Eunomia: Scaling Concurrent Index Structures Under Contention Using HTM

Weihua Zhang, Xin Wang, Shiyu Ji, Ziyun Wei, Zhaoguo Wang, and Haibo Chen, *Senior Member, IEEE*

Abstract—Hardware transactional memory (HTM) is an emerging hardware feature. HTM simplifies the programming model of concurrent programs while preserving high and scalable performance. With the commercial availability of HTM-capable processors, HTM has recently been adopted to construct efficient concurrent index structures. However, with the expansion of data volume and user amount, data management systems have to process workloads exhibiting high contention; meanwhile, according to our experiments, the conventional HTM-base concurrent index structures fail to provide scalable performance under highly-contented workloads. Such performance pathology strictly constrains the usage of HTM on data management systems. In this paper, we first conduct a thorough analysis on HTM-based concurrent index structures, and uncover several reasons for excessive HTM aborts incurred by both false and true conflicts under contention. Based on the analysis, we advocate Eunomia, a design pattern for HTM-based concurrent index structure which contains several principles to improve HTM performance, including splitting HTM regions with version-based concurrency control to reduce HTM working sets, partitioned data layout to reduce false conflicts, proactively detecting and avoiding conflicting requests, and adaptive concurrency control strategy. To validate their effectiveness, we apply such design principles to construct a scalable concurrent B+Tree and a skip list using HTM. Evaluation using key-value store and database benchmarks on a 20-core HTM-capable multi-core machine shows that Eunomia leads to substantial speedup under high contention, while incurring small overhead under low contention.

Index Terms—Hardware transactional memory, concurrent index structure, data conflicts

1 Introduction

The emergence of hardware transactional memory (HTM) [1] provides a new opportunity to construct efficient concurrent data structures. HTM exploits cache coherence mechanisms to protect the consistency of critical sections, which may approach the performance of fine-grained locking or even lock-free schemes while preserving the simplicity of programming with coarse-grained locks. For this reason, there have been plenty of efforts to design concurrent index structures (e.g., skip list, search tree, hash table) using HTM [2], [3], [4], which was shown to achieve satisfying performance in data management systems.

However, due to the increased number of processor cores [5], a skewed distribution of key accesses [6], or the contention on shared entities in databases [7], data management systems have to confront with highly-contented workloads [7], [8], [9]. Yet, according to our experiments, conventional HTM-based concurrent index structures fail to deliver high performance and good scalability when workloads exhibit high contention. With such pathological performance, the universality of HTM is severely constrained.

This paper attempts to answer a natural question: with the assistance of HTM, can we construct a concurrent index structure that delivers high and scalable performance even under high contention? To answer this question, we first present a detailed analysis of the performance of a recent concurrent HTM-based B+Tree used as the index in several in-memory data management systems [2], [3], [4]. Our analysis uncovers several key issues leading to non-scalable performance under contention.

First, traditional HTM-based index structure protects the consistency of operations in a large, monolithic HTM region, whilst most data conflicts actually occur in some certain parts of the entire index structure. This leads to high retry cost. Second, a wide variety of ordered index structures store records in a dense and consecutive manner, which incurs severe false sharing problem under HTM due to a coarse-grained (i.e., cache line) conflict checking. Third, many index structures intrinsically contain pervasive shared meta-data to maintain semantics, and accesses to shared variables usually cause conflicts among transactions. The above design defects are derived from some common features of index structures, such as uneven distribution of modifications and dense memory layout. These features of index structures are the basis of our design pattern.

Based on the analysis of performance issues and data structure features, we present Eunomia, a design pattern that attempts to tackle the above issues with the following design guidelines. First, based on the observation that modifications distribute unevenly within concurrent index

- W. Zhang, X. Wang, S. Ji, and Z. Wei are with the Software School, Shanghai Key Laboratory of Data Science, and Parallel Processing Institute, Fudan University 200433, Shanghai, China.
E-mail: {zhangweihua, xin_wang, syji14, weizy14}@fudan.edu.cn.
- Z. Wang is with the Department of Computer Sciences, New York University, New York, NY 10003. E-mail: zhaoguo@nyu.edu.
- H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiaotong University, Shanghai 200240, China.
E-mail: haibochen@sjtu.edu.cn.

*Manuscript received 17 Apr. 2017; revised 7 June 2017; accepted 29 June 2017. Date of publication 20 July 2017; date of current version 13 July 2018.
(Corresponding author: Zhaoguo Wang.)
Recommended for acceptance by L. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2729551*

structures, we partition a monolithic HTM region into parts according to the *index phase* and *operation phase* in such concurrent index structure; each part protects the atomicity of different parts with HTM mechanism respectively. A version-based scheme is designed to guarantee the overall consistency at the boundary between different HTM regions. With such scheme, most conflicts only cause retries within the partitioned transaction pieces, instead of the entire monolithic transaction. Second, to eliminate false conflicts incurred by consecutive data layout and meta-data accesses, Eunomia refactors the index structure in a partitioned way, which dispatches concurrent requests to different segments. Third, to throttle the conflicting requests, Eunomia adopts an efficient mechanism, which anticipates potential conflicts and avoids them accordingly. Finally, Eunomia adopts an adaptive contention control mechanism, which can detect various contention rates and achieve high performance under both high and low contention.

We have applied these design guidelines to two representative concurrent index structures: B+Tree and skip list. Experimental results using the YCSB benchmark to evaluate key-value store performance show that under high contention, Eunomia can yield 5X-11X speedup over conventional HTM-based B+Tree and 4X-6X speedup over HTM-based skip list. We also evaluate the performance of concurrent index structures in a database system with TPC-C benchmark, and the results show that Eunomia yields substantial speedup with database workloads.

In summary, this work makes the following contributions.

- A comprehensive analysis of HTM-based concurrent index structures under high contention.
- A design pattern with four design guidelines for scalable concurrent HTM-based index structures.
- Applying the design pattern to two concrete index structures, yielding high performance and scalability with contended workloads.

