Benchmarking and Analysis of Software Data Planes

21-Dec-2017

Maciek Konstantynowicz
mkonstan@cisco.com

Patrick Lu
patrick.lu@intel.com

Shrikant M. Shah
shrikant.m.shah@intel.com

Table of Content

Space intentionally left blank.

"*Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*"

"*Write a paper promising salvation, make it a "structured" something or a "virtual" something, or "abstract", "distributed" or "higher-order" or "applicative" and you can almost be certain of having started a new cult.*"

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger Wybe Dijkstra,
"EWD896: On the nature of Computing Science",
"EWD 709: My hopes of computing science",
"EWD249: Notes On Structured Programming", page 7.

"Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte."
"I made this letter longer, only because I have not had the leisure to make it shorter."
Blaise Pascal, Provincial Letters: Letter XVI (4 December 1656).

Legal Statements from Intel Corporation

FTC Disclaimer

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Please refer to the test system configuration in *Section 13 Appendix: Test Environment Specification* and *Section 14. Appendix: Benchmarking Tools Use Guidelines*.

FTC Optimization Notice
Optimization Notice: Intel's compilers and DPDK libraries may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

'Mileage May Vary' Disclaimer
Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks

Estimated Results Disclosure
Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Dependencies Disclosure
Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

Trade mark Notice
Intel, Xeon, the Intel logo, and other Intel technologies mentioned in this documents are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Other Disclaimers
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

1 Introduction

1.1 Preface

There is a growing expectation that Internet network services, underpinned by network functions, need to evolve and become dynamically instantiated in locations and at capacity governed by continuously changing and moving service users' demands. At the same time, these network functions are required to dynamically and seamlessly handle IP host/route reachability, end-point security and other fundamental aspects of network services, delivering them at line-rate speeds with data plane performance being the key factor. All of these requirements cannot be achieved in practice with dedicated Hardware-based network service implementations, opening an opportunity for a new generation of Software-based network functions.

The problem of dynamically and efficiently instantiating Software based Internet services at a moment notice is not new and has been already tackled by cloud-native architectures and solutions. It is then natural to consider a cloud-native approach for the new Software network functions, so that they can benefit from all of the main cloud-native properties, including:

1. Cloud portability - public, private, hybrid, multi-cloud;
2. Extensibility - modular, pluggable, hookable, composable;
3. Self-healing - automatic placement, restart, replication, scaling;

However, for the cloud-native model to work here, the new Software network functions must above all provide the high-performance data plane. This is possible only and only if the network functions are purposefully coded and optimized for compute platforms (servers and any CPU based devices), addressing their Input / Output and memory characteristics. It is the "compute-native" network function SW code that is essential to achieve efficiency, high throughput (packets and bandwidth) and low latency cloud-native data plane, in order to meet the expectations of wide portability and applicability to many Internet network service use cases.

This technical paper introduces the main concepts underpinning a systematic methodology for benchmarking and analysis of compute-native network SW data plane performance running on modern compute server HW. It applies first principles of computer science to performance measurements, describing involved aspects of SW-HW integration and focusing on the optimal usage of compute HW resources critical to the data plane performance.

Authors believe that following described methodology and defined performance metrics should enable the industry community to arrive to a well-defined benchmarking standard and apples-to-apples comparison between different network data plane SW applications. Furthermore, by accepting these metrics, and using them as a "feed-back loop" for continuous native code optimizations, the community can continue the drive towards breaking the barrier of One Terabit SW network data plane speeds per single 2RU server and increasing the density of data plane network functions and services per unit of compute.

1.2 Motivation

Analyzing and optimizing performance of software applications has become increasingly challenging due to the overall computing system complexities involved in their execution. The compute system stack consisting of layers of Compute Hardware, Operating System and Software Application, the interactions between these layers and the continuous rapid technological advancements across the layers, all make the application performance analysis and optimizations an intricate task.

Performance optimization challenge becomes ever greater if the aim is to aid in developing performance centered programming patterns and techniques with broad applicability, great appeal and designed for longevity.

The rising wave of Software Defined Network (SDN) services invading The Internet and Telecom industry by storm, with associated drive towards proliferation of Network Function Virtualization (NFV), calls for development of a methodical approach for analyzing (and optimizing) the performance of network functions implemented in software.

Network functional area most sensitive to performance optimizations is the data plane. This is due to the two main properties of network data plane that are difficult to address with compute systems:

i) Extremely high bandwidth demands for Input / Output operations;

ii) Tight and strict time budget for completing packet processing operations.

Translating these properties into requirements imposed on modern network data planes to make them handle Gigabit Ethernet rates:

i) Input / Output bandwidth: 10 Gigabit/sec for 10GbE interface, 100 Gigabit/sec for 100GbE interface, and in the future Terabit rates;

ii) Per packet processing time budget dictated by requirement to process rates of Millions of packets/sec (Mpps): less than 67 nanoseconds (nsec) to handle rates up to 14.88 Mpps for 10GbE line rate, less than 6.7 nsec to handle rates up to 148.8 Mpps for 100GbE line rate.

This technical paper aims to help to address these challenges by defining a methodology for systematic performance benchmarking and analysis of compute-native Network Function data planes executed on Commercial-Off-The-Shelf (COTS) servers, using available open-source measurement tools. The following key aspects are covered:

i) Description of modern server hardware resources vital for executing network applications;

ii) Software interactions with hardware and optimizations of software-hardware interface;

iii) Evaluation of common compute system bottlenecks encountered when benchmarking network data planes including processor, memory and network I/O resources.

In order to illustrate applicability of defined methodology and proposed measurement tools, the paper reports benchmarking results and their analysis for a number of example network data plane applications – DPDK, FD.io VPP, OVS-DPDK – running on modern high-performance servers.

1.3 Document Structure

The paper is organized as follows.

Section 2. Target Applicability describes how the proposed benchmarking methodology and analysis apply to Network Functions (NF) designs and deployment use cases; furthermore, it specifies sample NF applications used for benchmarking and analysis in this paper.

Section 3. NF Benchmarking draws differences and similarities between benchmarking compute and networking data plane software workloads, and derives a set of baseline performance metrics for NF data plane evaluations.

Section 4. NF Performance Tests and Results Analysis explains the basic principles of proposed performance analysis methodology capturing both utilization efficiency of HW resources and network performance metrics, illustrating them with analysis of sample NF results.

Section 5. Intel x86_64 – Performance Telemetry and Tools walks thru the telemetry points in Intel® Xeon® machines, including CPU micro-architecture, I/O and memory sub-systems vital for executing NF data plane functions; describes used measurement tools.

Section 6. Compute Performance Analysis using Intel TMAM is dedicated to a detailed performance analysis of the benchmarked workloads using Top-down Micro Architecture Method (TMAM).

Section 7. Memory Performance Analysis covers memory performance metrics, with analysis of results measured for tested NF applications, and Software tool used.

Section 8. PCIe Performance Analysis delves into PCIe transactions and bandwidth consumed by Ethernet frames, description of Intel® Direct Data IO (DDIO) technology critical to efficient NF data planes; followed by analysis of PCIe performance measurements for tested NF workloads, and Software tool used.

Section 9. Inter-Core and Inter-Socket Communication briefly reviews aspects related to multi-core and multi-socket configurations.

Section 10. Performance Tuning Tips highlights common techniques for achieving peak performance of NF data plane applications executed on the prescribed platforms.

Section 11. Conclusions summarizes the applicability of proposed benchmarking methodology and analysis to evaluate NF data plane applications, compare them and identify areas of code improvement.

Sections 12. to 18. include references, test environment specifications, deeper levels of TMAM analysis and index of figures, tables and equations.

2 Target Applicability

2.1 Network Function Topologies

Described benchmarking and analysis methodology applies to a set of Network Function (NF) data plane packet path and topology scenarios including packet processing and forwarding between: i) physical interfaces, ii) physical interfaces and multiple Virtual Machines, and iii) physical interfaces and multiple Containers.

2.2 Baseline Packet Path

The baseline data plane design benchmarked in this paper includes the NF application running as a user application on a compute host, processing and forwarding packets between the physical network interfaces hosted on the Network Interface Cards (NICs) within the system. Linux is used as a host Operating System, to manage access to available compute resources. NF application is running in Linux user-mode, taking direct control of the NIC devices, and enabling it to receive and transmit packets through the physical network interfaces with minimal involvement of Linux kernel in data plane operation.

The baseline NF data plane benchmarking topology is shown in *Figure 1*.

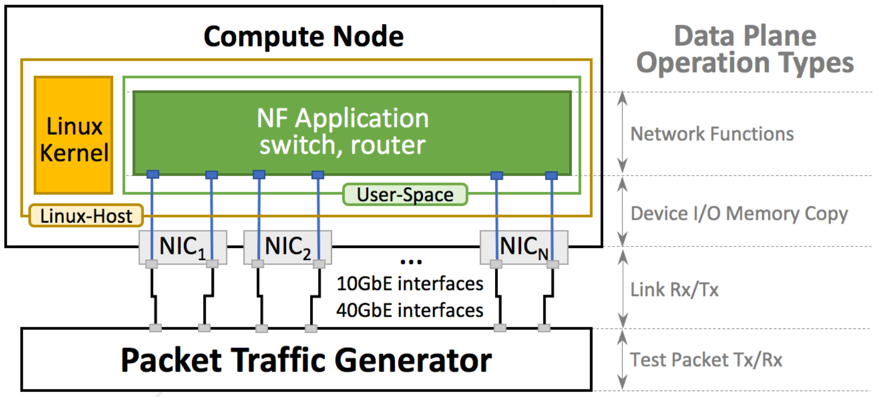


Figure 1. Baseline NF data plane benchmarking topology.

In order to measure the actual performance of evaluated sample NF applications, number of different hardware configurations and hardware resource allocations are employed, including the scaled-up multi-thread and multi-core layouts.

Presented baseline setup has two main functional parts, i) driving the physical network interface (physical device I/O) and ii) packet processing (network functions). Both parts are present in majority of deployments, hence their performance and efficiency can be used as a baseline benchmarking reference for evaluating compute native scenarios. Other more complex NF designs involve adding virtual network interfaces (virtual I/O, memory-based) and more network functions, providing richer composite functionality but at the same time using more compute resources. In other words, the baseline NF benchmarking data described in this paper can be treated as an upper ceiling of NF application capabilities.

Sample Virtual Machine (VM) and Container based NF designs are briefly described in the following sections. Benchmarking, analysis and optimizations of those composite NF designs is subject to future study.

2.3 With Virtual Machines

A sample design with NF applications running in VMs and NF "service-chain" forwarding provided by a common virtual switch NF application running in user-mode is shown in *Figure 2*.

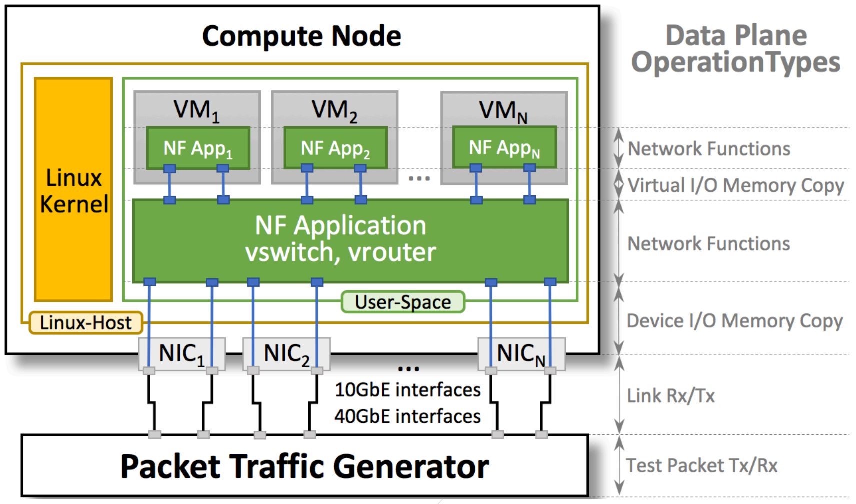


Figure 2. NF service topologies with NF apps in VMs, connected by vswitch, vrouter.

2.4 With Containers

A sample design with NF running in Containers and NF "service-chain" forwarding provided by a common virtual switch NF application running also in Container is shown in *Figure 3*.

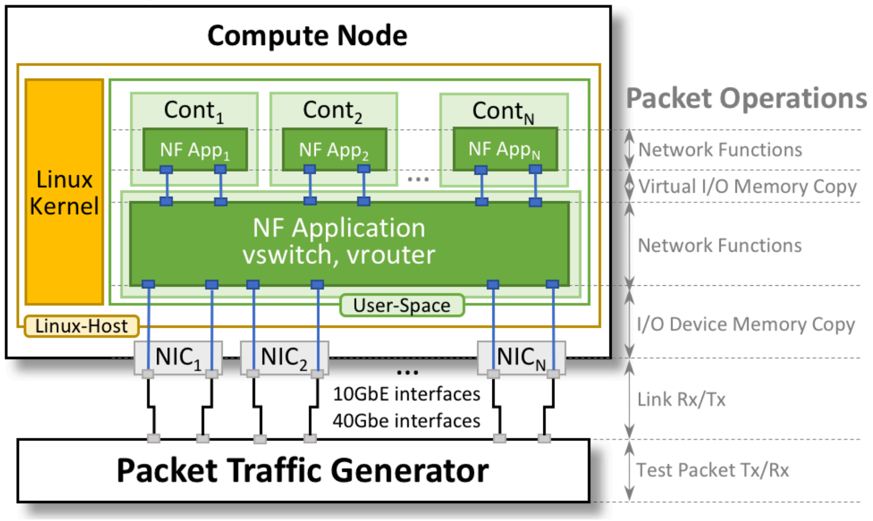


Figure 3. NF service topologies with NF micro-apps in Containers connected by vswitch, vrouter.

A variation of NF "service-chain" for NF applications running in Containers using FD.io memif virtual interface instead of forwarding thru a virtual switch NF, is shown in *Figure 4*. Note the smaller number of virtual interface and packet processing "hops" involved in data plane spanning the same number of NF applications.

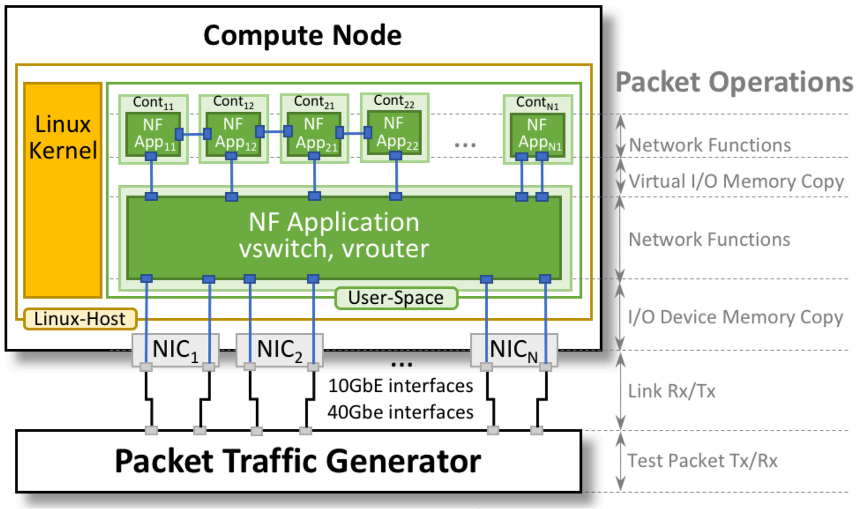


Figure 4. NF service topologies with NF micro-apps in Containers connected directly and by vswitch, vrouter.

2.5 Baseline vs. VMs vs. Containers

From performance analysis and benchmarking perspective, there is one common element stands out in VM and Container based packet path designs when compared to the baseline NF design. It is the need to use a performant fast virtual interface interconnecting the NF applications within the compute machine. In most cases this involves memory copy operation(s) that significantly impact the NF data plane performance. Good examples of virtual interfaces optimized for that purpose are VM Qemu vhost-user and FD.io VPP Memif for Containers and user-mode processes. More distributed NF designs and topologies that involve multiple number of compute machines are just combinations of described NF designs, making the performance analysis described in this paper directly applicable to those cases.

3 NF Benchmarking Metrics

3.1 Measuring Computer System Performance

Assessing computer system performance is not a trivial task due to complexity of modern compute systems and a variety of performance improvement techniques used in computer hardware designs. To address this the industry adopted the classic processor performance equation that defines execution time as the main and only complete and reliable measure of computer performance[1].

This paper proposes to use the same equation and the program execution time as the fundamental measure of NF application performance and efficiency.

3.2 Benchmarking Compute Applications

Performance of generic compute applications can be measured using the classic computer performance equation that defines the program execution time as a reliable measure of performance:

$$program\_unit\_execution\_time\ [sec] = \frac{\#instructions}{program\_unit} * \frac{\#cycles}{instruction} * cycle\_time$$

Equation 1. Classic computer performance equation.

The equation includes all the key factors that determine time to execute a program:

1. *#instructions/program_unit* – number of instructions per program_unit, or how big the executable program_unit is for a specific task.
2. *#cycles/instruction* – number of CPU core clock cycles per instruction (CPI), or how complex are those instructions and how well are they executed on specific CPU hardware; often expressed as a reciprocal metric – *#instructions/cycle* (IPC).
3. *cycle_time* [sec] – duration of a clock cycle measured in seconds, or how fast is the actual CPU hardware executing the instructions; represented by inverse metric of CPU core frequency = cycles/second.

Clearly it is not easy to translate the program unit variable to a modern complex compute application workload. That is where various benchmarking approaches, suites, and standards define a variety of program units, that are then applied to measure different compute systems, their respective sub-systems and operations. Examples of benchmarking suites include Standard Performance Evaluation Corporation (SPEC), CoreMark® (EEMBC benchmark), Princeton Application Repository for Shared-Memory Computers (PARSEC), NASA Advanced Supercomputing (NAS), and Stanford Parallel Applications for Shared Memory (SPLASH).

[1] "*Computer Organization and Design, The Hardware/Software Interface*" by David A. Patterson and John L. Hennessy, Section 1.6 Performance, ISBN: 978-0-12-407726-3.

3.3 Benchmarking NF Applications

Applying the classic computer performance equation to the Network Function application and substituting program unit with per network packet processing operations results in a modified performance equation:

$$packet\_processing\_time\ [sec] = \frac{\#instructions}{packet} * \frac{\#cycles}{instruction} * cycle\_time$$

Equation 2. Modified computer performance equation for NFV.

This approach, in essence, is treating the NF application workload running on a computer, as just another program. All generic computer science software workload efficiency and performance evaluation methodologies and best practices equally apply to NF workloads.

The Internet and packet networking world, on the other hand, evaluates performance of network devices (packet processing systems) by using a different set of metrics defined in IETF specifications RFC 2544[2] and RFC 1242[3], with major metrics including:

a) *packet throughput* measured in packets-per-second [pps];

b) *bandwidth throughput* measured in bits-per-second [bps];

c) *packet loss ratio* PLR;

d) *packet delay* (PD) and *delay variation* (PDV);

The natural unit of work in networking is a data packet.

Marrying both benchmarking worlds, computing with networking, and to enable simple apples-to-apples comparison between NF systems, a single data packet-centric program execution efficiency metric is proposed for benchmarking NF data plane packet processing – *#cycles/packet* (CPP):

$$CPP = \frac{\#instructions}{packet} * \frac{\#cycles}{instruction}$$

Equation 3. NF data plane efficiency equation binding CPP, IPP and IPC metrics.

Applying it to the modified performance equation:

$$packet\_processing\_time\ [sec] = CPP * cycle\_time\ [sec] = \frac{CPP}{CPU\_freq\ [Hz]}$$

Equation 4. NF computer performance equation with CPP.

And making a final connection, following is a formula binding the IETF benchmarking packet throughput metric and the CPP metric:

[2] RFC 2544, "Benchmarking Methodology for Network Interconnect Devices", March 1999, https://tools.ietf.org/html/rfc2544.

[3] RFC 1242, "Benchmarking Terminology for Network Interconnection Devices", July 1991, https://tools.ietf.org/html/rfc1242.

$$throughput\ [pps] = \frac{1}{packet\_processing\_time\ [sec]} = \frac{CPU\_freq\ [Hz]}{CPP}$$

Equation 5. Binding the packet Throughput [pps] and CPP benchmarking metrics.

CPP represents NF application program execution efficiency for a specific set of packet processing operations. Following sections show how the CPP metric can be put to effective use for comparing network workload performance across different packet processing scenarios, NF applications and compute platforms. To further characterize NF application efficiency underpinning CPP, a number of additional compute performance metrics are also described, with analysis of their applicability to benchmarking NF workloads.

Clearly it is hard to measure CPP on an individual packet basis in real-time high-performance NF system. Measurements reported in this paper use average values of CPP measured across packet flows undergoing the same packet processing operation.

3.4 Compute Resources Usage

Optimizing performance of a compute system usually involves going through an iterative process of analysis and tuning across involved Software and Hardware system components and layers. Network centric software applications exercise and stress multiple parts of the CPU micro-architecture, and the first order performance analysis is to establish which of these parts are top-level limiting hotspots and bottlenecks. This in turn translates into a set of basic questions and top-level performance and efficiency metrics:

1) **Packet processing operations on CPU core(s)**
    a. What is the efficiency of the NF software and compiler to perform specified packet operations – *How many instructions are executed per packet?*
    b. What is the instruction execution efficiency of an underlying CPU micro-architecture – *How many instructions are executed per CPU core clock cycle?*
2) **Memory bandwidth** – *What is the memory bandwidth utilization?*
3) **I/O bandwidth** – *What is the PCIe I/O bandwidth utilization?*
4) **Inter-socket transactions** – *What is the inter-processor cross-NUMA connection utilization?* (applicable for multi-socket machines)

Figure 5 below. depicts the high-level performance probing points related to above questions in the two-socket compute server based on Intel® Xeon® processor E5 v4 Family.

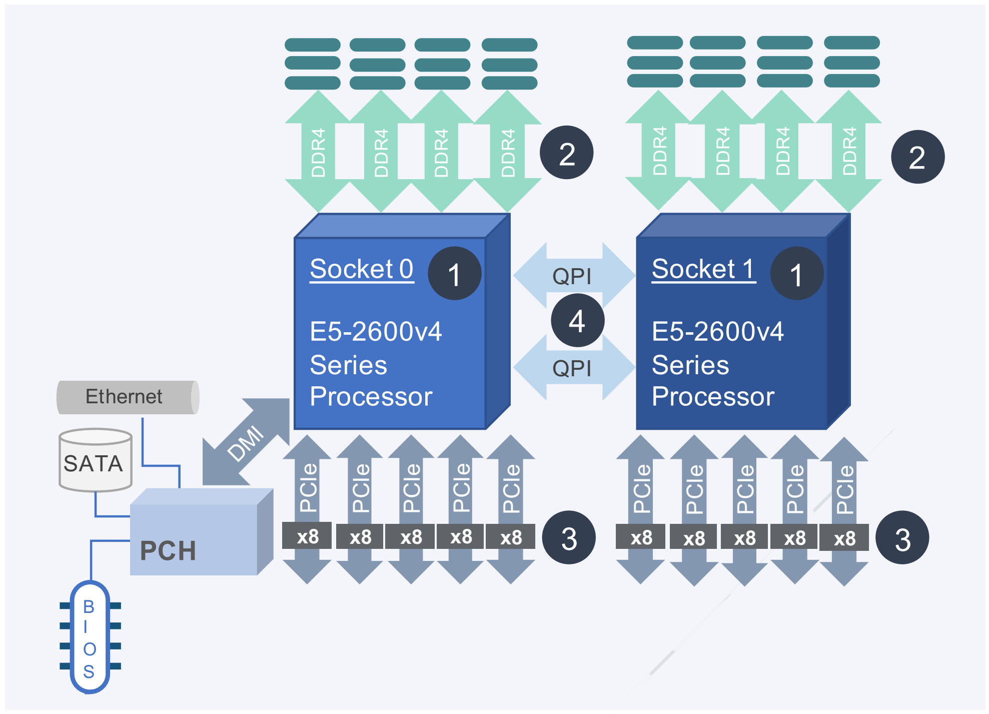


Figure 5. Points of high-level performance statistics in two-socket Intel® Xeon® server.

In many cases performance metrics for networking workloads are expressed in terms of packet processing in packet/sec [pps] or Gbits/sec [Gbps], or packet connections established per seconds, or some other packet-centric operations/sec. This leads to expressing the basic performance questions from the perspective of packet-centric operations, as follows:

1) **Packet processing operations** – *How many CPU core cycles are required to process a packet*?
2) **Memory bandwidth** – *How many memory-read and -write accesses are made per packet?*
3) **I/O bandwidth** – *How many bytes are transferred over PCIe link per packet?*
4) **Inter-socket transactions** – *How many bytes are accessed from the other socket or other core in the same socket per packet?*

The main goal for any performance optimization exercise is to get the best performance with the minimum CPU micro-architecture resources.

For network workloads, as outlined in *Section 3 NF Benchmarking Metrics*, the key indicator is the number of clocks required to process a packet. Recalling the CPP equation:

$$CPP = \frac{\#instructions}{packet} * \frac{\#cycles}{instruction}$$

Equation 6. NF computer performance equation with CPP.

The first metric, *#instructions/packet IPP*, depends on the program structure and logic, complier optimizations and several other optimization techniques that can bring down the IPP ratio for specific packet processing function. Code optimization examples include DPDK and VPP vectorization code employing CPU Vector instructions such as SSE2, AVX2 to process multiple packets with a single instruction.

