A Data-Centric Hardware Accelerator for Efficient Adaptive Radix Tree

Jin Zhao†, Yu Zhang†, Jun Huang†, Weihang Yin†, Hui Yu†, Hao Qi†, Zixiao Wang†, Longlong Lin§, Xiaofei Liao†, Hai Jin†

†*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, China*

§*College of Computer and Information Science, Southwest University, Chongqing, China*

{zjin, zhyu, jun_huang, hannyin, huiy, theqihao, zwang62}@hust.edu.cn longlonglin@swu.edu.cn {hjin, xfliao}@hust.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132959

***Abstract*—*Adaptive Radix Tree* (ART) is a widely used tree index structure prevalent in various domains such as databases and key-value stores. Despite many solutions have been proposed to improve the performance of ART, they still suffer from significant redundant tree traversals and serious synchronization cost when concurrently performing the operations (e.g., read/write) over ART. In this work, we observe that most operations of real-world workloads tend to target a small subset of ART nodes frequently, exhibiting strong temporal and spatial similarities among the operations. Based on this observation, we propose a data-centric hardware accelerator, called *DCART*, to efficiently support the operations over ART. Specifically, DCART proposes a novel data-centric processing model into the accelerator design to coalesce the operations associated with the same ART nodes and adaptively cache the frequently traversed ART nodes and their search results, thereby fully exploiting the similarities among the operations for lower tree traversal and synchronization overhead. We implemented DCART on the Xilinx Alveo U280 FPGA card and compared it with the cutting-edge solutions, DCART achieves 21.1×-44.2× speedups and 71.1×-148.9× energy savings.**

I. Introduction

Adaptive Radix Tree (ART) [6], [8], [9], [17], [19], [27] is an efficient tree index structure, which plays a crucial role in numerous applications like large-scale database systems [7], [11] and key-value stores [12], [31]. With the wide application of ART, many optimizations (such as DART [28], Heart [17], and SMART [11]) are designed to improve the performance of the operation (e.g., read or write a key-value item) over ART. However, when concurrently performing the operations over ART, these solutions still suffer from significant redundant tree traversals and serious synchronization cost on general-purpose processors due to the following two problems.

First, different operations usually traverse the same nodes of ART individually, which incurs significant *redundant tree traversal overhead*. Moreover, the irregular and unpredictable tree traversal cause massive random data accesses, resulting in poor data locality. It eventually causes most proportion (up to 90.1%) of the fetched nodes in the on-chip memory to be redundant, underutilizing the memory bandwidth significantly. Second, the same node of ART may be handled by different operations concurrently, which brings serious *synchronization cost*. This is because existing ART design applies a lock-based algorithm [8], [9] for concurrency control and the real-world workloads typically perform the operations frequently over a few common nodes of ART. As a result, the synchronization cost wastes much execution time (up to 53.6%).

Through analyzing the operation distribution of the real-world workloads, we have two findings. First, the same nodes of ART may be frequently handled by different operations within a short time interval, which shows temporal similarity. This motivates us to coalesce the processing of the operations targeting the same nodes to achieve lower synchronization and tree traversal cost. Second, many operations get involved with only a small subset of ART nodes, which exhibits spatial similarity. This suggests that these frequently traversed nodes and their search results can be cached in the on-chip memory to further reduce off-chip communications.

Based on the observations, this paper proposes an efficient **data-centric** hardware accelerator *DCART*, which leverages a novel data-centric *Combine-Traverse-Trigger* processing model to effectively handle the operations over ART by fully exploiting the similarities among them. Specifically, DCART features specialized hardware pipelines to dynamically *combine* the operations targeting the same nodes of ART and then *traverse* the nodes of ART associated with these combined operations so as to *trigger* their processing together. Moreover, DCART dynamically maintains the search results (i.e., which are essentially the shortcuts) for the frequently traversed nodes of ART, enabling multiple operations to reuse these shortcuts to quickly search their target nodes. In this way, the redundant tree traversals and synchronization overhead can be effectively alleviated, ensuring higher utilization of the processing unit and fewer off-chip communications. To achieve better data locality, DCART further specializes the memory subsystem to preferentially resident the frequently traversed nodes of ART in the on-chip memory for multiple operations.

We have implemented DCART on a Xilinx Alveo U280 FPGA card. Experimental results show DCART outperforms the cutting-edge ART implementations (i.e., SMART [11] and

This paper is supported by National Key Research and Development Program of China (No. 2023YFB4503400), NSFC (No. 62402457), and Key Research and Development Program of Hubei Province (No. 2023BAB078). This work is also supported by Ant Group through CCF-Ant Research Fund (No. CCF-AFSG RF20240204). Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.

