IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 June 2023, accepted 25 June 2023, date of publication 5 July 2023, date of current version 2 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3292582

RESEARCH ARTICLE

An Integrated Solution to Improve Performance of In-Memory Data Caching With an Efficient Item Retrieving Mechanism and a Near-Memory Accelerator

MINKWAN KEE, CHIWON HAN, AND GI-HO PARK, (Member, IEEE)
Computer Science and Engineering, Sejong University, Gwangjin-gu, Seoul 05006, Republic of Korea

Corresponding author: Gi-Ho Park (ghpark@sejong.edu)

This work was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) under Grant 2018R1A2B6002534 and Grant 2021R1H1A2013419.

ABSTRACT This paper proposes both software and hardware mechanisms based on the near-memory processing (NMP) accelerator to improve the linked list traversal of the in-memory caching. From a software perspective, we propose a simple but an effective mechanism called ITEM JUMP to reduce the number of traversal on list iteration, and additionally, LSB-first parallel linked list traversal unit, which is an NMP-based hardware accelerator is proposed to improve parallel comparison performance of items. The evaluation result shows LSB-first parallel linked list traversal unit can achieve about 34 times better performance in item comparisons than the case where there is no hardware accelerator, and ITEM JUMP can reduce the number of items retrieved by up to 42%. The proposed NMP-based hardware accelerator also reduces the memory access overhead by 61%–83% compared to a simple parallel linked list traversal unit that simply loads and compares data as fast as possible.

INDEX TERMS Database system, accelerator architectures, memory architecture, in-memory database, linked list traversal acceleration, near memory processing, parallel comparison.

I. INTRODUCTION

Since the emergence of the big data concept, technologies for analyzing and utilizing big data have been rapidly developed. Many services based on big data are also expanding into real-time areas such as traffic analysis, financial and media services. For such a real-time service, a large amount of data must be accessed at high speed, however the memory access speed of the existing database (DB) is slow, which is insufficient to support a real-time service based on big data. An in-memory caching is developed to enable fast memory accesses by caching the database in main memory for this reason. Main memory is mainly composed of DRAM, and because DRAM has a faster memory access speed than HDDs (hard disk drives) and SSDs (solid state drives), data can be retrieved and utilized at high speed through in-memory caching. Various in-memory caching techniques have already been developed [1], [2] and various companies such as Youtube, Twitter, and Facebook [3], [4], [5] are using them. Although in-memory caching using DRAM provides fast data accesses, recent high-performance computing systems have deep memory hierarchy [6], which results in high power consumption and high latency when large key-value data is loaded into the core from DRAM through the multi-level cache layer. In-memory caching using DRAM provides fast data access, but modern high-performance computing systems have a deep memory hierarchy. That causes high power consumption and high latency when loading large key-value data from DRAM to core through multi-level caches. The size of the key-value data handled by in-memory caching is very large, and a lot of data is loaded into upper level memory like cache memory while data retrieving. It can deepen cache memory traffic and degrade system performance by evicting other data that is more likely to be used within the cache.

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

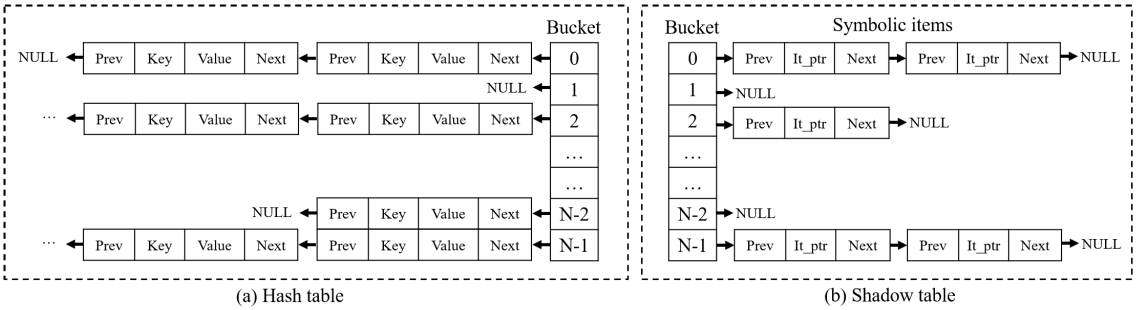


FIGURE 1. A structure of hash table and shadow table.

It is a well-known fact that most in-memory database systems maintain items of DB using linked list and hash function. It facilitates for the insertion/removal of a specific dataset from the chain, while the list iteration operation suffers a lot from a low speed by entailing the pointer-chasing operation for the datasets scattered irregularly within the memory. Various operations of in-memory DB such as GET, SET, INSERT, DELETE are processed accompanied with linked list traversal. In fact, there is a need for processing of linked list traversal to find a specific item in almost all in-memory DB queries, such as data insertion, update, and deletion, as well as data reference-related operations. For example, before the insert query is performed, the data is actually inserted after checking whether the data is already cached in memory db. Therefore, making faster the linked list traversal plays an important role in improving the query processing performance of the in-memory DB.

This paper proposes an integrated solution combining both software and hardware mechanism to improve the performance of a linked-list traversal (LLT) which is kernel operation of in-memory caching. To effectively process LLT, we propose (1) an ITEM JUMP mechanism to reduce average LLT depth as a software approach and (2) near-memory processing (NMP)-based hardware accelerator for parallel LLT as a hardware approach. The NMP is a concept of a computer architecture that processes data-intensive workloads using processing elements adjacent to the memory without transferring data to the host CPU. Further, the NMP has the advantage of being able to swiftly receive data from memory due to its inherent memory adjacent nature. An LLT is a typical data-intensive operation that simply consists of loading items and comparing them; therefore, the NMP-based hardware accelerator is recommended and appropriate for processing LLT operation. The proposed ITEM JUMP mechanism and LSB-first parallel linked list traversal unit (LFP-LLT) is able to achieve about 34 times better performance in item comparisons than the case where there is no hardware accelerator and can reduce the memory access overhead by 50%–80% compared to a simple parallel LLT unit (SP-LLT) that simply loads and compares data as fast as possible.

II. BACKGROUND

This section briefly explains the analysis on the structure of cached items in Memcached, the characteristics of the LLT operation, and the overview of the NMP technology that aims to process operations in the memory layer.

