check the functions that have the highest bad speculation rate:

- If you see high branch mispredict rates, use a more predictable branching pattern, or change the software to avoid branches by using conditional selects.

Check the functions that are the most backend bound:

- If you see high data cache misses, reduce data size, reduce data copies and moves, and improve access locality.
- If you see high pipeline congestion on a specific issue queue, alter your software to move load a different queue. For example, converting run-time computation to a lookup table if your program is arithmetic limited.

Check the most retiring bound functions:

- Apply SIMD vectorization to process more work per clock.
- Look for higher-level algorithmic improvements.

3.9 Data caveats

The Streamline CLI tools provide you with function-attributed performance metrics. To implement this using the Arm PMU, we take an interrupt at the start of the sample window to zero the counters, and at the end of the sample window to capture the counters.

This pair of context-switches has an overhead on the running software. The absolute value of some metrics can differ to the value that would be reported if our sample was non-invasive. However, functions will rank correctly, and trends are directionally accurate when showing the impact of an optimization.

Our methodology has three known side-effects that impact the metrics:

- At the start of the sample window, it takes some cycles to refill the pipeline when returning from the context switch. This means we retire fewer instructions in the sample window than normal steady-state execution.

- At the end of the sample window, issued instructions that are queued in the issue queue are cancelled to reduce the context switch latency. This means we see a much higher number of instructions that are speculatively issued but not retired than normal steady-state execution.
- The kernel code run at the start of the sample window places higher pressure on caches and other cache-like structures. However, for most software the impact of this is minor.

Impacted metrics

We are aware of the following impact on the default top-down metrics shown in the formatted report.

- Retiring - Reports lower than normal
- Frontend bound - Reports higher than normal
- Bad speculation - Reports higher than normal
- Backend bound - Reports lower than normal

4. Custom sl-format.py reports

The Streamline CLI analysis tool, `sl-analyze`, outputs a raw CSV file containing all of the profiling metrics that were generated. For a complex application, this is large and difficult to use for manual review.

The `sl-format.py` script provides a method to extract a filtered subset of the data to an XLSX spreadsheet, including generation of interactive tables and custom cell formatting. The presentation format is specified using a YAML configuration file, allowing easy reconfiguration of the visualization.

The script requires a PMU-based profile, and can optionally merge in results from an SPE-based profile.

Passing a custom configuration

Optionally, you can pass a custom configuration file using the `--config` argument.

```
python3 sl-format.py -o <out.xlsx> <in.csv> [--config <conf.yaml>]
```

If no configuration is specified, a default presentation suitable for a profile recorded using `-c workflow_topdown_basic` is used.

4.1 Configuration syntax

The configuration file is a YAML file containing an ordered list of metrics. Each metric is presented as a column in the output table, with each identified application function in the source application as a row.

Basic syntax

Each metric must specify the data `src_name`, which is the column title in the input CSV file. The metrics can optionally specify the `dst_name`, which is the column name to use in the XLSX output. If no `dst_name` is specified, the `src_name` is used.

```
---
- series:
  src_name: symbol
  dst_name: Function
- series:
  src_name: "Metrics: Sample Count"
  dst_name: Samples
...
```

Symbol name series

The raw function names (`src_name: symbol`) in the CSV include full parameter lists, which can help disambiguating functions in software that makes heavy use of operator overloading. In many

applications this is not necessary and simply clutters the visualization. You can set `strip_params: true` for the `symbol` source column to discard parameters.

Arm recommends that the series for the function name, as well as other string-like columns, include the `dtype: str` property. This property stops empty cells in these columns being interpreted as a floating point NaN value.

Symbol filtering

The raw data includes all symbols that were sampled during the profile. Many symbols are often of low significance, with few samples compared to the overall sample count. The formatted data can discard low significance rows to make the data easier to use.

To enable filtering for a series, you can add the following options:

- `filter: significance`, and
- `min_row_significance: <val>` with a significance value between 0 and 1.

For example, a minimum significance of 0.01 in the “samples” column indicates that any function with fewer than 1% of the total samples should be discarded.

4.2 Column highlight styles

Data columns can include basic styling rules to help highlight cells with values that are worth investigating.

style: absolute_ramp_up

```
- series:
  style: absolute_ramp_up
  min_ramp: 2
  max_ramp: 4
```

This style is based on the absolute value of each cell. It increases from no highlight for a value below `min_ramp` to a maximum intensity highlight for a value above `max_ramp`.

style: absolute_ramp_down

```
- series:
  style: absolute_ramp_down
  min_ramp: 2
  max_ramp: 4
```

This style is based on the absolute value of each cell. It increases from no highlight for a value above `max_ramp` to a maximum intensity highlight for a value below `min_ramp`.

style: relative_ramp_up

```
- series:
  style: relative_ramp_up
  min_ramp: 0.9
```

```
max_ramp: 1.0
```

This style is based on the value of a cell relative to the min/max range of the column, where a threshold of 0.0 indicates the minimum value of the column and 1.0 indicates the maximum value of the column. The example above highlights cells in the top 10% of the column range.