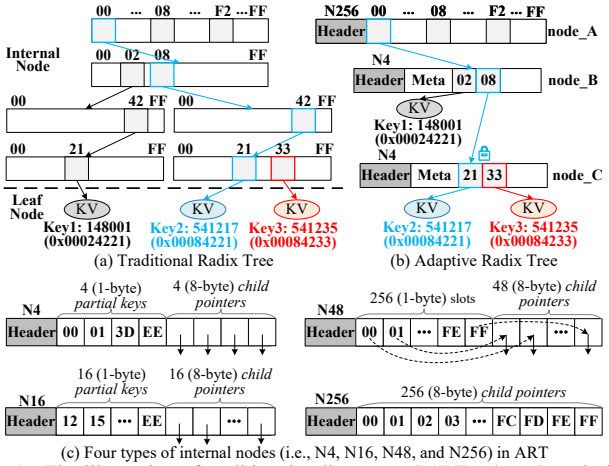

Fig. 1. The illustration of traditional radix tree and ART, where hexadecimal partial keys are shown to enhance clarity

CuART [6]) running on Intel Xeon CPU and NVIDIA A100 GPU by 35.9×-44.2× and 21.1×-31.2× with 92.7×-148.9× and 71.1×-126.2× energy savings, respectively.

II. Background and Motivation

A. Adaptive Radix Tree

Adaptive Radix Tree (ART) [7]–[9], [11], [17], [19], [27], [28] standing out as a widely used tree index structure crafted to enhance the memory efficiency of traditional radix tree. As depicted in Figure 1(a) and (b), both the traditional radix tree and ART organize the segmented keys within the top-down search path of the tree. Specifically, in traditional radix tree, each internal node contains an array of child pointers, where each pointer corresponds to a *partial key* (i.e., a segment of bits of the whole key). Note that an internal node in the traditional radix tree typically reserves space for all 256 potential partial keys. However, due to the sparse distribution of actual keys [8], [11], [17], many of these pointers remain empty, resulting in inefficient memory usage. To resolve this issue, as shown in Figure 1(b), ART [8], [9] optimizes the traditional radix tree by reducing tree height with path compression and introducing four well-designed internal node structures (i.e., `N4`, `N16`, `N48`, and `N256` as shown in Figure 1(c)), where each accommodating a distinct number of pointers (4, 16, 48, and 256, respectively). For better memory efficiency, ART dynamically selects the most suitable internal node structure according to the actual data distribution.

For each operation (e.g., read or write a key-value item) over an ART, it needs to traverse the internal nodes of this ART by performing top-down partial key matching so as to locate the target node of this operation. When concurrently performing the operations over ART, a lock-based algorithm (i.e., the *Read-Optimized Write EXclusionn* (ROWEX) protocol [9]) is used to achieve concurrency control of ART. In detail, it adopts the node-level write locks (e.g., `node_C` must be locked when an operation related to `Key2` needs to modify it), and any read or write is executed atomically. Note that if the operation causes a change in the type of node (e.g., `N4` split into `N16` when `N4` is full), its parent node also needs to be locked.

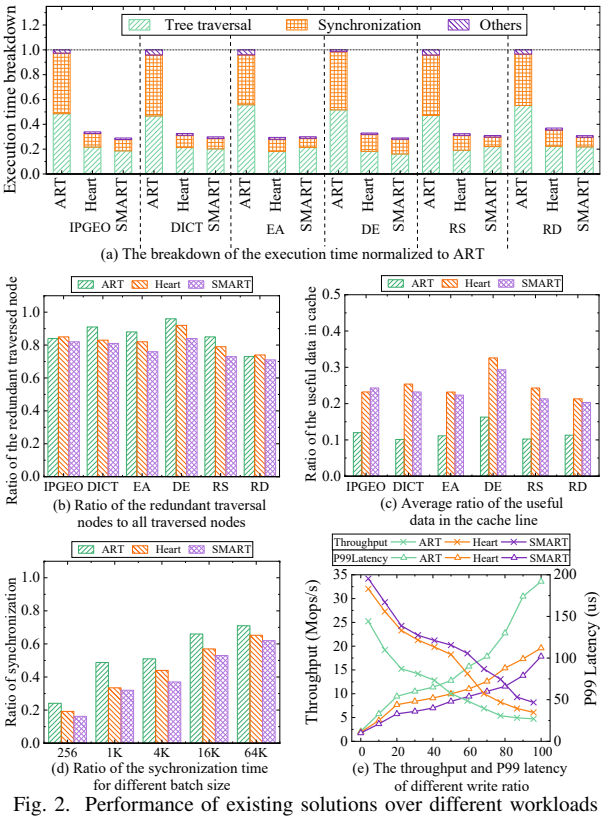

Fig. 2. Performance of existing solutions over different workloads

B. Challenges of Existing Solutions

Although many solutions [6]–[9], [11], [17], [19], [27], [28] have been developed to enhance the performance of the operations (e.g., read or write a key-value item) over ART, they still suffer from significant redundant tree traversal and serious synchronization overhead when concurrently processing the operations over ART. These unique challenges lead to inefficient utilization of the cores and memory bandwidth of the general-purpose processors. To demonstrate these problems, we evaluated three state-of-the-art ART implementations (i.e., ART [9], Heart [17], and SMART [11]) running on Intel Xeon CPU. Note that the details of the platform and benchmarks used in this evaluation are described in Section IV-A. Figure 2(a) shows that, although SMART outperforms other under all circumstances, the majority of execution time for the processing of operations on SMART (more than 95.82%) is still consumed by tree traversal and synchronization. We illustrate the reasons for the above inefficiency as follows.

