Toward an Ideal NDN Router on a Commercial Off-the-shelf Computer

Junji Takemasa
Osaka University
Suita, Osaka, Japan
j-takemasa@ist.osaka-u.ac.jp

Yuki Koizumi
Osaka University
Suita, Osaka, Japan
ykoizumi@ist.osaka-u.ac.jp

Toru Hasegawa
Osaka University
Suita, Osaka, Japan
t-hasegawa@ist.osaka-u.ac.jp

ABSTRACT

The goal of the paper is to present what an ideal NDN forwarding engine on a commercial off-the-shelf (COTS) computer is supposed to be. The paper designs a reference forwarding engine by selecting well-established high-speed techniques and then analyzes state-of-the-art prototype implementation to know its performance bottleneck. The microarchitectural analysis at the level of CPU pipelines and instructions reveals that dynamic random access memory (DRAM) access latency is one of bottlenecks for high-speed forwarding engines. Finally, the paper designs two prefetch-friendly packet processing techniques to hide DRAM access latency. The prototype according to the techniques achieves more than 40 million packets per second packet forwarding on a COTS computer.

CCS CONCEPTS

• **Networks** → **Routers**; *Network design principles*; Packet-switching networks;

KEYWORDS

Named Data Networking, Forwarding, Router Architecture

ACM Reference format:
Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. 2017. Toward an Ideal NDN Router on a Commercial Off-the-shelf Computer. In *Proceedings of ICN '17, Berlin,Germany, September 26–28, 2017,* 11 pages.
DOI: 10.1145/3125719.3125731

1 INTRODUCTION

A software router, which is built on a hardware platform based on a commercial off-the-shelf (COTS) computer, becomes feasible because of recent advances in multi-core CPUs and fast networking technologies for COTS computers. For notation simplicity, a COTS computer is simply referred to as a computer, hereafter. It is a promising platform for both IP and future Named Data Networking (NDN) routers since it is low-cost, energy-efficient, and flexible. Fast IP packet forwarding is a research issue for a long time and hardware architecture for CPU-based forwarding engines is designed [1, 15] assuming that data structures for IP forwarding are stored on fast memory devices like static random access memory (SRAM) devices. Dobrescu et al. [5] exploit parallelism by leveraging multiple CPU cores or servers. The forwarding engines and servers which the studies assume are current COTS computers and their success shows ability of fast packet forwarding of software routers without using special devices like general-purpose computing on graphics processing unit (GPGPU) devices [6]. Successively, compact data structures have become a research issue and many trie-based structures are designed. Among them, Degermark et al. [4] design multi-bit trie data structures by replacing consecutive elements in a trie to a single element. Such efforts enable it to store increasing IP prefixes on latest SRAM devices.

Inspired by the success, focusing on time complexity caused by the larger number of name prefixes than that of IP prefixes, significant efforts have been directed toward sophisticated data structures and algorithms to optimize the time complexity for fast NDN packet forwarding [3, 17, 18, 26]. We have enough design choices such as *Bloom filter-based*, *trie-based*, and *hash table-based* data structures and algorithms. Despite their successes, most of the data structures and algorithms assume that main part of data structures is stored on fast memory devices. However, it is obvious that slow memory devices, i.e., dynamic random access memories (DRAMs), should be used. Hiding slow memory access latency is a research issue in fast NDN packet forwarding, which is not well discussed in IP packet forwarding. Thus, data structures and algorithms for high-speed software NDN routers must be selected from the perspectives of the number of DRAM accesses as well as their time complexity. Bloom filter-based data structures obviously require more DRAM accesses than hash table-based ones since Bloom filter-based ones generally use hash tables behind their Bloom filters. We focus on either trie-based or hash table-based data structures and algorithms.

One way to resolve the slow DRAM access latency is to use a compact data structure so that the number of DRAM accesses is reduced. Trie-based data structures are generally smaller than hash table-based ones since common data, such as common prefixes, can share the storage area for storing the common part in the case of trie-based ones. For instance, Song et al. [18] propose a trie-based compact FIB so that it can be placed on SRAMs to avoid slow DRAM accesses. Though the data structure is suitable for hardware NDN routers, it is not suitable for software NDN routers. Unlike on hardware routers, data cannot be pinned at CPU caches[1] on computers since eviction and replacement on the CPU caches are governed by their CPUs rather than user-space programs and operating systems. In other words, fetching data from DRAMs is

ICN '17, Berlin,Germany

DOI: 10.1145/3125719.3125731

[1] To avoid confusion, we refer to caches on a CPU and an NDN router as *cache* and *content store (CS)*, hereafter.

unavoidable on computers, and therefore the compactness of data structures is not the best metric for choosing data structures.

In order to address the problem, this paper claims that *hiding the DRAM access latency as well as reducing the number of DRAM accesses is a vital key to developing a high-speed software NDN router.* This is because the number of DRAM accesses never gets zero even though the sophisticated data structure and algorithm are adopted. Latest CPUs support *instruction and data prefetch*, which allows instructions and data to be fetched to CPU caches in advance of actual usage. With the prefetch mechanism, software NDN routers can hide the large DRAM access latency. In the case of a trie-based data structure, it is hard to hide the DRAM access latency with the prefetch mechanism because where to access cannot be estimated in advance of its access. More precisely, since a trie is searched by traversing vertices from its root, the address of the next vertex to be traversed cannot be estimated before its parent is fetched. A hash table-based data structure is a more promising than the existing data structures since to hide the DRAM access latency is easier than a trie-based one since its access patterns are predictable if hashes of keys are available in advance.

Among proposing hash table-based data structures and algorithms, our software NDN router is based on the design proposed by So et al. [17] because it is one of the fastest software as far as we know. So et al. [17] also suggest to use the prefetch mechanism; however, their implementation does not fully utilize a CPU due to waits for several DRAM accesses for which cannot be compensated by the data prefetch mechanism. More precisely, their FIB lookup algorithm reduces the number of DRAM accesses, whereas it increases the number of accesses that are not hidden by using the data prefetch mechanism. Hence, there is still room for further improvements in the forwarding speed of software NDN routers.

In this paper, we realize the further improvements with the following two steps: First, to unveil true bottlenecks of a software NDN router, we conduct bottleneck analyses on a software NDN router from the following two perspectives: *comprehensive* and *microarchitectural analyses*. For the comprehensive bottleneck analysis, many of NDN functions and tables are implemented and investigated how they utilize CPU caches rather than analyzing each of the functions and tables. This is because that these functions and tables share a single CPU cache, of which eviction and replacement policies are governed by CPUs, and hence they have correlated with each other. Regarding the microarchitectural bottleneck analysis, we analyze the time spent for processing packets at the level of CPU pipelines and instructions rather than macroscopic analyses, which measure overall computation time or throughput.

Second, on the basis of the observations, we propose prefetch-friendly protocol processing techniques toward an ideal software NDN router. Our extension is just one step extension from the design [17]; nevertheless, the last one step toward an ideal software NDN routers is challenging and significantly improves the forwarding speed of a software NDN router.

