ARM PMU Whitepaper

As part of Learning when reading the ARM Whitepaper on “Arm Neoverse N1 Core: Performance Analysis Methodology”. I made few points as part of understanding listing them down.

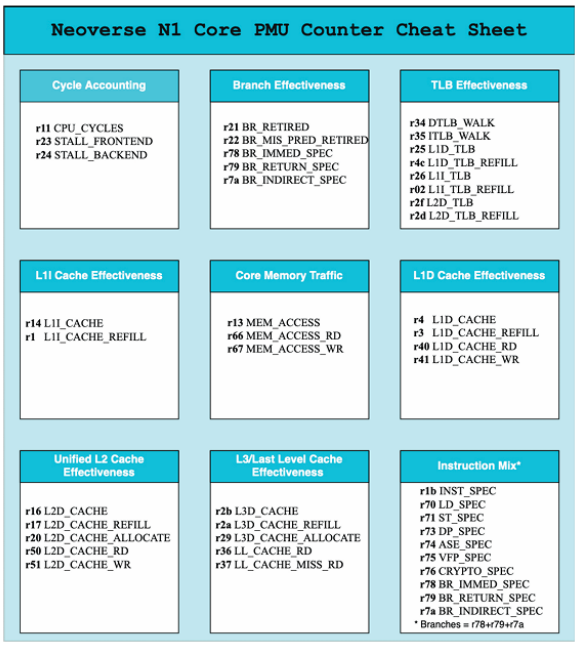
Links :

1. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-n1-core-performance-v2.pdf
2. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-v1-core-performance-analysis.pdf

The PMU Metrics for CPU :

Neoverse N1 Core PMU Events Cheat Sheet for Workload Characterization

Though the Neoverse N1 core supports 100+ hardware counters, not all are needed for an initial characterization of the workload execution. Figure 2 is a cheat sheet of major Performance Monitoring Events for a first pass workload characterization exercise on a Neoverse N1 CPU.

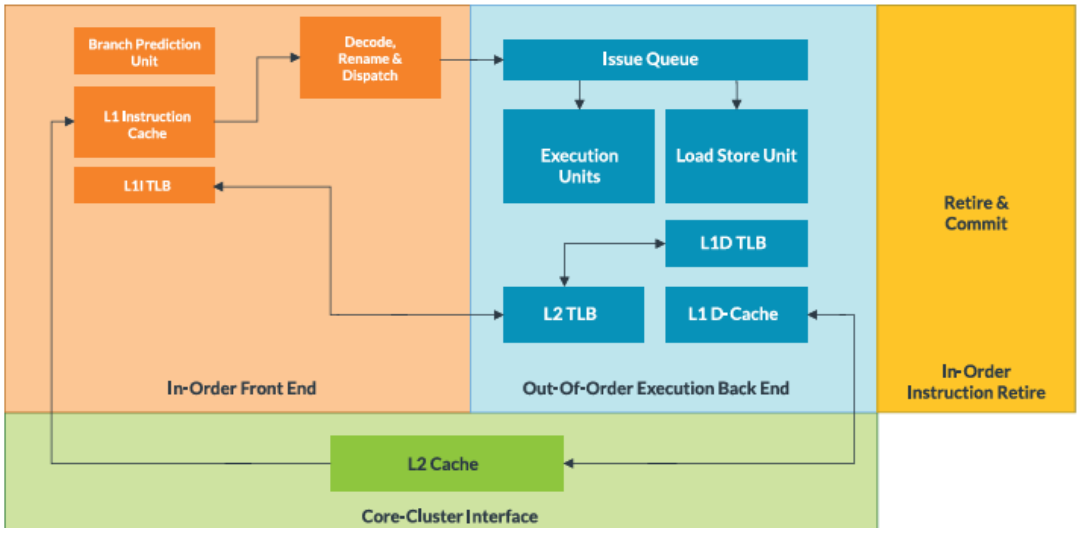


The Neoverse CPU Architecture :

Neoverse N1 Core

The Neoverse N1 core is an out-of-order super scalar machine which can dispatch/retire up to 8 instructions per cycle[4]. A super scalar processor has three major phases in its pipeline: in-order fetch and decode of the instruction stream, out of order execution of operations, and a final in-order commit/retirement of the instruction. The in-order fetch/decode part is called the Front End and the out of order execution part is called the Back End. The CPU also has a memory subsystem that is responsible for all the memory operations and their ordered execution. Design details of all these major CPU blocks vary across micro-architectures.

In this document, we will describe the blocks that are common in most super-scalar architectures and highlight the performance metrics associated as they apply to the N1 implementation. For detailed explanations for the Neoverse N1 micro-architecture, please refer to the Neoverse N1 Technical Reference Manual[2] and Neoverse Software Optimization Manual[5]. Figure 3 below shows the high level block diagram of the Neoverse N1 CPU.



Front End

The Front End of the CPU is an in-order pipeline that handles fetching instructions from the I-Cache, decoding those instructions, and queuing them for the execution engine in the Back End. The architectural instructions can get broken down into micro-operations in the decode stage. These micro-operations are queued and dispatched to the execution engines according to their availability. Apart from the fetch, decode and dispatch units, there is a rename block that keeps track of the operational dataflow and dependencies to make sure that the executed operations are committed in-order. Another important unit in the Front End is the Branch Predictor. This unit predicts both the direction of branches as well as target addresses for indirect branches. Note that an out of order processor can fetch multiple instructions in advance to fill the pipeline and execute them speculatively. Branch prediction techniques help with fetching the instructions in the right program order as much as possible, as branch mis-predictions lead to pipeline flushes and wasted cycles.