Challenge 1: Massive redundant tree traversal cost incurs the underutilization of memory bandwidth. Different operations may need to individually traverse the same nodes of ART, incurring significant redundant tree traversal cost. As depicted in Figure 1(b), when processing two operations associated with `Key2` and `Key3`, respectively, each operation needs to individually conduct partial key matching along the nodes `node_A`→`node_B`→`node_C`. Such duplicate tree traversal incurs many redundant off-chip communications (e.g., the accesses to `node_A`, `node_B`, and `node_C`). As shown in Figure 2(b), more than 77.8% of traversed nodes are redundant when handling the operations over SMART, while this ratio

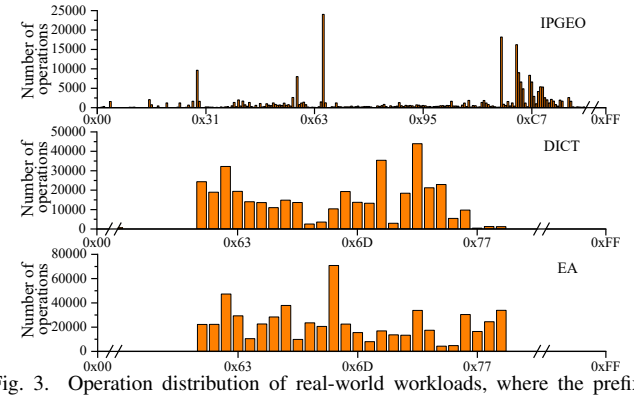

Fig. 3. Operation distribution of real-world workloads, where the prefixes (ranging from 0x00 to 0xFF) of the keys are listed along the horizontal axis and the number of operations led by different prefixes are shown vertically

can be even as high as 86.1% and 82.5% for ART and Heart, respectively. Besides, massive irregular and unpredictable data accesses during the tree traversal cause significant random data accesses, incurring poor data locality. Note that the partial key and child pointer are typically sized of 1 and 8 bytes, respectively, significantly smaller than the 64-byte cachelines of general-purpose processors. Thus, only a small portion of the data loaded into the cache line (20.2% on average in Figure 2(c)) is useful. This eventually results in many superfluous fragmented off-chip communications, underutilizing the on-chip memory and memory bandwidth significantly.

Challenge 2: The concurrency control of ART suffers from serious synchronization overhead. Existing ART implementations typically use lock-based algorithms [6]–[9], [19], [27], [28] to achieve concurrency control, which incurs serious synchronization overhead among different operations over ART. As depicted in Figure 1(b), when the operation associated with Key2 has acquired a lock for the node (e.g., node_C) of ART, the operation associated with Key3 can be conducted only when the operation associated with Key2 has been completed and released the lock. Although several works (e.g., Heart [17] and SMART [11]) endeavor to alleviate the node-level lock contention using the atomic primitive *Compare-And-Switch* (CAS), atomic operations are still very inefficient. This is because the operations over ART suffer from poor data locality, which leads to frequent cache misses, whereas a CAS operation on an Intel processor experiences a slowdown of more than 15 times when data resides in RAM compared to when it resides in L1 cache [21]. As shown in Figure 2(d), when the number of operations increases in the real-world workload *IPGEO*, the ratio of the execution time associated with concurrency control to the total execution time increases from 16.2% to 62.1% over Heart and SMART, while this ratio increases from 24.1% to 71.3% over ART. In addition, from Figure 2(e), we can observe that the performance deteriorates rapidly when the ratio of write operations increases in *IPGEO* (i.e., more lock acquirement and release are required).

C. Similarities among the Operations over ART

Figure 3 describes the statistical studies on the operation distribution of three real-world workloads. We observe that there are strong spatial and temporal similarities among the operations over ART. It provides opportunities to address the limitations of existing architectures and enhance the performance of the operations over ART.