The key contributions of this paper are summarized as follows: First, as far as we know, this is the first study that summarizes observations found in existing studies to build high-speed NDN routers on computers and compiles the observations into a design guideline. According to the guideline, we reveal that a hash table is a suitable data structure for the PIT, CS, and FIB of a software NDN router. Second, through the microarchitectural bottleneck analysis, we reveal that the bottleneck of a software NDN router is the DRAM access latency. The comprehensive analysis reveals dependencies between data structures for NDN protocol processing. Based on it, we develop prefetch-friendly NDN protocol processing that exploits the potential computing capabilities of computers by hiding most of the DRAM access latency. Third, we implement a proof-of-concept prototype and empirically prove that it realizes the nearly optimal forwarding speed.

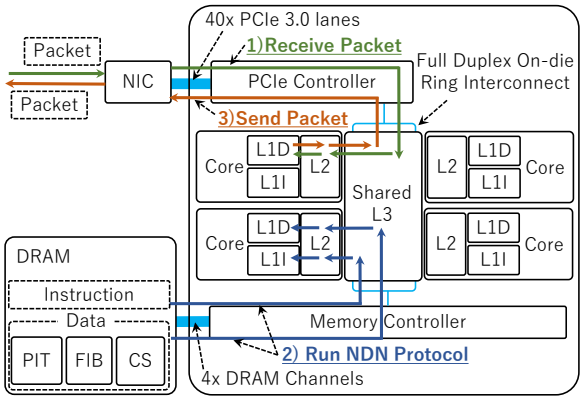


Figure 1: Router hardware architecture

The rest of this paper is organized as follows. First, we briefly overview the reference software router architecture in Section 2. In Section 3, we summarize existing techniques for high-speed NDN routers and compile them into a rationale behind our high-speed software NDN router. Then, our prefetch-friendly packet processing is presented in Section 4. We explain a method to analyze bottlenecks of software NDN routers in Section 5. We implement a proof-of-concept prototype of our router and empirically measure the packet forwarding rate in Section 6. We briefly introduce related work in Section 7, and finally conclude this paper in Section 8.

2 REFERENCE ARCHITECTURE

In this section, we first summarize observations found in existing studies and compile them into reference hardware and software architecture. Then, we describe the remaining issues.

2.1 Hardware Architecture

A reference hardware platform is illustrated in Fig. 1. It is based on the latest computer architecture, i.e., Broadwell-based Intel Xeon family CPUs [10], double data rate fourth-generation synchronous (DDR4) DRAMs, and latest network interface cards (NICs).

CPU Cores and Caches. Each CPU has several CPU cores, and they are interconnected with one or two full-duplex on-die ring interconnects. The CPU has L1 instruction, L1 data, L2, and L3 caches. For the sake of notation simplicity, L1 instruction and L1 data caches are referred to as L1I and L1D caches, respectively. In contrast to the L3 cache, which is shared with all CPU cores, each of the CPU cores has an exclusive L2, L1I, and L1D caches.

Data Transfer between CPU and NIC/DRAM. The DRAM and NICs are connected with the CPU via a memory controller and a PCI express (PCIe) 3.0 controller on its die. PCIe devices, such as network interface cards (NICs), are connected via the PCIe controller. The CPUs and NICs support the Intel Data Direct I/O (DDIO) technology [8], which allows data to be transferred directly between NICs and the L3 cache without passing through the DRAM. The data transfer rate via the PCI controller is 96 Gbytes/s and it is likely that data is prefetched at any time after all packets in the NIC are transferred to the L3 cache in a batch. The memory controller supports DDR4 DRAM and has four DDR4 DRAM channels. The allocation rule of memory requests from the memory controller to the DRAM channels is determined by the memory management library provided by DPDK [11]. Data, including NDN tables and instructions, is placed on the DRAM and it is fetched via the memory controller. The access latency to the DRAM is one or more orders of magnitude more than that to CPU caches. To hide the DRAM access latency, the CPU has two types of prefetch mechanisms: *hardware* and *software prefetch mechanisms*. The CPU automatically fetches instructions and data with the hardware prefetch mechanism if their accesses can be predicted in advance. In addition, data can be speculatively fetched to the caches using the software prefetch mechanism, which is controlled by the software with the PREFETCH instruction [9]. How to issue the PREFETCH instruction is a vital key to hide the large DRAM access latency.

2.2 Software Architecture

We describe the software from the following three perspectives: multi-threading, packet I/O processing, and NDN protocol processing strategies.

2.2.1 Overview of Software Design Issues. The packet forwarding process of NDN is more complex than that of IP due to its flexible name structure, and hence reducing the computation time for processing Interest and Data packets is a crucial issue, as we have revealed in our previous study [20]. To reduce the computation time, the software must be optimized from the following two perspectives: First one is to optimize the time complexity of the NDN packet forwarding process and second one is not to make computing units idle for exploiting capabilities of computers.

To optimize the time complexity, many studies present sophisticated data structures and algorithms; however, most of the data structures and algorithms are not specialized for performing on a computer. In Section 3, a hash table-based data structure and algorithms for the data structure is carefully chosen in order to exploit the potential capabilities of computers.

To exploit capabilities of computers, the software must eliminate obstacles that make computing units idle. There are several factors that make the computing unit idle: 1) *mutual exclusion*, 2) *hardware interrupts*, and 3) *instruction pipeline stalls*. When an area on memory devices is locked with mutual exclusion, threads waiting the area being free become idle. In the similar way, hardware interrupts and instruction pipeline stalls also cause threads being idle. One way to compensate for such idle time is to increase the concurrency, i.e., the number of threads, so that other threads can use idle computing units. For instance, a prototype software NDN router, named Augustus [13], uses Hyper-Threading Technology. However, this approach may introduce many context switches, and the context switches also results in significant overheads. In this sense, we suggest that the number of threads should be kept equal to or less than the number of CPU cores.

Most of the obstacles due to mutual exclusion and hardware interrupts have been already resolved by existing studies. The rest of this section summarizes the studies and compiles a design guideline from observations found in the studies.

2.2.2 Eliminating Mutual Exclusion. To utilize computing capabilities derived from multi-core CPUs, we have to carefully design multi-threading strategies. We choose the strategy of assigning one thread to one NDN packet rather than assigning a sequence of threads to one packet as a pipeline to prevent the pipeline from being disturbed by I/O waits. We focus on two important issues for the multi-threading strategies: to eliminate mutual exclusion and to balance the workloads for processing packets equally across all threads. For resolving two factors, we focus on exclusive NDN tables and chunk-level sharding. Mutual exclusion for any shared data would result in a serious performance degradation.

To eliminate mutual exclusion, each thread exclusively has its own FIB, PIT, and CS, thereby preventing it from accessing the tables of the other threads, as the existing prototypes of software NDN routers adopt [13, 17].

Arriving packets are forwarded to threads according to their names by using receiver side scaling (RSS), which allows NICs to send different packets to different queues to distribute the workloads among CPUs. As Saino et al. [16] have investigated, chunk-level sharding, which distributes chunks to CSs based on the result of a hash function computed on the names of the chunks, distributes chunks equally to each CS. According to their result, the paper assumes that the software distributes packets equally to each thread according to hashes of their names at the NICs using RSS. Please note that partial name matching of Interest and Data names is not supported in the paper because the sharding mechanism using RSS requires that a Data packet is explicitly requested by its exact name.

2.2.3 Eliminating Hardware Interrupts. To eliminate hardware interrupts, we adopt the DPDK user-space driver [11], which allows user-space software to bypass the protocol stacks of operating systems and provides a way to transfer data directly from NICs to the software.

