2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00040

QEI: Query Acceleration Can be Generic and Efficient in the Cloud

Yifan Yuan[1], Yipeng Wang[2], Ren Wang[2], Rangeen Basu Roy Chowhury[2], Charlie Tai[2], Nam Sung Kim[1]
[1]*UIUC*, [2]*Intel*
{yifany3, nskim}@illinois.edu {yipeng1.wang, ren.wang, rangeen.basu.roy.chowhury, charlie.tai}@intel.com

***Abstract*—Data query operations of different data structures are ubiquitous and critical in today's data center infrastructures and applications. However, query operations are not always performance-optimal to be executed on general-purpose CPU cores. These operations exhibit insufficient memory-level parallelism and frontend bottlenecks due to unstructured control flow. Furthermore, the data access patterns are not cache- or prefetch-friendly. Based on our performance analysis on a commodity server, query operations can consume a large percentage of the CPU cycles in various modern cloud workloads. Existing accelerator solutions for query operations do not strike a balance between their generality, scalability, latency, and hardware complexity.**

In this paper, we propose QEI, a generic, integrated, and efficient acceleration solution for various data structure queries. We first abstract the query operations to a few regular steps and map them to a simple and hardware-friendly configurable finite automaton model. Based on this model, we develop the QEI architecture that allows multiple query operations to execute in parallel to maximize throughput. We also propose a novel way to integrate the accelerator into the CPU that balances performance, latency, and hardware cost. QEI keeps the main control logic near the L2 cache to leverage existing hardware resources in the core while distributing the data-intensive comparison logic to each last-level cache slice for higher parallelism. Our results with five representative data center workloads show that QEI can achieve $6.5\times\sim11.2\times$ performance improvement in various scenarios with low overhead.

***Index Terms*—data query, on-chip accelerator, near-cache processing**

I. Introduction

Nowadays, the combination of diverse applications and infrastructure in data centers has created great challenges for both cloud service providers and chip makers in improving data center hardware's performance and efficiency. Researchers have explored the adoption of specialized hardware (or accelerators), such as FPGAs [12, 28, 49, 66] and GPUs [9, 17, 31], to improve the performance and efficiency of important parts (compute kernels) of such workloads. These accelerators are usually connected to the CPU via an I/O interface such as PCIe. Due to the long communication latency between the core and the PCIe device [41, 58], the compute kernel has to run for a significant amount of time to amortize the communication overhead, making them unsuitable for accelerating **fine-grained** and **latency-sensitive** operations [7]. One such operation is *data query*.

Data query (or lookup), in general, refers to the process of retrieving data for a given key from one of a handful of popular data structures (see Sec. II for details). **Query operations exist in almost all data center workloads.** For example, a firewall can use a list of blacklisted keywords to query on a traffic flow to identify malicious requests; a network packet can query on a routing table to determine the output port in a virtual switch; a web server can send query to a database to retrieve a user's profile. Optimizing such operations can benefit a wide range of workloads.

In addition to numerous software optimizations for query operations, there have been a few proposals to use specialized hardware to accelerate these operations [25, 45, 48, 54, 79, 81]. Almost all of them propose to offload the query operations entirely or partially to an accelerator integrated inside a CPU chip. Compared with PCIe-based devices, these on-chip accelerators considerably reduce the communication overhead, making them appealing for the query operations.

However, there are still substantial limitations with these existing solutions. First, the accelerator should efficiently deal with a wide range of popular applications to justify being integrated into a general-purpose CPU. Most existing solutions either focus on a particular application (*e.g.*, only hash table lookups are accelerated [79, 81]) or require multiple instances to support different data structures [45], all of which lack generality and efficiency. Second, many existing proposals assume a loose integration of the accelerator with the CPU cores, which still has latency concern, given query operations' strict latency sensitivity in many workloads. Third, the hardware complexity and cost of these solutions make them less practical. For example, the queried data structures seldom reside in a contiguous memory address space (*i.e.*, larger than a 4KB page) [8, 26], which necessitates an address translation of some sort and may require a dedicated memory management unit (MMU) in the accelerator for it to be high-performance. This incurs non-trivial on-chip hardware costs (see Sec. VII for detail). These limitations make many of the existing proposals less attractive for a general-purpose platform. It is highly desirable to have a versatile, efficient, and balanced accelerator design with low latency and high throughput.

To this end, we propose QEI, which strives to achieve this fine balance. We first investigate the processing steps of various data query operations on popular data structures and observe that they share very similar inputs/outputs and execution patterns. Based on this observation, we abstract data query operations using five steps with three types of operations: memory access, comparison, and arithmetic. With this abstraction, we can map each data query operation to a distinct configurable finite automaton (CFA) – a finite automaton with fixed transition rules

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00040

but configurable parameters. We design QEI to be capable of executing these CFAs in a way that makes it possible for a single accelerator to process multiple types of data query operations, improving the performance for a wide range of workloads.

QEI consists of three main components: (1) a `Query State Table` to store the state information of in-flight query operations, (2) a `CFA Execution Engine` that is capable of supporting multiple CFAs for different data structures and can be extended via firmware update to support new data structures, and (3) a `Data Processing Unit` comprising of various processing elements such as `ALUs` and `Comparators`. The query is initialized by a new instruction (with two flavors) and is processed by the appropriate CFA model for state transitions and intermediate data processing via micro-operations. Depending on the flavor, the result is returned to either the core or the designated memory space. Such architecture enables QEI to support a wide range of queries efficiently with shared hardware resources.

Regarding how the accelerator should be integrated into a CPU, prior works generally explore two directions. They either use a distributed design by integrating the accelerator in the core or last-level Cache (LLC) (see Fig. 6a) or a centralized design by placing the dedicated accelerator hardware away from the core tile (connected to the on-chip fabric through a dedicated port) (see Fig. 6b). In this paper, we propose a novel integration scheme that builds on the advantages of these schemes to balance throughput, latency, and design complexity. More specifically, in our integration scheme, the accelerator is tightly coupled to the core while still being able to extract a large amount of memory-level parallelism (MLP) by overlapping many query operations (see Fig. 6c). We place the majority of QEI close to each core's L2 cache and second-level TLB (L2-TLB) for resource sharing and put the `Comparators` into the Caching and Home Agent (CHA) of each LLC slice to avoid moving large amounts of data into private caches. With this scheme, QEI can conveniently leverage L2-TLB for address translation while performing the data-intensive comparisons in a near-data fashion to exploit parallelism in Non-Uniform Cache Access (NUCA) [37, 44] design, as well as to avoid private cache pollution.

We evaluate QEI using the Sniper simulator [11] with five representative cloud data center workloads. The results show that QEI can achieve $\sim 8\times$ speedup on average and as high as $\sim 10\times$ speedup over the baseline software implementation. We also demonstrate how the integration schemes impact the performance delivered by the accelerator.

II. Background and Challenges

A. Data Structures in Cloud Workloads

Cloud workloads in data centers are usually data-intensive [26, 42], and the data is typically organized in various data structures with different performance and memory trade-offs. In this section, we discuss the characteristics of some popular data structures.

Hash Table. A basic hash table is an array where one can store each key-value pair at a dedicated location indexed by hashing the key. Hash tables often appear in network function virtualization (NFV) environments, where network functions run on general-purpose servers instead of specialized hardware boxes [36, 56]. Some examples of such network functions are virtual switch [63], firewall [29], and load balancer [21]. Hash tables are also used for in-memory key-value stores [24] and machine learning workloads [13, 73].

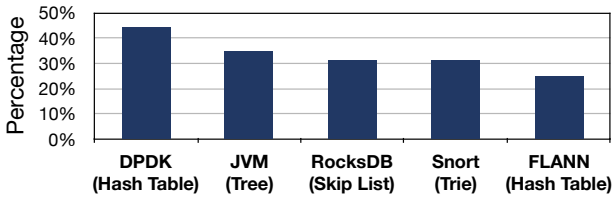


Fig. 1: Percentage of data query operation among total execution time in different workloads (data structures).

Tree.[1] Tree is a hierarchical data structure with each tree node storing the key and data. One of the most popular tree data structures used in the cloud is the object tree managed by runtime languages for garbage collection. For example, the garbage collector in Java Virtual Machine (JVM) maintains the live objects in a tree data structure [1]. A garbage collection event causes a traversal of the object tree to mark and move live objects and recycle dead ones. As most cloud applications are written in managed languages such as Java, garbage collection is a major consumer of CPU cycles [55].

