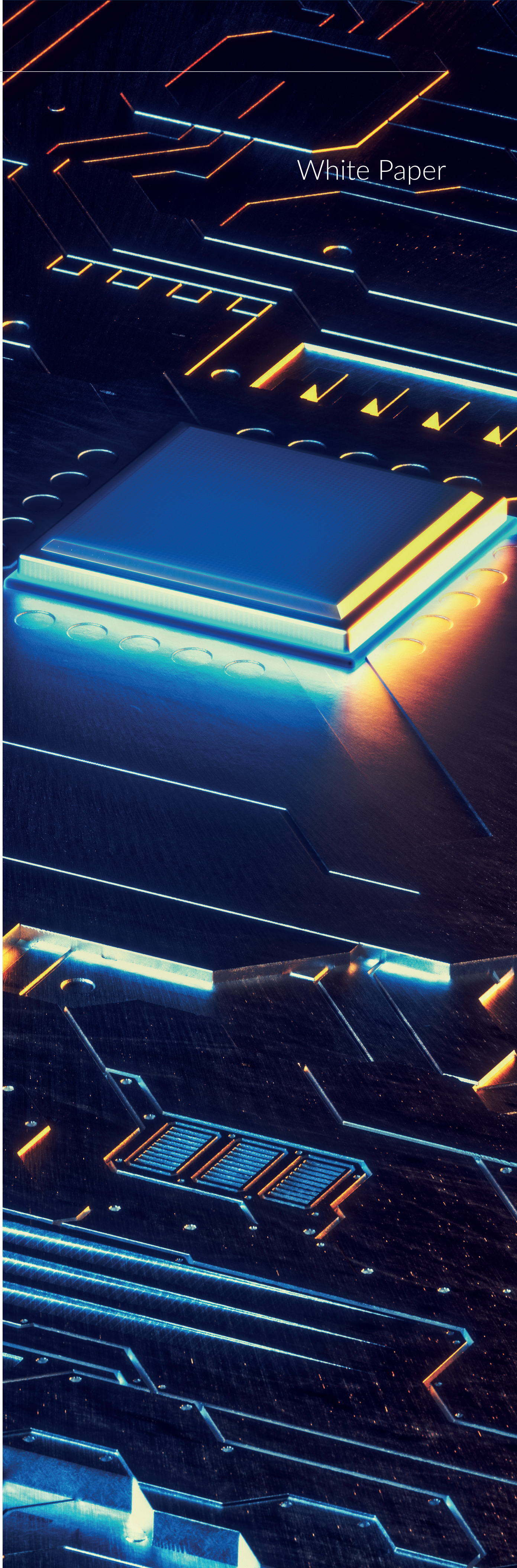
White Paper

Arm Neoverse N1 Core: Performance Analysis Methodology

Performance Analysis on Neoverse N1 Core Using Hardware PMU Events

By Jumana Mundichipparakkal, Krishnendra Nathella, and Tanvir Ahmed Khan



Contents

1 Introduction

The Arm Neoverse ecosystem is growing substantially with many Arm hardware and software partners developing applications and porting their workloads onto Arm-based cloud instances. With Neoverse N1 based systems becoming widely available, many real-world workloads are showing very competitive performance and significant cost savings when compared to legacy systems. Some recent examples include: H.264 video encoding, memcached, Elasticsearch, NGINX and more.

To maximize their execution performance, developers use performance analysis and workload characterization techniques to study performance characteristics of applications. Server class systems support a wide range of performance monitoring techniques to measure workload efficiency, evaluate their resource requirements, and track resource utilization. Such measurements are useful to tune both software and hardware and also help guide future system design requirements.

The Arm Neoverse micro-architecture has been developed with both high performance and power efficiency in mind. As such, our philosophy to performance monitoring might differ slightly from what software developers have used to analyze systems based on other architectures. This paper outlines a methodology for workload characterization using the Performance Monitoring Unit (PMU) capabilities on the Neoverse N1 CPU to identify and eliminate performance bottlenecks. The intended audience is software developers and performance analysts working on software optimizations, tuning, and development.

The content of this paper is divided into four chapters:

- The first chapter introduces the hardware PMU on Neoverse N1 with a list of most relevant PMU events for workload characterization.
- The second chapter presents a workload characterization methodology using Neoverse N1 core PMU events.
- The third chapter illustrates how the Linux perf tool can be used to collect Neoverse N1 PMU events.
- The final chapter demonstrates a workload characterization and hot spot analysis with an example workload case study.

2 Neoverse N1 Performance Monitoring Events

Performance monitoring requires collecting application execution information on a given system which can be obtained by software or hardware means. Software monitoring techniques provide software traces and events counted by system software. Hardware monitoring techniques work by collecting hardware events directly from the CPU/System. For collecting hardware events, modern processors implement a dedicated Performance Monitoring Unit (PMU) which facilitates the measurement of a wide range of hardware execution related events. Profiling hardware events can provide insight into the code execution behavior on the various micro-architectural units. Multiple events can be monitored and correlated with software execution in order to identify optimization opportunities and evaluate whether the workload is taking advantage of the underlying micro-architecture in an optimal way. Some of the events that are supported include instructions retired, elapsed CPU cycles, cache/TLB accesses, and branch predictions.

Arm Performance Monitoring Unit

Arm architecture supports PMU capability via an optional extension to the architecture called Performance Monitors Extension. An Arm PMU hardware design (Figure 1) consists of the following components:

- PMU configuration registers for control and event selection
- PMU event counters
- Dedicated function counters

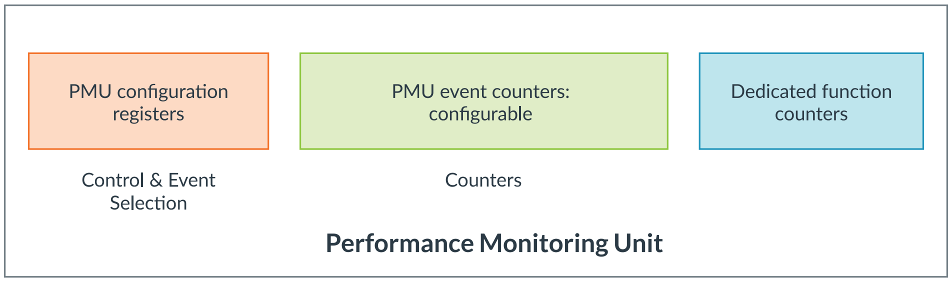

PMU configuration registers
PMU event counters: configurable
Dedicated function counters
Control & Event Selection
Counters
Performance Monitoring Unit

Figure 1: Performance Monitoring Unit

Arm PMU hardware employs multiple configuration registers including the PMCR and PMEVTYPER registers, for unit control and measurement selection respectively[1]. The PMU hardware also incorporates a set of event counters to count the raw hardware events as requested by the user. Each event has a unique hex eventcode associated with it designed by the hardware vendor, which gets set in the PMEVTYPER register. Apart from configurable counters, AArch64 has a dedicated counter for CPU cycles which is a required functionality on all Arm compatible designs. Implementation of the PMU hardware is implementation specific in regard to the number of counters available and event types that can be counted including the respective event codes.

When a PMU event is configured, one of the event counters is assigned to the digital logic associated with the event being measured. PMU events can be measured using the Linux perf tool, either using the software listed event names for common "hardware events" or the dedicated hex event code associated with the raw "PMU hardware events" specified by the architecture. A typical processor might support hundreds of events for both performance and debug purposes, but a subset can be selected for a workload characterization exercise to study high level execution bottlenecks and resource utilization. The rest of the events specific to core subsystem units can be used for root causing a performance issue in depth, once the performance limiting unit has been identified from the characterization exercise.