A. MEMCACHED OVERVIEW

Memcached implements a simple and light-weight key-value tuple interface using the least recently used (LRU) eviction policy. It operates on simple data types of key-value pairs stored in memory, similar to NoSQL databases, but it is not persistent like NoSQL. Communication with multiple clients is executed through the network communication, and clients can issue instructions using various command queries (e.g., GET(key), SET(key, value), DELETE(key), and so on.).

Figure 1(a) illustrates the data structure of hash tables used to lookup cache items in Memcached. The hash table data structure is an array of bucket and each of that consists of the multiple cache items. The array size (N) is always a power of 2 to facilitate in finding the correct bucket quickly using 2^{m-1} as a hash mask. The bucket that is having the hash value is quickly determined by the bit-wise AND operation. Each bucket is constructed as single-linked list of cache item and is terminated with a NULL pointer. The cache item data structure consists of the key, data, flags, and pointers. Cache items are sorted by recent access time and the LRU entry is replaced by a newly inserted cache item when the cache is full. The number of buckets in the hash table is determined during the run time of Memcached. If items are continuously inserted, these will result in large item retrieving time due to long-linked list and it will affect the performance. Memcached supports to extend the hash table during run time to solve this problem, but it incurs a large overhead because the process of expending a bucket involves creating a new hash table and copying all of data to the new hash table from the old hash table. To minimize the latency overhead incurred by the expansion of hash table, both the old and new hash tables can be accessed while the hash table is being expanded. Each bucket in the old and new hash tables has a flag to check whether items in the bucket have been moved or not, so the flag is referred to determine which hash table to access.

B. NEAR-MEMORY PROCESSING ARCHITECTURE

In recent times, interest in the concept of NMP, as an inherited architecture model from processor-in-memory (PiM), has been reignited with the emergence of big data applications. We assume a NMP system which is directly connected with

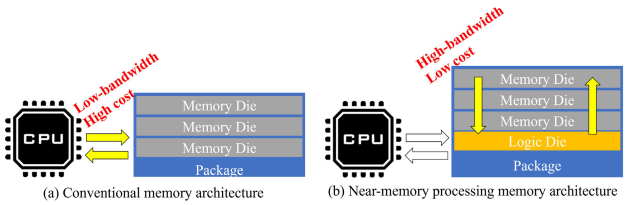


FIGURE 2. Near memory processing architecture.

memory controller of main memory as shown in Figure 2, and NMP is operated as a co-processor of host processor like a GPU. The target application can be efficiently processed on the NMP accelerator opposed to the conventional manner that suffers from high memory overhead on access huge amount of data by loading all data to CPU for processing it. Figure 2(a) shows a conventional system architecture. When processing data in CPU or GPU that is physically distant to DRAM, physical distance between the processor and the main memory causes high latency and energy consumption, besides, it is not easy to increase the bandwidth of the data. As can be seen in Figure 2(b), our baseline NMP architecture has logic die within stacked DRAM, which is a hybrid memory cube (HMC). An accelerator in logic die accelerates offloaded operations of target applications with low memory overheads. The stacked DRAM of NMP architecture supplies data directly to the accelerator on the logic die via Through silicon vias (TSVs), which makes it possible to process applications efficiently by reducing data movement overhead in memory hierarchies in aspects of energy consumption, high bandwidth and low latency of handling data. The architecture of HMC can achieve high density of memory by stacking DRAM dies, and stacked DRAM is divided into vertically aligned partitions which is called a vault. A vault contains a memory controller called a vault controller that allows access to data, and TSVs are placed vertically through the stacked memory for providing high I/O bandwidth.

Similar to SIMD/GPU engines, the manner of acceleration in the NMP engine is triggered by allocating specific workloads/tasks in the application, such as kernels, functions, codes, or threads. Thus, the workload determination that will be running on the NMP engine is one of the most important key factors for reaping the benefit from NMP-based system. We analyze Memcached application to decide the NMP workload which can be expected to obtain performance and memory overhead benefits from NMP accelerator execution.

III. MOTIVATION AND ANALYSIS

A. PROFILING METHODOLOGY

GPROF [7] and KCachegrind [8] are used as profiling tools to extract the NMP workload within Memcached application. Furthermore, the detailed statistics are obtained by attaching several performance counters at the standard Memcached. Additionally, a machine that contains Intel i7-6700K @3.80 GHz, 32 GB of total system memory in the system platform is used for profiling. A last-level cache has 8 MB and

TABLE 1. The types of workloads.

Abbreviation	Workload	Operations	App example
Work-A	Update heavy	Read: 50%, Update: 50%	Session store recording recent actions
Work-B	Read mostly	Read: 95%, Update: 5%	Photo tagging
Work-C	Read only	Read: 100%	User profile cache, where profiles are constructed elsewhere
Work-D	Read latest	Read: 95%, Insert: 5%	User status updates, people want to read the latest
Work-E	Read-modify-write	Read: 50%, Read-modify-write: 50%	User database, where user records are read and modified by the user or to record user activity

```
1   Item *assoc_find(const char *key, const size_t nkey, const uint32_t hv){
2     item *it; /* cache item */
3
4     /* Indexing a hash bucket array */
5     it = hashtable[hv & hashmask(hashpower)];
6
7     /* Item scanning */
8     while(it){
9       if((nkey == it->nkey) && (memcmp(key, it->key, nkey) == 0))
10              return it; /* item hit */
11      else{ it = it->h_next; } /* chasing of the next cache item */
12    }
13  }
```

FIGURE 3. Kernel codes of the Memcached.

it is shared among all cores. Yahoo! datasets from YCSB [9] benchmarks and five workloads of YCSB having diverse fractions of Memcached commands are used in our work. The workloads used in this paper are summarized and presented in Table 1.

B. DETAILED ANALYSIS OF MEMCACHED

We identified two functions that had the largest contribution on execution time of the Memcached workloads. The assoc_find function contributed most to program execution time. This function used approximately 40%, on average, of the total execution time in the Memcached. The execution time of assoc_find always depends on the amount of cached data, the frequency of data access, and the hash table setting of Memcached; it takes a large portion of the execution time at most cases. The function was invoked from almost all functions related to various commands to retrieve a cache item, such as GET, DELETE, UPDATE, and INSERT. The second time consuming function is related to the hashing function (Jenkins_hash in standard Memcached), which computes a hash value using the key and hash mask. On average, this function occupied about 17% of total execution time for our workloads. We focus on the assoc_find function as a target NMP workload to accelerate in this paper.