2.3 Remaining Design Issues

The remaining issues are twofold: one is to choose an appropriate data structure and algorithm that exploit the potential performance of computers and the other is to eliminate instruction pipeline stalls. Section 3 chooses the appropriate data structures and algorithms. Section 4 propose prefetch-friendly packet processing techniques that hide large DRAM access latency, focusing on waits for data to be fetched from the DRAM as a main cause of the pipeline stalls.

3 RATIONALE BEHIND DATA STRUCTURE AND ALGORITHM

3.1 Overview

The NDN packet processing mainly consists of the three tables, i.e., CS, PIT, and FIB, and algorithms to manipulate the tables. They have

different properties in their access patterns and search algorithms, and therefore we develop different design guidelines specialized for them.

CS and PIT Design. In contrast to a FIB, which is a read-only table during the packet forwarding process, a CS and a PIT are modified during the process, that is, entries are inserted to and deleted from the CS and PIT. Hence, a data structure for the PIT and CS should support insertion and deletion as well as lookup of entries at constant average cost, i.e., O(1) average time complexity. A hash table is one of the best data structures having such O(1) algorithms. Note that most of the existing approaches to accelerate the computing speed of per-packet caching, such as object-oriented caching [23] and cache admission algorithms [19], are applicable to a hash table-based CS. The unified table [17], where the CS and PIT are merged to reduce the number of lookup operations, is also applicable to a hash table-based CS and PIT.

FIB Design. Since a FIB is a mostly read-only table, the computation complexity of inserting and deleting entries does not have much impact on the forwarding speed. In contrast, its lookup algorithm has to be carefully selected since longest prefix matching is more complex than exact matching, which is used for searching the CS and PIT. In the following subsections, we discuss advantages and disadvantages of several FIB design choices and chose a hash table-based FIB for our software NDN router.

3.2 Data Structures for FIB

There exist promising three kinds of data structures to realize a FIB: 1) Bloom filter-based FIB, 2) Trie-based FIB, and 3) Hash table-based FIB. We choose a hash table as an underlying data structure.

3.2.1 Bloom Filter-Based FIB. A *Bloom filter* [2] is a space-efficient data structure for representing a set and it supports membership queries. The space efficiency is realized at the cost of a small portion of false positives. Several Bloom filter-based FIBs are proposed. For instance, Wang et al. [24] have developed a Bloom filter-based FIB and a lookup algorithm for the FIB. Dai et al. [3] have proposed Bloom filter-based indexes to accelerate accesses to entries in a CS, PIT, and FIB. To compensate for the false positives incurred by Bloom filters, the Bloom filter-based FIBs generally use hash tables behind the Bloom filters, which results in the increase in accesses to the CPU caches or the DRAM compared to pure hash table-based FIBs. The increase in the number of accesses to the CPU caches or the DRAM makes the elimination of the instruction pipeline stalls difficult, and hence we do not suggest to use Bloom filter-based FIBs on software NDN routers.

3.2.2 Trie-Based FIB. A *trie* is a tree-based data structure that stores an associative array. In the case that a trie is used as an underlying data structure of a FIB, its keys are prefixes and its values are corresponding faces, and faces are looked up by traversing vertices from the root vertex. *Patricia* is a compressed version of a trie, where a vertex that is the only child is merged with its parent.

The advantages of Patricia-based FIBs are twofold: one is that the storage size for storing a Patricia-based FIB is, in general, smaller than that of a hash table-based FIB because several prefixes having a common part in the prefixes can share the storage area for storing the common prefix. For instance, Song et al. [18] propose a compact binary Patricia, which is small enough so that it is placed on the SRAM of a hardware router, as a data structure of a FIB.

The other advantage is that the probability of a part of a Patricia-based FIB being on the CPU caches is larger than that of a hash table-based FIB. Since the Patricia-based FIB is always traversed from its root vertex, vertices of which depth is low are likely to be on the CPU caches. In contrast, accesses to hash table are independent due to the random nature of hash values, and hence any part of a hash table-based FIB is unlikely to be on the CPU caches.

The disadvantage of Patricia-based FIBs is the difficulty in hiding the DRAM access latency. Despite the small size of the Patricia, it is impossible to store the whole Patricia on the CPU caches because of the following two reasons: First, the size of the CPU caches is much smaller than that of the SRAM on hardware routers. Furthermore, the size of the L3 cache for a CPU core becomes even smaller since the L3 cache are shared with all CPU cores. For instance, the average size of the L3 cache for each CPU core is about 2.5 Mbytes in the case of Xeon E5-2699 v4 CPU, which has the 55 Mbytes L3 cache shared by 22 CPU cores [10]. Second, data cannot be pinned on the caches since eviction and replacement of data on the caches are governed by CPUs. In other words, a part of the Patricia must be stored on the DRAM in the case of a software NDN router.

The latency to access the DRAM must be hidden not to make a CPU core idle; however, in the case of the Patricia, a sophisticated algorithm should be designed to hide DRAM access latency as described below. Since the Patricia is searched by traversing vertices from the root, the address of the next vertex to be traversed cannot be estimated before its parent is fetched.

3.2.3 Hash Table-Based FIB. A *hash table* is also a data structure to realize an associative array. In the similar way to a Patricia-based FIB, its keys are prefixes and its values are faces corresponding to the keys. Many studies adopt a hash table as an underlying data structure of a FIB. So et al. [17] propose a hash table-based FIB, which packs pointers to several hash entries on a single hash bucket so that the hash bucket fits a single CPU cache line, and it reduces the number of data fetches from the DRAM. We refer to buckets and entries of a hash table as *hash buckets* and *hash entries* so that they are easily differentiated with the other entries like FIB or PIT entries.

In contrast to a Patricia-based FIB, the advantage of a hash table-based FIB is that to hide the DRAM access latency is easier than a Patricia-based FIB since its access patterns are predictable if hashes of keys are available in advance. Its disadvantage is that the data consisting of the hash table-based FIB would not be on the CPU caches, i.e., it would always be in the DRAM. This is because there is no correlation among accesses to a hash table due to the nature of complete randomness in hash values. This causes frequent data fetches from the DRAM.

The Patricia-based FIB and the hash table-based FIB have pros and cons compared to each other. However, the hash table-based FIB outperforms the Patricia-based one in terms of the number of DRAM accesses, as we will evaluate it in Section 6. Accordingly, we select a hash table as an underlying data structure of the FIB.

3.3 Algorithms for Hash Table-Based FIB

To search a hash table-based FIB, lookup operations need to be iterated until the longest matched prefix is found. Several algorithms that determine the order of the iteration are proposed to reduce the number of iterations. There exist the following promising two algorithms: *longest-first search* and *2-stage search* algorithms. In the following subsections, we discuss their advantages and disadvantages and we select a longest-first search algorithm since its simplicity enhances our prefetch-friendly packet processing described in Section 4.

3.3.1 Longest-First Search Algorithm. The longest-first search algorithm is a commonly used algorithm. It starts a longest prefix match operation with the longest prefix of a name and continues to a shorter prefix until it finds the longest matched prefix.

3.3.2 2-Stage Search Algorithm. The 2-stage search algorithm [17] reduces the average number of iterations by starting the search from a certain shorter prefix (M components) than the queried prefix. Every FIB entry having a prefix with M components maintains the maximum length of prefixes that start with the same prefix in the FIB. By checking the maximum length, it determines either continuing to search for a shorter prefix or restarting to search for a longer one.