Arm Performance Monitoring Unit Implementation References

PMU events implemented by a specific core implementation are listed in the Technical Reference Manual for that product. The behavior of these events are defined in the respective Arm Architecture Reference Manual[1] as "Common Events". The Common Events are generic definitions that apply to all micro-architectures, however most of them are soft requirements and may not necessarily be implemented.

Neoverse N1 Performance Monitoring Unit

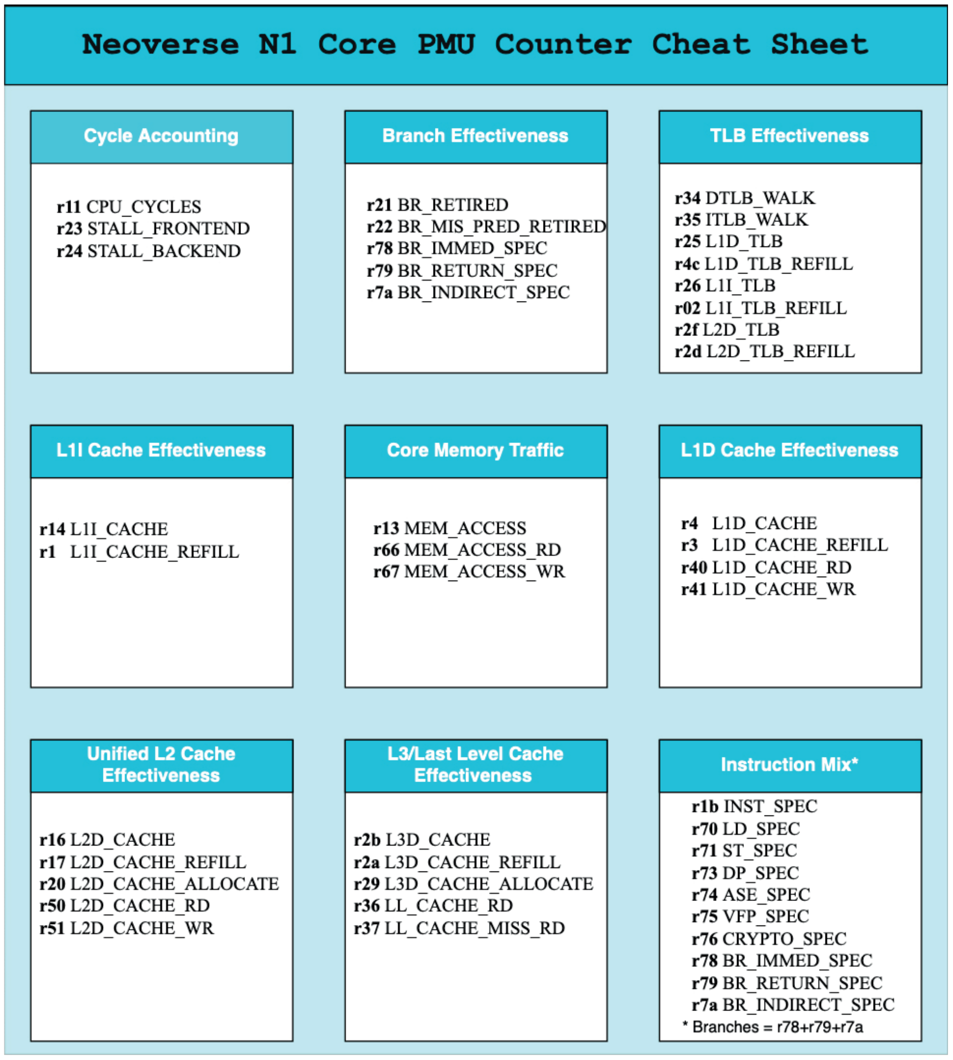
The Neoverse N1 CPU implements the PMU extensions of the Arm v8[1] with support for 100+ hardware events. The Neoverse N1 PMU has 6 configurable counter registers and 1 dedicated function counter to count CPU cycles.

The PMU events implemented by the Neoverse N1 core are listed in the ARM Neoverse N1 Core Technical Reference Manual (TRM) Part D[2]. These events are defined in the Arm v8[1] as "Common Events". In addition to the TRM, we also provide Neoverse N1 PMU Guide[3], a supplementary guide to the hardware PMU events implemented by the core. This PMU Guide provides detailed descriptions of PMU events categorized per CPU block. Micro-architectural and architectural definitions required for better understanding each PMU event is included, with relevant definitions marked as reference to each PMU event description. The Neoverse N1 PMU Guide also adds an exclusive CPU execution flow chapter that demonstrates key CPU execution flows that the memory subsystem, with depiction of PMU events being counted in each stage.

We recommend using the N1 PMU Guide[3] as the go-to reference manual for event descriptions for performance analysis activities using PMU events.

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.



Neoverse N1 Core PMU Counter Cheat Sheet

Cycle Accounting

r11 CPU_CYCLES
r23 STALL_FRONTEND
r24 STALL_BACKEND

Branch Effectiveness

r21 BR_RETIRED
r22 BR_MIS_PRED_RETIRED
r78 BR_IMMED_SPEC
r79 BR_RETURN_SPEC
r7a BR_INDIRECT_SPEC

TLB Effectiveness

r34 DTLB_WALK
r35 ITLB_WALK
r25 L1D_TLB
r4c L1D_TLB_REFILL
r26 L1I_TLB
r02 L1I_TLB_REFILL
r2f L2D_TLB
r2d L2D_TLB_REFILL

L1I Cache Effectiveness

r14 L1I_CACHE
r1 L1I_CACHE_REFILL

Core Memory Traffic

r13 MEM_ACCESS
r66 MEM_ACCESS_RD
r67 MEM_ACCESS_WR

L1D Cache Effectiveness

r4 L1D_CACHE
r3 L1D_CACHE_REFILL
r40 L1D_CACHE_RD
r41 L1D_CACHE_WR

Unified L2 Cache Effectiveness

r16 L2D_CACHE
r17 L2D_CACHE_REFILL
r20 L2D_CACHE_ALLOCATE
r50 L2D_CACHE_RD
r51 L2D_CACHE_WR

L3/Last Level Cache Effectiveness

r2b L3D_CACHE
r2a L3D_CACHE_REFILL
r29 L3D_CACHE_ALLOCATE
r36 LL_CACHE_RD
r37 LL_CACHE_MISS_RD

Instruction Mix*

r1b INST_SPEC
r70 LD_SPEC
r71 ST_SPEC
r73 DP_SPEC
r74 ASE_SPEC
r75 VFP_SPEC
r76 CRYPTO_SPEC
r78 BR_IMMED_SPEC
r79 BR_RETURN_SPEC
r7a BR_INDIRECT_SPEC
* Branches = r78+r79+r7a

Figure 2: Neoverse N1 PMU Events Cheatsheet

3 Neoverse N1 Performance Analysis Methodology

Performance optimization and software tuning is more challenging than ever today. Modern high-end processors typically have high core counts with complex instruction sets, various levels of parallelism, and deeper memory hierarchies. Also, a significant proportion of large-scale cloud workloads run on virtual machines, which makes it even harder to trace the workload execution behavior to evaluate the power-performance efficiency in the data centers. However, correlating the software execution with the micro-architectural behavior is critical in optimizing software for efficient execution on the underlying hardware.

A single core forms the fundamental execution block of the processor. These cores get combined in multiple configurations with memory subsystems to form a computing system. In this chapter, we will look into the high-level details of a single Neoverse N1 core and a methodology for workload characterization using the hardware PMU events of the core.

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.