The paper is organized as follows. Section 2 introduces the background about HTM and concurrent index structures. In Section 3, we conduct a detailed analysis on the pathological performance of HTM-based index structures under high contention. Section 4 discusses common features of concurrent index structures and presents Eunomia design pattern. Section 5 presents how to apply the Eunomia design pattern on a B+Tree. In Section 6, we give out the experiments on two concurrent index structures. In Section 7, we present a system evaluation under TPC-C workload. The related work is summarized in Section 8, and we conclude the paper in Section 9.

2 Background

This section introduces the necessary background regarding HTM and index structures, especially the usage of HTM to design concurrent index structures.

2.1 HTM Semantics

With the commercial popularization of IBM z- and p-Series [10], [11] and Intel Haswell [12] processors, HTM has been widely available to the mass market. Here we use Intel's RTM (Restricted Transactional Memory) as an example to illustrate the semantics and quirks of HTM.

RTM provides *xbegin* and *xend* primitives to enclose a critical region which ought to be executed transactionally. Memory addresses read and written within an RTM region constitute the read-set and write-set respectively. A conflicting access occurs if one RTM transaction (i.e., a running instance of an RTM region) has a read set that overlaps with another concurrent transaction's write set or if their write sets overlap. RTM provides strong atomicity [13]. If an RTM transaction conflicts with concurrent memory operations from other transactional or non-transactional code, the processor will abort the transaction. If an RTM transaction is aborted, all its writes will be discarded, and the program state will be rolled back to the beginning of the execution. Otherwise, all memory modifications within an RTM region will appear to happen atomically.

However, as a hardware mechanism, RTM provides no forward progress guarantee. Consequently, it is the programmers' responsibility to provide a fall-back handler when an RTM transaction retries a predefined threshold. In practice, the fall-back handler usually acquires a coarse-grained lock, all other transactionally executing threads eliding the same lock will abort, and the execution serializes on the lock [14]. Hence, the performance of an RTM transaction will fall back to a coarse-grained lock scheme with an additional cost of RTM aborts.

2.2 Concurrent Index Structures

An index structure is a data structure that facilitates fast data retrieval in a data management system [15]. Representative index structures include search trees [16], [17], [18], skip lists [19], and hash maps [2], [3]. With the expansion of user amount and data scale, index structures have to confront with highly-concurrent requests. To provide correct indexing with high performance, concurrent index structures have to resolve the data races with small synchronization overhead. Shrinking the granularity of synchronization point is a conventional technique to improve the performance of concurrent index structures. Many concurrent index structures attempt to achieve high performance by using fine-grained locks [20], [21], [22]. Such fine-grained synchronization unleashes the concurrency, while it increases the programming complexity. Moreover, programs with too many subtle locks can be deadlock-prone. With the availability of processors supporting atomic instructions, lock-free methods are also widely used to construct efficient concurrent index structures [23], [24], [25], [26], [27]. However, lock-free schemes are based on atomic instructions, which only make the program more complex and hard to reason about the correctness.

Since B+tree and skip list are two most widely used concurrent index structures [2], [19], [28], [29], we use them for case study in the following paper. A B+Tree is a B-Tree in which internal nodes store keys, and only leaves are associated with values [30] (Fig. 1a). When there is an access to B+tree, the request first traverses along the edges to locate the target leaf node. Then the request updates/gets the value. The node will split if the leaf node is full. The split will propagate upwards along the edges if the parent node is full too. Meanwhile, skip list is a layered data structure, and each layer is an ordered linked list (Fig. 1b). Each layer acts as an express lane for the lower one. Data are stored in the bottom layer. When there is an access to skip list, the request starts from the top layer and finds the largest node

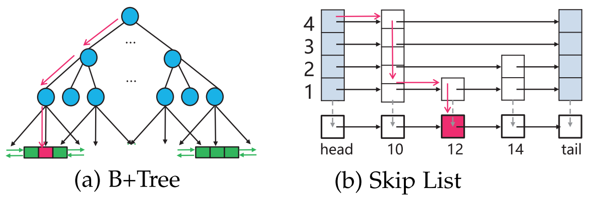


Fig. 1. Data structure of B+Tree and skip list.

which is smaller than the requested key. Then the request moves to the next layer and continues the search. Finally the request reaches the bottom layer at which moment it can decide whether the key exists.

2.3 HTM-Based Index Structures

With the commercial availability of HTM-capable processors, HTM-based concurrent data structures have drawn much attention from academia. HTM provides high performance while simplifying the programming paradigm, which makes it a perfect candidate for devising highly-concurrent index structures. Conventional HTM-based index structures protect the consistency of concurrent operations by adopting an HTM region at the boundary of the critical section. As researched in DBX [2], the first successful attempt to design HTM-based database, the programmers could substitute the complex lock-based synchronization with two simple HTM primitives: *xbegin* and *xend*. Such programming paradigm offloads the responsibility of guaranteeing the concurrent correctness from programmers to hardware architecture, and HTM-based index structures like B+Tree and skip list have been successfully used in many database designs [3], [4].

3 Motivation

HTM-based index structures have been pervasively used in many database systems, and have been proven to deliver scalable performance under mainstream workloads. However, with the expansion of data capacity and user amount, modern concurrent index structures have to confront with different kinds of contented workloads. Prior research has revealed that most concurrent index structures with software-based synchronization techniques (e.g., lock, atomic instruction) suffer varying degrees of performance slowdown when workloads exhibit high contention [7], [8], [9], [31]. In this section, we explore the performance behavior of HTM-based index structures under highly-contented workloads and analyze the reasons for the performance pathology.