Neoverse N1 can fetch up to 4 instructions per cycle and dispatch a maximum 8 micro-operations per cycle.

Back End

The Back End of the CPU handles the execution of the micro-operations which are dispatched to the relevant execution units for processing. Neoverse N1 supports multiple execution units including the Branch unit, Load/Store unit and Arithmetic units including the advanced vector engines. The number of execution units and cycles taken for the execution of an instruction can vary per micro-architecture; therefore, instruction execution latency and throughput are implementation dependent. Once instruction execution is complete, the results are stored and committed in-order when dependencies are resolved.

Neoverse N1 has 4 integer execution units, 2 Floating point/SIMD pipelines and 2 load/store pipelines, which allows up to 8 micro-operations to be dispatched into the execution pipeline every cycle.

Memory Subsystem

The Memory subsystem of the CPU handles the execution of load and store operations which relies heavily on the memory hierarchy levels. Neoverse N1 has a dedicated L1/L2 cache per core, where the L2 cache is shared between the L1 Data cache and the L1 Instruction cache. The Load Store Unit controls the data flow between the caches and to memory.

Neoverse N1 has two load/store units, which can both handle read and write operations. The L1 Data Cache is a 64kB 4-way set associative design and L2 Cache is an 8-way set associative cache with up to 1 MB in size which is configurable per implementation. The private L2 cache of the core connects to the rest of the system via an AMBA 5 CHI interface.

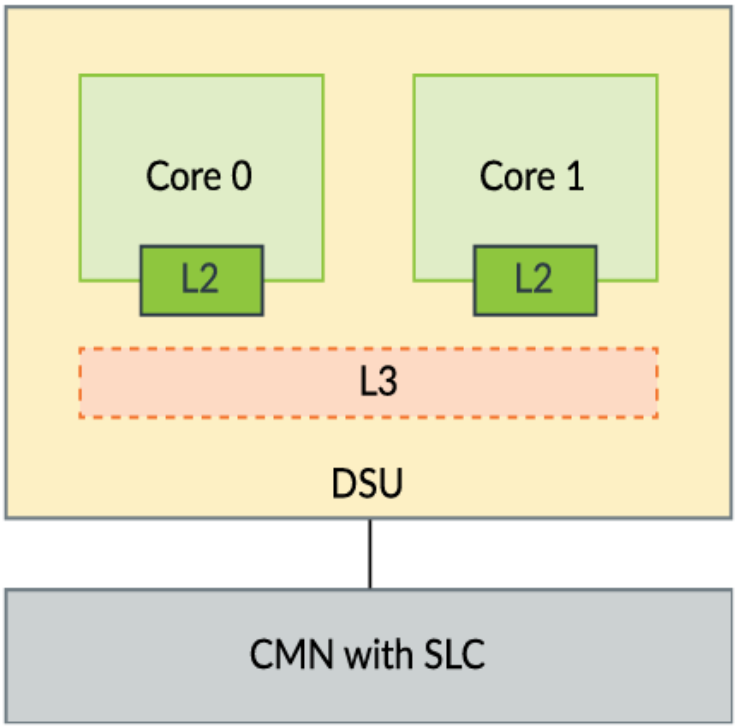


Figure 4: Neoverse N1 DSU Cluster System with Optional L3

An alternate configuration is a direct connect system as in figure 5, where as the name suggests, the cores are directly connected to the Coherent Mesh Interconnect interface called CAL. These systems don't support an L3 Cache as there is no DSU cluster present.

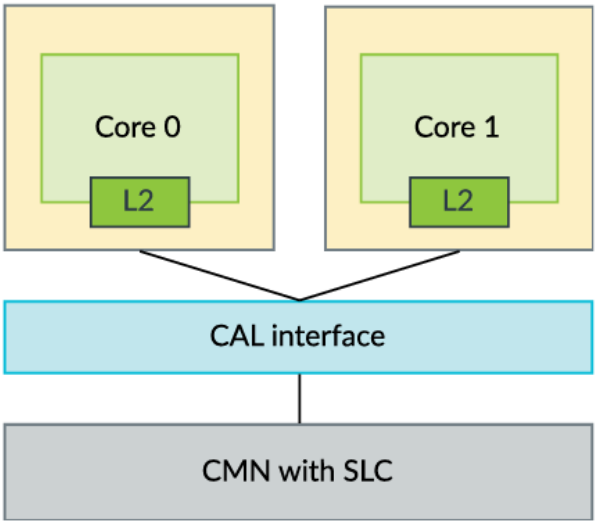


Figure 5: Neoverse N1 System with Direct Connect Configuration

Performance Analysis Methodology

For workload analysis, both raw hardware events as well as some useful metrics derived from them can be used for characterization. Understanding all the events and deciding which events to use is a non-trivial task as each workload has unique behaviors and potential bottlenecks. Moreover, to pin-point to a specific hardware problem, one may need to have in-depth knowledge of the micro-architecture. To make this process easier, we will highlight some of the raw events and derived metrics that can help conduct an initial characterization of the workload. Following this process should help with formulating the top-level characteristics of a workload and root-cause a performance issue later in the deep dive process.