The second metric, *#cycles/instruction*, or equivalent reciprocal metric *#instructions/cycle* (IPC), is one of the most important indicators of an execution efficiency on a specific CPU micro-architecture.

Software developers use several optimization techniques to achieve peak IPC. A good example is FD.io VPP, where vector packet processing employs adaptive packets batching and graph-of-nodes program structure to optimize use of CPU core cache hierarchy for both data and instructions, in turn reducing per packet memory access and clock cycles per packet.

In the tested generation of Intel® CPU micro-architecture (code-named Broadwell), the ALU execution unit can retire up to 4 instructions per each clock cycle. This simply means that theoretical IPC is 4.0. Extremely compute oriented workloads can have IPC of more than 3. However, IPC of 2.5 to 3 is still considered very efficient.

Following sections walk through each of these benchmarking dimensions, describing performance counters available in Intel® Xeon® processor E5 v4 family x86_64 micro-architecture, listing and explaining associated metrics and available measurement tools, as well as illustrating their applicability and use for sample NF applications.

4 NF Performance Tests and Results Analysis

4.1 Benchmarked NF Applications

A set of diverse NF applications has been used to illustrate the applicability of performance evaluation and analysis methodology described in this paper. They are listed in increasing level of packet processing complexity in *Table 1*.

Idx	Application Name	Application Type	Benchmarked Configuration
1	EEMBC CoreMark®[4]	Compute benchmark	Runs computations in L1 core cache.
2	DPDK Testpmd[5]	DPDK example	Baseline L2 packet looping, point-to-point.
3	DPDK L3Fwd	DPDK example	Baseline IPv4 forwarding, /8 entries.
4	FD.io VPP[6]	NF application	vSwitch with L2 port patch, point-to-point.
5	FD.io VPP	NF application	vSwitch MAC learning and switching.
6	OVS-DPDK[7]	NF application	vSwitch with L2 port cross-connect, point-to-point.
7	FD.io VPP	NF application	vSwitch with IPv4 routing, /32 entries.

Table 1. Example applications benchmarked in this paper.

The first benchmark is chosen to compare pure compute performance against rest of benchmarks having I/O as well.

The benchmarks 2. and 3. cover basic packet processing operations covering both I/O and compute aspects of the system. The packet processing functionalities increase with each benchmark in the order, and so does the compute requirements.

The last four benchmarks, listed as 4. to 7. cover the performance of the virtual switch, one of the most important ingredient in NF infrastructure. Virtual switch applications are tested in L2 switching and IPv4 routing configurations, covering both different implementations and various packet switching scenarios.

[4] EEMBC CoreMark - http://www.eembc.org/index.php.

[5] DPDK testpmd - http://dpdk.org/doc/guides/testpmd_app_ug/index.html.

[6] FDio VPP – Fast Data IO packet processing platform, docs: https://wiki.fd.io/view/VPP, code: https://git.fd.io/vpp/.

[7] OVS-DPDK - https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview.

4.2 Test Environment

4.2.1 Test Topology

All benchmarking tests described and referred to in this paper used a simple two-node topology with System Under Test server node and packet Traffic Generator node. Physical test topology is illustrated in *Figure 6*.

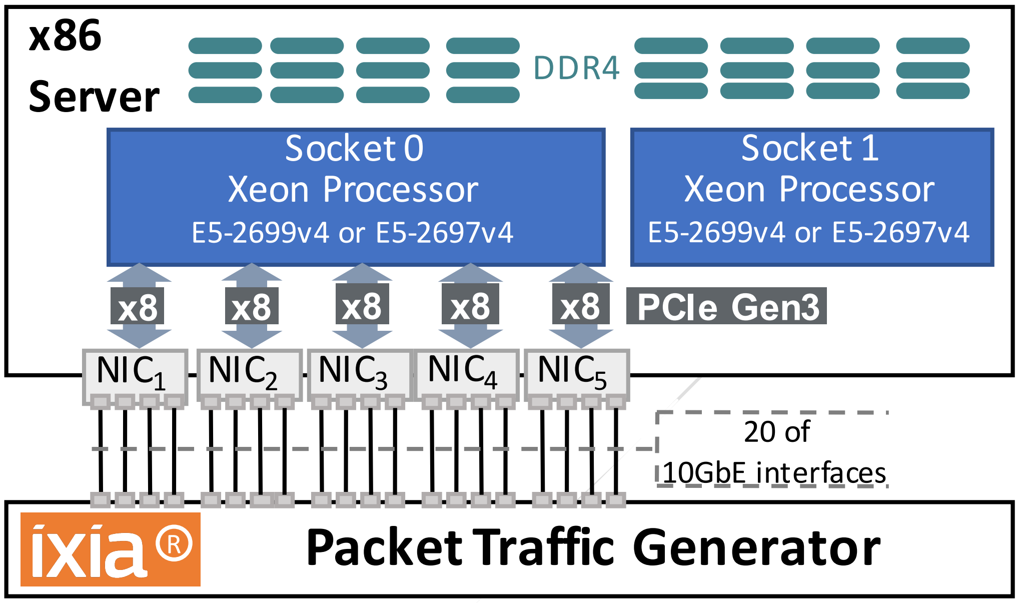


Figure 6. NF Applications Performance Test - Physical Topology.

4.2.2 Tested Configurations

For the sake of the focused evaluation, the paper limits the specifics of performance analysis methodologies to a single CPU micro-architecture, namely Intel® Xeon® E5 v4 Family (formerly known as Broadwell-EP). However, the methodology is equally applicable to other recent Intel x86_64 micro-architectures and other CPU types.

All applications run in user-mode on Linux. To evaluate dependencies on key CPU parameters e.g. core frequency, the benchmarks are executed on two Xeon® servers, each with two CPU sockets and different Intel® Xeon® E5 v4 family processors.

Idx	Core Frequency	Core Density	Intel® Xeon® Processor Model
Server1	2.20 GHz (Low)	22C (High)	E5-2699v4 55MB 145W
Server2	3.20 GHz (High)	8C (Low)	E5-2667v4 25MB 135W

Table 2. Benchmarked server processor specifications.

Except processors, the servers are otherwise identical from both Hardware and Software stack perspective. The exact compute server specifications in terms of used Hardware[8] and Software[9] operating stack have been provided in *Section 13. Appendix: Test Environment Specification*.

NF applications' data planes are benchmarked while running on a single physical CPU core, followed by multi-core tests to measure performance speed-up when adding CPU core resources. In order to stay within the known network I/O limits of the system, following multi-core and multi-10GbE port combinations have been chosen.

Number of 10 GbE ports used per test					
# of cores used / **Benchmarked Workload**	**1 core**	**2 core**	**3 core**	**4 core**	**8 core**
DPDK-Testpmd L2 Loop	8	16	20	20	20
DPDK-L3Fwd IPv4 Forwarding	4	8	12	16	16
VPP L2 Patch Cross-Connect	2	4	6	8	16
VPP L2 MAC Switching	2	4	6	8	16
OVS-DPDK L2 Cross-Connect	2	4	6	8	16
VPP IPv4 Routing	2	4	6	8	16

Table 3. Benchmark test variations for listed software applications.

Two main network I/O bottlenecks that drove above choices are: i) 14.88 Mpps 10GbE linerate for 64B Ethernet frames, and ii) 35.8 Mpps frame forwarding rate limit per used NIC cards (Intel® X710-DA4 4p10GbE). PCI Gen3 x8 slots' bandwidth has not been identified as a bottleneck in any of the benchmarks reported in this paper.

All tests are executed without and with hardware Symmetric Multi-Threading[10] using Intel® Hyper-Threading, with consistent mappings of threads to physical cores to 10GbE ports.

All tests are executed using CPU cores located on a single socket and using single NUMA node resources.

4.2.3 Compute Systems Under Test

All benchmarked Software applications were executed on Supermicro® servers, each fitted with two Intel® Xeon® E5 v4 Family processors.

Details of Systems Under Test and benchmarking environment have been provided in *Section 13. Appendix: Test Environment Specification*.

[8] Hardware – Xeon® server motherboards by Supermicro®, NICs by Intel® 4p10GE X710.

[9] Software stack – Operating System Linux 16.04 LTS.

[10] Symmetric Multi-Threading (SMT) – hardware-based parallel execution of independent threads to better utilize micro-architecture resources of CPU core.

4.2.4 Packet Traffic Generator and Offered I/O Load

Ixia®[11] packet traffic generator was used for all tests. Purpose developed automation test tools used Ixia Python API for controlling the traffic generator and to ensure consistent execution across multiple test iterations.

Configured network I/O packet load for the L2 tests involved 3,125 distinct (source, destination) MAC flows generated per interface, and highest scale of 50,000 flows for 16 of 10GbE interfaces. Each IPv4 test involved 62,500 distinct (source, destination) IPv4 flows per interface, and highest scale of 1,000,000 IPv4 flows. All flows were configured with 64Byte Ethernet L2 frame size.

Details of packet traffic generator configuration have been provided in *Section 13. Appendix: Test Environment Specification*.

4.3 Benchmark Results and Analysis

4.3.1 Measurements

Following tables show the test results for benchmarked NF applications including all identified high-level performance and efficiency metrics: i) Throughput *#packets/sec* [Mpps], ii) *#instructions/packet* (IPP), iii) *#instructions/cycle* (IPC) and iv) resulting *#cycles/packet* (CPP). EEMBC CoreMark® benchmark results are listed for comparison of CPU core usage metrics, more specifically *#instructions/cycle***.**

All benchmarked NF applications focus on packet header processing. Therefore, all benchmarks were conducted with smallest possible Ethernet frame size (64B), as it creates maximum stress scenario on processor cores, network devices, and interactions among them. In other words, the benchmarks are aimed at achieving maximum packets per second processing rate and throughput.

Results for the two tested processor types, Intel® Xeon® E5-2699v4 2.2 GHz and Intel® Xeon® E5-2667v4 3.2 GHz, are listed in *Table 4* and *Table 5* respectively.

Benchmarked Workload	**Throughput [Mpps]**		**#instructions /packet**		**#instructions /cycle**		**#cycles /packet**	
Dedicated 1 physical core with =>	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**
CoreMark [Relative to CMPS ref*]	1.00	1.23	n/a	n/a	2.4	3.1	n/a	n/a
DPDK-Testpmd L2 Loop	34.6	44.9	92	97	1.4	2.0	64	49
DPDK-L3Fwd IPv4 Forwarding	24.5	34.0	139	140	1.5	2.2	90	65
VPP L2 Patch Cross-Connect	15.7	19.1	272	273	1.9	2.4	140	115
VPP L2 MAC Switching	8.7	10.4	542	563	2.1	2.7	253	212
OVS-DPDK L2 Cross-connect	8.2	11.0	533	511	2.0	2.6	269	199
VPP IPv4 Routing	10.5	12.2	496	499	2.4	2.8	210	180
*CoreMarkPerSecond reference value	- score in the reference configuration: E5-2699v4, 1 Core noHT.							

Table 4. Performance and efficiency measured on Intel® Xeon® E5-2699v4 2.2 GHz.

[11] Other names and brands may be claimed as the property of others.

Benchmarked Workload	Throughput [Mpps]		#instructions /packet		#instructions /cycle		#cycles /packet	
Dedicated 1 physical core with =>	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**
CoreMark [Relative to CMPS ref*]	1.45	1.79	n/a	n/a	2.4	3.1	n/a	n/a
DPDK-Testpmd L2 Loop	47.0	63.8	92	96	1.4	1.9	68	50
DPDK-L3Fwd IPv4 Forwarding	34.9	48.0	139	139	1.5	2.1	92	67
VPP L2 Patch Cross-Connect	22.2	27.1	273	274	1.9	2.3	144	118
VPP L2 MAC Switching	12.3	14.7	542	563	2.1	2.6	259	218
OVS-DPDK L2 Cross-Connect	11.8	14.6	531	490	2.0	2.2	272	220
VPP IPv4 Routing	15.1	17.8	494	497	2.3	2.8	212	180
*CoreMarkPerSecond reference value	- score in the reference configuration: E5-2699v4, 1 Core noHT.							

Table 5. Performance and efficiency measured on Intel® Xeon® E5-2667v4 3.2 GHz.

Next sections provide analysis and interpretation of reported benchmarking results. Initial analysis of measured (and derived) baseline packet processing performance metrics is delivered first, followed by analysis of throughput speedup due to core frequency increase, use of Intel Hyper-Threading and use of multi-threading across multiple cores. Initial analysis is concluded with review of memory bandwidth consumption, PCIe I/O bandwidth consumption and inter-socket transactions measured during the benchmarking.

4.3.2 Initial Analysis

Here are the initial observations of measured baseline performance and efficiency metrics. Any references to measured values with Intel Hyper-Threading enabled (**HT** columns in tables) are quoted outside the round brackets, values with Intel Hyper-Threading disabled (**noHT** columns in tables) are quoted inside the round brackets. In cases when different values are measured on processors E5-2699v4 (2.2 GHz) and E5-2667v4 (3.2 GHz), they are referred to as a range of values N-M, respectively.

4.3.2.1 Instructions-per-Packet

Instructions-per-Packet metric (IPP, #instructions/packet) greatly depends on the number and type of packet processing operations required to realize a specific network function (or set of network functions), and how optimally they are programmed. One expects the simpler the function, the smaller number of instructions per packet, as illustrated in *Figure 7* for benchmarked NF applications.

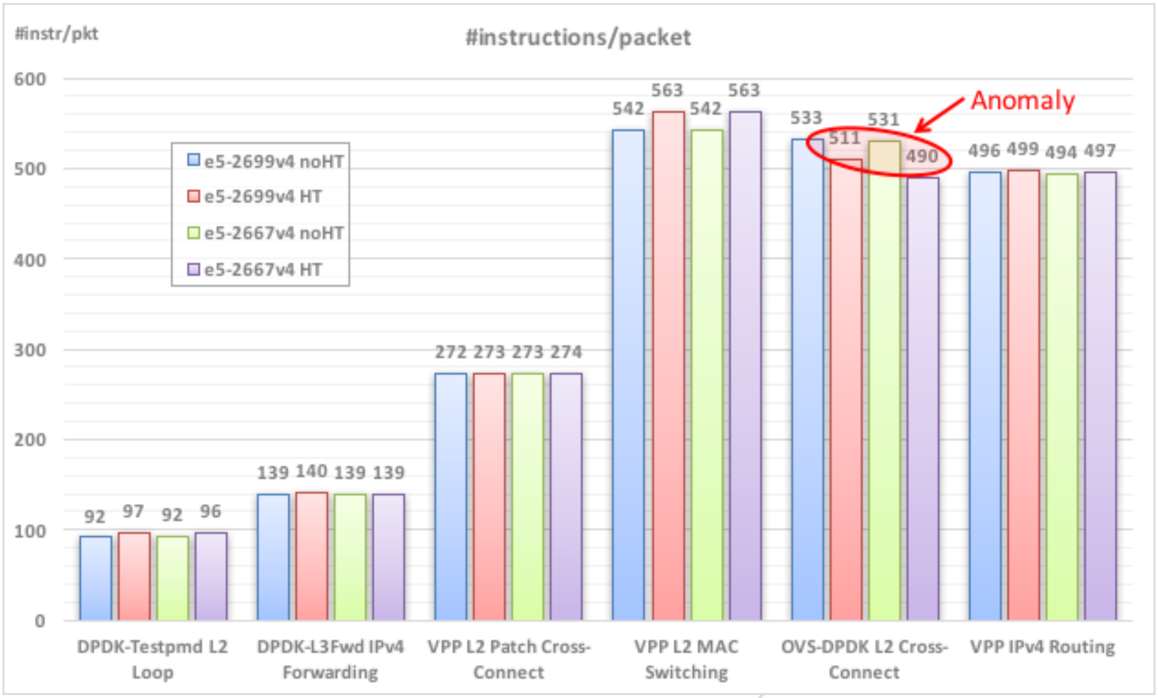


Figure 7. Number of instructions per packet for benchmarked applications.

And this is exactly what can be glanced from listed results when comparing **DPDK-Testpmd L2 packet looping** function yielding IPP of **96-97 (92)** with **VPP** and **OVS-DPDK L2 Cross-connect**, yielding **273-274 (273)** and **490-511 (531-533)** respectively. Significant IPP difference between VPP and OVS-DPDK indicates more optimally programmed operations on VPP for this relatively simple L2 Cross-connect network function. Notably **VPP L2 Switching** has a lower IPP of **563 (542)**, when compared to OVS-DPDK L2 Cross-connect.

Similar effect of substantial difference in offered network functionality is visible when comparing **DPDK-L3Fwd IPv4 forwarding** with **VPP IPv4 routing** functions, yielding IPP of **139 (139-140)** and **497-499 (494-496)** respectively. VPP implements a complete set of production-ready IPv4 routing functions that **DPDK-L3Fwd** lacks, including counters, error checks, complete header processing.

There is another aspect worth noting here. All of the NF applications tested do rely on the same DPDK NIC driver, and albeit they may differ in usage of the driver code, DPDK driver is the common program component.

With **DPDK-Testpmd** implementing the thinner network function from the tested lot (it is just a basic packet loop function between Rx and Tx), **DPDK-Testpmd** program spends most of its instructions on I/O interface operations between the CPU core and the NIC card(s), and as such provides a good estimate of the associated IPP cost of these operations for other NF applications tested.

There is one anomaly observed for OVS-DPDK L2 Cross-Connect, with ~5% difference in instruction/packet count between the two processors tested in HT mode (511 vs. 490), due to different #instructions/cycle measured for this case. Further explanation is given in *Section 4.3.2.3 Instructions-per-Cycle.*

Note that the reported instructions-per-packet are measured indirectly, calculated using the formula IPP = IPC * CPP. IPC is measured through the performance counters whereas CPP is derived from the packet throughput rate measured by the traffic generator and the core frequency (see *Equation 5*).

4.3.2.2 Instructions-per-Packet – I/O vs. Packet Processing Operations

Additional insight into the instructions-per-packet metric for tested NF applications is provided by using Intel Processor Trace (PT) tool on tested Intel® Xeon® E5 v4 Familyprocessors. PT is an Intel CPU feature that records branch retiring histories and stores them in highly compressed format in memory. Through post-processing PT data, users can reconstruct the exact runtime program execution flow and identify functions and number of instructions executed for different types of per packet operations.

For the purpose of this paper, PT data was captured using Linux perf-record tool and then translated to instruction logs with Linux perf-script tool. All of the benchmarked NF applications use loops to process packet in batches and the instruction logs represent a unroll view of these batch processing loops. With further post-processing, the instruction logs were divided into groups starting at a DPDK receive function and ending at the next DPDK transmit function. Each group of instruction logs signifies packet processing of single packet batch. Packet batch size information can be obtained either via understanding of specific NF application or by inspecting DPDK receive function instruction counts (since packets are received in a loop and each loop iteration will have fixed number of instructions). Finally, after dividing instructions-per-packet-batch by packet-batch-size, one can estimate the instructions-per-packet with additional inside on the type of functions and operations being executed per packet.

Figure 8 shows post-processed PT data generated for benchmarked NF applications, splitting per packet instructions into three categories: a) I/O operations, b) packet processing operations and c) application other operations.

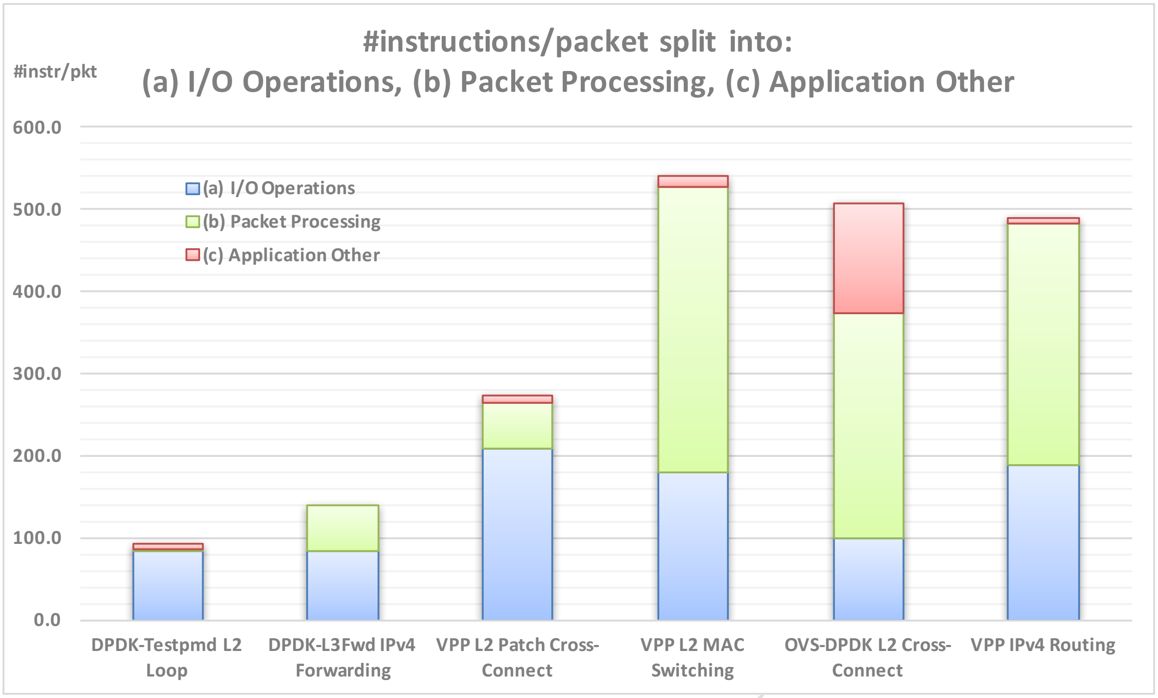


Figure 8. Instructions per packet split into I/O, packet processing, application other.

Presented Processor Trace data must be treated as indicative, due to restricted functionality of the first generation of PT supported on tested Intel® Xeon® E5 v4 Familyprocessors, with tracing consuming substantial resources. Next generation of Intel® Xeon® processors support enhanced PT functionality with greatly reduced resource footprint, enabling more granular and accurate analysis of run-time execution of functions and instructions, and allows for core clock cycle usage tracking.

Even from this indicative data, it is clear the DPDK-Testpmd IPP metric is dominated by I/O operations with almost no instructions spent on processing packets. For DPDK-L3Fwd and VPP L2 Patch Cross-Connect I/O operations still dominate the IPP budget, but packet processing instruction count becomes substantial, about 40% and 20% of the overall IPP value respectively. For remaining NF applications, IPP metric is dominated by packet processing operations, with I/O constituting 20 to 40% of the overall IPP value. Noticeably, for all tested scaled-up VPP configurations, both L2 MAC switching and IPv4 routing, I/O instructions still consume 40% of the overall instruction-per-packet metric, a substantial amount.

4.3.2.3 Instructions-per-Cycle

There is a number of underlying reasons behind the low (i.e. below 2) values of Instructions-per-Cycle metric (IPC, #instructions/cycle). The most common is CPU core waiting for the data from various levels of cache or system memory. This especially applies to memory and I/O intensive programs like NF data planes. On the other extreme, IPC can go artificially high if software program is polling for a variable to be updated by I/O or another core in a tight loop. In this case, little effective work is done, but IPC goes high due to execution of a small piece of code in the tight loop. In such case, per CPP *Equation 3*, this would also mean that IPP will show high number of instructions per packet even though part of the instructions are consumed while polling and not for actual packet processing. Nevertheless, IPC is usually the first attribute to be looked at for any program performance analysis.

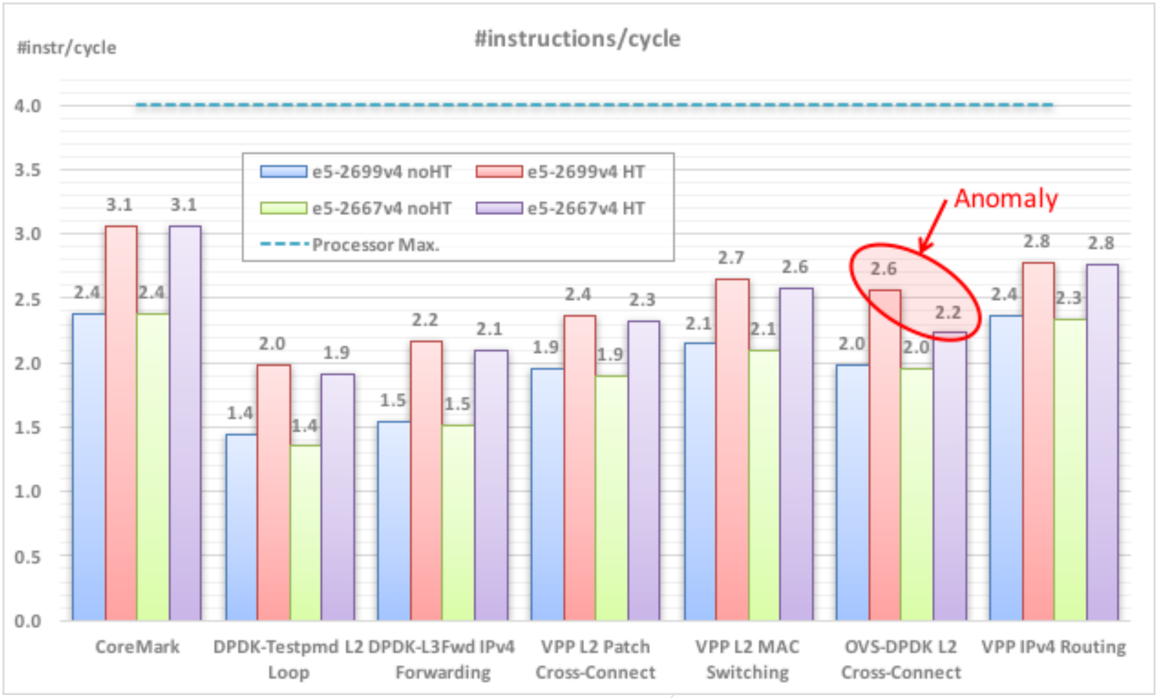


Figure 9. Number of instructions per core clock cycle for benchmarked applications.

