EECE7352: Computer Architecture Project Report

"Cache Timing Analysis on x86 and ARM Architectures"

Instructor: Dr. David Kaeli

Group Members:

Nikhil Anil Prakash
Christina Ann Pramode
Ruchit Jayesh Dharji

Abstract

As processor architectures become increasingly diverse, understanding the latency differences across memory hierarchies on x86 and AArch64 platforms is critical for optimizing performance in cross-platform applications. This study provides a comprehensive analysis of access latencies at each cache level (L1, L2, and L3) and main memory (SD-RAM), aiming to quantify these latencies and identify key architectural features driving the observed differences. By delving into the design nuances of x86 and AArch64 architectures, the study sheds light on the impact of factors such as cache organization, associativity, and prefetching mechanisms. Furthermore, it explores potential optimization strategies tailored to mitigate latency bottlenecks in memory-bound applications, offering valuable insights for developers and system architects striving to enhance performance across heterogeneous computing platforms.

1 Introduction

1.1 Purpose

This study focuses on delivering a detailed comparative analysis of memory access latencies across various levels of the memory hierarchy—L1, L2, L3 caches, and main memory (SD-RAM)—on both x86 and AArch64 architectures. With the growing importance of cross-platform applications, understanding these latency differences is essential for identifying performance bottlenecks and devising strategies to optimize memory-bound workloads.

1.2 Background

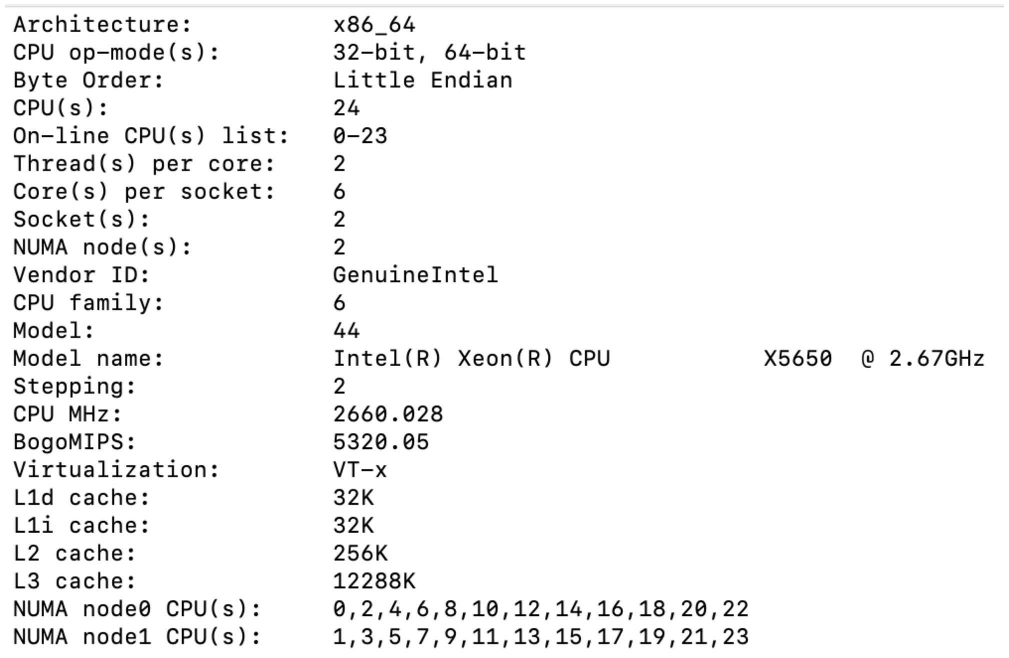
As computational systems continue to evolve, the growing diversity in processor architectures has a profound impact on application performance, especially in the context of memory operation efficiency. The speed at which memory operations are handled directly influences overall system responsiveness and workload execution. Understanding these variations in memory access behavior across different architectures is critical for optimizing performance, particularly in heterogeneous and cross-platform environments where architectural nuances can lead to significant performance disparities.