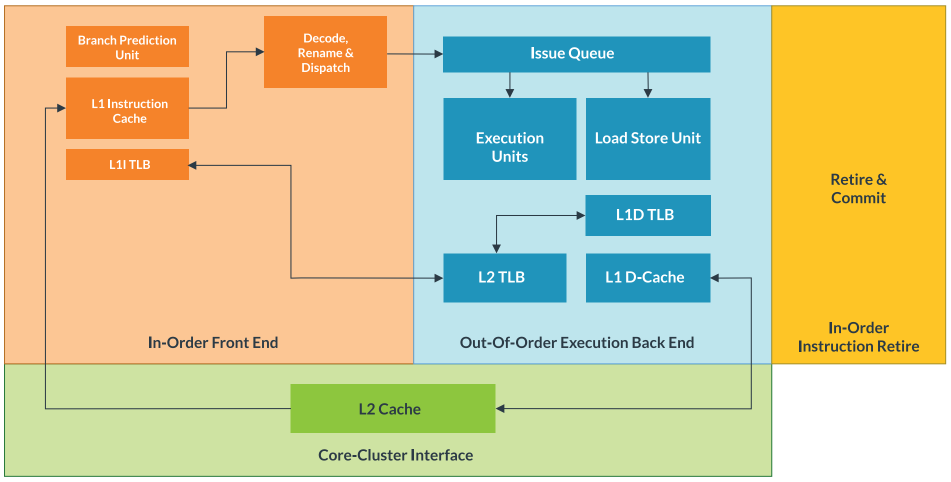


Figure 3: Arm Neoverse N1 CPU Building Blocks

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.

Neoverse N1 Cluster Configurations

The Neoverse N1 systems come in different configurations, depending on the implementation choices made in the interconnect and cluster/system level last level caches. Some of the optional configurations in Neoverse N1 systems are depicted below in figures 4 and 5. Figure 4 shows a configuration in which multiple cores can be configured in a Dynamic Sharing Unit (DSU) based cluster system with two cores in a single cluster. This DSU cluster contains a snoop filter and an optional L3 cluster cache that can be shared by the cores within the cluster. This optional L3 Cache can be up to 2 MB in size in design.

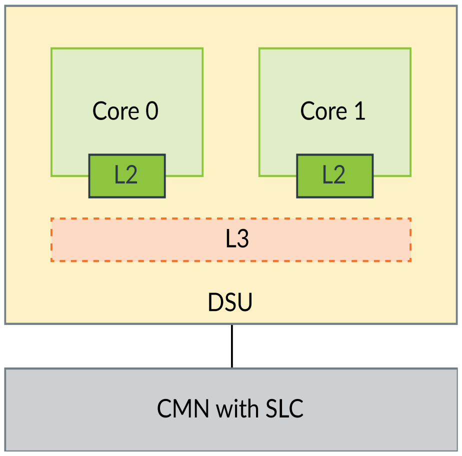


Figure 4: Neoverse N1 DSU Cluster System with Optional L3

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.

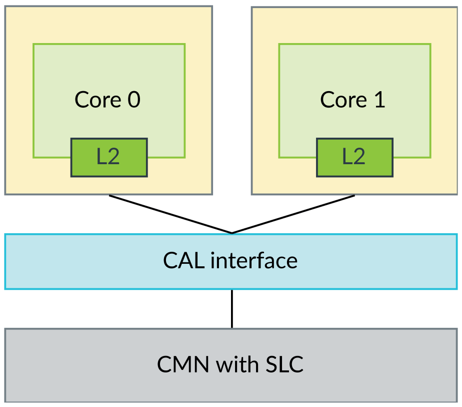


Figure 5: Neoverse N1 System with Direct Connect Configuration

All systems with the coherent mesh interconnect support a shared system-level cache which can be up to 256 MB in size. Understanding the cache hierarchy and configuration of the system being analyzed is crucial in deriving insights from the cache effectiveness PMU events. It is always best to check with the Silicon Provider for details on the system configuration for the underlying system including the cache sizes.

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.

Instructions Per Cycle

The combination of cycles spent, and instructions executed give an overview of the CPU workload and execution time on any architecture. The number of instructions represents the amount of work the CPU does, and cycles represent the total time it took to do it. Instructions Per Cycle (IPC) is a key metric for evaluating the performance of a workload on a micro-architecture, which can be derived as below:

```
IPC = INST_RETIRED/CPU_CYCLES
```

In a non-stalled pipeline, the IPC will be the highest and can be a direct measurement of the instruction level parallelism the processor supports. The higher the IPC achieved, the better the pipeline efficiency during the workload execution. If the IPC is very low, it means the cycles spent are stalled hugely which could point to potential performance issues. Neoverse N1 pipeline has a maximum IPC of 4, as it can fetch 4 instructions per cycle.

For instruction count, Neoverse N1 supports INST_RETIRED and INST_SPEC events, both counting the instructions executed but in different stages of the pipeline. Whilst INST_RETIRED counts the total instructions that are architecturally executed in a program, INST_SPEC counts the total number of instructions decoded speculatively to the processor. INST_SPEC can give a better indication of the total utilization of the execution unit, as it counts instructionswhich could have been executed but were not necessarily committed. A large difference between INST_SPEC and INST_RETIRED usually indicates that high branch misprediction rate or other faults that would cancel speculative instructions that have been decoded which is inefficient.

Note that INST_RETIRED for a workload will be same on all implementations of an architecture, where as cycles vary depending on the micro-architectural implementation and system configuration.

Cycle Accounting/Pipeline Stalls

As discussed above [Neoverse N1 Core], an out of order CPU has an in-order Front End unit which fetches instructions and decodes them into micro-operations which are issued to the Back End for execution. The fetch unit in the Front-End fetches instructions from the L1I Cache, which requires accesses to the L1I TLB for getting the physical addresses as well as a branch prediction unit to speculatively fetch subsequent instructions. While the Back End may execute micro-operations out-of-order all instructions are committed in-order with respect to dependencies.

The pipeline described above allows for high throughput of instruction execution. However, stalls in the pipeline may occur due to a variety of reasons, such as fetching wrong code path due to branch mispredictions or waiting for data from memory or L2/L3 caches. In addition, executing excessive wrong path code reduces the number of useful cycles spent by the CPU. To evaluate the efficiency of pipeline execution, Neoverse N1 supports two top level STALL events to evaluate the number of cycles stalled in the Front End and the Back End of the CPU.