Observation 1: The same nodes of ART may be frequently handed by different operations within a short period of time, which exhibits the temporal similarity. Figure 3 shows that the nodes corresponding to the keys with the prefix 0x67 are traversed and handled by over 24,000 operations for *IPGEO*. This motivates us to propose a novel ***data-centric*** *Combine-Traverse-Trigger* (CTT) processing model to efficiently coalesce the operations over ART by fully exploiting the temporal similarity. This model first *combines* the operations targeting the same nodes of ART according to the prefixes of their keys. It then *traverses* the nodes of ART to search the target nodes of these combined operations and finally *triggers* the processing of these operations together. Thus, each node of ART needs to be traversed only once to drive the processing of multiple operations, and the concurrent operations associated with this node can be serialized to naturally alleviate synchronization overhead. This is fundamentally different from the ***operation-centric*** processing model used in existing solutions [6]–[9], [11], [17], [19], [27], [28] that launch each operation to individually perform the tree traversal over ART to retrieve and atomically handle its target nodes. Take Figure 1(b) as an example, our proposed CTT processing model first combines the operations associated with Key2 and Key3 because their keys have same prefix, and then traverses the nodes of ART through conducting the partial key matching along node_A→node_B→ node_C. After that, the combined operations will be triggered to perform together without the synchronization of node_C when these operations need to modify node_C. Besides, the duplicate partial key matching from node_A to node_C can be eliminated by coalescing the same tree traversals, achieving lower tree traversal overhead.

Observation 2: Many operations involve only a small fraction of nodes within ART, which exhibits the spatial similarity. Figure 3 shows that more than 96.65% of tree traversals is responsible for accessing only 5% of the nodes in ART. The results depict that the node traversal distribution is usually skewed. This prompts us to enhance the performance of our proposed data-centric CTT processing model via the following aspects. First, the frequently traversed nodes (denoted as the high-value nodes) can be preferentially resident in the on-chip memory, efficiently exploiting the spatial similarity among the operations for fewer off-chip communications. Second, the partial key matching results associated with these nodes (e.g., the prefix of the key and the address of the corresponding node such as <0x000842, address of node_C> in Figure 1(b), which is essentially a *shortcut*) can be cached to be directly reused by other operations that need to handle them (i.e., the operations that need to handle node_C), further reducing the tree traversal overhead.

Challenges: Although our proposed data-centric CTT processing model ensures much lower synchronization cost and fewer tree traversals, its software-only implementation suffers from high runtime overhead (see Section IV-B) due to the

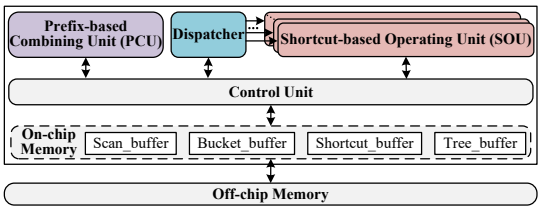

Fig. 4. Architecture of DCART

following reasons. First, it requires expensive runtime cost to dynamically coalesce the operations and maintain the shortcuts on the fly, which may impede the performance improvement brought by exploiting the similarities among the operations over ART. Second, the tree traversal involves data-dependent branches, which induces low instruction-level parallelism. To overcome these challenges, we present a data-centric hardware accelerator *DCART*, which customizes specialized hardware designs to efficiently handle the operations over ART.

III. DCART Architecture

A. DCART Overview

Figure 4 depicts the architecture of DCART, which contains three key hardware units (i..e, *Prefix-based Combining Unit* (PCU), *Dispatcher*, and *Shortcut-based Operating Unit* (SOU)) and on-chip buffers. Specifically, the PCU is employed to combine the operations targeting the same ART nodes by assigning the operations into disjoint buckets based on the prefixes of their keys. The *Dispatcher* is used to dispatch these buckets to different SOUs for parallel processing, in which the operations targeting the same nodes are only handled by a single SOU, alleviating synchronization overhead. Each SOU is employed to efficiently handle the combined operations in each bucket and also maintain the shortcuts to eliminate the duplicate tree traversal cost. On-chip memory is composed of several buffers to cache different types of data and also isolate the accesses to these different data (e.g., tree structure and operations), efficiently reducing off-chip communications. To fully exploit the spatial similarity of the operations over the ART, it also adaptively caches the frequently handled nodes for multiple operations, avoiding the data thrashing of them.

B. Prefix-based Operation Combining

To alleviate the lock contention among the concurrent operations over ART, the PCU sequentially scans the keys of the operations and dynamically assigns them into different disjoint buckets according to the prefixes of these keys. As shown in Figure 5, the PCU performs three stages as a pipeline to efficiently achieve the operation combining. Note that sixteen tables are created in the off-chip memory to store the combined operations for different buckets, where each table (e.g., *Bucket_Table*$_i$) is specified to a particular bucket (i.e., the i^{th} bucket) using a label (defined by the prefix). When the concurrent operations are arrived, at the *Scan_Operation* stage, it sequentially obtains the key of each operation. Then, the *Get_Prefix* stage calculates the specified prefix (e.g., *Prefix*$_i$) of this key, where the first 8 bits of the key are used as the specified prefix by default. After that, at the *Combine_Operation* stage, it assigns this operation into the corresponding *Bucket_Table* by matching *Prefix*$_i$ with