From all benchmarked workloads, **CoreMark** scores the highest IPC of **3.1 (2.4)** with Hyper-Threading yielding a 29% increase due to more efficient use of out-of-order core execution engine that HT brings. CoreMark program fully executes in L1 cache, and clearly does not suffer from cache hierarchy or memory induced delays, what explains such a high score, out of theoretical maximum of 4.0 in tested CPUs. CoreMark IPC score is used as a reference to compare NF workloads against.

The closest to CoreMark is **VPP IPv4 Routing** with IPC scores of **2.8** (**2.3-2.4)** yielding 17% increase with HT. Here VPP is as efficient as CoreMark w/o HT, and only 10% less efficient w/ HT. Knowing the levels of I/O load and cache/memory load involved, this indicates extremely optimized code in VPP for IPv4 routing path.

Next are **VPP L2 Switching** and **OVS-DPDK L2 Cross-connect** scoring **2.6-2.7 (2.1)** and **2.7 (2.2)** respectively. These are still good IPC scores, especially w/HT, but clearly in both cases L2 packet paths are less performance optimized compared to VPP for IPv4 routing path.

VPP L2 Cross-connect follows with IPC of **2.3-2.4** (**2.2)** and scores lower than VPP L2 switching path, an interesting phenomenon. Most likely it is down to I/O packet move operations dominating L2 Cross-connect packet processing path, with associated intense interactions with cache hierarchy, memory and I/O sub-systems. Further runtime measurements are required to fully determine the reason here (e.g. by using recently available Intel® Processor Trace tooling), and are subject to further study.

Trailing the pack from IPC score perspective are **DPDK-Testpmd** and **DPDK-L3Fwd** with IPC scores of **1.9-2.0** (**1.4)** and **2.1-2.2** (**1.5)**. Albeit still scoring IPC above 2 w/ HT, both DPDK sample applications have substantially lower scores compared to all other applications tested. Similarly, to VPP L2 Cross-connect case, it is mostly due to their operations being dominated by I/O vs. packet processing, as explained the lower IPC score in the opening paragraph to this section. Further study and runtime measurements are required to fully determine the reason and difference against other workloads tested.

Repeated measurements registered one anomaly for **OVS-DPDK L2 Cross-Connect**, where IPC gets reduced by 15% (from 2.6 to 2.3) when scaling the frequency in HT mode. This indicates the workload is less efficient in hiding the latency of Last Level Cache (LLC), memory and PCIe I/O access when two parallel threads are run at higher core frequency. This is due to the fact that LLC, memory, and PCIe complex have their own frequency domains independent from the core, and IPC may not necessarily remain the same as the load increases and a core's access patterns to these domains change.

IPC is measured using Intel® on-chip Performance Monitoring Units (PMUs) hardware counters embedded in tested CPUs. *Section 5.1 Telemetry Points in Intel® Xeon® E5 Processor Architecture* describes the PMU architecture of Intel® Xeon® E5 v4 Family processors in more detail.

Note of caution: One should not overstress the importance of IPC metric as a standalone program execution efficiency measure. It is IPC in combination with IPP that more accurately represent the actual network function implementation efficiency. And this brings us to the CPP metric.

4.3.2.4 Cycles-per-Packet

Cycles-per-Packet metric (CPP, #cycles/packet) is the direct measure of time spent by compute machine in processing a packet. Serious software optimization techniques analyze cycles consumed by different packet processing functions and try to save every cycle possible. Clearly this technique has been applied to all NF applications tested, as all of them measure good CPP values.

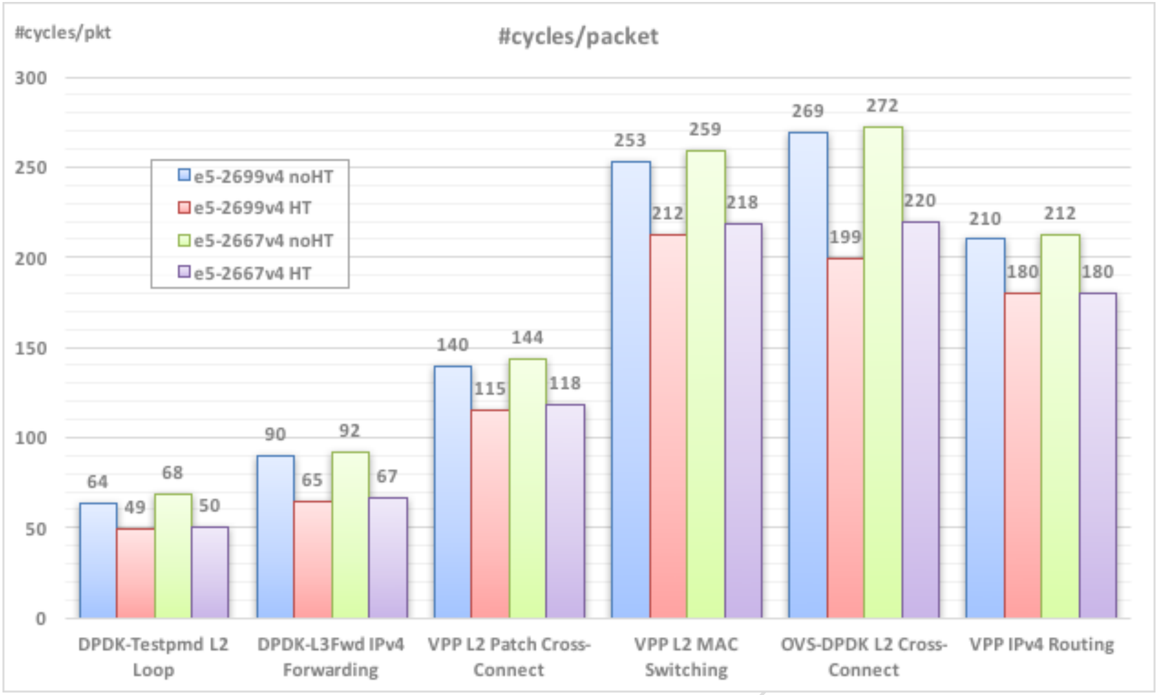


Figure 10. Number of core clock cycles per packet for benchmarked applications.

Still there are some interesting differences.

DPDK-Testpmd and **DPDK-L3Fwd** lead the pack with lowest CPP values of **49-50 (64-68)** and **65-67 (90-92)** respectively. These low CPP values reflect the fact that both applications are dominated by DPDK I/O operations, with minimal additional packet processing. All other benchmarked NF applications and packet paths use the same DPDK I/O operations, but then they implement complete network functions that call more packet processing operations. And it all adds up.

What is interesting are the lower CPP values for VPP compared to OVS-DPDK. They do result from VPP leading with both IPP and IPC metrics across all packet paths as noted earlier. Among different VPP packet paths, **L2 Cross-connect** comes as the lowest-cost cycle-wise with **115-118 (140-144)**, what is not surprising as it is the simplest packet path. But then surprisingly it is followed by **VPP IPv4 Routing** packet path with impressive CPP of **180 (210-212)**, ahead of **VPP L2 Switching** with CPP of **212-218 (253-259**). This is expected due to L2 switching path having to deal with both source and destination address lookup, as seen in higher #instr/packet measurements reported in the earlier section. All VPP packet paths compare favorably with **OVS-DPDK L2 Cross-connect** with CPP of **199-220 (269-272)**.

All tests show lower CPP value (an improvement, as lower is better) for tests w/ HT compared to w/o HT, confirming expectation that Hyper-Threading improves physical core utilization efficiency. However, the relative change differs across the NF applications. DPDK-Testpmd shows 23% to 26% decrease of CPP value. Similarly, DPDK-L3Fwd shows 27% to 28% decrease. VPP on the other hand shows a lower decrease of CPP between 14% and 18% depending on the packet path tested. OVS-DPDK measured CPP value decrease of 20% to 30%.

Also note that when CPP remains almost the same for both high and low frequency CPUs, it directly implies that the performance scales linearly with cpu frequency.

Reported average #cycles-per-packet are measured indirectly, calculated using the formula CPP = Core_Frequency / Throughput [pps].

4.3.2.5 Packets-per-Second Throughput

Measured packet throughput [Mpps] values are inversely proportional to reported CPP values, therefore the same observations noted for CPP equally apply here. The Mpps per core metric is very commonly used as a basic performance sizing metric for NF data plane capacity planning. Especially that it also used as the main reference value for analyzing multi-threading performance speedup.

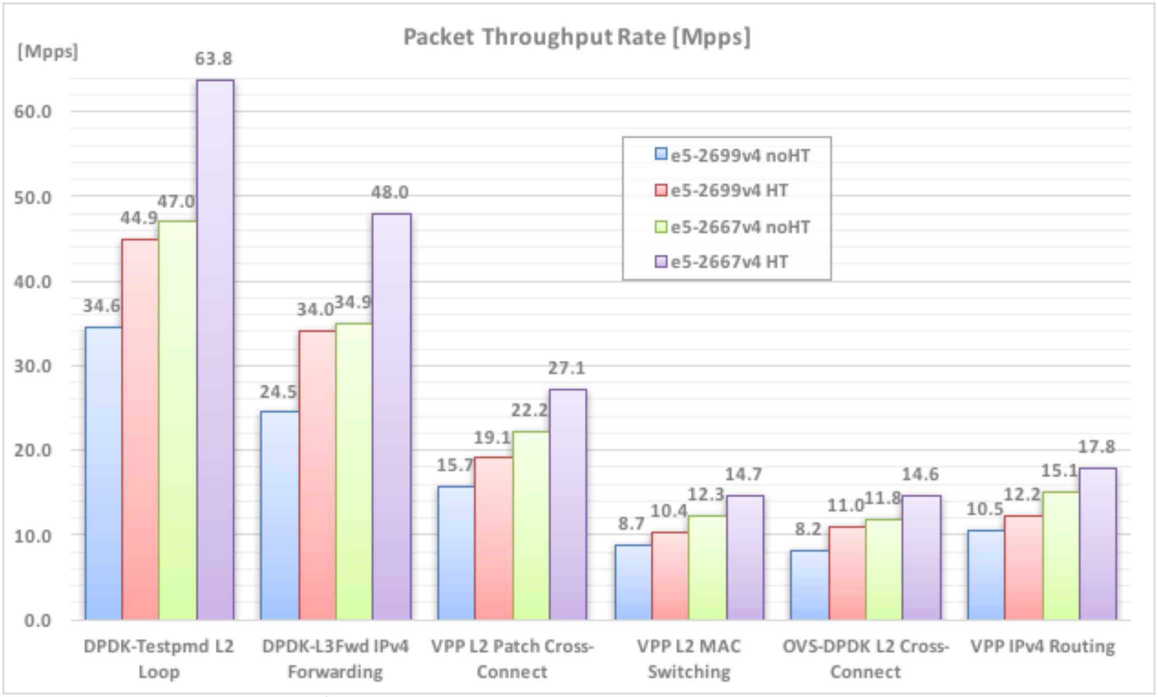


Figure 11. Packet Throughput Rate for benchmarked applications with a single core.

Reported packet throughput [Mpps] values are measured directly using Ixia® traffic generator.

4.3.2.6 Initial Conclusions

From reported performance data and the initial observations, it is clear that all tested NF applications have been quite well optimized for performance on Intel® Xeon® E5 v4 Family processors. Noted efficiency differences between the DPDK Testpmd and L3Fwd and all other NF workloads result from DPDK applications focusing mainly on I/O operations with minimal packet processing. Furthermore, looking specifically at cycles/packet (CPP) metrics measured for VPP and OVS-DPDK, VPP clearly leads with lowest CPP values and highest packet throughput rates for all tested packet paths and configurations. This is a good indicator of levels of optimizations present in VPP data plane and its leadership in software data plane space.

It is also clear that CPP is indeed the base efficiency metric that allows for direct comparison of NF data plane implementations. It applies when comparing the same network function implemented in different NF Apps running on the same HW, and equally when running the same NF Apps on different HW or in different configurations. CPP is a metric that binds other efficiency metrics (see *Equation 3*) and the metric that directly translates into the main packet forwarding performance metric, the packet throughput rate (see *Equation 5*).

4.3.3 Throughput Speedup Analysis

4.3.3.1 Processor Core Frequency

One expects performance to proportionally scale with processor core frequency, in perfect case. For pure compute workloads, the faster the clock frequency, proportionally more work is executed. For networking loads, perfect scaling with frequency means constant CPP, and proportionally more packets being processed per second. And this indeed applies to the benchmarked NF applications.

Figure 12 shows relative packet throughput increase between E5-2699v4 processor clocked at 2.2 GHz and E5-2667v4 processor clocked at 3.2 GHz.

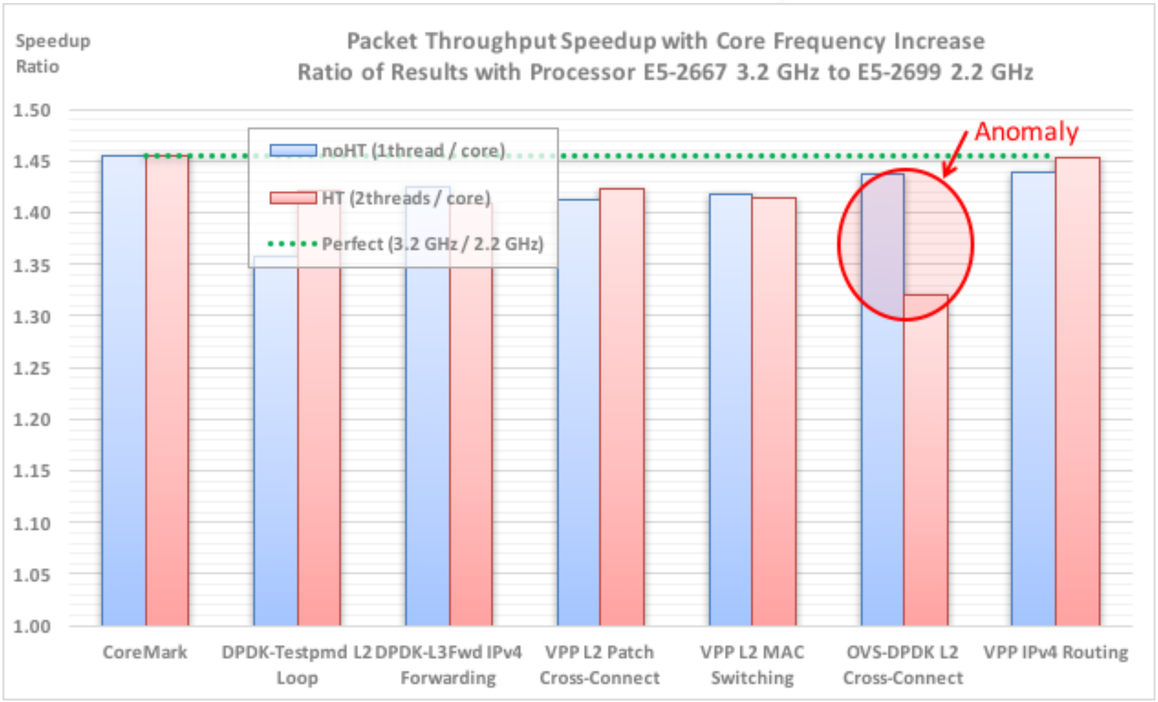


Figure 12. Packet throughput speedup with core frequency increase.

Although benchmarked NF applications show reasonably linear scaling of performance with frequency, such close to perfect scaling may not be always achievable for a number of reasons. Here some common examples:

- By increasing core frequency, performance could hit to Gigabit Ethernet link line-rate, NIC packet throughput limit or PCIe slot I/O bandwidth limit.

- Workload cannot hide latency of Last Level Cache, memory, or I/O access. LLC, memory, and PCIe have their own independent frequency domains different than the core, and if they become an impacting factor the performance would not linearly scale with the core frequency. This is the most likely reason behind the anomaly observed for OVS-DPDK L2 Cross-Connect, with lower than expected throughput speedup.
- Cores are contending for the same cache-line frequently (e.g. spin locks), wasting uncertain number of cycles at different core frequencies.

4.3.3.2 Intel Hyper-Threading with Multi-Threading

Intel® Hyper-Threading (HT), an implementation of Simultaneous Multi-Threading (SMT), is a technique for improving the overall instruction execution efficiency of superscalar processor CPUs by using hardware multi-threading. In general, SMT is expected to permit multiple independent program threads of execution to better utilize the resources provided by any modern processor architecture.

Intel HT enables single physical processor core to appear and behave as two logical processors to the operating system. Each logical processor has its own architecture state and has its own full set of data registers, segment registers, control registers, debug registers, and most of the Model Specific Registers (MSR) used to control x86 cores. Each hyper-thread has also its own advanced programmable interrupt controller (APIC). The logical cores share the Frontend and Backend resources in the physical cores including L1, L2 caches, execution engines, instruction decoder, schedulers, buffers, uncore interface logic.

The core achieves hyper-thread level parallelism by out-of-order scheduling to maximize the use of execution units during each cycle. In essence, a hyper-thread would try to use unused execution slots when its pairing hyper-thread is either idle or cannot make forward progress due to execution stalls. If both hyper-threads are competing, they would share resources in a fair manner. As a result, performance metrics per physical core could change when using hyper-threads compared to the non-hyper-thread situation:

i) Overall #instructions/packet metric could improve due to the hyper-thread level parallelism;

ii) Percentage of retiring instructions could increase due to the increased utilization of the core execution resources;

iii) Average percentage of the core backend bound penalty could decrease;

iv) Average percentage of the core frontend bound penalty could increase, as more instructions are demanded and executed by the core backend.

The performance change between Hyper-Thread and non-Hyper-Thread setups highly depends on the characteristics of the programs running on each thread.

For more detailed description of Intel Hyper-Threading technology please refer to available online Intel documentation[12].

---

[12] Intel Hyper-Threading, https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

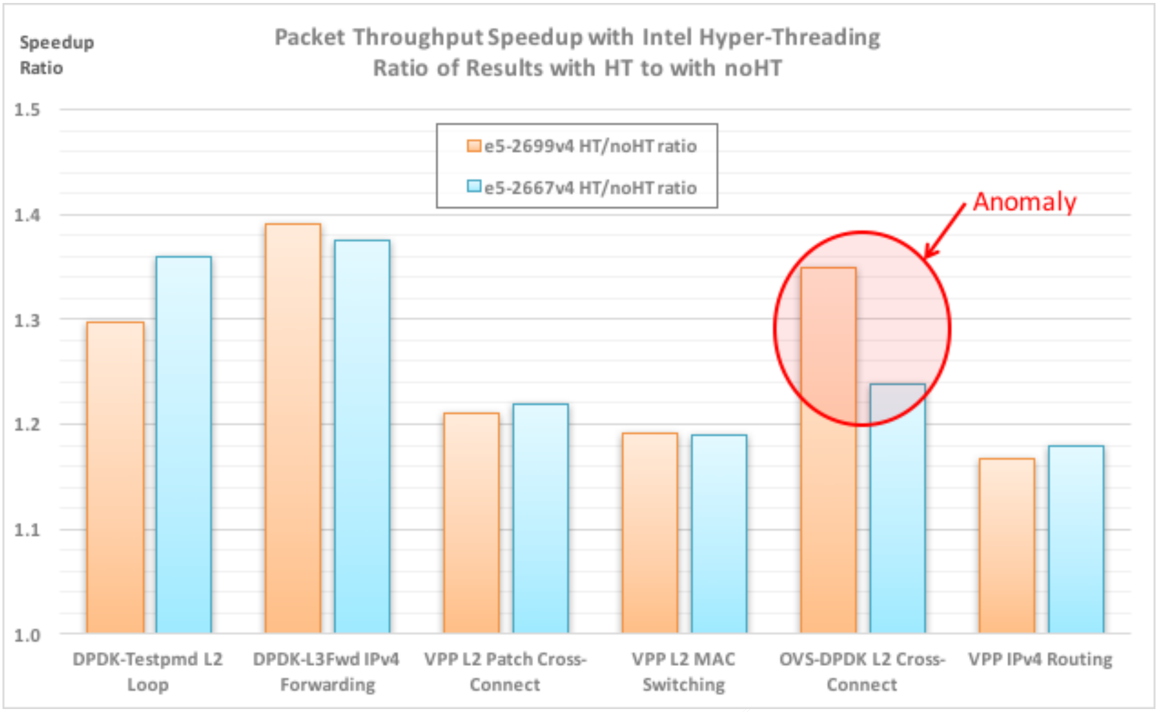


Figure 13. Packet throughput speedup with Intel Hyper-Threading.

All benchmarked NF applications demonstrate a packet throughput increase when paired threads are running on Hyper-Thread enabled processors. DPDK-Testpmd and DPDK-L3Fwd exhibit greatest performance speedup of ~1.3 to ~1.4 (~30 to ~40% increase), with the rest of the NF applications' speedup due to use of HT in the range of ~1.13 to ~1.25 (~13 to ~25%). The main difference between these two groups of NF applications is the former ones executing a limited number of packet processing operations, with their #instructions/packet budget dominated by I/O operations, as pointed out in *Section 4.3.2.1 Instructions-per-Packet*. Further analysis of this phenomena is provided in *Section 6 Compute Performance Analysis using Intel TMAM*. that describes analysis and interpretation of processor performance counters data using Intel TMAM methodology.

4.3.3.3 Multi-Core with Multithreading

Most of NF applications are nowadays capable of running multiple SW threads dealing with data plane packet processing. Subject to the actual SW implementation, the gain of running multithreaded and multi-core greatly depends on contention for shared HW resources (cache, memory, I/O) and synchronization (locks, synchronization, load imbalance). Only in ideal situation the speedup resulting of using multiple cores is actually linear.

Figure 14 graphs show measured packet throughput performance for all benchmarked NF applications, running in multi-threaded configurations ranging from 1 to 8 cores, without Hyper-Threading (1 thread per each core) and with Hyper-Threading (2 threads per each core). For comparison, perfect linear multi-core speedup is plotted based on single core performance. All tests are done with 64Byte Ethernet frames on the compute machine with Intel® Xeon® 2699v4 2.2 GHz processors. Very similar speedup behavior has been observed on the compute machine with Intel® Xeon® 2667v4 3.2 GHz CPUs (results not plotted). To ensure equal load distribution per core, all L2 tests were done with 3,125 MAC flows per 10GbE port, and all IPv4 tests were done with 62,500 IPv4 flows per 10 GbE port. Resulting highest scale tested with 16 10GbE ports included 50,000 MAC flows and 1,000,000 IPv4 flows respectively.

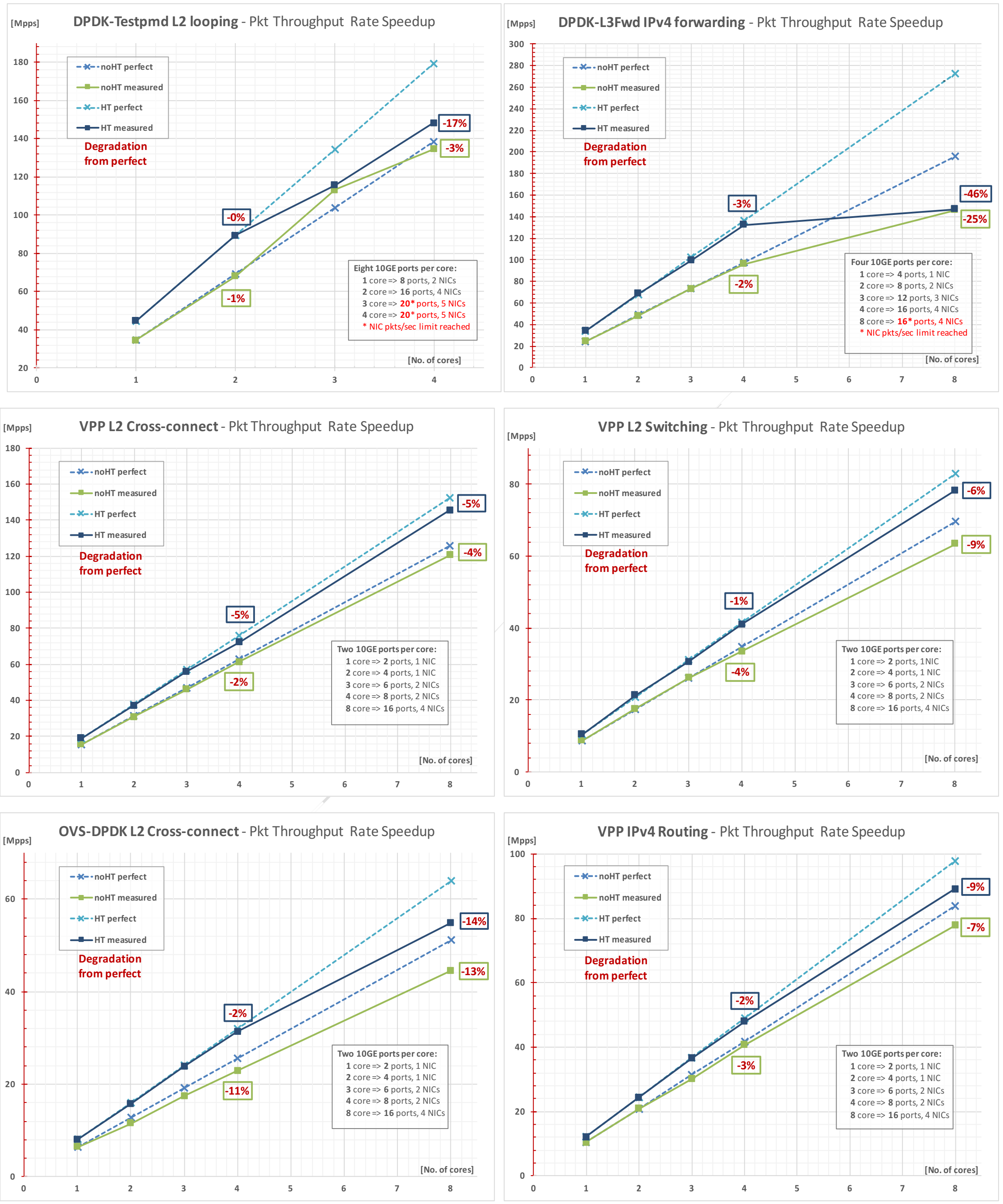


Figure 14. Packet throughput speedup with Multithreading and Multi-core.