Trie. We distinguish trie from tree because they have distinct implementations and usages. In a basic trie, each child node is indexed by a distinct byte. The node itself does not store a key. Instead, the path to a leaf node implicitly represents a unique key. Trie is commonly used for literal matching or prefix matching. For example, in networking workloads, a routing table applies longest prefix matching (LPM) on IP addresses to decide a route [4, 67]. Intrusion prevention systems (IPSs) use literal matching to decide if a networking request is malicious by matching the request's content with a list of keywords [16].

Linked list. Unlike trees and hash tables, linked list is a data structure that can be more easily extended and maintained during runtime. People usually choose linked list when data updates happen frequently. For faster data query speed, a special type of linked list called skip list [65] is also widely used. Skip list keeps its data sorted and maintains multiple levels of linked list so that the query thread can skip nodes during traversal. One can find skip list usages in database applications such as RocksDB [23].

To better demonstrate the overheads of query operations in cloud workloads, we conduct a profiling study with DPDK, RocksDB, and FLANN on a commodity server with two Intel® Xeon® 8160 CPUs [40] and 64GB DDR4 memory (see Sec. VI for benchmarks details and configuration). Based on our performance profiling with Intel® VTune® [39] and previous works [27, 55], as summarized in Fig. 1, the data query

[1]We do not strictly distinguish tree and graph since they share similar operations from the hardware point of view. For graph-specific issues, such as circle handling, programmers need to properly deal with them in the software.

operations in various workloads take up 23%∼44% of the CPU time. We further use vTune's top-down analysis to investigate the architectural bottleneck of query operations. Workloads can be categorized as being backend bound or frontend bound. We find hash table queries to be backend bound due to the excessive amount of data accesses. For instance, DPDK workload is 7.5% frontend bound and 63.9% backend bound[2]. We observe higher frontend pressure for queries into lined list or tree due to the large number of instructions and data-dependent branches, primarily from pointer chasing. Specifically, the RocksDB workload is 25.9% frontend bound and 9.5% backend bound. Although the out-of-order (OoO) execution of the modern CPU core helps with instruction-level parallelism, we find each query operation can easily generate hundreds of dynamic instructions. Core's ROB and Load-Store Unit can be quickly saturated. These findings motivate us to accelerate query operations by reducing data access overhead (backend bottleneck), dynamic instruction count (frontend/backend bottleneck), and irregular control flow (frontend bottleneck).

B. Challenges of Designing Query Accelerator

Challenge 1: generality. Query operations on different data structures can have distinct implementations in software. Previous works only focus on accelerating a specific operation on a particular data structure [45, 79, 81] or develop specific hardware for each data structure [48], resulting in restricted usage scenarios and poor extensibility.

Challenge 2: latency. We find that previous accelerator studies rarely discuss the communication latency between the CPU core and the accelerator. This is partially attributed to the long execution time of many offloaded computing kernels, which amortizes such latency. For example, one can offload a large portion of the computing time in machine learning workloads to a GPU or a dedicated accelerator [17, 62]. Communication only happens at the kernel initialization and the data retrieval stages, which is amortized by the kernel's long execution time. However, not all use cases can tolerate such latency. For example, as discussed by Kalia *et al*. [41], the average networking response latency can be tripled when operations are offloaded to a GPU. Our targeted query operations are fine-grained and often used in latency-sensitive workloads, including networking and database applications. The jitters and latency to serve each query are critical to the observed quality of service [15, 69, 72].

Challenge 3: design complexity and cost. Adding an accelerator to the CPU die means reduced area and power budget for CPU cores and other components. When the accelerator is not used, it becomes dark silicon, wasting area and leakage power [22]. Hence, it is desirable to minimize the area and power consumption by the accelerator. One example is the MMU, which can substantially increase the hardware cost of an accelerator. Previous works either assume largely consecutive memory space via huge page [35, 54, 79] or dedicated memory management hardware [5, 48]. Using huge page can easily cause fragmentation, and there is no guarantee that huge pages are available on a system that is not freshly booted. Using dedicated hardware, on the other hand, increases the hardware area and power budget significantly. Later on, we will show that an extra TLB can take a significant amount of area in silicon (see Sec. VII).

[2]These numbers show the percentage of unused pipeline slots caused by either frontend or backend issues.

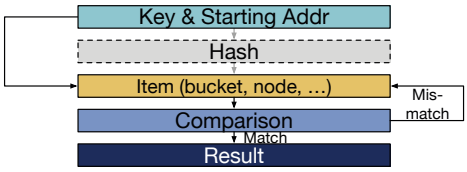


Fig. 2: Abstraction of query operations.

```
struct node{
    void* _key; void* _value;
    struct node *_next;
};
void* query_linkedlist                    Key & Starting Addr
(void* key, struct node *root){
    struct node *current = root;          Item (node)
    while(current != NULL){
        if(!memcmp(current->_key, \       Comparison
        key, KEY_LENGTH)){
            return current->_value;       Result
        }
        current = current->_next;         Item (node)
    }
    return NULL;                          Result
}
```

List 1: Routine of the linked list query and its corresponding steps in the abstraction.

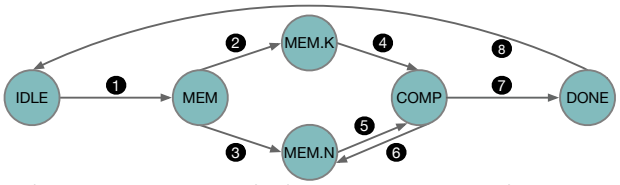
III. Abstracting Query Operations

Data queries, regardless of the exact type of the data structure, have very similar characteristics. In this section, we summarize these similarities and build an abstract model that can fit different data structures and query algorithms (see Fig. 2).

Input/output. Each data query operation requires two inputs and one output. The first input is the `key` to be queried, and the second input is a pointer to the data structure (`starting address`). Both inputs can be passed to the accelerator through reference, *i.e*., via pointers to memory locations. The output (`result`) of the operation is the data being queried, *e.g*., a node in a linked list. In real applications where the result can be large, a pointer to the actual data is used as the `result`.

Execution pattern. Given the inputs, the query operation starts by accessing an initial location in the data structure. For a tree and a linked list, the query begins at the root node. For a hash table, an `offset` is generated by hashing the key, and the query operation starts from `starting address + offset`. For each item in the data structure (either a node in a tree or linked list or an entry in a hash table bucket), the item's key is read out and compared against the queried key. If the keys match, the associated data is returned as the `result` of this query operation. Otherwise, the query operation will iterate to the next item linked by the current one until a match is found or all potential items have been examined.

We demonstrate the execution pattern with a linked list query operation shown in List 1. Other data structures share similar flows for query operations with minor modifications.



1. {inst, , mem.node & mem.key req}
2. {mem.node recved, , }
3. {mem.key recved, , }
4. {mem.key recved, , comparison}
5. {mem.node recved, , comparison}
6. {comparison recved, mismatch, mem.node req}
7. {comparison recved, match, send final result}
8. {final result sent, , }

Fig. 3: CFA of linked list query operations. State transition format: {trigger event, condition(s), action(s)}.

A. CFA Model

Given the common inputs and output, and the formalized execution patterns, we observe that data query operations can be easily represented as CFA. Typically, accelerators use fixed pipelines or functional elements to implement specific algorithms. However, we choose CFA as our model because (1) the steps in data query operations are relatively regular and fixed, which fits the expressiveness of a CFA's state transitions. (2) CFA enables us to decouple the control logic and the execution units in the hardware design. Different query operations can share the same execution units (*e.g.*, ALUs and Comparators) to amortize the hardware cost and maintain high generality. (3) Compared to the basic finite automaton, CFA is more flexible and enables us to process multiple instances of a single data structure with different parameters. This allows us to implement multiple CFAs in the accelerator to support various data structures and query algorithms efficiently.

We continue the linked list example to illustrate how CFA can be applied to a query operation in detail (see Fig. 3). Then we briefly describe how the other data structures differ from the linked list. A query instruction will trigger the idle CFA to issue memory requests for both the queried key and the starting node (❶). Depending on the order in which the results of these two requests are returned, the CFA executes the first comparison via ❷ ❹ or ❸ ❺. If the comparison result is "mismatch", the CFA goes back to "MEM.N" state and issues the memory request for the next node (❻). The operation continues until a "match" is found, or the next node is *NULL*. If a "match" is found, the CFA returns query result (the node's value) and becomes idle again via ❼ ❽.