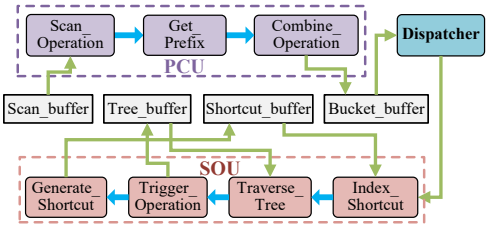

Fig. 5. Microarchitecture of DCART, where blue solid lines and green solid lines denote data flow and on-chip memory transfers, respectively

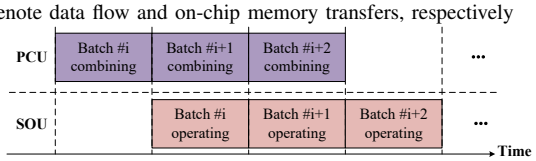

Fig. 6. The overlap between PCU and SOU

the labels of *Bucket_Tables* and then sets this operation as combined. In this way, the operations that target the same nodes are combined into the same bucket. It enables the *Dispatcher* to dispatch the operations that target the same node to be handled by only a single SOU, avoiding the acquirement of locks to perform synchronization for concurrent control.

C. Shortcut-based Operating

When a SOU is dispatched with a bucket of operations, this SOU sequentially retrieves an operation from this bucket and searches the target node of this operation by traversing the tree or using the shortcuts (to be introduced later), efficiently conducting the operations that target the same node together with lower tree search overhead. Note that a hash table, i.e., *Shortcut_Table*, is created in the off-chip memory to store the partial key matching results to be reused by multiple operations, avoiding significant redundant tree traversal overhead. Each entry of *Shortcut_Table* is the form of <Key_ID, Address_Target_Node, Address_Parent_Node>.

Specifically, as shown in Figure 5, each SOU contains four stages, which are conducted as a pipeline. At the *Index_Shortcut* stage, the SOU pops an operation from the assigned bucket and obtains the key (e.g., Key$_x$) of this operation. Then, it uses the Key$_x$ to obtain the addresses of the target node (e.g., Node_X) and its parent node (e.g., Node_Y) according to the *Shortcut_Table*. If the corresponding addresses can be obtained from the *Shortcut_Table* (e.g., Node_X has been handled by another operation previously), the *Traverse_Tree* stage will directly use these addresses to fetch Node_X and Node_Y from the tree structure (i.e., the ART), avoiding the tree traversal. Otherwise, the *Traverse_Tree* stage will use Key$_x$ to obtain Node_X and Node_Y through performing top-down partial key matching over the tree. At the *Trigger_Operation* stage, the SOU triggers all retrieved operation (e.g., read or write) on the target node to be performed together. If Node_X and Node_Y are obtained by traversing the tree, the *Generate_Shortcut* stage will generate a shortcut associated with Key$_x$, i.e., creating an entry (i.e., <Key$_x$, Address_Node_X, Address_Node_Y>) in *Shortcut_Table*. Note that the corresponding entry in *Shortcut_Table* needs to be updated when this operation causes a change in the type of Node_X. As the operations that target the same node being coalesced, the SOU typically obtains the

target nodes of the operations directly based on the maintained shortcuts and also achieves better data locality.

D. Overlap of Combining and Operating

The combining of the operations needs to assign them into different disjoint buckets on the fly, which introduces extra runtime cost. Fortunately, the interleaving between combining and operating for each operation batch provides the potential opportunity to hide the operation combining overhead. Thus, we can sequentially divide the arrived operations into a series of batches. As shown in Figure 6, when the i^{th} batch of operations has been handled by PCU, the Dispatcher will assign the combined operations associated with the i^{th} batch to be handled by SOUs. In the meantime, the PCU starts combining the operations of the $(i+1)^{th}$ batch. By such means, the overheads of the operation combining can be usually hidden for better performance.

E. Value-aware Memory Hierarchy

To efficiently access the ART nodes, arrived operations, *Bucket_Tables*, and *Shortcut_Table*, as shown in Figure 4, four on-chip buffers, i.e., *Tree_buffer*, *Scan_buffer*, *Bucket_buffer*, and *Shortcut_buffer*, are employed to cache these data, respectively. However, the high-value nodes (introduced in Section II-C) may be evicted from the *Shortcut_buffer* due to the irregular tree traversal over ART. The access to the tree structure typically accounts for the most portion of all data accesses. Therefore, we design a value-aware data management strategy to efficiently manage the *Tree_buffer*, while DCART uses LRU [4] to manage the other buffers by default. In detail, for the management of the *Tree_buffer*, DCART uses the values of the nodes to determine the victims. Note that a node (e.g., `Node_X`) will be handed more times when the corresponding bucket owns more operations, thus we use the number of the operations in the corresponding bucket to approximate the value of this node (i.e., $Value_x$), which can be obtained after the operation coalescing. When a node `Node_X` is requested, if the *Tree_buffer* is not full, `Node_X` will be directly cached in the *Tree_buffer*. Otherwise, DCART will evict the node with the lowest value (i.e., $Value_{low}$) from the *Tree_buffer* and cache `Node_X` in the *Tree_buffer* when $Value_x$ is greater than $Value_{low}$. In this way, DCART can effectively prevent cache thrashing for high-value nodes (i.e., frequently traversed nodes), ensuring better data locality.

