TECHNOLOGY GUIDE

Intel Corporation

Intel® AVX-512 – Writing Packet Processing Software with Intel® AVX-512 Instruction Set

Authors

Ray Kinsella

Chris MacNamara

Georgii Tkachuk

Reviewers

Ciara Power

Vladimir Medvedkin

1 Introduction

Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set is a powerful addition to the packet processing toolkit. As Intel's latest generation of SIMD instruction set, Intel® AVX-512 (also known as AVX-512) is a game changer, doubling register width, doubling the number of available registers, and generally offering a more flexible instruction set compared to its predecessors. Intel® AVX-512 has been available since the 1st Generation Intel® Xeon® Scalable processors and is now optimized in the latest 3rd Generation Intel® Xeon® Scalable processor and the Intel® Xeon® D processor, with compelling performance benefits. The performance, configurations, and feature set may vary for the Intel® Xeon® D processor.

This paper is the second in a series of white papers that focuses on how to write packet processing software using the Intel® AVX-512 instruction set. This paper describes how SIMD optimizations are enabled in software development frameworks such as DPDK and FD.io. It includes examples of using Intel® AVX-512 in these frameworks as well as the performance benefits obtained, demonstrating performance gains in excess of 300% in microbenchmarks[1].

The previous paper in this series is an introduction for software engineers looking to write packet processing software with Intel® AVX-512 instructions. The paper provides a brief overview of the Intel® AVX-512 instruction set along with some pointers on where to find more information. It also describes microarchitecture optimizations for Intel® AVX-512 in the latest 3rd Generation Intel® Xeon® Scalable processors. An executive summary of these papers is also available.

This white paper is intended for organizations developing or deploying packet processing software on the latest 3rd Generation Intel® Xeon® Scalable processors.

It is a part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

[1] Benchmarked with DPDK 20.02 L3FWD-ACL and DPDK-TEST_ACL. See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	February 2021	Initial release.
002	April 2021	Benchmark data updated. Revised the document for public release to Intel® Network Builders.
003	February 2022	Added information regarding Intel® Xeon® D processor in the Introduction section.

1.1 Terminology

ABBREVIATION	DESCRIPTION
ACL	Access Control List
AES	Advanced Encryption Standard
CPE	Customer Premise Equipment
DPDK	Data Plane Development Kit (dpdk.org)
FD.io	Fata Data I/O (an umbrella project for Open Source network projects)
FIB	Forwarding Information Base
HPC	High Performance Computing
IDS/IPS	Intrusion Detection Systems/Intrusion Prevention Systems
IP	Internet Protocol
ISA	Instruction Set Architecture
LPM	Longest Prefix Match
MAC	Media Access Control address
PMD	Poll Mode Driver
SIMD	Single Instruction Multiple Data (term used to describe vector instruction sets such as Intel® SSE, Intel® AVX, and so on)
SSE	Streaming SIMD Extensions (Intel® SSE is a predecessor to Intel® AVX instruction set)
VPP	FD.io Vector Packet Processing, an Open Source networking stack (part of FD.io)
VxLAN	Virtual Extensible LAN, a network overlay protocol.
Vhost	Virtual Host

1.2 Reference Documentation

REFERENCE	SOURCE
Intel® 64 and IA-32 Architectures Optimization Reference Manual	https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
Intel® 64 and IA-32 Architectures Software Developer Manual	https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
Intel® AVX-512 - Packet Processing with Intel® AVX-512 Instruction Set Solution Brief	https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-packet-processing-with-intel-avx-512-instruction-set-solution-brief
Intel® AVX-512 – Instruction Set for Packet Processing Technology Guide	https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-instruction-set-for-packet-processing-technology-guide

2 Overview

SIMD code has been used in Packet Processing for some years now. DPDK, for example, has been using SIMD instructions since its 2014 1.7.0 release. That release debuted with Intel® SSE 4.2 support in the Poll Mode Driver (PMD) for Intel's 10-Gigabit Ethernet Products. Since then Intel has added support for Intel® AVX2 and, now most recently, the AVX-512 instruction sets to DPDK.

DPDK's use of SIMD instructions has grown over time, expanding to include PMDs supporting Intel's 40- and 100-Gigabit Ethernet Product portfolio as well as PMDs for hardware from other vendors, the Virtio PMD, and associated Vhost backend. DPDK's SIMD use has also grown to include a number of libraries such as the DPDK Longest Prefix Match (LPM), Forwarding Information Base (FIB), and Access Control List (ACL) libraries. A recent example is DPDK adding Cryptodev support for Intel's new Vector AES (VAES) instruction set extension supported on 3rd Generation Intel® Xeon® Scalable processors, which offers significant performance improvements in AES cryptography. Partly due to the increased use of vectorization as well as the generation-on-generation Intel microarchitectural enhancements, DPDK's performance on Intel platforms has improved dramatically over time[2]. DPDK's recent enablement of Intel® AVX-512 is described in detail in subsection 4.1.

Similarly, FD.io VPP debuted in the open source community with Intel® SSE 4.2 support in 2015, subsequently adding support for Intel® AVX2 and Intel® AVX-512 instruction sets in 2018. When FD.io VPP debuted as an Open Source Project, vectorization usage was initially confined to a small number of VPP graph nodes such as classifiers (ACLs). Since that time, similar to DPDK,

[2] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

vectorization usage in FD.io VPP has grown rapidly to the point where it is now used widely in places such as the VLIB library, the glue between the graph nodes, the Ethernet and IP layers, and in the FD.io VPP native PMDs such as the AVF plugin for Intel's 40- and 100-Gigabit product portfolio, Virtio, and MemIF plugins. FD.io's recent enablement of Intel® AVX-512 is described in detail in subsection 4.2

This document is for software engineers wanting to write Packet Processing software with Intel® AVX-512 (henceforth referred to as AVX-512) and is structured as follows:

- Vectorization Approaches describes two approaches to writing vectorized code: using vector intrinsics and using compiler vector literals approaches, as used by DPDK and FD.io VPP respectively.
- Microarchitecture Optimizations describes how these optimizations are enabled at runtime in DPDK and FD.io VPP, detailing the configuration switches and APIs that enable these optimizations.
- Optimization Examples describes examples of optimizations developed with AVX-512 instructions drawn from DPDK and FD.io VPP, where AVX-512 is used to accelerate the DPDK's Forwarding Information Base (FIB) Library and FD.io VPP's VxLAN decapsulation[3].

3 Vectorization Approaches

It is worth noting that DPDK and FD.io take slightly different approaches to vectorization of code. DPDK typically favors a slightly more hand-crafted approach while FD.io VPP trusts the compiler to do a little more optimization.

3.1 DPDK approach

The DPDK approach typically favors the use of vector intrinsics, which are built-in functions that are specially handled by the compiler. These vector intrinsics usually map 1:1 with vector assembly instructions. Consider, for example, 8-bit vector addition with the AVX-512 intrinsic _mm512_add_epi8.

The following code sample loads two 512-bit vectors of 8-bit integers from the unaligned memory locations `dest` and `src`, performs an addition of the two vectors with _mm512_add_epi8, and returns the results.

```
#include <immintrin.h>
#include <stdint.h>

uint8_t *addx8(uint8_t *dest, uint8_t *src) {

    __m512i a = _mm512_loadu_epi64 (src);
    __m512i b = _mm512_loadu_epi64 (dest);

    _mm512_storeu_epi64(dest, _mm512_add_epi8(a, b));

    return dest;}
```

The code snippet above generates the following assembler code with the Clang 10 compiler, where the almost 1:1 mapping from vector intrinsic to assembly instructions can be observed.

Note: The Intel® AVX-512 intrinsic, _mm512_add_epi8, is converted to a **vpaddb** assembly form, and the intrinsic, _mm512_loadu_epi8, is converted to a memory read from `zmmword ptr(s)`.

```
addx8:                  # @addx8
   mov                  rax, rdi
   vmovdqu64            zmm0, zmmword ptr [rdi]
   vpaddb               zmm0, zmm0, zmmword ptr [rsi]
   vmovdqu64            zmmword ptr [rdi], zmm0
   vzeroupper
   ret
```

[3] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

3.2 FD.io VPP approach

The FD.io VPP approach makes use of compiler language extensions called *Vector Literals*. These are compiler built-in data-type attributes and operators and are considered by some developers to be easier to code with compared to using vector intrinsics as described in the previous section. Vector literals are supported by both GCC and Clang compilers.