1.3 Scope

The scope of this study encompassed systematic measurements of cache hits and misses across all levels of the memory hierarchy—L1, L2, L3 caches, and SD-RAM—using meticulously tailored memory access patterns. This methodical approach ensured precise quantification of latency behaviors, providing a clear understanding of how different memory layers interact under various access scenarios. By isolating and analyzing these patterns, the study offers critical insights into the efficiency of cache utilization and memory access mechanisms, forming a robust foundation for performance optimization in diverse computational environments.

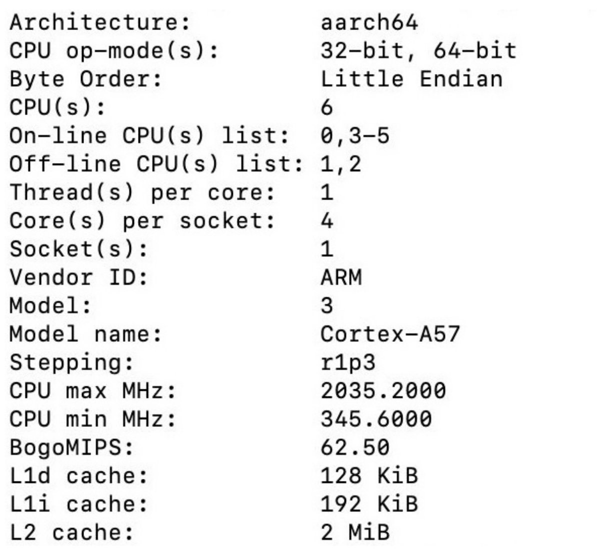
2 Tools and Environment

The study was conducted on COE Linux systems, which include systems with both x86 and AArch64 architectures, providing a versatile environment for cross-platform analysis. For precise latency measurements, the study utilized x86 Time Stamp Counters (TSC), along with equivalent tools available on AArch64, ensuring accurate and consistent timing data across both architectures. This approach allowed for high-resolution measurement of memory access latencies, essential for the comparative analysis of cache and memory performance on the two platforms. The CPU specifications for the respective processors, obtained using the 'lscpu' command, are shown in Figures 1 and 2.



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
Stepping:              2
CPU MHz:               2660.028
BogoMIPS:              5320.05
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
```

Figure 1: x86 CPU Specifications