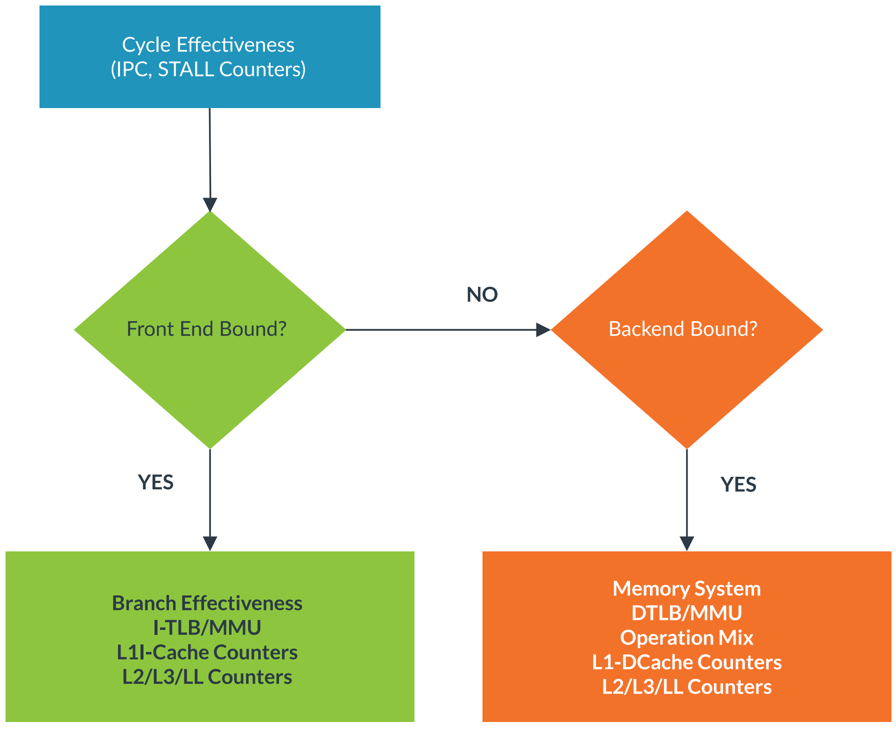


Figure 6: Cycle Effectiveness Evaluation Methodology

The STALL_FRONTEND and STALL_BACKEND events can be used to provide an indication of the major bottlenecks during the execution of the workload. Using these two events, we can derive the relative percentage of stalled cycles in the Front End and Back End respectively:

```
STALL FRONTEND Rate = STALL_FRONTEND/CPU_CYCLES
STALL BACKEND Rate  = STALL_BACKEND/CPU_CYCLES
```

A relatively high Front End stall rate indicates that cycles are being wasted due to pipeline stalls in the in-order Front End division, while a relatively high Back End stall rate shows cycles are wasted due to pipeline stalls in the Back End. This breakdown helps to narrow down the dominating blocks that can be further analyzed to identify performance 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

The N1 PMU Guide[3] details all the raw hardware events that can be referred for each of these CPU blocks. We will cover select events recommended from the N1 PMU CheatSheet[Chapter 2] including some of the metrics that can be derived for characterization of CPU blocks next.

Branch Effectiveness

Branch mispredictions are costly in a deeply pipelined CPU, causing frequent pipeline flushes and wasted cycles. As a general rule, workloads typically contain on average, 1 branch in every 6 instructions. Though modern CPUs have optimized branch prediction units, there are many use cases like ray tracing, decision tree algorithms, etc. that are branch heavy and hard to predict. In some of these applications there can be hundreds of unique branch paths to take and the target may be input data dependent.

Branch prediction performance can be evaluated by using the two raw PMU events, BR_MIS_PRED_RETIRED and BR_RETIRED. BR_MIS_PRED_RETIRED gives an account of the total branches executed but were mis-predicted. This means that the direction of code path in the mis-predicted basic block was wrong and the following operations in the path were wasted, causing a pipeline flush. BR_RETIRED counts the total branches architecturally executed by the CPU.

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 MPKI provides total branch mis-predictions per kilo instructions and Branch Mis-prediction Rate gives an indication of the ratio of branches that were mis-predicted to overall branches. Both the metrics can be used to further investigate using perf record, to determine which functions are causing the increase branch miss rates [Chapter 4].

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.

TLB/MMU Effectiveness

Another important performance evaluation step is to check the virtual memory system performance, that affects the instruction fetch performance in front-end and memory access performance on the data side. The processor needs to translate a virtual address to physical address for any instruction/data memory access before it accesses the respective cache. Note that a program's view of memory is virtual address, but the processor works with the physical address when accessing cache or memory.

Virtual to physical mappings are defined in the page translation tables which reside in system memory. Accessing these tables requires one or more memory accesses which take many cycles to complete – this is referred to as a page table walk. However, to make these translations faster, Translation Lookaside Buffers (TLBs) cache translation table walks, greatly reducing the number of accesses to system memory.

Neoverse N1 implements a two level TLB hierarchy. The first level contains separate, dedicated TLBs for the instruction and data (load/store) address translations. Total accesses to these TLBs are counted by L1I_TLB and L1D_TLB respectively. The second level contains a unified L2 TLB that is shared by both I-side and D-side accesses. There are corresponding REFILL counters, that count the refills in these TLB levels. Those accesses that cause a page table walk due to misses in the I-side and D-side TLBs are counted by events, ITLB_WALK and DTLB_WALK respectively.

For evaluating the TLB effectiveness, there are four metrics that can be derived from the raw events:

```
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
```

ITLB MPKI and DTLB MPKI provide the rate of TLB Walks per kilo instructions for instruction and data accesses respectively. These derived metrics help to evaluate and correlate the TLB efficiency with respect to the total instructions. DTLB Walk Rate provides rate of DTLB Walks to the overall TLB lookups made by the program. Note that this is same as DTLB_WALK/MEM_ACCESS as every MEM_ACCESS causes a L1D_TLB access. ITLB walk rate provides a percentage of ITLB walks to the overall TLB lookups initiated from the instruction side.

Instruction Mix

The Neoverse N1 micro-architecture has 8 execution units which can process five types of operations: branch, single cycle integers, multi-cycle integers, load/store unit with address generation, and advanced floating point/SIMD operations. Instructions that are issued to these execution units can be counted by the following PMU events:

- 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

Note that these are speculatively executed counts as these instructions are counted at the issue stage and give an estimate of the execution unit utilization, but not the retired instruction mix of a program. Neoverse N1 does not support retired event counters for counting the architectural instruction mix. Neoverse N1 supports events to further break down the branch operations into immediate, indirect and return branches, counted by events BR_IMMED_SPEC, BR_INDIRECT_SPEC, and BR_RETURN_SPEC respectively. Sum of these three branch operation events can be used to compute the total branches. For evaluating the load on the execution units of the CPU, it is best to derive percentage of each types of operation with respect to the INST_SPEC counter, which counts the total instructions issued for execution.

Example:

```
Memory Instruction Percentage = (LD_SPEC +  ST_SPEC)/INST_SPEC*100 (memory instruction
    ↪percentage in the overall execution)
```

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.

Cache Effectiveness

The Neoverse N1 implements a multi-level cache hierarchy. The first level (L1) includes a dedicated cache for instructions and a separate dedicated cache for data accesses. The second level (L2) is a unified L2 cache that is shared between code and data. Further down the hierarchy, the system could have an optional L3 cache in the core cluster and an optional shared system level cache (SLC) in the interconnect. L3 and SLC caches are implementation options.

The Neoverse N1 core supports hierarchical PMU events for all the cache hierarchy levels. For each level of cache, there are total access counts and refill counts. Note that AArch64 does not support cache MISS counters, but only REFILLs. A cache miss could lead to multiple cache line refills if the access is on a cache line boundary or multiple cache misses could be satisfied by a single REFILL. Refer to the N1 PMU Guide[3] for details on the cache event counter descriptions. Cache policies and associativity details can also be referred to for more details in the Chapter Micro-architecture details in the N1 PMU Guide[3].