To use vector literals, the developer declares a data type as having the attribute `vector_size`, which declares the data type as a vector of values and describes the data type's size in bytes. A 64-byte `vector_size` indicates that the data type must be an alias of a 512-bit register, and that any operations performed on a variable of that data type usually generates AVX-512 instructions.

This assertion is dependent on correctly setting those compiler flags that permit the use of AVX-512 instructions. For more details, see subsection 4.2. Similarly, a 32-byte `vector_size` aliases a 256-bit register and usually generates AVX2 instructions.

Table 1 describes some examples of VPP data types declared as vector literals.

Table 1. Examples of Declaring Vector Literals

VPP DATA TYPE	C DATA TYPE	VECTOR SIZE (BYTES)	INSTRUCTION SET	DESCRIPTION
u8x64	uint8_t	64	AVX-512	Packed Array of 64 x 8-bit Integers
u8x32	uint8_t	32	AVX2	Packed Array of 32 x 8-bit Integers
u8x16	uint8_t	16	SSE	Packed Array of 16 x 8-bit Integers
u64x8	uint64_t	64	AVX-512	Packed Array of 8 x 64-bit Integers
u64x4	uint64_t	32	AVX2	Packed Array of 4 x 64-bit Integers
u64x2	uint64_t	16	SSE	Packed Array of 2 x 64-bit Integers

Vector literals give developers access to a standard set of operators (arithmetic, bitwise, and so on) provided by the compiler. The developer trusts the compiler to generate the equivalent vector instructions targeted at appropriate instruction set generation, and to optimize where possible. FD.io VPP also provides a library of macros to perform common operations (splat, scatter, gather, and so on) to compliment vector literals[4]. These can be found in the FD.io VPP `vppinfra/vector*.h` header files.

```
#include <immintrin.h>
#include <stdint.h>
typedef uint8_t u8x64 __attribute__((aligned(8)))
                      __attribute__((__vector_size__(64)));
uint8_t *addx8(uint8_t *dest, uint8_t *src) {
    u8x64 a = *(u8x64*) src;
    u8x64 b = *(u8x64*) dest;
    *(u8x64 *) dest = a + b;
    return dest;
}
```

The code snippet above generates the following assembler in Clang 10, which is equivalent to the DPDK vector intrinsics snippet described in the previous section. Note that the **+** operator is converted to a **vpaddb** instruction, and that the cast and pointer dereferences `*(u8x64*)` is converted to a memory read from `zmmword ptr(s)`.

> Intel maintains the *Intel® Intrinsics Guide* that provides a complete database of Intel® Architecture SIMD intrinsics, all the way from Intel® MMX to Intel® AVX-512 instruction sets, along with descriptions of functionality, throughput, and latency, on different generations of Intel® Microprocessors, along with mappings to SIMD instructions.

[4] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex . Results may vary.

```
addx8:                  # @addx8
    mov                 rax, rdi
    vmovdqu64           zmm0, zmmword ptr [rdi]
    vpaddb              zmm0, zmm0, zmmword ptr [rsi]
    vmovdqu64           zmmword ptr [rdi], zmm0
    vzeroupper
    ret
```

Table 2 lists a subset of the standard C operators natively supported by vector literals. The full list is available in the Clang documentation.

Table 2. Examples of Operations Supported by Vector Literals

OPERATION	OPERATORS
Extraction	[]
Increment/decrement	++, – –
Arithmetic	+,*,/ and %
bitwise operators	&,|,^ and ~
shift operators	>>,<<
logic operators	!, &&, ||
Assignment	=

It is worth noting that the GCC and Clang compilers also support two built-in functions that are complementary to vector literals:

- `__builtin_shufflevector` function that provides an alias to vector permutation/shuffle/swizzle operations, and
- `__builtin_convertvector` function that provides an alias to vector conversion operations.

These built-in functions work like the vector literals described above, automatically generating vector instructions appropriate to the size of the data type passed as arguments. That is, when passed a 64-byte vector literal these built-ins usually generate AVX-512 instructions with the compiler optimizing where possible.

4 Microarchitecture Optimizations

Software built on DPDK or FD.io VPP typically must run on a diverse number of platforms, including different generations of microarchitecture (1st, 2nd, and 3rd Generation Intel® Xeon® Scalable processors) and different processors-types (Intel® Xeon® and Intel Atom®). The difficulty is that creating platform-specific software releases adds complexity and costs for software vendors. The ideal solution, therefore, is to have one binary release of a given product that runs without difficulty on diverse platforms and also optimizes where possible[5].

DPDK and FD.io VPP achieve auto-optimization for their execution environment with a single binary—using a technique called function multi-versioning to create multi-architecture binaries—which is described in the following sections.

4.1 DPDK's function multi-versioning

The DPDK 20.11 release added two new APIs to the environment abstraction layer (EAL) to get and set the maximum SIMD bitwidth. The maximum SIMD bitwidth implies the largest register size in bits and the associated SIMD instruction set that can be used by DPDK libraries, drivers, and applications. The APIs are listed in Table 3.

[5] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Table 3. DPDK 20.11 RTE VECT APIs

API	DESCRIPTION
`uint16_t rte_vect_get_max_simd_bitwidth(void);`	This API returns a value from the enumeration `rte_vect__max_simd`, indicating the maximum permitted register size to be used by DPDK applications.
`int rte_vect_set_max_simd_bitwidth(uint16_t bitwidth);`	This API sets a value from the enumeration `rte_vect_max_simd`, indicating the maximum permitted register size to be used by DPDK applications.

The `rte_max_simd` enumeration details are listed in Table 4.

Table 4. DPDK `rte_vect_max_simd` Enumeration

ENUMERATION	DESCRIPTION
`RTE_VECT_SIMD_DISABLED`	Indicates that DPDK must follow scalar only code paths.
`RTE_VECT_SIMD_128`	Indicates that DPDK may follow scalar and Intel® SSE4.2 code paths, prioritizing Intel® SSE4.2.
`RTE_VECT_SIMD_256`	Indicates that DPDK may follow scalar, Intel® SSE4.2 and Intel® AVX2 code paths, prioritizing Intel® AVX2.
`RTE_VECT_SIMD_512`	Indicates that DPDK may follow scalar, Intel® SSE4.2, Intel® AVX2 and Intel® AVX-512 code paths, prioritizing Intel® AVX-512.

In DPDK it is common for one algorithm to have multiple implementations, each supporting a different generation of SIMD technologies. In this common optimization technique, there may be several microarchitecture-optimized versions of the same algorithm in a given library or PMD simultaneously. DPDK[6] is therefore able to optimize for the microprocessor on which it is executing by choosing the fastest possible implementation of the algorithm at runtime, provided other prerequisite conditions are met.

DPDK uses the value set by `rte_vect_set_max_simd_bitwidth` and the capabilities of the microprocessor to determine which microarchitecture-optimized function version to use. A good example is the DPDK Virtio PMD, which uses AVX-512 optimized code paths when the following conditions are met:

- The microprocessor supports AVX-512, detected by calling `rte_cpu_get_flag_enabled(RTE_CPUFLAG_AVX512F)`
- The application has enabled AVX-512, detected by calling `rte_vect_get_max_simd_bitwidth(void)`
- In addition, the Virtio PMD-specific requirements are also met:
  - Virtual Machines' Virtio interface supports the Virtio 1 standard
  - Virtual Machines' Virtio interface has enabled Virtio in order feature flag

The DPDK EAL also supports a `force-max-simd-bitwidth` command-line parameter to override calls to `rte_vect_set_max_simd_bitwidth`. This parameter is useful for testing purposes. Provided other prerequisite conditions are met:

- Specifying `--force-max-simd-bitwidth=64` disables vectorized code paths
- Specifying `--force-max-simd-bitwidth=512` enables AVX-512 code-paths

Note: When the application does not specify a preference through the `rte_vect_set_max_simd_bitwidth` API, a microarchitecture-specific default is used. On Intel® Xeon® microarchitectures, the default is to use Intel® AVX2 instructions, while on Intel Atom® platforms the default is to use Intel® SSE4.2 instructions.