```
Architecture:          aarch64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                6
On-line CPU(s) list:   0,3-5
Off-line CPU(s) list:  1,2
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Vendor ID:             ARM
Model:                 3
Model name:            Cortex-A57
Stepping:              r1p3
CPU max MHz:           2035.2000
CPU min MHz:           345.6000
BogoMIPS:              62.50
L1d cache:             128 KiB
L1i cache:             192 KiB
L2 cache:              2 MiB
```

Figure 2: ARM CPU Specifications

3 Methodology

The program designed measures cache and memory access performance at different levels—L1, L2, L3 cache, and SDRAM—within COE x86 and AArch64 systems. It employs precise timing mechanisms to accurately record the time taken to access data from these various storage points, leveraging inline assembly to ensure high precision. This setup plays a critical role in identifying the efficiency of cache utilization and understanding the impact of cache architecture on overall system performance.

3.1 Buffer Value Initialization

To achieve precise control over program execution and data caching, we aim to initialize the program such that it begins execution at the exact start of a cache set. This requires the program's starting address to have its last 12 binary digits set to zero, ensuring alignment with the beginning of a cache line and enabling predictable caching behavior.
We plan to "pin" the cache to this initial address, so when the program fetches data, it will start at the top of the cache set. This approach ensures efficient data retrieval and optimal utilization of cache memory. The buffer sizes we are using are carefully matched to the system's cache sizes, maximizing cache efficiency and predictability in data handling. This configuration is critical for achieving high performance and deterministic behavior in memory-intensive operations.

3.1.1 x86 Architecture

The cache configurations of the system obtained through the command 'getconf -a — grep CACHE' are described below.
L1D Cache:

- Ways of Associativity: 8
- Number of Sets:
- Coherency Line Size: 64
- Size: 32 KB

L2U Cache:

- Ways of Associativity: 8
- Number of Sets:
- Coherency Line Size: 64
- Size: 256 KB

L3U Cache:

- Ways of Associativity: 16
- Number of Sets:
- Coherency Line Size: 64
- Size: 12 MB

3.1.2 ARM Architecture

The cache configurations of the system are described below.
L1D Cache:

- Ways of Associativity: 2
- Number of Sets: 256
- Coherency Line Size: 64
- Size: 32 KB

L2U Cache:

- Ways of Associativity: 16
- Number of Sets: 2048
- Coherency Line Size: 64
- Size: 2048 KB

It is observed that the ARM A57 Cortex CPU does not have an L3 cache. Consequently, further analysis is conducted within the constraints of the available hardware and its limitations.

3.2 Timer Mechanisms in ARM and x86 Architectures

A thorough understanding of timers and timing instructions in both x86 and ARM architectures is crucial for achieving accurate performance measurement. This knowledge ensures precise tracking of execution times, allowing for reliable comparisons and performance analysis across different platforms.

3.2.1 x86 Architecture