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
```

Cache REFILL Characterization

For Neoverse N1 systems with a DSU cluster, N1 also supports two cache REFILL variants that can be used to measure if the cache refill data came from inside the cluster or outside the cluster. In this configuration REFILL counts can be divided into INNER and OUTER refill operations[3].

Remote Cache Access

For Neoverse N1 systems with multiple sockets or SOCs, N1 supports the REMOTE_ ACCESS event which counts the memory transactions that were handled by the data source from another chip.

Last Level Cache Counter Usage

As we saw in [Chapter 3] N1 System Configurations section, Neoverse N1 systems could support cluster level L3 caches and a shared system level cache. Neoverse N1 implements two sets of cache hierarchical events for L3 and LL (Last Level).

L3 cache is an optional cache and the respective events are counted only if the core implements the L3 cache. This means that if the core does not have an L3 cache, this event should count zero. However, if system is configured in a dual-core cluster system, this event could count the peer core traffic from snooping within the cluster.
[Check your SOC specification for details]

On systems which support a shared system level cache, LL_CACHE_RD counts the total accesses to the SLC. In a system that has the SLC configured to count LL_CACHE_RD events, LL_CACHE_RD counter counts total SLC accesses made by the core and LL_CACHE_MISS_RD counts the access missed at SLC.

To study the last level read behavior, Last level cache read miss metrics can be derived as:

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

Another useful metric to measure the SLC hit percentage for the read traffic is the SLC Read Hit%. 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
```

4 Performance Analysis Using Linux Perf Tool

The Linux perf tool[7] is a widely used open source tool for collecting software and hardware performance events from different sources within the hardware and system software. The kernel employs a perf_event subsystem to collect measurable events including the hardware PMU events from the processor itself. As shown in Figure7, each core has its own dedicated PMU hardware and the kernel perf driver collects events from each core PMU separately.

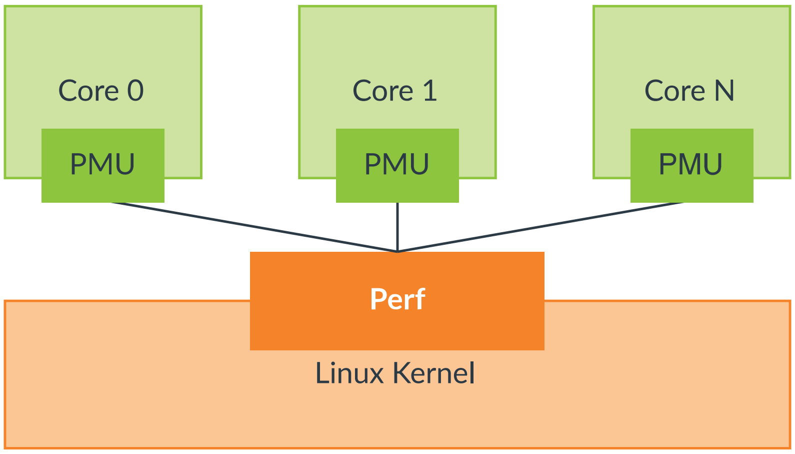


Figure 7: Performance Monitoring System

The Linux perf_event system provides an interface between Linux kernel and user space performance monitoring tools to collect the raw hardware events as needed. Linux perf tool is such an open source tool available in Linux that it can be used for performance monitoring, which supports two types of measurement techniques:

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

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

All the above listed performance measurement techniques within Linux perf tool for both counting and event sampling mode are available the below functionalities:

- stat: provide performance counter statistics for overall execution of the program
- record: record the execution performance with percentage of samples for each event per libraries and functions
- report: generate a report of the recorded sample using record
- annotate: annotate a report with samples % on the disassembly of the code

When high accuracy is needed, for example when profiling hot loops or significant portions of code, the "Counting" mode should be preferred for its accuracy – noting that it might require multiple profiling iterations whenever many different events must be logged. For Neoverse N1 CPU, it is best to count maximum 6 events at a time to get a dedicated counter for each event. When number of events is more than the total number of counters available, the counter is time multiplexed between events and the the final count is scaled for the total time period. This multiplexed counting could cause accuracy issues but is usually reliable unless a precise measurement is needed.

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. It is important to be aware that this method has some limitations causing accuracy issues. 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. Though this approach has some accuracy limitations, it is still the best way to get closer to identifying the hot spots in the code execution. Linux perf allows tuning the sampling frequency, which helps to study variations in the event counts if the data shows large inconsistencies across runs.

For more details and examples on how to use Liux perf tool, refer to https://www brendangregg.com/linuxperf.html

We will cover how to use Linux perf tool functionalities for workload characterization and hot spot analysis in Chapter 4.

Collecting Hardware PMU Events for Arm Architecture using Linux Perf Tool

In order to enable PMU event collection, the Linux Kernel must be built with CONFIG_HW_PERF_EVENTS enabled in the kernel config. Most of the production builds have this config option checked enabled, but if you are building a custom kernel remember to enable this config option. In addition, there are two system settings which need to be configured as root user in order to obtain kernel symbols and add extra privileges:

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

The perf_event_paranoid control affects privilege checks in the kernel, and setting this to -1 permits opening events that might reveal sensitive information or could impact the stability of the system. See the details from kernel documentation at:
https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html#unprivileged-users
https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#perf-event-paranoid

The kptr_restrict control affects whether kernel addresses are exposed (e.g. via /proc/kallsyms). Some developers use this technique to get kernel symbol resolution when they don't have the vmlinux to hand (or where KASLR is in use). See the details from kernel documentation on kptr_restrict: https://www.kernel.org/doc/html/latest/admin-guide/

In both the cases, there are potential security implications so please check the official kernel documentations and consult your system administrator before enabling them.

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. Perf stat counts the total count of a specified event, provided as the hex register code. 0x8 is the hex code for instructions retired and 0x11 is the hex code for CPU cycles on the Arm architecture. These events are provided to the perf stat -e option with a prefix 'r' to it.

```
perf stat -a -e r8,r11 sleep 10
```

The above command should count total count of instructions and cpu cycles on all the CPU for 10 seconds. Linux perf allows to count for a particular CPU, per process, per thread etc, which can be found in the perf stat man page. Result from the above perf stat run gives:

```
21663586606      r8
14839358570      r11
```

Note that sometimes perf will silently fail if an event is not supported or enabled. Check with your CPU TRM for the event support on your system.

The examples in the following sections use the raw event numbers to indicate which events should be monitored. However, Linux kernel versions 4.17 and later support accessing N1 core PMU events using named values. For earlier versions, the raw event number must be used. Refer to the Arm Neoverse N1 PMU Guide for mapping named events to numbers. For automation tools for PMU event measurement, Arm provides machine readable JSON files with all the events and their event codes in the Github repository at ARM-Software/PMU-Data[9].

Collecting Hardware PMU Events using Counting Mode

For counting mode, use the ‘perf stat’ command from Linux Perf tool as shown below. To count all events for characterization, as recommended in the cheatsheet [Chapter 2], typical solution is to capture events in batches of the total counter registers available in the platform.

An example run on the command line for instructions and cycles using event names (using hex code is added below):

```
ubuntu@linux-1:~$ perf stat -e r8,r11 -- sleep 10

Performance counter stats for 'sleep 10':

    1242692      r8
    1000757      r11

10.001721954 seconds time elapsed

10.000772000 seconds user
0.000000000 seconds sys
```

Collecting Hardware PMU Events using Sampling Mode

For sampling events, use the ‘perf record’ command from Linux Perf tool as below. For more details on how to conduct sampling and analyze the sampled data with these command lines, refer to these Linux perf Examples[8].

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

Once a workload characterization is performed collecting events using counting mode and following the methodology outlined in Chapter 3, a subset of events can be shortlisted for sampling and hot spot analysis as demonstrated in the upcoming [Chapter 5].

5 Performance Analysis on Neoverse N1: Case Study

In this section, we will demonstrate how to 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. We present an 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".

Case Study: DynamoRIO Strided Benchmark

For our case study, we run the Stride Benchmark from the DynamoRIO tests. [Source Reference https://github.com/DynamoRIO] 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.

Experiment Setup

Feature	****
Platorm	Noeverse N1 Software Development Platform(N1SDP)
Frequency	2.6 GHz
OS	Ubuntu 20.04.2 LTS

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

Observation Note (Table 1): The obtained IPC of 0.22 is much lower than those measured for most workloads on the Neoverse N1. For comparison, a large benchmark like SPEC CPU(r) 2017[11] (estimated) workloads that are intended to stress the system manages to achieve at least an average IPC > 1 on this CPU. The maximum achievable IPC is 4 on Neoverse N1, which is typically achieved by small & heavily optimized kernels rather than large applications.