IV. Evaluation

A. Experiment Setup

DCART Settings. We implement DCART on the Xilinx Alveo U280 FPGA card, which is equipped with the XCU280 FPGA chip. The FPGA provides 1.3 M LUTs, 2.6 M Registers, 9 M BRAM resources, and 8 GB HBM. DCART has 16 SOUs, and we implement the on-chip memory of DCART using the BRAM resources. More parameter details of DCART are depicted in Table I. We use Xilinx Vivado 2019.1 to obtain the clock rate for DCART and conservatively use 230MHz in our experiments. Besides, we also implement our data-centric CTT processing model on a machine equipped with two 48-core Intel Xeon Platinum 8468 processors and 1024 GB DRAM.

TABLE I
PARAMETER DETAILS OF DCART

Compute	1×PCU, 1×Dispatcher, 16×SOUs
On-chip memory	*Scan_buffer* (512 KB), *Bucket_buffer* (2 MB), *Shortcut_buffer* (128 KB), *Tree_buffer* (4 MB)

The CPU version (i.e., the software-only implementation of our processing model) is called *DCART-C*.

Workloads. We use three real-world workloads (i.e., *IP address record by GeoLite2-Country* (IPGEO)[1], DICT[2], and *E-mail Addresses* (EA)[3]) and three synthesized workloads [9] with 50M dense 8-byte integer keys (DE), 50M random sparse 8-byte integers keys (RS), and 50M random dense 8-byte integers keys (RD), respectively. For each workload, the operations consist of 50% read and 50% write by default.

Baselines. We compare DCART with two cutting-edge CPU-based tree indexes (i.e., ART [9] and SMART [11]) and a state-of-the-art GPU-based tree index (i.e., CuART [6]), where ART and SMART are used as the CPU baselines and CuART is used as the GPU baseline. Note that SMART is designed for disaggregated memory, thus we port them to shared-memory by re-implementing them from scratch. All experiments run on a machine configured with two 48-core Intel Xeon Platinum 8468 processors, 1024 GB DRAM, an NVIDIA A100 GPU, and a Xilinx Alveo U280 FPGA. Like existing solutions [1], [3], [22], [26], we employ CPU Energy Meter [15], nvidia-smi [18], and xbutil [25] to measure the energy consumption of the CPU-based solutions (i.e., ART [9], SMART [11], and DCART-C), the GPU-based solution (i.e., CuART [6]), and the FPGA-based solution (i.e., DCART), respectively.

B. Overall Performance

Lock Contentions. Figure 7 evaluates the number of lock contentions induced by different solutions. It shows that the lock contentions induced by DCART-C and DCART are only 3.2%-19.7% of that by other solutions. This is because our data-centric CTT processing model only needs to acquire a single lock for multiple operations that need to handle the same node by efficiently coalescing these operations, alleviating massive lock contentions for lower synchronization overhead.

Partial Key Matches. Figure 8 shows the number of partial key matches performed by different solutions. It depicts that the number of partial key matches of DCART-C and DCART are only 3.2%-5.7%, 6.5%-14.3%, and 8.8%-15.9%, of ART, SMART, and CuART, respectively. The reasons are twofold. First, different operations can share the common tree traversals through combining these operations according to their prefixes. Second, the shortcuts of the frequently traversed nodes can be reused by multiple operations to directly search their target nodes, further reducing the tree traversal overhead.

Performance. Figure 9 evaluates the execution time of different solutions. We can find that DCART-C only slightly outperforms ART, SMART, and CuART, although our processing model can efficiently reduce redundant tree traversals

[1]https://github.com/analogic/ipgeo#dbsql
[2]https://github.com/dwyl/english-words
[3]https://archive.org/details/300MillionEmailDatabase

