WHITE PAPER

Arm Statistical Profiling Extension: Performance Analysis Methodology

Authors:

Jumana Mundichipparakkal*

Rui Chang*

Yibo Cai*

Darin Greene

Michael Williams

Tushar Chouhan

Contents

Arm Statistical Profiling Extension: Performance Analysis Methodology

ABSTRACT

This paper presents a methodology for workload characterization and root cause analysis using the Arm® Statistical Profiling Extension (SPE) demonstrated on a Neoverse ™ N1 core. The target audience are software developers and performance analysts in software development, analysis, optimization, and tuning. This paper may also help silicon engineers to conduct performance analysis and debugging.

1. Introduction

The Statistical Profiling Extension (SPE) is an Arm® architectural feature extension that supports enhanced instruction execution profiling in the CPU.

This paper presents a methodology for workload characterization and root cause analysis using the SPE demonstrated on a Neoverse ™ N1 core. The target audience are software developers and performance analysts in software development, analysis, optimization, and tuning. This paper may also help silicon engineers to conduct performance analysis and debugging.

The aim of this paper is to extend the top-down performance methodology using Performance Monitor Unit (PMU) events[1]. It demonstrates additional profiling techniques using the SPE feature implemented in Arm processors that can be applied to root cause analysis.

This paper contains the following:

- Performance analysis using Arm SPE introduces the Arm SPE as a performance analysis feature that can be used for root cause analysis and memory analysis on AArch64-based platforms that implement SPE.
- Using SPE for performance root cause analysis presents a case study using SPE for precise sampling for performance root cause analysis.
- Using SPE for memory access analysis demonstrates memory access analysis with SPE using the industry standard LMbench workload.
- Using SPE for data sharing analysis presents data sharing analysis using the Linux perf tool c2c feature which uses SPE for profiling.

2. Performance Analysis by Arm SPE

2.1. What is Arm SPE?

The Arm Statistical Profiling Extension (SPE) feature provides a hardware-assisted CPU operation profiling mechanism. It enables more detailed profiling capabilities than traditional sampling mechanisms using PMU events for performance analysis and tuning use cases. The SPE feature extension is specified as part of Armv8-A architecture, with support from Arm v8.2 onwards, as described in the *Arm Architecture Reference Manual for A-profile architecture*[2].

The Statistical Profiling Unit (SPU) records key data points and traces of the execution lifecycle of operations that are sampled in the backend of the CPU. The SPE sampling process is built in as part of the instruction execution path in the CPU pipeline. Each sampled operation generates a sample record. Each record consists of multiple packets containing key operation execution information including the following:

- PC value of the operation
- Data addresses accessed
- Pipeline latency
- A set of PMU events triggered
- Timestamp

The operations sampled are statistically selected based on configurable sampling intervals, with the option to add a jitter for randomness. Even though the sampled operations are statistically sampled, the sample record provides precise information on the operation execution including the PC value and timestamp, which helps to investigate code hotspots. This precise sampling mechanism mitigates the skid issue in locating hotspots, associated with traditional sampling mechanisms using PMU events. PMU event-based sampling takes interrupts which introduce latency in recording the sample, thereby causing the skid. SPE records the operation execution information precisely, including the PC and timestamp, in the CPU registers while being sampled. SPE then stores the profiles in memory after the corresponding instruction has completed execution.

SPE also helps with detailed analysis of performance-critical operations such as branches, memory accesses, and SVE, by providing extra execution details like branch mispredictions, cache misses, memory access latency, and SVE predicate utilization. These operations can have a huge impact on efficient execution for a given micro-architecture. For example, the memory access analysis feature provided by SPE is very useful for optimization and tuning of memory bandwidth and latency heavy workloads.

2.2. SPE usage models and methodology

Some of the key methodologies that can be applied for performance analysis using SPE-profiled data are as follows:

01 Precise sampling for hotspot detection in source code

02 Memory access analysis

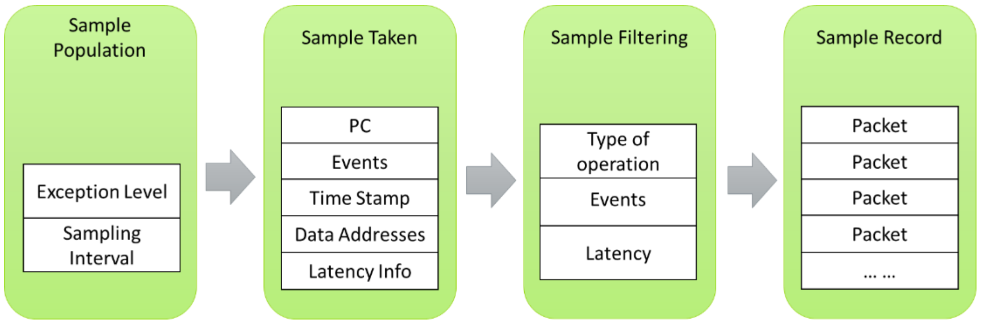
03 Data sharing analysis

2.3. SPE hardware and tracing mechanism

The SPU periodically selects an operation for profiling based on the underlying sampling mechanism. For each sampled operation, SPE tracing can be divided into four stages, starting with sampling the operation and ending when the operation is retired, and the sample record is stored in memory.

Figure 2.1 shows these four stages.

Figure 2.1 The four stages of sampling using SPE



The four stages of sampling using SPE are:

01 The SPU statistically selects operations to sample at the backend of the CPU. The sampling interval is configurable. To get more insights into the sampling quality, the CPU supports several PMU events for statistical analysis of the sampling mechanism. For more information, see Statistical sampling in SPE.

02 Record a trace of key operation execution information for the sampled operation. For more information, see SPE sample records. The SPU stores the trace data internally in the CPU during its execution lifetime until the operation retires, is flushed due to mis-speculation, or generates an exception.

03 The SPU supports post-filtering of sample records, which helps to reduce the memory required for storage. A full record can be up to 64 bytes. For more information about SPE record filtering configuration, see Filtering options in SPE.

04 After filtering, the SPE record is stored as a sequence of sample packets in memory. These records can then be processed using tools such as the Linux perf tool. For more information about monitoring tools that can be used for extracting SPE data, see SPE monitoring tools.

2.3.1. Statistical sampling in SPE

SPE collects trace data periodically, using a down counter that selects a micro-operation to profile. The counter counts the number of speculative micro-operations dispatched, decrementing by one for each micro-operation starting with a count set by the sampling interval which can be programmed by software. The CPU can also be programmed to add a random jitter to this interval each time it is reloaded. When the counter reaches zero, the next micro-operation is identified to be sampled. The micro-operation is then profiled throughout its execution lifetime. The counter is reloaded with the sampling interval.

Table 2.1 shows the PMU events that can be used for statistical analysis of the SPE sampling mechanism.

Table 2.1 PMU events for statistical analysis

Event	Description
SAMPLE_POP	Total sampling population. This is the total number of operations that have executed since the profiling started, regardless of whether or not they were selected for sampling by SPE during the profiling process.
SAMPLE_FEED	Total selected samples. This is the total number of operations that were selected for sampling by SPE.
SAMPLE_FILTRATE	Total recorded samples. This is the number of samples that were taken and recorded in memory, and not removed by post filtering.
SAMPLE_COLLISION	Total number of operations selected for sampling that collided with a previously sampled operation and were therefore not sampled. This situation occurs when a sampled operation does not finish before the next counter tick.

Table 2.2 lists the included analysis metrics.

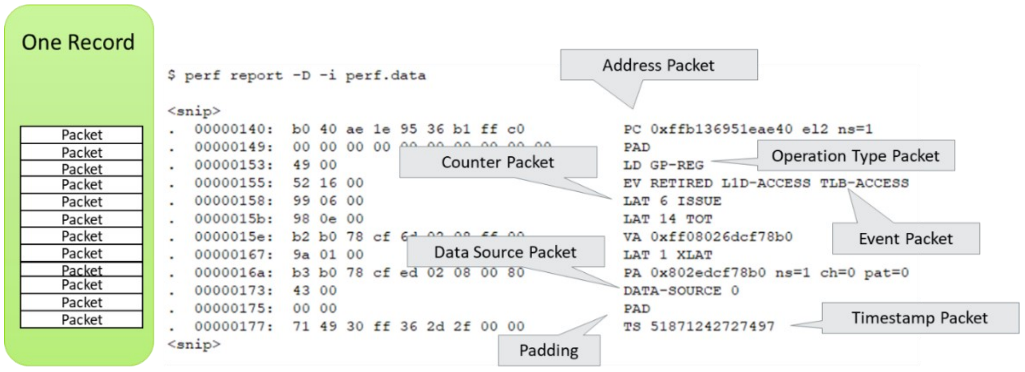
Table 2.2 Analysis metrics

Metric	Description	Formula
spe_sampling_ratio	Ratio of total samples profiled by SPE to the total operations in the sampling population	SAMPLE_FEED/ SAMPLE_POP
spe_filter_ratio	Ratio of total samples profiled by SPE after post filtering to the total operations that were sampled	SAMPLE_FILTRATE/ SAMPLE_FEED
spe_collision_ratio	Ratio of total samples that were not picked due to collision to the total operations in the sampling population	SAMPLE_COLLISION/ SAMPLE_POP

2.3.2. SPE sample records

An SPE sample record is a record of the operation that captures insights into the execution lifecycle of the operation starting at the backend where the SPU starts data collection. Figure 2.2 shows an example SPE sample record.

Figure 2.2 An example SPE sample record



```
$ perf report -D -i perf.data

<snip>
.  00000140:  b0 40 ae 1e 95 36 b1 ff c0          PC 0xffb136951eae40 el2 ns=1
.  00000149:  00 00 00 00 00 00 00 00 00 00       PAD
.  00000153:  49 00                               LD GP-REG
.  00000155:  52 16 00                            EV RETIRED L1D-ACCESS TLB-ACCESS
.  00000158:  99 06 00                            LAT 6 ISSUE
.  0000015b:  98 0e 00                            LAT 14 TOT
.  0000015e:  b2 b0 78 cf 6d 02 08 ff 00          VA 0xff08026dcf78b0
.  00000167:  9a 01 00                            LAT 1 XLAT
.  0000016a:  b3 b0 78 cf ed 02 08 00 80          PA 0x802edcf78b0 ns=1 ch=0 pat=0
.  00000173:  43 00                               DATA-SOURCE 0
.  00000175:  00 00                               PAD
.  00000177:  71 49 30 ff 36 2d 2f 00 00          TS 51871242727497
<snip>
```

The key data points captured by an SPE sample record include the following:

- Address packet:
  - All operations provide an address packet which provides the PC value.
  - Virtual address for the data accessed.
  - Physical address for the data accessed.
- Latency measurements (LAT):
  - Total latency: Cycle count from the operation being dispatched for issue to the operation being completed. This metric is included for all operations.
  - Issue latency: Cycle count from the operation being dispatched for issue to the operation being issued for execution. This includes any delay in waiting for the operation being ready to issue. This metric is included for all operations.
  - Translation latency: Cycle count from a virtual address being passed to the MMU for translation to the result of the translation being available. This metric is included for all load, store, and atomic operations.
  - Derived execution latency: There is no direct execution latency captured in SPE, but it can be derived as:

    Execution latency = Total latency – Issue latency – Translation latency

Note that this calculation is invalid if the counter saturates.

- Event packet:

  Captures all the SPE-supported PMU events that are generated by the operation execution.

- Operation type packet:

  Captures the operation type, for example load, store, or branch, and other specific information about that operation type. For example, a load might show that it accessed data from a general-purpose register.

- Data source packet:

  Provides the cache or memory source for data accesses. Note that the data source encoding is CPU-specific. It depends on the CHI data source supported by the system fabric for sources outside the CPU.

- Context packet:

  Provides information about the Exception level in which the operation is executed.

- Timestamp packet:

  Provides the exact time when the operation was sampled. This can be used for correlations with other activities in the system.

- Padding:

  The CPU adds padding to force all records to be the same size.

2.3.3. Filtering options in SPE

SPE provides hardware configuration options to filter the data of interest from an SPE sample record before storing it to memory. This helps with targeted profiling for specific operation types, events, and threshold latency values. For example, a user might choose to profile the following:

- Only record load operations.

- Only record store operations.
- Only record branches.
- Only record loads with a latency of over 10 cycles.
- Only record loads that caused a TLB miss.

For more information about the filtering capabilities supported by the SPU, see section D14.5 in the *Arm Architecture Reference Manual for A-profile architecture*.

2.4. SPE monitoring tools

SPE profiling is enabled in the Linux perf tool[3] for data collection. To profile using SPE, the `arm_spe_0` kernel driver enables SPE profiling from the perf tool command line.

The following examples show commands for profiling with SPE:

- Profile all instruction types:

```
perf record -e arm_spe_0// -- test_program
```

- Profile using `-c` to specify the sampling interval:

```
perf record -c 4096 -e arm_spe_0// -- test_program
```

- Profiling using additional control options to filter data:

```
perf record -e arm_spe_0/branch_filter=1,ts_enable=1,
pct_enable=1,pa_enable=1,load_filter=1, jitter=1,store_filter=1/ --
test_program
```

Table 2.3 summarizes the SPE perf tool options.

Table 2.3 SPE perf tool options

SPE perf tool options	Description
`branch_filter`	Record all branch and exception return operations
`event_filter`	Filter samples based on events that are triggered
`jitter`	Add random jitter to the sampling interval
`load_filter`	Record all load operations, including vector loads and atomic operations

SPE perf tool options	Description
`min_latency`	Defines the minimum total latency for filtered operations
`pa_enable`	Enable physical address collection
`pct_enable`	Enable physical timestamp collection
`store_filter`	Record all store operations, including vector stores and all atomic operations
`ts_enable`	Enable timestamp sampling

Arm has also released a helper tool, SPE-Parser[4], to support SPE data collection and analysis for real world workloads. The `spe_parser` tool is available from the tools folder in the Arm Telemetry Solution repository [5]:
https://gitlab.arm.com/telemetry-solution/telemetry-solution/

SPE-Parser is a Python program that parses raw `perf record` data from SPE sampling to help with the export and analysis of SPE traces. The latest version of SPE-Parser supports exporting the data to both CSV and Parquet formats. See the README of the tool for instructions about how to use the tool.

Figure 2.3 shows an SPE profiling exercise carried out with the help of the SPE-Parser. It analyzes the load/store latency of a test program.