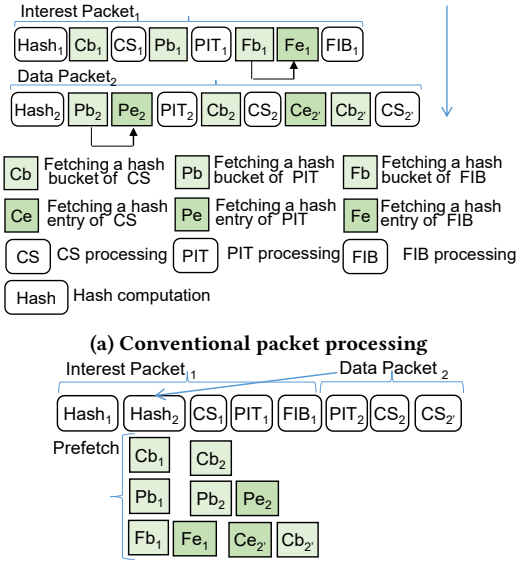
3.3.3 Discussion on Advantages and Disadvantages. According to the simulation results in [17], the 2-stage search algorithm reduces the number of iterations compared to the longest-first search algorithm; nevertheless, it may increase the number of fetches of data from the DRAM. For each iteration, both algorithms fetch a hash bucket from the DRAM. Additionally, they fetch several hash entries from the DRAM. Assuming that the probability of hash collisions is sufficiently low, the longest-first search algorithm ensures that the number of fetches of hash entries is one regardless of the number of iterations. This is because it fetches a hash entry only when it finds the longest matched prefix. In contrast, the 2-stage search algorithm fetches hash entries of a prefix with M components and the longest matched prefix when it does not find a FIB entry with a component length of M. As we will discuss later in Section 4, hiding the accesses to hash entries is more difficult than those to hash buckets. For these reasons, we use the longest-first search algorithm for our software NDN router.

4 PREFETCH-FRIENDLY PACKET PROCESSING

Prefetching is a key technique to hide CPU cycles during the pipeline stalls caused by data fetches from DRAMs. An important requirement to the success is that sufficient time passes after the prefetch by the access. After identifying data fetches of which cycles cannot be hidden by conventional packet processing flows, this section designs a prefetch-friendly packet processing flow to circumvent pipeline stalls caused by the identified fetches.

4.1 Unhidden Data Fetches in Conventional Packet Processing

Since data pieces stored on the DRAM are only the hash buckets and hash entries of the three hash tables like a CS, a PIT and a FIB,



(a) Conventional packet processing

(b) Prefetch-friendly packet processing

Figure 2: Packet processing flow

it is important to know precisely when a thread knows an address to a hash bucket or a hash entry and when it accesses the bucket or the entry. Figure 2(a) shows typical flows of handling an Interest packet and a Data one. In the figure, the two packets are received in sequence and we call them the first packet and the second packet, respectively. The suffixes 1 and 2 mean the first packet and the second one, respectively. The two flows are the longest ones, which means that two flows are the most time-consuming. For example, in the case of an Interest packet, all three hash tables are accessed. Since the Interest packet does not hit the CS, the thread searches the hash table of FIB to know the outgoing face after searching the PIT and creating an entry at the PIT. Please note that since a PIT entry created in the PIT is written back to the DRAM in the background, hereafter, we do not discuss such write back. On the contrary, when a Data packet is received, the entry in the PIT is removed and then the Data packet is inserted into the CS. It means that the hash bucket and the hash entry for the replaced Data packet in the CS are accessed as well, as shown by $Cb_{2'}$ and $Ce_{2'}$ in the figure.

From Fig. 2(a), we obtain two observations preventing stalls from being hidden. First, the first hash bucket (Cb_1 or Pb_2 in the figure) cannot be prefetched because its address is determined by the hash computation just before. Apparently, there is no time between the address computation, i.e., hash computation, and the data fetch. Second, hash tables accessed after accessing that of either the CS or the PIT are not predetermined and they are determined after checking the CS or the PIT.

4.2 Prefetch Strategies

We design the two prefetch strategies, i.e., *Pre-Hash Computation* and *Speculative Hash Table (HT) Lookup*, with the following assumptions: Since four DRAM channels are provided, it is reasonable that

multiple data pieces in the DRAM are fetched in parallel, i.e., up to four data fetches. The number of CPU cycles consumed by handling a PIT, a CS and a FIB is larger than that consumed by a fetch of either a hash bucket or a hash entry. It means that such data fetches are hidden by computation of handling the tables.

We describe the two strategies according to Fig. 2(b). The sequence in Fig. 2(b) is obtained from that of Fig. 2(a) by applying the strategies. First, Pre-Hash Computation is inspired by the first observation of the previous subsection. In order to hide stalls caused by a fetch of the first hash bucket, we should leverage computation of a subsequent packet. It means that a sequence of computation should be reconsidered for multiple packets. We choose two consecutive packets because the large number of packets incurs states for handling multiple packets, which may be stored on DRAMs in the worst case. Thus, we take an approach to precompute a hash value of the subsequent packet just after finishing the hash computation of the first packet. In parallel, hash buckets of the CS, PIT and FIB are fetched from the DRAM as illustrated by Cb_1, Pb_1 and Fb_1 in the figure. The hash computation hides stalls caused by a fetch of the first hash bucket.

Second, after fetching the above-mentioned hash buckets, hash entries of all the hash tables, i.e., the CS, PIT and FIB, of the first packet become able to be fetched from the DRAM. Thus, all the hash entries, including those which may not be accessed due to longest prefix match results, are speculatively fetched even if some of hash entries are not used later. The strategy is called Speculative HT Lookup. Speculative HT Lookup is applied to hash entries for the subsequent packet after fetching hash buckets for it. Please note that it is applied to all hash buckets and hash entries which correspond to name prefixes of which length is shorter than the name of the packet to hide stalls in the case of longest prefix matching.

Stalls which are caused by late prefetches and overheads incurred by redundant prefetches of Speculative HT Lookup are discussed later in Section 6. Another concern is that prefetching hash buckets and entries of the second packet to the CPU cache would evict or overwrite those of the first packet from the CPU cache; however, it is likely that such evictions would not occur because the average size of newly fetched data for processing the two packets is relatively smaller than that of the CPU cache. The average size of data fetched for processing two packets is approximately 9,000 (4,500 bytes/packet) [20], whereas the average size of the L3 cache for one CPU core is approximately 2.5 Mbytes because 55 Mbytes is shared with 22 CPU cores [10].

5 METHOD OF BOTTLENECK ANALYSIS

To identify bottlenecks of software NDN routers, we conduct a *microarchitectural analysis*, which analyzes how individual hardware components of the CPU spend time for processing NDN packets at the level of instructions and instruction pipelines. This section summarizes the method of the microarchitectural analysis.

5.1 Analysis on Pipeline Stalls

To know the bottleneck of the software NDN router, we analyze where hazards occur in the pipeline and how long the pipeline is stalled due to the hazards. We use the top-down micro-architecture analysis method proposed by Yasin [25]. Before analyzing the pipeline hazards, we explain the pipeline and flows of instructions and data on a CPU.