The pseudocode of the assoc_find function is shown in Figure 3. All the items are traversed by this function for a cache item associated with the provided key. To perform this work, the function is composed of two operating parts: indexing of a bucket array using a result forwarded from the hash function through a bit-wise AND operation (line 5) and

retrieving cache items stored in the indexed bucket (line 8–12). The item traversal consists of comparison between an nkey value and a key value. The nkey is a one-byte variable that refers to the length of the key, so if the nkey of the target item and that of the retrieved item is different, comparison of key values that relatively take much larger then nkey is skipped. It is possible from the code sequence to point out that the item traversal operation is composed of a linked list chain, which is one of the main reasons for the high contribution to the total execution time.

The LLT depth for retrieving items in a bucket is one of the most important factors in memory access time for keys and nkeys. As mentioned above, the in-memory caching application expands the hash table when the number of cached items increases. However, since the maximum LLT depth of the bucket is not limited, LLT depth can be deeper when items are gathered in a same bucket. In addition, the method of reducing the depth of the linked list by expanding the hash table needs to perform a hash function operation on all items, so it causes the large overhead about aspects of memory I/O and energy consumption if the huge amounts of items had been already cached. The recent research about in-memory caching [10] showed that item accesses are concentrated on hot spots, so performance of LLT operation is significantly degraded if the data in the hot spot is not located at the head of the linked list. These characteristics of LLT operation require a quick item search mechanism even when a LLT depth is deep. If this is possible, It can be also minimized the expansion of the hash table of the bucket mentioned above to reduce unnecessary memory overhead. For this purpose, we first analyze the item hit position under different average LLT depths.

The analysis for item hit position under various LLT depths is presented in Figure 4. The average results of five tasks, as shown in Table 1, are plotted according to item load in Figure 4. The x-axis represents the depth of an LLT while the y-axis represents the cumulative value of the hit ratio from the corresponding depth value. We categorize the item access load based upon three categories, low depth, middle depth and high depth for the analysis of hit position according to various item access loads. Low depth, middle depth and high depth are workloads with an average linked list search depth of 2, 4, and 8, respectively. The three types of workload are simulated by controlling the number of buckets in the hash table of Memcached. This is simulated by increasing the item access load in the bucket as the number of buckets decreases.

The analysis of hit position of items shows that ratio of hit on head position slightly decreases when the item load in the bucket increases, and the cumulative curve of the hit ratio becomes linear curve. As the item access load increases, an item in head position, which inserted in recently, is pushed out speedily and the hit ratio cumulative curves gradually become linear curves.

A cache item is retrieved through the memory comparing operation (i.e., memcmp function), comparing the item key against the provided key as presented in Figure 3. The default implementation, however, can be surprisingly slow

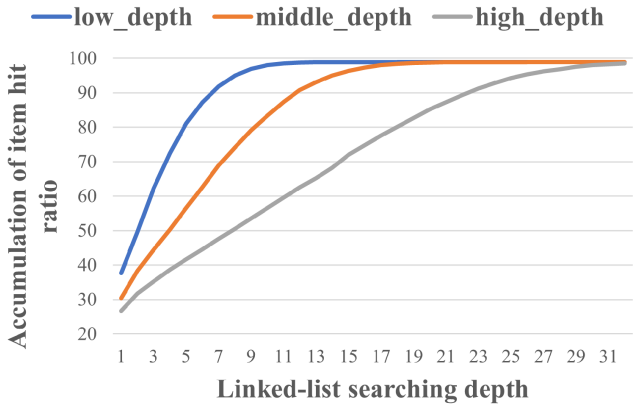


FIGURE 4. The analysis of linked list traversal depth in buckets.

for long key lengths on the in-memory caching system, as it compares byte-by-byte sequentially from the front of the key. It is already shown by [11] that the default implementation memcmp function suffers a lot from long latency due to the method of comparison. Particularly, when keys that only differ near the tail are compared, the operation suffers from latency which is proportional to the length of the key. Item misses can be determined after comparing about 70% of the key string length in the item comparison operation in our evaluated workloads.

C. ANALYSIS RESULTS

It is found that LLT operation is the most important performance bottleneck of Memcached and hits that were concentrated in the head position spread evenly to the tail of the linked list as the item access load increases. Considering these analysis results, we derived two directions to improve the LLT operation performance as (1) reducing the average number of key comparisons to mitigate the reduction of hit ratio of the head position caused by high item access load in the bucket, and (2) executing parallel comparison of key values for rapid execution of LLT opertion even when the average LLT dpeth is high. Furthermore, two mechanisms are proposed based on analysis results, which are (1) ITEM JUMP mechanism to increase a hit ratio of the head position in each bucket and (2) an NMP-based hardware accelerator to perform the comparison of a key of an item in parallel.

IV. ITEM JUMP: A SOFTWARE SOLUTION TO ACCELERATE THE CACHE LINKED LIST TRAVERSAL

It is already confirmed that the efficiency of LRU policy decreases when the item access load in buckets increases from the analysis of hit position of various workloads. In this case, items that are likely to be re-retrieved are continuously pushed out from the MRU(Most recently used) position, then the mechanism is required to maintain items that are likely to be accessed frequently first. A simple mechanism called ITEM JUMP is optimized and adopted, which is proposed in our prior work [12], to have a different data structure to reduce the average linked list traversal depth as a software solution.

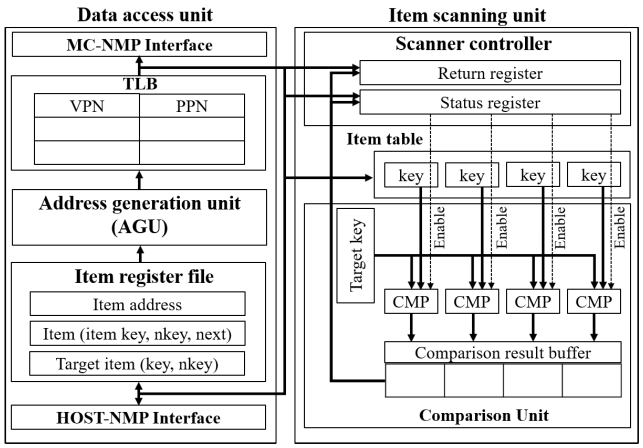


FIGURE 5. An overview of proposed NMP-based item scanner architecture.