Figure 2.3 SPE profiling exercise using the SPE parser

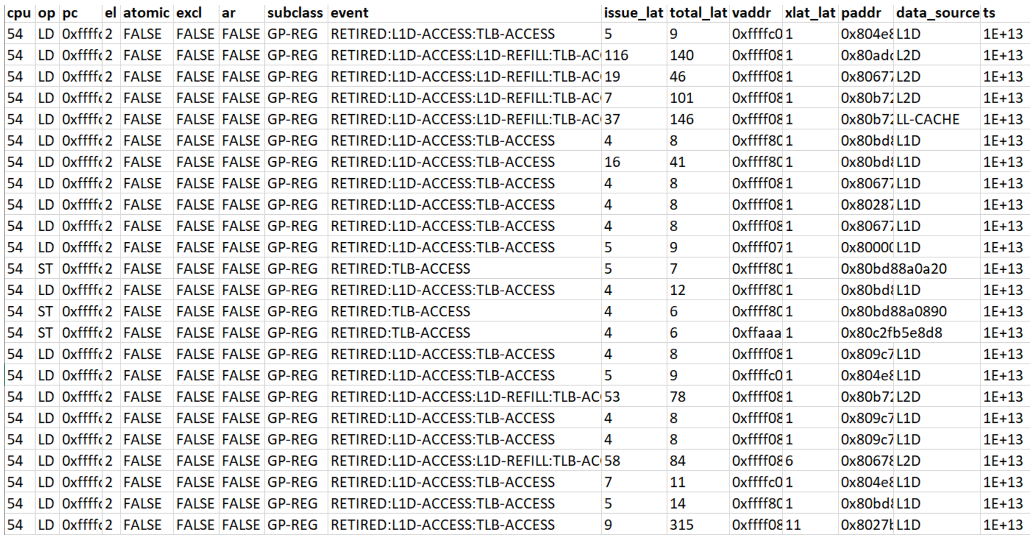
```
# Step 1- install spe-parser
$ git clone https://gitlab.arm.com/telemetry-solution/telemetry-solution
$ cd telemetry-solution/tools/spe_parser
$ pip install .

# Step 2- Record SPE performance data
$ perf record -e
arm_spe_0/branch_filter=1,ts_enable=1,pct_enable=1,pa_enable=1,load_filter=1,jitter=1,sto
re_filter=1,min_latency=0/ -- test_program

# Step 3- Parse the perf binary data and generate output in CSV format
spe-parser perf.data -t csv
```

Figure 2.4 shows the output of the previous steps, containing SPE load/store events parsed and exported to CSV format.

Figure 2.4 An example of SPE-Parser output in CSV format



cpu	op	pc	el	atomic	excl	ar	subclass	event	issue_lat	total_lat	vaddr	xlat_lat	paddr	data_source	ts
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	5	9	0xffffc0	1	0x804e8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	116	140	0xffff08	1	0x80ad	L2D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	19	46	0xffff08	1	0x80677	L2D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	7	101	0xffff08	1	0x80b72	L2D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	37	146	0xffff08	1	0x80b72	LL-CACHE	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff80	1	0x80bd8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	16	41	0xffff80	1	0x80bd8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x80677	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x80287	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x80677	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	5	9	0xffff07	1	0x80000	L1D	1E+13
54	ST	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:TLB-ACCESS	5	7	0xffff80	1	0x80bd88a0a20		1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	12	0xffff80	1	0x80bd8	L1D	1E+13
54	ST	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:TLB-ACCESS	4	6	0xffff80	1	0x80bd88a0890		1E+13
54	ST	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:TLB-ACCESS	4	6	0xffaaa	1	0x80c2fb5e8d8		1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x809c7	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	5	9	0xffffc0	1	0x804e8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	53	78	0xffff08	1	0x80b72	L2D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x809c7	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	4	8	0xffff08	1	0x809c7	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-AC	58	84	0xffff08	6	0x80678	L2D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	7	11	0xffffc0	1	0x804e8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	5	14	0xffff80	1	0x80bd8	L1D	1E+13
54	LD	0xffff	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCESS:TLB-ACCESS	9	315	0xffff08	11	0x8027b	L1D	1E+13

3. Using SPE for root cause analysis

This case study shows how the Arm SPE feature was used to optimize Apache Arrow CSV writer[6] code, achieving a 40% performance improvement.

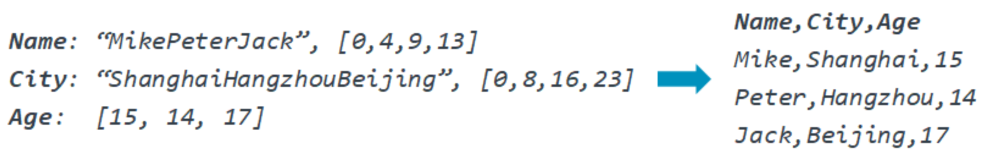
3.1. About the Apache Arrow CSV writer

Apache Arrow is an open-source project for efficient columnar data interchange. The library supports a variety of data structures that can be moved without the overhead of serialization and deserialization. This makes them highly efficient for in-memory computation. Apache Arrow supports multiple languages. This case study uses the C++ implementation of the Arrow CSV Writer.

Unlike a traditional row-based dataset, the Arrow data is column-based. Fields in the same column are contiguous in memory. This type of columnar format is especially convenient for Online Analytical Processing (OLAP) workloads.

Figure 3.1 shows how Arrow CSV Writer converts column-based Arrow data to row-based CSV data.

Figure 3.1 Converting Apache Arrow format data to CSV



```
Name:  "MikePeterJack", [0,4,9,13]
City:  "ShanghaiHangzhouBeijing", [0,8,16,23]
Age:   [15, 14, 17]
```

```
Name,City,Age
Mike,Shanghai,15
Peter,Hangzhou,14
Jack,Beijing,17
```

The left side of Figure 3.1 shows the Arrow data storage format, in which fields of the same column are stored contiguously in memory. The three names Mike, Peter, and Jack are packed in one large column buffer "`MikePeterJack`", with an offset array `[0,4,9,13]` to slice the individual names from that buffer.

The right side of Figure 3.1 shows the same data in CSV format. Each row represents one record and is stored contiguously in memory.

The following sections describe the steps taken to analyze the performance of the CSV writer library on a Neoverse N1 platform and evaluate if there are any optimization opportunities for this library.

This case study was performed on a Neoverse N1 platform built with Ubuntu 20.04 aarch64 OS and clang-12 compiler. The Apache Arrow code used is from the release build (`-O3`). The baseline performance is evaluated on commit `b34f9dfb4` and optimized performance is evaluated on commit `b0422f8df`.

3.2. Evaluate baseline performance

Run the CSV writer benchmark to evaluate the baseline performance in Instructions Per Cycle (IPC) and bandwidth in GB/s.

Figure 3.2 perf stat instruction cycles

```
$ perf stat -e instructions,cycles \
       -- original/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0
-------------------------------------------------------------------------------------------
Benchmark                  Time        CPU         Iterations   UserCounters...
-------------------------------------------------------------------------------------------
WriteCsvStringNoQuote/0   279445 ns   279440 ns   2000         bytes_per_second=1.473G/s

 Performance counter stats for 'original/arrow-csv-writer-benchmark':

        3783506612      instructions              #    2.22  insn per cycle
        1706928603      cycles

       0.569347591 seconds time elapsed

       0.561436000 seconds user
       0.008020000 seconds sys
```

The benchmark result is as follows:

– Processes 1.5GB to 1.8GB of Arrow data per second

– IPC is greater than 2

First, use a top-down methodology to locate hotspots in the code and identify potential bottlenecks or opportunities for performance improvement. For more information about this top-down methodology, see *Arm® Neoverse™ V1 Core: Performance Analysis Methodology white paper*.

On Neoverse N1, stage 1 analysis uses the Cycle Accounting metric group. Table 3.1 shows the results obtained for these metrics.

Table 3.1 Arrow Writer baseline: Cycle Accounting metrics

Frontend Stalled Cycles	Backend Stalled Cycles	Useful Cycle
2.99%	9.63%	87.38%

These results show that the workload is performing efficiently at ~87% useful cycles with a relatively high backend stall rate compared to the front-end stall rate. This is expected for a library function where all the low-hanging optimizations may have been already applied.

As part of the top-down methodology, it is recommended to measure the stage 2 metrics MPKI, Miss Ratio and Operation Mix metric groups for further analysis.

Table 3.2 shows the MPKI results.

Table 3.2 Arrow Writer baseline: MPKI metrics

branch_ mpki	l1d_cache_ mpki	l1i_cache_ mpki	l2_cache_ mpki	l1d_tlb_ mpki	l1i_tlb_ mpki	l2_tlb_ mpki	dtlb_ mpki	itlb_ mpki
0.924	16.998	0.590	0.554	1.822	1.029	0.004	0.002	0.002

Table 3.3 shows the Miss Ratio results.

Table 3.3: Arrow Writer baseline: Miss Ratio metrics

branch_mispredic tion_ratio	l1d_cache_mi ss_ratio	l1i_cache_mis s_ratio	l2_cache_mis s_ratio	l1d_tlb_mis s_ratio	l1i_tlb_miss _ratio	l2_tlb_miss _ratio	dtlb_walk _ratio	itlb_walk _ratio
0.004	0.047	0.002	0.009	0.005	0.004	0.002	0.000	0.000

Table 3.2 and Table 3.3 show that both MPKI and Miss Ratio have relatively high L1D cache metrics and branch metrics.

Figure 3.3 shows the Operation Mix metrics.

Figure 3.3 Apache CSV Writer baseline: Operation Mix

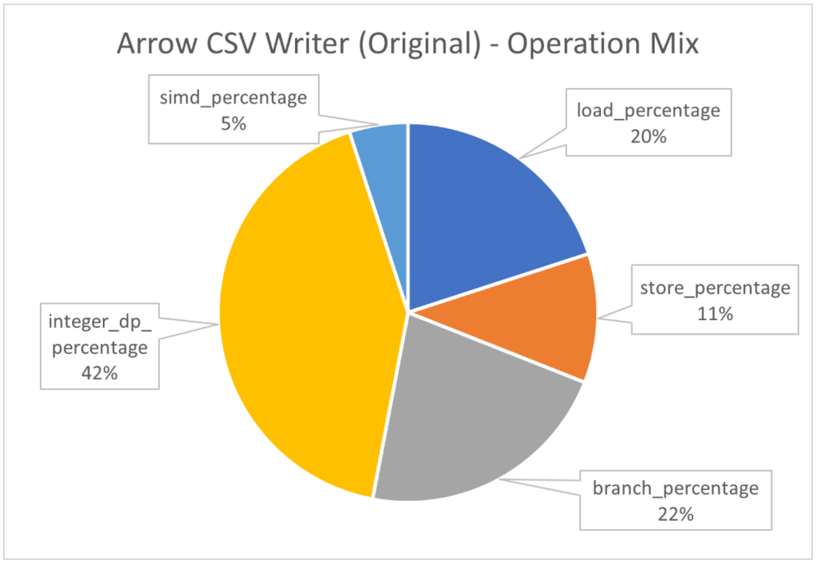


Figure 3.3 shows that the workload has a very high number of integer instructions at 42%, followed by branches at 22%, loads at 20%, and stores at 11%. A very high arithmetic operation count is common and as we are getting very high retiring, it is best to look for vectorization opportunities. The second largest block comes from branch operations (22%) which is also expected but high. Because we found high MPKI and Miss Ratios against branches and L1D, our next performance analysis strategy is to narrow down our hotspot analysis to sample the code at both branch effectiveness and L1D cache events. We will also sample at the high DP% to look for vectorization opportunities.

3.3. Profile branch L1D cache events for hotspot detection

3.3.1. Sample using perf PMU events

Let's profile branches sampling L1D PMU events and annotate the original code using the `perf record` command.

Figure 3.4 perf record command example

```
$ perf record -e L1D_CACHE_REFILL -F 12000 \
       -- original/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0

$ perf report --stdio --no-child
# Samples: 6K of event 'L1D_CACHE_REFILL'
# Event count (approx.): 64258102
#
# Overhead  Command          Shared Object               Symbol
# ........  ...............  ..........................  ......................
#
    53.64%  arrow-csv-write  libc-2.31.so                [.] __GI___memcpy_simd
    32.33%  arrow-csv-write  libarrow.so.900.0.0         [.] arrow::csv::
     3.92%  arrow-csv-write  libarrow.so.900.0.0         [.] memcpy@plt
     1.93%  arrow-csv-write  libc-2.31.so                [.] __memchr_generic
     ... omitted ...
```

The profiling results show that `memcpy` produces the highest number of L1D misses. Let's annotate the source code of the `memcpy` binary and locate the memory accesses that cause cache misses.

```
$ perf annotate --stdio
 Percent |      Source code & Disassembly of libc-2.31.so for L1D_CACHE_REFILL
               (3357 samples, percent: local period)
----------------------------------------------------------------------------------------
         : 3      Disassembly of section .text:
         :
         : 5      0000000000085010 <explicit_bzero@@GLIBC_2.25+0x2a8>:
        4.98 :   85010:  add     x4, x1, x2
        1.97 :   85014:  add     x5, x0, x2
        4.80 :   85018:  cmp     x2, #0x80
        0.15 :   8501c:  b.hi    850e8 <explicit_bzero@@GLIBC_2.25+0x380>  // b.pmore
        7.11 :   85020:  cmp     x2, #0x20
        0.24 :   85024:  b.hi    850a0 <explicit_bzero@@GLIBC_2.25+0x338>  // b.pmore
        7.26 :   85028:  cmp     x2, #0x10
        0.21 :   8502c:  b.cc    85044 <explicit_bzero@@GLIBC_2.25+0x2dc>  // b.lo, b.ul
        0.66 :   85030:  ldr     q0, [x1]
       10.25 :   85034:  ldur    q1, [x4, #-16]
        5.28 :   85038:  str     q0, [x0]
        0.00 :   8503c:  stur    q1, [x5, #-16]
        0.06 :   85040:  ret
        8.23 :   85044:  tbz     w2, #3, 8505c <explicit_bzero@@GLIBC_2.25+0x2f4>
        1.42 :   85048:  ldr     x6, [x1]
        3.92 :   8504c:  ldur    x7, [x4, #-8]
        3.07 :   85050:  str     x6, [x0]
        ... omitted ...
        0.33 :   850a0:  ldp     q0, q1, [x1]
        5.46 :   850a4:  ldp     q2, q3, [x4, #-32]
        6.06 :   850a8:  cmp     x2, #0x40
        0.06 :   850ac:  b.hi    850c0 <explicit_bzero@@GLIBC_2.25+0x358>  // b.pmore
        0.00 :   85110:  stp     q0, q1, [x3, #16]
        ... omitted ...
        1.08 :   85114:  ldp     q0, q1, [x1, #80]
        2.04 :   85118:  stp     q2, q3, [x3, #48]
        0.57 :   8511c:  ldp     q2, q3, [x1, #112]
        3.89 :   85120:  add     x1, x1, #0x40
        0.00 :   85124:  add     x3, x3, #0x40
        0.03 :   85128:  subs    x2, x2, #0x40
        0.00 :   8512c:  b.hi    85110 <explicit_bzero@@GLIBC_2.25+0x3a8>  // b.pmore
        ... omitted ...
```

The `perf record` result suffers from a significant number of perf skid issues. The hotspot profile shows that there are many hot spots causing L1D cache misses. However, the skid issues mean that the result is misleading because many code lines are not memory operations. We can fix this issue by using the precise sampling mechanism provided by Arm SPE, as shown in Sample using SPE for profile branch L1D cache events.