A basic top-level analysis methodology that can be followed starts with an accounting of the cycle usage for execution as in Figure 6.

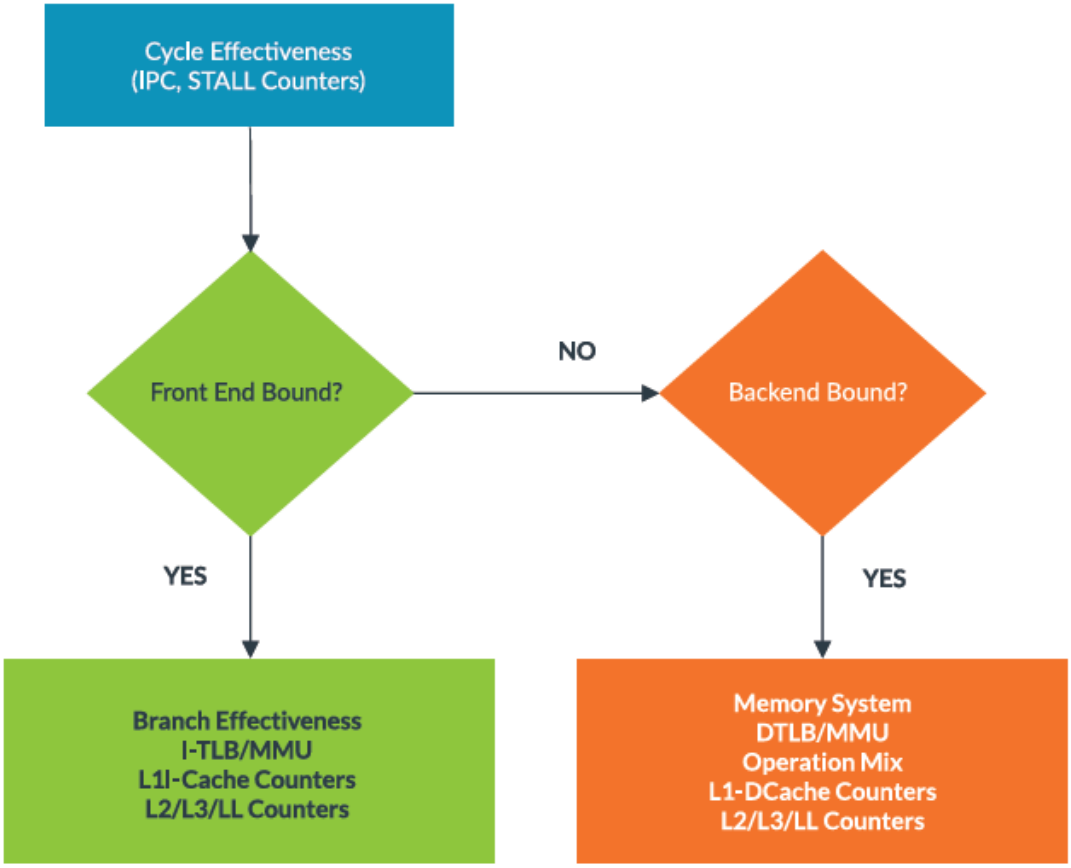


Figure 6: Cycle Effectiveness Evaluation Methodology

bottlenecks as demonstrated in Figure 6. Below is a list of CPU blocks/units to analyze for further decomposition of a Front End Bound workload:

- ITLB events
- I-Cache Events: L1I + L2/Last level Cache events
- Branch Effectiveness events

Below is a list of CPU blocks/units to analyze for further decomposition of a Back End Bound workload:

- DTLB events
- Memory System related events
- D-Cache Counters: L1D + L2/Last level Cache events
- Instruction Mix

Two performance metrics that can be derived for a high level evaluation of the branch execution performance with respect to the overall program execution are:

```
Branch MPKI = BR_MIS_PRED_RETIRED/INST_RETIRED * 1000
Branch Mis-prediction Rate = BR_MIS_PRED_RETIRED/BR_RETIRED
```

Branch Mix/Prediction Performance

Branch prediction units work differently depending on the branch type. There are three main components :

- Branch History Table (BHT) that stores the history of conditional branches, taken or not.
- Branch Target Buffer (BTB) that stores the target address for indirect branches
- Return Address Stack (RAS) that stores the function return branches.