A. STRUCTURE OF ITEM JUMP

Figure 1(b) presents the data structure for the ITEM JUMP mechanism. ITEM JUMP has a shadow table that maintains a shadow bucket corresponding to the bucket in the hash table, and each shadow bucket is constructed with symbolic items. Contrary to items in a hash table, each symbolic item in a shadow bucket consists of a pointer to the actual item in the hash table and pointers to the previous and next symbolic item. And each symbolic item is logically linked to a specific item within the bucket array. The key in the item structure can be longer than 200 bytes, and the value occupies memory usually larger than the key value. On the other hand, a symbolic item only have 3 pointers, so it takes up to 24 bytes under the 64 bits addressing environment. Since the size of a symbolic item is generally less than one key value, it does not cause significant overhead. There is a possibility for the breaking of link between the actual and symbolic item when the linked original item is evicted from the bucket. So, a background thread called the symbolic cleaner is implemented to handle an unlinking item, which removes a symbolic item whenever the original cache item is evicted from its LRU chain. We add a flag to the item structure for checking if the item is linked to the symbolic item in the shadow bucket. If the evicted item is linked to the shadow item, a thread which is deleting the item signals the symbolic cleaner to remove the symbolic link, and then evicts the cached item. Since the shadow bucket has only the address of the data, it is not necessary to maintain data consistency, and it is important to remove the symbolic link when the actual data is deleted. The flag only needs 1 bit for each item, and removing symbolic link is executed in the background thread, so the overheads of synchronization between the shadow bucket and the normal bucket are negligible.

B. OPERATION MODEL OF ITEM JUMP

The proposed ITEM JUMP has a simple operation model. First, the retrieval score of the inserted item into the bucket is initialized to zero and the score is increased when the item is retrieved. Note that in the insertion policy, a cache item with two or more retrieval scores (i.e., a re-retrieved item) in the bucket array is registered as a symbolic item in the corresponding shadow bucket. If the shadow bucket is filled in the process of item registration, symbolic item of the tail is evicted. The replacement policy (LRU or FIFO etc.) is adopted when a hit occurs in a symbolic item within a shadow bucket. Note that the shadow bucket is constructed with the limited number of symbolic items. Our method preferentially scans the symbolic items contained in the shadow bucket rather than the bucket in the linked list traversal operation. Thus, the proposed ITEM JUMP mechanism can alleviate the critical performance degradation and data traffic caused by list iteration for entire items, by jumping to a few of cached items within the bucket resulting in improved performance.

If the item retrieval fails in the shadow table, it will experience further higher performance penalty when the item in MRU position shows higher hit count than symbolic item in shadow bucket. Therefore, the number of symbolic items to be maintained in the shadow bucket array should be carefully considered to alleviate item retrieval failure penalties. To decide the number of items in the shadow bucket, we analyzed the efficiency of shadow table according to the number of symbolic entries in Section VI to decide the number of items in the shadow bucket.

V. NMP KERNEL ACCELERATOR: A HARDWARE SOLUTION TO ACCELERATE THE CACHE LINKED LIST TRAVERSAL

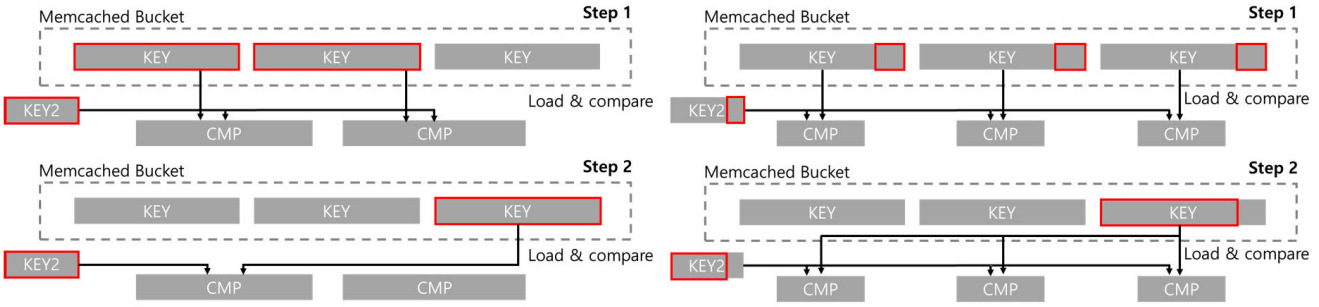
NMP-based hardware accelerator is proposed for improving performance of key comparison in this section. Even if ITEM JUMP is applied, the performance of linked list traversal can be degraded in a situation where the item access load is high. To solve this problem, the proposed hardware accelerator is designed to perform parallel comparisons with minimal data load based on access characteristics based on the measurement result.

A. PARALLEL ITEM SCANNER OVERVIEW

Figure 5 shows the overview of the NMP-based parallel item scanner with two units: data access unit and linked list traversal unit.

The scanner controller is primarily responsible for the management of register files, chasing of cache items, and notification of the list traversal result. The scanner controller notifies the list iteration result into the host processor, and the item register file is used to extract the specific fields from the item structure and chase the next cache item. The TLB and address generation unit are used to load data from main memory to parallel item scanner.

In our method, the item table is used to store the fetched key and nkey data from the main memory and each entry is connected to item comparators. The key or nkey data in the item table are passed to comparators and, then, comparison results are stored into the comparison result buffer. Finally,



(a) Simple parallel item comparator

(b) LSB prioritized parallel item comparator

FIGURE 6. The comparison scheme of each hardware accelerators.

the scanner controller provides the retrieved item to the host from this buffer.

The item comparator is used for comparing the key or nkey values and that is the part where optimization has to be performed practically. The scanner controller can send enable signals to each comparator, and the key value of the target item is transmitted to all comparators. We will discuss two types of proposed parallel item scanners. The first one is a simple parallel item scanner (SPIS) that is simple but executes effective comparison. The LSB-first parallel item scanner (LFPIS) is more improved parallel item scanner design based on the profiling results.

B. SPIS: SIMPLE PARALLEL ITEM SCANNER

We can simply think of deploying a large number of comparators to execute the key comparisons in parallel. In general, when performing SIMD operations, it is processed by reading multiple data word at once and then performing comparison operations simultaneously for each word, so SPIS is a hardware structure that describes this existing method. For example, if the key value is 32 bytes, the SPIS operates by method of reading eight 4 bytes words and comparing them in parallel. In this way, practically the lack of memory bandwidth limits the performance improvement so, it is more important to find the appropriate number of comparators. It is assumed that 16 of 4 bytes-comparators are placed taking into account the comparison cycles and memory bandwidth in this work. We can use these hardware resources to compare 16 items in 4 bytes units at the same time and comparing single item with the size of 64 bytes is also possible.