3.3.2. Sample using SPE for profile branch L1D cache events

The following example uses SPE.

```
$ perf record -e arm_spe_0/load_filter=1,store_filter=1,jitter=1/ \
       -- original/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0

$ perf report --stdio --no-child
# Samples: 6K of event 'l1d-miss'
# Overhead  Command          Shared Object                Symbol
# ........  ...............  ...........................  ...............................
#
    30.55%  arrow-csv-write  libc-2.31.so                 [.] __GI___memcpy_simd
    12.86%  arrow-csv-write  libarrow.so.900.0.0          [.] arrow::csv::…
     5.43%  arrow-csv-write  libarrow.so.900.0.0          [.] arrow::csv::…
     4.31%  arrow-csv-write  libarrow.so.900.0.0          [.] arrow::ArraySpan::…
     ... omitted ...
```

The profiling results show that `memcpy` produces the highest number of L1D misses. Let's annotate the source code of the `memcpy` binary and locate the memory accesses that cause misses.

```
$ perf annotate --stdio
Percent |  Source code & Disassembly of libc-2.31.so for l1d-miss
           (1886 samples, percent: local period)
--------------------------------------------------------------------------------------
         : 3     Disassembly of section .text:
         :
         : 5     0000000000085010 <explicit_bzero@@GLIBC_2.25+0x2a8>:
    0.00 :   85010:  add     x4, x1, x2
    0.00 :   85014:  add     x5, x0, x2
    0.00 :   85018:  cmp     x2, #0x80
    0.00 :   8501c:  b.hi    850e8 <explicit_bzero@@GLIBC_2.25+0x380>  // b.pmore
    ... omitted ...
+-> 0.00 :   85110:  stp     q0, q1, [x3, #16]
|  38.44 :   85114:  ldp     q0, q1, [x1, #80]
|   0.00 :   85118:  stp     q2, q3, [x3, #48]
|   4.56 :   8511c:  ldp     q2, q3, [x1, #112]
|   0.00 :   85120:  add     x1, x1, #0x40
|   0.00 :   85124:  add     x3, x3, #0x40
|   0.00 :   85128:  subs    x2, x2, #0x40
+-- 0.00 :   8512c:  b.hi    85110 <explicit_bzero@@GLIBC_2.25+0x3a8>  // b.pmore
    ... omitted ...
```

The previous annotation using perf record shows the series of memory operations that could be causing the misses.

As expected, `memcpy` is a frequently repeating loop copying blocks of memory. The instruction `ldp q0, q1, [x1, #80]` loads 64 bytes, the L1D cache size, from the input buffer into two Neon registers. This causes L1D cache misses because the input buffer is much bigger than the L1D cache size.

We now try to map this to the C++ source code to look for optimization opportunities.

```
auto valid_function = [&](arrow::util::string_view s) {
  memcpy(output + *offsets, s.data(), s.length());
  memcpy(output + *offsets + s.length(), end_chars_.c_str(), end_chars_.size());
  *offsets += static_cast<int64_t>(s.length() + end_chars_.size());
  offsets++;
  return Status::OK();
};
```

The first `memcpy` operation copies a string `s` from the input buffer. In the test dataset, the strings are mostly long enough to trigger the hot loop in the above assembly code. The second `memcpy` operation copies an end character which is at most two bytes in size. This does not trigger the hot loop. We cannot control the test dataset size and pattern, so there is little we can do to avoid this cache miss.

3.4. Profile branch mispredictions for hotspot detection

3.4.1. Sample using perf PMU events

Let's now profile branches and annotate the original code using the `perf record` command.

```
$ perf record -e BR_MIS_PRED_RETIRED \
       -- original/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0

$ perf report --stdio
# Overhead  Command          Shared Object               Symbol
# ........  ...............  ..........................  ..........................
#
    42.89%  arrow-csv-write  libc-2.31.so                [.] __GI___memcpy_simd
    21.45%  arrow-csv-write  libarrow.so.900.0.0         [.] arrow::csv::…
     3.25%  arrow-csv-write  libarrow_testing.so.900.0.0 [.] arrow::util::…
     2.10%  arrow-csv-write  libarrow.so.900.0.0         [.] arrow::ArraySpan::…
     2.01%  arrow-csv-write  libarrow.so.900.0.0         [.] arrow::csv::…
     1.78%  arrow-csv-write  libarrow.so.900.0.0         [.] memcpy@plt
```

The profiling result shows that `memcpy` produces the highest number of branch mispredictions. Let's annotate the source code of the `memcpy` binary and locate the branches that cause mispredictions.

```
$ perf annotate -s __GI___memcpy_simd --stdio
 Percent |          Source code & Disassembly of libc-2.31.so for BR_MIS_PRED_RETIRED
                    (1270 samples, percent: local period)
-------------------------------------------------------------------------------------------
         :
         : 3     Disassembly of section .text:
         :
         : 5     0000000000085010 <explicit_bzero@@GLIBC_2.25+0x2a8>:
    4.42 :   85010:  add     x4, x1, x2
    0.00 :   85014:  add     x5, x0, x2
    1.29 :   85018:  cmp     x2, #0x80
    1.40 :   8501c:  b.hi    850e8 <explicit_bzero@@GLIBC_2.25+0x380>
    7.53 :   85020:  cmp     x2, #0x20
    0.00 :   85024:  b.hi    850a0 <explicit_bzero@@GLIBC_2.25+0x338>
    6.26 :   85028:  cmp     x2, #0x10
    0.00 :   8502c:  b.cc    85044 <explicit_bzero@@GLIBC_2.25+0x2dc>
    2.39 :   85030:  ldr     q0, [x1]
   24.49 :   85034:  ldur    q1, [x4, #-16]
   11.00 :   85038:  str     q0, [x0]
    0.00 :   8503c:  stur    q1, [x5, #-16]
    0.00 :   85040:  ret
    0.79 :   85044:  tbz     w2, #3, 8505c <explicit_bzero@@GLIBC_2.25+0x2f4>
   13.41 :   85048:  ldr     x6, [x1]
    2.59 :   8504c:  ldur    x7, [x4, #-8]
    1.55 :   85050:  str     x6, [x0]
    0.00 :   85054:  stur    x7, [x5, #-8]
    0.55 :   85058:  ret
```

The previous annotation using `perf record` suffers from severe skid issues because the annotated hot spots occur in many locations and are not accurate because most of them are not branch instructions. This sampling skid is a well-known issue with PMU-based sampling due to the latencies caused by taking interrupts during sampling. SPE-based profiling can help with this issue as demonstrated in Sample using SPE for profile branch mispredictions.

3.4.2. Sample using SPE for profile branch mispredictions

```
$ perf record -e arm_spe_0/branch_filter=1,jitter=1/ \
       -- original/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0
$ perf report --stdio
#
# Children      Self  Command          Shared Object              Symbol
# ........  ........  ...............  .........................  ......................
#
    79.53%    79.53%  arrow-csv-write  libc-2.31.so               [.] __GI___memcpy_simd
     4.45%     4.45%  arrow-csv-write  libc-2.31.so               [.] __memchr_generic
     2.63%     2.63%  arrow-csv-write  libarrow.so.900.0.0        [.] arrow::compute::…
     1.67%     1.67%  arrow-csv-write  libarrow.so.900.0.0        [.] arrow::csv::…
     0.99%     0.99%  arrow-csv-write  ld-2.31.so                 [.] _dl_lookup_…
```

The SPE annotated sample shows that the `libc memcpy` library is causing a high number of branch mispredictions, 79.53%. As seen before, this version of the `memcpy` function is highly optimized based on the buffer size to be copied. The size boundaries are checked in the order of 128, 32, 16, 8 and 4. SIMD registers are used to copy large data blocks.

To locate the mispredicted branches in the `memcpy` library, we can use the `perf annotate` command on the SPE sampled record to investigate the `memcpy` assembly code.

```
$ perf annotate -s __GI___memcpy_simd --stdio
 Percent |         Source code & Disassembly of libc-2.31.so for branch-miss
                   (9727 samples, percent: local period)
-----------------------------------------------------------------------------------------
         :
         : 3     Disassembly of section .text:
         :
         : 5     0000000000085010 <explicit_bzero@@GLIBC_2.25+0x2a8>:
    0.00 :   85010:   add     x4, x1, x2
    0.00 :   85014:   add     x5, x0, x2
    0.00 :   85018:   cmp     x2, #0x80
    0.00 :   8501c:   b.hi    850e8 <explicit_bzero@@GLIBC_2.25+0x380>
    0.00 :   85020:   cmp     x2, #0x20
    0.26 :   85024:   b.hi    850a0 <explicit_bzero@@GLIBC_2.25+0x338>
    0.00 :   85028:   cmp     x2, #0x10
    0.83 :   8502c:   b.cc    85044 <explicit_bzero@@GLIBC_2.25+0x2dc>
    0.00 :   85030:   ldr     q0, [x1]
    0.00 :   85034:   ldur    q1, [x4, #-16]
    0.00 :   85038:   str     q0, [x0]
    0.00 :   8503c:   stur    q1, [x5, #-16]
    0.00 :   85040:   ret
   97.28 :   85044:   tbz     w2, #3, 8505c <explicit_bzero@@GLIBC_2.25+0x2f4>
    0.00 :   85048:   ldr     x6, [x1]
    0.00 :   8504c:   ldur    x7, [x4, #-8]
    0.00 :   85050:   str     x6, [x0]
    0.00 :   85054:   stur    x7, [x5, #-8]
    0.10 :   85058:   ret
```

From the annotated assembly code above, we can see that almost all the mispredictions (97.28%) came from the branch opcode at address `0x85044` for samples traced by SPE. It is important to note that while SPE is a statistical sampling approach, we can still expect hot loops to be caught by this approach.

To further understand the branch execution behavior, the branching logic in the function is summarized as follows:

```
97.28 : 85044: tbz w2, #3, 8505c <explicit_bzero@@GLIBC_2.25+0x2f4>
```

This instruction jumps to the branch target address if the third bit of register w2 (`memcpy` buffer size to be copied) is not set. The code only arrives at this instruction when the buffer size is less than 16. So, this instruction logic can be written in C as `if (size < 8) goto .L8505c;`. Branch mispredictions are caused by this logic where the buffer size needs to be resolved from the comparison operation to determine the code path.

To evaluate potential opportunities to improve this branch performance, we examined the main loop of the Arrow CSV Writer. The main logic of the CSV writer is to convert Arrow column-based data to CSV row-based data. To do this, the Arrow CSV writer continuously copies data fields from the Arrow buffer to the CSV buffer. After copying each data field, it must append either a comma delimiter or end-of-line to the copied data. The following example code shows the hot loop of the library:

```
Loop {
    memcpy(csv_line, arrow_field, size_of_field);         // copy a field of normal size
    memcpy(csv_line, delimiter_or_end-of-line, 1_or_2);   // copy a very short separator
}
```

The Arrow CSV writer calls `memcpy` with a data field (normal size) and a very short delimiter (1 or 2 bytes) repeatedly. This pattern makes it harder to train the CPU branch predictor because the two `memcpy` operations are combined in a single loop. The first `memcpy` operation populates fields in column order, which is easy to predict because fields of the same column are often of similar size. However, the second `memcpy` operation with the delimiter breaks this prediction path.

It might appear that using `memcpy` to append a delimiter is inefficient, because this could be as simple as appending characters using `*buf++ = ',';`. However, the reason for using `memcpy` is that the delimiter is configurable at run time and can be any length in theory. The Arrow code must be general enough to handle this situation, even though the delimiter and end-of-line are almost never longer than 2 characters.

3.5. Optimizing the CSV writer

Once the potential performance issue has been identified, the function calls in the hot loop can be modified to change the code path. The approach we used to improve this is to define a simpler `memcpy` clone, `copy_separator`, as a helper function to copy delimiter and end-of-line characters, keeping the Arrow `libc` `memcpy` for copying the normal data fields of Arrow buffer as shown in Figure 3.5.

Figure 3.5 CSV writer code optimization

```
Loop {
    memcpy(csv_line, arrow_field, size_of_field);        // copy a field of normal size
    ~~memcpy(csv_line, delimiter_or_end-of-line, 1_or_2);  // copy a very short separator~~
    copy_separator(csv_line, delimiter_or_end-of-line, 1_or_2); // copy separator
}

void copy_separator(char* dst, const char* src, int n) {
    if (n == 1) memcpy(dst, src, 1);
    else if (n == 2) memcpy(dst, src, 2);
    else memcpy(dst, src, n);
}
```

The helper function `copy_separator`, might not look like an optimization at first sight as it still calls `memcpy`. However, this `memcpy(src,dst,n)` is simpler and guaranteed for the compiler to optimize to single load/store instructions when the buffer size is known at compile time. We expect this helper function to be executed in most cases because using a delimiter size longer than two characters is rare and will fall back to using the `libc memcpy`.

3.5.1. Benchmark optimized code

To test for performance improvements, we profile the optimized code again.

```
$ perf stat -e instructions,cycles,BR_RETIRED,BR_MIS_PRED_RETIRED \
       -- optimized/arrow-csv-writer-benchmark --benchmark_filter=WriteCsvStringNoQuote/0
------------------------------------------------------------------------------------------
Benchmark                  Time        CPU         Iterations   UserCounters...
------------------------------------------------------------------------------------------
WriteCsvStringNoQuote/0   192238 ns   192233 ns   2000         bytes_per_second=2.142G/s

 Performance counter stats for 'optimized/arrow-csv-writer-benchmark --
benchmark_filter=WriteCsvStringNoQuote/0':

        3063222827      instructions              #    2.58  insn per cycle
        1186036951      cycles
         609796441      BR_RETIRED
           1507719      BR_MIS_PRED_RETIRED

       0.395666338 seconds time elapsed

       0.395778000 seconds user
       0.000000000 seconds sys
```

Comparing the performance of the optimized code against the original code:

- The benchmark throughput improves from 1.5~1.8GB/s to 2.1GB/s, with a 20% to 40% uplift. The original code has high variance in performance based on our observations, which is the reason for the range of throughput values.
- Total instructions drop from 3.78E+9 to 3.06E+9 and branches drop from 8.50E+8 to 6.10E+8.
- IPC increases slightly from 2.22 to 2.58.