Neoverse N1 supports three events, BR_IMMED_SPEC, BR_RETURN_SPEC and BR_INDIRECT_SPEC, to categorize immediate, indirect and return branches executed respectively. Getting a break down of the branch type helps to deep dive into the performance of each of these sub blocks within the branch prediction unit. Note that these events count both correctly predicted and mis-predicted branches. Corresponding events that only count mis-predicted branches are not supported. On Neoverse N1, Statistical Profiling Extensions (SPE) [2 6] can be used to attribute branch –mis-predicts to individual branches, giving a more targeted analysis than PMU events alone can.

```
ITLB MPKI = ITLB_WALK/INST_RETIRED * 1000
DTLB MPKI = DTLB_WALK/INST_RETIRED * 1000

ITLB Walk Rate = ITLB_WALK/L1I_TLB
DTLB Walk Rate = DTLB_WALK/L1D_TLB
```

- LD_SPEC: Load instructions issued
- ST_SPEC: Store instructions issued
- ASE_SPEC: Advanced SIMD instructions issued
- VFP_SPEC: Floating point instructions issued
- DP_SPEC: Integer data processing instructions issued
- BR_IMMED_SPEC: Immediate branch instructions issued
- BR_INDIRECT_SPEC: Indirect branch instructions issued
- BR_RETURN_SPEC : Return branch instructions issued

Core Memory Traffic

The MEM_ACCESS event counts the total number of memory operations that were issued by the Load Store Unit (LSU) of the core. As these operations first get looked up in the L1D_CACHE, both the events L1D_CACHE and MEM_ACCESS count at the same rate. Neoverse N1 also supports two additional events, MEM_ACCESS_RD and MEM_ACCESS_WR, that can provide the read and write traffic breakdown respectively. Note that these events are not the same as LD_SPEC and ST_SPEC since they count memory instructions issued, but not necessarily executed.

For all the cache hierarchy levels of the core, a set of useful metrics can be derived to study the cache behavior. For an example, L1 data cache metrics can be derived as:

```
L1D Cache MPKI = L1D_CACHE_REFILL/INST_RETIRED*1000
L1D ache Miss Rate = L1D_CACHE_REFILL/L1D_CACHE
```

To study the last level read behavior, Last level cache read miss metrics can be derived as:
Another useful metric to measure the SLC hit percentage for the read traffic is the SLC Read Hit%.

```
LL Cache Read MPKI = LL_CACHE_MISS_RD/INST_RETIRED*1000
LL Cache Read Miss Rate = LL_CACHE_MISS_RD/LL_CACHE_RD
```

Last level cache events do not have a write variant in Neoverse N1 since SLC is only used as an eviction cache for the core.

```
SLC Read Hit %= (LL_CACHE_RD - LL_CACHE_MISS_RD) / LL_CACHE_RD * 100
```

Counting :

- This event statistics help to characterize the overall workload execution behavior, without providing any details on where in the program a particular event occurred.
- Best approach for an initial workload characterization exercise to identify performance limitations of the workload.

Event Based sampling:

- This overflow interrupt records the event count and also the instruction pointer address and register information.
- Used to construct profiling information about the application, including stack trace and function level annotations.
- It is easy to locate the libraries and code portions that contribute to the large portion of the sampled event.
- Event Sampling mode is highly useful for hot spot analysis on a large portion of code, which relies on a statistical approach to sample different events over a large portion of time or code.
- Sampling delay, i.e., between the counter overflow and interrupt handler, which causes skid in the data obtained, that is, data stored during the sampling process and may not be the exact point where the event occurred.
- Another issue comes from the speculative execution style of the processor, where some instructions that executed and triggered events may not be valid if they were on the wrong code path.

- An easy test to verify that PMU events are being counted properly is to use the perf stat functionality of Linux perf tool to count instructions and cycles.

Use Case Study of CPU Performance Analysis :

Use the performance analysis methodology outlined in section [Chapter 3] **for workload characterization and hot-spot analysis with core PMU metrics** captured from Neoverse N1 systems using Linux perf.

- Example workload characterization case study running on the Neoverse N1 Software Development Platform(N1SDP), which has 4 Neoverse N1 cores.
- The PMU events recommended [Chapter 2] are collected in batches of 6 events at a time using Linux Perf tool, “perf stat”.

Stride Benchmark from the DynamoRIO tests :

- The stride micro-benchmark 10 is a pointer chasing benchmark that **accesses values in a 16MB array**, with array position being determined by the pointer being chased.
- The pointer position is a **function of a constant value set in the array before the pointer chasing kernel runs.**
- The maximum achievable **IPC is 4** on Neoverse N1, which is typically **achieved by small & heavily optimized kernels** rather than large applications.

Phase 1: Workload Characterization using Counting Mode

IPC is the first metric to look at in order to evaluate the overall workload execution efficiency.