It increases from no highlight for a relative value below `min_ramp` to a maximum intensity highlight for a relative value above `max_ramp`.

style: relative_ramp_down

```
- series:
  style: relative_ramp_down
  min_ramp: 0.0
  max_ramp: 0.1
```

This style is based on the value of a cell relative to the min/max range of the column, where a threshold of 0.0 indicates the minimum value of the column and 1.0 indicates the maximum value of the column. The example above highlights cells in the bottom 10% of the column range.

It increases from no highlight for a relative value above `max_ramp` to a maximum intensity highlight for a relative value below `min_ramp`.

style: stdev_ramp_up

```
- series:
  style: stdev_ramp_up
  min_ramp: 0.5
  max_ramp: 1.0
```

This style is based on the value of a cell relative to the number of standard deviations from the mean. A threshold of N indicates a value that is N standard deviations from the mean. The example above starts highlighting cells that are 0.5 standard deviations higher than the mean, ramping to full intensity for cells that are 1 standard deviation higher than the mean.

It increases from no highlight for a value below `min_ramp` standard deviations, to a maximum intensity highlight for a value above `max_ramp` standard deviations.

style: stdev_ramp_down

```
- series:
  style: stdev_ramp_down
  min_ramp: -1.0
  max_ramp: -0.5
```

This style is based on the value of a cell relative to the number of standard deviations from the mean. A threshold of N indicates a value that is N standard deviations from the mean. The example above starts highlighting cells that are 0.5 standard deviations lower than the mean, ramping to full intensity for cells that are 1 standard deviation lower than the mean.

It increases from no highlight for a value above `max_ramp` standard deviations, to a maximum intensity highlight for a value below `min_ramp` standard deviations.

5. Troubleshooting

This section outlines some common problems that can be encountered when you are deploying the tools, and their solutions.

5.1 Capture data size is very large

When capturing with `-I poll`, the size of the data capture is very large.

The polling mode is provided as a fallback for systems without the kernel patches. To capture data for function-attribution without the patches, the tool must use a very high sample rate which increases the captured data size.

[Apply the kernel patch](#), and run `s1-record` without the `-I` option, which is equivalent to `-I inherit`. The kernel patch implements strobed sampling, allowing us to alternate between a fast “mark” window that is captured and a slow “space” window that is skipped.

5.2 Capture reports file descriptor exhaustion

The application runs out of file descriptors when capturing with `-I poll`.

The polling mode is provided as a fallback for systems without the kernel patches. To attach Perf counter groups to a new application thread we must open new file handles for each counter group for every core in the system.

[Apply the kernel patch](#), and run `s1-record` without the `-I` option, which is equivalent to `-I inherit`. The kernel patch allows Perf counter groups to be inherited by new child threads, avoiding the need to open new file handles.

5.3 Capture impacts application performance

Application performance is impacted by running under `s1-record` on a high core count system.

Capturing and storing the profiling data has an overhead on the running system, especially when multiplexing counters with Perf.

Limit the running application to a subset of the CPU cores, leaving a small number of cores free for `s1-record`. For example, on a 64 core system Arm recommends limiting the application to 60 cores.

6. Additional Resources

The following links provide additional information about optimizing for Arm Neoverse CPUs.

The full specifications for the Arm top-down methodology:

- [Arm Telemetry Solution Top-down Methodology Specification](#)
- [Arm Neoverse N1 Performance Analysis Methodology](#)
- [Arm Neoverse V1 Performance Analysis Methodology](#)

The performance counter guides for specific Neoverse products:

- [Arm Neoverse N1 PMU Guide](#)
- [Arm Neoverse N2 PMU Guide](#)
- [Arm Neoverse V1 PMU Guide](#)
- [Arm Neoverse V2 PMU Guide](#)

The software optimization guides for specific Neoverse products:

- [Arm Neoverse N1 Software Optimization Guide](#)
- [Arm Neoverse N2 Software Optimization Guide](#)
- [Arm Neoverse N3 Software Optimization Guide](#)
- [Arm Neoverse V1 Software Optimization Guide](#)
- [Arm Neoverse V2 Software Optimization Guide](#)
- [Arm Neoverse V3 Software Optimization Guide](#)
- [Arm Neoverse V3AE Software Optimization Guide](#)
- [Arm Neoverse E1 Software Optimization Guide](#)

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

export laws 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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0902-01	25 July 2024	Non-Confidential	Updated for version 9.2.2 plus added Troubleshooting section.
0902-00	17 June 2024	Non-Confidential	First release

Change history

The revisions tables describe the technical changes between released issues of this document.

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
<i>italic</i>	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 <u>underline</u>	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: <div>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></div>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and 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.

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.