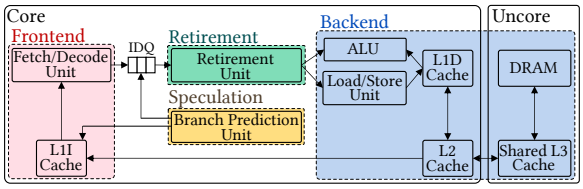


Figure 3: Hardware units on a CPU and four main stages of instruction pipeline

The pipeline has four main stages: *frontend*, *backend*, *speculation*, and *retirement*, as shown in Fig. 3. The frontend stage is responsible for fetching instructions from DRAMs or caches and feeding them to the backend stage, and the speculation stage supports the frontend stage in its instruction fetches, using the branch prediction unit. In contrast, the backend stage is responsible for executing the instructions, fetching necessary data from the DRAM or the caches and storing data to the caches. Finally, instructions executed successfully are retired at the retirement stage.

The CPU cycles spent for processing the pipeline are also categorized into four major parts: CPU cycles spent during the pipeline stalls at the *frontend* stage due to *instruction cache misses*, that spent during pipeline stalls at the *backend* stage due to *data cache misses*, that wasted at the *speculation* stage due to *branch prediction misses*, and that spent when *instructions are successfully executed and retired* at the *retirement* stage. In the ideal case, where instructions are executed without pipeline stalls, the CPU cycles spent at the stages except for the retirement stage become zero. That is, the CPU cycles spent at the retirement stage are equivalent to the minimum computation time.

The pipeline stalls at each of the stages can be further categorized into more specific reasons. The pipeline stalls at the backend stage are divided into the following three categories: pipeline stalls due to waits at *load* units, at *store* units, and at *core* units, such as instruction execution units. Since the pipeline stalls at the backend stage dominate the overall pipeline stalls, we omit detailed investigations of the pipeline stalls at the frontend and speculation stages. Among them, the pipeline stalls due to waits at load units dominate the overall pipeline stalls at the backend stage, and hence we further categorize the pipeline stalls at the load units into the following four categories: pipeline stalls to wait data to be fetched from the *L1D*, *L2*, *L3* caches, and *DRAM*.

5.2 Method to Measure CPU Cycle

CPU cycles spent at each of the pipeline stages are measured with a performance monitoring unit (PMU) on CPUs. A PMU offers a set of counters, which count the number of occurrence of hardware events. Such events include the pipeline stalls at the frontend, backend, and speculation stages and the instruction retirement at the retirement stage. Using the PMU counters, we measure CPU cycles spent at each of the four pipeline stages.

6 BOTTLENECK ANALYSIS

In this section, we implement a prototype of the designed software NDN router and conduct microarchitectural and comprehensive bottleneck analyses. The purposes are threefold: the first one is to validate our rationale, i.e., a hash table-based FIB is suitable for software NDN routers, the second one is to show that the bottleneck of existing software NDN routers is the unhidden DRAM access latency, and the last one is to prove that our proposal eliminates the bottleneck and realizes the nearly optimal forwarding speed.

6.1 Prototype Implementation

We implement a prototype of the proposed prefetch mechanisms and basic functions of NDN which include longest prefix match on FIB lookup, exact match on CS lookup and PIT lookup, and FIFO-based CS replacement. When an Interest packet is received, the prototype performs NDN functions in the same order as the block diagram of [27] except for the functions below. A nonce is not calculated because CPU cycles of the calculation is much smaller than that of the whole Interest packet forwarding [7]. Only the Best Route Strategy is supported. Partial name matching of Interest and Data names and selectors are not supported because a goal of the paper is fast name-based longest prefix match. The FIB does not have entries other than those for Interest forwarding such as those for RTT estimation because fast stateful forwarding is not a goal of this paper. When a Data packet is received, the prototype performs NDN functions in the same order. The CS is stored on DRAM devices to avoid long read latency from hard disk devices. On the contrary, the proposed prefetch mechanism is used for handling the hash table-based CS, PIT, and FIB. The prototype employs Interest and Data packets that conform to the NDN-TLV packet specification [22], and Interest and Data packets are encapsulated by IP packets, of which headers hold the hashes of their name and are used for RSS at the NICs.

6.2 Experiment Conditions

For the experiments, we use two computers with a Xeon E5-2699 v4 CPU (2.20 GHz × 22 CPU cores), eight DDR4 16 GB DRAM devices, and two Intel Ethernet Converged Network Adapter XL710 (dual 40 Gbps Ethernet ports) NIC. The operating system on the computers is Ubuntu 16.04 Server.

One computer is used as our router and the other is used as a consumer and a producer. The two computers are connected with four 40 Gbps direct-attached QSFP+ copper cables. The two links are used for connecting the consumer and the router and the other two are used for the router and the producer. That is, the total bandwidth between the consumer and the router and the router and the producer is 80 Gbps. The consumer sends Interest packets to the producer via the router. The producer returns Data packets in response to the Interest packets via the router. We set the payload size of Data packets to 128 bytes. For the experiments, the consumer generates 80 million Interest packets with different 80 million names so that all Interest packets miss at the CS and the PIT.

To generate workloads and FIB entries for experiments, we use the results of the analysis on HTTP requests and responses of the IRCache traces [12] conducted in [17]. 13,548,815 FIB entries are stored at the FIB. The average number of components in Interest packets is set to 7 and the average length of each component is set to 9 characters. The average number of components in prefixes of FIB entries are set to 4 so that the average number of FIB lookup operations per one Interest packet is slightly larger than that in [17].

6.3 Analysis on FIB Data Structures

In this subsection, we estimate the performance of a Patricia-based FIB and a hash table-based one to select either a Patricia or a hash table as an appropriate FIB data structure for software NDN routers. To differentiate the two data structures, we estimate the average number of DRAM accesses and the average CPU cycle spent during a single longest prefix match operation.

6.3.1 Estimation Model. As a Patricia-based FIB, we select a binary Patricia [18]. As an algorithm to search a hash table-based FIB, we select the longest-first search algorithm. Without our prefetch-friendly packet processing, FIB entries are always on the DRAM in the case of the hash table-based FIB. In contrast, vertices of which depths are low are likely to be on the L3 cache in the case of the binary Patricia. Hence, we first derive the number of DRAM accesses in the case of the Patricia-based FIB.

We derive the number of DRAM accesses in the following steps: First, we calculate the probability P_d of a vertex v_d of which depth is d being on the L3 cache by using R_d, which is the number of bytes fetched to the L3 cache during two consecutive accesses to v_d. $I(v_d)$ denotes the interval of the two consecutive accesses to v_d. If R_d is larger than the size of the L3 cache S, v_d will be fetched from the DRAM; otherwise it will be on the L3 cache. Second, R_d is derived from the product of N_d, which is the average number of packets processed during the interval $I(v_d)$, and r, which is the number of bytes fetched to the L3 cache for processing one packet.