Metrics	Value
Instructions	10,040,907,789
Cycles	43,809,490,290
IPC	0.22

Table 1: IPC

Cycle Accounting

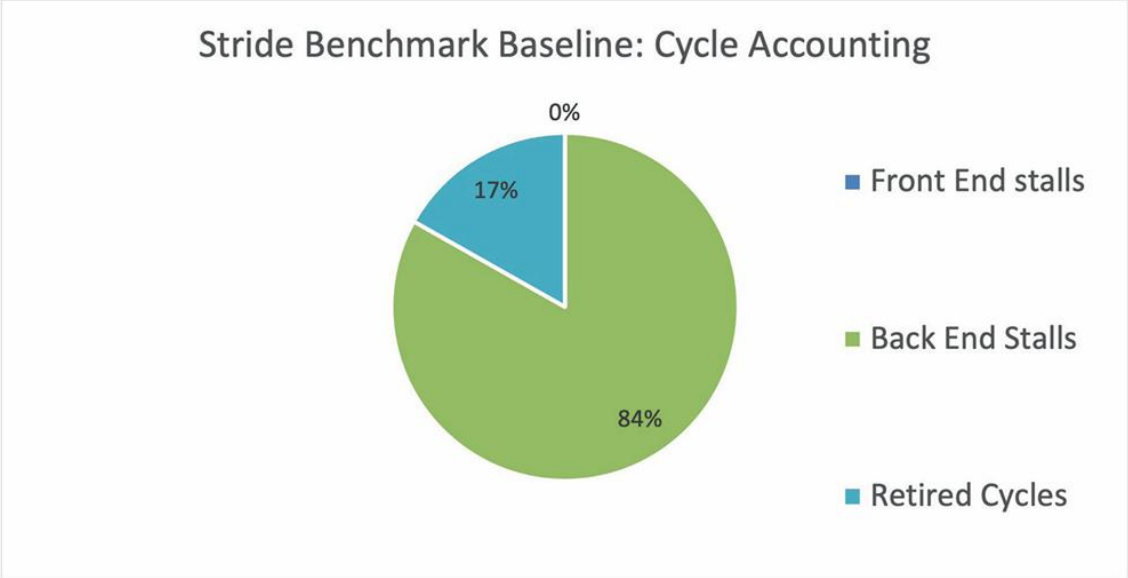


Figure 8: Stride Benchmark: Cycle Accounting

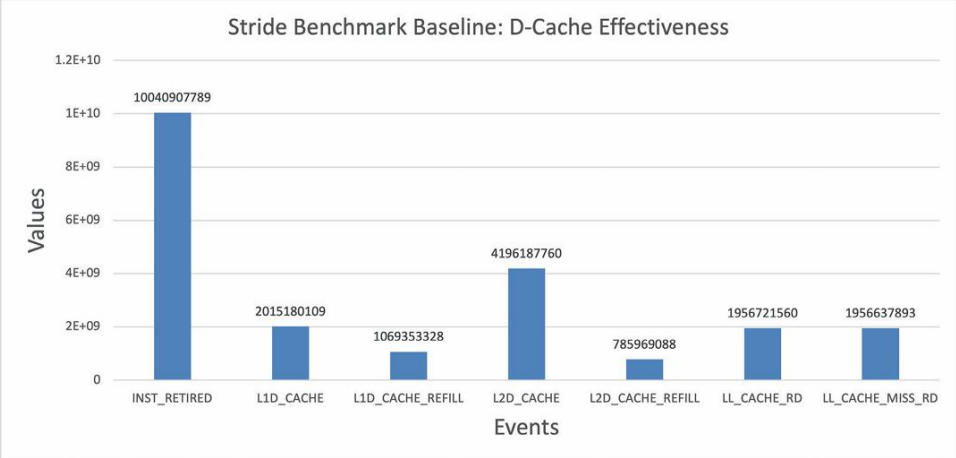
As first step to identify performance bottlenecks of the workload, let us look at the distribution of cycles spent using the Cycle Accounting related events (Table 2), following the methodology in Figure 6.

Metrics	Value
Cycles	43,809,490,290
Stall Front End	4,020,406
Stall Back End	36,777,347,524

Table 2: Cycle Accounting Metrics

The overall cycle distribution percentage (Figure 8) shows that the workload is significantly Back End bound.

Figure 9: Strided Benchmark: D-Cache Effectiveness



The D-Cache effectiveness events (Figure 9) show misses in all levels of the data cache hierarchy with significant last Level cache read misses compared to total reads. This suggests that the workload is memory bound.

On a side note, as L2 is a unified cache to L1 D-Cache and L1-I Cache, the L2 and last level cache misses can also be caused by instruction misses in the Front End. It would always make sense to check for L1-I misses numbers while evaluating your caches performance. However, for this workload, we do not expect L1-I misses because of negligible Front End stalls.

A few detailed cache effectiveness metrics can be derived as below in Table 3:

Metric	MPKI Value
L1 I-Cache MPKI	0
L1 D-Cache MPKI	106
L2 Cache MPKI	78
LL Cache Read MPKI	195

Metric	Miss Rate Value
L1 I-Cache Miss Rate	0

Metric	Miss Rate Value
L1 D-Cache Miss Rate	0.53
L2 Cache Miss Rate	0.18
LL Cache Read Miss Rate	0.99

Table 3: D-Cache Effectiveness Metrics

Observation Note (Table 3): L1 D-Cache MPKI is significantly high at 106, with 53% of the L1 D-Cache accesses resulting in refill. Instruction cache record show no L1-I missed instructions and L1 I-Cache MPKI and miss rates are negligible, as expected. L2 Cache MPKI is high as well at 78, with 53% of accesses requiring a refill. Additionally, last level cache read MPKI show very strong pressure on our memory bandwidth resources because 99% of the read requests were not satisfied and required an LL request back to main memory. With no L1-I misses, L2 and last level cache pressure is only coming from the Back End memory system.