Let's look at the improvements in the key top-down methodology metrics in Table 3.4, Table 3.5, and Figure 3.6.

MPKI

Table 3.4 Arrow Writer optimized: MPKI

branch_mpki	l1d_cache_mpki	l1i_cache_mpki	l2_cache_mpki	l1d_tlb_mpki	l1i_tlb_mpki	l2_tlb_mpki	dtlb_mpki	itlb_mpki
0.494	20.990	0.871	0.909	2.304	1.351	0.003	0.011	0.002

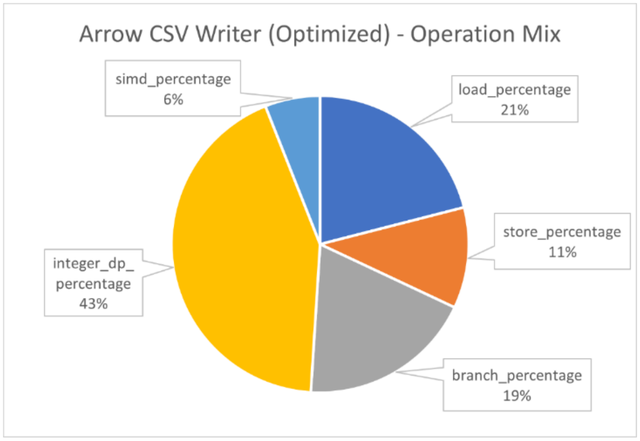
Miss Ratio

Table 3.5 Arrow Writer optimized: Miss Ratio

branch_misprediction_ratio	l1d_cache_miss_ratio	l1i_cache_miss_ratio	l2_cache_miss_ratio	l1d_tlb_miss_ratio	l1i_tlb_miss_ratio	l2_tlb_miss_ratio	dtlb_walk_ratio	itlb_walk_ratio
0.003	0.057	0.005	0.011	0.006	0.006	0.001	0.000	0.000

Operation Mix

Figure 3.6 Apache Arrow CSV Writer (optimized): Operation Mix



Comparing the branch-related micro-architecture metrics before and after optimization from Table 3.4, Table 3.5, and Figure 3.6 we observe the changes in the branch effectiveness metrics, as shown in Table 3.6.

Table 3.6 Branch effectiveness metrics before and after optimization

	Branch MPKI	Branch Misprediction Ratio	Branch Operations Percentage
Before	0.924	0.4%	22.12%
After	0.494	0.3%	19.57%
Reduction	43%	0.1%	2.55%

Table 3.6 shows that all the branch-related metrics reduced significantly with our optimization which contributed to the performance uplift. As expected, the branch

instructions were reduced with the helper function converting the code to loads and stores.

3.6. Summary

Using a top-down methodology and SPE, we identified an opportunity to optimize a branch in the Arrow CSV writer hot loop, adding an extra helper function to achieve a considerable performance improvement. The SPE precise sampling capability helped to locate the branch instruction which provided hints to the potential performance optimization opportunity. The precise sampling in SPE specifically helped to locate the exact branch instruction by mitigating the skid issue associated with PMU sampling.

You can see the upstream pull request in GitHub: https://github.com/apache/arrow/pull/13394

We highly recommend using SPE for hotspot analysis and root cause detection while tuning code on Arm Neoverse cores.

4. Using SPE for memory access analysis

SPE-based profiling can provide key insights into the execution of memory operations. These insights can be used to derive memory latency values for loads as well as other information such as the distribution of execution latency.

In this case study, we demonstrate how SPE can be used for memory access analysis using the LMbench [7] benchmark, which is commonly used to measure memory bandwidth and latency on target platforms.

We use SPE to profile LMbench execution and compare the memory latency and bandwidth data from SPE to LMbench measurements and evaluate how the two approaches correlate. SPE also provides further insights into the execution of operations, for example the list of key PMU events triggered during execution, and operation latency distributions.

Figure 4.1 SPE memory latency measurements

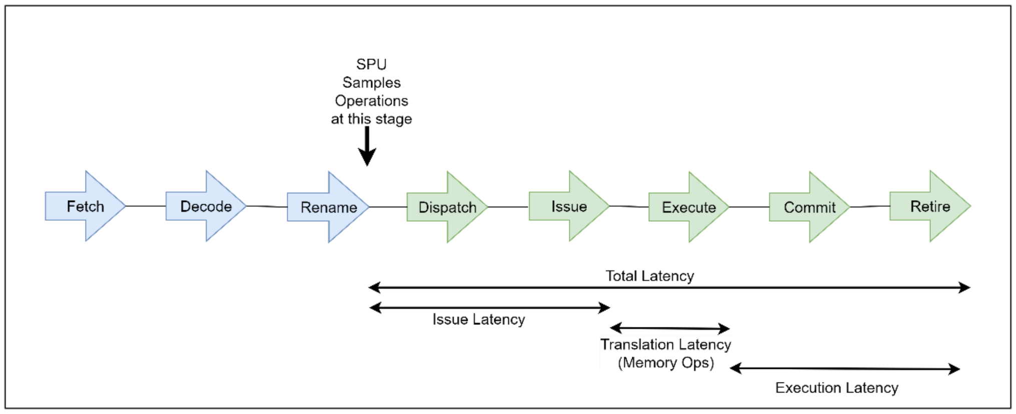


Table 4.1 defines the SPE latency measurements, which can be mapped against the micro-architectural diagram in Figure 4.1 for the pipeline stages where the measurements are taken from.

Table 4.1 SPE Latency metrics definition

Latency	Description
Total Latency (LAT TOT)	Counts cycles starting from the rename stage until the operation is retired. Operations: All
Issue Latency (LAT ISSUE)	Counts cycles starting from the rename stage until the operation is issued for execution by the Issue Queue. This helps to evaluate execution delay caused by the operation waiting to be issued. Operations: All
Translation Latency (LAT XLAT)	Counts cycles starting from the operation issue stage for memory access operations from when the virtual address is passed to the MMU for address translation until the result of the translation is available. Operations: Load, store, and atomic operations.
Execution Latency (Derived) LAT_TOT – (LAT_ISSUE + LAT_XLAT)	Derived latency cycle measures the execution latency for the operation, which is essentially the latency of execution in the functional unit when the operation is issued for execution. This is particularly helpful in case of load operations to measure the memory subsystem latency for a load.

SPE also provides data source information, recording hierarchical data source hits for loads if supported by the system. Data source encoding depends on the CHI encoding supported by the system interconnect and respective CPU events supported by the process. Figure 4.2 and Table 4.2 show data source encoding on a Neoverse N1 system.

Figure 4.2 Neoverse N1 data sources

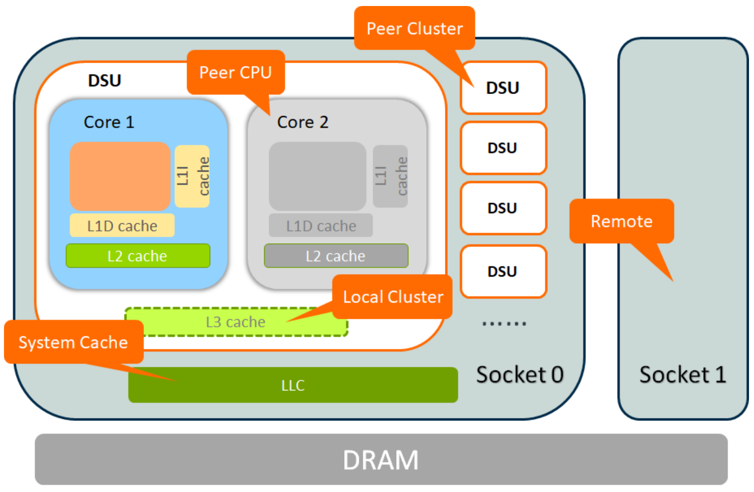


Table 4.2 Neoverse N1 data source bit encoding

Value	Name	Notes
`0b0000`	L1 data cache	Level 1 data cache in the core memory subsystem
`0b1000`	L2 cache	Level 2 unified cache in the core memory subsystem
`0b1001`	Peer CPU	Snoop from peer CPU within the same DSU
`0b1010`	Local cluster	L3 cache if present in the DSU
`0b1011`	System cache	Last level cache
`0b1100`	Peer cluster	Peer cluster should be a core in another DSU of the same CPU chip
`0b1101`	Remote	Remote CPUs in another socket
`0b1110`	DRAM	RAM in the local chip

4.1. LMbench introduction

LMbench is a portable benchmark suite of ANSI/C microbenchmarks for UNIX/POSIX that supports a variety of memory system analysis metrics including memory latency and bandwidth. For more information on LMbench, see https://lmbench.sourceforge.net/. LMbench provides system designers with performance costs of memory access operations on a given platform.

For memory latency and bandwidth measurements, the two main tests available from LMbench are as follows:

- `lat_mem_rd`: This test is for memory read latency benchmarking. Strided reads are performed for different memory sizes, providing latency information for different levels of data access.
- `bw_mem`: This test is for memory bandwidth benchmarking. Several different benchmarks are available, including the following:
  - `frd`: full memory reads
  - `fwr`: full memory writes
  - `rd`: strided memory reads
  - `wr`: strided memory writes

4.2. Memory read latency test

The `lat_mem_rd` test from LMbench performs a memory read latency test. This test performs repeated memory read operations with different memory sizes to stress the memory subsystem. In the code path, the read data is cached when the size of the read is smaller than a cache level. This ensures that the read latency matches the access latency to that cache level when the read data is cached. As the read size increases, the data is stored in lower cache levels following the cache hierarchy of the system, which provides the corresponding read latency until the access reaches the farthest memory source, which is usually DRAM.

Using SPE-enabled memory access operations, we can check whether the read is getting data from the hierarchical data source levels targeted by the LMbench test by analyzing the profiled data of hits and cycles for the memory access operation. In addition to direct latency information like issue latency, translation latency, and total latency, SPE can also provide additional insights into the execution of memory access operations. For example, SPE packets record PMU events such as TLB Refill events, which indicates that the sampled memory access operation resulted in a TLB miss.

4.2.1. Step 1: Run lmbench lat_mem_rd with SPE profiling enabled

The `lat_mem_rd` test provides the following configuration options to obtain memory read latency information:

- The first option `-P 1` means using only one process to do the benchmarking.
- The second option is the maximum memory size to read in megabytes (MB). Here we use 1024 which is large enough to hit DRAM.
- The third option is memory read stride size. Fine selected size could avoid impact from prefetch and hence we will use 4096.

Figure 4.3 lat_mem_rd run example

```
$ taskset 0x10000000000 ./lat_mem_rd -P 1 1024 4096
   "stride=4096
   0.00391 1.429
   0.00586 1.429
   0.00781 1.429                <-- Collect SPE data for P1
   0.00977 1.429
   ....
   0.12500 5.048
   0.14062 5.082
   0.15625 5.057                <-- Collect SPE data for P2
   0.17188 5.035
   ....
   14.00000 36.567
   15.00000 36.551
   16.00000 36.57               <-- Collect SPE data for P3
   18.00000 36.725
   ....
   576.00000 100.889
   640.00000 101.149
   704.00000 101.015            <-- Collect SPE data for P4
   768.00000 101.141
   ....
   1024.00000 101.054
```

The output in the Figure 4.3 shows that the LMbench test reports a range of reads from 0.00391MB to 1024MB in size, with latency for each read rising from 1.429 ns to 101.054 ns. While running LMbench we also profiled it using SPE to extract similar performance information from the micro-architecture. We run `lat_mem_rd` on core 40, and then we manually trigger perf SPE to profile 4 times against it at the four points shown in Figure 4.3.

The Arm SPE filtering mechanism allows us to only record load operations when profiling memory reads. We configure `arm_spe_0` by setting `branch_filter=0`, `load_filter=1` and `store_filter=0`. For each run, a `perf.data` file is generated from the SPE records. For profiling, we run the following command on a SPE supported platform with Linux perf driver for `arm_spe` installed.

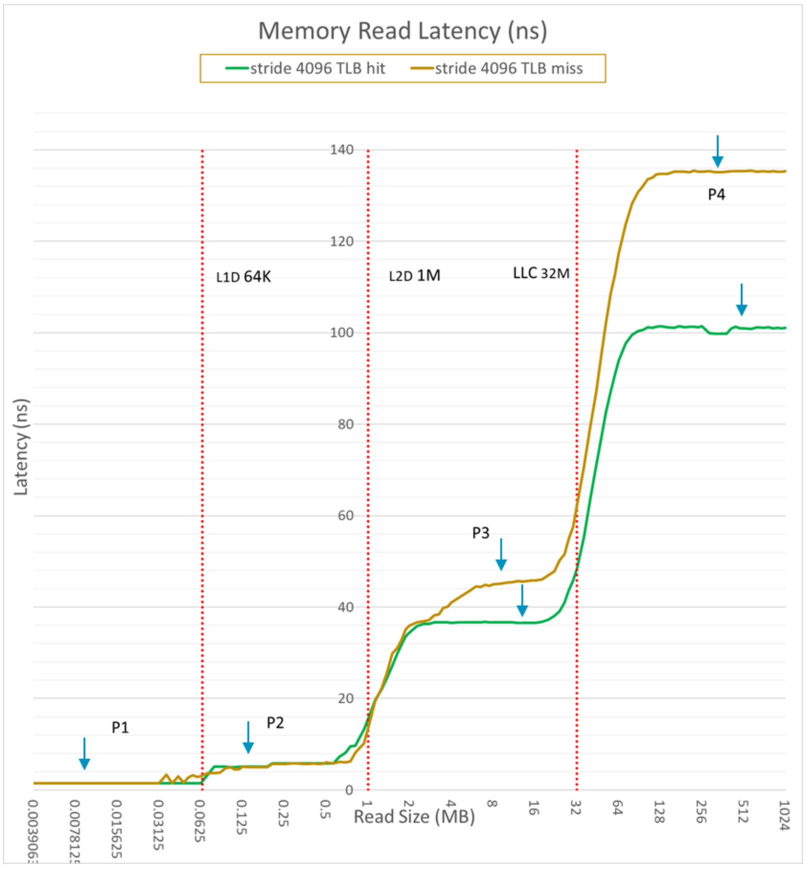
```
$ perf record -C 40 -e
arm_spe_0/branch_filter=0,ts_enable=1,pct_enable=1,pa_enable=1,load_filter=1
,jitter=1,store_filter=0,min_latency=0/
```

Next, we run the same test again using -t option with `lat_mem_rd` to create TLB miss scenarios for the memory loads to evaluate memory translation costs using SPE data. The latency curve in Figure 4.4 shows the results from the test, identifying each of the four parts of the curve with the -t option on or off. The command to run this test with the `-t` option is as follows.

```
$ taskset 0x10000000000 ./lat_mem_rd -P 1 -t 1024 4096
```

The arrows on the chart indicate where the SPE profiling was triggered during the benchmark execution. The read access size changes produced by LMbench tests are manually observed to identify the region of interest for profiling.