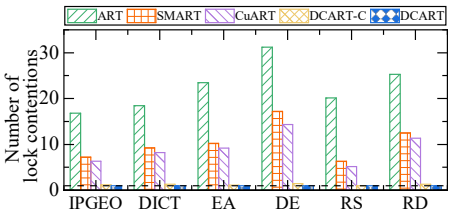

Fig. 7. Number of lock contentions

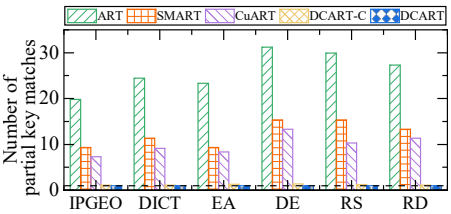

Fig. 8. Number of partial key matches

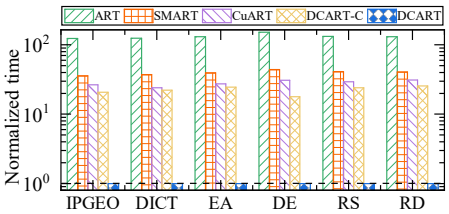

Fig. 9. Normalized execution time

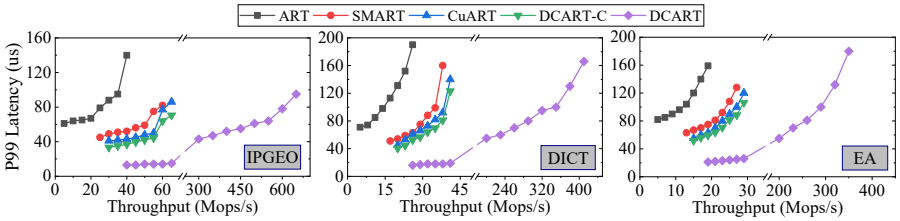

Fig. 10. The throughput-latency curves under different real-world workloads

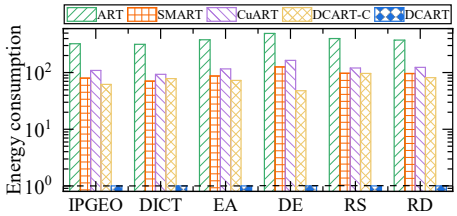

Fig. 11. Normalized energy consumption

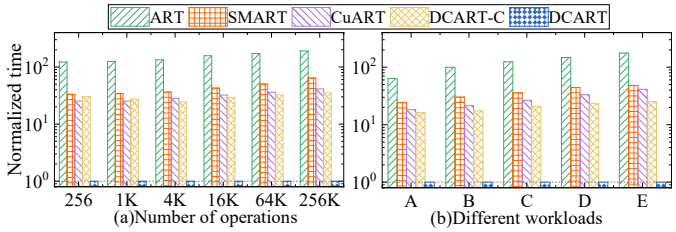

Fig. 12. Sensitivity studies

and synchronization overhead when handling the operations over ART. This is because DCART-C requires high runtime overhead to dynamically combine the operations, maintain the shortcuts, and irregularly traverse the tree structure on the fly. This impedes the overall efficiency of our processing model. Different from DCART-C, DCART can achieve 123.8×-151.7×, 35.9×-44.2×, and 21.1×-31.2× performance improvements compared with ART, SMART, and CuART, respectively. The reasons are manifold. First, DCART can reduce the runtime cost arising from our processing model because of its sophisticated hardware designs. Second, DCART can preferentially resident the frequently traversed ART nodes on the on-chip memory, achieving better data locality. Figure 10 evaluates the throughput-latency curves of different solutions with three real-world workloads, using various numbers of the operations. It shows that DCART achieves lower P99 latency and higher throughput in comparison with other solutions.

Energy Savings. Figure 11 shows that DCART reduces the energy consumption of ART, SMART, CuART, and DCART-C by 315.1×-493.5×, 92.7×-148.9×, 71.1×-126.2×, and 48.1×-97.6×, respectively.

C. Sensitivity Studies

Figure 12(a) evaluates the performance of different solutions when handling *IPGEO* with different number of concurrent operations. It shows that DCART achieves better performance as the number of operations increases. This is because DCART can efficiently coalesce the operations to alleviate more lock contention and tree traversal cost. Figure 12(b) evaluates the performance of different solutions with various workloads of *IPGEO*, where the workloads contain A (100% read), B (75% read, 25% write), C (50% read, 50% write), D (25% read, 75% write), and E (100% write). The results show that better performance improvement can be obtained as the ratio of write increases (which means more lock contention).

V. Related Work

Two categories of index structures (i.e., hash indexes and tree-based indexes) are most widely used today for databases.

Hash Indexes. Hash indexes [2], [14], [16], [32] are flat data structures that are able to support fast point access within constant lookup time complexity, i.e., O(1). However, because hash tables scatter the keys randomly, they are unable to support range queries efficiently.

Tree-based Indexes. Tree-based indexes [10], [20], [29], [30] are critical for many applications requiring range queries, and most previous databases typically apply the variants of B+tree [5], [13], [23], [24] to build range indexes. However, B+tree suffers from write amplification. In comparison with it, ART [8], [9] has smaller write amplification because it does not hold the entire keys in its internal nodes. Recently, many optimizations [7], [12], [19], [27], [31] have been proposed for ART. DART [28] introduces a triplet-based search mechanism to guarantee its scalability. Heart [17] proposes a PM-friendly node structure and a CAS-based concurrency control method to reduce the high PM overhead and improve scalability. SMART [11] is proposed to achieve efficient ART for disaggregated memory. To further improve the performance of ART, CuART [6] offloads the read and write operations onto GPUs. However, these solutions still suffer from serious redundant tree traversals and synchronization cost when concurrently performing the operations over ART. In contrast, DCART can effectively overcome these challenges by fully exploiting the temporal and spatial similarities among these operations.