Querying a binary search tree or a skip list is similar to linked list, with a slight modification to the comparison state (adding ">" and "<" to know the traversal direction for transition ❻). For a trie, the next node is indexed by one or more bytes of the queried key. Within a node, we search an index table (*e.g.*, an array) for a match to traverse to the next node. Between "MEM.N" and "COMP", we can insert a state to search the index table. The CFA transits to "DONE" state when we cannot find a match or the node is the leaf node. For a hash table query, one extra state for hash calculation needs to be inserted before ❶. For ❶, the memory node is now a key bucket indexed by the hash value. Also, ❻ will load the next entry from the same bucket. With these basic states and transitions, the accelerator can even operate on combined data structures such as a hash table of linked lists. To achieve this, we can treat each combined data structure as a unified and unique data structure and assign a unique "subtype" and a dedicated CFA to it.

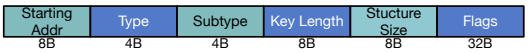


Fig. 4: Format of data structure header (with field size).

B. Software Usage Model

As noted above, the accelerator should contain multiple CFA models to support different data structures and query algorithms with the same hardware components. To initiate a specific query operation, the software needs to communicate the specifics of the data structure and the query to the accelerator. This allows the accelerator to use the appropriate CFA and configuration/parameters for the query. For example, the accelerator needs to know the length of the key for comparison. It also needs to know the type of the data structure to invoke the appropriate CFA for the query. These configuration parameters are uniquely specified for each queried data structure. We call this set of configuration parameters the "metadata". We define a single-cacheline (*i.e.*, 64B) header to store the metadata (see Fig. 4). This header's fields include the pointer to the data structure, type and subtype (*e.g.*, number of entries in a hash table bucket) of the data structure, the length of the key stored, the size of the entire data structure (for static data structures such as hash table), other flags and reserved bits for future extension. The software is responsible for populating the header properly, and the CFA parses the parameters from it before executing a data query operation.

IV. QEI DESIGN

A. QUERY Instructions

To initiate the query operations on QEI and leverage the software abstraction we built in Sec. III, we define an instruction called `QUERY`. This instruction has two flavors – (1) blocking and (2) non-blocking – which target two distinct use cases.

QUERY_B ***reg.key/result*** ***mem.header_addr***

This instruction sends the `header_address` and `key_address` to the accelerator and waits for the `result` to be returned in the same register as `key_address` before it can retire. Note that it does not block succeeding instructions from entering core's pipeline (if slots available).

QUERY_B can be used when there is no independent work available and can be used in small batches, determined by the resource limitations of the accelerator and the core pipeline, to maximize the parallelism. However, once the accelerator resources are filled up, forward progress will be blocked until at least one of the query instruction completes. The OoO core will continue to execute independent instructions until resource limitations are hit due to the incomplete query instructions blocking the head of the OoO window.

QUERY_NB ***impl_reg.header_addr*** ***mem.result_addr*** ***reg.key***

This flavor of the query instruction has an extra operand indicating the address that the accelerator can write the result to.

```
inline static void*
query_b(void* key, void* header){
    void* result = key;
    __qei_query_b(header, result);
    return result;
}
inline static void
query_nb(void** keys, void* header, void** results){
    for(int i = 0; i < BATCH_SIZE; i ++){
        __qei_query_nb(header, keys, results);
        keys ++;
        results ++;
    }
}
inline static void
polling_result(void** results){
    __m512 zmm;
    while(1){
        __qei_snapshot_read(results, zmm);
        if(_mm512_cmpeq_epi64_mask(zmm, 0) != 1){
            break;
        }
    }
}
```

List 2: An example of QEI instructions usage.

This instruction retires from the core as soon as the accelerator accepts the request. After the query operation is done, the accelerator writes the result to the designated `result_addr` provided by the instruction. The non-blocking version does not prevent forward progress, and the program can perform other independent work while the query is being processed by the accelerator, thus maximizing the parallelism without blocking the core resources (*e.g.*, ROB). However, the software is responsible for reading the result from memory and checking the result's completion flags. One way to check for completion is by occasionally polling the result from the output address, but this costs extra cycles. The overhead can be reduced by using wide SIMD *SNAPSHOT_READ* instruction similar to HALO [79]. We do not choose hardware interrupt because it requires the OS to handle the interrupt, which is not cheap.

We show a code snippet that uses the query instructions in buitin format in List 2. As indicated in the snippet, *QUERY_NB* should be used in algorithms where other independent tasks can be time-multiplexed with the query operations. These instructions can be batched to maximize the parallelism, but care must be taken to prevent overflowing the accelerator resources. An overflow will prevent the accelerator from accepting further query requests and will prevent them from retiring, eventually blocking the machine.

Update operations (*e.g.*, insert, delete) are still in software. Also, memory concurrency is handled by the software by using appropriate locks and barriers. Since QEI targets read-intensive cases, the "synchronization" happens infrequently. Besides, the overhead of lock and barrier operations can be significantly reduced by using purpose-built hardware mechanisms [14, 51, 77, 79, 82], which are orthogonal to the design of QEI.

B. QEI *Microarchitecture*

QEI's goal is to efficiently execute the data query operations of different data structures with a shared pattern (see Sec. II). Based on the CFA model we apply in Sec. III, we propose a flexible design that can handle several common data structure queries and can also be extended to support emerging data structures and query algorithms.

Since we want to leverage memory access parallelism, QEI must support multiple in-flight query operations. This can

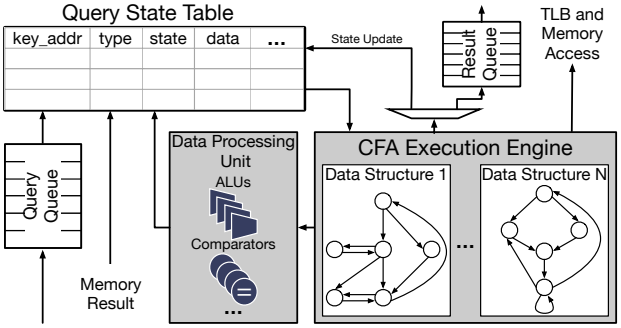


Fig. 5: QEI accelerator microarchitecture.

be done in one of two ways - (1) parallel CFAs by naively replicating all the hardware for the CFA as many times as the number of queries, or (2) pipelined CFAs by only working on one query operation at a time but pipelining multiple queries one after the other. QEI chooses pipelined CFAs but in an OoO fashion. Typically, each query operation requires a series of memory accesses, and the latency for these memory accesses is quite high. During these memory accesses, the CFA for that operation is stuck in the same state until the data comes back and can be processed to determine the next state. Instead of letting a single query monopolize the CFA hardware, in QEI, the current state of the operation is saved in a table, and the hardware is allowed to work on another query whose data might already be ready. In this way, we time multiplex the hardware to process multiple in-flight queries.

We depict QEI microarchitecture in Fig. 5 and describe QEI's three main components.

Query State Table (QST). To hide memory access latency and exploit parallelism, like many other accelerator designs, QEI supports multiple in-flight query operations in parallel. `Query State Table` stores the current state of all the in-flight queries. Specifically, this information includes the `key_address` (8B), `result_address` - valid only for non-blocking queries (8B), `type` - type of the data structures (1B), `state` - the current state in the corresponding CFA (1B), `data` - intermediate data or scratch space (64B), `query_mode` - blocking or non-blocking (1b), and a `ready bit` (1b). When inserting a new outstanding query from `Query Queue`, QST finds the first empty entry and sets the ready bit to 1, and the index of this entry (QST ID) is used for addressing during the state transitions. A completed query releases its QST entry and marks the ready bit to 0. Software is responsible for tracking the availability of the QST slots to make sure that QEI accelerator is not overflown. The intermediate data field is used to stage either a cacheline (*i.e.*, 64B) worth of data from memory or intermediate results from arithmetic/comparison operation. QST acts as a scheduler tableËvery cycle, it selects a ready entry (in a FIFO manner) to be processed by the `CFA Execution Engine` and its state to be updated.

CFA Execution Engine (CEE). `CFA Execution Engine` is responsible for processing the entries in the QST and update their state. CEE contains the state transition rules for CFAs of multiple data query flows for various data structures. The rules

to be applied to an entry depend on the value of the `type` field. The CEE processes the intermediate data and updates the state of an entry accordingly, writing back the updated state to the QST. Each state update can be accompanied by one of several different operations on the intermediate data - (1) memory access (in cacheline granularity, *i.e.*, 64B per access)[3], (2) arithmetic operation, and (3) comparison, which is issued to the appropriate `Data Processing Unit`. The intermediate data is read from the QST at this time and sent to the processing element along with the QST ID of the query. Once the operation finishes, which can take one or several cycles, the new result is written back to the QST, and the entry is marked ready for further processing. Software can achieve a similar time-multiplexing effect by either using software pipelining techniques or helper threads. However, due to the complex software implementations, they cannot compete with the efficiency of hardware CFAs.