The CPUID instruction plays a critical role as a serializing mechanism, ensuring absolute precision in the execution order of instructions. It guarantees that all preceding instructions are fully executed before it begins, and no subsequent instructions are allowed to start until it completes, making it indispensable for accurately placing the RDTSC instruction in the instruction stream. The RDTSC instruction then reads the current value of the Time Stamp Counter (TSC) into registers, providing a high-resolution timing mechanism crucial for performance measurement. For even greater accuracy, especially in scenarios requiring precise timing of later stages of memory access, the RDTSCP instruction offers an enhanced capability. It not only reads the TSC but also ensures that all preceding instructions have fully completed before execution, making it the ideal choice for critical timing applications where exact synchronization of instruction completion is essential.

3.2.2 ARM Architecture

In ARM architectures, performance monitoring and precise timing are managed through a series of specialized system instructions that interact directly with the Performance Monitoring Unit (PMU). The mrs (Move Register to System) and msr (Move to System Register) instructions play a crucial role in this process. They are used to read from and write to the PMU control register (pmcr_el0), respectively. Initially, mrs fetches the current settings of the PMU control register into a general-purpose register. This value is then modified using the orr operation to set the least significant bit, which activates the Performance Monitor Counter. The modified value is written back into the PMU control register using msr, thereby enabling the cycle counter. To ensure that the timing measurements reflect the exact point in the execution, the isb (Instruction Synchronization Barrier) is employed. This barrier guarantees that all instructions preceding it are completed before any subsequent instruction is executed, crucial for preventing any pipeline or out-of-order execution effects from skewing the cycle count. This setup facilitates the direct reading of the cycle counter register (pmccntr_el0) using mrs, similar to how RDTSC operates in x86, providing a measure of the number of cycles elapsed.

3.3 Detailed Function Descriptions

3.3.1 Initialization Function - init()

The init() function prepares the environment for subsequent tests. It populates buffers representing different cache levels (L1, L2, L3) with random data to simulate typical operational conditions and prevent any caching optimizations that could skew results. Additionally, it sets up file streams for detailed logging of the performance data, and allocates memory for dynamic arrays that will store the timing results, ensuring efficient handling during the execution of tests.

3.3.2 SDRAM Access Measurement - doSDRAMTrace()

doSDRAMTrace() aims to measure the latency of data access from SDRAM by intentionally bypassing the cache. It utilizes the clflush instruction to invalidate specific cache lines, forcing data accesses to retrieve directly from SDRAM. For the ARM processor, this is done using the DC CIVAC instruction. Cycle counts for x86 are captured using RDTSC and RDTSCP before and after the data access, with CPUID ensuring that the measurements are serialized and accurate. For ARM, cycle counts are measured using the PMU and appropriate instruction synchronization barriers. This function provides a baseline for understanding the cost of memory accesses when caches do not contain the requested data.