Assuming that the probabilities of digits $\{0, 1\}$ in prefixes and names are identical, the probability of v_d being accessed at each longest prefix match operation is derived as $1/2^d$. Thus, v_d is accessed once every $1/(1/2)^d = 2^d$ longest prefix match operations in average. Since Interest and Data packets are one-for-one in principle, we can assume that one Data packet is processed between two longest prefix match operations. Under this assumption, the average number of Data packets being processed during the interval $I(v_d)$ is 2^d. Therefore, R_d is estimated as $R_d = r2^d$. If $R_d > S$, the vertex v_d is fetched from the DRAM. The total number of DRAM accesses is derived by adding such DRAM accesses for all vertices between the root vertex and leaf vertices. According to the analytical model [18], the average depth of all leaf nodes, which hold FIB entries, is $\log(n) + 0.3327$, where n denotes the number of FIB entries. We define a binary variable B_d and $B_d = 1$ if $R_d > S$, 0 otherwise. By using B_d, the average number of DRAM accesses is derived as $\sum_i^{\log(n)+0.3327} B_i$.

Next, we derive the number of DRAM accesses in the case of a hash table-based FIB. Since all hash buckets and hash entries of the hash table-based FIB are assumed to be in the DRAM, the numbers of hash bucket fetches and hash entry fetches are estimated as the number of lookup operations and one, respectively.

6.3.2 Estimation Results. According to the analyses conducted in [17], we set the number of FIB entries to 13,548,815, which is

the number of unique names in the IRCache traces [12], and the average number of table lookup operations per one longest prefix match operation is set to 3.87. We empirically measure r with PMU registers, and the number of bytes fetched from DRAM for processing one Data packet is 511 bytes. Assuming that Xeon E5-2699 v4 CPU, of which L3 cache size is the largest among commercially available CPUs, we set the L3 cache size S to 55 Mbytes. In this case, the average depth of the Patricia-based FIB is 24.01, and thus the average number of DRAM accesses in the case of the Patricia-based FIB is $\sum_{i}^{\log(n)+0.3327} B_d = 7$. In contrast, the number of DRAM accesses in the case of the hash table-based FIB is 3.87 + 1 = 4.87. Accordingly, we select a hash table as an underlying data structure of the FIB of our software NDN router.

6.4 Microarchitectural Bottleneck Analysis on Naive Software NDN Router

This section conducts the microarchitectural bottleneck analysis on a naive software NDN router, which corresponds to one of existing software NDN routers designed according to the rationale in Section 3, to identify a true bottleneck of the existing software NDN router. That is, the naive software NDN router does not have our prefetch-friendly packet processing.

6.4.1 Pipeline Stalls. The CPU cycles spent during the pipeline stalls at the frontend, backend, and speculation stages and the instruction executions at the retirement stage is summarized in Table 1(a). We measure the CPU cycles spent for processing one Interest or one Data packet in the following three cases: The Interest packet processing flow in the case of CS misses, the Interest packet processing flow in the case of CS hits, and the Data packet processing flow. To simplify the notation, these three cases are denoted by *I-miss*, *I-hit*, and *Data* in the tables. I-miss consumes the largest CPU cycles. Only the flow includes the FIB lookup procedure, and hence the design of FIB data structures and FIB lookup algorithms is an important issue, as we have discussed in the previous subsection. For all the three cases, the pipeline stalls at the backend stage spent the nearly one third or more of the entire CPU cycles. In the rest of paper, we focus on eliminating stalls at the backend stage because they are significant compared to those at the other stages and because a root cause of such stalls is identified as described below. Please note that eliminating stalls at the other stages is left for further study.

6.4.2 Backend Stalls. To reveal causes of the long stalls at the backend stage, we measure the stalls due to the load, store, and core units. The results are summarized in Table 1(b). Most of the pipeline stalls at the backend stage are caused by the load units waiting data to be fetched from either the caches or the DRAM.

We further drill down the stalls at the load units: stalls to wait data to be fetched from the L1D, L2, L3 caches, and DRAM. The results are summarized in Table 1(c). Most of the CPU cycles due to the stalls at the load units are spent to wait data to be fetched from DRAMs. It is a straightforward result since the DRAM access latency is much longer than those of the L1D, L2, and L3 caches. However, the number of accesses to DRAMs, which is measured with the number of retired load instructions, is surprisingly small, i.e., 4 or less, as shown in Table 2. This is because that most of data fetches from DRAMs are hidden by hardware prefetching mechanisms of the CPU and such data would be provided to core units without stalling the pipeline. However, a few DRAM accesses that cannot be prefetched by using the hardware prefetching mechanism cause the significant pipeline stalls and they would lead to the significant degradation in the forwarding speed.

6.4.3 Multi-Threaded Case. We next evaluate the CPU cycles in the case of the multi-threaded software. The results are summarized in Table 3. We allocate 2 CPU cores to the operating system and the remaining 20 CPU cores to the software NDN router. While the CPU cycles due to the stalls to wait data to be fetched from the DRAM increase as the number of threads increases, the number of L3 cache misses does not increase.

6.4.4 Observations. The pipeline stalls to wait data to be fetched from the DRAM spent the CPU cycles mostly. For the Interest packet processing flow, 31.3% of the CPU cycles are spent during the pipeline stalls to wait data to be fetched from the DRAM. Hence, to hide the latency due to DRAM accesses is an important design issue for high-speed software NDN routers.

Another important observation is that the CPU cycles due to the stalls to wait data to be fetched from the DRAM increases as the number of threads increases. This is because that there is contention at the memory controllers on the CPUs and the wait time due to the contention increases as the number of threads increases. In contrast, the average number of L3 cache misses does not change even if the number of threads increases. This is counterintuitive because the L3 cache is shared by all CPU cores. The reason for this effect is that the most part of the three tables, i.e., the PIT, CS, and FIB, is on the DRAM regardless of the number of threads, and therefore data of the tables will be fetched every time they are used. To validate that the prefetch-friendly packet processing hides the DRAM access latency, it has to be carefully evaluated in the case that the number of threads is high.

6.5 Performance Gain Due to Prefetch-Friendly Packet Processing

We next evaluate how the proposed prefetch-friendly packet processing, i.e., Pre-Hash Computation and Speculative HT Lookup, hides CPU cycles during DRAM accesses and eliminates the stalls at the backend stage.

6.5.1 Single-Threaded Case. First, we evaluate the CPU cycles for processing packets in the single-threaded case to focus on evaluating how Pre-Hash Computation and Speculative HT Lookup contribute hiding CPU cycles during DRAM accesses. For comparison purposes, we use the four variants of the software NDN routers: *Naive*, *Bucket Prefetch*, *Pre-Hash*, and *SHTL*, to which existing and our proposed optimization techniques are incrementally applied. *Naive* is the software NDN router designed according to the design rationale in Section 3 and it does not have any prefetch techniques. This is equivalent to the software proposed by So et al. [17] except for that it uses the longest-first search algorithm rather than the 2-stage search algorithm. *Bucket Prefetch* is based on *Naive* but it prefetches hash buckets for arriving packets according to the existing prefetch technique proposed by So et al. [17]. *Pre-Hash* is based on *Naive* but it has Pre-Hash Computation proposed in Section 4.