4.1.1 DPDK Libraries

Some DPDK libraries enable the calling application to specify a preferred algorithm implementation through their API. Examples of such libraries are the DPDK Forwarding Information Base (FIB) and Access Control List (ACL) libraries. For example, the DPDK ACL library has two search calls to find a matching ACL rule for a given input buffer:

[6] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Table 5. DPDK 20.11 ACL API

API	DESCRIPTION
`int rte_acl_classify (const struct rte_acl_ctx *ctx, const uint8_t **data, uint32_t *results, uint32_t num, uint32_t categories)`	Performs search for a matching ACL rule for each input data buffer. Automatically determines the optimal search algorithm to use based on microprocessor capabilities and any preference specified through the EAL APIs.
`int rte_acl_classify_alg (const struct rte_acl_ctx *ctx, const uint8_t **data, uint32_t *results, uint32_t num, uint32_t categories, enum rte_acl_classify_alg alg)`	Performs search for a matching ACL rule for each input data buffer using the algorithm specified, provided it is supported by the microprocessor and permitted by the DPDK EAL.

Table 6 lists the lookup algorithms supported by the DPDK ACL library, used with the `rte_acl_classify_alg` function.

Table 6. DPDK `rte_acl_classify_alg` Enumeration

ENUMERATION	DESCRIPTION
`RTE_ACL_CLASSIFY_SCALAR`	Scalar implementation, does not require any specific HW support.
`RTE_ACL_CLASSIFY_SSE`	Intel® SSE4.1 implementation, can process up to eight flows in parallel. Requires SSE 4.1 support. Requires maximum SIMD bitwidth to be at least 128 bits.
`RTE_ACL_CLASSIFY_AVX2`	Intel® AVX2 implementation; can process up to 16 flows in parallel. Requires AVX2 support. Requires maximum SIMD bitwidth to be at least 256 bits.
`RTE_ACL_CLASSIFY_AVX512X16`	Intel® AVX-512 implementation, can process up to 16 flows in parallel. Uses 256-bit width SIMD registers. Requires AVX512 support. Requires maximum SIMD bitwidth to be at least 256 bits.
`RTE_ACL_CLASSIFY_AVX512X32`	Intel® AVX-512 implementation, can process up to 32 flows in parallel. Uses 512-bit width SIMD registers. Requires AVX512 support. Requires maximum SIMD bitwidth to be at least 512 bits.

Similar to the DPDK Virtio PMD, the DPDK ACL library uses AVX-512 optimized lookup code paths when the following conditions are met:[7]

- The microprocessor supports AVX-512, detected by calling `rte_cpu_get_flag_enabled(RTE_CPUFLAG_AVX512F)`.
- The application has enabled AVX-512, detected by calling `rte_vect_get_max_simd_bitwidth(void)`.
- Once the prior conditions are met, the `rte_acl_classify` function automatically selects an AVX-512 optimized lookup algorithm. The `rte_acl_classify_alg` function uses the algorithm specified in the `alg` parameter, allowing AVX-512 optimized lookup algorithms to be selected, once the prior conditions are met.

4.2 FD.io VPP's multi-arch variants

FD.io VPP is implemented as a directed graph of nodes, with each graph node containing some state and specifying a function to process packets. Each graph node is typically encapsulated inside a separate C source file and resulting object file at build time. In the FD.io VPP build system, many graph nodes are designated as multi-arch variants. This designation causes the FD.io VPP build system to build a separate variant of the graph node for each generation of microprocessor microarchitecture supported by FD.io VPP.

[7] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

```
$ make run-release

vpp# show node ethernet-input
node ethernet-input, type internal, state active, index 388
  node function variants:
    Name              Priority  Active
    icl                    100    yes
    skx                     -1
    hsw                     50
    default                  0
```

Figure 1. Multi-arch Variants in FD.io VPP

As shown in Figure 1, on Intel® Architecture, FD.io VPP currently supports optimizing for 3rd Generation Intel® Xeon® Scalable processors – codenamed Ice Lake (ICL), 1st Generation Intel® Xeon® Scalable processors - codenamed Skylake (SKX), Intel ® Xeon® E processors - codenamed Haswell (HSW), and Intel Atom® microarchitectures (default). Therefore, a number of microarchitecture-specific object files are built per graph node, with one per supported microarchitecture, and then linked into a single FD.io VPP binary.

At runtime, FD.io VPP detects the microarchitecture on which it is executing and automatically configures each graph node to use the multi-arch variant optimized for that microarchitecture. As shown in Figure 1, the multi-arch variant prioritization causes FD.io VPP to have a preference for the ICL variant when executing on 3rd Generation Intel Xeon Scalable processors The 3rd Generation Intel Xeon Scalable processor-optimized variant has a preference for AVX-512 instructions, while the 1st Generation Intel Xeon Scalable processor (SKX) and Intel Xeon E processor (HSW) variants have a preference for AVX2 instructions, and the default preference is for the omnipresent SSE4.2 instructions.

The IPv4 forwarding packet processing pipeline comprises the graph nodes shown in Figure 2.

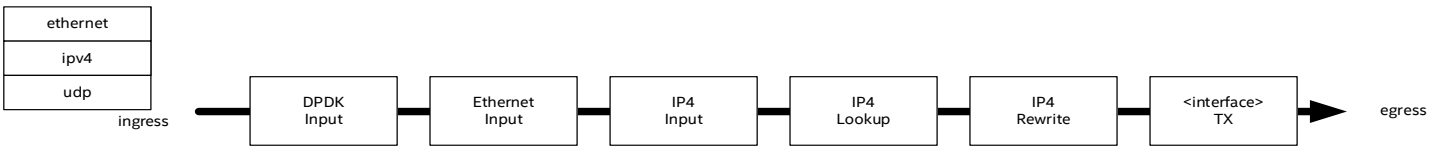

Figure 2. FD.io VPP IPv4 Packet Processing Pipeline

Figure 3 shows that on the 3rd Generation Intel® Xeon® Scalable processors, FD.io VPP automatically executes microarchitecture-specific variants of the Ethernet Input and IP4 Lookup graph nodes containing Intel® AVX-512 instructions.

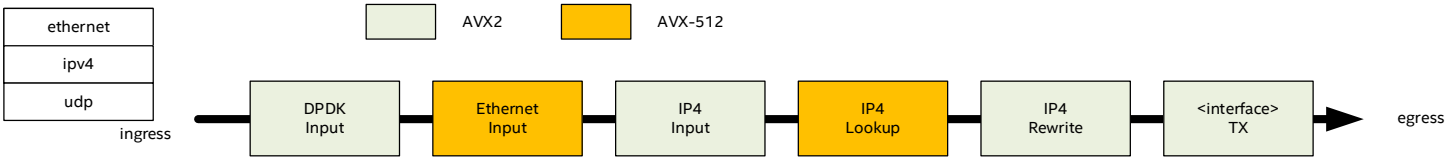

Figure 3. FD.io VPP IPv4 Packet Processing Pipeline on 3rd Generation Intel® Xeon® Scalable Processors

In a majority of cases, to optimize the multi-arch variants for the specific microarchitecture generations, FD.io VPP trusts the compiler to optimize the code where it can by specifying `-mtune=icelake-client`, `-mprefer-vector-width=avx512`, and so on. These settings instruct the compiler to optimize using the Intel® AVX-512 instruction set on 3rd Generation Intel Scalable processors wherever it can. In addition, hand-crafted optimizations may also be introduced wrapped in `CLIB_HAVE_VEC512` defines. In this way, the `vlib` and `vppinfra` infrastructure libraries are hand-optimized with Intel® AVX-512 instructions, and the functionality of these libraries are used throughout the FD.io VPP codebase[8].