3.1 Pathological Performance Under High Contention

Here we use two representative HTM-based concurrent index structures, i.e., B+Tree and skip list, derived from DBX [2] as examples to illustrate the issues of HTM-based concurrent index structure under contention. The HTM-based B+Tree (namely HTM-B+Tree) adopts HTM regions to protect operations of the B+Tree such as get, put, and delete. This design was later adopted and shown to be effective in other distributed in-memory databases [3], [4]. Besides, skip list [19] is a simpler alternative to balanced search trees; the HTM-based skip list is also widely adopted in many database systems [2].

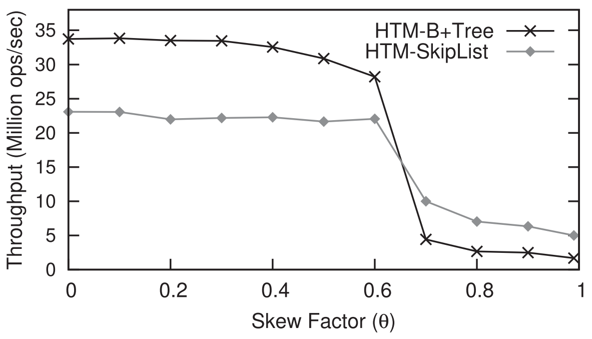


Fig. 2. Performance under different contention rates.

Conventional HTM-based index structures use a monolithic HTM region marked by *xbegin* and *xend* primitives to protect the entire data retrieval process; such a coarse-grained HTM region eliminates the complexity of maintaining fine-grained locks and makes it easy to reason about correctness. As a result, HTM-based index structures are shown to have much better performance compared to the state-of-the-art lock-based concurrent index structures (i.e., Masstree [21]) under low to modest contention [2].

While the HTM-based concurrent index structures deliver high performance under low and modest contention, the performance is found to exhibit dramatic collapse under high contention. To illustrate this, we evaluate the throughput of an HTM-based B+Tree and an HTM-based skip list using the YCSB benchmark with the Zipfian input distribution [6], [32]. The experimental setup is the same as that in Section 6.1. We set a skew factor θ to control the span of hot keys. Higher the θ, higher the probability that multiple threads access the same hot region in the key set. All the performance results are collected using 16 threads (a few cores are reserved for controlling threads). Threads are distributed equally on two sockets.

As the data in Fig. 2 show, with low contention rate (i.e., skew factor $\theta < 0.6$), the HTM-based B+Tree achieves high and stable performance. However, when the contention rate increases (e.g., $\theta > 0.6$), the performance of an HTM-based B+Tree shows a sharp collapse. When $\theta = 0.9$, the performance decreases to lower than 3 million ops/s. HTM-based skip list exhibits similar performance behavior.

To understand the underlying reasons behind the performance collapse, we collect the number of HTM aborts. Since adding performance counters to each HTM region severely hinders the overall throughput, here we set performance counters in every 10 operations so that the performance with HTM counters deviates little from that without counters. As the data in Fig. 3a show, the HTM abort rate of B+Tree increases sharply with the contention rate; the HTM abort rate for $\theta = 0.9$ is around 47X higher than that for $\theta = 0.5$. Similarly, the HTM abort rate of skip list also rises by 22X when the skew factor increases from 0 to 0.99 (Fig. 3b). The collected CPU cycles also show that frequent HTM aborts and retries waste more than 94 percent of the total CPU cycles under high contention.

3.2 Analysis of Design Defects

To understand the underlying reasons for the collapsed performance under high contention, we perform a detailed analysis and uncover three main sources of aborts.

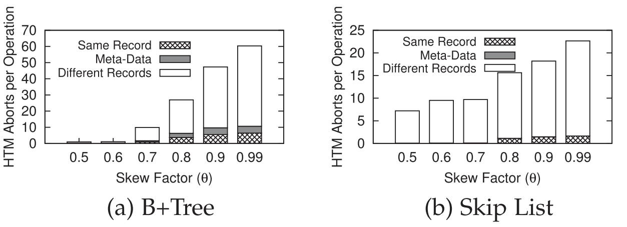


Fig. 3. HTM aborts incurred by different reasons.

High Retry Overhead. While using a monolithic transaction for the entire index structure provides consistency with trivial effort, it also increases the time consumed by retry operations. Our analysis finds that the distribution of conflicts is non-uniform in B+Tree and skip list: more than 90 percent of conflicts occur in the leaf level of B+Tree, and about 83 percent of conflicts occur at the lowest two levels of skip list. In this case, a conflict will abort the entire index traversal from beginning to end, even though there is actually no conflict in other parts of the index structure. According to the experimental results, such futile retry process takes up around 60 percent of the total execution time in the B+Tree and about 58 percent of the total execution time in the skip list.

False Conflicts. Are conflicts incurred by requests accessing different records. False conflicts stem from two primary reasons. The first one is *cache line sharing of consecutive records*. Consecutive and dense memory layout pervasively exists in ordered index structures. For example, B+Tree arranges keys within a node in a continuous manner to provide an ordered store, and skip list stores pointers to each node densely in an array. Such data layout causes severe conflicts under high contention. Since HTM detects conflicts at cache line granularity, concurrently accessing data in the same cache line would result in increased conflict rates within nodes. The second reason is *accessing the shared meta-data*. For instance, a conventional B+Tree inherently contains pervasive shared variables to maintain tree structure invariants (e.g., the number of layers and version number of nodes), and skip list stores the maximum height of a node in it. We categorize conflicts incurred by shared meta-data as false conflicts since their target records are actually different. Since it is difficult to measure the exact percentage of false conflicts directly, we approximate the decomposition by excluding other affected factors and estimating the abort rate. To estimate the impact of “same record”, we modified the Zipfian distributed workloads and to prevent different threads from accessing the same records and collect the reduction on HTM aborts. We calculate the HTM aborts from accessing different records (e.g., when inserting consecutive records) by subtracting previous rates from the total rate. As to shared meta-data, it is hard to remove all shared variables in the tree structure as some are indispensable to proceed with execution. We remove shared variables for version and node status, and estimate the reduction of aborts. As shown in Fig. 3a, 87-90 percent of conflicts in B+Tree are caused by requests to different keys, which is the primary reason for the growth of abort rate. Besides, the conflicts incurred by shared meta-data contribute 6-10 percent, which is also a non-negligible source. As shown in Fig. 3b, for HTM-based skip list, conflicts incurred by shared meta-data (e.g., max height of nodes) can be neglected, while conflicts incurred by different records contribute 92-99 percent. It is also notable that the abort rates of B+Tree and skip list are different. The reason behind it is the difference in their organization. The records in B+Tree’s leaf nodes are stored in a consecutive manner. On the other hand, the skip list stores pointers to each node densely in an array, but each node only holds one record individually. For these two data structures, the modification operations mainly occur in records. In B+Tree, the consecutively-stored records incur both true and false conflicts frequently due to cache line sharing and shared meta-data; while in skip list, modifications to a single record will not affect the sibling records directly since they are not stored consecutively in the same cache line. Therefore, the abort rate of B+Tree is higher than that of skip list.