Table 1: The CPU cycles spent in the instruction pipeline

(a) Entire pipeline

	I-miss	I-hit	Data
Total	1470	860	565
Frontend	161	40	73
Backend	662	359	220
Speculation	161	18	42
Retirement	486	443	230

(b) Backend

	I-miss	I-hit	Data
Backend	662	359	220
Load	512	256	146
Store	10	11	14
Core	140	92	60

(c) Load unit

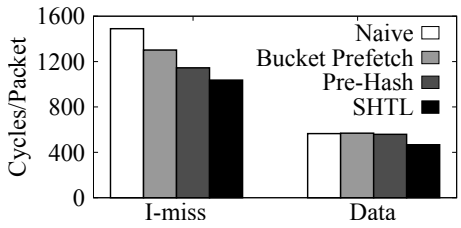
	I-miss	I-hit	Data
Load	512	256	146
L1D	33	66	20
L2	0	0	0
L3	19	28	32
DRAM	460	162	94

Table 2: The number of retired load instructions

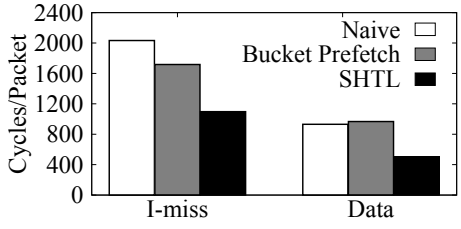
	I-miss	I-hit	Data
Load	328	289	127
L1D	322	286	122
L2	1	1	2
L3	1	1	2
DRAM	4	1	1

Table 3: The CPU cycles against the number of threads

The number of threads	1	10	20
Entire Pipeline	1470	1567	2033
DRAM cycles	460	640	878
The number of L3 misses	4	4	4



(a) Single-threaded



(b) Multi-threaded (20 threads)

Figure 4: The CPU cycles for processing one NDN packet

SHTL represents the software NDN router that has Speculative HT Lookup in addition to Pre-Hash Computation.

Figure 4(a) shows the CPU cycles for processing one Interest and Data packet. *I-miss* represents the CPU cycle for processing one Interest packet in the case of CS misses. Bucket Prefetch reduces 12.62% of the CPU cycle compared to Naive. Pre-Hash reduces additional 12.04% of the CPU cycle compared with Bucket Prefetch because hash buckets of the CS are efficiently prefetched with Pre-Hash Computation. In addition to Pre-Hash Computation, SHTL further reduces 9.37% of the CPU cycle compared to Pre-Hash Computation since Speculative HT Lookup hides the DRAM access latency for fetching hash entries of the FIB. Finally, SHTL reduces 30.34% of the CPU cycle compared to Naive.

In contrast to the case for Interest packets, the CPU cycle for processing one Data packet does not change by using Bucket Prefetch and Pre-Hash. This phenomenon is interpreted as follows: In the experiments, Data packets are returned shortly after the corresponding Interest packets are processed. When a Data packet arrives at the router, hash buckets and entries necessary for processing the Data packet are likely to be on the CPU caches since they are fetched to the CPU caches for processing the corresponding Interest packet. Bucket Prefetch and Pre-Hash, therefore, do not contribute to reduce the CPU cycles. In contrast to the experiments, if Data packets were not returned shortly, some hash buckets and entries would not be on the CPU caches. In this case, Bucket Prefetch and Pre-Hash might not reduce the CPU cycles, and the CPU cycles would be worse than that of Fig. 4(a). That is, the CPU cycle reduction due to Bucket Prefetch and Pre-Hash is not underestimated.

On the contrary, SHTL prefetches such hash buckets and hash entries irrespective of whether they are still on the CPU cache or not. SHTL reduces about 17.36% of the CPU cycle compared to Naive, Bucket Prefetch, and Pre-Hash because the DRAM access latency of accesses to hash entries and buckets to a replaced Data packet in the CS is hidden by SHTL.

6.5.2 Multi-Threaded Case. Next, we evaluate the CPU cycles reduced by the prefetch-friendly packet processing in the case of the multi-threaded software. Figure 4(b) shows the CPU cycles spent for processing one packet in the case that the software is running with 20 threads. The CPU cycles of SHTL, which includes both Pre-Hash Computation and Speculative HT Lookup, does not change compared to the single-threaded case, whereas those of Naive and Bucket Prefetch are larger than those of the single-threaded case. As we have investigated in Table 3, the DRAM access latency increases when the number of threads is large. Since Naive and Bucket Prefetch have unhidden DRAM accesses, the latency of the unhidden DRAM accesses also causes the increase in the total CPU cycles spent for processing an Interest and a Data packet. In contrast, SHTL hides most of the DRAM accesses. More precisely, SHTL does not hide accesses to hash entries of the CS since the software does not have enough amount of operations between the accesses to a hash bucket and a hash entry of the CS to hide the accesses, but SHTL hides all the other DRAM accesses. Therefore,

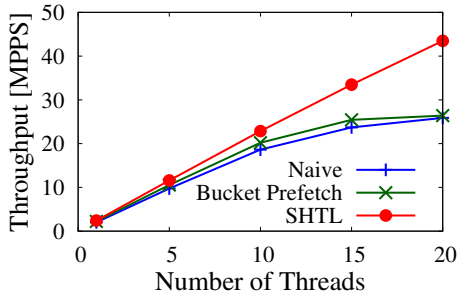


Figure 5: The forwarding speed against the number of threads

the CPU cycles spent for processing packets are approximately constant regardless of the number of threads. As a result, the absolute difference between the CPU cycles of Bucket Prefetch and SHTL becomes larger compared to the single-threaded case due to the larger DRAM access latency.

6.6 Forwarding Speed

We finally evaluate the forwarding speed of the software NDN routers. To show the worst case forwarding speed, we evaluate the situation where all Interest packets miss the CS since I-miss is heavier than I-hit, as shown in Table 1.

Figure 5 shows the forwarding speed against the number of threads. The forwarding speed of SHTL is approximately linear to the number of threads, whereas those of Naive and Bucket Prefetch are not. SHTL hides most of the DRAM accesses, whereas Bucket Prefetch still has unhidden DRAM accesses. As we have shown in Fig. 4(b), the CPU cycles spent for processing packets increase in the case of Naive and Bucket Prefetch due to the unhidden DRAM accesses. This degrades the forwarding speed. In contrast, SHTL hides most of the DRAM accesses and therefore the forwarding speed is not degraded. SHTL finally realizes up to 43.48 MPPS forwarding. We believe that it is one of the fastest software NDN router realized on a computer. Note that the forwarding speed of 43.48 MPPS is not the upper bound of the router since we use only one of two CPUs. If additional NICs can be installed on this computer, the forwarding speed will scale up to about 80 MPPS.

6.7 Overheads Due to Unnecessary Prefetches

Speculative HT Lookup causes unnecessary prefetches when the prefetched data is not used. The unnecessary prefetches cause the two kinds of overheads: One is the extra consumption of the DRAM channels and the other is the extra computation time to obtain the addresses of data pieces to be prefetched. We conduct additional measurements to validate the overheads are sufficiently small.

The first overhead is negligible because the number of bytes prefetched by Speculative HT Lookup is sufficiently small compared to that of all the fetched data. For example, to process one Interest packet in the case of a CS miss, the software fetches 16.6 cache lines in average. In contrast, the number of wastefully prefetched cache lines is 3 in the case that the number of components in the name of an Interest packet is 4. More precisely, three hash buckets for the components of which lengths are one, two, and three are wastefully prefetched. The second overhead is also negligible because the CPU cycle spend for executing the single prefetch instruction is less than 48 CPU cycles. Such overheads are small enough compared to the benefits of Speculative HT Lookup, as we show them in the previous subsections.

7 RELATED WORK

We briefly summarize related studies that are not introduced above.