Cycle Accounting

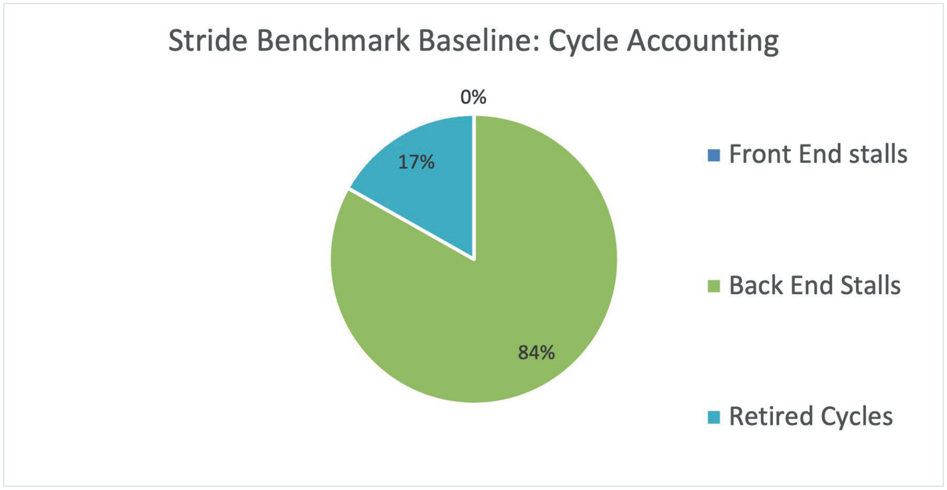


Figure 8: Stride Benchmark: Cycle Accounting

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.

Observation Note (Figure 8): 83% of the total cycles are stalled in the Back End and 0% are stalled in the Front End, which accounts for 83% of the overall execution time wasted by pipeline stalls.

As the workload is heavily Back End bound, we will now drill down the Back End events one by one. We'll first look at cache performance and instruction mix to see if the workload is core or memory bound. We already have a hint that it could be memory bound from looping pointer chase code which essentially performs accesses to CPU caches in this case. Next, we will look at the cache performance and instruction mix since they can reveal if something has caused our application to become core-bound or if we have cache performance issues.

D-Cache Effectiveness

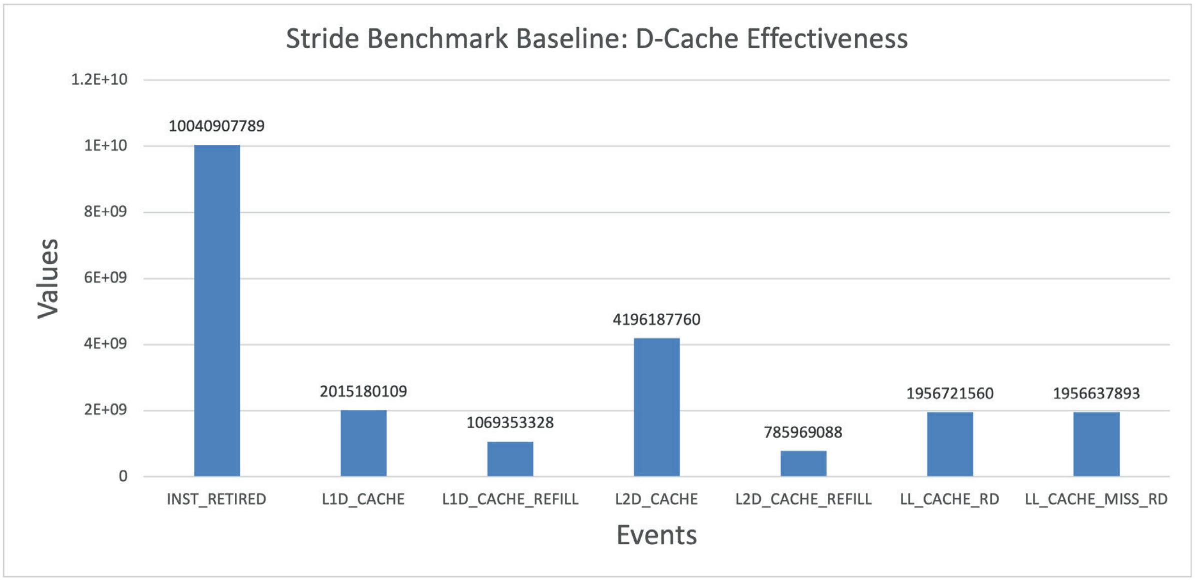


Figure 9: Strided Benchmark: D-Cache Effectiveness

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.

Let us look at the instruction mix next to see the percentage of memory instructions being executed.

Instruction Mix

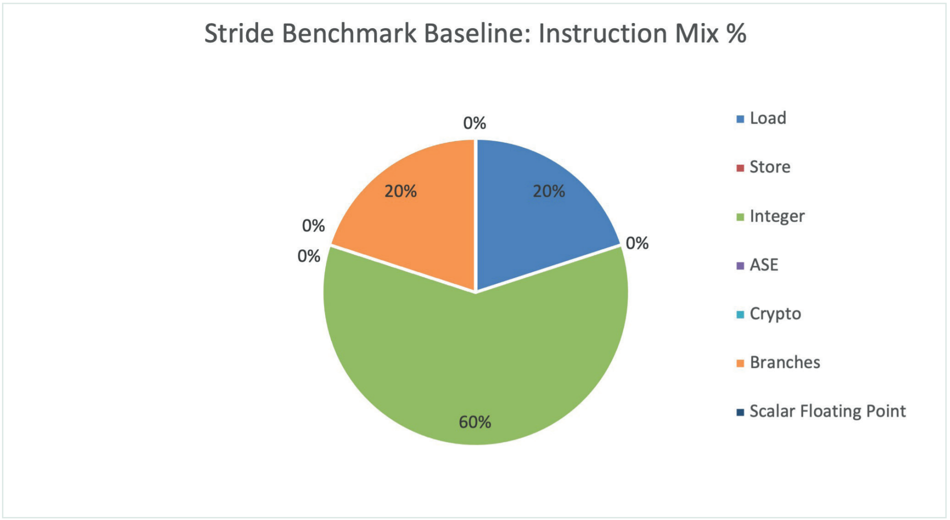


Figure 10: Strided Benchmark: Instruction Mix

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

Table 4: Instruction Mix Metrics

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.

Branch Effectiveness

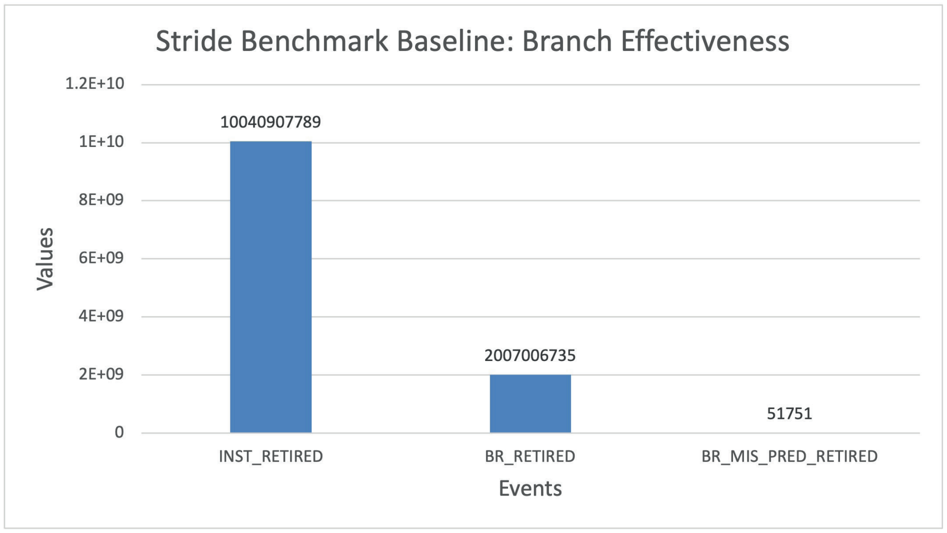


Figure 11: Stride Benchmark: Branch Effectiveness

To evaluate the branch prediction effectiveness, mis-prediction rates and MPKI metrics are derived below in Table 5:

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

Table 5: Branch Effectiveness Metrics

Observation Note (Figure 11): Branch mispredictions look negligible for this workload. We would have seen Front End stalls otherwise; it's as expected.