True Conflicts. Are conflicts incurred by requests accessing exactly the same record. For workloads under high contention, the probability that multiple requests access the same record simultaneously is inherently high, which is a significant source of conflict. From Fig. 3, we can observe that 9-12 percent of conflicts are incurred by requests to the same records in B+Tree, and 1-7 percent in skip list.

4 Eunomia Design Pattern

Based on the above analysis, this section first summarizes some common features in concurrent index structures, which are the primary reasons for the performance slowdown under contention. Then we propose the Eunomia design pattern, which comprises four design guidelines to scale concurrent index structures under contention using HTM.

4.1 Features of Concurrent Index Structures

The analysis in Section 3 reveals several defects of HTM-based concurrent index structures. Such design defects stem from the gap between HTM semantics and some common features of concurrent index structures. To fully take advantage of HTM mechanism, it is necessary to understand the common features of concurrent index structures, which are the basis of our design pattern.

Multiple Phases During Data Access. Most concurrent index structures improve the efficiency of data searching at the cost of additional storage space to maintain the index data structure. For example, B+Tree maintains hierarchical inner nodes to reduce the time complexity of data locating; skip list stores keys in multiple levels to reduce the hops in searching a particular key. Therefore, the index retrieval process of those concurrent index structures shares a common pattern: the request entering the data structure first traverses along the index edges to search the target key, then it performs modifications to the target key according to different semantics. Such data access can be divided into an *index phase* and an *operation phase*. Taking a concurrent B+Tree index as an instance, the request first traverses along the tree edges to locate the target leaf node in index phase. After the request finds the target leaf node, it modifies the node and updates/inserts new value in operation phase.

Uneven Modification Distribution. As discussed above, in index phase, requests mainly traverse along index structure; few modifications occur in this phase. In operation phase, requests update the records and modify the index structure accordingly; most modifications take place in this phase. Therefore the boundary between the two phases divides the

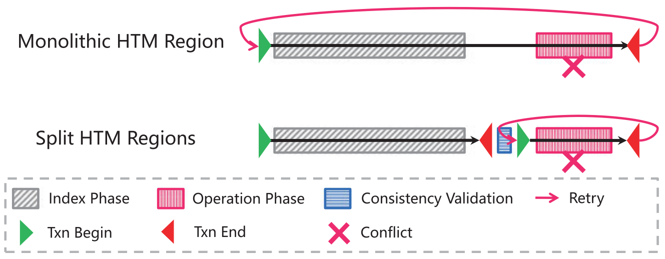


Fig. 4. The split HTM transactions.

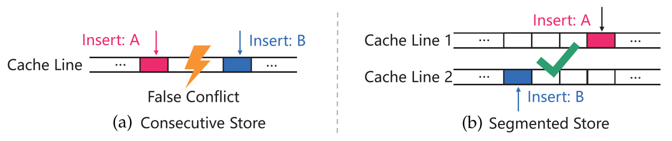


Fig. 5. The segmented data layout.

concurrent index structures into a modification-sparse part and a modification-intensive part. The majority of data modifications actually occur in the operation phase. When protecting the concurrent index structure by HTM, the uneven distribution of data modification leads to the uneven distribution of conflicts; that is why 90 percent of conflicts occur in the leaf layer of B+Tree, and over 80 percent of conflicts occur at the bottom two levels of skip list. Conventional HTM-based concurrent index structures protect the whole access process within one monolithic HTM region. In such a manner, a request needs to traverse the concurrent index structure from the root node for each retry, even most conflicts occurred in the operation phase, which leads to high retry overhead.

Ordered and Consecutive Data Layout. Most concurrent index structures store records in a dense and consecutive manner due to the following reasons: first, guaranteeing the efficiency of range query operations which return a sequence of sorted keys; second, enabling fast data retrieval such as binary search; third, reducing the memory consumption. For example, B+Tree stores keys consecutively in leaf nodes, and skip list arranges pointers to nodes in different levels in an array. These ordered data (such as leaf nodes in B+Tree) are often modified in the operation phase of concurrent index structures. When such data structure are combined with HTM, which detects conflicts in cache line granularity, it will incur high likelihood of false conflicts.

4.2 Eunomia Design Principles

Based on the performance issues and features analyzed above, we advocate Eunomia, a design pattern for scaling concurrent index structures using HTM under highly-contented workloads. Eunomia is constituted with the following principles.