Prototypes of software NDN routers have been developed. Kirchner et al. [13] have implemented their software NDN router, named Augustus, in two different manners: a standalone monolithic forwarding engine based on the DPDK framework [11] and a modular one based on the Click framework. Though the forwarding speed of Augustus is very high, it does not approach the potential forwarding speed realized by computers. Hence, analyzing bottlenecks of software NDN routers remains open.

To exploit the CPU caches is an important issue to realize the fast packet processing. Vector packet processing (VPP) [21] enhances packet processing speed by processing a vector of packets at once rather than processing each of them individually. The key idea behind VPP is to reduce the processing time by increasing the hit probability at the CPU instruction caches. However, in the case of the software NDN router, the instruction cache misses, i.e., the instruction pipeline stalls at the frontend stage, do not occupy a significant amount of the overall pipeline stalls. Furthermore, unlike VPP, our approaches focus on hiding the DRAM access latency, assuming that data cannot be pinned on the CPU caches.

Fast packet processing is also a hot research topic in the area of networked applications. Li et al. [14] develop a multi-threaded in-memory key-value stores that exploits DPDK and RSS. They have revealed that cache misses at the L3 cache increases as the number of threads increases. However, our NDN software implementation scales up to 20 threads without increasing the L3 cache misses.

8 CONCLUSION

This paper identifies what an ideal software NDN router on computers is supposed to be in the following steps: 1) we conducted the detailed study on the existing techniques for high-speed NDN packet forwarding and compiled them into the design rationale toward an ideal software NDN router. 2) We conducted the microarchitectural and comprehensive bottleneck analyses on the software NDN router and have revealed that to hide the DRAM access latency is a vital key toward an ideal software NDN router. 3) We have proposed that two prefetch-friendly packet processing techniques to hide the latency. Finally, the prototype implemented according to the rationale and the prefetch-friendly packet processing techniques achieves more than 40 million packets per second (MPPS) packet forwarding on a single computer.

ACKNOWLEDGMENTS

This work has been supported by the EU-Japan ICN2020 Project (EU HORIZON 2020 Grant Agreement No. 723014 and NICT Contract No. 184); and Grant-in-Aid for JSPS Fellows (17J07276). We thank our shepherd, Craig Partridge, for his insightful feedback.

REFERENCES

[1] Abhaya Asthana, Catherine Delph, H. V. Jagadish, and Paul Krzyzanowski. 1992. Towards a Gigabit IP Router. *Journal of High Speed Networks* 1, 4 (Oct. 1992), 281–288.

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM* 13, 7 (July 1970), 422–426.

[3] Huichen Dai, Jianyuan Lu, Yi Wang, and Bin Liu. 2015. BFAST: Unified and Scalable Index for NDN Forwarding Architecture. In *Proceedings of IEEE INFOCOM*. IEEE, 2290–2298.

[4] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. 1997. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of ACM SIGCOMM*. ACM, 3–14.

[5] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proceedings of ACM SOSP*. ACM, 15–28.

[6] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of ACM SIGCOMM*. ACM, 195–206.

[7] Toru Hasegawa, Yuto Nakai, Kaito Ohsugi, Junji Takemasa, Yuki Koizumi, and Ioannis Psaras. 2014. Empirically Modeling How a Multicore Software ICN Router and an ICN Network Consume Power. In *Proceedings of ACM ICN*. ACM, 157–166.

[8] Intel Corporation. 2012. Intel® Data Direct I/O Technology (Intel® DDIO): A Primer. (2012). Retrieved April 24, 2017 from http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf

[9] Intel Corporation. 2016. Intel® 64 and IA-32 Architectures Optimization Reference Manual. (2016). Retrieved April 24, 2017 from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[10] Intel Corporation. 2016. Intel® Xeon® Processor E5 v4 Family. (2016). Retrieved April 24, 2017 from http://ark.intel.com/products/family/91287

[11] Intel Corporation. 2017. Data Plane Development Kit (DPDK). (2017). Retrieved April 24, 2017 from http://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html

[12] IRCache. 1995. IRCache Project. (1995). http://www.ircache.net/

[13] Davide Kirchner, Raihana Ferdous, Renato Lo Cigno, Leonardo Maccari, Massimo Gallo, Diego Perino, and Lorenzo Saino. 2016. Augustus: a CCN router for programmable networks. In *Proceedings of ACM ICN*. ACM, 31–39.

[14] Sheng Li, Hyeontaek Lim, Victor W. Lee, JungHoAhn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2016. Achieving One Billion Key-Value Requests per Second on a Single Server. *IEEE Micro* 36, 3 (May 2016), 94–104.

[15] Craig Partridge, Philip P. Carvey, Ed Burgess, Isidro Castineyra, Tom Clarke, Lise Graham, Michael Hathaway, Phil Herman, Allen King, Steve Kohalmi, Tracy Ma, John Mcallen, Trevor Mendez, Walter C. Milliken, Ronald Pettyjohn, John Rokosz, Joshua Seeger, Michael Sollins, Steve Storch, Benjamin Tober, Gregory D. Troxel, David Waitzman, and Scott Winterble. 1998. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking* 6, 3 (June 1998), 237–248.

[16] Lorenzo Saino, Ioannis Psaras, and George Pavlou. 2016. Understanding Sharded Caching Systems. In *Proceedings of IEEE INFOCOM*. IEEE, 1–9.

[17] Won So, Ashok Narayanan, and David Oran. 2013. Named Data Networking on a Router: Fast and DoS-resistant Forwarding with Hash Tables. In *Proceedings of ACM/IEEE ANCS*. IEEE, 215–226.

[18] Tian Song, Haowei Yuan, Patrick Crowley, and Beichuan Zhang. 2015. Scalable Name-Based Packet Forwarding: From Millions to Billions. In *Proceedings of ACM ICN*. ACM, 19–28.

[19] Junji Takemasa, Kosuke Taniguchi, Yuki Koizumi, and Toru Hasegawa. 2016. Identifying Highly Popular Content Improves Forwarding Speed of NDN Software Router. In *Proceedings of IEEE Globecom Workshop*. IEEE, 1–6.

[20] Kosuke Taniguchi, Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. 2016. Poster: A Method for Designing High-speed Software NDN Routers. In *Proceedings of ACM ICN*. ACM, 203–204.

[21] The Fast Data Project (FD.io). 2016. Vector Packet Processing. (2016). Retrieved April 20, 2017 from https://fd.io/technology

[22] The Named Data Networking (NDN) project. 2014. NDN Packet Format Specification. (2014). Retrieved April 20, 2017 from http://named-data.net/doc/ndn-tlv/

[23] Yannis Thomas, George Xylomenos, Christos Tsilopoulos, and George C. Polyzos. 2015. Object-oriented Packet Caching for ICN. In *Proceedings of ACM ICN*. ACM, 89–98.

[24] Yi Wang, Tian Pan, Zhian Mi, Huichen Dai, Xiaoyu Guo, Ting Zhang, Bin Liu, and Qunfeng Dong. 2013. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters. In *Proceedings of IEEE INFOCOM Mini Conference*. IEEE, 95–99.

[25] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *Proceedings of IEEE ISPASS*. IEEE, 35–44.

[26] Haowei Yuan and Patric Crowley. 2015. Reliably scalable name prefix lookup. In *Proceedings of ACM/IEEE ANCS*. IEEE, 111–121.

[27] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 66–73.