Figure 4.4 Memory read latency curve



The platform under test has four hierarchical memory sources:

- Private L1 and L2 caches in the core
- A last level interconnect cache (LLC) in the fabric shared by all cores
- DRAM

The cache sizes are 64KB L1D cache, 1MB L2 cache and 32MB LLC. As expected, we see four plateaus in the latency curve in Figure 4.4, each corresponding to the L1 data cache, L2 data cache, LLC, and RAM hits for memory reads. The curve generated from the SPE data matches well with the cache size on the platform we used for generating this data.

4.2.2. Step 2: Memory access analysis using SPE-profiled data

We process the SPE-profiled data to derive further insights into memory access analysis. The three main data points from SPE that can be validated against the LMbench benchmark results are:

- Latency: The latency value generated by `lat_mem_rd` test should match the SPE derived latency values statistically.
- Data sources: The four plateaus in the LMbench latency curve to match the hierarchical memory access data sources: L1D, L2, LLC and DRAM.
- TLB miss events: Higher latency data obtained on P3 and P4 for the `-t` enabled test that causes TLB misses (indicated by the brown line in Figure 4.4) should result in TLB miss related PMU events being triggered.

For this analysis, we used the `spe-parser` tool to process the SPE-profiled raw data collected using Linux perf tool. For more information about both these tools, see SPE monitoring tools.

Run the following command to process the captured perf record data in `perf.data` and save all the SPE records into a CSV file.

```
$ spe-parser perf.data -t csv
```

Figure 4.5 shows the processed SPE records from the SPE parser tool for the sampling point P2. Each row is one record for an SPE-profiled instruction such as a load. The fields are defined as follows:

- CPU refers to the core on which the operation ran.
- OP means operation types, either load, store, branch and so on.
- PC is the program counter address.
- EL refers to the exception level. 0 means running from user space and 2 means running from kernel space.

– ATOMIC, EXCL, and AR indicate if the operation is atomic, exclusive, or acquire-release.

– SUBCLASS GP-REG means the operation targets general purpose registers.

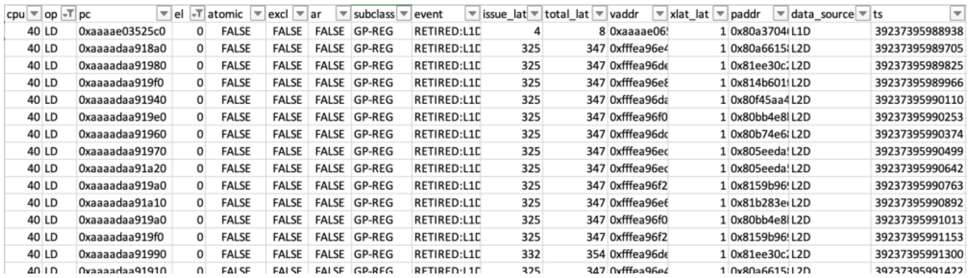
– VADDR and PADDR are the virtual and physical addresses of the data.

– DATA SOURCE is where the data is loaded from. This is a supported hierarchical data source such as L1 data cache, L2 cache, LLC, or RAM.

– TS is the time stamp counter for the operation.

There are three types of latencies recorded by the micro-architecture as explained in Using SPE for memory access analysis: TOTAL_LAT (total latency), ISSUE_LAT (issue latency), and XLAT_LAT (translation latency).

Figure 4.5 Processed SPE records for lat_mem_rd



cpu	op	pc	el	atomic	excl	ar	subclass	event	issue_lat	total_lat	vaddr	xlat_lat	paddr	data_source	ts
40	LD	0xaaaae03525c0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	4	8	0xaaaae06	1	0x80a3704	L1D	39237395988938
40	LD	0xaaaadaa918a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96e4	1	0x80a6615	L2D	39237395989705
40	LD	0xaaaadaa91980	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96de	1	0x81ee30c	L2D	39237395989825
40	LD	0xaaaadaa919f0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96e8	1	0x814b601	L2D	39237395989966
40	LD	0xaaaadaa91940	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96da	1	0x80f45aa4	L2D	39237395990110
40	LD	0xaaaadaa919e0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96f0	1	0x80bb4e8	L2D	39237395990253
40	LD	0xaaaadaa91960	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96dc	1	0x80b74e6	L2D	39237395990374
40	LD	0xaaaadaa91970	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96ec	1	0x805eeda	L2D	39237395990499
40	LD	0xaaaadaa91a20	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96ec	1	0x805eeda	L2D	39237395990642
40	LD	0xaaaadaa919a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96f2	1	0x8159b96	L2D	39237395990763
40	LD	0xaaaadaa91a10	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96e6	1	0x81b283e	L2D	39237395990892
40	LD	0xaaaadaa919a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96f0	1	0x80bb4e8	L2D	39237395991013
40	LD	0xaaaadaa919f0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96f2	1	0x8159b96	L2D	39237395991153
40	LD	0xaaaadaa91990	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	332	354	0xfffea96de	1	0x81ee30c	L2D	39237395991300
40	LD	0xaaaadaa91910	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D	325	347	0xfffea96e4	1	0x80a6615	L2D	39237395991422

The SPE-processed data may contain records triggered by instructions from the full program that are not directly related to the hot loop of the benchmark code executing the read or writes. To identify the hot loop of the code, SPE can help by filtering records using the PC address. In this case, we can use the PC address range to extract the records to locate benchmark-specific operations. By inspecting the source code of `lat_mem_rd`, we find that the `benchmark_loads()` function is the benchmark portion of the program. Using assembler inspection tools such as objdump, we can locate the PC of this function with offset 189c~1a28 which are the code addresses of interest as shown in Figure 4.6.

Figure 4.6 benchmark_loads function assembly code

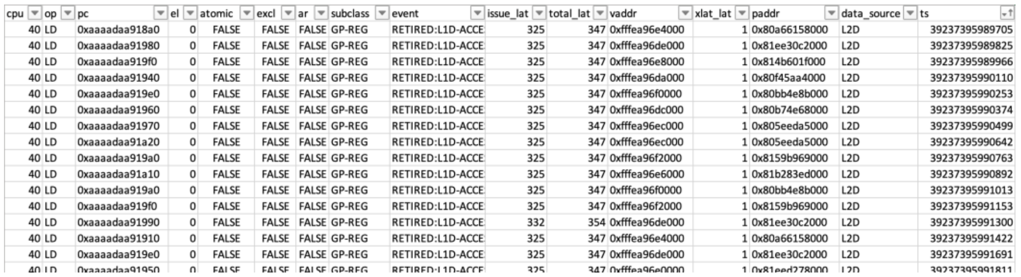
```
000000000000184c <benchmark_loads>:
    184c:       a9be7bfd        stp     x29, x30, [sp, #-32]!
    1850:       910003fd        mov     x29, sp
    1854:       a90153f3        stp     x19, x20, [sp, #16]
    1858:       aa0103f4        mov     x20, x1
    185c:       f9400833        ldr     x19, [x1, #16]
    1860:       f9405422        ldr     x2, [x1, #168]
    1864:       8b020441        add     x1, x2, x2, lsl #1
    1868:       8b010c41        add     x1, x2, x1, lsl #3
    186c:       d37ef421        lsl     x1, x1, #2
    1870:       f9404e82        ldr     x2, [x20, #152]
    1874:       9ac10841        udiv    x1, x2, x1
    1878:       91000424        add     x4, x1, #0x1
    187c:       d1000403        sub     x3, x0, #0x1
    1880:       b5000e20        cbnz    x0, 1a44 <benchmark_loads+0x1f8>
    1884:       aa1303e0        mov     x0, x19
    1888:       9400071c        bl      34f8 <use_pointer>
    188c:       f9000a93        str     x19, [x20, #16]
    1890:       a94153f3        ldp     x19, x20, [sp, #16]
    1894:       a8c27bfd        ldp     x29, x30, [sp], #32
    1898:       d65f03c0        ret
    189c:       f9400260        ldr     x0, [x19]
    18a0:       f9400000        ldr     x0, [x0]
    18a4:       f9400000        ldr     x0, [x0]
    18a8:       f9400000        ldr     x0, [x0]
    18ac:       f9400000        ldr     x0, [x0]
    18b0:       f9400000        ldr     x0, [x0]

    <snip>

    1a10:       f9400000        ldr     x0, [x0]
    1a14:       f9400000        ldr     x0, [x0]
    1a18:       f9400000        ldr     x0, [x0]
    1a1c:       f9400000        ldr     x0, [x0]
    1a20:       f9400000        ldr     x0, [x0]
    1a24:       f9400000        ldr     x0, [x0]
    1a28:       f9400013        ldr     x19, [x0]
    1a2c:       eb02003f        cmp     x1, x2
    1a30:       91000442        add     x2, x2, #0x1
    1a34:       54fff341        b.ne    189c <benchmark_loads+0x50>  // b.any
    1a38:       d1000463        sub     x3, x3, #0x1
    1a3c:       b100047f        cmn     x3, #0x1
    1a40:       54fff220        b.eq    1884 <benchmark_loads+0x38>  // b.none
    1a44:       d2800002        mov     x2, #0x0                        // #0
    1a48:       b5fff2a4        cbnz    x4, 189c <benchmark_loads+0x50>
    1a4c:       17fffffb        b       1a38 <benchmark_loads+0x1ec>
```

After filtering for the region of interest in the program records, we obtain the records as shown in Figure 4.7 which can be used for further analysis.

Figure 4.7 Filtered SPE Records for lat_mem_rd



cpu	op	pc	el	atomic	excl	ar	subclass	event	issue_lat	total_lat	vaddr	xlat_lat	paddr	data_source	ts
40	LD	0xaaaadaa918a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96e4000	1	0x80a66158000	L2D	39237395989705
40	LD	0xaaaadaa91980	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96de000	1	0x81ee30c2000	L2D	39237395989825
40	LD	0xaaaadaa919f0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96e8000	1	0x814b601f000	L2D	39237395989966
40	LD	0xaaaadaa91940	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96da000	1	0x80f45aa4000	L2D	39237395990110
40	LD	0xaaaadaa919e0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96f0000	1	0x80bb4e8b000	L2D	39237395990253
40	LD	0xaaaadaa91960	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96dc000	1	0x80b74e68000	L2D	39237395990374
40	LD	0xaaaadaa91970	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96ec000	1	0x805eeda5000	L2D	39237395990499
40	LD	0xaaaadaa91a20	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96ec000	1	0x805eeda5000	L2D	39237395990642
40	LD	0xaaaadaa919a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96f2000	1	0x8159b969000	L2D	39237395990763
40	LD	0xaaaadaa91a10	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96e6000	1	0x81b283ed000	L2D	39237395990892
40	LD	0xaaaadaa919a0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96f0000	1	0x80bb4e8b000	L2D	39237395991013
40	LD	0xaaaadaa919f0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96f2000	1	0x8159b969000	L2D	39237395991153
40	LD	0xaaaadaa91990	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	332	354	0xfffea96de000	1	0x81ee30c2000	L2D	39237395991300
40	LD	0xaaaadaa91910	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96e4000	1	0x80a66158000	L2D	39237395991422
40	LD	0xaaaadaa919e0	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96de000	1	0x81ee30c2000	L2D	39237395991691
40	LD	0xaaaadaa91950	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L1D-ACCE	325	347	0xfffea96e0000	1	0x81eed278000	L2D	39237395991811

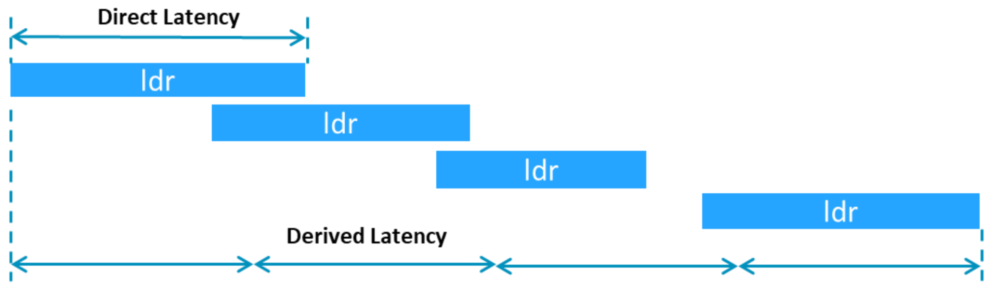
4.2.3. SPE latency data analysis

We can derive the memory latency value using SPE profile in two different ways:

– Direct latency reported within SPE recorded micro-architecture data as discussed in Section 4 which provides the total latency, issue latency, translation latency and execution latency.

– Derived latency that can be derived from the SPE recorded data for the benchmark. We define it as total execution time of the function divided by the total load instructions executed.

Figure 4.8 Direct and Derived Latencies



4.2.4. Derived latency calculation

To calculate of derived latency, we first calculate the total execution time of the function as follows:

Total execution time = Timestamp of last sampled load - Timestamp of first sampled load

The timestamps are obtained from SPE records. This time obtained is converted to nano seconds, using the conversion factor of 40 (derived from value in `CNTFRQ_EL0`) for the platform under test.

Total execution time in nano seconds can be derived as:

*Total execution time in nano seconds = Total execution time * 40*

Next, we need the total number of load instructions executed during this period. Because SPE is a statistically sampled record, we only profile one load in a sampling interval set by the SPE configuration register which is not the total number of loads executed. To derive the total instructions executed for this function, we use the total sampling interval for SPE and derive the memory reads based on the sampling rate.