The state transition rules for several CFAs for querying common data structures are pre-defined in the CEE. However, the CEE is designed as a microcoded control machine (*i.e.*, configurable), and a firmware update [47, 83], with new state transition rules, can be applied to support emerging data structures and query algorithms. In our design, the number of states is limited by the size of the `current_state` field in the QST and allows for 256 states. This is sufficient for the algorithms we experimented with. However, the field can be made larger if other data structures' queries require more states in their CFAs.

Data Processing Unit (DPU). `Data Processing Unit` consists of multiple processing elements or function units used to perform certain operations on intermediate data. The processing elements include ALUs, comparators, and a hashing unit. For each related state transition in the corresponding CFA, a micro-operation can be issued to a data processing element. QEI 's micro-operations include memory access (read), arithmetic and logic operations, and comparison. The micro-operation sequence in a query is defined by its CFA. For example, querying a hash table may require computing a hash function with an input key to produce an output value for further lookup. To do this, CEE first issues a memory micro-operation to fetch the key and a subsequent micro-operation to the hash unit to generate the hashed value. The hashing unit supports common hash functions. Hash functions not supported by the hashing unit can be decomposed into several simple arithmetic operations, such as shift and bit-wise Boolean operations, and calculated using a series of micro-operations. Comparison is another critical step in data query operations for most, if not all, data structures. The comparator elements in the DPU are capable of conducting bit-wise comparisons ($>$, $<$ or $=$) of 64-bit values each cycle.

C. Life of a Query Operation

A query instruction is fetched like any other instruction by the CPU core and is scheduled by the OoO scheduler to be executed on QEI. Once a query instruction has been issued, depending on whether it is a blocking or non-blocking instruction, the instruction behaves like a `load` or a `store`.

A blocking query operation behaves like a load, occupies space in the Load Queue, and is blocked waiting for data to come back from QEI. Once the query is completed by QEI, the final data is returned to the Load-Store Unit, which wakes up the query instruction to get the data and write it back to the physical register file before it completes and is marked as such in the ROB. From the point of view of the core pipeline, this is very similar to a long-latency load.

A non-blocking query operation behaves like a store. However, it does not have strict ordering requirements as the software guarantees that ordering violations do not have any functional impact on it. The instruction goes through the Load-Store Unit as a store and is immediately completed upon execution once it is done communicating the required information to QEI. Upon completion of the request by QEI, the result is written back to a memory location.

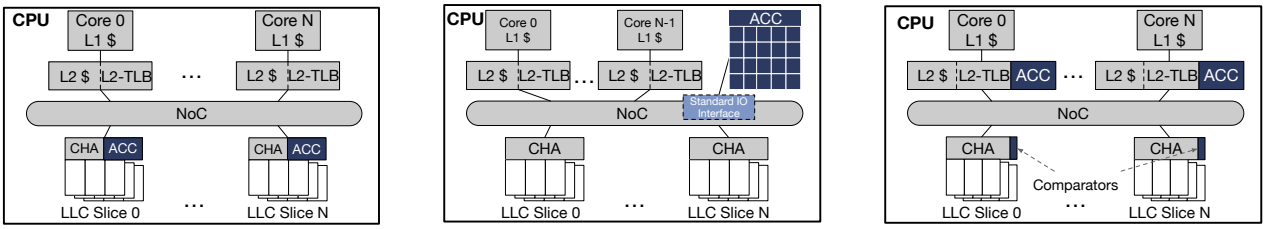
Once a query has been issued to QEI, it writes the pointer to the key to the `key_address` field and the pointer to the header in the `data` field of an empty QST entry. The pointer to the header is only needed for fetching the metadata and is discarded once processing begins. The state is set to "START", and the ready bit is set. When the CEE begins processing this entry, the first thing it does is issue a read for the metadata and update the state. Once the metadata is ready to be parsed, required fields are extracted from it and written to corresponding QST fields. From here on, the type-specific CFA kicks in. As the CFA goes through various states, it issues micro-operations to the DPU for fetching and processing intermediate data and eventually completes the query. After that, the state of this data query request is changed to "DONE". At this point, the `result` is returned to the core or designated address via the `Result Queue`. The corresponding entry is released by setting it to "IDLE" and notifying the core.

D. Exceptions and Interrupts

When processing a query operation, QEI accesses memory and performs arithmetic operations on intermediate data. This can result in several types of faults and exceptions, *e.g.*, accessing memory that does not belong to the current thread. Once an exception occurs, QEI transitions the query to the "EXCEPTION" state. For a blocking query, the exception information is sent to the core through the `Result Queue`. For a non-blocking query, the error code is written to the result memory address so that after polling the designated memory address, the software can find the exception. The entry is then released. Every query that causes an exception eventually gets reported to the core through one of the above two mechanisms, and no special action is required by the application or the core. The software, however, must handle the exception if it needs to recover gracefully.

Interrupts, including timer interrupts for context switches, are handled by flushing QEI. No special action is required for blocking queries since QEI only holds state for incomplete queries, which will be flushed from the core on an interrupt. If

[3] QEI is backed by the regular cacheable memory in the coherence domain, following the write-back policy. We assume weak ordering for both the memory accesses issued by QEI and the query requests sent to QEI by the core. If strong ordering is required for software's update operations, lock/mfence instructions should be applied manually in the software.



(a) CHA-based. **(b)** Device-based. **(c)** Core-integrated.

Fig. 6: Different integration schemes for QEI.

TABLE I: Comparison of different integration schemes.

Scheme	Accelerator-Core Latency (cycle)	Accelerator-Data Latency (cycle)	Hardware Cost	Mem Mgmt HW	NoC Hot Spot	Private $ Pollution	Scalability
CHA-based	40∼60	10∼50	Low	Dedicated/ Shared	No	No	Good
Device-based	100∼500	100∼500	Medium/High	Dedicated	Yes	No	Medium
Core-integrated	10∼25	20∼40	Low	Shared	No	No	Good

QST holds any non-blocking queries, an `abort code` is written to its result memory location so that the application can restart the queries after interrupt handling. This means the flush is not instantaneous and can take a few cycles, depending on the number of non-blocking queries in the QST. To reduce the latency, the writes are done using non-temporal stores, and stores to the same cacheline can coalesce. Technically QEI only needs to wait until all the addresses have been translated for the stores, after which the stores are guaranteed to complete and are handled by the memory element in the DPU. The core cannot start executing interrupt handler code until QEI has been successfully flushed. However, it can start fetching interrupt handler instructions to parallelize some of the work.

V. Integrating QEI into a CPU

As evident from Sec. II-B, the design of QEI needs to strike a delicate balance between generality, latency, design complexity, and cost. Having described how QEI can adapt to various data query operations, we now consider integrating QEI accelerator into the CPU chip, *i.e.*, where to physically place QEI and how to interface it with the other components of the CPU in order to achieve its goal. In this section, we discuss several possibilities (demonstrated in Fig. 6) and their respective advantages and disadvantages (summarized in Tab. I). More specifically, we try to answer the following question: how should the accelerator be integrated into the CPU?

We first consider an intuitive scheme [45, 54, 80], where a dedicated functional unit is fully embedded inside the general-purpose core. The accelerator and the core share the MMU and the private caches. Although this design has very low initiation latency, it does not scale well: First, the accelerator resources are private to the core and can only execute queries from this core. Second, the accelerator competes for the data TLB and the private caches and can negatively interfere with the core. Third, the data access latency is not much better than if the query is executed using general-purpose instructions. Due to these limitations, we restrict ourselves to a qualitative evaluation of this design.