Splitting HTM Transactions with Consistency Validation. There exist multiple phases during request traversal, and the HTM-protected index structure can be divided into a conflict-sparse index phase and a conflict-intensive operation phase. Traditional HTM-based index structures use a large monolithic transaction region to protect the consistency of the entire data retrieval with little programming complexity. Consequently, a conflict in operation phase incurs retry throughout the index structure, leading to high retry overhead. To solve this problem, an intuitive way is to decompose a large HTM transaction into multiple smaller HTM transactions. Since data conflicts distribute unevenly in concurrent index structures, the split boundary ought to be decided by the likelihood of data conflicts so that retries incurred by conflicts in conflict-intensive regions will not impact the requests in conflict-sparse regions. For example, in concurrent B+Trees, the delimiter between index phase and operation phase is between the inner layers and the leaf layer. Therefore, if the monolithic HTM region is divided into two parts, the boundary ought to be set above the leaf layer so that the conflicts happened in operation phase only incur retries in this stage, instead of causing futile retry operation throughout the entire index structure, as depicted in Fig. 4. However, naively splitting the HTM region no longer guarantees the atomicity and consistency of the entire data access operation. To solve this problem, Eunomia uses a version-based opportunistic consistency validation to detect and avoid potential inconsistency problems. The detailed implementation is introduced in Section 5.

Partitioning Data Layout. Consecutive and dense memory layout incurs high false conflict rate when multiple threads access the data in the same cache line. To address this issue, Eunomia fpartitions the continuous records into multiple segments and requests to adjacent records will be randomly distributed to different segments located in different cache lines as shown in Fig. 5. Since requests to adjacent data are scattered to different segments randomly, the false conflict rate can be reduced (Fig. 5). For operations requiring ordered data (e.g., range query), Eunomia uses buffers to store sorted keys gathered from multiple segments temporarily. Hence, the original ordering semantics can still be maintained. The detailed implementation is introduced in Section 5.

Conflict Control Module (CCM). The conflicts distribute unevenly in the index data structure, and most conflicts actually occur in the modification-dominated operation phase. To prevent multiple threads from colliding in operation phase, we propose conflict control module at the boundary between index phase and operation phase. CCM has two main functions: detecting/avoiding potential true conflicts, and throttling the number of threads entering the operation phase. Especially, CCM adopts two main techniques (Fig. 6). First, CCM employs fine-grained advisory locks to detect and serialize all requests accessing the same key. Thus potential true conflicts are eliminated. Second, CCM adopts a Bloom filter [33] based mechanism to prevent requests searching for inexistent keys from entering the operation phase, so as to constrain the number of threads in operation phase.

Adaptive Contention Control Strategy. While the above design guidelines are helpful for a high contention scenario, applications usually exhibit changing workloads with different levels of contention. As the design choices to handle high contention may bring overhead under low contention, Eunomia uses an adaptive contention control strategy to detect contention rates and bypass extra cost when contention is low.

5 Case Study: Euno-B+Tree

In this part, we illustrate how to apply Eunomia design patterns to implement concrete HTM-based index structures. The baseline is a traditional HTM-based B+Tree, which has

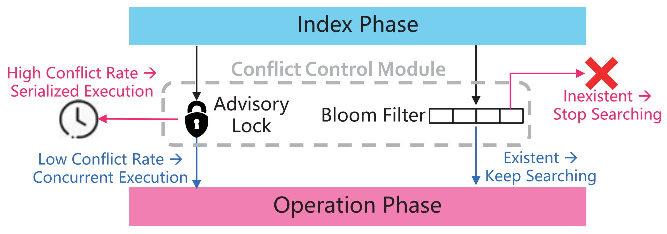

Index Phase
Conflict Control Module
High Conflict Rate → Serialized Execution
Advisory Lock
Bloom Filter
Inexistent → Stop Searching
Low Conflict Rate → Concurrent Execution
Existent → Keep Searching
Operation Phase

Fig. 6. Design of conflict control module.

been widely used in many systems [2], [3], [4]. We apply each Eunomia principles to the baseline and implement a scalable HTM-based B+Tree (namely Euno-B+Tree) under contention.

5.1 Data Structure Design

Splitting HTM Transactions: Reducing Retry Cost. To reduce the retry cost, Eunomia partitions the monolithic HTM region at the delimiter between index phase and operation phase. For B+Tree, the split boundary ought to be above the bottom leaf layer. Therefore we decompose the HTM region of HTM-based B+Tree as shown in Fig. 7. Euno-B+Tree splits the original HTM region into two individual parts: *upper region*, which protects the atomicity of internal nodes traversing, the read-dominated index phase; and *lower region*, which protects the atomicity of leaf nodes accessing, the conflict-prone operation phase.

Trivially splitting the monolithic HTM region introduces inconsistency issue when node splits: consider if a thread tries to insert record A, while a concurrent thread tries to read record B in the same leaf node. The read request will get the leaf node pointer by traversing the tree index in the upper region. However, before it enters (or retries) the lower region, the leaf node may be split due to the concurrent insertion and record B is moved to the sibling leaf node. The read request will fail to get record B. The problem is the read request gets the leaf node pointer in one HTM region, while it searches the leaf node in another HTM region. As a result, the read request is not aware of concurrent splitting.

To solve the problem, Euno-B+Tree adopts a version-based consistency validation approach. The overall consistency is guaranteed by a version number tracking the split operation (Fig. 8). The version number is created with the leaf node, and is updated when the leaf node splits. At the end of the upper region, the request finds a pointer to the target leaf node, and reads the version number of this node into a local variable before exiting the upper region. When the request enters the lower region, the version number will be checked. An inconsistency means

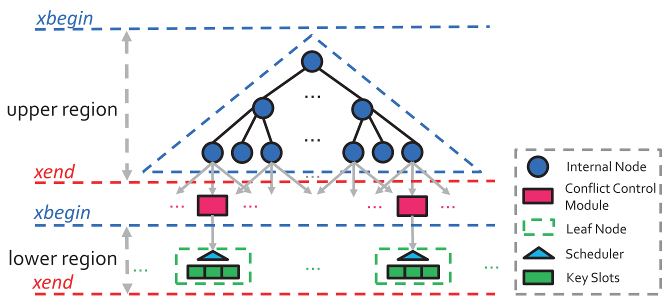


Fig. 7. Overview of Euno-B+Tree structure.