Considering that the items in buckets follow LRU(Least recently used) policy, the SPIS uses the method of searching items sequentially from MRU position. As the YCSB utilizes a key size of 32-33 bytes, we configured load data and performed parallel comparisons on two items in units of 32 bytes. The corresponding operation of SPIS is simplified and expressed in Figure 6(a). In Step 1, two keys are loaded and processed on two 32 bytes-comparator, and then the other item is processed in Step 2. If the key is larger than 32 bytes and the comparison result of 32 bytes from MRU is missed, the scanner controller does not load remaining bytes. The SPIS is easy to implement because it does not require complex control mechanism. Further, it also has the advantage of being able to achieve high performance if the many item hits occurred in MRU position by the high efficiency of the LRU policy in the bucket.

C. LFPIS: LSB-FIRST PARALLEL ITEM SCANNER

It is found out from the hit position analysis that the increased number of items connected in linked list and actively access to them will reduce the efficiency of the LRU policy. It necessitates further increase in the parallelism of the comparison of items to ensure high performance in this case; however, as mentioned earlier, simply increasing parallelism will make comparison slow for a single item when memory bandwidth is not enough to support the parallel comparison. For minimizing this problem, we can control the missed items after comparison in first 4 bytes do not load additional remaining data, and after that, memory resources are assigned for hit items. However, this method also cannot prevent performance degradation for single item comparison, because it was possible to determine whether the item was hit after comparing 70% of the average key length in YCSB. To prevent the performance degradation of a single item comparison while supporting parallel comparison of multiple items, it is necessary to identify the part of key values used to efficiently determine the item misses. As we were able to discriminate key misses after comparing 70% of the key values when the memcmp function compares key values from MSB, we inversely compared key values from LSB. Figure 7 shows the number of bytes of key values from LSB that can efficiently determine item misses.

The x-axis in Figure 7 represents the size of the bytes performing the comparison from the LSB, and the y-axis shows miss ratio determined by the comparison of that bytes. It was possible to determine almost 90% of all misses with just one byte of LSB, and the ratio increased up to 3 bytes. Then, in low depth work and middle depth work, it was possible to distinguish all misses by comparing 4 bytes from LSB, and 5 bytes comparison had a little improvement only for high depth work.

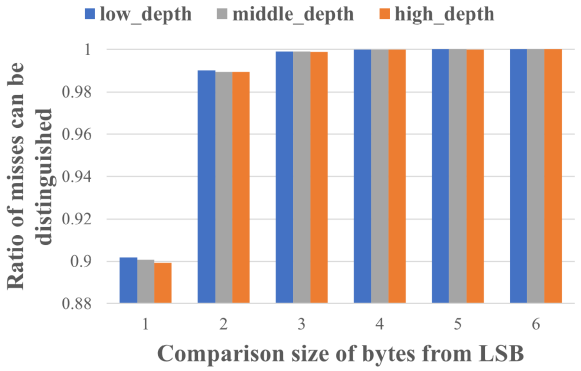


FIGURE 7. The analysis for comparison size of bytes from LSB.

Based on the analysis results of Figure 7, LFPIS is proposed and its operation is described in Figure 6(b). In the proposed LFPIS, 16 items are allocated to the 4 bytes-comparator and compared them with LSB (Step 1). The LFPIS then allocates its hardware resources to matched items and continues comparison for hit items (Step 2). We designed LFPIS to achieve relatively higher performance than SPIS when the average item searching depth becomes so deeper. Conversely, low average item searching depth can degrade the performance of LFPIS. However, it is negligible as most misses can be determined in the first 4 bytes comparison itself.

D. DATA REQUEST OPTIMIZATION FOR LFPIS

It can be seen that the nkey comparison is performed prior to the key comparison in Figure 3. The nkey is a variable to store a length of the key. If the nkey of a retrieved item are different from the nkey of a target item, then there is no necessity to load and compare the key of retrieved item. Because the size of a key value is more than 30 times larger compared to the size of an nkey value in the YCSB workload, it can effectively reduce the unnecessary requested data size. However, as the size of nkey is only 1 byte, there is a disadvantage that it is not possible to efficiently use hardware resources in parallel comparison operations. The other disadvantage is the occurrence of delay between loading the keys and comparing nkeys restricting scanner controller from loading keys before comparison of nkeys. To resolve these issues, the proposed LFPIS is optimized so that it loads a couple of bytes of the keys together with nkeys. This reduces the number of unnecessary data requests and simplifies the steps of loading key comparisons after nkey comparisons. It is assumed that the 1 byte nkey and the 3 bytes key of items are loaded together and assigned to 16 comparators in this work. Only the 3 bytes of a key can determine 99% key misses, as shown in Figure 7, so the performance benefit is rarely reduced. If a miss occurs in the nkey, only 3 bytes of unnecessary memory load is added. Since the overhead of 3 bytes per item is not small, the memory overhead may increase in an environment in which the key length is highly variable. Moreover, the benefit of improving the comparison performance due to the simplified comparison operation step can be maintained regardless of the overhead. The length of the target key and the retrieved key is different approximately 20%–30% in the YCSB benchmark.

Function

```
void* itemKeyCompare(void* ptr1, uint32 offset_key,
    uint32 offset_nkey, uint32 offset_next, const char* key, uint32 nkey)
/* Parameters
ptr1: starting address of item in the linked-list chain
offset_key: offset of key field in the item
offset_nkey: offset of nkey (length of key) field in the item
offset_next: offset of starting address of the next item
key: requested key
nkey: length of requested key
*/
```

```
1  Item *assoc_find(const char *key, const size_t nkey, const uint32_t hv){
2    item *it; /* cache item */
3
4    /* Indexing a hash bucket array */
5    it = hashtable[hv & hashmask(hashpower)];
6
7    /* Item scanning */
8    it = itemKeyCompare( it, offset_key, offset_nkey, offset_next, key, nkey);
9  }
```

FIGURE 8. An application programming interface for proposed hardware accelerator.

E. PROGRAMMING MODEL FOR PROPOSED ITEM SCANNER

To fully utilize our ITEM JUMP with parallel item scanners, a dedicated application programming interface (API) is defined that triggers the proposed parallel item comparator, as presented in Figure 8. The defined parallel item comparator API has six parameters. Parameters *ptr_l* and *offset_next* are used to chase the subsequent cache item, and *offset_key* parameter is used to extract the key field data from the item structure. Similar to an offset key, an *offset_nkey* parameter is used to extract the length of the key information in the item structure. The code in Figure 3 modified to use the API is shown in Figure 8. A software developer can easily execute the proposed parallel item comparator by just transferring a few bytes of information without any data manipulation.