All benchmarked NF applications demonstrate close to perfect linear multi-core scaling. Significant degradation is only observed for **DPDK-Testpmd** and **DPDK-L3Fwd**, and this is due to reaching the hardware limit of packet throughput per NIC (35.8 Mpps).

When operating within the packet throughput limits per NIC (35.8 Mpps) and per 10GbE port (14.44 Mpps), the performance degradation vs. perfect in tests up to 8 cores, stays well **within the -10%,** with the exception of **OVS-DPDK** where it goes up to **-14%**.

4.3.4 Further Analysis

Further analysis of performance test results and associated collected hardware performance counters data require deeper understanding of modern processor CPU micro-architecture and a well-defined interpretation approach for analyzing underlying compute resource utilization and hotspots limiting program execution performance.

Intel® *Top-down Microarchitecture Analysis Method* (TMAM) is well suited to address this for modern Intel processors. In short, "TMAM is a practical method to quickly identify true bottlenecks in out-of-order processors including the Intel® Xeon®. The developed method uses designated performance counters in a structured hierarchical approach to quickly and correctly identify dominant performance bottlenecks." TMAM analysis has been successfully used for CPU performance analysis of various types of workloads and equally applies for use with NF applications. The methodology is extensively documented in Intel® Optimization Manual[13] and a number of published papers[14]. The method is adopted by multiple tools including Intel® VTune and Linux open-source PMU-tools[15], helping developers adopt this approach without going through intricacies of the performance monitoring architecture and performance events.

Section 6 Compute Performance Analysis using Intel TMAM contains a primer on TMAM and how it applies to NF applications, followed by sample PMU-tools based measurements for tested NF applications and data analysis.

4.4 Memory Bandwidth Consumption

Runtime usage of system memory bandwidth is another important high-level efficiency metric that need to be analyzed in the context of implemented packet operations. The memory bandwidth utilization does not only indicate the headroom left on the memory channels, but also provides an important metric of per packet memory accesses.

Table 6 shows Memory consumed in high network bandwidth scenario i.e. workloads running with 4Cores/8Threads.

[13] "Intel Optimization Manual" – Intel® 64 and IA-32 architectures optimization reference manual.

[14] Technion 2015 presentation on TMAM , Software Optimizations Become Simple with Top-Down Analysis Methodology (TMAM) on Intel® Microarchitecture Code Name Skylake, Ahmad Yasin. Intel Developer Forum, IDF 2015. [Recording].

[15] Linux PMU-tools, https://github.com/andikleen/pmu-tools.

Benchmarked Workload	Throughput [Mpps]	Memory Bandwdith [MB/s]
Dedicated 4 physical cores with HyperThreading enabled, 2 threads per physical core, 8 threads in total => 4c8t		
DPDK-Testpmd L2 looping	148.3	18
DPDK-L3Fwd IPv4 forwarding	132.2	44
VPP L2 Cross-connect	72.3	23
VPP L2 Switching	41.0	23
OVS-DPDK L2 Cross-connect	31.2	40
VPP IPv4 Routing	48.0	1484

Table 6. Memory bandwidth consumption for tested NF applications.

In most cases Memory bandwidth is close to 0 MB/s due to effective use of Intel® DDIO (Direct Data IO) feature. More detailed description of DDIO technology and its applicability to NF applications is provided in *Section 8.3 Intel® Direct Data IO Technology (DDIO)*.

Furthermore, *Section 7 Memory Performance Analysis* is dedicated to memory performance, including NF test measurements with Intel PCM *pcm-memory.x* tool and results analysis.

4.5 I/O Bandwidth Consumption

Input/Output plays a significant role in NF applications' performance. Understanding and monitoring I/O behavior in the architecture is thus extremely important. *Table 7* shows the average PCIe I/O bandwidth consumption for benchmarked NF applications.

Benchmarked Workload	Throughput [Mpps]	PCIe Write Bandwidth [MB/s]	PCIe Read Bandwidth [MB/s]	PCIe Write #Transactions/ Packet	PCIe Read #Transactions/ Packet
Dedicated 4 physical cores with HyperThreading enabled, 2 threads per physical core, 8 threads in total => 4c8t					
DPDK-Testpmd L2 looping	148.3	13,397	14,592	1.41	1.54
DPDK-L3Fwd IPv4 forwarding	132.2	11,844	12,798	1.40	1.51
VPP L2 Cross-connect	72.3	6,674	6,971	1.44	1.51
VPP L2 Switching	41.0	4,329	3,952	1.65	1.51
OVS-DPDK L2 Cross-connect	31.2	3,651	3,559	1.83	1.78
VPP IPv4 Routing	48.0	4,805	4,619	1.56	1.50

Table 7. PCIe I/O bandwidth consumption for tested NF applications with 64B Ethernet frames.

For the test scenario with a fixed 64B Ethernet frame size, PCIe bandwidth is proportional to the packet rate.

Number of PCIe read transactions per packet varies depending on the descriptor size the software chooses for the Ethernet NIC cards. For all cases except OVS-DPDK, the packet descriptor size is 16B, resulting in 32B of total overhead per 64B packet (16B descriptor for Ethernet Transmit, and 16B for Ethernet Receive). This results into 1.5x of PCIe read transactions per packet. OVS-DPDK software uses extended size descriptors (32B) for Ethernet Receive, resulting into higher PCIe read/packet rate ratio.

PCIe write transactions/packet ratio depends both on the descriptor size as well as the descriptor write back policy chosen by the software. In some of the tested cases, Ethernet cards are programmed to write back only 1 Transmit descriptor per every 16th packet, with the Receive descriptors being the same as for PCIe read transactions. This results in lower PCIe writes/packet ratio of [1B(rx_descriptor) +16B (tx_descriptor) +64B (packet)]/64B(packet_size) =~1.265. However, due to non-coalesced descriptor write, the counters round up the descriptor size to 64B, resulting in over reporting PCIe write bandwidth, compared to reality.

Section 8 PCIe Performance Analysis is dedicated to PCIe I/O performance, including usage of Intel PCM *pcm-pcie*.x monitoring tool, NF test measurements and data analysis.

4.6 Inter-Socket Transactions

The last part of compute system architecture to look at is the CPU core to core communication involving inter-socket transactions for Intel® Xeon® two- and four-socket server configurations. Intel Architecture is CPU cache coherent architecture, meaning that when CPU or I/O accesses a cache-line, the architecture ensures that it gets the most recent version of the cache-line. It also ensures that only one modified data copy exists in the system, whether it is in one or more level of core caches of either socket or system memory. Such operation involves snoop transactions on the Intel® QuickPath Interconnect (QPI), an interconnect link between the two CPU sockets. Understanding these transactions is important for achieving the peak performance out of a multi-socket system.

All benchmarking of NF applications reported in this paper has been performed with NF application threads pinned to processor socket0, excluding any inter-socket transactions and QPI involvement in program execution. Hence no measurement data is provided in this version of the paper.

Focused analysis of inter-socket transactions onto the NF application performance is subject to further study.

5 Intel x86_64 – Performance Telemetry and Tools

Systematic, focused and repeatable performance analysis of any application requires availability of suitable and reliable performance telemetry points (counters) with associated measurements and monitoring tools. There are several commercial tools, e.g. Intel *VTune*^{TM}, and open source tools e.g. Linux *perf*, Linux *pmu-tools*, Intel *PCM*.

This section provides overview of the open source tools. First though is a brief description of the underlying hardware based performance telemetry architecture used by those tools, to aid understanding.

5.1 Telemetry Points in Intel® Xeon® E5 Processor Architecture

For any Software applications to take advantage of CPU microarchitectures, one needs to know how the application is utilizing available hardware resources. One way to obtain this knowledge is by using the on-chip Performance Monitoring Units (PMUs). PMUs are dedicated pieces of logic within a CPU core that count specific Hardware events as they occur in the system. Examples of these events include Cache Misses and Branch Mispredictions. These events can be observed, counted and combined to create useful high-level metrics such as cycles-per-instruction (CPI) or its reciprocal equivalent instructions-per-cycle (IPC).

Figure 15 below shows the performance counters available within the Intel Xeon E5-2600 v4 series processor micro-architecture.

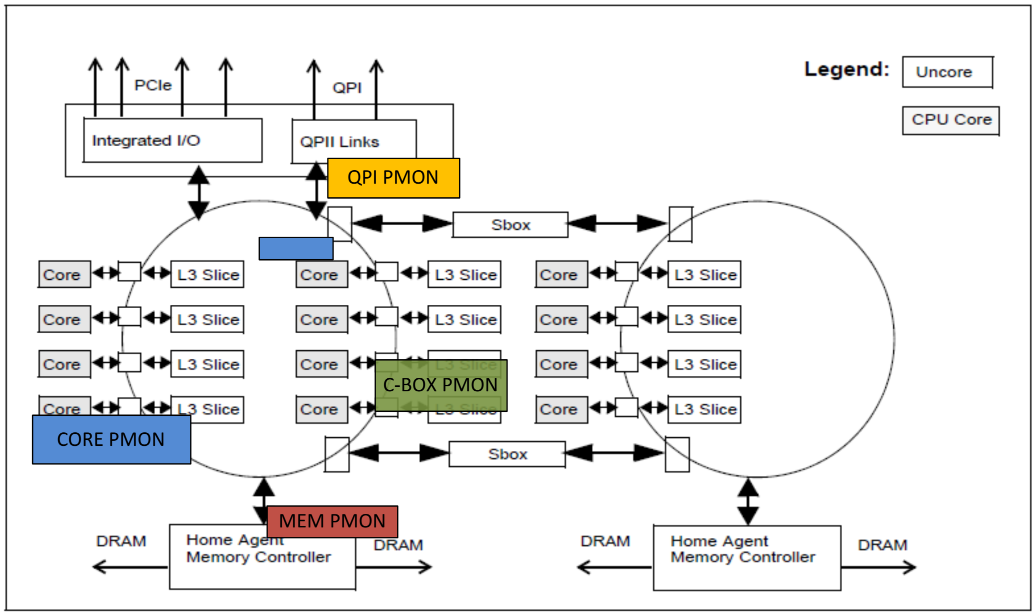


Figure 15. High Level view of Intel® Xen® E5 v4 Family processor Architecture.

In traditional applications, most of the attention goes towards performance counters in CPU cores. However, for system level performance analysis of I/O centric applications like NFV, in addition to CPU cores equally important are also uncore, I/O, and Processor interconnect counters. Cumulatively, there are more than one thousand performance monitoring events that can help understand microarchitecture activities while running an application. They are grouped into the following categories:

1. CORE PMON - Core specific events monitored through Core performance counters. Examples include number of instructions executed, TLB misses, cache misses at various cache levels, branch predictions.
2. C-BOX PMON - C-Box performance monitoring events used to track LLC access rates, LLC hit/miss rates, LLC eviction and fill rates, and to detect evidence of back pressure on the LLC pipelines. In addition, the C-Box has performance monitoring events for tracking MESI state transitions that occur as a result of data sharing across sockets in a multi-socket system.
3. MEM PMON – Memory controller monitoring counters. There are four counters in the E5-2600 E5 v4 Family processor architecture which monitor activities at memory controller.
4. QPI PMON - QPI counters monitor activities at QPI links, the interconnect link between the processors.

Some of the Core counters are enabled only when Hyper-Threading (HT) is turned off. It is recommended to turn HT off in the BIOS setting when one wants to do deep dive analysis with all the performance monitoring infrastructure, as presented in this paper.

5.2 Performance Monitoring Tools

Number of tools and code libraries enables monitoring available performance counters and enabling their analysis. Some utilities like Intel *VTune*^{TM} and Linux *perf top* can even point to hot spots in the source code that cause high count of selected event.

Here is a brief description of applicability of three open-source tools that have been used in this paper for performance measurements and analysis:

- **Linux Perf** – a generic Linux kernel tool. It supports multiple generations of x86 and many other CPU architectures. Perf incorporates basic performance events for each architecture. Perf tool can also be used to conduct performance hot-spot analysis at source code level.
- **PMU-Tools** – a set of open-source utilities built upon Linux Perf, providing rich analysis tools to a user. PMU_Tools supports download of an enhanced set of performance monitoring events for a particular architecture using (event_download.py) on top of what is available with Linux Perf. Additionally, PMU-Tools includes powerful performance analysis tool based on the Top-down Micro-Architecture Method (TMAM), as described in this paper.
- **PCM tool chain** – PCM is another open-source tool for monitoring various performance counters. PCM consists of a set of utilities, each one focusing on specific parts of architecture including Core, Cache hierarchy, Memory, PCIe, NUMA. It is easy to use and great for showing high-level statistics. One of the main advantages of PCM tools is that one can observe variety of CPU core, CPU uncore,

and memory events in real time. For example, one observe in real time how IPC, LLC miss, memory and PCIe consumption vary with change in network loads.

Table 8 below shows mapping of open source tools to specific performance analysis area.

Analysis Area	Performance Tools
CPU Cores	pmu-tools top-down analysis.
CPU cache Hierarchy	pmu-tools, pcm.x.
Memory bandwidth	pcm-memory.x, pmu-tools.
PCIe bandwidth	pcm-pcie.x
Inter-Socket transactions	pcm-numa.x, pmu-tools.

Table 8. Mapping of performance tools to CPU architecture analysis area.

6 Compute Performance Analysis using Intel TMAM

6.1 TMAM Overview

Analyzing and optimizing applications' performance has become increasingly hard due to continuously increasing processor microarchitecture complexities, Software applications diversity and huge volume of measurement data produced by performance tools.

Intel Top-down Microarchitecture Analysis Methodology[16] (TMAM) has been developed and successfully applied to address this problem. TMAM provides fast and accurate performance analysis of variety of workloads by employing a structured drill-down approach to investigate bottlenecks in out-of-order processors, while running steady workloads. The hierarchical top-down approach saves time, helps users to quickly identify bottlenecks and to focus on the most important areas for performance optimization. PMU-tools[17] have been developed in open-source to help conduct performance analysis using TMAM approach.

Before diving into TMAM performance analysis, it is worthwhile to get familiar with further details of Intel® Xeon® E5 v4 Family processor micro-architecture. *Section 3.4 Compute Resources Usage* has already described the entire compute system and its block diagram in *Figure 5* featuring processor cores, uncore, I/O, and interconnect system blocks in Intel Xeon processor architecture. Zooming into the processor core itself, *Figure 16* below illustrates the functional units present within a Core, with the CPU core pipeline divided conceptually into two halves, the *Frontend* and the *Backend*. The *Frontend* part implements an in-order code execution and is responsible for fetching the program code represented in architectural instructions and decoding them into one or more low-level Hardware operations referred to as micro-operations (µOps). The µOps

[16] A Top-Down Method for Performance Analysis and Counters Architecture, Ahmad Yasin. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, https://sites.google.com/site/analysismethods/yasin-pubs.

[17] PMU tools: https://github.com/andikleen/pmu-tools.

are then fed to the *Backend* in a process called allocation. Once allocated, the *Backend* (implementing an out-of-order execution) is responsible for monitoring when µOp's data operands are available and executing the µOps in available execution unit. The completion of a µOp's execution is called *Retirement*, and happens when results of the µOp's execution are committed to the architectural state of the processor, written to CPU registers or written back to memory. Usually most µOps pass completely through the pipeline and retire, but sometimes speculatively fetched µOps may get cancelled before retirement – like in the case of *Bad Speculation* with mispredicted branches.

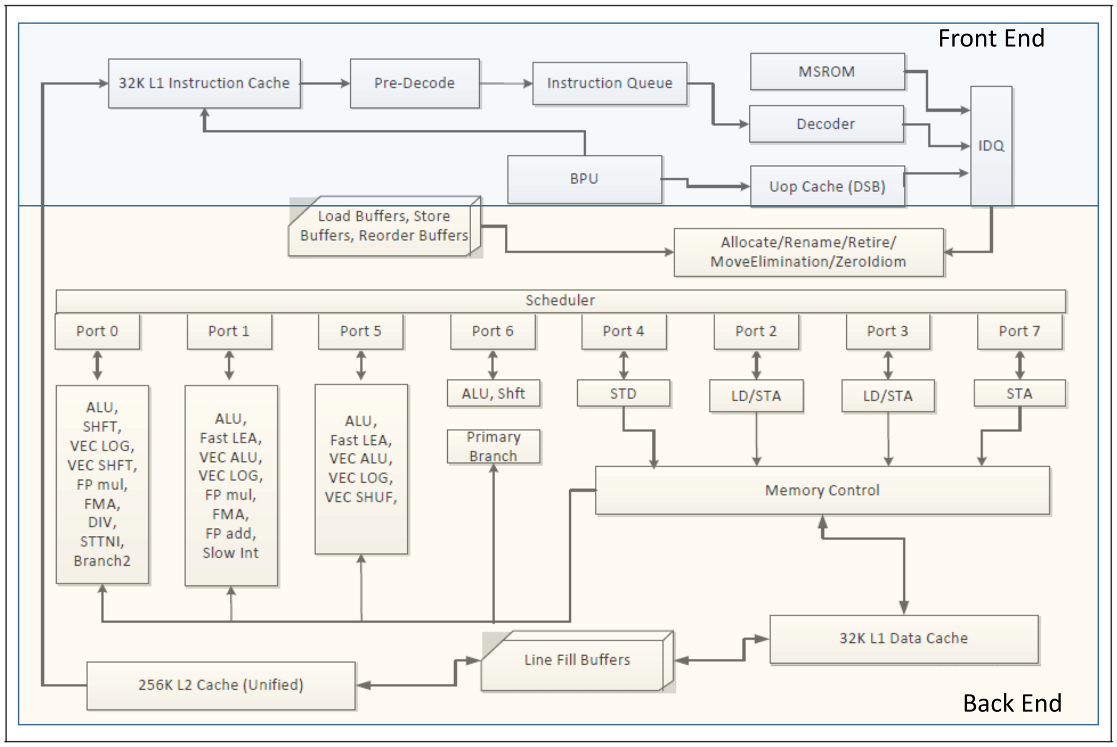


Figure 16. Block Diagram of Intel® Xen® E5 v4 Family processor Core Architecture.

Modern processor microarchitectures support over a thousand of events through their Performance Monitoring Units (PMUs). However, it is frequently non-obvious to determine which events are useful in detecting and fixing specific performance issues. It often requires an in-depth knowledge of both the microarchitecture design and PMU specifications to obtain useful information from raw event data. That is where the predefined events and metrics combined with the top-down characterization method help enormously, enabling conversion of the measured performance data into actionable information.

From the top-down perspective, one can think of the architecture as consisting of two main high-level functional blocks:

1) **Frontend** – responsible for fetching the program code represented in CPU architecture instructions and decoding them into one or more low-level hardware operations called µOps.

2) **Backend** – responsible for scheduling µOps, µOps execution and their retiring per original program's order.

The first analysis point is whether the µOps are issued to the execution pipeline. If they are not issued, either the architecture *Frontend* is the bottleneck or the *Backend* is the stalling party. On the other hand, if µOps are issued then EITHER most of them are executed, completed and can be *Retired* OR some of them are executed, but not completed due to *Bad Speculation* and cannot get retired. *Figure 17* shows a flow diagram for this simple analysis process.

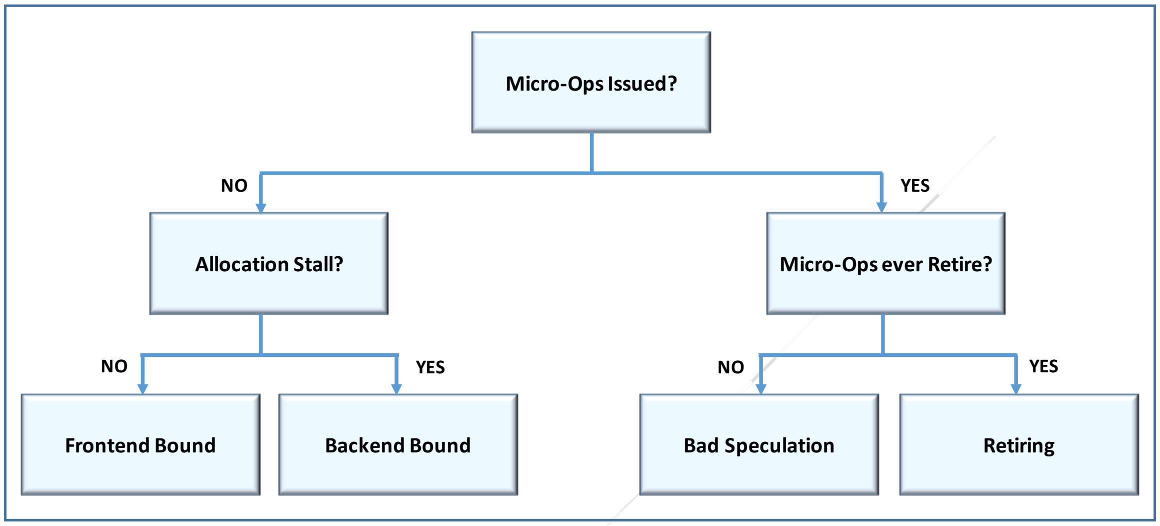


Figure 17. Logical steps in TMAM.

The first level of TMAM break-up helps a user to focus on one or two specific domains, which could influence the performance the most and then drill down into the second level of hierarchy. The process is repeated to the deeper levels until performance issue is found. *Figure 18* shows the first four levels of TMAM hierarchy for Intel® Xeon® E5 v4 Family processor microarchitecture.

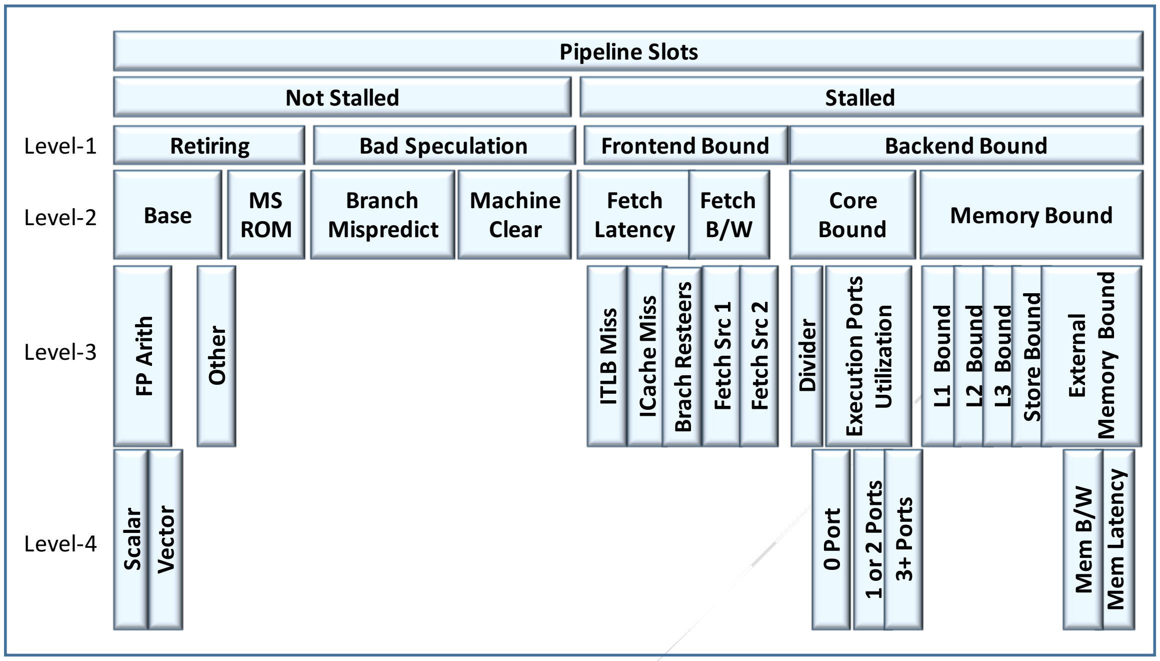


Figure 18. TMAM Hierarchy

For further detail on TMAM analysis please refer to Intel Developer Zone resources[18].