CHA-based Schemes. To solve the scalability issue, the most recent work, HALO [79], proposed a CHA-based scheme, depicted in Fig. 6a. CHA is the LLC controller that is attached to each LLC slice. The CHA-based scheme exploits parallelism by placing accelerators in each CHA and distributing the query requests to these accelerators based on a hash function specific to the NUCA architecture of the particular CPU. This scheme has two major advantages. First, computations and key comparisons are moved closer to the LLC. In many cloud workloads, the data is larger than the core's private caches (*i.e.*, 1MB L2 cache for Intel® Xeon® Skylake CPU [78]). Thus, moving computation near LLC can effectively reduce the memory access latency and private cache pollution [79]. Second, the accelerators are naturally distributed with LLC slices, which maximizes the parallelism of the query operations. However, CHA itself does not provide address translation capability. HALO's usage scenario assumes that the full data structure can reside within one contiguous page (*i.e.*, huge page). This assumption does not always hold for many cloud workloads [8, 26], especially when the targeted data structure is a dynamic one, such as linked list. To accommodate querying different kinds of data structures, address translation capability becomes a necessity. With the CHA-based scheme, one may add MMU or TLB into the CHA or use the core's MMU or IOMMU for address translation. Adding MMU to a CHA introduces non-negligible hardware cost and leads to TLB coherence and manageability problems. Using core's MMU or IOMMU adds extra round-trip latency to each memory access and eats into the performance benefits of the accelerator. We show the penalty of these design choices in Sec. VII.

Device-based Schemes. Another popular way to integrate hardware accelerators into a CPU is as a device attached to a high-speed on-chip or off-chip bus (see Fig. 6b). Intel®'s CXL [18] and IBM's OpenCAPI [25, 59] are recent efforts to provide such capability in their proprietary CPUs. This scheme has the least impact on design as it does not change the design of the core or the on-chip network. Different accelerators attached to the same standard device interface can share the memory management hardware (*e.g.*, IOMMU for IO devices) and other interfacing logic. In addition, such standard interfaces

can easily support third-party IP blocks. However, a major issue with this scheme is the high request and response latency and limited bandwidth compared to the more integrated designs. The round-trip latency to an OpenCAPI device can be as high as 300 ns [10], preventing many latency-sensitive workloads from benefiting from such design. When multiple devices connected to openCAPI are in active use, this latency can be even worse. Since the accelerator is not distributed across the chip, it also creates a single hotspot on the chip, introducing fabric congestion and thermal issues. In our experiments, we observe that each QEI accelerator can saturate as much as 8% of the mesh NoC bandwidth. For a modern CPU with 20 cores or more, if the accelerator is not fully distributed across the chip, it is easy to cause a hotspot. Note that even if the NoC bandwidth is not saturated, a higher bandwidth utilization caused by the hotspot will lead to much longer latency [34].

Alternatively, the accelerator can also be directly connected to the NoC as a heterogeneous core. DASX [48] is one example. This scheme can keep the access latency of core-accelerator and data-accelerator lower than through the standard device interface. However, this design requires the accelerator to behave like a regular core, which significantly complicates the hardware design. For example, the accelerator has to handle address translation and coherence messages properly on its own, making the hardware design non-trivial [46]. The accelerator also occupies one NoC stop, which could have been used by a general-purpose core.

A. QEI *Core-Integrated Scheme*

We propose a novel integration scheme for QEI, called "Core-integrated" in the paper depicted in Fig. 6c, that has the advantages of being close to the core and yet highly scalable.

The main components of QEI are integrated alongside the core's L2 cache. It shares the memory access hardware units with the L2 cache and uses the L2-TLB, which is typically close to the L2 cache, for address translation. We leverage existing hardware mechanisms in the core without significant changes in their microarchitecture. Since QEI uses the L2 resources, it does not contend for L1 cache and L1 TLB, reducing negative interference. We place the memory-intensive operation – the key comparison – in CHAs to maximize the parallelism and efficiency through near-data computing. We add `Comparators` in each of the CHA across the chip to maximize the throughput. These CHA-based comparators access the data directly from the LLC, preventing private cache pollution and reducing round-trip latency. Depending on the type of query operation, the key can sometimes be huge. Hence, leaving them in the LLC and doing the comparison in-place can significantly increase throughput and reduce latency.

The interface between the `CEE` and `Comparators` is extended to traverse the on-chip network using remote micro-operations. The `CEE` calculates the address of the memory location to be compared to the key, translates the address using the L2-TLB, and issues a remote operation to the appropriate `Comparator` (based on the NUCA hash function) to perform the key comparison and return the result. Note that certain query algorithms might not use the remote comparison feature, and a local comparison in QEI might be sufficient. QEI does fetch cachelines to obtain the next set of pointers in some cases (*e.g.*, linked list query), and a small key comparison can be done in one of the DPU if the key is part of the fetched cacheline.

TABLE II: Simulated CPU model configuration.

Item	Configuration
Cores	24 OoO cores, 2.5GHz
Caches	8-way 32KB L1D/L1I, 16-way 1MB L2, 11-way 33MB shared LLC (split to 24 slices)
LQ/SQ/ROB Entries	72/56/224
Memory Controllers	6 DDR4-2666 channels, 19.2GB/s per channel, 4 8-chip DIMMs per channel
QEI Accelerator	five ALUs per DPU two comparators per CHA for CHA-based/Core-integrated ten comparators per DPU for Device-based
NoC	Mesh
Process	22nm

VI. METHODOLOGY

A. *Simulator*

We implement QEI in Sniper [11], a multi-core x86 simulator. We configure the simulator to model a modern Intel® Skylake-SP server CPU [40] (see Tab. II). We simulate QEI in five different integration methods, listed below, and compare the results.

- **CHA-TLB.** This scheme is similar to HALO described in [79]. It integrates the accelerator inside each CHA with a dedicated 1024-entry TLB for address translation.
- **CHA-noTLB.** Similar to the first scheme. But this scheme completely leverages the core's MMU for address translation.
- **Device-direct.** This scheme attaches the accelerator directly to the NoC as a special core [48].
- **Device-indirect.** It simulates a dedicated accelerator which is connected to the NoC via a standard device interface.
- **Core-integrated.** The new QEI scheme proposed in this paper. The `QST`, `CEE`, and some of the `DPU` of the accelerator are placed in the core, close to L2-cache/TLB, while the `comparators` are distributed in the CHAs.

In the Core-integrated, CHA-TLB, and CHA-noTLB schemes, we configure the accelerator to support ten in-flight query operations (*i.e.*, each QST has ten entries), which can keep a decent balance between performance and cost (*i.e.*, $50\% \sim 90\%$ occupancy). For fairness, in Device-Direct and Indirect schemes, we configure the accelerator to support 10×24 (number of cores) in-flight operations.

We use McPAT [50] and CACTI [6] for power and area evaluation in an incremental way. That is, we first configure the CPU in Tab. II and get the baseline power/area. We then add components of the QEI accelerator into the configuration. For components like ALUs and TLBs, we used the default models in the tools. We also change the connection-related configuration, *e.g.*, TLB port counts. We subtract the baseline value from the value with the QEI accelerator, thus get the final value of QEI itself.

B. *Benchmarks*

DPDK. DPDK [38] is a popular networking application development library for kernel-bypass network functions. We

use the optimized cuckoo hash table library [19] from DPDK to setup an L3 Forwarding Information Table (FIB) and evaluate its performance. During the evaluation, we configure the hash table to contain various numbers of keys, 16-Bytes in length, to simulate a regular TCP/IP packet header. We also connect multiple hash tables to implement tuple space search algorithm [74] to show the parallelism of QEI.

JVM. Due to the limitations of the simulator, we are not able to directly simulate JVM. Hence we extract OpenJDK [60]'s serial Mark-and-Sweep garbage collection functionality and set up an independent benchmark for the garbage collection process. We dump a real object tree from running Derby [76], a relational database, in SPECjvm2008 [75] and use it as the input to our benchmark.

RocksDB. RocksDB [23] is a persistent key-value store using Log-Structured Merge-Tree algorithm [61]. Since QEI targets in-memory and in-cache acceleration, we focus on the in-memory *memtables* of RocksDB rather than querying the data on the disk. The *memtable* of RocksDB is a skip list. We first insert 10k items into the database and then do random queries. We use *db_bench*, the standard performance testing tool for RocksDB, to test with 100B key size and 900B value size for each data item.

Snort. Snort [68] is a popular network IPS. It uses Aho-Corasick (AC) algorithm [2] for literal matching to detect potential malicious packets. We follow an efficient open-source implementation [33] to show the speedup of QEI for querying the trie data structure. The dictionary contains around 40K keywords, and we query a 1KB string of characters.

FLANN. FLANN [57] is a library that implements similarity search algorithms widely used in search engines, *e.g.*, searching similar images from a large image database. We run the Locality Sensitive Hashing (LSH) algorithm, which queries a series of hash tables. We use the 100K-item dataset and default parameters for LSH, which are 12 hash tables with 20B key size.