In the code snippet below, the FD.io VPP `vppinfra` function `clib_memset_u64` is an equivalent to libc's `memset` function. It provides a good example of these kind of hand-crafted optimizations. It uses the `CLIB_HAVE_VEC*` defines to create AVX-512, AVX2 and scalar variants of the function, with `always_inline` causing the function to then be inlined into 3rd Generation Intel

[8] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Xeon Scalable processors, 1st Generation Intel Scalable processors, and other architecture-specific graph node variants. Note the heavy use of the vector literals described in subsection 3.2 in this function.

```
static_always_inline void
clib_memset_u64 (void *p, u64 val, uword count)
{
  u64 *ptr = p;
#if defined(CLIB_HAVE_VEC512)
  u64x8 v512 = u64x8_splat (val);
  while (count >= 8)
    {
      u64x8_store_unaligned (v512, ptr);
      ptr += 8;
      count -= 8;
    }
  if (count == 0)
    return;
#endif
#if defined(CLIB_HAVE_VEC256)
  u64x4 v256 = u64x4_splat (val);
  while (count >= 4)
    {
      u64x4_store_unaligned (v256, ptr);
      ptr += 4;
      count -= 4;
    }
  if (count == 0)
    return;
#else
  while (count >= 4)
    {
      ptr[0] = ptr[1] = ptr[2] = ptr[3] = val;
      ptr += 4;
      count -= 4;
    }
#endif
  while (count--)
    ptr++[0] = val;
}
```

5 Optimization Examples

This section describes Intel® AVX-512 optimizations to the DPDK FIB library and FP.io VPP tunnel decapsulation.

5.1 DPDK FIB library

The following example is taken from the DPDK FIB Library, where Intel® AVX-512 instruction set is used to parallelize table lookups to the Forwarding Information Base (FIB) lookup table. The DPDK FIB library is commonly used in software such as Virtual Routers to help direct Internet Protocol (IP) packets to the next router or gateway (next hop) on the path to their intended destination.

The DPDK FIB Library lookup is designed to receive an IP address as input and return a next hop index. A developer may then use the next hop index to look up a separate table, outside of the FIB library. The table will usually contain additional application-specific information, such as the next hop's MAC address, an egress interface, and so on. The FIB library supports different lengths of next hop indexes depending on the intended use case, ranging from CPE (customer-premises equipment) devices that are required to accommodate a handful of next hops, to a carrier grade router that may need to accommodate millions of next hops. The next hop index length is specified at table creation. It is immutable at runtime and may be defined as 8-bit (IPv4 only),16-bit, 32-bit or 64-bit.

In order to understand the AVX-512 optimization, it is useful to first describe the scalar version of the code and its limitations. Also, in order to produce a readable example, this section initially focuses on the first 24 Most Significant Bits (MSBs) of the IPv4 lookup algorithm. However, it will become clear that the same technique is scaled to also parallelize IPv6 lookups.

The example that follows is an IPv4 address lookup to retrieve a 32-bit next hop index. As shown in Figure 4, the FIB IPv4 lookup traverses two tables—tbl24 and tbl8—where the most significant 24-bits of the IPv4 address are used to index into the **tbl24** table and the least significant 8-bits, along with information from **tbl24** is used to index into the **tbl8** table. When the Least Significant Bit (LSB) in a given **tbl24** entry is set, it indicates that **tlb8** table must be referenced for the next hop index (1). When the LSB is clear, the next hop index is the 31 MSBs of the **tbl24** entry.

Table **tbl24** is sized in memory, dependent on the size of the next hop index (2). An 8-bit next hop index implies a 16-megabyte table, a 16-bit next hop index implies a 32-megabyte table, 32-bit next hop index implies a 64-megabyte table, and so on. The size of the **tbl8** table is similarly a function of the size of the next hop index.

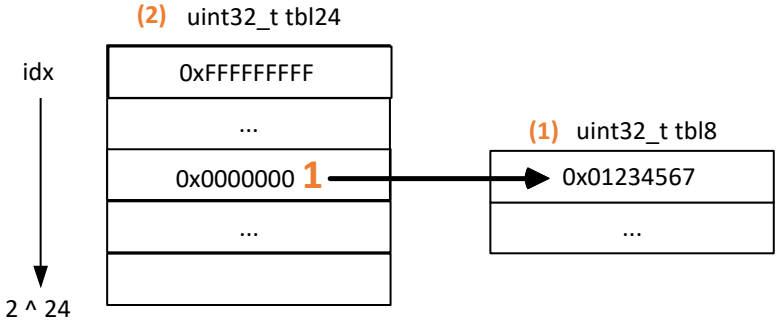

Figure 4. DPDK FIB Library IPv4 Tables

The FIB library supports a number of scalar and vectorized next hop lookup algorithms.[9] The scalar/macro IPv4 lookup implements the IPv4 lookup logic as a macro (or template), where **tlb24** and its entries are recast at build time as a vector of 8- to 64-bit integers depending on the size of the next hop index. The resulting functions, `dir24_8_lookup_bulk_[1b,2b,4b,8b]`, are then inlined into the calling logic.

Figure 5 shows the steps to retrieve a 32-bit next hop index from **tbl24** with the scalar/macro IPv4 lookup. The lookup logic is broken into three stages: retrieving an entry from the **tbl24** table `(get_tbl_entry)`, shifting out the LSB `(shift_to_bits)`, and then optionally looking up the **tbl8** table.

- **get_tbl_entry:** an index is generated from the most significant 24 bits of the IP address by shifting right by eight bits. The **tbl24** table entry is then read at the memory offset generated from the index and the next hop size.
- **shift_to_bits:** When the LSB in the **tlb24** table entry is clear, the entry is shifted right to remove the LSB, what remains is returned as the next hop index, shown as (1) in Figure 5.
- **lookup_tlb8:** When the LSB in the **tlb24** table entry is set, the entry is shifted right to remove the LSB, the upper bits are then combined with the least significant 8-bits of the IP address to index into table **tlb8**, shown as (2) in Figure 5.