TMAM data collection and analysis can be carried out using Intel *VTune*^{TM} or PMU-tools. Benchmark tests detailed in this paper were conducted using EMON tool (Part of Intel *VTune*^{TM}) for both data collection and the analysis, mainly because of availability of the test automation framework in the testing environment. Exactly the same data collection and analysis is available and can be carried out using PMU-tools. Associated PMU-tools command references for each of the analysis steps have been provided in *Section 15. Appendix: Deep-dive TMAM Analysis*.

TMAM performance analysis has been applied to the NF workloads described in this paper. *Table 9* shows TMAM Level-1 measurements for the benchmarked CoreMark and NF workloads running under load on Intel® Xeon® E5-2699v4 processor, with data plane threads utilizing single processor core, with processor running in no-Hyper-Threading (noHT) and then in Hyper-Threading (HT) mode.

[18] Intel Developer Zone, Tuning Applications Using a Top-down Microarchitecture Analysis Method, https://software.intel.com/en-us/top-down-microarchitecture-analysis-method-win.

Core Pipeline Slots	Not Stalled				Stalled			
TMAM Level-1 Metrics	%Retiring		%Bad_ Speculation		%Frontend_Bound		%Backend_Bound	
Processor Mode: noHT, HT	noHT	HT	noHT	HT	noHT	HT	noHT	HT
CoreMark	53.6	67.7	3.2	2.4	6.8	20.1	36.3	9.8
DPDK-Testpmd L2 Loop	34.1	47.0	3.8	4.7	1.1	14.8	61.1	33.4
DPDK-L3Fwd IPv4 Forwarding	36.9	51.8	0.6	0.8	0.9	22.0	61.6	25.4
VPP L2 Patch Cross-Connect	47.6	57.8	1.7	0.6	3.4	16.9	47.3	24.7
VPP L2 MAC Switching	52.4	66.4	1.1	0.4	2.7	15.9	43.8	17.3
OVS-DPDK L2 Cross-Connect	44.6	57.7	7.4	3.9	10.9	26.4	37.0	12.0
VPP IPv4 Routing	57.4	67.4	1.1	0.8	2.5	14.8	38.9	17.0

Table 9. TMAM Level-1 Analysis.

All listed percentage-based Level 1 TMAM performance metrics add up to 100% (per workload type and per noHT/HT test run), representing a ratio of all pipeline slots executed over the measurement time.

Initial look at these results shows some interesting patterns and differences across the tested workloads.

Starting with *%Retiring* (ratio of pipeline slots the µOps are successfully executed and retired, higher value is better), the highest scores at **66.4..67.7%** (HT mode) are achieved by **CoreMark** and surprisingly **VPP IPv4 Routing** and **VPP L2 MAC Switching** indicating that these are extremely efficient and optimized code implementations, which are successfully hiding cache and memory latencies while processing packets.

Following with *%Bad_Speculation* (ratio of pipeline slots pre-fetching and executing non-useful operations, lower value is better), the clear winner is **VPP** (all configurations) and **DPDK-L3Fwd**, scoring values **<1%**. This is an extremely low value indicating that for these applications the core is speculatively executing the branches correctly most of the time, spending 99% of cycles doing useful work. Besides efficient CPU branch predictor architecture implementation, several other factors, such as the efficient code compilation favorable to branch predictor (e.g. using compiler hints likely()/unlikely()), and small and repeated code execution footprint play a role to make this metric small.

Measurements of *%Frontend_Bound* Stalls (ratio of pipeline slots the Frontend fails to supply the pipeline at full capacity when there were no Backend bound stalls, lower value is better) and *%Backend_Bound* Stalls (ratio of pipeline slots the µOps are not delivered from µOp queue, denoted as IDQ in *Figure 16,* to the pipeline due to Backend being out of resources to accept them, lower value is better) clearly show different balance between both metrics for noHT and HT modes. In noHT case *%Backend_Bound* stalls dominate for all workloads, indicating the Frontend is not a pinch point. In Hyper-Thread mode this changes as each HT core tries to dispatch instruction to maximize the use of Backend resources, while also stressing the shared Frontend pipe line. This results in the aggregate contribution from Backend related stalls decreasing, while Frontend becoming increasingly a bottleneck.

Following sections delve into further analysis of the first two levels of TMAM measurements, describing each category in more detail and providing related measurement and analysis of benchmarked NF workloads.

6.2 %Retiring

The first category for TMAM top level analysis is *%Retiring*. This metric is the measure of the number of µOps successfully retired and hence represents the execution efficiency.

The Retiring ratio is calculated as follows:

$$\%Retiring = \frac{UOPS\_RETIRED.RETIRE\_SLOTS}{4 * CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 7. TMAM Level-1: %Retiring.

The event *UOPS_RETIRED.RETIRE_SLOTS* counts the number of retirement slots used each clock cycle. The theoretical best-case scenario for Intel® Xeon® E5 v4 Family processor architecture is one with 4 µOps being retired per cycle, one on each slot. Clearly, this metric should be as high as possible. Since most architecture instructions map to single µOp, this metric is also indicative of instructions-per-cycle (IPC).

When benchmarks are run with Hyper-Threading, each thread competes for the use of execution and retirement slots. As a result, increased number of µOps are retired, giving *%Retiring* ratio higher than corresponding non-Hyper-Threading cases. This explains higher values for HT mode reported in *Table 9* above.

Higher *%Retiring* ratio means more µOps are executed and retired per cycle. However, higher ratio does not necessarily imply that there is no room for further performance improvement of an application. Further optimization options include using better algorithms or vectorizing the code (with SSE4, AVX2 instructions), making CPU cores do the same work with lower number of µOps consuming less CPU clock cycles.

Comparing benchmarked NF workloads, the highest scores at **66.4..67.7%** (HT mode) are achieved by **CoreMark** and **VPP IPv4 Routing** and **VPP L2 MAC Switching**. Remaining workloads are 10..20% behind. For all workloads using HT improves %Retiring metric by 17% to 40%.

Note that there is one subtle difference between **CoreMark** and benchmarked NF workloads. Whereas the former deals with the same set of data in L1 cache, the NF workloads continuously access and operate upon the new set of data written and read by network interface cards. High values of this metric for NF workloads indicate that they are not penalized by the latency of accessing dynamic data content written/read by external devices. This is the result of the efficient NF code implementations AND CPU hardware prefetchers, that when combined help hiding the cache and memory access latencies resulting in very high number of instructions being retired per clock.

Retiring pipeline slots are broken further into two sub-categories called *Base* and *Microcode_Sequencer*. *Table 10* lists TMAM Level-2 drill-down into the %Retiring measurements for the benchmarked CoreMark and NF workloads.

Core Pipeline Slots	Not Stalled					
TMAM Level-1&2 Metrics	%Retiring		%..Base		%..Microcode_ Sequencer	
Processor Mode: noHT, HT	noHT	HT	noHT	HT	noHT	HT
CoreMark	53.6	67.7	53.6	67.7	0	0
DPDK-Testpmd L2 Loop	34.1	47.0	34.0	47.0	0.0	0.0
DPDK-L3Fwd IPv4 Forwarding	36.9	51.8	36.9	51.6	0.1	0.2
VPP L2 Patch Cross-Connect	47.6	57.8	47.2	57.4	0.3	0.4
VPP L2 MAC Switching	52.4	66.4	52.1	65.7	0.2	0.7
OVS-DPDK L2 Cross-Connect	44.6	57.7	44.3	57.4	0.4	0.3
VPP IPv4 Routing	57.4	67.4	57.2	67.1	0.2	0.3

Table 10. TMAM Level-1: %Retiring - breakdown into Level-2 metrics.

Listed Level-2 TMAM metrics add up to their parent Level-1 metric, representing a further subdivision ratio of all pipeline slots executed over the measurement time.

6.2.1 %Retiring.Base

%Retiring.Base metric represents ratio of pipeline slots when the CPU core was retiring regular µOps, the ones that did not originate from the microcode-sequencer. Software logic could be rewritten to improve this metric and further reduce the instruction count that require microcode-sequencer. Alternatively, software could also employ other techniques such as vectorization.

%Retiring.Base ratio is calculated as follows:

$$\%\,Retiring.Base = 1 - \frac{IDQ.MS\_UOPS}{UOPS\_ISSUED.ANY} * \%Retiring = 1 - \%\,Retire.MS$$

Equation 8. TMAM Level-2: %Retiring.Base.

It is clear that this metric is the dominant factor for the reported benchmarks.

6.2.2 %Retiring.Microcode_Sequencer

Certain instructions in Intel Architecture are complex and are broken into multiple µOps using Micro Sequencer logic. Examples of such instructions include CPUID, sine, and cosine. Such instructions can also potentially slow down the execution. For performance reasons, software implementation should minimize the use of such instructions.

$$\%\,Retiring.MS = \frac{IDQ.MS\_UOPS}{UOP\_ISSUED.ANY} * \%Retiring$$

Equation 9. TMAM Level-2: %Retiring.Microcode_Sequencer.

High use of Micro Sequencer is usually an indication of a performance issue. However, some instructions, such as REP MOVSB (memory string moves), make heavy use MS, yet work every efficiently. It is evident from the *Table 10* above, that use of micro sequencer logic is insignificant for the benchmarked applications.

6.3 %Bad_Speculation

The second top-level category in TMAM, %*Bad_Speculation*, quantifies a scenario when the pipeline is busy fetching and executing non-useful operations.

Intel Xeon architecture employs a sophisticated speculative branch prediction logic to attain high execution efficiency. It is very likely that multiple instructions are being speculatively executed ahead of making a decision on a conditional branch instruction. If the branch prediction is proven to be incorrect at the execution of the branch instruction, instructions that were speculatively executed never retire and the execution slots they used are essentially wasted. Besides, such mis-predictions also cause pipeline flushes consuming additional cycles, while recovering to the correct execution flow. The *%Bad_Speculation* metric accounts for both of these effects. Clearly, one would like to have *%Bad_Speculation* number as low as possible.

Overall cycles wasted are captured as:

$$\%\,Bad\_Speculation = \frac{((UOPS\_ISSUED.ANY\ -\ UOPS\_RETIRED.RETIRE\_SLOTS)\ +\ 4 * INT\_MISC.RECOVERY\_CYCLES\_ANY)}{4 *\ CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 10. TMAM Level-1: %Bad Speculations.

Table 11 below lists TMAM Level-2 drill-down into the *%Bad_Speculation* measurements for the benchmarked **CoreMark** and NF workloads.

Core Pipeline Slots	Not Stalled					
TMAM Level-1&2 Metrics	%Bad_ Speculation		%..Branch_ Mispredicts		%..Machine_ Clears	
Processor Mode: noHT, HT	noHT	HT	noHT	HT	noHT	HT
CoreMark	3.2	2.4	3.1	2.3	0.1	0.1
DPDK-Testpmd L2 Loop	3.8	4.7	3.7	4.7	0.1	0.1
DPDK-L3Fwd IPv4 Forwarding	0.6	0.8	0.4	0.7	0.2	0.1
VPP L2 Patch Cross-Connect	1.7	0.6	0.9	0.4	0.8	0.2
VPP L2 MAC Switching	1.1	0.4	0.8	0.3	0.4	0.1
OVS-DPDK L2 Cross-Connect	7.4	3.9	7.1	3.9	0.3	0.0
VPP IPv4 Routing	1.1	0.8	0.7	0.6	0.5	0.2

Table 11. TMAM Level-2: %Bad_Speculation - breakdown into Level-2 metrics.

Based on data listed in *Table 11*, benchmarked workloads show very low ratio for this category, indicating only a small amount of cycles are wasted due to bad speculations. It implies that the most branch instructions are predicted correctly by the underlying architecture. Also, *%Machine_Clears* metric is less dominant than *%Branch_Mispredicts*.

Among tested workloads, **VPP** scores consistently values **<1%** for all tested configurations. Same for **DPDK-L3Fwd**. **CoreMark**, **OVS-DPDK**, **DPDK-Testpmd** utilize pre-fetching less effectively, but still score fairly low values **2.4..4.7%**. HT improves this metric in all cases, but DPDK-Testpmd.

TMAM methodology classifies the *%Bad_Speculation* slots into the two Level-2 performance metrics: *%Branch_Mispredicts* and *%Machine_Clears*.

6.3.1 %Bad_Speculation.Branch_Mispredicts

This metric tells about the percentage of wasted cycles due to *Branch Mispredicts* events. Understanding this counter helps make the program control flow friendlier to the branch predictor.

$$\%\,Branch\_Mispredicts = \frac{(BR\_MISP\_RETIRED.ALL\_BRANCHES\_PS)}{(BR\_MISP\_RETIRED.ALL\_BRANCHES\_PS + MACHINE\_CLEARS.COUNT)} * (\%Bad\ Speculation)$$

Equation 11. TMAM Level-2: %Bad_Speculation.Branch_Mispredicts.

The event `BR_MISP_RETIRED.ALL_BRANCHES_PS` counts the number of branches that are incorrectly predicted as the branch target.

All tested workloads exhibit low values for %*Branch_Mispredict*s, well below 8%.

If above %*Branch_Mispredict*s ratios is more than 5%, hot-spot profiling for *%Bad_Speculation* and above two events is recommended. The tools such as Linux *Perf* or VTune^{TM} can help rearrange the logic to help minimize these events.

6.3.2 %Bad_Speculation.Machine_Clears

The event `MACHINE_CLEARS.COUNT` counts the number of times the pipeline is cleared due to various Machine Clear events. The prominent causes of Machine Clear event include memory ordering conflict and self-modifying code. For example, an ordering conflict can occur when a snoop request is issued and the machine is uncertain if memory ordering will be preserved as another core is in the process of modifying the same data.

$$\%\,Machine\_Clears = \frac{(MACHINE\_CLEARS.COUNT)}{(BR\_MISP\_RETIRED.ALL\_BRANCHES\_PS + MACHINE\_CLEARS.COUNT)} * (\%Bad\ Speculation)$$

Equation 12. TMAM Level-2: %Bad_Speculation.Machine_Clears.

Since the benchmarked NF workloads employ run-to-completion packet processing, the participating cores are not transferring data between themselves. Hence, ordering conflict is not expected for any of the workloads. Furthermore, these workloads do not use self-modifying code. These two main factors make %Machine_Clears value low and optimal.

6.4 %Frontend_Bound Stalls

The Frontend of the pipeline on recent Intel Xeon E5 processor microarchitectures can allocate up to four µOps per cycle, while the Backend can retire four µOps per cycle. *%Frontend* bound stalls denote the ratio of pipeline slots the Frontend fails to supply the execution pipeline at full capacity and delivers less than 4 µOps per cycle, while the Backend is still requesting µOps. Refer to *Section 6.1 TMAM Overview* and *Figure 16* for details on the Frontend and Backend functionality.

The *Frontend_Bound* category covers also several other types of pipeline stalls. While it is less common for the Frontend portion of the pipelines to become the application's bottleneck, there are some cases where the Frontend can contribute in a significant manner to machine stalls.

%Frontend_Bound stalls metric is calculated using the following formula:

$$\% \, Frontend\_Bound = \frac{IDQ\_UOPS\_NOT\_DELIVERED.CORE}{4 * CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 13. TMAM Level-1: %Frontend_Bound.

The `IDQ_UOPS_NOT_DELIVERED.CORE` counts the number of issue pipeline *slots* at every core clock when no µOp was delivered from the Frontend to the Backend while there is no Backend stall. Higher ratio means that Frontend of the pipeline is delivering less than 4 uops when Backend is demanding µOps, but cannot get enough. In all of these cases execution engines are starved due to Frontend stalls.

In TMAM hierarchical approach, *%Frontend_Bound* can be divided further into two Level-2 metrics – *%Frontend_Latency* and *%Frontend_Bandwidth*, as described in the next two sections.

Table 12 below shows the TMAM Level-2 drill-down into the %Frontend_Bound Stalls measurements for the benchmarked CoreMark and NF workloads.

Core Pipeline Slots	Stalled					
TMAM Level-1&2 Metrics	**%Frontend_ Bound**		**%..Frontend_ Latency**		**%..Frontend_ Bandwidth**	
Processor Mode: noHT, HT	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**
CoreMark	6.8	20.1	1.9	11.5	4.9	8.5
DPDK-Testpmd L2 Loop	1.1	14.8	0.6	12.3	0.5	2.6
DPDK-L3Fwd IPv4 Forwarding	0.9	22.0	0.4	16.0	0.4	6.1
VPP L2 Patch Cross-Connect	3.4	16.9	1.7	13.7	1.6	3.2
VPP L2 MAC Switching	2.7	15.9	1.6	11.7	1.1	4.2
OVS-DPDK L2 Cross-Connect	10.9	26.4	3.4	16.4	7.5	10.0
VPP IPv4 Routing	2.5	14.8	1.4	11.8	1.1	2.9

Table 12. TMAM Level-1: %Frontend_Bound Stalls - breakdown into Level-2 metrics.

Table 12 reveals the benchmarked CoreMark and NF workloads are not Frontend bound for non-Hyper-Threading cases. In other words, Frontend is always ready to deliver µOps when the Backend asks for them. When tests are carried out with Hyper-Threading, the execution engine demands increased number of µOps pertaining to the software running on both hyper-threads of a core. Increased pressure on the Frontend to provide more µOps to the Backend results in higher number of Frontend stalls.

For all benchmarks, %Frontend metric is less than 11% for non-HT mode and less than 26% for HT mode. For well optimized workloads Frontend bound stalls are expected to be below 30%, and optimizing it further down is unlikely to yield any substantial performance improvements. All benchmarked applications fall into this category with **VPP IPv4 Routing** and **DPDK-**Testpmd L2 Loop scoring values **<15%**.

For more information please refer to Section B 5.7 in *Intel® 64 and IA-32 Architectures Optimization Reference Manual*[19].

6.4.1 %Frontend.Frontend_Latency

This metric indicates how often a CPU core was stalled due to latency issues at the Frontend of the pipeline. It includes the Frontend stalls caused by Instruction-cache misses, iTLB[20] misses, branch mispredictions, and those resulting from µOps delivery switching back and forth between decoded I-Cache and legacy decoder. In such cases, the Frontend delivers no µOps for some number of clocks while Backend was requesting them.

$$\%\,Frontend\_Latency = \frac{IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_0\_UOPS\_DELIV.CORE}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 14. TMAM Level-2: %Frontend_Bound.Frontend Latency.

The event IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE counts the core cycles in which no µOp is delivered from the Frontend to the Backend in any of the 4 pipeline slots while there was *no Backend stall*.

Results in *Table 12* above indicate that both DPDK and VPP code has been written in a way to minimize Instruction Cache and iTLB misses. The respective next level down TMAM performance event counts confirm these misses are indeed negligible. In addition as the %*Branch_Mispredict*s metric is also negligible. This indicates that it is the back and forth switching between decoded I-Cache and legacy decoder are the main source of higher Frontend Latency metric values.

6.4.2 %Frontend_Bound.Frontend_Bandwidth

This metric quantifies the fraction of slots a logical core was stalled due to Frontend bandwidth issues. For example, inefficiencies at the instruction decoders, or code restrictions for caching in the DSB[21] (decoded µOps cache) are categorized under *%Frontend_Bandwidth*. In such cases, the Frontend typically delivers non-optimal amount of µOps to the Backend (i.e. less than four per clock cycle in Intel® Xeon ® E5 v4 Family microarchitecture).

$$\%\,Frontend\_Bandwidth = \frac{(UOPS\_NOT\_DELIVERED.CORE - 4 * IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_0\_UOPS\_DELIV.CORE)}{4 * CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 15. TMAM Level-2: %Frontend_Bound.Frontend Bandwidth.

%Frontend Bandwidth values listed for tested workload in *Table 12* are very low and should not cause any major impact on performance. Optimal code is expected to yield

[19] Intel® 64 and IA-32 Architectures Optimization Reference Manual - https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[20] iTLB - Translation Look Aside Buffer for Instructions. This high speed on-chip SRAM that caches logical to physical address transaction entries for instruction space.

[21] DSB - Decoded Stream Buffer, caches a small set of µOps instructions after decoding.

values below or around 10%, and all benchmarked workloads meet this criterion. Values higher than 10% could be a cause of concern and usually call for further investigation and optimization.

6.5 %Backend_Bound Stalls

As noted previously, the Backend can retire up to four µOps per cycle in Intel® Xeon® E5 processor microarchitecture. The %Backend bound stalls denote the slots where the µOps are not delivered from µOp queue (IDQ) to the execution pipeline because the Backend did not have free resources to accept them.

The majority of un-optimized applications have a high value of %Backend Bound. Resolving Backend issues is often about resolving sources of load and store latencies that cause µOp retirement to take longer than necessary.

A simple formula for *%Backend* bound stall is:

$$\%Backend\_Bound = 1 - (\%Frontend\_Bound + \%Bad\_Speculations + \%Retiring)$$

Equation 16. TMAM Level-1: %Backend_Bound.

Table 13 below lists the TMAM Level-2 drill-down into the %Backend_Bound Stalls measurements for the benchmarked CoreMark and NF workloads.

%Backend_Bound stalls are further divided into two distinct categories: *%Memory* Bound and *%Core* Bound.

Core Pipeline Slots	Stalled					
TMAM Level-1&2 Metrics	**%Backend_Bound**		**%..Memory**		**%..Core**	
Processor Mode: noHT, HT	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**
CoreMark	36.3	9.8	8.5	3.2	27.8	6.4
DPDK-Testpmd L2 Loop	61.1	33.4	37.1	19.6	24.0	13.8
DPDK-L3Fwd IPv4 Forwarding	61.6	25.4	37.5	14.5	24.1	10.9
VPP L2 Patch Cross-Connect	47.3	24.7	24.4	14.3	22.9	10.3
VPP L2 MAC Switching	43.8	17.3	19.3	8.9	24.6	8.3
OVS-DPDK L2 Cross-Connect	37.0	12.0	15.2	6.2	21.8	5.7
VPP IPv4 Routing	38.9	17.0	18.3	8.8	20.6	8.2

Table 13. TMAM Level-1: %Backend_Bound Stalls - breakdown into Level-2 metrics.

At the high-level, the Backend bound measurements show high values for all noHT cases. CoreMark, which operates on L1 cache only, is among the lowest for this metric, closely followed by VPP IPv4 routing and OVS-DPDK L2 Cross-Connect. DPDK-Testmpmd L2 Loop and DPDK-L3Fwd show the highest degree of stalls in noHT case. When workloads run in HT mode, the metric improves drastically as one thread can continue to execute in parallel while the sibling one is waiting either for memory systems and core execution unit to allocate more resources.

6.5.1 %Backend_Bound.Memory_Bound

%Memory bound metric corresponds to the stalls pertaining to accesses to the memory system i.e. cache hierarchy, system memory, and store buffers. The misses at various levels of caches are usually the main contributors to this category. The formula for

counting %Memory bound is a bit complex, involving a number of conditional statements, and can be found in PMU-tools scripts *bdx_server_rations.py*[22].

%Memory_Bound stalls can be further drilled down to *%L1_Bound*, *%L2_Bound*, *%L3_Bound* (Last Level Cache bound), *%System_Memory_Bound*, and *%Store_Bound*.

Table 14 below lists further break-down of *%Backend_Bound.Memory_Bound* Level-2 metric into the Level-3 constituent measurements for the benchmarked CoreMark and NF workloads.

Core Pipeline Slots	Stalled											
TMAM Level-2&3 Metrics	%Backend_ Bound.Memory		%..L1_Bound		%..L2_Bound		%..L3_Bound		%..System_ Memory_Bound		%..Store_ Bound	
Processor Mode: noHT, HT	noHT	HT	noHT	HT	noHT	HT	noHT	HT	noHT	HT	noHT	HT
CoreMark	8.5	3.2	14.1	19.1	0	0	0	0.1	0	0.3	0	0
DPDK-Testpmd L2 Loop	37.1	19.6	5.4	9.1	5.4	6.9	7.9	14.4	0.0	0.0	37.6	22.6
DPDK-L3Fwd IPv4 Forwarding	37.5	14.5	7.8	10.7	1.8	3.6	12.3	16.5	0.0	0.0	25.9	13.6
VPP L2 Patch Cross-Connect	24.4	14.3	4.2	11.5	0.0	0.0	15.7	16.9	0.0	0.0	25.4	19.5
VPP L2 MAC Switching	19.3	8.9	3.7	12.0	0.0	0.0	14.6	14.6	0.0	0.0	14.7	9.3
OVS-DPDK L2 Cross-Connect	15.2	6.2	5.1	12.1	0.3	0.9	10.7	14.9	0.0	0.0	13.0	7.1
VPP IPv4 Routing	18.3	8.8	3.9	10.9	0.0	0.0	12.6	12.9	0.0	0.0	16.4	10.9

Table 14. TMAM %Backend_Bound.Memory_Bound - further breakdown statistics.

Note that *%L1_Bound*, *%L2_Bound* and *%L3_Bound* metrics, unlike other described TMAM metrics, are calculated by taking the ratio of the number of clocks consumed for accessing L1/L2/L3 caches, System_Memory and Store units) to the core (unhalted/working) clocks. There is no direct relation between these metrics and %Backend_Bound.Memory metric, hence the metrics do not add up.

Efficient utilization of CPU core cache hierarchy is extremely important, as it enables hiding the latency of accessing DRAM memory. Functionality offered by core cache hierarchy is analogous to fast memory (e.g. SRAM) used in purpose-built network forwarding processors to deliver high-speed data plane performance for network applications.