As we have identified the workload is memory bound, let us also look at how the TLBs are performing. We would not expect L1I TLB performance issues as we have negligible Front End Stalls.

TLB Effectiveness

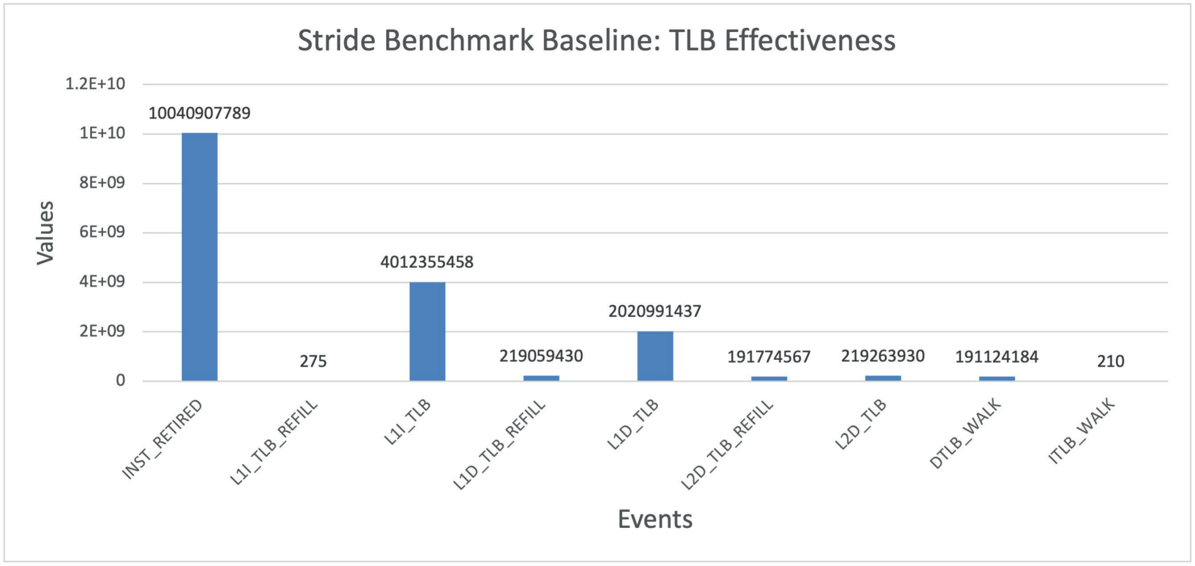


Figure 12: Stride Benchmark: TLB Effectiveness

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.

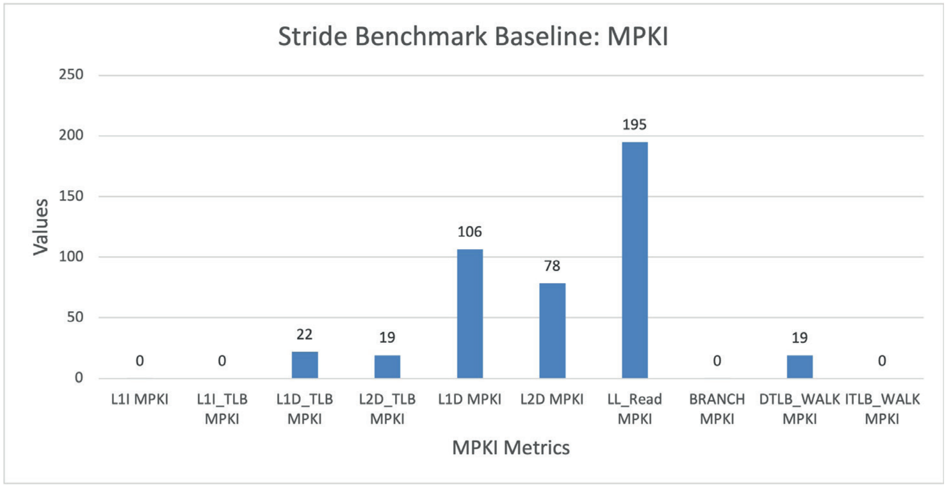


Figure 13: Stride Benchmark: MPKI

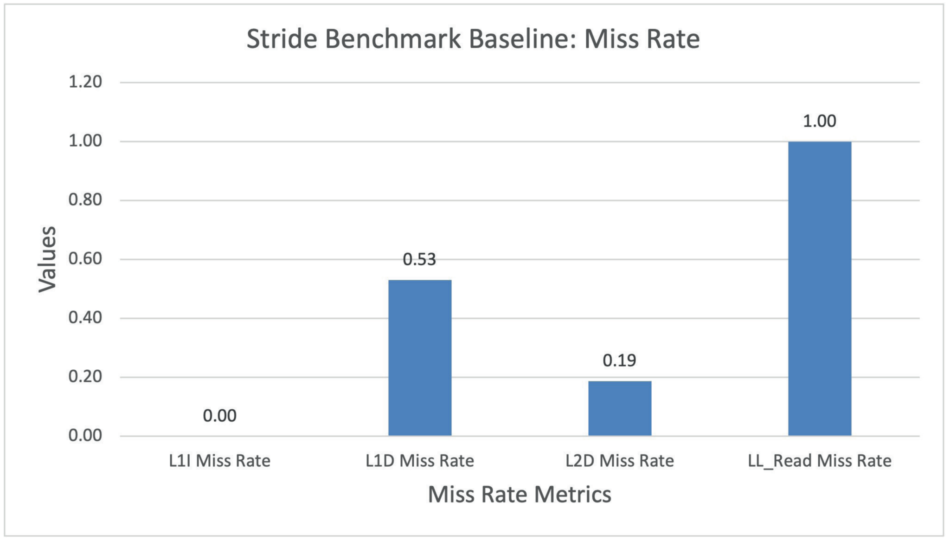


Figure 14: Stride Benchmark: Miss Rate

Workload Execution Remarks

The strided micro benchmark is poorly performing with a very low IPC of 0.22, as measured on the N1SDP platform. From the characterization data, 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.

The characterization evidence supports the workload behaviour (as mentioned in the test source code remarks) that there is a memory bottleneck in our system and we should next investigate how to address it.

Events for Hotspot Analysis

To identify the code execution bottlenecks for optimizing the code, we should further deep dive into two aspects: Back End stalls and hierarchical D-Cache events. For further investigation, the short listed events for hot spot analysis apart from INST_RETIRED and CPU_CYCLES include:

- L1D_CACHE
- L1D_CACHE_REFILL
- L2D_CACHE
- L2D_CACHE_REFILL
- LL_CACHE_RD
- LL_CACHE_MISS_RD

Phase 2: Hot spot Analysis using Perf Event Sampling Mode

We collected perf sample data for all the short listed events for hot spot analysis and studied the annotated disassembly code.