We identify the query-related snippets in each benchmark as Region-of-Interest (ROI), rewrite these snippets with QEI instructions, run the entire benchmark, and report the performance improvement of such ROIs with QEI. All benchmarks (including baselines) are complied by GCC 5.4 with "O3" optimization and are evaluated in single-thread mode, mostly with default parameters. For each benchmark, we generate queries as quickly and densely as possible and feed them to the benchmarks. This stresses QEI to show peak performance.

VII. Evaluation

A. Query Operation Speedup

We show the data query operation speedup of all benchmarks with different schemes in Fig. 7. The overall trend is, the Core-integrated scheme and the CHA-based schemes have comparable performance over all benchmarks, while the Device-based schemes sometimes have a significant performance gap compared to the other schemes.

Not surprisingly, the CHA-TLB scheme achieves the best performance in all benchmarks (*i.e.*, up to 12.7× speedup).

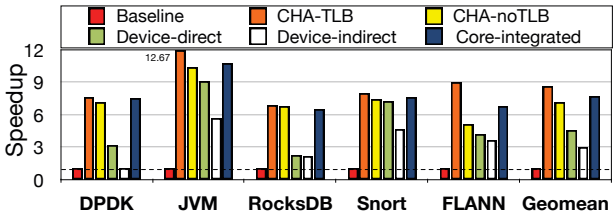


Fig. 7: Speedup of lookup operations in different workloads with different schemes.

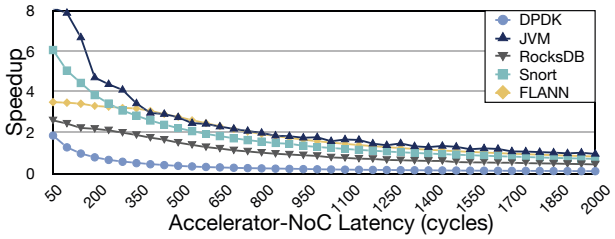


Fig. 8: Speedup of Device-indirect scheme against different NoC-accelerator latency.

The performance gain mainly comes from two aspects. First, data queries are fully distributed to all the accelerators on the CHAs. This is the fully scalable model, as we discussed in previous sections. Second, thanks to the dedicated TLBs, the accelerator can perform address translation locally, without extra round-trips to the core's MMU when TLB-hit. Because of the relatively large TLB size (same as the L2-TLB size), there are few TLB misses in our tests. On the other hand, the CHA-noTLB scheme performs worse than the CHA-TLB because of the extra latency to the core's MMU. However, the performance gap between the two CHA-based schemes is 0.5%∼17.9%, not as much as we initially expected. This is because the parallelism of CHA-based schemes hides such latency to some extent.

The Core-integrated scheme can achieve at most 10.4× speedup compared to software baselines. As expected, this scheme enjoys both parallelism and near-data advantages. Keys are compared across multiple CHAs and stay in the LLC. Meanwhile, it leverages the core's L2-TLB for convenient address translation. This reduces the design complexity and cost and eliminates the round-trip latency for address translation in the CHA-noTLB scheme. Since QEI's `Query State Context Table` is not scaled out of the core, this slightly constrains its parallelism. Thus, the integrated scheme has a 0.9%∼15.0% performance gap compared to the fastest CHA-TLB scheme. However, such a small gap does not defeat the Core-integrated scheme's prominent advantages regarding design complexity and cost (we will show this later), which renders it a more practical solution in real-world CPUs over CHA-based ones.

Regarding the two Device-based schemes, the performance is worse than the other schemes. This is mainly due to the long access latency involved in these two schemes, which counteracts the benefit of processing multiple in-flight query operations in parallel. For the Device-direct scheme, although it accesses the cache with the latency similar to a regular

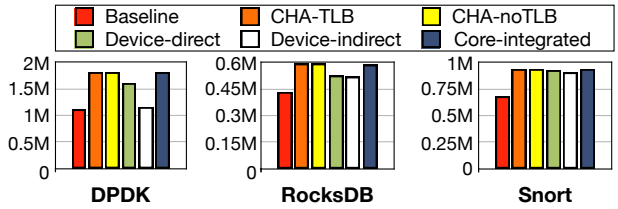


Fig. 9: End-to-end query/packet per second improvement.

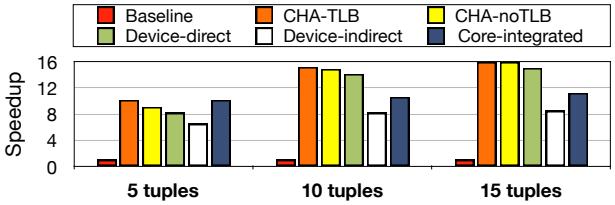


Fig. 10: Speedup of query operations in tuple space search with different schemes.

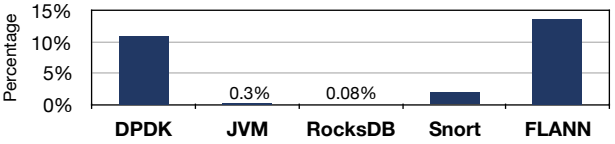


Fig. 11: Core's executed instructions with QEI compared to the software baseline (in percentage).

core, the core-accelerator communication overhead is still significant. For the Device-indirect scheme, the cost is even higher since each data access from the accelerator leads to a costly round trip through the device interface. To better understand the impact of the access latency, we also conduct a latency sensitivity study for the Device-indirect scheme and demonstrate it in Fig. 8. Here we sweep the accelerator's data access latency, which reflects the overhead of the device interface (including the protocol translation and coherence handling), from 50 cycles to 2000 cycles. We observe a non-trivial performance drop of all workloads when we increase the communication latency. Although the industry keeps improving the throughput and bandwidth of standard interfaces such as OpenCAPI, recent data still shows a much more significant round-trip time of 300ns [10] comparing to CHA-based or core-integrated model. One way to improve throughput under outstanding latency overhead is to process queries in batch to hide the latency. Although this approach effectively improves throughput [31, 41], it can also lead to much worse average latency and tail latency, as investigated in [5]. This is not desirable for our targeted latency-sensitive workloads.

Besides the different integration schemes, the characteristics of the workloads also affect the efficiency of the accelerator. First, the degree of parallelism that QEI can achieve depends on the "density" of the query operations in the workloads. Take RocksDB as an example. The code size of its "seek" loop, where one query operation is conducted, is relatively large. That is, RocksDB executes many other operations (*e.g.*, key's pre-processing, *memcpy*, and thread management) besides looking up the data structure when process each request. Hence, the core's ROB is filled up pretty quickly since the block version of the query instruction is not retired. In other words, the performance improvement is bounded by the core rather than the accelerator. The core's resource limits the parallelism we can achieve by QEI. Other benchmarks, such as JVM, have a relatively higher query density, enabling the core to issue as many query requests as possible before ROB is filled up. Thus, the effect of parallelism is more prominent.

Data structures themselves affect the accelerating performance as well. For each query operation of the hash table (*e.g.*, DPDK), the number of memory accesses (namely, header, key, bucket, and key-value pair) is relatively small and fixed compared to other data structures such as skip list. As a result, the processing time of each query is relatively short. In this case, the latency for the core to communicate with the accelerator becomes more prominent. Hence, the Device-based schemes' performance fails to compete with other schemes with much shorter communication latency. For comparison, other data structures such as the tree in JVM and the trie in Snort have much more memory accesses for a single query operation (*e.g.*, 39.9 on average in our JVM benchmark). Moreover, the core to accelerator latency is amortized by the relatively long processing time. Consequently, the performance of the Device-based scheme is closer to that of other schemes. This proves that the Device-based scheme is more suitable for larger kernels that can run for a relatively long time.

For the full application (*i.e.*, not a library or routine of the program), we also demonstrate the end-to-end query-per-second improvement in Fig. 9. It shows that QEI improves end-to-end throughput by 36.2%∼66.7%. Meanwhile, QEI's Core-integrated integration scheme's performance gain is at the same level as the CHA-based schemes'. Note that, since query operations are ubiquitous, just like other accelerators for operations like Malloc [43], and garbage collection [55], even if the end-to-end performance improvement is not amazingly high, it still helps save a huge number of CPU cycles and thus improve the efficiency and throughput of the data center.

B. Non-blocking Query Evaluation

We further evaluate the performance benefit of the *QUERY_NB* instruction. As discussed, some applications limit the parallelism they can benefit from QEI schemes because of their own characteristics. Thus, we evaluate the non-blocking version of query instruction with one representative workload, which demonstrates the ideal use cases of the instruction.