6.5.1.1 %Backend_Bound.Memory_Bound.L1_Bound

The metric %L1 bound represents the percentage of cycles for which a core is stalled to access data present in the L1 cache. Normally access to L1 cache has the lowest latency. However, a core may encounter higher latency in some cases such as a load was blocked on the older stores, DTLB miss, and so on.

$$\%L1\_Bound = \frac{(CYCLE\_ACTIVITY.STALLS\_MEM\_ANY - CYCLE\_ACTIVITY.STALLS\_L1D\_MISS)}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 17. TMAM Level-3: Backend_Bound.Memory_Bound.L1_Bound.

The event CYCLE_ACTIVITY.STALLS_MEM_ANY counts the core cycles while memory subsystem (any level of cache or memory) has an outstanding load.

The event CYCLE_ACTIVITY.STALLS_L1D_MISS counts the core cycles while L1 cache miss demand load is outstanding (data is served from sources other than L1 cache).

[22] PMU-tools, bdx_server_rations.py - https://github.com/andikleen/pmu-tools/blob/master/bdx_server_ratios.py.

Note that idle latency (only one outstanding read at a time) for accessing L1 cache is 4 to 5 cycles for E5 Xeon® processor architecture.

6.5.1.2 %Backend_Bound.Memory_Bound.L2_Bound

The metric %L2 bound denotes percentage of cycles a core stalls while loading data which was present in L2 cache.

$$\%L2\_Bound = \frac{(CYCLE\_ACTIVITY.STALLS\_L1D\_MISS - CYCLE\_ACTIVITY.STALLS\_L2D\_MISS)}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 18. TMAM Level-3: Backend_Bound.Memory_Bound.L2_Bound.

The event CYCLE_ACTIVITY.STALLS_L2D_MISS counts the core cycles while L2 cache miss demand load is outstanding. Note that idle latency (only one outstanding read at a time) for accessing L2 cache is 12 cycles for E5 Xeon® processor architecture.

This metric is zero for all **CoreMark** and **all VPP** configurations, and close to zero for all other benchmarked workloads. This implies that either the cores are not accessing data present in L2 cache, or cores are accessing data residing in L2 cache but they do not encounter stalls, as hardware and/or software prefetchers hide the L2 cache access latencies.

6.5.1.3 %Backend_Bound.Memory_Bound.L3_Bound

The metric %L3_Bound denotes percentage of cycles the cores stall, while loading data that was present in L3 cache (LLC) or contented with the sibling core.

$$L3\_Hit\_Fraction = \frac{\mathrm{MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT}}{(\mathrm{MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT} + 7 * \mathrm{MEM\_LOAD\_UOPS\_RETIRED.L3\_MISS})}$$

$$\%L3\_Bound = \frac{L3\_Hit\_Fraction * CYCLE\_ACTIVITY.STALLS\_L2D\_MISS}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 19. TMAM Level-3: Backend_Bound.Memory_Bound.L3_Bound.

The event MEM_LOAD_UOPS_RETIRED.L3_HIT counts the Retired load uops with L3 cache hits as data sources.

The event MEM_LOAD_UOPS_RETIRED.L3_MISS counts the Retired load uops with L3 cache miss as data sources.

Note that idle latency for accessing LLC is ~40 cycles on E5 Xeon processor architecture.

Clearly, CoreMark application is not L2, L3 cache and System Memory bound since it executes from L1 cache and its complete data footprint fits into L1 cache. In contrast, all NF benchmarked workloads exhibit higher percentage in the %L3 bound metric compared to %L1, %L2. This indicates that cores spend more cycles while accessing L3 (LLC) cache. As it will be explained later, NF applications, by making effective use the DDIO technology, read and write packets and descriptors from LLC rather than System memory due to the effective use of DDIO technology. The stalls encountered while accessing L3 are reflected in the %L3 bound metric.

DDIO technology is described and its applicability discussed later in this paper in *Section 8 PCIe Performance Analysis*.

6.5.1.4 %Backend_Bound.Memory_Bound.System_Memory

This metric indicates how often a core was stalled while accessing external memory. Note that idle latency (measured when a core issues only one outstanding memory read request at a time) for accessing memory cache is in range of 140+ cycles for E5 Xeon® processor architecture.

$$\%System\_Memory\_Bound = \frac{(1 - L3\_Hit\_Fraction) * CYCLE\_ACTIVITY\ \ STALL\_L2DMISS}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 20. TMAM Level-3: Backend_Bound.Memory_Bound.System_Memory.

Owing to effective use of DDIO by the benchmarked NF applications, the metric %System_Memory Bound is close to zero. The LLC essentially acts as a fast SRAM to help exchange packets between Ethernet ports and the cores at high rate. It logically serves the same purpose as the multiple fast SRAMs employed in the purpose-built network processors or ASICs.

6.5.1.5 %Backend_Bound.Memory.Store_Bound

The Store bound category indicates the fraction of cycles where store buffers are full. In out-of-order architecture, the store operations are executed after the retirement of store instructions. Note that the pressure on store buffers does not necessarily means execution stalls. However, such saturation of Store buffers may cause low utilization of the execution ports.

$$\%Store\_Bound = \frac{RESOURCE\_STALLS.SB}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

Equation 21. TMAM Level-3: Backend_Bound.Memory_Bound.Store_Bound.

The event RESOURCE_STALLS.SB counts the cycles stalled due to no store buffers available.

The possible reasons behind store buffers saturations include frequent false sharing of data between the cores, higher store latency, DTLB misses on stores, store data crossing cacheline boundary. For **DPDK Testpmd L2 loop**, **DPDK-L3fwd IPv4 Forwarding**, and **VPP L2 Patch Cross-Connect** workloads, this metric from noHT case is unusually high (>25%). The exact cause behind this has not been determined.

6.5.2 %Backend_Bound.Core Bound

%Core bound category corresponds to the execution starvation or sub-optimal utilization of execution ports. It represents the pipeline slots fraction where the core is the bottleneck for non-memory related situations. This metric can be calculated as follows:

$$\%Core\_Bound = \%Backend\_Bound - \%Backend\_Bound.Memory$$

Figure 19. TMAM Level-3: %Backend_Bound.Core Bound.

Table 15 below lists further break-down of *%Backend_Bound.Core* bound Level-2 metric into the Level-3 constituent measurements for the benchmarked CoreMark and NF workloads.

The metric *%Divider* denotes the fraction of cycles in which the Divider unit (depicted as Port0 exec unit in *Figure 15*) was active executing divide operations. Since none of the benchmarks are using heavy arithmetic divide functions, the contribution from this metric is negligible.

The metric *%Ports_Utilization* represents fraction of cycles where the performance was limited due to core compute stalls other than divider operations or memory stalls. Higher value denotes lack of instruction level parallelism. Heavy dependency among the contiguous instructions hampering parallel execution of instructions in the execution units would drive high values of this metric. In addition, any code sequence oversubscribing certain execution unit (other than the divider) would also contribute to this metric.

Core Pipeline Slots	**Stalled**					
TMAM Level-2&3 Metrics	**%Backend_ Bound.Core**		**%..Divider**		**%..Ports_ Utilization**	
Processor Mode: noHT, HT	**noHT**	**HT**	**noHT**	**HT**	**noHT**	**HT**
CoreMark	27.8	6.4	0	0	46.2	36.1
DPDK-Testpmd L2 Loop	24.0	13.8	0.0	0.0	36.4	21.2
DPDK-L3Fwd IPv4 Forwarding	24.1	10.9	0.0	0.0	30.7	24.0
VPP L2 Patch Cross-Connect	22.9	10.3	0.4	0.5	37.9	24.3
VPP L2 MAC Switching	24.6	8.3	0.2	0.2	37.0	25.9
OVS-DPDK L2 Cross-Connect	21.8	5.7	0.0	0.0	42.0	25.7
VPP IPv4 Routing	20.6	8.2	0.2	0.3	32.1	24.4

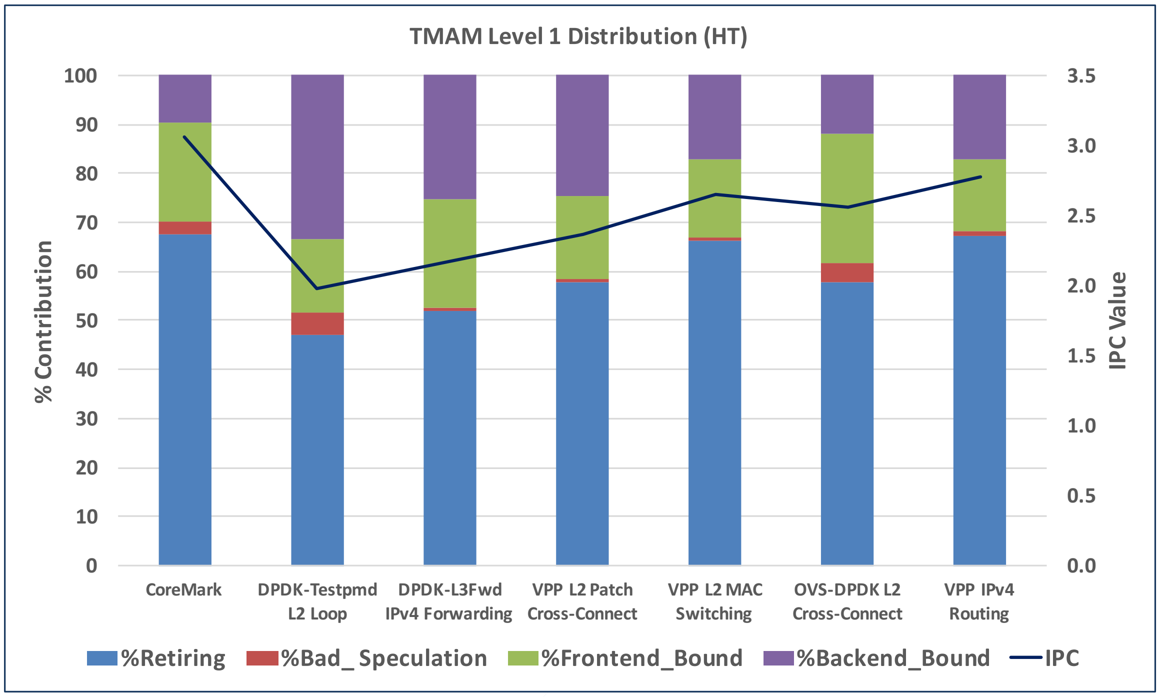
Table 15. TMAM Level-2: %Backend Core Bound - further breakdown statistics.

The causes behind higher %Ports_Utilization metric are not easy to pin point. In case of CoreMark and OVS-DPPK L2 Cross-connect with cores running in noHT mode, it is likely that higher counts against this metric are caused by the dependent instructions. Hyper-Threading does help reducing this count due to instruction level with two threads. It is interesting to note that Hyper-Threading for CoreMark is not improving this metric as much as other workloads. In this case, it is likely that oversubscribing of some execution unit is contributing to the core execution stalls. TMAM analysis at deeper levels could help understand the issue. Common best practice is to investigate %Ports_Utilization metric further, if it scores values of 25% or higher.

Core bound issues could be mitigated through better code generation through compiler. For example, compiler optimization flags could avoid sequence of dependent arithmetic instructions, could avoid divider stalls, and so on. On the other hand, software could be employing vectorization help mitigate the core stalls.

6.6 TMAM Measurements – Conclusions

TMAM provides a systematic performance characterization of the benchmarked workloads. *Figure 20* below shows the summary of TMAM Level-1 metrics' distribution and IPC for all benchmarked applications for both noHT (no Hyper-Threading enabled) and HT (Hyper-Threading enabled) cases.



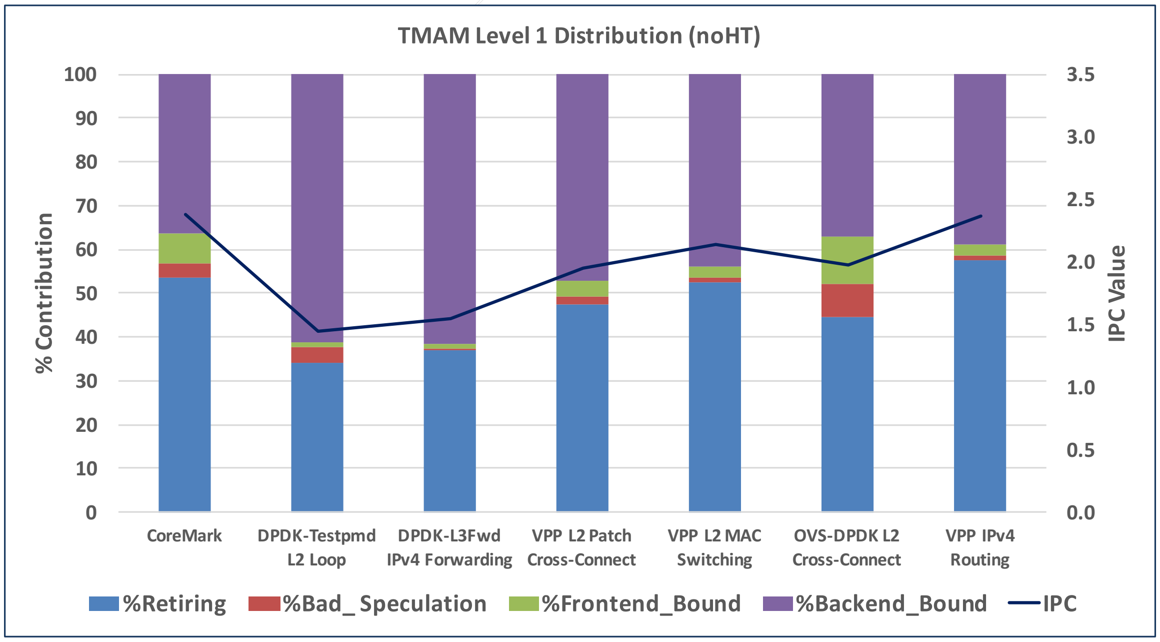


Figure 20. TMAM Level-1 Summary.

%Retiring: reflects the ratio of core pipeline slots the µOps are successfully executed and retired, relative to the maximum possible, <u>higher value is better</u>.

- In noHT mode, two tested workloads have low scores of less than 44%, **DPDK-Testpmd L2 Loop** and **DPDK-L3Fwd IPv4 Forwarding**. Compared to other NF workloads, they implement minimal packet processing focusing mainly on packet I/O operations. Packet I/O operations are Backend bound since a CPU core is mostly waiting for data written by NICs, making %Backend_bound stalls metric dominating for these two applications.
- In HT mode, the highest scores close to 70% are achieved by **CoreMark** and surprisingly **VPP IPv4 Routing** and **VPP L2 MAC Switching**. This clearly indicates not only extremely efficient and disciplined code execution, but also optimized code implementation, that is successfully hiding cache and memory latencies while processing packets.
- In general, both out-of-order execution and efficient code implementation play roles in getting %Retiring metric relatively high. It increases in Hyper-Threading mode indicating more instructions are being executed and work done per CPU core clock cycle. **IPC (#instr/cycle)** closely follows this metric very closely as expected, with **CoreMark and VPP IPv4 Routing scoring highest values, 3.1 and 2.8, respectively**.

%Bad_Speculation: represents the ratio of core pipeline slots pre-fetching and executing non-useful operations, <u>lower value is better</u>.

- This metric is insignificant for all workloads, with **lowest values <1%** measured for **VPP** (all configurations) and **DPDK-L3Fwd**.
- The efficient branch predictor unit in the Intel® Xeon® E5 v4 Family processor architecture plays a vital role in keeping this metric at a low value. In addition, the appropriate use of compiler hints in the VPP and DPDK source code help generate branch predictor friendly binaries.

%Frontend_Bound: captures the ratio of core pipeline slots the Frontend fails to supply the pipeline at full capacity, while there are no backend stalls, <u>lower value is better</u>.

- In noHT mode , the metric is relatively low indicating that the Frontend is ready most of the time to supply µOps when the Backend is ready to accept them.
- In HT mode, the metric increases compared with noHT case, as the number of instructions being issued increase with HT. The lowest values are measured for **VPP IPv4 Routing** and **DPDK-Testpmd L2 Loop** scoring values **<15%**. **OVS-DPDK** measured >26%. However, the percentage contribution for all tested applications is still below the threshold of 30% and as such not cause of concern. In all the cases, the main contributor to this metric is the Frontend latency. The main cause of Frontend latency is attributed to the wasted cycles due to back and forth transition between legacy and decoded I-Cache while delivering the µOps.

%Backend_Bound: represents the ratio of core pipeline slots the µOps are not delivered from µOp queue to the pipeline due to Backend being out of resources to accept them, <u>lower value is better</u>.

- In noHT mode, the metric values are high as all benchmarked workloads experience stalls caused by the loads from various levels of caches and store operations, and underuse of execution ports. **CoreMark**, which operates on L1 cache only, is among the lowest for this metric, closely followed by **VPP IPv4 Routing** and **OVS-DPDK L2 Cross-Connect**. **DPDK-Testmpmd L2 Loop** and **DPDK-L3Fwd IPv4 Forwarding** show the highest degree of stalls in noHT mode.
- In HT mode, the metric values get substantially reduced, due to parallel thread execution and threads balancing the use of execution resources. In addition due to pipelining of load requests from both threads, average cache access latency goes down. **CoreMark** scores lowest value here, due to its operation only L1 cache. It is closely followed by **OVS-DPDK L2 Cross-connect**, **VPP IPv4 Routing** and **VPP L2 MAC Switching**, all scoring <20%.
- This metric is important to understand since it reveals the penalty associated while accessing various levels of cache hierarchy and system memory, and store operations. Besides, it captures the possible inefficiency in the executions of µOps. Higher value of this metric negatively impacts %Retiring metric and hence the IPC.
- The %System_memory utilization (component %Backend_bound metric) for all benchmarked workloads is zero, unlike for many other benchmarks and application workloads. This simply means that all memory references are served by various levels of caches rather than memory. This is due to the optimal use of DDIO, heavy use of vectorization, and software pre-fetching that all help hide cache and memory latencies.

In summary, TMAM analysis provides a good and fairly straightforward performance measurement and analysis methodology to systematically assess levels of Software application optimization, and adapts very well to Network Function workloads. In addition to helping to identify any system bottlenecks across the Software-Hardware stack running NF workloads, it also allows for a fairly granular performance and efficiency comparison between those stacks when under network service load.

7 Memory Performance Analysis

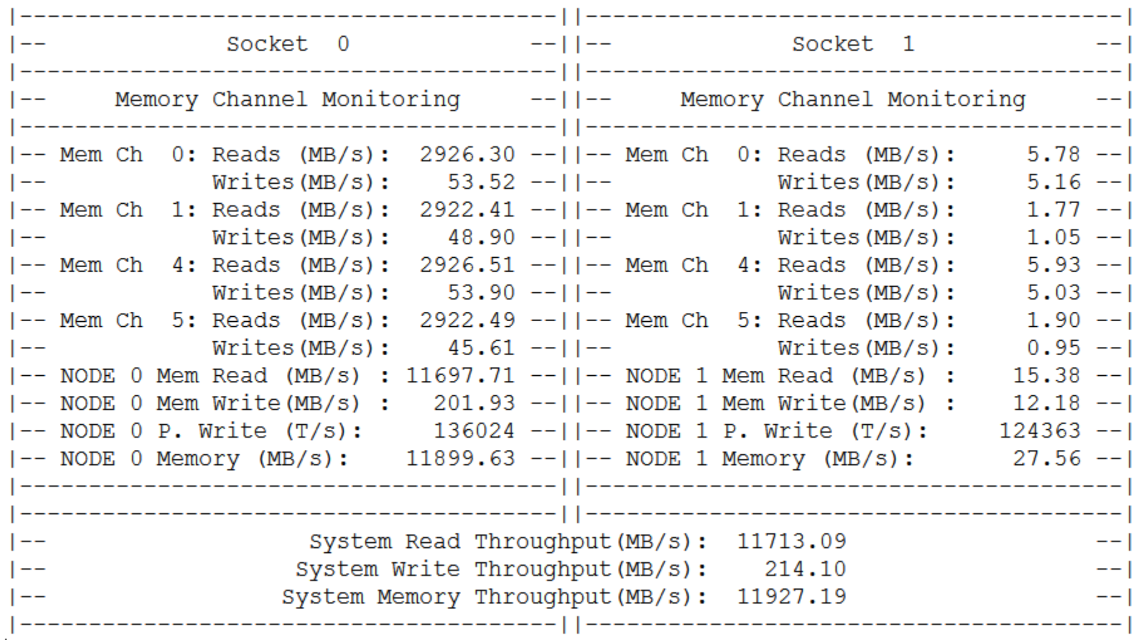
After the core compute complex, system memory is the second most important sub systems in the platform architecture driving an application level performance. For most dedicated networking and general-purpose architectures, the system memory acts as a central agent facilitating interactions between cores and I/O devices. For example, even simple packet routing operation involves at least four interactions with the system memory: i) NIC writes an incoming packet to system memory, ii) core reads the packet header from system memory, iii) core modifies and writes it back to memory, and finally, iv) the NIC reads it from memory for packet transmission. The similar set of operations are needed for processing NICs' descriptors. In essence, routing a packet could result into minimum 5-to-8 operations to system memory. If small packets (64B size) are routed at rate of 100 Million Packet/s, the routing application could easily consume system memory bandwidth in access of 5x64x100 Mpps = 32,000 MBytes/s. Even though features like Direct Data IO technology (explained in the next section) alleviate number of operations to memory, monitoring system memory characteristics is still crucial for understanding possible stalls encountered by both CPUs and PCIe devices, and tuning the applications for optimum memory bandwidth usage.

The memory performance characteristics are measured with two metrics – system memory bandwidth and latency of memory accesses. The following sub sections delve into these aspects.

7.1 Monitoring Memory Bandwidth using pcm-memory.x

The PCM toolchain includes pcm-memory.x utility which provides a holistic view of ongoing-memory bandwidth usage in real-time of Intel® Xeon® E5 processor series architecture. The pcm-memory.x utility uses Performance Monitoring Counters associated with the memory controllers. These counters count all the memory transactions including the ones originated from CPU Cores and PCIe devices. In addition they also count many other memory transactions generated automatically by the architecture, such as memory reads for TLB page walk, RFOs (memory read transactions for Read For Ownership by core). The memory bandwidth reported by pcm-memory.x is thus the sum of all the traffic observed at the system memory channels and hence reflects the true loading on the memory channels.

Figure 21 below shows an example output of pcm-memory.x while the VPP router application is forwarding 1518 Byte packets.



```
|---------------------------------------||---------------------------------------|
|--             Socket  0             --||--             Socket  1             --|
|---------------------------------------||---------------------------------------|
|--     Memory Channel Monitoring     --||--     Memory Channel Monitoring     --|
|---------------------------------------||---------------------------------------|
|-- Mem Ch  0: Reads (MB/s):  2926.30 --||-- Mem Ch  0: Reads (MB/s):     5.78 --|
|--            Writes(MB/s):    53.52 --||--            Writes(MB/s):     5.16 --|
|-- Mem Ch  1: Reads (MB/s):  2922.41 --||-- Mem Ch  1: Reads (MB/s):     1.77 --|
|--            Writes(MB/s):    48.90 --||--            Writes(MB/s):     1.05 --|
|-- Mem Ch  4: Reads (MB/s):  2926.51 --||-- Mem Ch  4: Reads (MB/s):     5.93 --|
|--            Writes(MB/s):    53.90 --||--            Writes(MB/s):     5.03 --|
|-- Mem Ch  5: Reads (MB/s):  2922.49 --||-- Mem Ch  5: Reads (MB/s):     1.90 --|
|--            Writes(MB/s):    45.61 --||--            Writes(MB/s):     0.95 --|
|-- NODE 0 Mem Read (MB/s) : 11697.71 --||-- NODE 1 Mem Read (MB/s) :    15.38 --|
|-- NODE 0 Mem Write(MB/s) :   201.93 --||-- NODE 1 Mem Write(MB/s) :    12.18 --|
|-- NODE 0 P. Write (T/s):     136024 --||-- NODE 1 P. Write (T/s):     124363 --|
|-- NODE 0 Memory (MB/s):    11899.63 --||-- NODE 1 Memory (MB/s):       27.56 --|
|---------------------------------------||---------------------------------------|
|---------------------------------------||---------------------------------------|
|--                 System Read Throughput(MB/s):  11713.09                    --|
|--                System Write Throughput(MB/s):    214.10                    --|
|--               System Memory Throughput(MB/s):  11927.19                    --|
|---------------------------------------||---------------------------------------|
```

Figure 21. Output of pcm-memory.x utility.

The following four metrics are important to observe in the pcm-memory.x output:

- **Aggregate memory bandwidth consumption (shown as Node and System Throughput in above figure):** This metric gives high level view of system wide memory bandwidth consumption and is an important indicator of available memory bandwidth headroom left on the memory channels. Assuming that available memory bandwidth on a single socket Intel® Xeon® E5-2699 v4 processor is ~70 GBytes/s, the above example shows that only ~17% of that is consumed and there is still ample headroom left on memory system. This metric not only implies the current application is not memory bandwidth bound, but it also indicates that there is an opportunity of running other application in parallel to leverage the unused memory bandwidth. As memory bandwidth usage increases, memory latencies seen by both Cores and PCIe devices gradually rise. Such increases in latency often affect application level performance. However, relationship between increase in memory latency and degradation in performance is usually non-linear. Nevertheless, the goal for the performance optimization should be to reduce memory bandwidth consumption as much as possible. The use of DDIO is one such software optimization technique which could minimize memory bandwidth consumption for NF applications.