The sampling interval configured for SPE can be obtained from the two system registers PMSIRR_EL1 and PMSIDR_EL1. For more information about these registers, see section D14.3.4 in the *Arm Architecture Reference Manual for A-profile architecture*. In practice, we can get the sampling interval using the following two steps:

01 Get the base sampling interval:

   a. If you specify an interval with `-c` to perf tool, the base sampling interval is the value that you input rounded by 256.
   b. If you do not specify with `-c` to perf tool, the default base sampling interval is defined by `/sys/devices/arm_spe_0/caps/min_interval`. Note that it is not recommended to use the minimum interval because it has the highest overhead.

02 Get the actual sampling interval:

   The actual sampling interval depends on the following two factors:

   - `jitter` option when running perf SPE
   - `ernd` value which is the mapping of PMSIDR_EL1.ERnd.

Table 4.3 Factors impacting actual sampling interval

jitter	/sys/devices/arm_spe_0/caps/ernd	Actual sampling interval
0		Base sampling interval + 1
1	0	Base sampling interval + 128
1	1	Base sampling interval + 1

For our test platform, we derive the total load instructions executed as follows:

Sampling interval = PMSIRR_EL1. INTERVAL * 256 + 128 = 1152

Total load instruction executed = Total loads sampled * Sampling interval

Finally, we have the derived latency as:

Derived latency = Total execution time in nano seconds / Total load instruction executed

4.3. Latency correlation between SPE data and LMbench results

Table 4.4 shows the computed results from SPE data, for both derived and direct latencies as well as the benchmark results from LMbench. The latency values from SPE are in cycles, while the latency values from the `lat_mem_rd` test are in nano seconds. To compare the data, we convert the cycles from the SPE to nanoseconds using the CPU frequency. For our 2.8 GHz platform, we divide the cycles by 2.8 to get nano seconds.

Table 4.4 Latency data from both LMbench and SPE

Test	Stride 4096	P1/L1D	P2/L2D	P3/LLC	P4/RAM
Lat_mem_rd	Benchmark reported latency (ns)	1.429	~5.0	~36.6	~101
	SPE derived latency (ns)	1.426	4.8	36.8	99.5
	Direct SPE: exec + xlate latency (ns)	1.429	7.86	68.9	N/A
	Direct SPE: total latency (ns)	34.6	123.9	905.0	Saturated
	Direct SPE: issue latency (ns)	33.2	116.1	836.1	Saturated
	Direct SPE: xlate latency (ns)	0.357	0.357	2.143	3.939

Test	Stride 4096	P1/L1D	P2/L2D	P3/LLC	P4/RAM
	Direct SPE: execution latency (ns) (total – issue – xlate)	1.071	7.500	66.8	N/A
Lat_mem_rd with "–t"	Benchmark reported latency (ns)	1.429	~5.7	~45	~135
	SPE derived latency (ns)	1.417	5.57	45.4	134.4
	Direct SPE: exec + xlate latency (ns)	1.429	8.4	77.5	N/A
	Direct SPE: total latency (ns)	34.6	141.4	1135	Saturated
	Direct SPE: issue latency (ns)	33.2	132.9	1057.1	Saturated
	Direct SPE: xlate latency (ns)	0.357	2.143	12.5	40.0
	Direct SPE: execution latency (ns) (total – issue – xlate)	1.071	6.250	66.1	N/A

We make the key observations from the correlation experiment with the help of the experiment results and data in Table 4.4, Table 4.5, and Table 4.6 for two test cases of LMbench `lat_mem_rd`. In the data presented in Table 4.4, Table 4.5, and Table 4.6, the yellow cells are the tests run without `-t` while grey cells are with the `-t` option enabled to stress the TLB.

4.3.1. Observation notes 1

From Table 4.4, the SPE derived latencies in both the test cases are very close to the result from `lat_mem_rd`, as expected. SPE measured total latency is very high with significant delay coming from the issue latency. However, derived execution latency with translation latency comes very close to the response latency.

This confirms that SPE can be a good profiling tool to estimate memory latency for workloads, even though the data is statistical in nature. For real world workloads, the exact data from SPE may not be the absolute number as SPE records data only at an interval and real-world code cannot be as repetitive as the LMbench test, but this can still be used to identify hot memory accesses and analyze relative memory latency measurements for the workload.

4.3.2. Observation notes 2

Another observation from Table 4.4 is that the latencies for P3 and P4 rise in the second test case with `-t` enabled. This is expected because the test causes TLB Refill for accesses that missed the virtual address translations in the TLB and may have caused table walks. As a result, the `xlate` latency for P3 and P4 rises for "`-t`" enabled case as shown in Table 4.4.

Table 4.5 shows the PMU events triggered during the load operations. As expected, for P3 and P4, TLB-REFILL events are triggered for the test with the `-t` option. PMU events triggered by an operation help to check for root causes while evaluating performance.

Table 4.5 PMU events logged by SPE

		Event statistics	
	-t enabled	Events	Count
P1 valid samples	No	RETIRED:L1D-ACCESS:TLB-ACCESS	94465
	Yes	RETIRED:L1D-ACCESS:TLB-ACCESS	256530
P2 valid samples	No	RETIRED:L1D-ACCESS:TLB-ACCESS	1
		RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS	94306
	Yes	RETIRED:L1D-ACCESS:TLB-ACCESS	2
		RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS	132329
P3 valid samples	No	RETIRED:L1D-ACCESS:TLB-ACCESS	2
		RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:TLB-REFILL:LLC-ACCESS:LLC-REFILL	101
		RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:LLC-ACCESS:LLC-REFILL	45254
	Yes	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:TLB-REFILL:LLC-ACCESS:LLC-REFILL	36886
P4 valid samples	No	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:TLB-REFILL:LLC-ACCESS:LLC-REFILL	1

		Event statistics	
	-t enabled	Events	Count
		RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:LLC-ACCESS:LLC-REFILL	7088
	Yes	RETIRED:L1D-ACCESS:L1D-REFILL:TLB-ACCESS:TLB-REFILL:LLC-ACCESS:LLC-REFILL	17057

4.3.3. Observation notes 3

Data source information is another key insight to derive from SPE. As shown in Figure 4.4, there are four plateaus for latency data that map to the four different data sources in the system cache hierarchy. Table 4.6 shows the corresponding data source for each plateau of the curve from SPE data. Yellow cells are the tests run without `-t` option while grey cells are with the `-t` option enabled to stress the TLB. Both show similar results. For P1 samples, all the data sources are from L1D. Similarly, for P2, P3, and P4, the samples report their target data sources as L2 cache, LLC, and RAM respectively.

Table 4.6 Data source information from SPE

	-t enabled	Total count	Data Source ratio
P1 valid samples	No	94465	L1D Hit 100%
	Yes	256530	L1D Hit 100%
P2 valid samples	No	94307	L2D Hit ~100%
	Yes	132331	L2D Hit 100%
P3 valid samples	No	45357	LLC Hit ~100%
	Yes	36886	LLC Hit 100%
P4 valid samples	No	7089	RAM Hit 100%
	Yes	17057	RAM Hit 100%

4.3.4. Summary

The memory latency analysis using SPE and correlation with LMbench shows that SPE can be used to estimate memory latencies, analyze the memory access patterns, and determine the root cause of the performance issues associated with memory accesses.

4.4. Memory read bandwidth test

To test memory bandwidth, we use the LMbench `bw_mem` test to conduct a memory bandwidth test while profiling the benchmark using SPE at the same time. For this test, we use the SPE to profile LMbench `bw_mem` to evaluate if the statistically inferred bandwidth data from SPE correlates with the LMbench measurements. Note that SPE is not accurate for memory bandwidth measurement because it is a statistical measurement using sampled operations. The purpose of this experiment is to demonstrate that SPE functionality can help with relative measurements while conducting optimization exercises and sensitivity studies. Memory bandwidth estimation using SPE as demonstrated here only applies to codes that are highly predictable and access patterns that are well known for micro-kernels.

LMbench `bw_mem` supports many memory bandwidth test modes including `rd`, `frd`, `wr`, `fwr`, and so on. For our experiments, we use the `frd` test which performs sequential memory reads for a given range with a 4-byte granularity and accumulates the results. For performance tests, the assumption is that the time taken by add operations will be negligible compared to the memory load operations. With that estimation, the memory read bandwidth can be calculated by total memory reads divided by the total execution time.

4.4.1. Step 1: Run LMbench bw_mem frd with SPE

The LMbench `bw_mem` test provides the following configuration options to measure memory bandwidth:

– `-P` specifies the number of parallel processes to stress memory.

– `-W` specifies whether the warmup process is needed.

– `-N` specifies the number of repetitions for the test.

– `Size` specifies the total memory size for the test.

– `Test mode` selects the required tests: `rd`, `wr`, `rdwr`, `cp`, `fwr`, `frd`, `fcp`, `bzero`, `bcopy`.

For example, to run the test with 10 processes and 128MB read size with `frd` for full read with no stride, use the following command:

```
$ ./bw_mem -P 10 128M frd
134.22 90728.36
```

The output from the LMbench tests run has two figures: `134.22` is the memory read size and `90728.36` is the total memory read bandwidth available.

For our bandwidth correlation experiment, we profile the `bw_mem` test using SPE with just one process (`-p 1`). To profile only memory reads with Arm SPE, we use its filtering mechanism to record only the load operations. We configure `arm_spe_0` by setting `branch_filter=0`, `load_filter=1` and `store_filter=0`.

For profiling, we run the following command.

```
$ perf record -C 40 -e
arm_spe_0/branch_filter=0,ts_enable=1,pct_enable=1,pa_enable=1,load_filter=1,jitter=1,store
_filter=0,min_latency=0/ -- taskset 0x10000000000 ./lmbench/bin/aarch64-linux-gnu/bw_mem -P
1 128M frd

134.22 9469.29
```

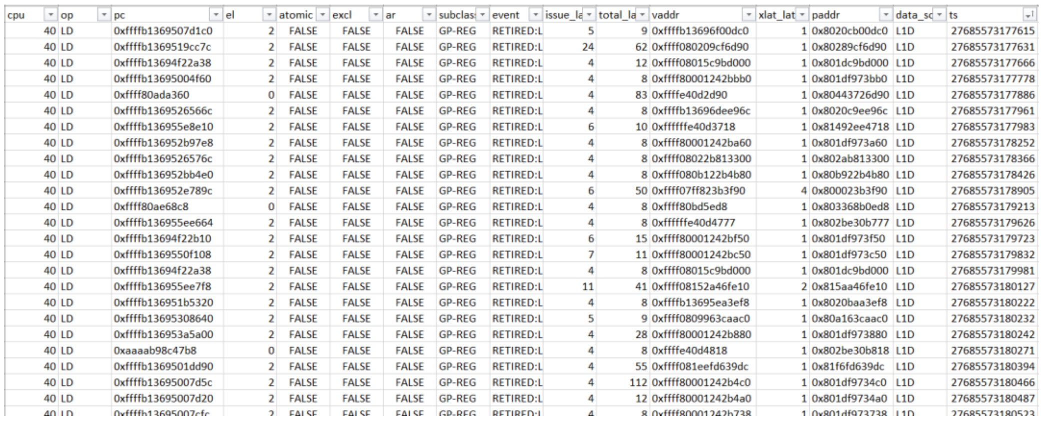
The test records bandwidth from the test which is 9469.29 MB/s.

We use the `spe-parser` tool introduced in SPE monitoring tools to process the SPE-profiled raw data collected using Linux perf tool. The following command provides a CSV output of the SPE-profiled raw `perf.data` from the Linux perf tool.

```
$ spe-parser perf.data -t csv
```

Figure 4.9 shows the processed SPE records from the SPE parser tool.

Figure 4.9 Processed SPE records for bw_mem



cpu	op	pc	el	atomic	excl	ar	subclas	event	issue_la	total_la	vaddr	xlat_lat	paddr	data_sc	ts
40	LD	0xffffb1369507d1c0	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	5	9	0xffffb13696f00dc0	1	0x8020cb00dc0	L1D	27685573177615
40	LD	0xffffb1369519cc7c	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	24	62	0xffff080209cf6d90	1	0x80289cf6d90	L1D	27685573177631
40	LD	0xffffb13694f22a38	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	12	0xffff08015c9bd000	1	0x801dc9bd000	L1D	27685573177666
40	LD	0xffffb13695004f60	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff80001242bbb0	1	0x801df973bb0	L1D	27685573177778
40	LD	0xffff80ada360	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	83	0xffffe40d2d90	1	0x80443726d90	L1D	27685573177886
40	LD	0xffffb1369526566c	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffffb13696dee96c	1	0x8020c9ee96c	L1D	27685573177961
40	LD	0xffffb136955e8e10	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	6	10	0xffffffe40d3718	1	0x81492ee4718	L1D	27685573177983
40	LD	0xffffb136952b97e8	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff80001242ba60	1	0x801df973a60	L1D	27685573178252
40	LD	0xffffb1369526576c	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff08022b813300	1	0x802ab813300	L1D	27685573178366
40	LD	0xffffb136952bb4e0	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff080b122b4b80	1	0x80b922b4b80	L1D	27685573178426
40	LD	0xffffb136952e789c	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	6	50	0xffff07ff823b3f90	4	0x800023b3f90	L1D	27685573178905
40	LD	0xffff80ae68c8	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff80bd5ed8	1	0x803368b0ed8	L1D	27685573179213
40	LD	0xffffb136955ee664	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffffffe40d4777	1	0x802be30b777	L1D	27685573179626
40	LD	0xffffb13694f22b10	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	6	15	0xffff80001242bf50	1	0x801df973f50	L1D	27685573179723
40	LD	0xffffb1369550f108	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	7	11	0xffff80001242bc50	1	0x801df973c50	L1D	27685573179832
40	LD	0xffffb13694f22a38	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff08015c9bd000	1	0x801dc9bd000	L1D	27685573179981
40	LD	0xffffb136955ee7f8	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	11	41	0xffff08152a46fe10	2	0x815aa46fe10	L1D	27685573180127
40	LD	0xffffb136951b5320	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffffb13695ea3ef8	1	0x8020baa3ef8	L1D	27685573180222
40	LD	0xffffb13695308640	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	5	9	0xffff0809963caac0	1	0x80a163caac0	L1D	27685573180232
40	LD	0xffffb136953a5a00	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	28	0xffff80001242b880	1	0x801df973880	L1D	27685573180242
40	LD	0xaaaab98c47b8	0	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffffe40d4818	1	0x802be30b818	L1D	27685573180271
40	LD	0xffffb1369501dd90	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	55	0xffff081eefd639dc	1	0x81f6fd639dc	L1D	27685573180394
40	LD	0xffffb13695007d5c	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	112	0xffff80001242b4c0	1	0x801df9734c0	L1D	27685573180466
40	LD	0xffffb13695007d20	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	12	0xffff80001242b4a0	1	0x801df9734a0	L1D	27685573180487
40	LD	0xffffb13695007cfc	2	FALSE	FALSE	FALSE	GP-REG	RETIRED:L	4	8	0xffff80001242b738	1	0x801df973738	L1D	27685573180523

As explained in Latency correlation between SPE data and lmbench results, SPE packets can be filtered to extract the hot loop of the code for further analysis. By inspecting the `bw_mem` source code, we identify that the `frd()` function is the benchmark portion of the code. We used the assembler inspection tool objdump, to locate the PC of this function with offset `1fbc~23b0` as shown in Figure 4.10.