Fig. 10 demonstrates the tuple space search results with 5, 10, and 15 tuples based on DPDK's hash library. Tuple space search can be parallelized naturally since the data query to each hash table is independent. For each query, a series of hash tables can be queried concurrently. Usage of the non-blocking query instruction can maximize such parallelism. In this test, the software polls the results every 32 keys. With the non-blocking instruction, it effectively sends $32 \times (tuple\_count)$ requests in parallel to the accelerator. The results show that as the number of tuples

TABLE III: Area and static power results of QEI.

Configuration	Area/mm^2	Static Power/mW
QEI-10	0.1752	10.8984
QEI-10+TLB	0.5730	30.9049
QEI-240	1.0901	20.8764

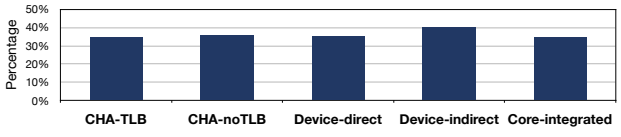


Fig. 12: QEI's average dynamic power consumption compared to the software baseline (in percentage).

increases, the speedup also increases due to the increasing parallelism. We also notice that the performance of the Device-based schemes becomes much better than using the blocking instruction. This is because the performance degradation caused by the long access latency of both core to accelerator and accelerator to data is amortized by executing many in-flight operations. This shows that to compete with our proposed Core-integrated scheme and CHA-based scheme, the application has to generate hundreds of requests simultaneously to fully utilize the parallelism of the accelerator. As mentioned in Sec. VI, the QEI of Core-integrated scheme only processes ten in-flight queries concurrently in the integrated CFA. This limited the parallelism even though the comparators can still process key comparisons in parallel. Still, the Core-integrated scheme has a significant latency advantage, as we mentioned previously, which makes this scheme competitive when the tuple count is smaller.

C. Instruction Count Reduction

We demonstrate the results of the number of dynamic instructions executed by the core in ROIs in Fig. 11. As expected, with QEI, a significant amount of dynamic instructions in the ROIs can be eliminated. As mentioned in Sec. II-A and [42], the performance of cloud workloads is frequently bounded by core's frontend. Thus, reducing the dynamic instruction count can reduce the frontend pressure significantly. This, in turn, improves the efficiency of the whole application.

D. Area and Power Results

For the area and static power comparison, we compare three configurations. They are (1) QEI-10, which can handle ten in-flight queries simultaneously to represent the CHA and Core-integrated scheme, (2) QEI-10+TLB to handle ten in-flight queries simultaneously plus a dedicated TLB to represent CHA-TLB schemes, and (3) QEI-240 that can handle 240 in-flight queries to represent the two Device-based schemes. It is worth noting that for the CHA-based and Core-integrated schemes, the data is a single accelerator's area and power, while for the Device-based schemes, it is the total area and power of the centralized accelerator.

We show the area and static power results of these three configurations in Tab. III. In terms of area cost, the QEI-10 configuration only occupies less than 0.2mm^2 without TLB and ∼0.57mm^2 with TLB. The extra TLB incurs significant overhead here, which shows that although CHA-TLB achieves better performance, the hardware area budget limits its practicality. On the other hand, the larger device model (QEI-240), which enjoys a more relaxed design budget, takes up ∼1mm^2. Considering the size of a typical modern CPU core tile can be around 18mm^2 [78], the total area overhead is negligible. Similarly, the static power consumption of QEI is also small compared with the total thermal design power of a CPU chip, which can easily exceed 100W. Besides the raw power and area comparison, there are some other considerations of design trade-offs. For example, the Device-direct accelerator occupies one NoC stop, which can have been used by a regular core tile. With the Device-direct scheme, we need to remove one core from the CPU chip. This is the hidden cost that can not be shown in the area and power comparison. Other design complexities, such as the logic to make the accelerator behave like a core (*i.e.*, to answer coherence message properly, to manage address mapping), are not easy to be evaluated but are essential factors for accelerator design.

We show the normalized average dynamic power consumption per query in Fig. 12. From the results, the accelerators can reduce more than 60% dynamic power overhead compared to the software baseline. This power reduction comes from both the reduced frontend overhead and private cache accesses, which take up a considerable portion of the whole core activity.

With such power and area efficiency, and considering that QEI is integrated inside the CPU chip, which does not require extra cost for device purchase and maintenance, QEI can largely reduce the server's operational cost.

VIII. Related Work

On-chip accelerator for fine-grained operations is not a brand-new concept. While prior works [3, 32, 43, 52, 55, 64, 70, 71] focus on specific operations/applications, QEI is more generic for diverse data structures. We achieve this by abstracting the data query operations and map them to the accelerator's CFA. Minnow [80] also claims flexibility/programmability, but it does not clearly demonstrate the mapping between its model and software algorithms/routines, and thus its generality.

The most relevant works to ours are [35, 45, 48, 54, 79, 80], which also do data query/lookup/analytic accelerations. Each integration scheme has pros and cons, as we discussed in Sec. V. QEI is different from these works regarding both hardware design and integration scheme. No existing accelerator takes such a hybrid scheme and balances every aspect of a design.

Fully integrated designs [45, 54, 80] tightly couple the accelerator with the CPU core. Although the latency is minimum, they do not address the scalability and private cache pollution issues. Dedicated accelerator designs such as DASX [48] have relatively long latency comparing to core integrated design. Thus it is best for applications that are not latency-sensitive and can batch process many requests each time. The design complexity is also another concern since the accelerator needs extra logic to behave like a heterogeneous core. It also occupies an NoC stop, which could have been used by an additional general-purpose core. Other accelerators connected to standard device interfaces will have even longer

access latency, which leads to sub-optimal performance. Both Device-based schemes can create hotspots on the CPU chip and congestion in the NoC, which further increases the latency.

A recently proposed data query accelerator is HALO [79]. HALO targets on a specific data structure, hash table. It places accelerators inside each LLC slice to get the benefit of near-data computing and parallelism. Some near-memory solutions [20, 30, 35, 53] have also been proposed for data query/analytic operations. These approaches are limited to particular workloads, and the accelerators have to be able to do address translation either by hardware or software mechanisms. For hardware solutions, they need to have their own MMU or go through the core's MMU or IOMMU. This increases hardware cost and complexity or degrades the performance because of the extra round-trip latency. For software solutions, they require the application to guarantee that the entire data structure can reside in the same page, or a new memory mapping method has to be introduced, which is not easy for many existing workloads and OSs. Compared to these solutions, QEI keeps a decent balance among performance, design complexity, cost, and feasibility. In other words, QEI's generality and practicality distinguish it from other similar works.

IX. Conclusion

We propose QEI, a generic, integrated, and efficient accelerator design for speeding up fine-grained query operations in a diverse set of data structures for cloud infrastructures and applications. The generality of QEI comes from abstracting the various data query operations. The CFA model, which we map the abstraction to, guarantees efficient execution by simple hardware. We then propose a novel scheme for integrating QEI and evaluate it against other schemes. Our results with five representative cloud workloads show that QEI can achieve $6.5\times \sim 11.2\times$ performance improvement in various scenarios at low hardware cost and complexity.

Acknowledgement

We would like to thank David Koufaty, Ryan Carlson, Andrew Herdrich, Ravishankar Iyer, Rajesh Sankaran, as well as the anonymous reviewers for their insightful and helpful feedback. This research is supported by National Science Foundation (funding No. CNS-1705047) and Intel Corporation's Academic research funding.

References

[1] O. Agesen, D. Detlefs, and J. E. Moss, "Garbage collection and local variable type-precision and liveness in Java virtual machines," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, Jun. 1998.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, 1975.

[3] K. Angstadt, A. Subramaniyan, E. Sadredini, R. Rahimi, K. Skadron, W. Weimer, and R. Das, "ASPEN: A scalable in-SRAM architecture for pushdown automata," in *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, Fukuoka, Japan, Dec. 2018.

[4] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*, London, UK, Aug. 2015.

[5] N. Asmussen, M. Roitzsch, and H. Härtig, "M3x: Autonomous accelerators via context-enabled fast-path communication," in *Proceedings of 2019 USENIX Annual Technical Conference (ATC'19)*, Renton, WA, Jul. 2019.

[6] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, 2017.

[7] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, 2017.

[8] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th IEEE/ACM International Symposium on Computer Architecture (ISCA'13)*, Tel-Aviv, Israel, Jun. 2013.

[9] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, Feb. 2012.