VI. EVALUATION RESULTS

A. EVALUATION ENVIRONMENT

We used MacSIM cycle-level architecture simulator [13]. Table 2 summarizes the detailed simulation configuration. We used five workloads included in YCSB, as described in Section II (see Table 1). We modified the PINTool [14] to allow recognition of the defined API and some traces are reshaped to enable running at parallel item comparator.

B. PERFORMANCE OF ITEM JUMP

The main goal of the ITEM JUMP is to lower the depth of linked list traversal for searching items in buckets. To achieve this goal, LRU was maintained for the management of items in buckets, and a separate shadow table was created to maintain symbolic items likely to be accessed. At this point,

TABLE 2. Simulation configurations.

(a) Host processor	
Cores	X86 OoO core 3.4GHz
L1 I-cache	32KB, 4-way, 3-cycle latency
L1 D-cache	32KB, 4-way, 4-cycle latency
L2 Cache	Private, 256KB, 8-way, 15-cycle latency
L3 Cache	Shared, 1MB, 8 banks, 16-way, 24-cycle latency
(b) ITEM SCANNER	
Frequency	1.6GHz
Item comparator	16 comparators, 2 cycle latency per each comparator 4B comparison for each comparator
(c) Main memory	
Timing parameters	Activation 34 cycles, Precharge 17 cycles, Column 6 cycles
Bus width	16 Bytes per cycle

the number of symbol items in the shadow table can be an important factor in performance. Then, the average linked list traversal depth was analyzed according to the number of symbolic items in the shadow bucket from low depth work to high depth work, and the results are shown in Table 3. The results showed that more than one symbolic item in the shadow buckets have no significant improvement in linked list traversal depth. The difference of average linked list traversal depth was only up to 1.9%. Based on the analysis results, our proposal is designed to have only one symbolic item for each bucket of shadow table. If the number of symbolic items is more than one, a replacement policy is required for shadow bucket. Considering these aspects of the efficiency of item retrieval and the overhead of implementing a replacement policy, it is appropriate to use one symbolic item in each shadow bucket.

The average linked list traversal depth of the proposed ITEM JUMP mechanism compared to the LRU method is presented in Figure 9. The x-axis in Figure 9(a) represents workloads of the YCSB, and the y-axis displays the average linked list traversal depth. The evaluation results show that the ITEM JUMP has a lower average linked list traversal depth for almost all of workloads. The workload D was the only workload that conventional LRU achieved slightly lower linked list traversal depth than that of ITEM JUMP. The LRU policy also works efficiently in workload D of high depth work since the workload D mainly deals with the latest reading data (Table 1). However, ITEM JUMP has shown a difference of only 3.6% compared to LRU even in workload D of low depth work, and ITEM JUMP has reduced linked list traversal depth by 42% on average for overall low depth work. The proposed ITEM JUMP achieved high efficiency in a relatively simple way. However, as the item load increases, the average linked list traversal depth is also increased. Moving from low depth work to high depth work, we were able to realize that the improvement of linked list traversal depth reduction of the ITEM JUMP dropped from

TABLE 3. The average depth according to the number of symbolic items.

#shadow items	Low_depth	Middle_depth	High_depth
1	1.82616	3.69444	7.4794
2	1.80852	3.65878	7.43352
3	1.8417	3.69722	7.36114
4	1.83142	3.6545	7.3417

42% to 24%. Further, to achieve high performance in this case, this paper proposes a hardware accelerator to perform parallel comparison for items as well.

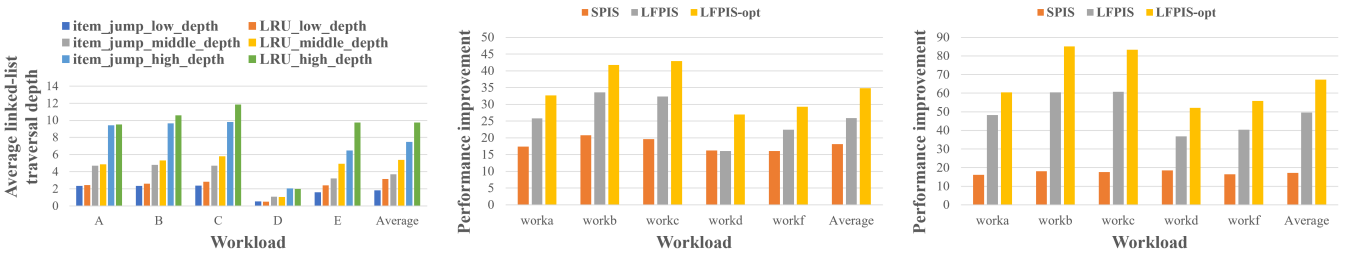
The proposed item jump uses a shadow table to reduce the search depth of the linked list. Since the shadow table occupies additional memory space, it is necessary to calculate the overhead of the additional memory space of the shadow table. The shadow table of this paper is configured to have one item per bucket, and each item consists of three address pointers. Therefore, a memory overhead of 24 bytes per bucket occurs. However, it is not easy to infer the memory overhead value because the linked list depth varies at each bucket in the hash table, and the size of the key-value varies according to system settings. For memory overhead calculation, it was calculated by assuming that the depth of the linked list is two and the size of the key-value pair is 256 bytes based on the workload characteristics used in this paper. Since the length of the key alone sometimes exceeds 200 bytes, assuming that the size of the key-value is 256 bytes is one of the common sizes. In order to maintain two items in the bucket, 544 bytes of memory space is required because two data and four pointers are contained in the bucket. Therefore, the overhead of the shadow table is maintained at around 4% on each bucket in this case. Assuming that the length of the linked list is at a low depth, it is 4%, so if the linked list is prolonged, the ratio of memory overhead will be further reduced as the depth increases. So, the memory overhead of shadow table is not significant unless the key-value size is very small.