Instruction Mix

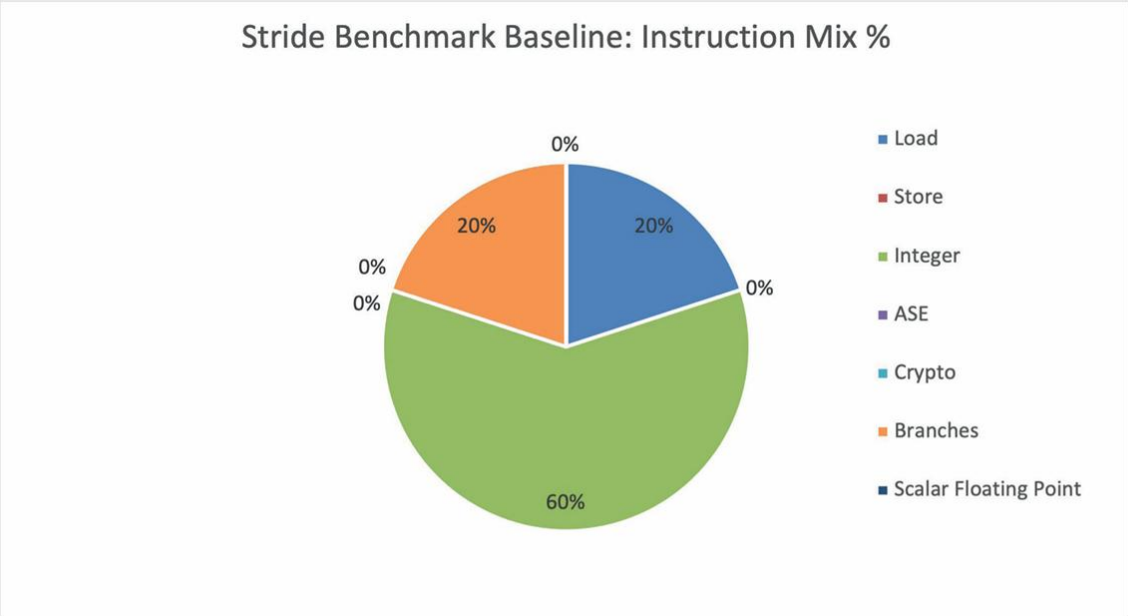


Figure 10: Strided Benchmark: Instruction Mix

Metrics	Value
INST_SPEC	10045062230
LD_SPEC	2009270409
ST_SPEC	6033560
DP_SPEC	6022101605
ASE_SPEC	480
VFP_SPEC	0
CRYPTO_SPEC	0
BR_IMMED_SPEC	2006033840
BR_RETURN_SPEC	1231654
BR_INDIRECT_SPEC	1338354

Observation Note (Figure 10): The instruction mix shows 60% integer operations, 20% load instructions and 20 % branches. As we see significant cache misses in our workload, this tells us that the work is memory bound and needs optimization on improving its cache pressure to improve performance.

Though the workload is not Front End bound, it is still worthwhile to check the Branch Effectiveness counts as the workload also constitutes 20% branches.

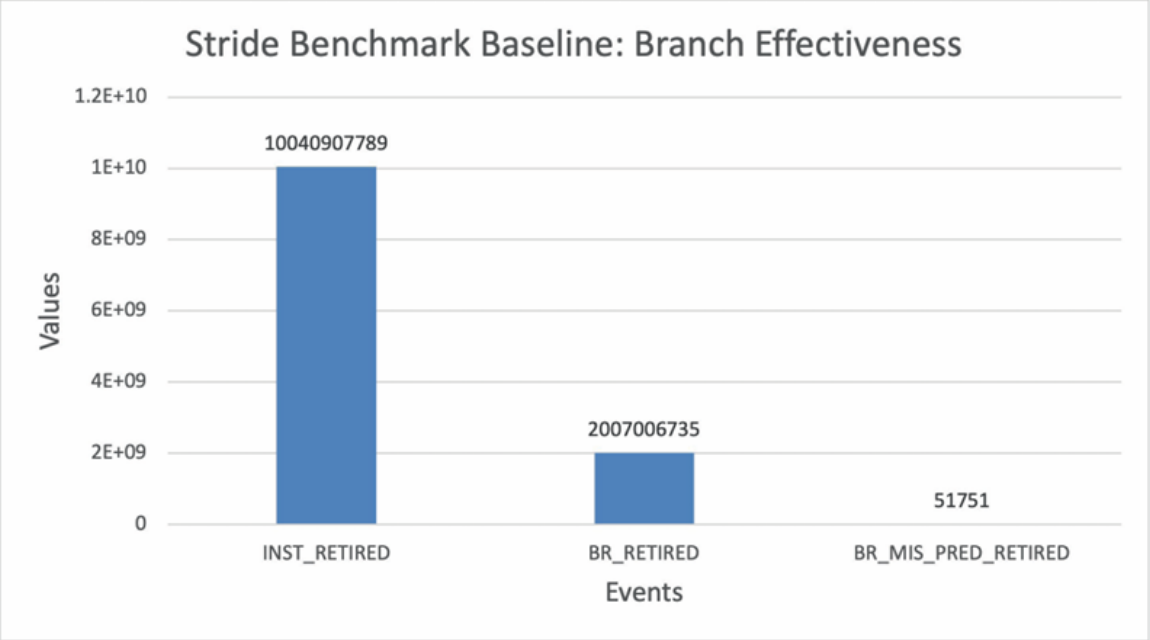


Figure 11: Stride Benchmark: Branch Effectiveness

Metrics	Value
Branch MPKI	0
Branch PKI	200
Branch Misprediction Rate	0

Table 5: Branch Effectiveness Metrics

To evaluate the TLB effectiveness, table walk MPKI metrics are derived below in Table 6:

Metrics	Value
ITLB MPKI	0
DTLB MPKI	14

Table 6: TLB MPKI metrics

Observation Note (Table 6, Figure 12): Instruction side TLB misses are negligible as expected, while data side TLBs exhibit notable walk counts. This suggests that the workload does incur data-side page misses resulting in page table walks for some memory accesses.