[9] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

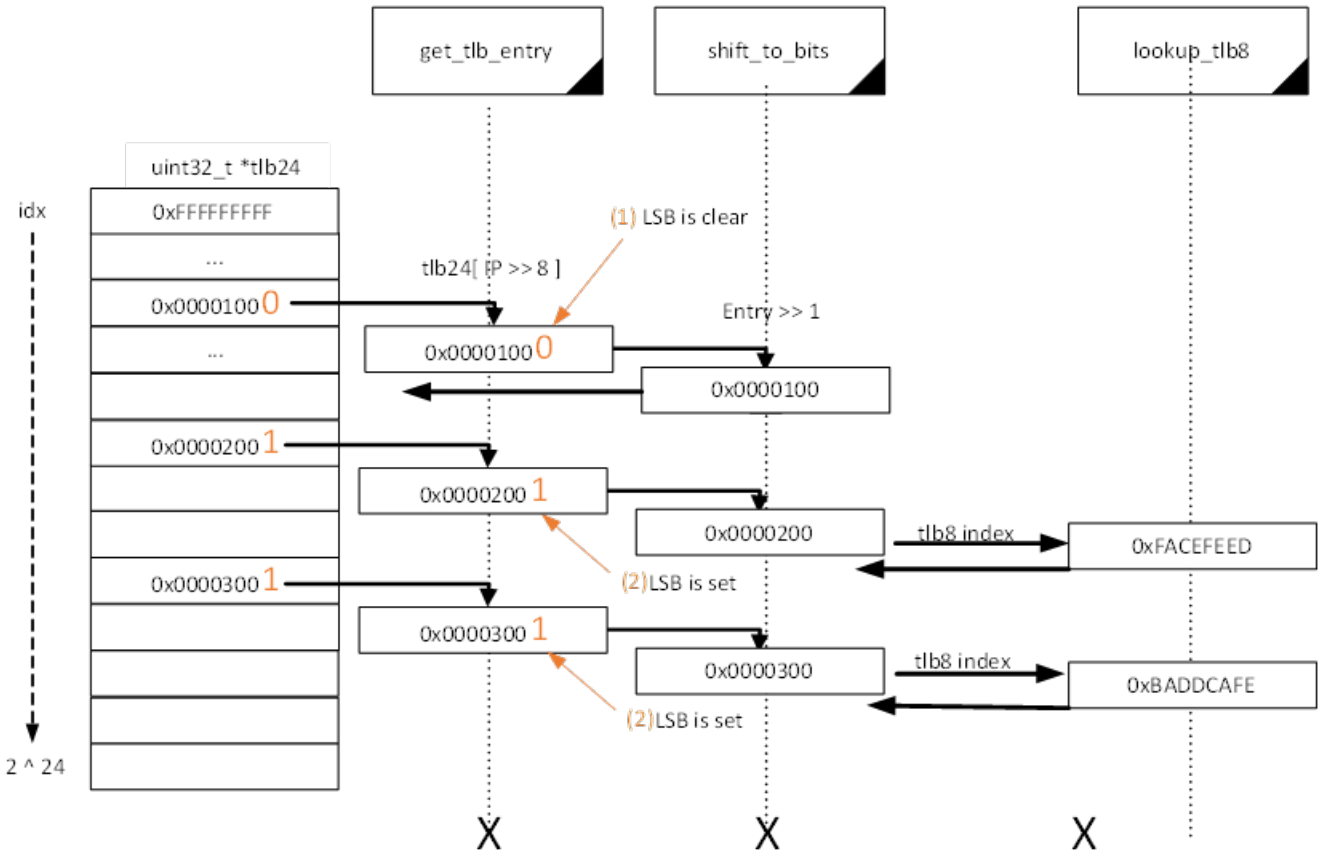

Figure 5. DPDK FIB Library IPv4 Next Hop Lookup

The vectorized IPv4 FIB lookup implements the IPv4 lookup logic with the AVX-512 instruction set, retrieving the next hop for 16 IPv4 addresses in parallel for the next hop sizes of 8-bit, 16-bit, and 32-bit, and retrieving eight (8) next hops in parallel for 64-bit next hop indexes.

Figure 6 shows the steps to retrieve 32-bit next hop indexes from the FIB **tbl24** table with AVX-512 instruction set. This breaks down into two stages: the **tbl24** table entry `(get_tlb_entry)` is retrieved using the upper 24-bits of the IP address and the LSB of the entry is then tested to determine if the **tlb8** table must be referenced.

- **get_tlb_entry:** For 16 IPv4s addresses in parallel, 16 table indexes are generated from the upper 24 bits of the IPv4 addresses, by shifting right by 8 bits. The intrinsic `_mm512_i32gather_epi32` then multiplies the 16 table indexes by a scaling factor of 1, 2 or 4, depending on the next hop index size in bytes—in this case, 4 bytes—to generate 16 offsets in memory. The intrinsic `_mm512_i32gather_epi32` then reads the 32 bits at the generated offsets to retrieve 16 **tbl24** entries in parallel.
- **test_lsb:** The LSB of each 16 **tbl24** table entry is then tested in parallel with the intrinsic `_mm512_test_epi32_mask`, with the table **tbl8** then referenced for those of the 16 **tbl24** table entries whose LSB is set. The `_mm512_test_epi32_mask` intrinsic returns a mask—used in subsequent masked operations—to eliminate branching and only referencing the **tbl8** table for those entries indicated.

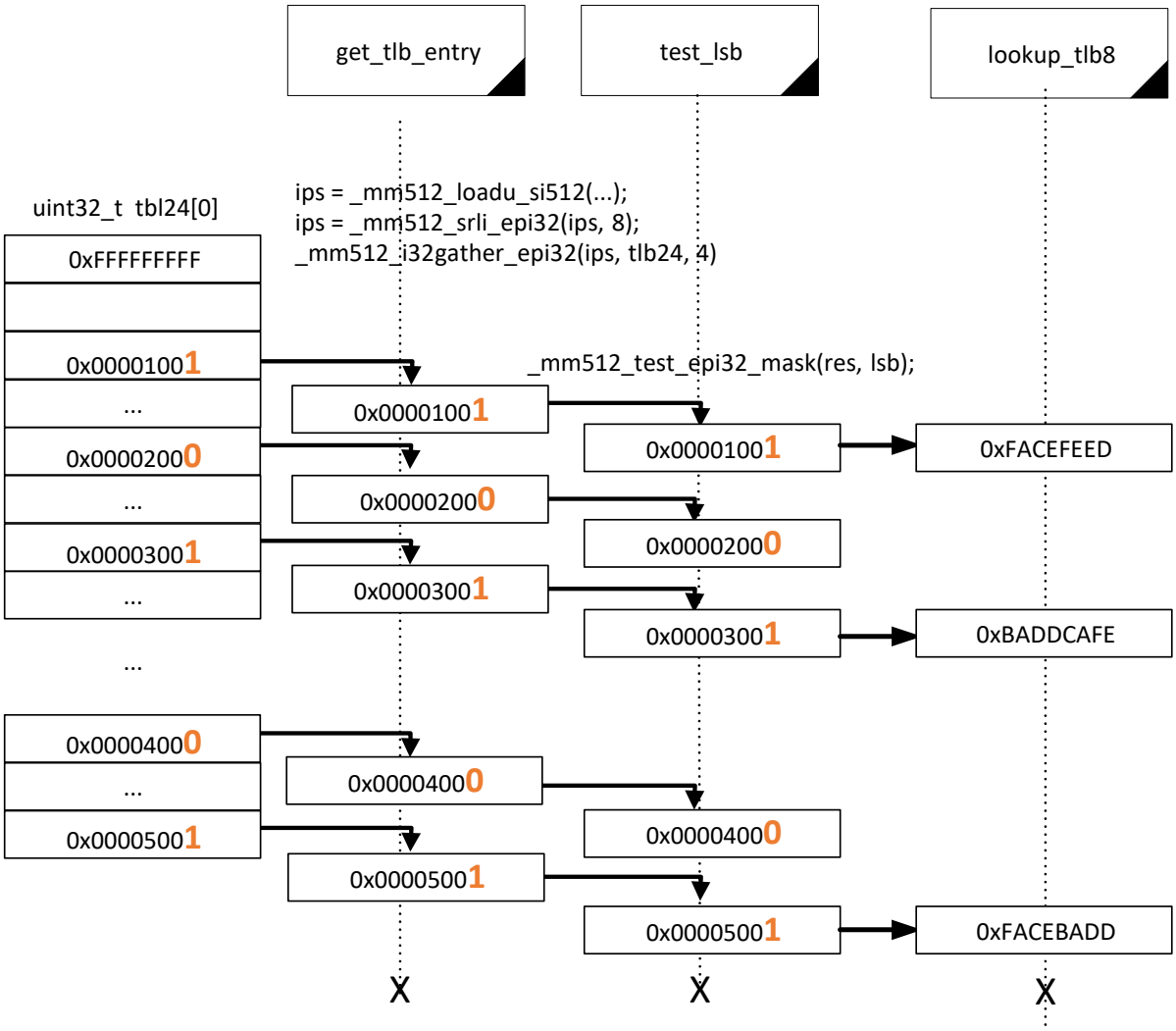

Figure 6. DPDK FIB Library IPv4 Next Hop Lookup with Optimization

In this way, **tbl24** lookups are parallelized to perform 16 lookups simultaneously, compared to the single lookup described previously. An interested reader can find the full code for the IPv4 table lookup here.

The DPDK FIB library implements IPv6 next hop lookups in a similar manner to IPv4. The first 24-bits of the IPv6 address are looked up in table **tbl24**, and subsequent 8-bit chunks of the 128-bit IPv6 address are recursively looked up in table **tbl8**. The AVX-512 optimization of IPv6 next hop lookup also works in a similar manner to IPv4, where 16 next hop indexes are retrieved in parallel for 16-bit and 32-bit next hop sizes, and eight next hop indexes are retrieved in parallel for 64-bit next hop sizes.

As shown in Figure 7, to prepare for a 32-bit next hop lookup, 16 IPv6 addresses (1) are transposed into four AVX-512 registers such that the first 32 bits of each address are transposed to the 1st register, the second 32 bits are transposed to the 2nd register, and so on. The first 24 bits of each packed 32-bit value in the 1st register are then used to retrieve 16 **tbl24** table entries in parallel. Table **tbl24** entries and the next 8 bits from the IPv6 addresses are then used to look up 16 **tbl8** table entries in parallel. Table **tbl8** entries and the next 8-bit chunk of the IPv6 addresses are then recursively read in this manner until the LSB of the **tbl8** entries are cleared, indicating that the next hop index may be read and returned. An interested reader can find the full code for the IPv6 table lookup here[10].