C. PERFORMANCE OF PARALLEL ITEM SCANNER

1) SPEED UP OF PROPOSED ARCHITECTURE

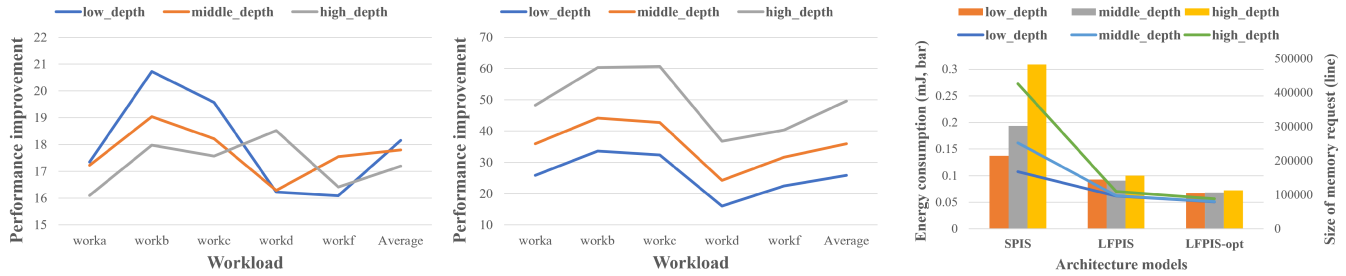
The speedup of four architectures is evaluated, including the normal model without hardware accelerators, SPIS, LFPIS, and LFPIS-opt. Here, the LFPIS with data request optimization is represented as LFPIS-opt. This is for observing the effect of data request optimization separately, and these architecture models are evaluated under the environment that ITEM JUMP mechanism is applied.

Figures 9(b) and 9(c) show the performance improvement of the asoc_find kernel function according to item access load for each architectures. In this graph relative performance of baseline, which is architecture does not have any hardware acceleration is considered 1. The SPIS achieves at least 18 times higher performance than the architecture without hardware accelerator. The LFPIS shows up to 33 times of performance improvement compared to the model without



(a) Average scanning depth of ITEM JUMP (b) Performance for low depth work (c) Performance for high depth work

FIGURE 9. The results of performance for ITEM JUMP and proposed hardware accelerators.



(a) SPIS (b) LFPIS (c) Energy consumption

FIGURE 10. The performance and data usage of proposed hardware accelerator according to various item access load.

hardware acceleration and 42.5% higher performance than SPIS even in low dpeth work, which does not have high parallelism. As most key misses can be determined by one step of parallel comparison operation in the LFPIS, the overhead of slow single key-value comparison is hidden. It could be seen that the performance difference between LFPIS and SPIS was even greater at 188% in the high depth work. We can also check the efficiency of data reqeust optimization by comparing the performance of LFPIS and LFPIS-opt. LFPIS-opt which showed higher performance of 30%–36% over LFPIS from low depth work to high depth work. As a reulst, the LFPIS-opt can achieve 92%–292% higher performance than SPIS. It is confirmed that the efficiency of data request optimization is not significantly affected by item access load, but the data request optimization is affected when the nkey value of the target item is different from the nkey value of the retrieved item.

Figures 10(a) and 10(b) present the results of the performance evaluation by increasing the item access load for SPIS and LFPIS. It is possible to note that the performance improvement increases in proportion to the item access load in the case of the SPIS. The results also show that the performance improvement of the high depth work are increased by 116%. Moreover, the SPIS is heavily influenced by the average linked list traversal depth of buckets. On the contrary, LFPIS showed that the performance improvement are increased only 6% for the same situation. Note that because comparison of 16 items is performed in parallel, there is no significant change in performance for the various item access load in LFPIS.

2) EFFICIENCY OF MEMORY TRAFFIC REDUCTION OF THE PROPOSED ARCHITECTURE

Tens to hundreds of bytes must be read from memory to compare the key of an item. The NMP loads all data form DRAM and costs of access to DRAM is rather costly. Therefore, minimizing memory overhead by preventing the loading of unnecessary data as much as possible is needed for better performance and minimizing energy consumption. In this paper, energy consumption and total requested data size were measured to analyze the memory traffic overhead and the results are presented in Figure 10(c). The bar graph in Figure 10(c) displays energy consumption of memory, and the line graph displays size of total memory request. To calculate energy consumption, CACTI 6.5 [15] was used for obtain values such as dynamic power consumption and static power consumption. As a result, LFPIS reduced 32.6%-67.5%, and LFPIS-opt reduced 51%-76% of enerrgy consumption compared to SPIS from low depth work to high depth work. Total size of memory requested was also reduced 43%-75% and 61%-83% in LFPIS and LFPIS-opt, respectively. As item access load is increased, the LFPIS and LFPIS-opt only rose up to 15%, and 11% in data request count and total requested data size, respectively, while simple architecture increased by 155% on two metrics. One of the important points of the result is that LFPIS-opt showed lower values than LFPIS for requested data size, as well as data request count. The data request optimization reduces the data request count, but since 99% of key misses can be distinguished by 3 bytes, 1 byte data size can be saved compared to LFPIS for each key comparison operation.

VII. RELATED WORK

A. IN-MEMORY CACHING SYSTEM

There are many approaches for improving the performance of in-memory big data management and processing. The approaches can be categorized into a software-level optimization and hardware-based acceleration. First, in software approaches, [16] modifies LRU update strategy, called a bag LRU, in which hash table locking mechanism is changed to allow for parallel access. [17] optimizes Memcached using optimistic cuckoo hashing and LRU-approximating eviction policy based on CLOCK. Byungchul Hong [18] applied request-level parallelism in his paper for parallel search of linked list. The author grouped the data hashed in different indexes and stored them in different memory banks, proposed mechanism allows different requests to be processed in parallel. Scott Lloyd [19] also changed the data placement mechanism to reduce the memory latency of linked list traversal. To improve pointer chasing performance of items in buckets, the author reduced the latency required for random access by storing data in the next index instead of linking items to a connected list. Kevin Hsieh [20] modified TLB to improve the performance of virtual to physical address translation operation by pipelining the address generation part and memory access part separately. Meanwhile, [21] proposes an on-chip coprocessor that is able to accelerate the hash index lookups operation in a hardware-based approach. Additionally, MICA [22] improves the performance for both read- and write-intensive workload by enabling parallel access to the partitioned data.

Previous researches to increase item searching performance of in-memory caching has been conducted in the direction of performing request-level parallel access to partitioned data, or keeping the linked list short by improving hash functions. In contrast, we utilized a simple and effective shadow table to reduce the traversal latency of the linked list without using complex hash functions, and designed a hardware accelerator that can achieve a short latency by performing list traversal linked in parallel to a single query request.