Workload Characterization Summary

Figure 13 and 14 show a summary of all the MPKI and Miss rates on one chart.

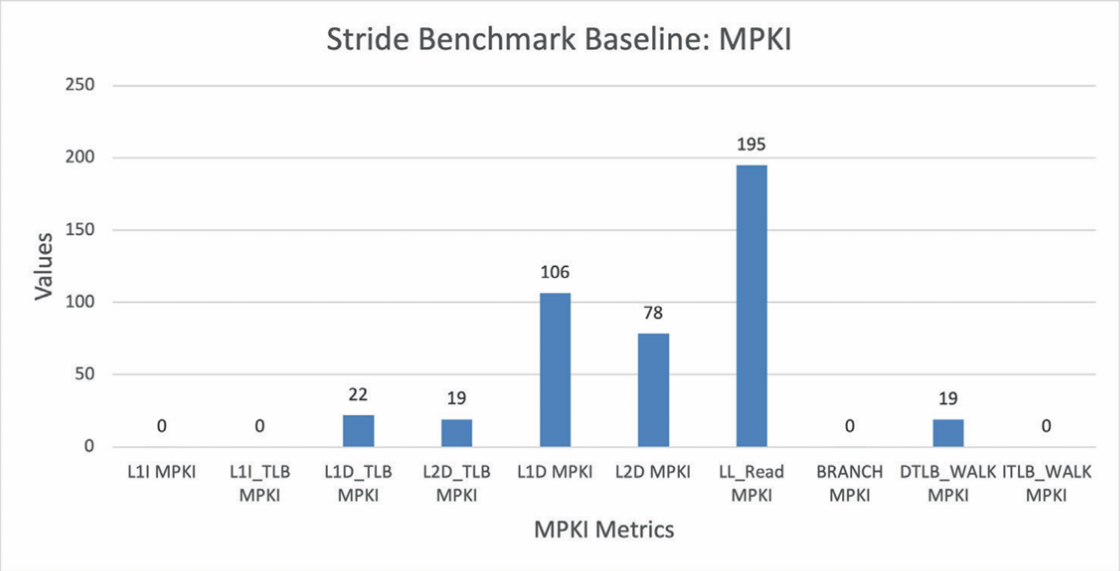


Figure 13: Stride Benchmark: MPKI

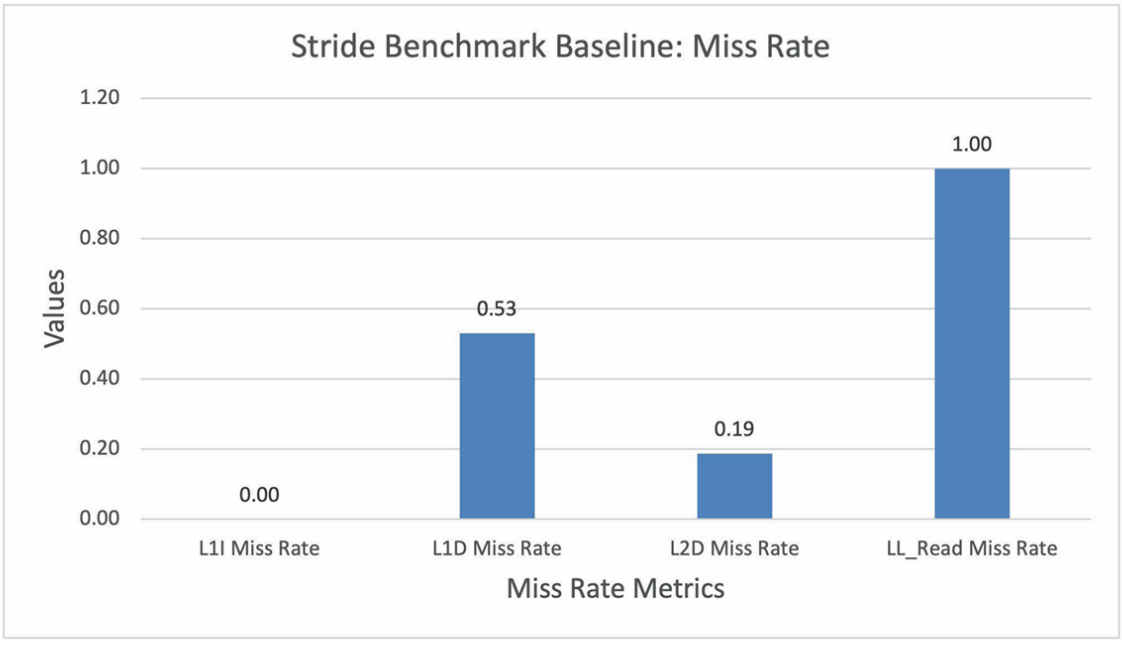


Figure 14: Stride Benchmark: Miss Rate

Conclusion of Use Case or Workload : Strided Benchmark

- Poorly performing with **IPC = 0.22.**
- The workload is heavily Back End bound with a **very high Back End stall rate of 83%.**
- The workload has **60% integers**, **20% branches** and **20% load operations.**
- The workload exhibits **significant Back End pressure in the data cache side**, with L1D MPKI of 106, L2 MPKI of 78 and Last Level Cache Read MPKI of 195.
- The **Front End of the CPU is operating smoothly** with stats like Branch MPKI, L1I MPKI and ITLB MPKI being negligible, which corresponds to the zero Front End stalls.