[10] A. Cantle and M. Byers, "Accelerating flash memory with the high performance, low latency, OpenCAPI interface," https://www.opencompute.org/files/Accelerating-Flash-Memory-with-the-High-Performance-Low-Latency-OpenCAPI-Interface-Final-3-21-19.pdf, 2018.

[11] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, 2014.

[12] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, Oct. 2016.

[13] B. Chen, T. Medini, J. Farwell, S. Gobriel, C. Tai, and A. Shrivastava, "SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," in *Proceedings of the 3rd Conference on Machine Learning ans Systems (MLSys'20)*, Austin, TX, Mar. 2020.

[14] J. Choe, A. Huang, T. Moreshet, M. Herlihy, and R. I. Bahar, "Concurrent data structures with near-data-processing: An architecture-aware implementation," in *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'19)*, Phoenix, AZ, Jun. 2019.

[15] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "A hybrid edge-cloud architecture for reducing on-demand gaming latency," *Multimedia Systems*, vol. 20, no. 5, 2014.

[16] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of Snort," in *Proceeding of DARPA Information Survivability Conference and Exposition II (DISCEX'01)*, Anaheim, CA, Aug. 2001.

[17] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, London, UK, Apr. 2016.

[18] CXL Consortium, "Compute Express Link (CXL)," https://www.computeexpresslink.org, accessed in 2020.

[19] DPDK, "DPDK programmer's guide: Hash library," http://doc.dpdk.org/guides/prog_guide/hash_lib.html, accessed in 2020.

[20] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel,

B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian data engine," in *Proceedings of the 44th IEEE/ACM International Symposium on Computer Architecture (ISCA'17)*, Toronto, Canada, Jun. 2017.

[21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, Santa Clara, CA, Mar. 2016.

[22] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th IEEE/ACM International Symposium on Computer Architecture (ISCA'11)*, San Jose, CA, Jun. 2011.

[23] Facebook, "RocksDB: A persistent key-value store for fast storage environments," https://rocksdb.org, accessed in 2020.

[24] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*, Lombard, IL, Apr. 2013.

[25] J. Fang, T. Y. Mulder, K. Huang, Y. Qiao, X. Zeng, H. P. Hofstee, J. Lee, and J. Hidders, "Adopting OpenCAPI for high bandwidth database accelerators," in *Proceedings of the 3rd International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'17)*, Denver, CO, Nov. 2017.

[26] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, London, UK, Mar. 2012.

[27] M. Fisk and G. Varghese, "Applying fast string matching to intrusion detection," 2001.

[28] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *Proceedings of the 45th IEEE/ACM International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, Jun. 2018.

[29] M. Frantzen, F. Kerschbaum, E. E. Schultz, and S. Fahmy, "A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals," *Computers & Security*, vol. 20, no. 3, 2001.

[30] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT'15)*, San Francisco, CA, Oct. 2015.

[31] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, Apr. 2017.

[32] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "HARE: Hardware accelerator for regular expressions," in *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, Oct. 2016.

[33] hankcs, "AC algorithm implementation," https://github.com/hankcs/AhoCorasickDoubleArrayTrie, accessed in 2020.

[34] High Speed Networking Lab, "Congestion control for network on chip," http://engineering.nyu.edu/highspeed/research/past-projects/congestion-control-network-chip, accessed in 2020.

[35] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD'16)*, Scottsdale, AZ, Oct. 2016.

[36] Y. Hu and T. Li, "Towards efficient server architecture for virtualized network function deployment: Implications and implementations," in *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, Oct. 2016.

[37] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA substrate for flexible CMP cache sharing," in *Proceedings of the 19th International Conference on Supercomputing (SC'05)*, Cambridge, MA, Jun. 2005.

[38] Intel Corporation, "Data Plane Development Kit (DPDK)," https://www.dpdk.org, accessed in 2020.

[39] Intel Corporation, "Intel® VTune™ Performance Ayalyzer," https://software.intel.com/en-us/intel-vtune-amplifier-xe, accessed in 2020.

[40] Intel Corporation, "Intel® Xeon® Platinum 8160 Processor," https://ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2_10-GHz, accessed in 2020.

[41] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Okaland, CA, May 2015.

[42] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd IEEE/ACM International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, Jun. 2015.

[43] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating memory allocation," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, Xi'an, China, Apr. 2017.

[44] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, San Jose, CA, Oct. 2002.

[45] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, Dec. 2013.

[46] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the complexity of hardware cache coherence and some implications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, 2014.

[47] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *Proceedings of 26th USENIX Security Symposium (USENIX Security'17)*, Vancouver, Canada, Aug. 2017.

[48] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan, "DASX: Hardware accelerator for software data structures," in *Proceedings of the 29th ACM International Conference on Supercomputing (ISC'15)*, Newport Beach, CA, Jun. 2015.

[49] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-direct: High-performance in-memory key-value store with programmable NIC," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[50] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd IEEE/ACM International Symposium*

on Microarchitecture (MICRO'09), New York City, NY, Dec. 2009.

[51] C.-K. Liang and M. Prvulovic, "MiSAR: Minimalistic synchronization accelerator with resource overflow management," in *Proceedings of the 42nd IEEE/ACM International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, Jun. 2015.

[52] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, Fukuoka, Japan, Oct. 2018.

[53] S. Lloyd and M. Gokhale, "Near memory key/value lookup acceleration," in *Proceedings of the 3rd International Symposium on Memory Systems (MEMSYS'17)*, Alexandria, VA, Oct. 2017.

[54] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Virtual Event, Mar. 2020.

[55] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *Proceedings of the 45th IEEE/ACM International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, Jun. 2018.

[56] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Seattle, WA, Apr. 2014.

[57] M. Muja and D. G. Lowe, "FLANN - Fast library for approximate nearest neighbors," http://www.cs.ubc.ca/research/flann/#publications, accessed in 2020.

[58] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*, Budapest, Hungary, Aug. 2018.

[59] OpenCAPI Consortium, "OpenCAPI," https://opencapi.org, accessed in 2020.

[60] Oracle Corporation, "OpenJDK," https://openjdk.java.net, accessed in 2020.

[61] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996.

[62] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*, Boston, MA, Oct. 2017.

[63] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Okaland, CA, May 2015.

[64] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus Prime: Accelerating data transformation in servers," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Virtual Event, Mar. 2020.

[65] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, 1990.

[66] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st IEEE/ACM International Symposium on Computer Architecture (ISCA'14)*, Minneapolis, MN, Jun. 2014.

[67] V. Ravikumar, R. Mahapatra, and J. Liu, "Modified LC-trie based efficient routing lookup," in *Proceeding of 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, Fort Worth, TX, Jan. 2002.

[68] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th Systems Administration Conference (LISA'99)*, Seattle, WA, Nov. 1999.

[69] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*, Napa, California, May 2011.

[70] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO'19)*, Columbus, OH, Oct. 2019.

[71] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira, "Accelerating business analytics applications," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA'12)*, New Orleans, LA, Mar. 2012.

[72] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," in *Proceedings of the 2010 International Conference on Very Large Data Bases (VLDB'10)*, Singapore, Sep. 2010.

[73] R. Spring and A. Shrivastava, "Scalable and sustainable deep learning via randomized hashing," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'17)*, Halifax, Canada, Aug. 2017.

[74] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proceedings of the 1999 ACM SIGCOMM Conference (SIGCOMM'99)*, Cambridge, MA, Aug. 1999.

[75] Standard Performance Evaluation Corporation, "SPECjvm2008," https://www.spec.org/jvm2008, accessed in 2020.

[76] The Apache DB Project, "Apache derby," https://db.apache.org/derby/, accessed in 2020.

[77] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, "Architectural support for fair reader-writer locking," in *Proceedings of the 2010 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, Atlanta, GA, Dec. 2010.

[78] WikiChip, "Skylake (server) - Microarchitectures - Intel," https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server), accessed in 2020.

[79] Y. Yuan, Y. Wang, R. Wang, and J. Huang, "HALO: Accelerating flow classification for scalable packet processing in NFV," in *Proceedings of the 46th IEEE/ACM International Symposium on Computer Architecture (ISCA'19)*, Pheoeix, AZ, Jun. 2019.

[80] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, Mar. 2018.

[81] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*, Columbus, OH, Oct. 2019.

[82] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures," in *Proceedings of the 34th IEEE/ACM International Symposium on Computer Architecture (ISCA'07)*, Jun. 2007.

[83] V. J. Zimmer and S. H. Robinson, "Methods and systems for microcode patching," United States Patent 8,296,528, Oct. 23, 2012.