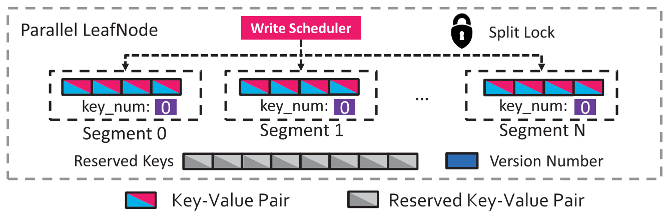


Fig. 8. The data layout of leaf nodes.

the target leaf node has been split. Therefore, the request needs to retry from the root node. For the above example, after the read request gets the leaf node pointer in the upper region, it snapshots its version into a local variable. When a leaf node is split, its version is updated. Then the read request will be aware of the split event by checking the version number at the beginning of accessing the lower region or a new retry.

Scattered Leaf Nodes: Reducing False Conflicts. To reduce the false conflicts incurred by consecutive and dense memory layout, Eunomia stored records in a scattered manner. In B+Tree, the dense and consecutive memory layout mainly exists in leaf layer, which is also located in operation phase, exacerbating the false conflict rate. Therefore, Euno-B+Tree redesigns the leaf layer in a scattered manner to reduce the likelihood of false conflict (Fig. 8). Each leaf node is separated into multiple segments located in different cache lines. Only the keys in the same segment are kept sorted and stored consecutively, and the value pointers are combined with keys for the convenience of sorting and reorganization. Besides, each segment has its meta-data to record the number of stored elements, which naturally splits the shared meta-data, another source of false conflicts. To avoid the conflict in the same segment, we use a *write scheduler* to assign each *put* operation to a random segment. Each leaf node maintains a lock (split lock) to serialize concurrent split operations because concurrent split operations on the same leaf node are highly likely to conflict with each other.

Also, we allocate *reserved keys* in each leaf node as a buffer to hold the sorted keys for operations requiring ordered results (e.g., range query). The reserved keys have the following functions.

- Storing old records for a node split. If a node is split, the *reserved keys* in each split node will be used to store old records. The length of reserved keys equals the number of records inherited from the original node.
- Storing records overflowed from segments. To make room for further concurrent insertions, if an insertion overflows a segment, the *reserved keys* will be dynamically expanded, and all records in segments will be moved to it.
- Storing sorted records for range queries. All of records in segments will be sorted and moved to reserved keys when there is a range query operation.

When *reserved keys* are expanded, the total size of segments is shrunk to keep that the size of all segments plus the size of reserved keys equals the size of the original space in a node. The records in *reserved keys* could also be searched and updated. However, the partitioned segments and *reserved keys* work together to scatter different operations, which reduces the probability of false sharing. Such a design sacrifices the performance of search operations. However, since each segment and *reserved keys* are already sorted, performing a merge sort is quick, and the sorted results can be reused for the following search operations.

Conflict Control Module: Reducing and Constraining Conflicts. To avoid potential true conflicts and throttle the number of threads entering operation phase, CCM is added at the boundary between index phase and operation phase. In B+Tree, we add the CCM above each leaf node, which is the delimiter between index phase and operation phase. The first function of CCM is to detect and eliminate potential true conflicts. Thus it uses fine-grained atomic advisory locks to prevent two conflicting operations from accessing the same record simultaneously. Then CCM adopts a Bloom filter based mechanism to constrain the number of requests entering the operation phase. Specifically, CCM has a hash function and two bit vectors: mark bits and lock bits.

- *Hash function.* The target key of a request will be hashed to a bit in the vector.
- *Lock bits.* The lock bits function as fine-grained atomic advisory locks attached to each slot in the leaf node. It detects and serializes all concurrent requests accessing the same key. It can avoid conflicting operations (i.e., put versus get and put versus put) to enter the HTM region simultaneously.
- *Mark bits.* Besides, the mark bits vector is used to indicate the existence of the corresponding key, resembling the working mechanism of Bloom filter. If a request searches for an inexistent key, the mark bits will prevent it from entering the leaf node; thus fewer threads could enter the leaf node, and conflict rate is reduced further.

We set the length of the bit vector twice as the leaf node's fanout so that the space overhead is kept below 5 percent while the false positive rate is kept under 6 percent.

Adaptive Concurrency Control. To reduce the overhead for low contention rate, we use a *contention detector* in CCM, which detects the contention rate and adjust the contention control strategy accordingly. First, the detector predicts the conflict probability of a leaf node based on historical data. When the conflict rate of a leaf node keeps below a threshold for a period, its contention rate is considered to be low. In such a condition, the new incoming requests will bypass the CCM and split locks, skipping the overhead brought by these modules.

5.2 Algorithms

Based on the above data structure design, we further relate how Euno-B+Tree handles common B+Tree operations.

Get/Put. Algorithm 1 shows the traversal procedure shared by both *get* and *put* operations. Due to the split HTM regions, the traversal procedure is naturally divided into two parts. The atomicity of each part is protected by an HTM region, one for the upper region (Lines 23-28) and one for the lower region (Lines 41-51) accordingly. We omit the fall-back path here for brevity. The CCM is used to prevent conflicting requests from entering the lower region simultaneously (Lines 29-40).