Stride Benchmark Source Code

"'

```
#include <stdint.h>
#include <string.h>
#include <iostream>

#define MEM_BARRIER() __asm__ __volatile__("" ::: "memory")

int
main(int argc, const char *argv[])
{
      // Cache line size in bytes.
      const int kLineSize = 64;
      // Number of cache lines skipped by the stream every iteration.
      const int kStride = 7;
      // Number of 1-byte elements in the array.
      const size_t kArraySize = 16 * 1024 * 1024;
      // Number of iterations in the main loop.
      const int kIterations = 2000000000;
      // The main vector/array used for emulating pointer chasing.
      unsigned char *buffer = new unsigned char[kArraySize];
      memset(buffer, kStride, kArraySize);

      // Add a memory barrier so the call doesn't get optimized away or
      // reordered with respect to callers.
      MEM_BARRIER();
```

```

    int position = 0;

    // Here the code will pointer chase through the array skipping forward
    // kStride cache lines at a time. Since kStride is an odd number, the main
    // loop will touch different cache lines as it wraps around.
    for (int loop = 0; loop < kIterations; ++loop) {

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

        position += (buffer[position] * kLineSize);
        position &= (kArraySize - 1);
    }

    // Add a memory barrier so the call doesn't get optimized away or
    // reordered with respect to callers.
    MEM_BARRIER();

    std::cerr << "Value = " << position << std::endl;

    return 0;
}
```

"'

Figure 15: Stride Benchmark Source Code

Stride Benchmark Main Function Disassembly

"'

```
Disassembly of section .text:

00000000000009a0 <main>:
9a0:    a9be7bfd    stp x29, x30, [sp, #-32]!
9a4:    d2a02000    mov x0, #0x1000000              // #16777216
9a8:    910003fd    mov x29, sp
9ac:    a90153f3    stp x19, x20, [sp, #16]
9b0:    97ffffd0    bl  8f0 <_Znam@plt>
9b4:    d2a02002    mov x2, #0x1000000              // #16777216
9b8:    aa0003f4    mov x20, x0
9bc:    528000e1    mov w1, #0x7                    // #7
9c0:    97ffffd4    bl  910 <memset@plt>
9c4:    52928001    mov w1, #0x9400                 // #37888
9c8:    52800013    mov w19, #0x0                       // #0
9cc:    72aee6a1    movk    w1, #0x7735, lsl #16
9d0:    3873ca82    ldrb    w2, [x20, w19, sxtw]
9d4:    71000421    subs    w1, w1, #0x1
9d8:    0b021a73    add w19, w19, w2, lsl #6
9dc:    12005e73    and w19, w19, #0xffffff
9e0:    54ffff81    b.ne    9d0 <main+0x30>   // b.any
9e4:    b0000094    adrp    x20, 11000 <__FRAME_END__+0x10298>
9e8:    d2800102    mov x2, #0x8                    // #8
9ec:    90000001    adrp    x1, 0 <_init-0x8b0>
9f0:    91302021    add x1, x1, #0xc08
9f4:    f947fa94    ldr x20, [x20, #4080]
9f8:    aa1403e0    mov x0, x20
9fc:    97ffffd5    bl  950 <
     ↪_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l@plt>
a00:    2a1303e1    mov w1, w19
a04:    aa1403e0    mov x0, x20
a08:    97ffffde    bl  980 <_ZNSolsEi@plt>
a0c:    97ffffbd    bl  900 <
     ↪_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@plt>
a10:    52800000    mov w0, #0x0                    // #0
a14:    a94153f3    ldp x19, x20, [sp, #16]
a18:    a8c27bfd    ldp x29, x30, [sp], #32
a1c:    d65f03c0    ret
```

"'

Figure 16: Stride Benchmark Main Function Disassembly

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 introduce skid 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.

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

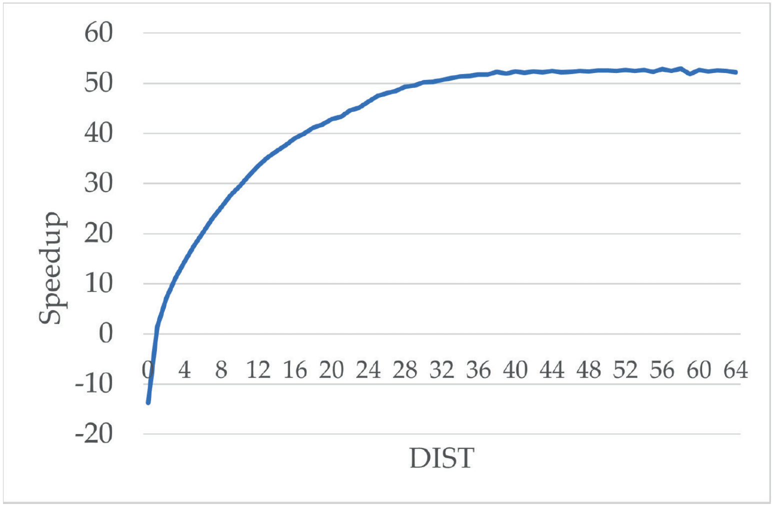


Figure 17: Speedup obtained for different prefetch distances

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:

```
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.

Final Summary

This case study has demonstrated how to use hardware PMUs of Neoverse N1 core or workload characterization based on the selected PMU events from [Chapter 2] and following the methodology in [Chapter 3]. Though this example was fairly straight-forward it demonstrates a methodology for isolating the cause of performance bottlenecks, correcting the issue, and verifying that the optimizations corrected the behavior causing the performance degradation. This basic flow may be applied to more sophisticated workloads to analyze performance issues.

Glossary

Term	Meaning
CMO	Cache Maintenance Operations
CPU	Central Processing Unit
LSU	Load Store Unit
MMU	Memory Management Unit
PE	Processing Element
PMU	Performance Monitoring Unit
SiP	Silicon Provider
SLC	System Level Cache
TLB	Translation Lookaside Buffer

Acknowledgements

We would like to thank Rodney Schmidt, Andrea Pellegrini, Frederic Piry, Michael Williams, Al Grant, James Greenhalgh, Mike Schinzler, Mark Rutland and Tamar Christina for their guidance and review of this paper.

External References

1. Arm®, “Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile Documentation”, https://developer.arm.com/docs/ddi0487/latest

2. Arm®, “Arm® Neoverse N1 Technical Reference Manual”, https://developer.arm.com/documentation/100616/latest

3. Arm®, “Arm® Neoverse N1 PMU Guide”, https://developer.arm.com/documentation/PJDOC-466751330-547673/r4p1/

4. A. Pellegrini et al., “The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC,” in IEEE Micro, vol. 40, no. 2, pp. 53-62, 1 March-April 2020, doi: 10.1109/MM.2020.2972222.

5. Arm®, “Arm® Neoverse™ N1 Software Optimization Guide Documentation”, https://developer.arm.com/docs/swog309707/a

6. M. Williams, “Statistical Profiling Extension for ARMv8-A”, https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/statistical-profiling-extension-for-armv8-a, 2017

7. “Linux perf Tool Wiki”, https://perf.wiki.kernel.org/index.php/Main_Page

8. “Linux perf Examples”, https://www.brendangregg.com/perf.html

9. “Arm PMU Event Repository”, https://github.com/ARM-software/data

10. “Dynamo RIO Stride Benchmark”, https://github.com/DynamoRIO/dynamorio/blob/master/clients/drcachesim/tests/stride_benchmark.cpp

11. “SPEC CPU2017”, https://www.spec.org/cpu2017