Figure 4.10 LMbench bw_mem frd assembly code

```
0000000000001f84 <frd>:
    1f84:       a9bf7bfd        stp     x29, x30, [sp, #-16]!
    1f88:       910003fd        mov     x29, sp
    1f8c:       f9401823        ldr     x3, [x1, #48]
    1f90:       b40021e0        cbz     x0, 23cc <frd+0x448>
    1f94:       d1000404        sub     x4, x0, #0x1
    1f98:       f9400c25        ldr     x5, [x1, #24]
    1f9c:       52800000        mov     w0, #0x0                        // #0
    1fa0:       14000004        b       1fb0 <frd+0x2c>
    1fa4:       d1000484        sub     x4, x4, #0x1
    1fa8:       b100049f        cmn     x4, #0x1
    1fac:       54002120        b.eq    23d0 <frd+0x44c>  // b.none
    1fb0:       eb05007f        cmp     x3, x5
    1fb4:       54ffff83        b.cc    1fa4 <frd+0x20>  // b.lo, b.ul, b.last
    1fb8:       aa0503e2        mov     x2, x5
    1fbc:       b9400041        ldr     w1, [x2]
    1fc0:       b9400446        ldr     w6, [x2, #4]
    1fc4:       0b060021        add     w1, w1, w6
    1fc8:       b9400846        ldr     w6, [x2, #8]
    1fcc:       0b060021        add     w1, w1, w6
    1fd0:       b9400c46        ldr     w6, [x2, #12]
    1fd4:       0b060021        add     w1, w1, w6
    1fd8:       b9401046        ldr     w6, [x2, #16]
    1fdc:       0b060021        add     w1, w1, w6
    1fe0:       b9401446        ldr     w6, [x2, #20]

    <snip>

    23a0:       b941f446        ldr     w6, [x2, #500]
    23a4:       0b060021        add     w1, w1, w6
    23a8:       b941f846        ldr     w6, [x2, #504]
    23ac:       0b060021        add     w1, w1, w6
    23b0:       b941fc46        ldr     w6, [x2, #508]
    23b4:       0b060021        add     w1, w1, w6
    23b8:       0b010000        add     w0, w0, w1
    23bc:       91080042        add     x2, x2, #0x200
    23c0:       eb02007f        cmp     x3, x2
    23c4:       54ffdfc2        b.cs    1fbc <frd+0x38>  // b.hs, b.nlast
    23c8:       17fffef7        b       1fa4 <frd+0x20>
    23cc:       52800000        mov     w0, #0x0                        // #0
    23d0:       9400082a        bl      4478 <use_int>
    23d4:       a8c17bfd        ldp     x29, x30, [sp], #16
    23d8:       d65f03c0        ret
```

4.4.2. Estimating bandwidth from SPE data

We estimate the memory read bandwidth from the SPE samples that are filtered for memory reads of the benchmark.

To calculate the estimated bandwidth, we need to first calculate the total memory size read by the benchmark and the total execution time of the benchmark. This lets us derive the estimated memory bandwidth as the SPE estimated total memory read divided by total execution time.

To calculate the total memory reads, we use the total sampling interval for SPE and derive the memory reads based on this sampling rate. The sampling interval configured for SPE can be obtained from the method mentioned in Derived latency calculation.

To calculate the total execution time, we use timestamp counter (`tsc`) value of the system from the beginning and end of the benchmark execution. This time, obtained as `tsc` from the system, is converted to nano seconds using the conversion factor of 40 (derived from value in `CNTFRQ_EL0`) for the platform under test.

We derive the estimated memory bandwidth from SPE data statistically for the LMbench `bw_mem` test as follows:

– Estimated Memory Read Size
    - Total Samples: 377965
    - Data size per load: 4
    - Sampling interval: 1024 + 128 = 1152
    - Estimated memory read size: 377965 * 1152 * 4 = 1,741,662,720 bytes
– Total Time Spent
    - Delta between start and stop system tsc
      27685591816027 – 27685587232500 = 4583527
    - System tsc to time (conversion factor 40 is derived from CNTFRQ_EL0)
      4583527 * 40 = 183,341,080 nano seconds
    - Total time spent = 0. 183341080 second
– Bandwidth

  1,741,662,720 / 0.183341080 = 9499.58 MB/s

4.4.3. Bandwidth comparison between SPE estimation and LMbench

Table 4.7 compares the estimated memory read bandwidth measurements using SPE data from Estimating bandwidth from SPE data and memory bandwidth reported by LMbench `bw_mem` from Step 1: Run lmbench bw_mem frd with SPE.

Table 4.7 Memory bandwidth correlation for LMbench bw_mem

	LMbench report	SPE estimation
Memory Read Bandwidth (MB/s)	9469.29	9499.58

It is observed that estimated bandwidth is quite comparable for this benchmark. Note that SPE should not be used for direct memory BW measurements as it is statistical approximation. However, this can be used for relative measurements of memory BW usage on deterministic workloads during optimization exercises, especially for tight loops and deterministic access patterns.

4.5. Summary

We have demonstrated how SPE-based profiling can provide valuable insights into memory access performance on a platform. We shared the different data points that can be extracted from SPE record of memory access points and how memory latency and memory bandwidth estimations can be derived statistically to aid workload characterization, hotspot analysis, and root cause analysis. It is important to note that SPE estimations are statistical in nature and are not precise measurements for a workload. On the other hand, for highly deterministic code such as LMbench microbenchmarks, we demonstrate how the SPE interpolated data comes very close to the benchmark score. The general idea for metric estimation is to obtain records only for operations sampled in a configurable interval which can be interpolated for the overall program execution. Even though these measurements are imprecise, they can help in sensitivity studies and design space explorations.

5. Using SPE for data sharing analysis

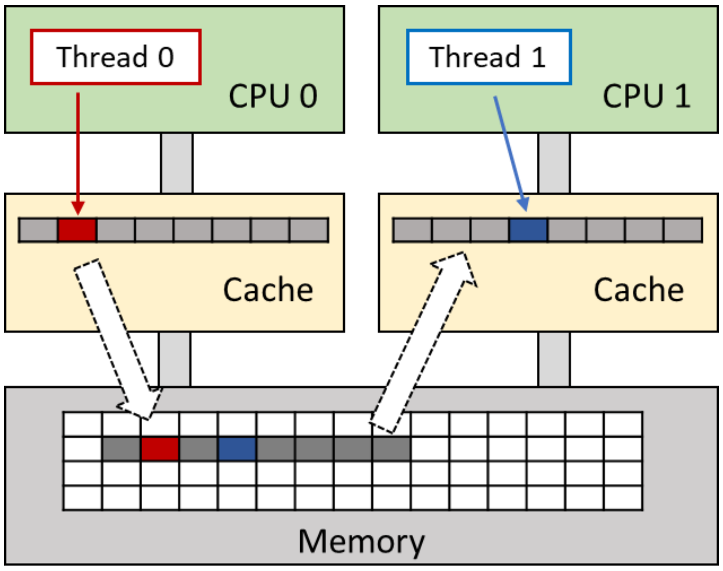
SPE profiling provides key insights into the execution of memory operations that can be used to identify data sharing issues in multi-threaded workloads.

This case study demonstrates how SPE can be used for data sharing analysis to help improve performance issues caused by multiple threads working on the same data set, which can cause latency issues due to the overhead of cache coherency mechanisms. We specifically demonstrate how SPE data is used by the Linux perf c2c tool to locate false sharing issues.

False sharing is one of the common performance issues in a multi-processor system. It occurs when one processor modifies data items on the same cache line used by another processor which is working with a different part of the same cache line. This means that a cache line modification by one CPU invalidates and updates the other CPU's cache copy, even though it is accessing a different part of the cache line and does not care about the updated portion of the cache line.

In Figure 5.1, threads 0 and 1 run on different NUMA nodes. Both threads load the same memory block into their cache line, even though they are consuming different data elements (indicated by the red and blue elements) in this cache line. False sharing occurs for example, when thread 0 updates its data in red which causes thread 1 to update its cache line too, even though it never references the data in red. This cache invalidation is costly and can impact the performance of the whole system if there are many cases of threads sharing cache lines. This performance cost is worse when the cores that share the cache lines are on different NUMA nodes.

Figure 5.1 False sharing of caches between multiple threads



5.1. Perf c2c introduction

To detect false sharing scenarios in a multithreaded application, the Linux perf tool provides the c2c feature, which stands for cache2cache. On an Arm machine, this perf tool feature is implemented based on analyzing the memory access data from SPE, particularly the data source information, as well as the data address and instruction PC address captured in the SPE packet introduced in Using SPE for memory access analysis. Perf c2c is enabled for Arm machines on Linux kernels newer than v6.0. Refer to the Using the Arm Statistical Profiling Extension to detect false cache-line sharing blog [8] by Linaro for more information about the support for perf c2c for Arm machines.

The perf c2c tool reports the following information that can help identify false sharing cases:

- The cache line addresses that may have false sharing issues.
- The data offset in a cache line that is accessed by different processes.
- The instruction addresses that access the cache lines.

- The local or remote (cross socket) access information.
- The NUMA nodes that are involved.

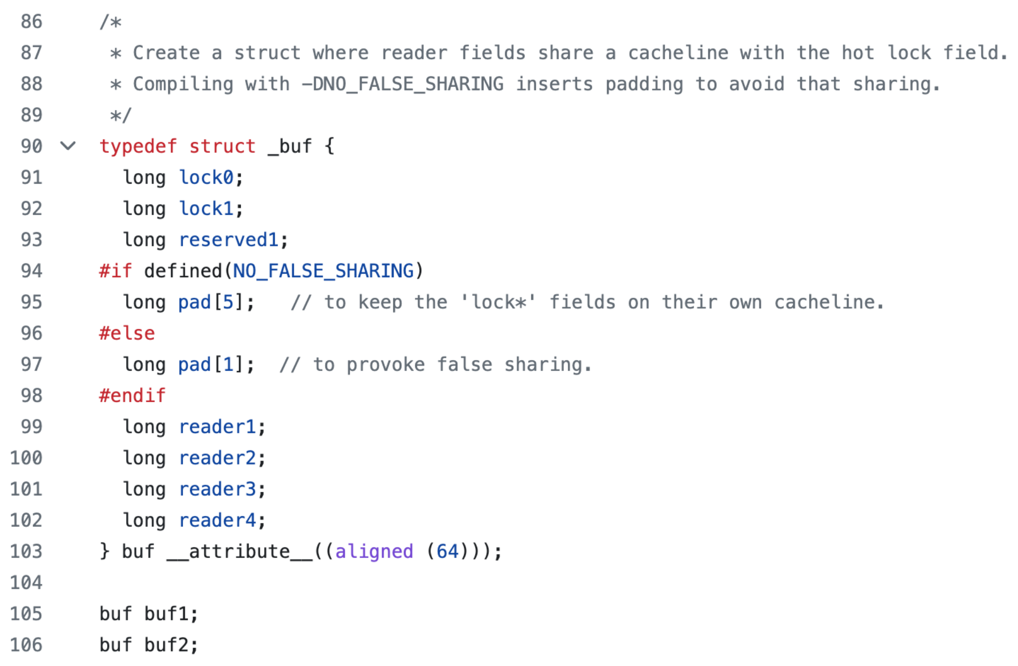
5.2. False sharing analysis using perf c2c

To demonstrate how to use perf c2c for false sharing analysis on an Aarch64 based platform enabled by SPE, we use the example code[9] from the github repository: https://github.com/joemario/perf-c2c-usage-files/blob/master/false_sharing_example.c

5.2.1. Code introduction

The code in `false_sharing_example.c` creates several threads based on user input on different NUMA nodes if there is more than one of them. All these threads access the same data structure `buf` defined as shown in Figure 5.2. There is a macro named `NO_FALSE_SHARING` checked in the `buf` definition. When `NO_FALSE_SHARING` is undefined, the `pad` field is the size of 1 `long` (8 bytes on Arm64), forcing the lock and reader fields in the same cache line. However, when the `NO_FALSE_SHARING` macro is defined, the `pad` is the size of 5 `long`s which stores the `lock` and `reader` fields in different cache lines.

Figure 5.2 buf structure definition in false_sharing_example.c

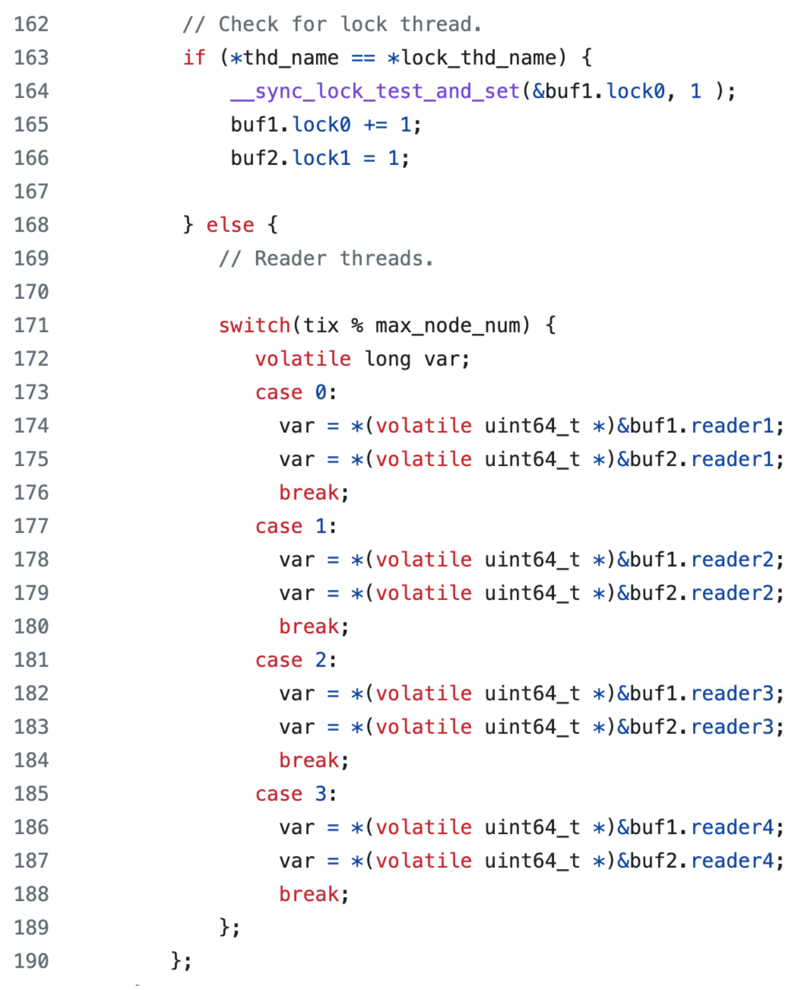


```
/*
 * Create a struct where reader fields share a cacheline with the hot lock field.
 * Compiling with -DNO_FALSE_SHARING inserts padding to avoid that sharing.
 */
typedef struct _buf {
  long lock0;
  long lock1;
  long reserved1;
#if defined(NO_FALSE_SHARING)
  long pad[5];   // to keep the 'lock*' fields on their own cacheline.
#else
  long pad[1];  // to provoke false sharing.
#endif
  long reader1;
  long reader2;
  long reader3;
  long reader4;
} buf __attribute__((aligned (64)));

buf buf1;
buf buf2;
```

The threads in the code are divided into two groups: lock threads and reader threads. All the threads run the same function named `read_write_func()`. Figure 5.3 shows the key logic of the function which shows the different path threads can take if they are lock threads or reader threads. Lock threads are set the lock0/lock1 fields in the structure while reader threads reading the reader1/reader2/reader3/reader4 fields.