- **Distribution of traffic across multiple DIMMs (as shown as Mem Ch 0,1,4,5 Reads, Writes**): For many networking applications, the memory controllers are subjected to large amount of concurrent memory accesses from multiple cores in CPU complex and PCIe agents in IO complex. These memory accesses are often short (64 to 512 Bytes) and at random physical addresses. Such address patterns could result into frequent time-consuming page open/close operations on DRAM and hence could potentially degrade DRAM throughput. A well-designed memory controller would spread system addresses uniformly across all memory channels, and across ranks and bank in DRAM so to minimize page open/close operations as well as to reduce the bias against certain address patterns. This metric shows whether memory bandwidth consumption is fairly distributed across all memory channels or not.
- **NUMA (Non-Uniform Memory Access) affinity:** Achieving an optimal application level performance in a dual or multiple processor socket configuration is often a challenge, especially when it involves heavy use of external devices. An ideal NUMA optimized application would use cores, PCIe, and memory resources only on one socket without accessing any of the resources on the other socket(s). A quick way to check NUMA awareness of an applications is to run it on the cores one socket and verify that memory bandwidth consumption (as shown by PCM-memory.x output) on the other socket is close to zero.
- **Memory Read vs. Write Ratio:** The statistics on Memory Read and Write bandwidth often help understand type of operations performed by the cores and PCIe devices. If these numbers are not in accordance with the application code path, tools like Linux “perf top” could be used to debug possible causes of unintended accesses. Besides, Intel Xeon Architecture employs various kind of hardware prefetchers to hide memory latency. In some cases, these prefetchers may end up reading memory locations which are not used by applications, thereby wasting memory bandwidth. In such cases, it would be desirable to turn off various hardware prefetchers, rerun the application, and recheck the performance and memory bandwidth.

7.2 Monitoring Memory Latency

One of the most common reasons behind sub-optimal IPC is the execution stalls caused by data/code misses at all levels of caches resulting in system memory reads. Note that core to memory latency is never a single value. In complex applications with multiple cores and PCIe devices accessing system memory, an individual core would experience a varied degree of memory latencies depending on the amount of concurrent traffic generated by itself and other agents. Measuring core to memory latency under concurrent loads is thus essential to understand the severity of execution stalls.

The loaded or dynamic memory latency can be expressed in the following way:

$$Average\_Memory\_Latency\ (measured\ within\ an\ interval) = \frac{\sum Outstanding\_Memory\_Read\_Requests\_in\_each\_Cycle}{\#Retired\_Read\_Requests}$$

Equation 22. System Memory Latency.

Section 15.5 Measuring Memory Latency outlines the steps to measure memory latency under concurrent loads.

8 PCIe Performance Analysis

NF workloads often involve significant amount of network traffic and hence consume high I/O bandwidth. Understanding interactions between Cores and Network Interface Cards (NICs) can help identify and resolve performance bottlenecks of these applications. This section first looks at how PCIe devices interact with Intel® Xeon® E5 v4 Family processor architecture using Direct Data IO Technology. It then moves to a deep dive into pcm-pcie.x, the tool for PCIe performance analysis.

8.1 Understanding PCIe Bandwidth Consumed by NIC

A packet processing application involves a series of transactions between NIC, memory or LLC and Core. *Figure 22* later in this paper shows sequence of operations involved in typical "bump in wire" use cases. Two main variants of these transactions are described in the following sections.

8.1.1 Transactions Originated by NIC

Network Interface Cards (NICs), such as Intel 82599, Intel XL710, generate two types of PCIe Read and Write transactions to memory:

- Ethernet packets: The PCIe transaction size depends on Ethernet packet size.
- Transmit (Tx) and Receive (Rx) descriptors: CPU cores and NICs communicate information through descriptors. The Tx descriptors contain packet physical address, number of bytes in the packets, and other control information. The Rx descriptors contain information on receive packet buffer addresses, number of bytes received, control and status of the received packets etc. These descriptors are stored in contiguous memory space. In many cases, NICs coalesce multiple descriptors while writing/reading to system memory so as to optimize PCIe efficiency. In the least favorable case, one descriptor is read/written at a time, which results into partial or

sub cache-line read/write PCIe transactions to LLC/memory and also lowers PCIe efficiency.

8.1.2 Transactions Initiated by CPU

CPU cores regularly write to NIC Receive and Transmit tail pointers to notify NICs that new descriptors are available to fetch and process. Core to PCIe device write transactions are in Memory Mapped I/O (MMIO) address space and they are mapped in Uncacheable region. MMIO writes are expensive operations - they can consume up to a few tens of core cycles. Besides, such operations can end up consuming portion of PCIe bandwidth, if they are used too often. Due to these two reasons, software should minimize Rx and Tx tail pointer update operations whenever possible. VPP and DPDK software try to limit tail pointer updates by issuing one update at every 16th or 32nd packet.

CPU to PCIe device Read transactions are even more costly, as they are dependent on core to PCIe device round trip latency. Such transactions can consume several hundred core cycles and hence should be avoided.

8.2 Calculating PCIe Bandwidth from Ethernet Packet Rate

Applying the explanations and understanding of PCIe bandwidth consumed by CPU and NIC interactions, we arrive to the two formulas for NIC network packet write and read to memory.

$$PCIe\_to\_Memory\_Write\_Bandwidth\_Consumption\ \ in\ MBytes/s \\ = (\,A + B + C\,) * Ethernet\_Packet\_Rate\ (in\ MPkts/s)$$

Equation 23. PCIe to memory write bandwidth consumption.

With NIC performing following PCIe to Memory/LLC write transactions for each packet:

- A – Number of Bytes in a received Packets
- B – Number of Bytes in Rx Descriptor write back per packet.
- C – Amortized Number of Bytes Tx Descriptor write back per packet

$$PCIe\_to\_Memory\_Read\_Bandwidth\_Consumption\ in\ MBytes/s \\ = (D + E + F) * Ethernet\_Packet\_Rate$$

Equation 24. PCIe to memory read bandwidth consumption.

With NIC performing following PCIe to Memory/LLC write transactions for each packet:

- D – Number of Bytes in Packet to be transmitted – 4 (Ethernet Checksum is normally calculated and added to a packet by the NIC, hence the last 4 Bytes of packets are not read from memory).
- E – Number of Bytes in Rx Descriptor per packet.
- F – Number of Bytes in Tx Descriptor per packet.

Assuming legacy descriptors of 16B for Network Interface Cards (NICs), such as Intel 82599, Intel XL710, the values of B, C, E, F would 16 Bytes each.

In many cases, software may want to save PCIe bandwidth and can program NIC to write back only one Tx descriptor at every Nth packet confirming the successful transmission of all previous packets. In such cases value of C is amortized over N packet.

Taking one Tx descriptor write back at every 16th packet, the value of C becomes 16B/(amortized over 16 packets), i.e. 1 Byte per packet.

Table 16 uses above formula to calculate the raw PCIe data bandwidth consumption for various packets sizes while forwarding packets at 10 Gbits/s line rate.

Ethernet Packet Size (in Bytes)	Calculated Packet Rate at 10 GE line rate (Million Packets/s)	Calculated PCIe Read Bandwidth Consumption (MBytes/s)	Calculated PCIe Write Bandwidth Consumption (MBytes/s)
64	14.88	1429	1205
128	8.45	1351	1225
200	5.68	1318	1233
256	4.53	1304	1236
384	3.09	1287	1241
512	2.35	1278	1243
768	1.59	1269	1245
1024	1.20	1264	1246
1280	0.96	1262	1247
1518	0.81	1260	1248

Table 16. Raw PCIe data bandwidth for 10GE line rate per Ethernet frame size.

The relationships between packet sizes, packet rates and theoretical PCIe bandwidth consumption, often help debug performance issues in the architecture.

8.3 Intel® Direct Data IO Technology (DDIO)

Intel® DDIO[23] technology in Intel® Xeon® E5 v4 Family processor family essentially enables PCIe devices to write data directly to the Last Level Cache (LLC) rather than system memory. In order to understand interactions between a PCIe device and LLC, the discussion is divided into following two scenarios:

1) A PCIe device (e.g. NIC) writes a cache-line to the system memory, the following DDIO rules apply:

[23] Intel® Direct Data I/O technology – http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html?wapkw=ddio; https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

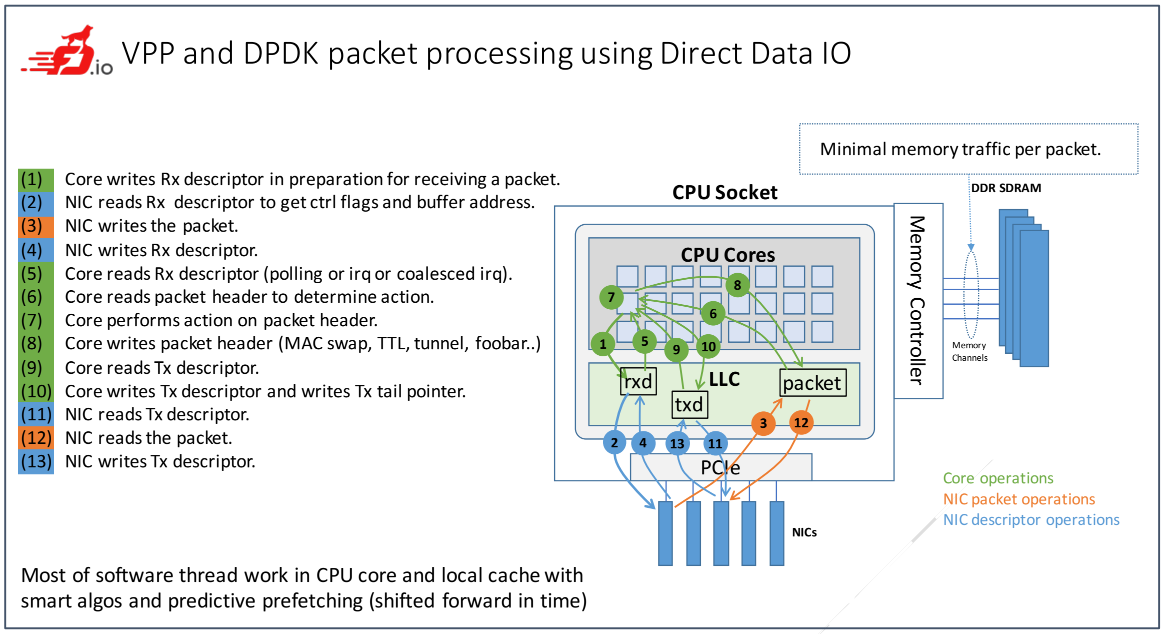
- If the same cache-line is already present in any of the cache ways of Last Level Cache (LLC), the cache-line is over-written by the new data.
- If the cache-line is not present in any of the cache ways of LLC, the cache-line is allocated in the LLC then the new data is written to LLC. At a later time, this cache-line is written back to the memory following the LRU policy. Note that only a subset of LLC ways are used for allocating cache-lines needed for PCIe write transactions.

2) A PCIe device is reading a cache-line from the system memory, the following DDIO rules apply:
   - If the cache-line is already present in any of the cache ways of Last Level Cache (LLC), the cache line is sent to the device without causing system memory read transactions. A cache-line can be present in the LLC because of two reasons - a core might have accessed it, or PCIe device might have written it previously.
   - If the cache-line is not present in any of the cache ways of Last Level Cache (LLC), it is read from system memory and sent to PCIe device.

DDIO technology brings two main advantages:

- **Saving in Memory Bandwidth:** Per explanation in the previous section, High packet rate network traffic could consume inordinate amount of memory bandwidth. A DDIO optimized network application can substantially reduce system memory consumption and help mitigate performance saturation arising from memory bottlenecks. This technology has been effectively leveraged in high packet rate data plane processing solutions like FD.io VPP and many other based on the Data Plane Development Kit (DPDK).
- **Low latency accesses to incoming data from NICs**: As DDIO allows PCIe device to write directly to LLC rather than system memory, there is a substantial saving in latency when a core wants to read the data written by a NIC. For example, a core can access header of a newly written packet with a latency equal or less than LLC latency. In absence of DDIO, a core has to access system memory for performing the same operation and could incur 3x or more latency.

DDIO is one the key features which enables high-speed network data plane performance. *Figure 22* depicts the life of packet for a DDIO optimized DPDK and VPP applications. Such applications use aggressive memory buffer recycling i.e. once packet buffers are processed and released, they are reallocated immediately. In this way, the Network card can rewrite these buffers with new packets while the buffers are still present in LLC. By keeping the application's active memory span to minimum, network packets essentially traverse between cores and NICs through LLC without incurring significant memory traffic.


VPP and DPDK packet processing using Direct Data IO

(1)	Core writes Rx descriptor in preparation for receiving a packet.
(2)	NIC reads Rx descriptor to get ctrl flags and buffer address.
(3)	NIC writes the packet.
(4)	NIC writes Rx descriptor.
(5)	Core reads Rx descriptor (polling or irq or coalesced irq).
(6)	Core reads packet header to determine action.
(7)	Core performs action on packet header.
(8)	Core writes packet header (MAC swap, TTL, tunnel, foobar..)
(9)	Core reads Tx descriptor.
(10)	Core writes Tx descriptor and writes Tx tail pointer.
(11)	NIC reads Tx descriptor.
(12)	NIC reads the packet.
(13)	NIC writes Tx descriptor.

Most of software thread work in CPU core and local cache with smart algos and predictive prefetching (shifted forward in time)

Minimal memory traffic per packet.

Figure 22. VPP and DPDK packet processing using Direct Data I/O.

8.4 PCIe Performance Monitoring

The PCM-PCIe.x utility of PCM toolchain offers a convenient way for measuring PCIe bandwidth in real time. By default, the statistics are shown in with ~1 sec display update rate. The command line for the tool is as follows:

```
# ./pcm-pcie.x -e 1
```

Figure 23 shows the sample output of pcm-pcie.x for the DPDK l3fwd example application forwarding 64B packets at 29.76 Mpps rate. Note that all numbers are in cacheline units. Actual bandwidth can be calculated by multiplying the numbers by 64. The detailed analysis of the performance numbers is discussed at the end of this section.

```
Skt | PCIeRdCur |  RFO   |  CRd  |  DRd  |  ItoM  |  PRd  |  WiL
 0      45 M     8587 K   449 K   102 M    33 M      0     1891 K       (Total)
 0       0          0      24     3276      0        0     1892 K       (Miss)
 0      45 M     8587 K   449 K   102 M    33 M      0        0         (Hit)
 1       0          0      79 K    66 K     0        0        0         (Total)
 1       0          0       0     5532      0        0        0         (Miss)
 1       0          0      79 K    61 K     0        0        0         (Hit)
----------------------------------------------------------------------
 *      45 M     8587 K   528 K   102 M    33 M      0     1891 K       (Aggregate)
```

Figure 23. pcm-pcie.x output.

Table 17 below describes the interpretation of individual events.

Events	Description	Notes:
PCIe/Core Read events (PCIe devices/Cores reading from memory)		

PCIeRdCur	PCIe read from system memory (not allocated line in LLC)	**Hit:** Counts the number of cachelines which were served from LLC to fulfill PCIe to Memory read requests. Such LLC hits occur when the requested data is already deposited in LLC due previous CPU read/write or writes from PCIe device to the same cache lines. **Miss:** Counts the number of cachelines which were read from system memory to fulfill PCIe to Memory read requests. The software optimization effort should aim for much larger PCIeRdCur Hit counts than Miss counts.
DRd	PCIe or Cores read from system memory (cache lines are allocated line in LLC)	This counter also includes CPU reading data in LLC
CRd	PCIe or read from system memory (allocate line in LLC)	This counter includes CPU reading instruction Code in LLC
PCIe write events (PCIe devices writing to memory)		
ItoM	PCIe write (full cache line size and cacheline aligned) to system memory. ItoM stands for Invalid state to Modified.	**Hit:** ItoM Hit counts the number of full cachelines PCIe device is trying to write to memory but they are already present in LLC (in Modified state). **Miss:** Miss represents the new cache lines being allocated while a PCIe device is writing to memory since they were not present (in Invalid state). In this case lines are allocated in the LLC. As DDIO enabled by default, the aggregate count would be high for many I/O centric workloads. The software optimization effort should aim for high ItoM Hit count. This can be done by reusing (recycling) the buffers quickly so that when a NIC writes new packets to memory, the old content of the buffer is still present in the LLC resulting in LLC hits.
RFO	PCIe write to System memory which is sub-cacheline (less	**Hit:** When PCIe devices is writing less than a cache line and cacheline is

	than 64Byte). The partial Write results in Read For Ownership (RFO) event	already present into LLC, this count would increment. **Miss:** When PCIe device is writing less than a cache line and cacheline is already present into LLC, this count would increment. In this case, whole cacheline is read from memory, and merge with new PCIe data and deposited in the LLC. Software optimization effort should minimize partial cache lines writes when possible by aligning buffers at cacheline boundaries. Besides, it should aim for high RFO Hit count compared to Miss rate. This can be done by reusing (recycling) the buffers quickly.
CPU to Memory Mapped IO events (CPU reading/writing to PCIe devices)		
PRd	Aggregated MMIO Read/CPU read transactions to memory mapped device memory on all PCIe devices	This counter counts number of Cpu to Memory Read operations in PCIe memory mapped I/O space. Such operations in the Uncacheable regions are very time consuming. An I/O optimized software should minimize this operation. This count should be very low (a few hundreds)
WiL	Aggregate MMIO Write/CPU write operations to memory mapped device memory on PCIe device	This counter counts number of CPU to Memory write in PCIe memory mapped I/O space. The most common operations are updates to the NIC's Tx and Rx Tail pointers. A well optimized software would minimize such operations. Keeping it below 1M/s per core is a good target to hit.

Table 17. PCM-PCIe.x events description.

8.5 Network Traffic Analysis with PCM-PCIe.x

This section illustrates the use of pcm-pcie.x for PCIe analysis using DPDK l3fwd application as an example application. *Figure 24* below shows the PCIe characteristics for this application forwarding 64B packets at rate of 29.8M Packets/s. Note that the ItoM and PCIedRDcur counts are in cache line (64B) granularity.

```
Skt | PCIeRdCur |  RFO   |  CRd   |  DRd   |  ItoM  |  PRd   |  WiL
 0       45 M      8587 K   449 K   102 M     33 M      0     1891 K        (Total)
 0        0          0       24     3276       0        0     1892 K        (Miss)
 0       45 M      8587 K   449 K   102 M     33 M      0       0           (Hit)
 1        0          0       79 K    66 K      0        0       0           (Total)
 1        0          0        0     5532       0        0       0           (Miss)
 1        0          0       79 K    61 K      0        0       0           (Hit)
----------------------------------------------------------------------
 *       45 M      8587 K   528 K   102 M     33 M      0     1891 K        (Aggregate)
```

Figure 24. PCM-PCIe.x output for DPDK-L3Fwd application.

The following observations can be made from this measurement:

- All transactions are on Socket 0.
- **ItoM (full cachleline writes):** Looking at the above figure, 33M transactions/second show 100% LLC hit, 0 on Miss. These means that all full cacheline PCIe writes are hitting the LLC. This tells that the software recycles packet buffers very effectively. Out 33 MT/s cacheline transactions, 29.8 MT/s are from 64 B Ethernet packet writes. The rest (33-29.8 = 3.2 MT/s) are for Rx descriptor write backs. Since each Rx descriptors are 16B in size, 3.2 MT/s full cacheline Rx descriptor write back transactions equate to 3.2 x (64/16) = 12.8 MT/d Rx descriptors. So, out of 29.8 M packets/s, less than half of the Rx descriptor writebacks seem to coalesce to create full cachelines. The rest of the Rx descriptors generate sub-cacheline writes. Note that for this application, Tx descriptors are written at every 16th packet. Hence, they do not generate full cache line transactions, and thus do not generate IoM event.
- **RFO (Partial cacheline writes):** Looking at the above figure, 8.58 M/s transactions show 100% LLC hit, 0 on Miss. For the give test, these counts are solely from Rx and Tx descriptor write backs. Considering 1 descriptor write at every 16th packet, 29.8/16 = 1.87M T/s are generated from Tx writeback transactions. Rest 6.7 MT/s transactions are generated from Rx descriptor write backs. Note that Rx descriptors can still be coalescing to 32, or 48, and that would generate 1 RFO event.
- **PCIeRdCur:** This event also shows 100% hit rate. This tells that all PCIe to memory read requests are fulfilled from LLC. In this case, 64B ethernet packets would generate 29.8 MT/s transactions. Assuming that the Rx and TX descriptor reads are coalesced to full 64B cacheline sizes, they would generate 29.8M/4 and 29.8M/4 cachelines/s respectively. Sum of ethernet packets and Rx and Tx descriptor adds to 45 MT/s, which is the same as measured by the PCM-PCIe output.
- **WiL:** Tx and Rx tail pointer are updated at every 32nd packet. So number of CPU to PCIe MMIO writes are 29.8Mpps/32+ 29.8 Mpps/32 = 1.8 M/s which matches the pcm-pcie.x output.

  In summary, PCM-PCIe.x offers great insight into processor's PCIe complex behavior and it could help debug and tune the system level performance.

9 Inter-Core and Inter-Socket Communication

Similar to numerous data center and cloud applications, many packet networking processing applications make best use of multicore architecture for scaling the performance. The cores running such applications often share code and data with other cores in the same CPU socket. In some cases cores may access caches and system memory from the remote sockets in a multi-socket platform. Frequent sharing of data between cores and accessing remote socket resources often lead to less than optimal performance. Understanding core to core and core to remote socket interactions is thus important for optimizing performance of a multi-socket system.

9.1 Inter-Core Interactions within the Socket

In many network applications, cores share the same set of data. Applicable cases include:

- Application runs in multi-threaded setup using multiple cores, with each thread running the same code and executing all stages of packet processing functions. However, the threads and associated cores share common memory locations for storing forwarding table, global counters, etc. FD.io VPP is an example of such implementation.
- Application divides packet processing work across multiple threads with processing done in the pipeline fashion. Threads with associated different cores execute certain network function. Here cores need to exchange metadata, parts of the packet buffers, and other info. DPDK ip_pipeline example code demonstrates the concept where a handful of cores handle network interfaces while rest of the cores do packet processing like Flow classification, ACL, metering, routing, and QoS commonly found in applications such as Provider Edge Router.
- In a virtualized setup, a core running qemu vhost-user task handles virtual Ethernet ports, packets, and copying of packets between host space (kernel-mode) and virtio space (user-mode). The core running qemu VM task reads the copied data. The opposite action happens when a core running VM copies the data to virtio and a core running vhost-user reads and eventually transmit the data.

In addtion, the operations like i) resource synchronization using spinlocks, ii) updates of global statistics counters and iii) software based queuing, also involve core to core interactions. Such operations result in migration of cacheline(s) between the L1/L2 caches across the cores. In general, core to core data transfers are expensive as they can consume several tens of cycles and hence should be avoided whenever possible.

Section 15.6 Inter-Processor Communications within the same socket describes the performance events for detecting core to core transfers.

9.2 Inter-Socket Interactions

Most current Operating systems support NUMA. They do a very good job of allocating the buffers on the memory controllers closer to the cores. Such NUMA optimization helps minimize core to system memory latency and hence the IPC.

However, sometimes there are occasions where a core ends up accessing locations that are mapped to the remote system memory. Following are few examples, where a core is subjected to access memory on the other socket:

- A master core allocates packet statistic counters in a buffer which is accessed by all the cores, including the ones in the remote socket.
- Network cards are on one socket and part of the application runs on both sockets.
- An application pre-allocates a packet buffer pool and the OS migrates an application to the cores on the other socket.
- In virtualized environment, openvswitch or vhost-user runs on cores on one socket, and VM runs on cores on the other socket, receiving packets through virtio interface.

In many cases, such cross-socket transfers could be reduced or eliminated through software tuning. However, the most important thing is to detect the occurrences of such transactions.

PCM.x tool offers an easy way to detect such cross-socket transfers over QPI (Quick Path Interconnect). *Figure 25* shows a sample output of PCM.x for a synthetic workload to illustrate inter-socket communication. Benchmark test setup described in this paper focused on single NUMA tests, hence there was no inter-socket communication.

```
Intel(r) QPI traffic estimation in bytes (data and non-data traffic outgoing from
CPU/socket through QPI links):


               QPI0     QPI1    |  QPI0   QPI1
-----------------------------------------------------------------------------------------
 SKT    0     2049 M   1959 M   |   10%    10%
 SKT    1     1531 M   1583 M   |    7%     8%
-----------------------------------------------------------------------------------------
Total QPI outgoing data and non-data traffic: 7123 M
```

Figure 25. PCM.x sample output for QPI transfers.

Note that the QPI counters count both Data cache lines, as well other non-data one (such as QPI control and snoop packets). It is therefore not easy to judge the exact impact of these counts on the performance of an applications. In general though, the above counts should be close to zero for NUMA optimized applications.

10 Performance Tuning Tips

This section describes some basic tuning techniques which could be handy while tuning a workload for high network performance.