[10] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

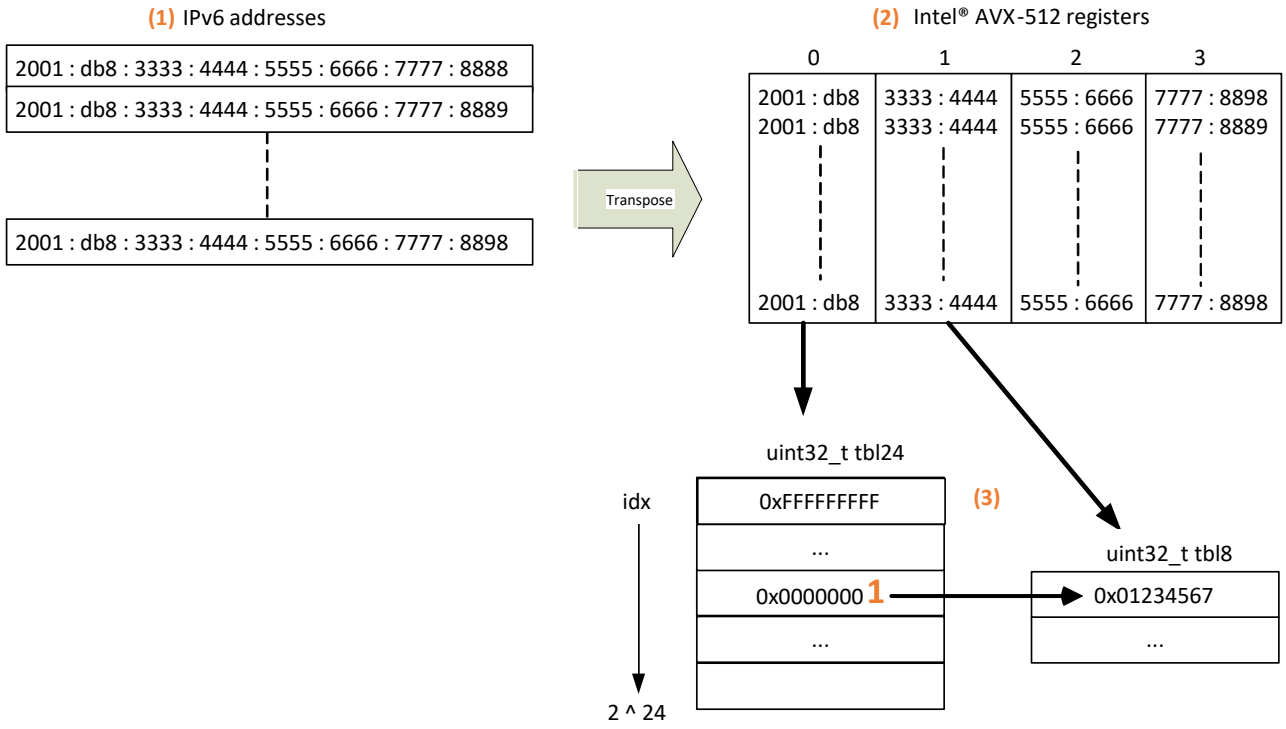

Figure 7. DPDK FIB Library IPv6 Next Hop Lookup with Optimization

The DPDK FIB library has a number of scalar non-parallelized implementations. Of them the highest performing is the scalar/macro lookup described in Figure 5. A microbenchmark was generated using the DPDK FIB Test Application (app/dpdk-test-fib). Table 7 lists the test configuration parameters.

Table 7. DPDK FIB Test Configuration

PARAMETERS	DESCRIPTION
Test by	Intel
Test date	28 March 2021
Platform	Intel Corporation Reference Platform*
# Nodes	1
# Sockets	2
CPU	Intel® Xeon® Gold 6338N CPU
Cores/socket, Threads/socket	32, 64
CPU Frequency	2.2GHz
L3 Cache	48MiB
microcode	0x8d055260
BIOS version	WLYDCRB1.SYS.0020.P86.2103050636
System DDR Mem Config: slots / capacity / run-speed	16 / 16384GB / 3200MT/s
Turbo, P-states	OFF
Hyper Threads	Enabled
OS	Ubuntu 20.04.1 LTS (Focal Fossa)
Kernel	5.4.0-40-generic
DPDK Version	DPDK 21.02
Compiler	GCC 9.3.0-10 ubuntu2
	*Intel® Reference Platform (RP) for 3rd Generation Intel® Xeon® Scalable Processor

Figure 8 illustrates the performance improvement with the AVX-512 parallelized IPv6 lookup compared to the scalar/macro lookup, where the AVX-512 parallelized lookup improves performance in excess of 35%[11].

[11] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

- **Clock Cycles** (shown on the primary y-axis) is the number of cycles consumed by the IPv6 lookup operation. This is measured by reading `rdtsc` before and after one million IPv6 lookups and then calculating a cycles-per-operation value.
- **Next Hop Size** (shown on the x-axis) is the next hop size in bytes.
- **% Improvement** (shown on the secondary y-axis) is the percentage improvement obtained using AVX-512 version over the baseline scalar/macro version of the code.

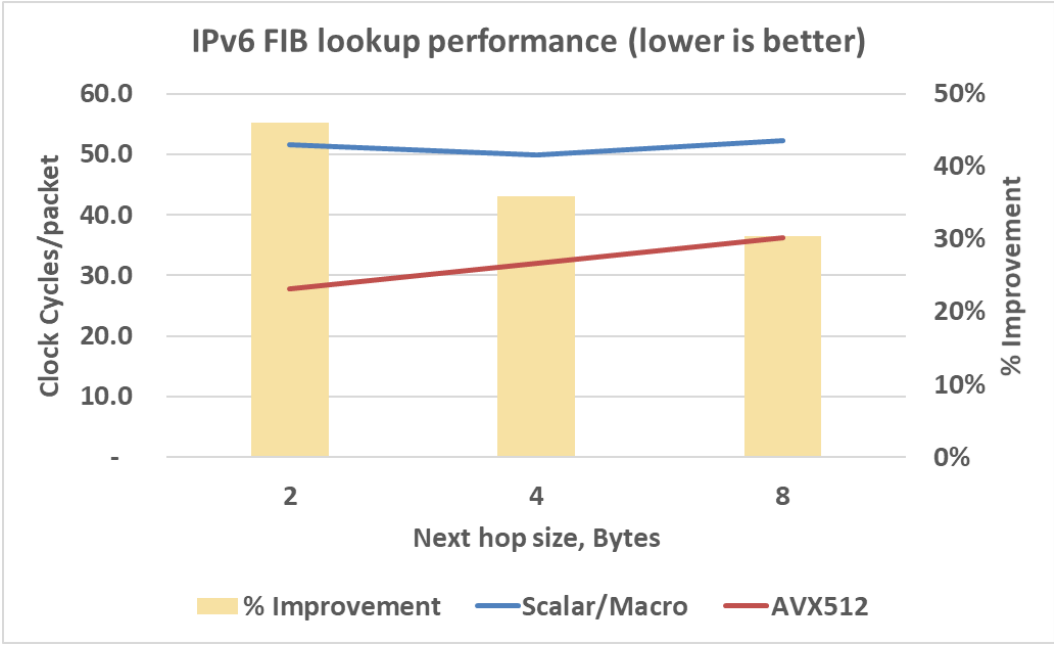

Figure 8. DPDK Optimization of FIB Library (Clock Cycles)

5.2 FD.io VPP tunnel decapsulation

The following example is taken from FD.io VPP, where the AVX-512 instruction set is used in network overlay termination to avoid expensive Tunnel Endpoint table lookups. In order to understand the optimization better, it is useful to first describe the scalar version of the code and its limitations.