We proposed an integrated approach by including a software and hardware mechanism. This enabled relatively simple mechanisms and hardware to have high performance and reduced data traffic.

B. NEAR-MEMORY PROCESSING

The concept of NMP has been studied as forms of PiM in the 1990s. The logic and DRAM are integrated by [23], [24] by connecting the SIMD-based engine and DRAM sense amplifiers.

Recently, [25] proposed an accelerator with a multiple coarse-grained reconfigurable architecture to accelerate a large loop body in big data applications. In [26], they employ a lot of in-order core in an NMP logic layer for in-memory analytics frameworks to offload the mass data traffic through the on-chip memory hierarchy. Different from these approaches, we decided to use a dedicated hardware accelerator rather than general computation engines to accelerate the linked list traversal operation of in-memory caching applications significantly.

VIII. CONCLUSION

The goal of this work is to improve the performance of a linked list traversal operation, which is the most important kernel operation of in-memory caching applications. We focused on reducing the average linked list traversal depth from software perspective, as well as supporting the parallel comparison to accelerate list iteration from hardware perspective.

From a software perspective, the proposed ITEM JUMP is a simple mechanism that maintains symbolic items to give priority access to items that are repeatedly re-referenced. The symbolic items in the shadow bucket are accessed prior to accessing items in the normal bucket. The proposed ITEM JUMP can reduce 42% of the average linked list traversal depth by maintaining only one symbolic item per bucket.

From a hardware perspective, we proposed the LFPIS mechanism and there were two optimization points of the LFPIS for linked list traversal performance. First, most misses of the key comparison can be determined by comparing only small LSB part of the key values. Second, the linked list traversal performance can be further improved by loading nkey and small part of key values at the same time. As a result, ITEM JUMP and LFPIS-opt improved the performance improvement of linked list traversal operation by up to 3,381% compared to the model that does not have hardware accelerator and 92%–292% performance improvement over the SPIS. LFPIS-opt can reduce data access overhead up to 83% compared to SPIS in terms of memory access overhead. In terms of performance and memory overhead, the LFPIS-opt has showed the highest performance, but depending on the characteristics of the item, SPIS and LFPIS can also be good choices. SPIS has the advantage of high performance and simple control in an environment where the average linked list traversal depth is low, and the memory overhead of the LFPIS may be lower than that of the LFPIS-opt in an environment where the nkey value of the item is highly variable. Therefore, it is necessary to design a hardware accelerator considering the characteristics of a given hardware resource and given data.

We focused on Memcached in this work, but similar approaches that are proposed in this paper can be applicable in various in-memory caching schemes, since the linked list traversal process is essential in in-memory caching application and it will be a future research topic.

REFERENCES

[1] S. Robbins. *Memcached: A Distributed Memory Object Caching System*. Accessed: 2023. [Online]. Available: http://memcached.org/

[2] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, Lombard, IL, USA, 2013, pp. 385–398.
[4] M. Rajashekhar and Y. Yue. (2012). *Twemcache: Twitter Memcached*. [Online]. Available: https: llgithub.comltwitter/twemcache
[5] Lior Abraham et al., "Scuba: Diving into data at Facebook," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1057–1067, Aug. 2013.
[6] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Dec. 2013.
[7] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.
[8] *Kcachegrind*. Accessed: 2023. [Online]. Available: http://kcachegrind.sourceforge.net/html/Home.html and https://hparch.gatech.edu/macsim.html
[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, Indianapolis, IN, USA, Jun. 2010, pp. 143–154.
[10] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "HotRing: A hotspot-aware in-memory key-value store," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb. 2020, pp. 239–252.
[11] R. T. Saunders. (2011). *A Study in Memcmp*. [Online]. Available: http://www.picklingtools.com/study.pdf
[12] H. Lim and G. Park, "JUMPRUN: A hybrid mechanism to accelerate item scanning for in-memory databases," in *Proc. Int. Conf. Big Data Smart Comput. (BigComp)*, Jeju, South Korea, Feb. 2017, pp. 231–238.
[13] *MacSim: A CPU-GPU Heterogeneous Simulation Framework*. [Online]. Available: http://comparch.gatech.edu/hparch/macsim.html
[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005.
[15] *CACTI6.5*. [Online]. Available: https://github.com/Chun-Feng/CACTI-6.5
[16] J. T. Langston Jr. (2012). *Enhancing the Scalability of Memcached*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/enhancing-the-scalability-of-memcached.html
[17] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, Lombard, IL, USA, 2013, pp. 371–384.
[18] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating linked-list traversal through NDP," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Haifa, Israel, Sep. 2016, pp.'113–124.
[19] S. Lloyd and M. Gokhale, "Near memory key/value lookup acceleration," in *Proc. Int. Symp. Memory Syst.*, Alexandria, VA, USA, Oct. 2017, pp. 1–10.
[20] 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 *Proc. IEEE 34th Int. Conf. Comput. Design (ICCD)*, Scottsdale, AZ, USA, Oct. 2016, pp. 25–32.
[21] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. IEEE MICRO*, Dec. 2013, pp. 468–479.
[22] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proc. NSDI*, Apr. 2014, pp. 429–444.
[23] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A memory-SIMD hybrid and its application to DSP," in *Proc. IEEE Custom Integr. Circuits Conf.*, May 1992, pp. 30.6.1–30.6.4.
[24] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proc. 27th Int. Symp. Comput. Archit.*, Vancouver, BC, Canada, Jun. 2000, pp. 161–171.
[25] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, Feb. 2015, pp. 283–295.
[26] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, San Francisco, CA, USA, Oct. 2015, pp. 113–124.

MINKWAN KEE received the B.S., M.S., and Ph.D. degrees in computer engineering from Sejong University, in 2012, 2014, and 2021, respectively. His research interests include hardware accelerators in embedded system and near-memory processing focused on performance and power consumption aspect of accelerators for next-generation computing systems.

CHIWON HAN is currently pursuing the Ph.D. degree with Sejong University. His research interests include AI accelerator design, computer architecture, distributed processing, FPGA, low-power edge system design, and sparse matrix optimization.

GI-HO PARK (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1993, 1995, and 2000, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Sejong University, South Korea. Before joining Sejong University, he was with the Processor Architecture Laboratory, System LSI Division, Samsung Electronics, as a Senior Engineer, from 2002 to 2008. His research interests include advanced computer architectures, AI accelerator design, memory system design, system on chip (SOC) design, and low-power edge system design.