Conclusion of this Benchmark : The characterization evidence supports the workload behaviour that **there is a memory bottleneck in our system** and we should next investigate how to address it.

Understand the Bottlenecks :

- Back End stalls
- Hierarchical D – Cache Events

Observation Note (Figure 16): Sampling cycles and instructions show 99% samples at ldrb instruction, which is the load instruction that access array elements by pointer chasing. The hot code region for instruction and cache miss events is also from the same instruction, and 99% of samples are taken there as well, sampled at the "subs" instruction after the load which accesses the array elements by pointer chasing. The annotated disassembly code in Figure 16 shows the "ldrb" and "subs" instructions highlighted. Note that perf sampling can introduceskid and hence hot code line is on the subs instruction after the load that is the bottleneck. This suggests that the main bottleneck in this application comes down to performance issues with the array access.

Phase 3: Code Optimization

One well known optimization for reducing memory pressure is prefetching, which can be done by hardware or software. In this case, the algorithm is chasing pointers in every seventh cache line and from close observation, it is clear that the stride has a pattern. We tried software preloading on the workload and obtained much better performance, which indicates that hardware prefetcher is not efficient with this pattern on the platform under test.
The code for prefetcher tuning is in the source code in Figure 15 (highlighted green).
A preprocessor directive is added inside the loop which enables software pre-loading and help tune the prefetch distance.

```
1  #if defined(ENABLE_PREFETCH) && defined(DIST)
2       const int prefetch_distance = DIST * kStride * kLineSize;
3       __builtin_prefetch (& buffer[position + prefetch_distance], 0, 0);
4  #endif
```

For the __builtin_prefetch tuning, we have used the option (0,0) to tune for read access and to prefetch into L1 Data Cache respectively, as we are only loading the array elements once, and the data is not reused for being stored in other hierarchy levels. Subsequently, we also trained the software preloader for multiple prefetch distance values until we got a saturated high performance at DIST = 40 as shown in Figure 17.

Optimized Code Characterization Summary

With software preloading, we achieved a performance uplift of 2x- execution time reduced from 16 seconds to 8 seconds. Let us now look at how we observe the performance uplift in PMU events for optimized code, and measure it against non-optimized code. We will first look at the IPC change.

Metrics	Baseline	Optimized	% Change
Instructions	10,040,907,789	14,031,047,062	+39.7%
Cycles	43,809,490,290	20,858,281,670	-52.4%
IPC	0.22	0.67	+193%

Table 7: Cycle Accounting% Change

Observation Note (Table 7): With the optimization applied, we have reduced cycles by half which matches the 2x performance lift obtained. We also have ~3x improvement in IPC, with 193% change. Note that, the additional code for prefetching has also increased the total instructions retired by 39.7%. Let us check the instruction additions comparing the the disassembly for the optimized code with the baseline code. A diff between the baseline and optimized code shows us the extra instructions executed as below:

disassembly for the optimized code with the baseline code. A diff between the baseline and optimized code shows us the extra instructions executed as below:

```
mov x3, #0x4600 // #17920
nop
add x0, x3, w19, sxtw
prfm pldl1strm, [x20, x0]
```

Let us see if the instruction mix reflects these changes in Table 7.

Metrics	Baseline	Optimized	% Change
Integer operations	6,022,101,605	8,016,721,840	+33%
Load operations	2,009,270,409	4,006,679,514	+99%

Table 8: Instruction Count% Change

Observation Note (Table 8): As expected, the load operations doubled, as the software preload instruction(prfm) is counted for the LD_SPEC event. Other instructions get counted with the DP_SPEC event, which is increased by 33%.

As our major performance bottleneck was the memory pressure, let us look at the improvements in the Cache Effectiveness metrics. MPKI is not an apple-apple comparison between the two runs as the workload executes ~40% more instructions with software pre-loading. Therefore, we will just look at the miss rates and access count change as shown in Table 9.

Metrics	Baseline	Optimized	% Change
L1 D-Cache Accesses	2,015,180,109	4,011,259,164	+99.4%
L1 D-Cache Refills	1,069,353,328	331,828,745	-68.96%
L2 Cache Accesses	4,196,187,760	4,195,830,182	-0.008%
L2 Cache Refills	785,969,088	966,401,485	+22.95%
LL Cache Reads	1,956,721,560	1,960,623,393	+0.19%
LL Cache Read Misses	1,956,637,893	1,960,505,782	+0.2%

Table 9: Cache Metrics% Change

Observation Note (Table 9): L1 D-Cache accesses are doubled as the optimized code executes twice the load operations with prefetch instruction counted as a load. As we prefetched into L1 specifically in read mode, we have been able to reduce the L1 D-Cache misses significantly by 68%, which attributes to the performance uplift of 2x obtained. L2 cache accesses and LL cache read accesses remain the same, as we did not prefetch into any of these hierarchy levels.