3.3.3 Cache Access Measurements - doL1Trace(), doL2Trace(), doL3Trace()

These functions are dedicated to measuring access times for L1, L2, and L3 caches. They manipulate cache states to ensure that data is precisely placed in the intended cache level during the measurement. This is achieved through tailored memory access patterns and strategic use of cache flushing and filling operations. Like the SDRAM measurements, x86 uses RDTSCP and CPUID and ARM uses the PMU respectively for capturing precise cycle counts thus illuminating the efficiency of various cache levels.

3.3.4 Data Logging - write_times_to_file()

The write_times_to_file() function records all the collected data into files. This includes cycle times and data values associated with each memory access, formatted for ease of analysis. This logging is critical for later review, comparison of cache and memory performance across different system configurations, and for graphing the results.

4 Analysis

The C codes for the x86 and ARM architectures are run using the command - 'taskset -c 0 executable_name'. The code generates three output files: "SDRAM_cache_hits.txt", "L1_cache_hits.txt", and "L2_cache_hits.txt". These files are further used as inputs for plotting the graphs in the upcoming section.

4.1 x86 Architecture

The graph in Figure 3 illustrates the L1, L2, and L3 cache hit times for the x86 machine deployed on AWS. The results indicate that L1 cache hits occur in approximately 58 cycles, L2 cache hits take around 66 cycles, and L3 cache hits take roughly 77 cycles. These measurements confirm that the program is accurately capturing cache hit times for each level of the memory hierarchy.

To measure the L1 cache hit time, a cache line is initially loaded and then reloaded, with the reload time being recorded. This straightforward approach ensures that the data access is served directly from the L1 cache.

For L2 cache hit time, the process involves targeting a specific cache set in L1. Initially, data is loaded into the CPU, making it available in all three cache levels. Then, new data is deliberately loaded into the same L1 set, overwriting the original data. When the overwritten data is accessed again, it is retrieved from L2, allowing the measurement of the L2 hit time. The method for measuring L3 cache hit time follows the same logic as for L2. After data in L2 is overwritten, subsequent access retrieves the data from L3, providing an accurate measurement of L3 hit time. This systematic approach ensures reliable latency measurements across the different cache levels.

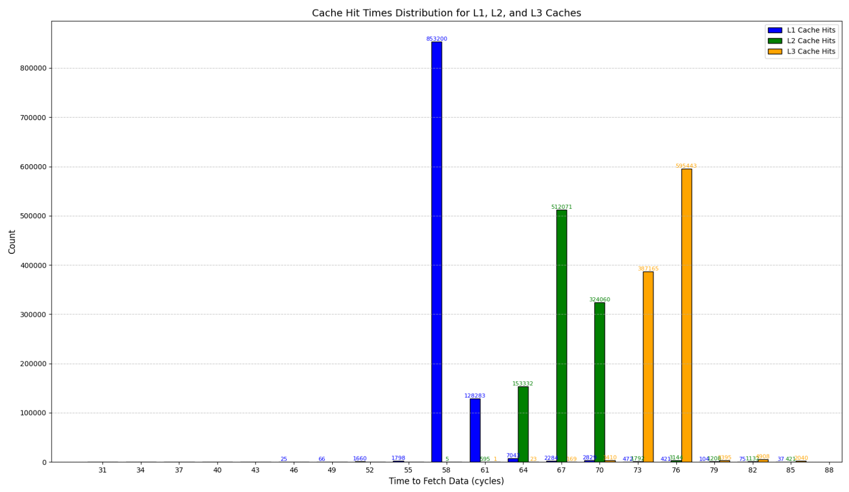


Figure 3: AWS x86 - Cache Hit Times Distribution

The graph in Figure 4 shows that on the COE x86 system, there is some overlap between the L1 and L2 cache hit times. Due to the small difference in their hit times, accurately sampling these caches is challenging. However, the measurements still allow us to estimate the L1 hit time at approximately 39 cycles and the L2 hit time at around 46 cycles.

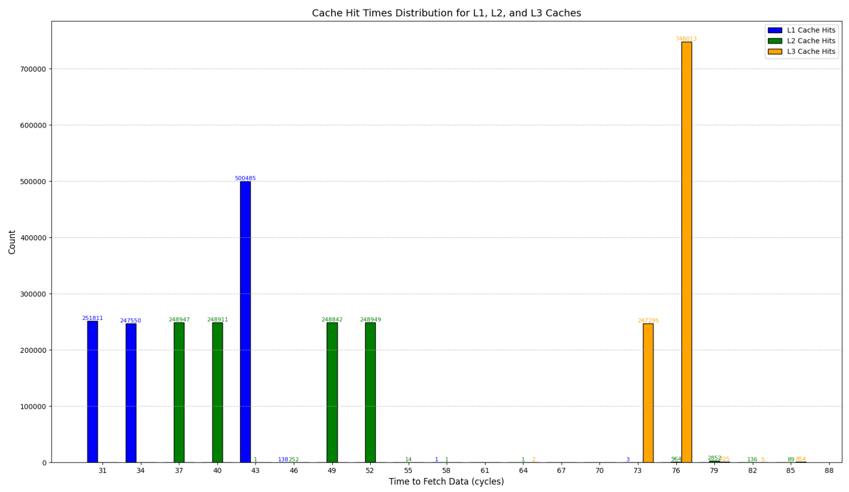


Figure 4: COE x86 - Cache Hit Times Distribution