10.1 Basic Tuning of the test infrastructure

	Recommendations	Notes
1.1	Tune the bios for performance. Avoid Speed-State, Turbo, Deep C-state for consistency in performance and better No packet Drop rate.	Refer to example BIOS setting on E5-2699v4 given in *Section 13.3 Server BIOS Settings*.
1.2	Tune PCIe network cards, other PCIe device for optimum performance. Turn off ASPM in BIOS, ensure PCIe devices use Max Payload size of 256B or more, Max Read request Size of 512 or more.	Use linux lspci –vvv command to check. Check BIOS, driver to debug possible issue.
1.3	Tune OS for low jitter.	Compile kernel with options which produce low jitter, avoid unnecessary services, use isolcpus when possible.
1.4	Ensure that Memory latency is as expected.	Run “mlc” to check idle latency.
1.5	Ensure that Memory bandwidth is as expected (general conservative formula on Xeon E5 – Expected b/w per socket is : 8 * DDR * DDR4 speed (e.g2400 for DDR2400) * number of channels.	Run “mlc” to check the speed.
1.6	Ensure that Memory utilization is balanced.	Run mlc, run pcm-memory.x and observer that utilization is equal on all sockets. If not, check BIOS setting to ensure that Cluster-on-die is off.

10.2 Simple performance debugging guidelines while running NF app

2.1	Ensure that IPC is 1.75 or more.	If not, compute is constrained by Cache, LLC or memory latency. Debug the code.

2.2	Check cycles per packets are within you expectation. CPP can be calculated using the formula explained in *Section 3.3 Benchmarking NF Applications*.	If CPP is substantially high, compile the code with aggressive optimization for the architecture, check code flow.
2.3	Check for memory bandwidth consumption. Check if Memory b/w packet is within the expected range.	Use PCM-Memory.x to measure memory bandwidth. Also ensure that all channels are uniformly used for the workloads.
2.4	If performance is below expectation, check for cross socket interactions.	Use command lines explained in Appendix C. Find the source of cross socket interactions.
2.5	If performance is below expectation, or want to boost performance conduct TMAM analysis.	Use pmu-tools.

11 Conclusions

Analyzing and optimizing performance of software applications continues to be an area of ongoing research and development, especially for NF applications. This paper described a proposed simple methodology of benchmarking and analyzing the most performance sensitive functional area of NF applications, their data plane.

Following specified benchmarking and analysis methodology and leveraging generally available test and measurement tools described in this paper, it is quite straightforward to evaluate NF applications. Using identified baseline NF data plane performance metrics one can benchmark those applications and compare them in terms of efficiency of using compute resources and their performance on modern COTS servers. Moreover, provided analysis of the baseline factors that drive NF data plane performance scalability (core frequency, simultaneous multi-threading, multi-core), underpinned by measurement data, should aid users in NF capacity planning and their production deployments. Equally, described benchmarking metrics, their meaning and optimal value guidelines should help program developers to identify coding patterns for efficient and performant NF data planes, and hopefully popularize benchmarking driven NF data plane development.

Applicability of described benchmarking and analysis methodology has been illustrated by benchmarking actual NF applications, including feature-rich and deployable NF applications, namely OVS-DPDK and FD.io VPP.

Authors believe that this paper can be used as a stepping stone to establish a reference standard best practice benchmarking methodology and analysis for NF applications' data planes.

Future work needs to focus on further development and tuning the benchmarking methodology to address variety of NF applications, improving automated testing tools and their availability, as well as continuous development of measurement and analysis tools that take advantage of ongoing processor telemetry advancements including the latest generation of Intel® Xeon® scalable processors.

12 References

[1] "Computer Organization and Design, The Hardware/Software Interface" by David A. Patterson and John L. Hennessy, Section 1.6 Performance, ISBN: 978-0-12-407726-3.

[2] RFC 2544, "Benchmarking Methodology for Network Interconnect Devices", March 1999, https://tools.ietf.org/html/rfc2544.

[3] RFC 1242, "Benchmarking Terminology for Network Interconnection Devices", July 1991, https://tools.ietf.org/html/rfc1242.

[4] EEMBC CoreMark - http://www.eembc.org/index.php.

[5] DPDK testpmd - http://dpdk.org/doc/guides/testpmd_app_ug/index.html.

[6] FDio VPP – Fast Data IO packet processing platform, docs: https://wiki.fd.io/view/VPP, code: https://git.fd.io/vpp/.

[7] OVS-DPDK - https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview.

[8] Intel Hyper-Threading, https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[9] "Intel Optimization Manual" – Intel® 64 and IA-32 architectures optimization reference manual.

[10] Technion 2015 presentation on TMAM , Software Optimizations Become Simple with Top-Down Analysis Methodology (TMAM) on Intel® Microarchitecture Code Name Skylake, Ahmad Yasin. Intel Developer Forum, IDF 2015. [Recording].

[11] Linux PMU-tools, https://github.com/andikleen/pmu-tools.

[12] A Top-Down Method for Performance Analysis and Counters Architecture, Ahmad Yasin. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, https://sites.google.com/site/analysismethods/yasin-pubs.

[13] PMU tools: https://github.com/andikleen/pmu-tools.

[14] Intel Developer Zone, Tuning Applications Using a Top-down Microarchitecture Analysis Method, https://software.intel.com/en-us/top-down-microarchitecture-analysis-method-win.

[15] PMU-tools, bdx_server_rations.py - https://github.com/andikleen/pmu-tools/blob/master/bdx_server_ratios.py.

[16] PMU-tools, bdx_server_rations.py - https://github.com/andikleen/pmu-tools/blob/master/bdx_server_ratios.py.

[17] Intel® Direct Data I/O technology – http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html?wapkw=ddio; https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

13 Appendix: Test Environment Specification

13.1 System Under Test – HW Platform Configuration

Mother Board	Super Micro* X10DRX
Processor	Intel Xeon E5-2699 v4/E5-2667v4, Dual Socket configuration
Memory	DDR4-2400, 1 DIMM per channel, 4 Channels for each socket
BIOS Version	Version 2, 12/17/2015
Network Cards	X710-DA4 quad 10 Gbe Port cards, 5 cards total

13.2 System Under Test and Tested Applications – Software Versions

Linux OS Distribution	Ubuntu 16.04.1 LTS x86_64
Kernel Version	4.4.0-21-generic
Fortville firmware version	FW 5.0 API 1.5 NVM 05.00.04 eetrack 800024ca
DPDK Version	DPDK 16.11
VPP Version	v17.04-rc0~143-gf69ecfe
QEMU version	2.6.2
OVS version	2.6.90
Guest OS and kernel	Ubuntu 16.04.1 LTS x86_64
VPP vnet version	v17.04-rc0~143-gf69ecfe

13.3 Server BIOS Settings

Menu (Advanced)	BIOS Submenu Items	BIOS Settings Used for the tests	BIOS Default
CPU Configuration: Advanced Power Management Configuration	Hyper-Threading (ALL)	Disable	Enable
	Power Technology	Disable	Custom
	Energy Performance Tuning	Disable	Enable
	Energy Performance BIAS Setting	Performance	Enable
	Energy Efficient Turbo	Disable	Enable
-> CPU P State Control	EIST (P-States)	Disable	Enable
	Turbo Mode	Disable	Enable
	P-State Coordination	HW_ALL	HW_ALL
-> CPU C State Control	Package C State Limit	[C0/C1 State]	[C6 (Retention)]
	CPU C3 Report	Disable	Enable
	CPU C6 Report	Disable	Enable
	Enhanced Halt State (C1E)	Disable	Enable
Chipset Configuration: North Bridge -> IIO Configuration	EV DFX Features	Enable	Disable
	Intel VT for Directed I/O (VT-d)	Disable	Enable
Chipset -> North Bridge -> QPI Configuration	Link L0 P	Disable	Enable
	Link L1	Disable	Enable
	COD Enable	Disable	Auto

	Early Snoop	Disable	Auto
	Isoc Mode	Disable	Disable
-> North Bridge ->Memory Configuration	Enforce POR	Disable	Auto
	Memory Frequency	2400	Auto
	DRAM RAPL Baseline	Disable	Auto
	A7 Mode	Enable	Enable
-> South Bridge	EHCI Hand-off	Disable	Auto
	USB3.0 Support	Disable	Enable
PCIe/PCI/PnP Configuration	ASPM	Disable	Enable
	Onboard LAN 1 OPROM	Disable	PXE

13.4 Packet Traffic Generator – Configuration

Traffic Generator	Ixia® Traffic Generator

Throughput Test	Ixia® Quick Test: throughput rate search for finding zero-frame loss packet throughput in compliance with RFC 2544
Search algorithm	Binary search.
Starting condition	10% of link rate.
Stopping condition	Search finds the <0.01% loss rate packet throughput and exceeds minimum rate change value.
Number of test trials per each search step	8.
Test trial duration	20 seconds.
Allowed packet loss	<0.01%.
Minimum rate change value	0.1 Mpps.

Test	Ixia packet flow definitions
All L2 Ethernet tests	3,125 distinct flows transmitted per interface.
	Each distinct flow with unique tuple of (Source_MAC_Address, Destination_MAC_Address).
All L3 IPv4 tests	62,500 distinct flows transmitted per interface.

	Each distinct flow with unique tuple of (Source_IPv4_Address, Destination_ IPv4_Address).
Common to all tests	Both packet header source and destination address fields incremented pairwise by 1 in a packet-by-packet sequence.
	Continuous packet flows at fixed rate, with packets equally spaced in time, no bursts.
	Single Ethernet frame size of 64B including Ethernet FCS, smallest standard Ethernet frame possible with IPv4 payload.

14 Appendix: Benchmarking Tools Use Guidelines

14.1 Linux 'perf'

How to install Linux 'perf' on Ubuntu 16.04:

```
apt-get install linux-perf
```

The following conventions are used within this and the following sections:

Set environmental variable $CORENO to the core(s) of interest.

e.g. for monitoring events on core 2, use

```
# export CORENO=2
```

Or, for monitoring events on core 2,3,4 use

```
# export CORENO=2-4
```

Monitoring any discrete event using linux 'perf':

```
# perf stat -e eventname -C$CORENO -I1000
For example,
perf stat -e cpu/event=0x79,umask=0x30,name=idq_ms_uops/ -C1 -I1000
```

Locating the hotspot for an event at source code level:

```
# perf top -e eventname -C$CORENO -I1000
For example,
perf top -e cpu/event=0x79,umask=0x30,name=idq_ms_uops/ -C1
```

Measuring #instructions/cycle (IPC):

```
# perf stat -e instructions,cpu-cycles -C$CORENO sleep 1
```

Capturing Intel Processor Trace (PT):

```
### Trace data will be in gigabytes if captures continuously.
### Thus we will use -S to enable snapshot mode.
### In snapshot mode, PT data is only store when perf instance received -USR2 signal
# perf record -S -e intel_pt// -C$CORENO
# sleep 1
# pkill perf -USR2
# sleep 1
# pkill perf -SIGINT
```

Decoding Intel Processor Trace (PT) data:

```
# perf script --itrace=i1i #i1i: instruction decode, granularity = single instruction

### i1i is the lowest level of information we can obtain from PT.
```

14.2 Performance Analysis using PMU-tools

How to install PMU-Tools

Pre-requisites: PMU-tools is based on Linux Perf utilities.

Download the tools from https://github.com/andikleen/pmu-tools and compile it.

Update the event list for your processor using the script “event_download.py “

Add the pmu-tools folder to the default path. This would allow ocperf.py and other utilities to run from any folder.

Monitoring a discrete event using pmu-tools:

```
# ocperf.py stat -e eventname -C$CORENO -I1000

For example,

# ocperf.py stat -e idq_ms_uops -C1 -I1000
```

Locating the hotspot for an event at source code level:

```
# ocperf.py top -e eventname -C$CORENO -I1000

For example,

# ocperf.py top -e idq_ms_uops -C1
```

Getting instructions per cycle (IPC):

```
# ocperf.py stat -e instructions,cpu-cycles  -C$CORENO  sleep 1
```

14.3 TMAM Analysis using PMU-tools

TMAM statistics can be easily gathered with a simple command line.

TMAM Level 1 events:

```
# toplev.py --core C$CORENO -l1 --no-desc -v --ignore-errata sleep 300
```

TMAM Level 2 events:

```
# toplev.py --core C$CORENO -l2 --no-desc -v --ignore-errata sleep 300
```

TMAM Level 3 events:

```
# toplev.py --core C$CORENO -l3 --no-desc -v --ignore-errata sleep 300
```

TMAM Level 4 events:

```
# toplev.py --core C$CORENO -l4 --no-desc -v --ignore-errata sleep 300
```

14.4 Installing and using PCM tools

Download the source code from https://github.com/opcm/pcm

Follow the installation steps.
As per the information on the above web site, PCM incorporate provides a number of command-line utilities for real-time monitoring:

- **pcm:** basic processor monitoring utility (instructions per cycle, core frequency (including Intel(r) Turbo Boost Technology), memory and Intel(r) Quick Path Interconnect bandwidth, local and remote memory bandwidth, cache misses, core and CPU package sleep C-state residency, core and CPU package thermal headroom, cache utilization, CPU and memory energy consumption)
- **pcm-memory:** monitor memory bandwidth (per-channel and per-DRAM DIMM rank)
- **pcm-pcie:** monitors PCIe bandwidth and other related statistics
- **pcm-numa:** monitors local and remote memory accesses
- **pcm-power:** monitors sleep and energy states of processor, Intel(r) Quick Path Interconnect, DRAM memory, reasons of CPU frequency throttling and other energy-related metrics
- **pcm-tsx:** monitors performance metrics for Intel(r) Transactional Synchronization Extensions
- **pcm-core and pmu-query:** query and monitor arbitrary processor core events

15 Appendix: Deep-dive TMAM Analysis using Linux *perf* and PMU-Tools

This section describes the commands for counting individual events in the TMAM hierarchy. Once performance bottleneck hotspots are found from the top level TMAM analysis as described in the Section 6 , these commands could be used for monitoring the specific events of interest while optimizing the code. For example, if TMAM finds high counts on the Bad_Speculation.Branch_Mispredicts, the event "br_misp_retired_all_branches" could be monitored during the code optimization and benchmarking cycles. These commands would also help locate hotspots in the code by running "perf top" or "ocperf.py top" on the selected processor core events.

For the sake of completeness, the commands are given for both Linux "perf" utility and PMU-tools.

15.1 Events related to TMAM %Retiring

%Retiring

```
# perf stat -e \
cpu/event=0x3c,umask=0x0,any=1,name=cpu_clk_unhalted_thread_any/,cpu/event=0xc2,umask=0x2
,name=uops_retired_retire_slots/,cpu/event=0xe,umask=0x1,name=uops_issued_any/,cpu/event=
0x79,umask=0x30,name=idq_ms_uops/ -C$CORENO  -I1000
```

or

```
# ocperf.py stat -e CPU_CLK_UNHALTED_THREAD_ANY,\
UOPS_RETIRED.RETIRE_SLOTS,UOPS_ISSUED.ANY,IDQ.MS_UOPS  -C$CORENO -I1000
```

15.2 Events related to TMAM *%Bad_Speculation*

%Bad_Speculation

```
# perf stat -e
cpu/event=0xe,umask=0x1,name=uops_issued_any/,cpu/event=0xc2,umask=0x2,name=uops_retired_
retire_slots/,cpu/event=0xd,umask=0x3,any=1,cmask=1,name=int_misc_recovery_cycles_any/ -
C$CORENO -I1000
```

or

```
# ocperf.py stat -e
UOPS_ISSUED.ANY,UOPS_RETIRED.RETIRE_SLOTS,INT_MISC.RECOVERY_CYCLES_ANY  -C$CORENO -I1000
```

%Bad_Speculation.Branch_Mispredicts

```
# perf stat -e cpu/event=0xc5,umask=0x0,name=br_misp_retired_all_branches/ -C$CORENO -
I1000
```

or

```
# ocperf.py stat -e BR_MISP_RETIRED.ALL_BRANCHES -C$CORENO -I1000
```

%Bad_Speculation.Machine_Clears

```
#  perf stat -e cpu/event=0xc3,umask=0x1,name=machine_clears_cycles/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e machine_clears.cycles -C$CORENO -I1000
```

15.3 Events related to TMAM %Frontend_Bound

%Frontend_Bound

```
# perf stat -e \
cpu/event=0x9c,umask=0x1,name=idq_uops_not_delivered_core/,cpu/event=0x3c,umask=0x0,any=1
,name=cpu_clk_unhalted_thread_any/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e IDQ_UOPS_NOT_DELIVERED.CORE,CPU_CLK_UNHALTED.THREAD_ANY -C$CORENO -
I1000
```

%Frontend_Bound.Frontend_Latency

```
# perf stat -e\
cpu/event=0x9c,umask=0x1,cmask=4,name=idq_uops_not_delivered_cycles_0_uops_deliv_core/ -
C$CORENO -I1000
```

or

```
# ocperf.py stat -e IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE -C$CORENO -I1000
```

%Frontend_Bound.Frontend_Bandwidth

```
# perf stat -e
cpu/event=0x9c,umask=0x1,name=idq_uops_not_delivered_core/,cpu/event=0x9c,umask=0x1,cmask
=4,name=idq_uops_not_delivered_cycles_0_uops_deliv_core/,cpu/event=0x3c,umask=0x0,any=1,n
ame=cpu_clk_unhalted_thread_any/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e \
IDQ_UOPS_NOT_DELIVERED.CORE,IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE,CPU_CLK_UNHAL
TED.THREAD_ANY -C$CORENO -I1000
```

%Frontend_Bound.Frontend_Latency.ICache_Misses

Intel® Xeon® E5 v4 Family processors have L1 instruction cache of 32K bytes. If execution path for an application spans beyond this range, instruction cache miss event is incremented. The penalty due to instruction cache is calculated using the following equation.

$$\%\,ICache\ Misses\ =\ \frac{ICACHE.IFDATA\_STALL}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

The event ICACHE.IFDATA_STALL measures the cycles for which a code fetch is stalled.

```
# perf stat -e
cpu/event=0x80,umask=0x4,name=icache_ifdata_stall/,cpu/event=0x3c,umask=0x0,name=cpu_clk_
unhalted_thread/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e ICACHE.IFDATA_STALL,CPU_CLK_UNHALTED.THREAD  -C$CORENO  -I1000
```

%Frontend_Bound.Frontend_Latency.ITLB_Misses(%)

For many network applications the execution path for processing a packet could be relatively small. Considering large ITLB (Instructions TLB), the chances of having ITLB miss are relatively small. Even if there is an ITLB miss, it is likely that the ITLB entry is present in the second level TLB. The overall impact of ITLB Miss could be calculated as follows:

$$\%\,ITLB\ Misses\ = \frac{7 * ITLB\_MISSES.STLB\_HIT + ITLB\_MISSES.WALK\_DURATION}{CPU\_CLK\_UNHALTED.THREAD\_ANY}$$

```
# perf stat -e
cpu/event=0x85,umask=0x60,name=itlb_misses_stlb_hit/,cpu/event=0x85,umask=0x10,cmask=1,na
me=itlb_misses_walk_duration/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e ITLB_MISSES.STLB_HIT,ITLB_MISSES.WALK_DURATION:c1 -C$CORENO -I1000
```

If %ITLB Misses is more than 0.01%, it is recommended to rearrange the code such that frequently accessed portions of the code fall into small number of the pages, thereby limiting ITLB misses. Alternatively, one can use large and super page size (2M or 1G) for the code segments to minimize ITLB misses.

15.4 Events related to TMAM *%Backend_Bound*

The events related to TMAM *%Backend_Bound, %Backend_Bound.Memory, %Backend_Bound.Core* could be measured through pmu-tools "top-level" analysis scripts.

```
# toplev.py --core C$CORENO -l2 --no-desc -v --ignore-errata sleep 300
```

There are many events pertaining to %Backend Level 1 and Level 2. Only a handful of them are described below.

%Backend_Bound.Memory.L1_Bound

```
# perf stat -e
cpu/event=0xa3,umask=0x6,cmask=6,name=cycle_activity_stalls_mem_any/,cpu/event=0xa3,umask
=0xc,cmask=12,name=cycle_activity_stalls_l1d_miss/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e cycle_activity.stalls_mem_any,cycle_activity.stalls_l1d_miss -
C$CORENO -I1000
```

%Backend_Bound.Memory.L2_Bound

```
# perf stat -e \
cpu/event=0xa3,umask=0xc,cmask=12,name=cycle_activity_stalls_l1d_miss/,cpu/event=0xa3,uma
sk=0x5,cmask=5,name=cycle_activity_stalls_l2_miss/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e CYCLE_ACTIVITY.STALLS_L1D_MISS,CYCLE_ACTIVITY.STALLS_L2_MISS -
C$CORENO -I1000
```

%Backend_Bound.Memory.L3_Bound

```
# perf stat -e
cpu/event=0xd1,umask=0x4,name=mem_load_uops_retired_l3_hit/,cpu/event=0xd1,umask=0x20,nam
e=mem_load_uops_retired_l3_miss/,cpu/event=0xa3,umask=0x5,cmask=5,name=cycle_activity_sta
lls_l2_miss/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e
MEM_LOAD_UOPS_RETIRED.L3_HIT,MEM_LOAD_UOPS_RETIRED.L3_MISS,CYCLE_ACTIVITY.STALLS_L2_MISS
-C$CORENO -I1000
```

%Backend_Bound.Memory.Store_Bound

```
# perf stat -e cpu/event=0xa2,umask=0x8,name=resource_stalls_sb/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e RESOURCE_STALLS.SB -C$CORENO -I1000
```

15.5 Measuring Memory Latency

$$Average\_Memory\_Latency\ (measured\ within\ an\ interval) = \frac{\sum Outstanding\_Memory\_Read\_Requests\_in\_each\_Cycle}{\#Retired\_Read\_Requests}$$

The core to system memory average memory latency can be measured using the following two events:

- The event OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD counts the number of offcore outstanding cacheable Core Data Read transactions in the super queue every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation).
- The event OFFCORE_REQUESTS.ALL_DATA_RD counts the demand and prefetch data reads.

```
# perf stat -e
cpu/event=0x60,umask=0x1,name=offcore_requests_outstanding_demand_data_rd/,cpu/event=0
xb0,umask=0x1,name=offcore_requests_demand_data_rd/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e
OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD,OFFCORE_REQUESTS.DEMAND_DATA_RD -C$CORENO
-I1000
```

15.6 Inter-Processor Communications within the same socket

Interactions between two cores within the same socket can be measured using the following three main performance events:

1. The event MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT, counts retired load uops which data sources were L3 hit and a cross-core snoop hit in the on-package core cache.
2. The event MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM, counts retired load uops which data sources were HitM responses from a core on same socket

(shared L3). Frequent sharing of modified line could be major source of performance bottlenecks. Such operations should be minimized when possible.

```
# perf stat -e \
cpu/event=0xd2,umask=0x2,name=mem_load_uops_l3_hit_retired_xsnp_hit/,cpu/event=0xd2,um
ask=0x4,name=mem_load_uops_l3_hit_retired_xsnp_hitm/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e \
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT,MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM  -
C2$CORNE -I1000
```

3. The event MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS counts retired load uops which data sources were L3 Hit and a cross-core snoop missed in the on-package core cache.

```
# perf stat -e cpu/event=0xd2,umask=0x1,name=mem_load_uops_l3_hit_retired_xsnp_miss/ -
C$CORENO -I1000
```

or

```
# ocperf.py stat -e MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS -C$CORENO -I1000
```

15.7 Inter-Socket Communications

The following events can be used to measure the number of cacheline accesses made to the other socket in a dual socket platform.

1. The event offcore_response.demand_data_rd.llc_miss.remote_hitm counts the memory loads which were fulfilled by the remote socket LLC and the requested cachelines were in the modified state.

```
# perf stat -e
cpu/event=0xb7,umask=0x1,offcore_rsp=0x103fc00001,name=offcore_response_demand_data_rd_ll
c_miss_remote_hitm/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e offcore_response.demand_data_rd.llc_miss.remote_hitm   -C$CORENO -
I1000
```

2. The event offcore_response.demand_data_rd.llc_miss.remote_hit_forward counts the memory loads which were fulfilled by the remote socket LLC where the requested lines were in Shared or Exclusive states.

```
# perf stat -e
cpu/event=0xb7,umask=0x1,offcore_rsp=0x87fc00001,name=offcore_response_demand_data_rd_llc
_miss_remote_hit_forward/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e offcore_response.demand_data_rd.llc_miss.remote_hit_forward   -
C$CORENO -I1000
```

3. The event offcore_response.demand_data_rd.llc_miss.remote_dram counts the memory loads which were fulfilled by the System memory attached to the remote socket.

```
# perf stat -e
cpu/event=0xb7,umask=0x1,offcore_rsp=0x63bc00001,name=offcore_response_demand_data_rd_llc
_miss_remote_dram/ -C$CORENO -I1000
```

or

```
#ocperf.py stat -e offcore_response.demand_data_rd.llc_miss.remote_dram   -C$CORENO -
I1000
```

15.8 Other Useful Events

DTLB misses can be monitored as follows:

```
# perf stat -e \
cpu/event=0x8,umask=0x60,name=dtlb_load_misses_stlb_hit/,cpu/event=0x49,umask=0x60,name=d
tlb_store_misses_stlb_hit/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e DTLB_LOAD_MISSES.STLB_HIT,DTLB_STORE_MISSES.STLB_HIT -C1 -I1000
```

The following commands can be used for monitoring the number of cycles when the Divider Unit of a core is active:

```
# perf stat -e cpu/event=0x14,umask=0x1,name=arith_fpu_div_active/ -C$CORENO -I1000
```

or

```
# ocperf.py stat -e ARITH.FPU_DIV_ACTIVE -C$CORENO -I1000
```

16 Index: Figures

17 Index: Tables

18 Index: Equations

END OF DOCUMENT