Figure 5.3 Code logic of read_write_func() in false_sharing_example.c



```
 // Check for lock thread.
 if (*thd_name == *lock_thd_name) {
     __sync_lock_test_and_set(&buf1.lock0, 1 );
     buf1.lock0 += 1;
     buf2.lock1 = 1;

 } else {
    // Reader threads.

    switch(tix % max_node_num) {
       volatile long var;
       case 0:
         var = *(volatile uint64_t *)&buf1.reader1;
         var = *(volatile uint64_t *)&buf2.reader1;
         break;
       case 1:
         var = *(volatile uint64_t *)&buf1.reader2;
         var = *(volatile uint64_t *)&buf2.reader2;
         break;
       case 2:
         var = *(volatile uint64_t *)&buf1.reader3;
         var = *(volatile uint64_t *)&buf2.reader3;
         break;
       case 3:
         var = *(volatile uint64_t *)&buf1.reader4;
         var = *(volatile uint64_t *)&buf2.reader4;
         break;
    };
};
```

When `false_sharing_example.c` is compiled with `NO_FALSE_SHARING` undefined, the false sharing scenario occurs between the threads accessing different data elements in the same cache line. It can be detected using the perf c2c tool, as shown in Run the test.

5.2.2. Run the test

To run the program for a suitable amount of time for the experiment, we modified the original `false_sharing_example.c` slightly to run more iterations as follows:

```
$ git diff ./false_sharing_example.c

       diff --git a/false sharing example.c b/false_sharing_example.c
       index 900f1ee..5eaa706 100644
       --- a/false sharing example.c
       +++ b/false_sharing_example.c
       @@ -55,7 +55,7 @@
        /*
         * A thread on each numa node seems to provoke cache misses
         */
       -#define        LOOP_CNT        (5 * 1024 * 1024)
       +#define        LOOP_CNT        (5 * 1024 * 1024 * 100)
```

The next step is to compile the code without `NO_FALSE_SHARING` defined and create the false sharing scenario.

We now run the test and record the execution time for performance evaluation. As our machine only has one NUMA node, parameter 1 passed to the executable, it results in running one reader thread and 1 lock thread on different cores.

```
$ time ./false_sharing 1

       348 mticks, reader_thd (thread 1), on node 0 (cpu 8).
       467 mticks, lock_th (thread 0), on node 0 (cpu 7).

       real    0m18.684s
       user    0m32.617s
       sys     0m0.004s
```

Next, we profile the `false_sharing` program with perf c2c:

```
$ perf c2c record -- ./false_sharing 1
$ perf c2c report --stdio
```

5.2.3. Perf c2c output

We use the Linux perf report utility to view the perf c2c analysis, which provides three output sections. We first discuss how to consume the perf c2c output and then take a closer look at the source code against the perf c2c outputs for false sharing detection.

5.2.4. Perf c2c output section 1

Section 1 in Figure 5.4 shows the perf c2c execution details. The perf c2c output shows the `arm_spe_0` profiling conducted for this analysis with the command including the following filters:

```
armm_spe_0/ts_enable=1,pa_enable=1,load_filter=1,store_filter=1,min_lat
ency=30/.
```

The perf c2c tool relies on the peer snoop indication from the SPE data source information to detect data sharing between threads.

Figure 5.4 perf c2c report: section 1

```
=================================================
                 c2c details
=================================================
  Events         :arm_spe_0/ts_enable=1,pa_enable=1,load_filter=1,store_filter=1,min_latency=30/
                 : dummy:u
                 : memory
  Cachelines sort on        : Peer Snoop
  Cacheline data grouping   : offset,iaddr
```

5.2.5. Perf c2c output section 2

Section 2 in Figure 5.5 shows the shared data cache line table which lists all the cache lines for which SNOOP (possible false sharing) is detected. The cache lines are ranked by the Peer Snoop percentage value.

Figure 5.5 perf c2c report: section 2

```
=================================================
            Shared Data Cache Line Table
=================================================
#
#         ----------- Cacheline ----------      Peer  ------- Load Peer -------    Total     Total     Total
# Index               Address  Node  PA cnt     Snoop    Total     Local   Remote  records     Loads    Stores
# .....  ..................  ....  ......   .......  .......  .......  .......  .......  .......  .......
#
      0      0xaaaab26320c0     0    8440    52.79%   169885   169885        0   504120   254956   249164
      1      0xaaaab2632080     0   16434    47.19%   151852   151852        0   699534   563014   136520


--------- Stores --------  ----- Core Load Hit -----  - LLC Load Hit --  - RMT Load Hit --  --- Load Dram ----
  L1Hit   L1Miss      N/A       FB       L1       L2   LclHit  LclHitm   RmtHit  RmtHitm      Lcl       Rmt
.......  .......  .......  .......  .......  .......  ........  .......  ........  .......  ........  ........
      0        0   249164        0    85071        0   169885        0        0        0        0         0
      0        0   136520        0   411162        0   151852        0        0        0        0         0
```

5.2.6. Perf c2c output section 3

Section 3 in Figure 5.6 shows the Shared Cache Line Distribution Pareto. This provides detailed information for each cache line that has detected SNOOP. For cache lines like 0xaaaab26320c0 which has the highest snoop hit percentage, it gives offsets in this cache line that are accessed by different threads. The code addresses of the threads are also provided that can be mapped to the source code for inspection.

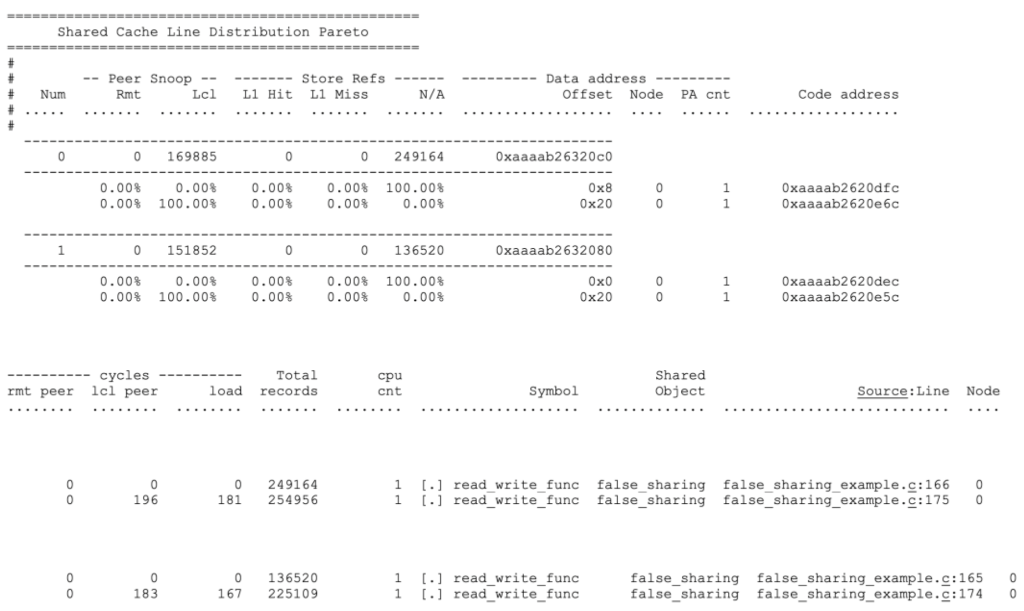
Peer Snoop is divided into two types:

- `Rmt` stands for remote which means there are snoop hits across different CPU sockets.
- `Lcl` stands for local cache snoop in the same CPU socket.

Data Addresses provide further details including the following:

- `Node` refers to the CPU sockets. Since our test platform only has one CPU socket, we only see local SNOOP and node 0 here.
- `Offset` shows the data offset accessed within the cache line.

Figure 5.6 perf c2c report: section 3



```
=================================================
      Shared Cache Line Distribution Pareto
=================================================
#
#        -- Peer Snoop --  ------- Store Refs ------  --------- Data address ---------
#   Num      Rmt      Lcl   L1 Hit  L1 Miss      N/A             Offset  Node  PA cnt        Code address
# .....  .......  .......  .......  .......  .......  .................  ....  ......  .................
#
  ----------------------------------------------------------------------
      0        0   169885        0        0   249164      0xaaaab26320c0
  ----------------------------------------------------------------------
           0.00%    0.00%    0.00%    0.00%  100.00%                0x8     0       1      0xaaaab2620dfc
           0.00%  100.00%    0.00%    0.00%    0.00%               0x20     0       1      0xaaaab2620e6c

  ----------------------------------------------------------------------
      1        0   151852        0        0   136520      0xaaaab2632080
  ----------------------------------------------------------------------
           0.00%    0.00%    0.00%    0.00%  100.00%                0x0     0       1      0xaaaab2620dec
           0.00%  100.00%    0.00%    0.00%    0.00%               0x20     0       1      0xaaaab2620e5c


---------- cycles ----------     Total       cpu                           Shared
rmt peer  lcl peer      load   records       cnt               Symbol        Object                 Source:Line  Node
........  ........  ........   .......  ........  ...................  .............  ..........................  ....


       0         0         0    249164         1  [.] read_write_func  false_sharing  false_sharing_example.c:166   0
       0       196       181    254956         1  [.] read_write_func  false_sharing  false_sharing_example.c:175   0


       0         0         0    136520         1  [.] read_write_func      false_sharing  false_sharing_example.c:165   0
       0       183       167    225109         1  [.] read_write_func      false_sharing  false_sharing_example.c:174   0
```

5.2.7. Code analysis with perf c2c output

From the "Shared Cache Line Distribution Pareto" output in section 3, the cache line address 0xaaaab26320c0, points to line 175 in `false_sharing_example.c` as being code that triggered lots of local peer snoop. This is an indication of possible cache line false sharing which can be checked by the cache line offsets being accessed for confirmation. A similar observation can be made for cache line address 0xaaaab2632080.

As shown in Figure 5.6, line 175 in the source file `false_sharing_example.c` reads the `reader1` field of `buf2` while line 166 sets the `lock1` field of `buf2`. As described previously, the two lines of code are executed by different threads on different cores, which are accessing different fields in the same cache line. This is a false sharing case, and perf c2c perfectly points this out with the detailed cache access data.

5.2.8. Fix and re-run

To fix this cache false sharing case, the simplest way is to ensure that the lock and reader fields are in different cache lines. As shown in the definition of `buf` structure in, Figure 5.2 if the `NO_FALSE_SHARING` macro is defined, it adds 5 `longs`, a total of 40 bytes, as padding to the field which stores the reader fields in different cache lines because the cache line size is normally 64 bytes.

To test this, we recompile the source code with the macro `NO_FALSE_SHARING` defined. For this test, the execution time reduces by about three times on our machine compared to the first `false_sharing` binary, as shown in Figure 5.7. We use perf c2c to profile this `no_false_sharing` program, which shows no Peer Snoop detection for the false`_sharing_example.c`.

Figure 5.7 Execution time of no false sharing test

```
$ time ./no_false_sharing 1
       52 mticks, reader_thd (thread 1), on node 0 (cpu 8).
       178 mticks, lock_th (thread 0), on node 0 (cpu 7).

       real    0m7.126s
       user    0m9.225s
       sys     0m0.000s
```

5.3. Summary

We demonstrated how the perf c2c tool enabled by SPE supports data sharing analysis on Arm platforms that implement SPE. The example case showed how this analysis can be used to solve a false sharing performance issue in multithreaded code.

Appendix A. Glossary

Term	Meaning
LLC	Last Level Cache
LSU	Load Store Unit
MMU	Memory /management Unit
PE	Processing Element
PMU	Performance Monitoring Unit
SIMD	Single Instruction Multiple Data
SiP	Silicon Provider
SPE	Statistical Performance Extension
SPU	Statistical Profiling Unit
TLB	Translation Lookaside Buffer

Appendix B. References

- [1] Arm®, "Arm® Neoverse™ V1 Core: Performance Analysis Methodology white paper", https://developer.arm.com/documentation/109199/0100/?lang=en
- [2] Arm®, "Arm® Architecture Reference Manual, for A-profile architecture documentation", https://developer.arm.com/docs/ddi0487/latest
- [3] Linux perf tool wiki, https://perf.wiki.kernel.org/index.php/Main_Page
- [4] Arm, SPE parser tool in the Arm Telemetry solution repository, https://gitlab.arm.com/telemetry-solution/telemetry-solution/-/tree/main/tools/spe_parser
- [5] Arm Telemetry Solution Repository, https://gitlab.arm.com/telemetry-solution/telemetry-solution
- [6] Apache Arrow CSV Writer, https://github.com/apache/arrow/pull/13394
- [7] LMbench, https://lmbench.sourceforge.net
- [8] Linaro, "Using the Arm Statistical Profiling Extension to detect false cache-line sharing", https://www.linaro.org/blog/using-the-arm-statistical-profiling-extension-to-detect-false-cache-line-sharing/
- [9] Joe Mario, "False sharing Example Code", https://github.com/joemario/perf-c2c-usage-files/blob/master/false_sharing_example.c