In FD.io VPP, network overlay protocols such as GENEVE, GTPU, and VxLAN use an optimization technique where a *hash* of the last packet as well as its Tunnel Endpoint information is cached. Then, if the hash of the next/current packet matches that of the last packet, an expensive Tunnel Endpoint table lookup can be avoided (shown in Figure 9 as `hash_get`). However, as shown in Figure 9, when the hash values do not match, the lookup is triggered, and the cached hash value is updated.

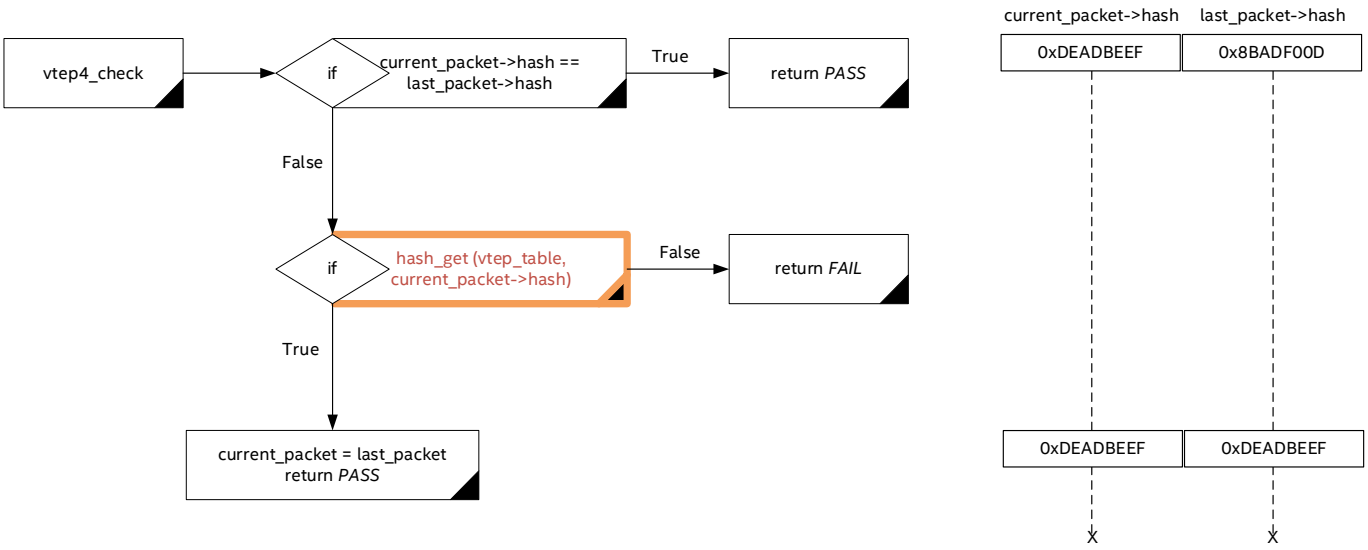

Figure 9. FD.io VPP Tunnel Decapsulation

This technique can work well with a few tunnel endpoints, and where network traffic exhibits a lot of batching. However, introducing multiple tunnel endpoints and interleaving packets from these endpoints cause the comparison of the cached value to miss on every packet and quickly diminish the value of this optimization.

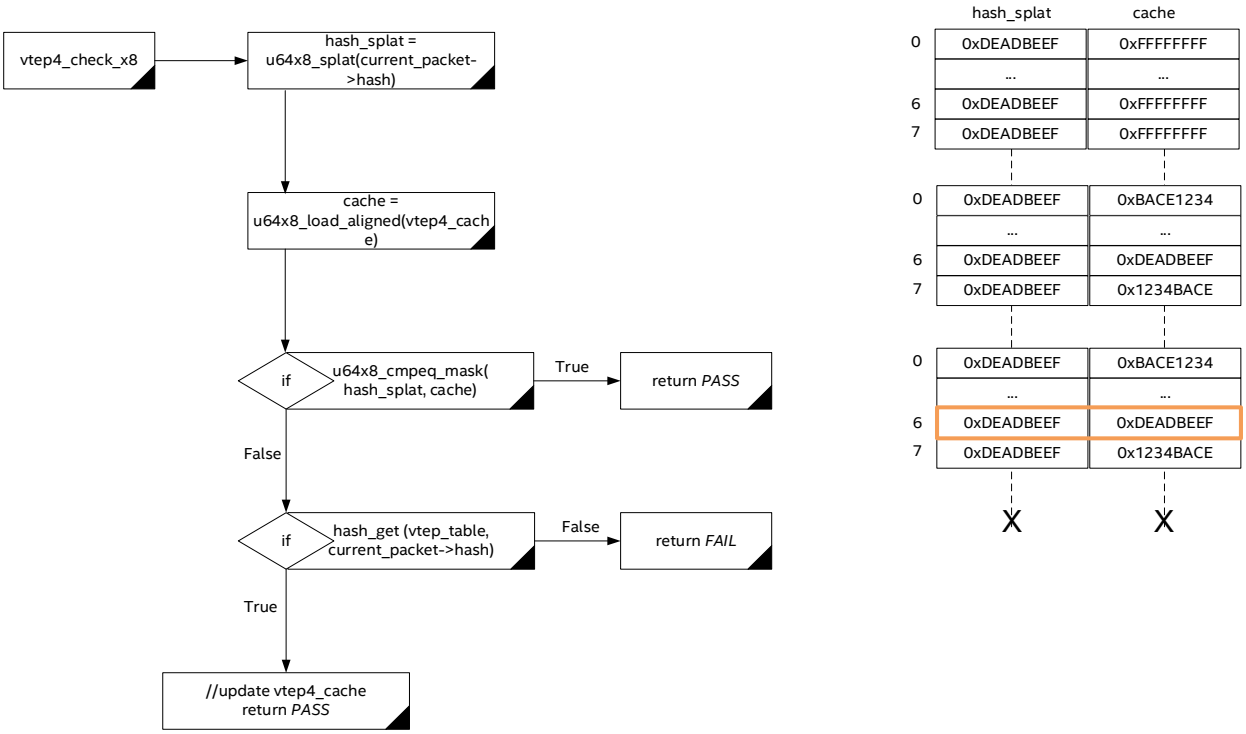

Figure 10. FD.io VPP Tunnel Decapsulation with Optimization

As shown in Figure 10, the AVX-512 optimized version `vtep4_check_x8` works as a drop in replacements for the original function `vtep4_check`. This version simply maintains a cache of eight Tunnel Endpoints compared to the single cached value in the original version of the code. As a developer you are then able to check the hash of the current packet against the hash of the eight Tunnel Endpoints simultaneously in a single `u64x8_cmpeq_mask (_mm512_cmpeq_epu64_mask)` operation. Note the heavy use of vector literals described in subsection 3.2 in Figure 10.

Figure 11 describes the flow of traffic through the FD.io VPP application. The Traffic Generator sends VxLAN encapsulated packets to a 100 GbE interface controlled by the FD.io VPP application. Each VxLAN tunnel is handled by a single virtual interface. The FD.io VPP application terminates the VxLAN tunnels and forwards the packets onto the next interface via an L2 bridge. The Traffic Generator receives the decapsulated packets and measures the performance in packets per second.

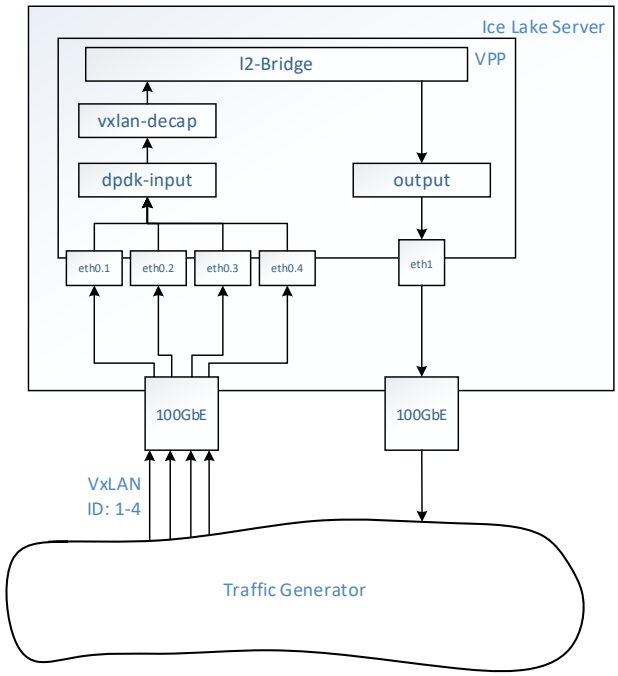

Figure 11. VPP VxLAN Tunnel Termination Application Traffic Flow Diagram (4-Tunnel Example)

Figure 12 shows AVX-512 optimizations improving performance of the VxLAN decapsulation graph node by over 80% for 2, 4, and 8 VxLAN Endpoints[12]. This then translates into a performance improvement of up to 11% for the entire packet processing application as shown in Figure 13, where the VxLAN decapsulation graph node accounts for roughly 15-25% of the total packet processing cost depending on the test scenario.