Algorithm 1. Get/Put (Two-Step Tree Traversal)

```
21: procedure Traverse(REQ_TYPE, key, newVal)
22: RETRY:
23: XBEGIN() //upper region
24:     node = root
25:     leaf = findLeaf(node, key)
26:     local_seqno = leaf.seqno
27:     ccm = leaf.CCModule //get the conflict control module
28: XEND()
29:     slot = ccm.hash(key)
30:     while !CAS(ccm[slot].lockBit, 0, 1) or leaf.isLocked() do
31:         spin()
32:     exist = ccm[slot].markBit
33:     if !exist then
34:         if REQ_TYPE == GET then
35:             record = null
36:         else //REQ_TYPE == PUT
37:             //insert the key if it does not exist
38:             insert = CAS(ccm[slot].markBit, 0, 1)
39:             if insert and leaf.isNearFull() then
40:                 leaf.lock() //hold the lock for split
41: XBEGIN() //lower region
42:     if local_seqno != leaf.seqno then
43:         consistent = false //inconsistency happens
44:     else
45:         if exist then
46:             record = leaf.getRecord(key)
47:         if REQ_TYPE == PUT then
48:             if record == null then
49:                 record = INSERT(leaf, key)
50:             record.value = newVal
51: XEND()
52:     if leaf.isLocked() then
53:         leaf.unlock()
54:     ccm[slot].lockBit = 0
55:     if !consistent then
56:         goto RETRY
57:     if REQ_TYPE == GET then
58:         return record
```

To process a put or a get request:

1) Euno-B+Tree first traverses along the tree edges from the root to reach the leaf node (Lines 23-28). The atomicity of the traversal is protected by an HTM region. Before exiting the HTM region, the request reads the current version number into a local variable (Line 26).
2) Before entering the lower region, Euno-B+Tree uses the CCM to serialize conflicting accesses on the same node (Lines 29-40). This is done by atomically checking and setting the lock bit of the corresponding key (Line 30). We use a set of atomic advisory locks to protect the atomicity of each byte in the bit vector. After a request sets the lock bit successfully, it will validate if its target key exists or not by checking the mark bit (Line 32). If it does not exist, for a *get* request, it will assign a *null* value to the record; for a *put* request, it will set its bit in the mark bits vector and initiate an insert procedure. For an insertion operation, if the leaf node needs to be split due to the

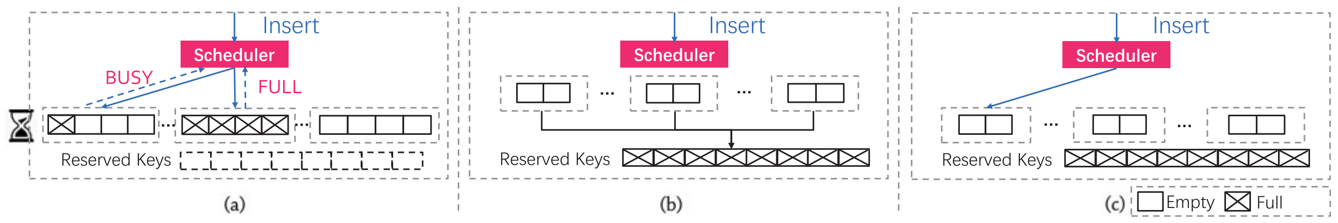


Fig. 9. Concurrent insertions to a leaf node.

capacity limitation, it will try to acquire the split lock before splitting (Line 40). The subsequent operations will be blocked by the lock from entering a node in the split process (Line 30).

3) Scanning the leaf node is protected in the *lower region* (Lines 41-51). In the beginning, it needs to check the version number of the leaf node. If it has been changed, it means the node was split by a concurrent request before entering or retrying the *lower region* (Line 43). In this case, a request in the *lower region* could scan the wrong leaf node. Thus it ought to retry from the root and search for the latest proper leaf node. Otherwise, the leaf node is still valid; the request will continue searching for the target key in this leaf node (Line 46). If the target key is not found in a leaf node, the put request will insert a new key (Line 49); otherwise, it will update the value of an existing key (Line 50).

Insert. Based on the partitioned leaf nodes introduced in Section 5.1, we propose the algorithm of insertion; Fig. 9 illustrates necessary steps.

1) Before an insertion request enters the leaf node, it first checks the CCM to detect potential true conflicts. The hash function in CCM maps the target key to a certain bit in the lock_bits. Then the request will be blocked if the target bit is being locked. Otherwise, it will set the target bit in lock_bits and mark_-bits until it finishes the insertion.
2) The write scheduler randomly distributes incoming requests to different segments. If all insertions are distributed to different available segments (segments with empty slots), then multiple insertions can be processed concurrently. If the target segment is full, the scheduler will retry the distribution attempt (Fig. 9a).
3) If the retry times exceed a threshold, then we can infer the leaf node is near-full or the key-value pairs stored in segments distribute unevenly. In this case, we move the elements in all segments to *reserved keys* (Fig. 9b), then clean the segments to accommodate new concurrent insertions. If the retries are incurred by uneven key-value distribution, after we reorganize the keys to *reserved keys*, there could remain sufficient room in segments to support further concurrent insertions.
   a) If there is still sufficient room to hold new keys (Fig. 9c), the scheduler continues to distribute concurrent insertions to different segments.
   b) If there is not sufficient room for new keys, the split lock is acquired for a further split operation.

By this means, the leaf node allocates concurrent insertions to different segments, and the shared meta-data are also naturally divided into several parts.

Split. When a node in B+Tree is full, a new insertion will trigger a split. The current node is split into two nodes with original keys evenly distributed. The content of the parent node will be adjusted accordingly, and the split propagates upwards if the parent nodes are full themselves. Given the insertion semantics related in the previous section, the keys in leaf nodes of Euno-B+Tree are arranged in a partially unordered manner (ordered within a segment, unordered among segments). Therefore splitting a node, in fact, includes both sorting and splitting steps.

1) When a node splits, Euno-B+Tree first locks the leaf node to block new incoming insertions; otherwise, they are highly likely to conflict when the keys are being reshuffled.
2) Then the keys in the original node are sorted and stashed to *reserved keys* in an ordered manner.
3) The original node is split according to the typical scheme of a B+Tree. The version number increases to keep consistency. The smallest key of the right leaf node is inserted into the parent node to make the newly-born node indexable.
4) In new nodes, old key-value pairs inherited from the old node are stored in *reserved keys*, and the remaining empty slots are evenly distributed to multiple segments. So that segments will be entirely empty to accommodate new records.