VI. Conclusion

This paper proposes a novel data-centric hardware accelerator DCART to enhance the efficiency of operations over ART. By coalescing the operations targeting the same ART nodes and adaptively caching the frequently traversed ART nodes and their search results, DCART significantly reduces the synchronization cost and off-chip communications, achieving high energy efficiency. Compared to the state-of-the-art solutions, i.e., SMART [11] and CuART [6], DCART gains 21.1×-44.2× speedups with 71.1×-148.9× energy savings.

REFERENCES

[1] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W. Wong, "Regraph: Scaling graph processing on hbm-enabled fpgas with heterogeneous pipelines," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*, 2022, pp. 1342–1358.

[2] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: an experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, 2021.

[3] M. Jay, V. Ostapenco, L. Lefèvre, D. Trystram, A. Orgerie, and B. Fichel, "An experimental comparison of software-based power meters: focus on CPU and GPU," in *Proceedings of the 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2023, pp. 106–118.

[4] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 284–296.

[5] W.-H. Kim, M. K. Ramanathan, X. Fu, S. Kashyap, and C. Min, "PACTree: a high performance persistent range index using pac guidelines," in *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles*, 2021, pp. 424–439.

[6] M. Koppehel, T. Groth, S. Groppe, and T. Pionteck, "Cuart - a cuda-based, scalable radix-tree lookup and update engine," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[7] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: write optimal radix tree for persistent memory storage systems," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, 2017, pp. 257–270.

[8] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Proceedings of the 29th IEEE International Conference on Data Engineering*, 2013, pp. 38–49.

[9] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 3:1–3:8.

[10] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 1195–1206, 2010.

[11] X. Luo, P. Zuo, J. Shen, J. Gu, X. Wang, M. R. Lyu, and Y. Zhou, "SMART: a high-performance adaptive radix tree for disaggregated memory," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023, pp. 553–571.

[12] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "Roart: Range-query optimized persistent art," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, 2021, pp. 1–16.

[13] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 2012 European Conference on Computer Systems*, 2012, pp. 183–196.

[14] A. Mathew and C. Min, "Hydralist: A scalable in-memory index using asynchronous updates and partial replication," in *Proceedings of the VLDB Endowment*, 2020, pp. 1332–1345.

[15] L. Munich, "CPU energy meter," 2021. [Online]. Available: https://github.com/sosy-lab/cpu-energy-meter

[16] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019, pp. 31–44.

[17] L. Nie, S. Zheng, B. Zhang, J. Xu, and L. Huang, "Heart: a scalable, high-performance art for persistent memory," in *Proceedings of the 41st IEEE International Conference on Computer Design*, 2023, pp. 487–490.

[18] NVIDIA, "Nvidia system management interface," 2023. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[19] W. Pan, T. Xie, and X. Song, "HART: A concurrent hash-assisted radix tree for DRAM-PM hybrid memory systems," in *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium*, 2019, pp. 921–931.

[20] K. Ren, Y. Guo, J. Li, X. Jia, C. Wang, Y. Zhou, S. Wang, N. Cao, and F. Li, "Hybridx: New hybrid index for volume-hiding range queries in data outsourcing services," in *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems*, 2020, pp. 23–33.

[21] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *Proceedings of the 2015 International Conference on Parallel Architectures and Compilation*, 2015, pp. 445–456.

[22] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 211–216.

[23] Q. Wang, Y. Lu, and J. Shu, "Sherman: a write-optimized distributed b+tree index on disaggregated memory," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1033–1048.

[24] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, "Building a bw-tree takes more than just buzz words," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 473–488.

[25] Xilinx, "Vitis unified software development platform 2020.2 documentation," 2020. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/index.html

[26] Z. Yang, K. Adamek, and W. Armour, "Accurate and convenient energy measurements for gpus: A detailed study of nvidia gpu's built-in power sensor," in *Proceedings of the 2024 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 307–323.

[27] J. Zhang, Y. Luo, P. Jin, and S. Wan, "Optimizing adaptive radix trees for nvm-based hybrid memory architecture," in *Proceedings of the 2020 IEEE International Conference on Big Data*, 2020, pp. 5867–5869.

[28] W. Zhang, H. Tang, S. Byna, and Y. Chen, "DART: distributed adaptive radix tree for efficient affix-based keyword search on hpc systems," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 24:1–24:12.

[29] Y. Zheng, R. Lu, Y. Guan, J. Shao, and H. Zhu, "Towards practical and privacy-preserving multi-dimensional range query over cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3478–3493, 2022.

[30] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: efficient range query for lsm-trees," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, 2021, pp. 51–64.

[31] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast rdma capable networks," *Proceedings of the 2019 International Conference on Management of Data*, pp. 56–67, 2020.

[32] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 461–476.