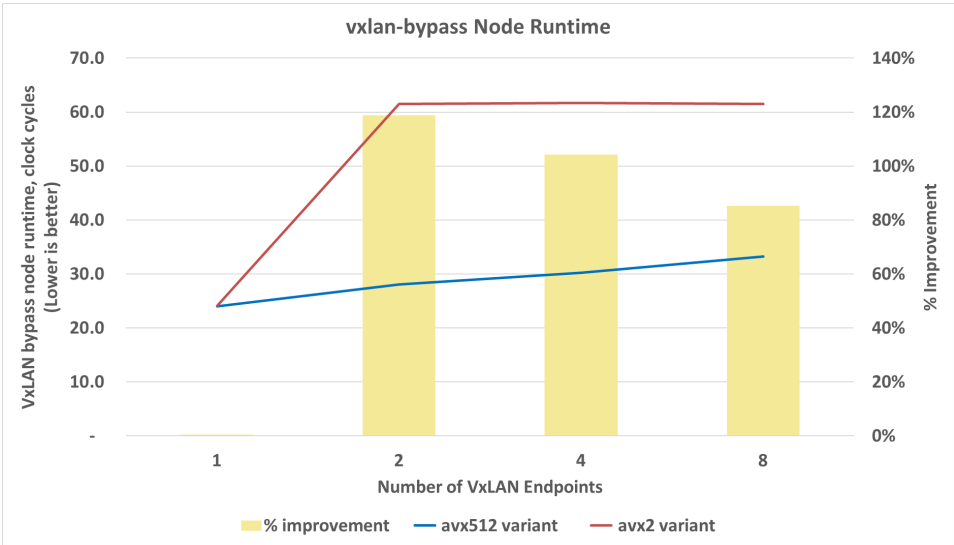

Figure 12. FD.io VPP Optimization of VPP Tunnel Termination (Clock Cycles)

Figure 12 illustrates the performance improvement with the AVX-512 optimized version VxLAN tunnel decapsulation when compared to the scalar version of the code, with the AVX-512 optimized version offering better scalability and resilience with up to eight Tunnel Endpoints.

- **Clock Cycles** (shown on the primary y-axis) is the number of cycles consumed by the VxLAN Tunnel decapsulation graph node. It is measured by the FD.io VPP *show runtime* internal metrics with a VxLAN Tunnel decapsulation test case. The *show runtime* CLI command in VPP shows the runtime cost of each VPP graph node in clock cycles per packet.
- **Number of endpoints** (shown on the primary x-axis) is the number of VxLAN Tunnels.
- **% Improvement** (shown on the secondary y-axis) is the percentage improvement obtained using AVX-512 version over the scalar version of the code.

[12] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Note: The single VxLAN endpoint case has a small drop in performance of 2 cycles per packet associated with additional cost of the AVX-512 compare. This drop, although visible in runtime statistics in Figure 12, is absent in the throughput measurements in Figure 13.

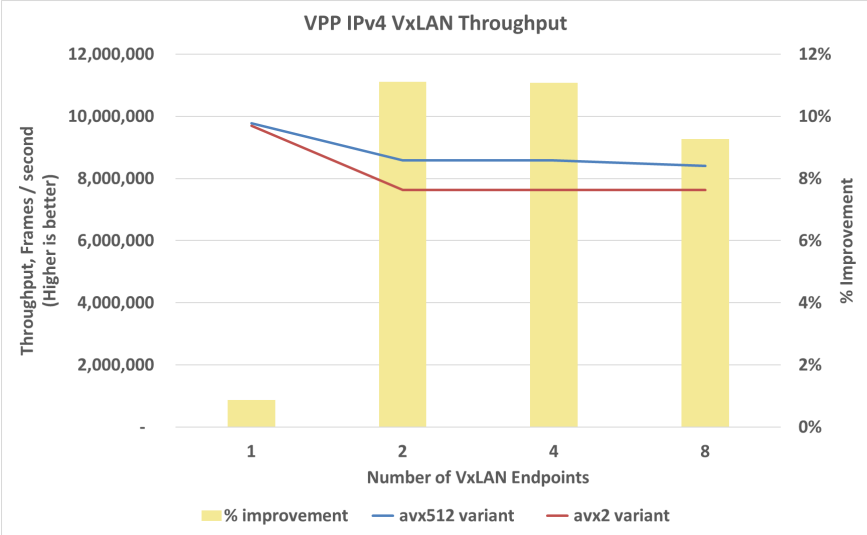

Figure 13: FD.io VPP Optimization of VPP Tunnel Termination (Throughput)

Figure 13 similarly illustrates the performance improvement in the VxLAN packet processing pipeline, where again we see improved scalability and resilience of the AVX-512 optimized version in the throughput numbers.

- **Frames/second** (shown on the primary y-axis) is the throughput of the VxLAN Tunnel decapsulation test case in frames per second.
- **Number of endpoints** (shown on the primary x-axis) is the number of Tunnel Endpoints.
- **% Improvement** (shown on the secondary y-axis) is the percentage improvement[13] obtained using the AVX-512 version over the baseline scalar version of the code.

The test configuration used to perform the above tests are listed in Table 8.

Table 8. VPP VxLAN Tunnel Termination Test Configuration

PARAMETER	DESCRIPTION
Test by	Intel
Test date	28 March 2021
Platform	Intel Corporation Reference Platform*
# Nodes	1
# Sockets	2
CPU	Intel® Xeon® Gold 6338N CPU
Cores/socket, Threads/socket	32, 64
CPU Frequency	2.2 GHz
L3 Cache	48MiB
microcode	0x8d055260
BIOS version	WLYDCRB1.SYS.0020.P86.2103050636
System DDR Mem Config: slots/capacity/run-speed	16/16384 GB/2667 MT/s
Turbo, P-states	OFF
Hyper Threads	Enabled
NIC	Intel Corporation Ethernet Controller E810-2CQDA2 for QSFP (rev 02)
OS	Ubuntu 20.04.1 LTS (Focal Fossa)
Kernel	5.4.0-40-generic
VPP Version	VPP 21.01-release

[13] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Compiler	GCC 9.3.0-10 ubuntu2
	*Intel® Reference Platform (RP) for 3rd Generation Intel® Xeon® Scalable Processor

6 Summary

If you are a Network Software Engineer, motivated to solve compute-bound problems in your packet processing software, this document's intention is to clearly establish that Intel® AVX-512 development skills are a worth-while addition to your skill set and provide enough pointers to get started.

The Vectorization Approaches section describes the different development approaches of vector intrinsics and vector literals for writing AVX-512 packet processing code. The Microarchitecture Optimizations section covers the software configuration and APIs that enable Intel® AVX-512 optimizations, detailing how Intel® AVX-512 support is detected and implemented in packet processing software. The

Optimization Examples section concludes this document by walking through simple examples of significant optimizations achieved in well-known packet processing software with Intel® AVX-512 instructions.

DPDK and FD.io are both great places to start writing packet processing software with the AVX-512 instruction set, as they contain features to detect and enable AVX-512 optimizations at runtime. They provide access to the vector intrinsics and literals used to write optimized packet processing code, and they provide examples of the kinds of optimizations that accelerate a packet processing application.

So, do not wait, now is the time to get started as DPDK and FD.io provide a great foundation for your next AVX-512 optimized packet processing application.

0222/DN/WIT/PDF 634192-003US