In contrast, in Figure 5 the L3 hit time is more distinctly measured, showing a value of approximately 77 cycles. Additionally, the SDRAM hit time on the COE x86 system is clearly observed to be around 180 cycles.

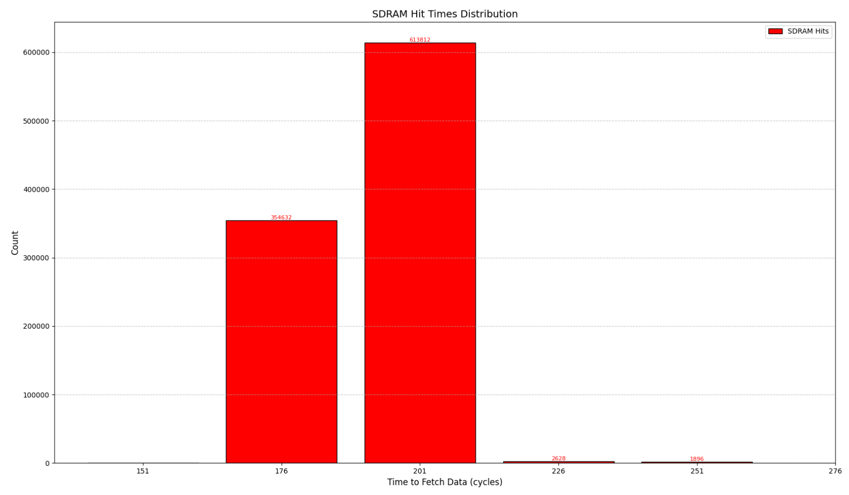


Figure 5: COE - SDRAM Hit Times Distribution

4.2 ARM Architecture

4.2.1 SDRAM

Figure 6 illustrates SDRAM fetches organized into buckets, each with a size of 25 cycles, ranging from 300 to 900 cycles. Notably, approximately 92% of the 1 million samples (920,000) exhibit a fetch period of around 450 cycles. This strongly suggests that the code is effectively targeting memory, as access times beyond 100 cycles would bypass the L1, L2, and L3 caches.

To achieve this result, we deliberately flush an arbitrary cache line, compelling the CPU to retrieve the data directly from SDRAM. The graph reflects the outcome of this procedure, averaged over 1 million samples, showcasing the consistent fetch behavior.

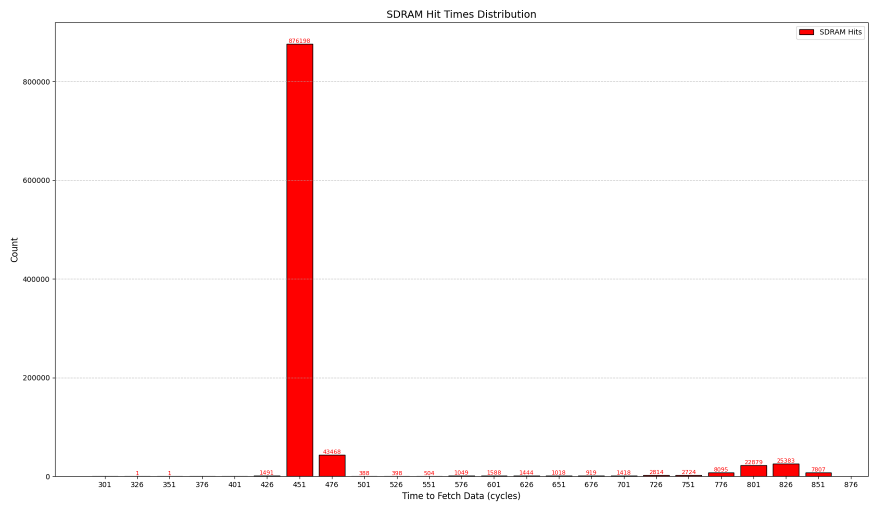


Figure 6: ARM - SDRAM Hit Times Distribution

4.2.2 L1 and L2 Cache

Figure 7 shows the time taken by Y samples for a bucket size of 3, plotted on the X-axis. Notably, approximately 990,000 out of 1 million samples take 49 cycles, which is the shortest access time observed on this system and falls within the typical range of L1 cache hit times.

For the L2 cache, which is the last level cache on this machine, it has a capacity of 2 MiB and consists of 2,048 sets. This architecture results in slightly more dispersed timing hits. The weighted average of L2 cache hit times is 77 cycles, approximately 25 cycles longer than L1, which aligns with expected behavior for hierarchical cache access.

To ensure accurate L1 cache hits, we first select a specific set within the L1 cache, completely fill it based on its associativity, and then fetch the most recently used address. This approach optimizes cache performance by leveraging the temporal locality of the data.

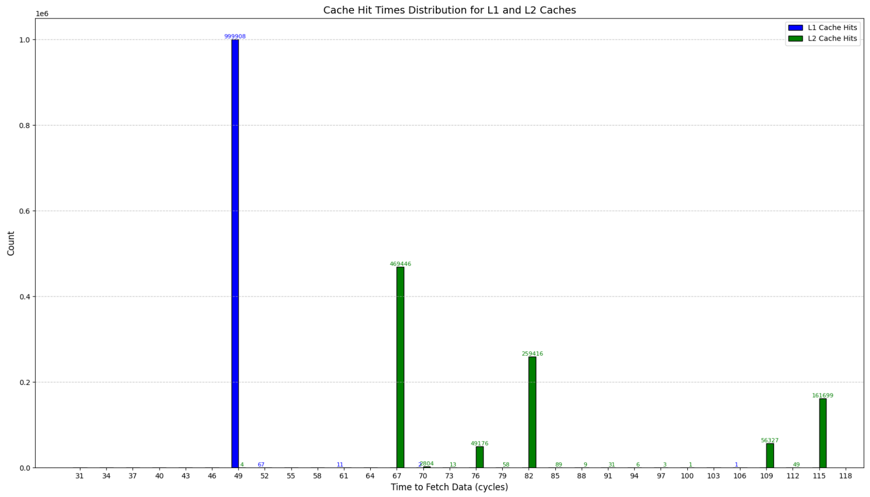


Figure 7: ARM - L1 & L2 Hit Times Distribution

5 Conclusion

This study successfully quantified and analyzed memory access latencies across the memory hierarchies of x86 and AArch64 architectures, shedding light on the architectural features and design choices that drive latency variations. A C program was developed for each architecture to perform these measurements. The findings revealed significant differences in cache organization, associativity, and prefetching mechanisms between the two architectures, which influence their efficiency in handling memory-bound workloads. These insights underscore the importance of tailoring application optimization strategies to the specific memory architectures of target platforms. By leveraging the strengths of each architecture, developers can mitigate latency bottlenecks, improve cache utilization, and enhance the performance of memory-bound applications.

References

[1] G. Paoloni, ”How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures,” Intel Corporation, Sep. 2010.

[2] Intel Corporation, ”Intel® 64 and IA-32 Architectures Software Developer Manuals,” Intel Corporation

[3] “How to Use Performance Monitor Unit(PMU) of 64-bit ARMv8-A in Linux,” Github.io, Mar. 02, 2016. `https://zhiyisun.github.io/2016/03/02/How-to-Use-Performance-Monitor-Unit-(PMU)-of-64-bit-ARMv8-A-in-Linux.html`