In such a *sorting-split-reorganizing* way, we can constrain the randomness of keys within the leaf node layer. The keys in the internal nodes are still stored in order.

5.3 Range Query and Deletion

Range queries, which access a set of consecutive keys, are an important interface for ordered indexes. In Euno-B+Tree, when a range query request reaches a leaf node, the node will be locked by the split lock, and key-value pairs stored in all segments will be moved and sorted in *reserved keys*. Hence, the scan iterator can get a sequence of ordered keys. Besides, the traversal process of a deletion is similar to that of a put operation; the tree structure simply labels the status of the record as deleted and clears the corresponding mark bits. Euno-B+Tree reuses the deletion and garbage collection scheme in DBX [2] to clean up the unused nodes. Instead of re-balancing the tree on every deletion instantly, we do the re-balance when the number of deletions exceeds a threshold. Previous research has proved that such a re-balance scheme has theoretical and empirical advantages [34], and has been adopted by many prior systems [2], [21].

6 Data Structure Evaluation

According to Eunomia design pattern, we have implemented an HTM-based B+Tree (namely Euno-B+Tree) and an HTM-based skip list (namely Euno-SkipList). Each of them only consumes less than 600 lines of C/C++ code. By evaluating the implementation, we try to answer the following questions:

- Does Eunomia solve all the challenges discussed in Section 3?
- Can Eunomia deliver better performance than a fine-grained locking schemes even under high contention?
- Can Eunomia achieve performance scalability under different contention levels?
- How does each design choice affect the performance?
- What is the memory overhead of Eunomia design pattern?
- Can the real applications (e.g., Database) take advantage of Eunomia (Section 7)?

6.1 Experimental Setup

All experiments were conducted on a 20-core server (two 2.30 GHz 10-core Intel® Xeon® E5-2650 chips) running Linux 3.19.0. Each core has private 32 KB L1 data cache, 32 KB L1 instruction cache, and 256 KB L2 cache. Each chip has a shared 25 MB L3 cache. The cache line size is 64 bytes. The total DRAM size is 256 GB. We use Intel's RTM to implement atomic regions.

We compare Euno-B+Tree with different concurrent B+Tree implementations:

1) An HTM-based B+Tree (namely HTM-B+Tree) adopted by many database systems [2], [3], [4], which uses HTM regions to protect the atomicity of the entire operation. We reuse the fall-back strategy and retry policy in DBX [2].
2) A highly optimized concurrent B+Tree implementation derived from Masstree [21]. It uses fine-grained locks to achieve good scalability. However, we still use the term "Masstree" for simplicity.
3) An HTM version of Masstree (namely HTM-Masstree). It uses HTM region to protect the entire Masstree operation (as in (1)), subsuming multiple elided locks.

We compare Euno-SkipList with different concurrent skip list implementations:

1) An HTM-based skip list adopted by many database systems [2], [3], which uses HTM to protect the atomicity of the entire operation.
2) A fine-grained lock skip list using the strategy proposed by Pugh [35], which utilizes a useful property to simplify the implementation that the distribution of levels within a skip list does not affect correctness (denoted as FGLock-SkipList).
3) A lock-free skip list using the strategy proposed by Fraser [36], which utilizes CAS instructions to maintain consistency while preserving high concurrency (denoted as LockFree-SkipList).

We here adopt the Yahoo! Cloud Serving Benchmark (YCSB) [37] for evaluating Euno-B+Tree, which is a representative benchmark for large-scale key-value storage. For each record, both key and value have 8 bytes fixed size. The get/put ratio is set to the default value of 50%/50%. For the B+Tree, we set the node fanout to 16. The average tree depth is 6, and run duration is set longer than 20 seconds to get stable performance. Unless otherwise specified, we use

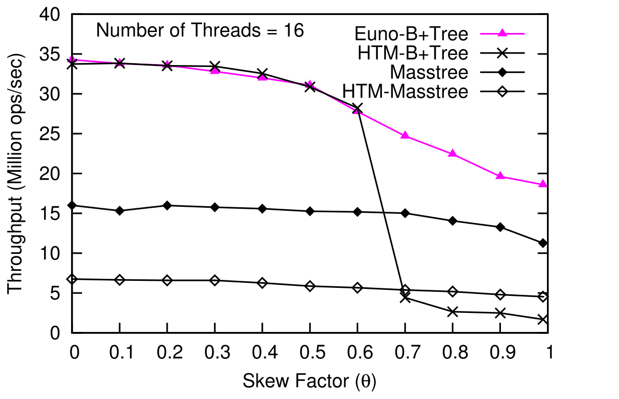


Fig. 10. Throughput of concurrent B+Trees under different contention rates.

Zipfian as the default input distribution for B+Tree, private to each thread (intra-thread locality). The Zipfian distribution has a skew factor θ, and the probability of accessing a key k is given by

$$P(k) \propto \left(\frac{1}{k}\right)^{\theta}. \tag{1}$$

Thus, we can easily increase the contention rate by increasing θ. With $\theta = 0$, all records are accessed with the same probability (uniform distribution); with $\theta = 0.99$, the "hottest" tenth of the values in the set are accessed by 41 percent of the requests.

For the skip list, we set the height to 16. The size of record is the same with B+Tree. Since the original Zipfian distribution does not provide enough contention for skip list, we design a new input distribution with a skew factor θ to control the contention rate. Higher the θ, higher the probability that multiple threads access the same hot region in the key set.

6.2 Throughput

To see whether Eunomia solves the issues discussed in Section 3, we initially repeat the experiment in Fig. 2. The throughput of B+Tree is shown in Fig. 10. When the contention is low or even modest ($\theta < 0.6$), Euno-B+Tree can obtain similar performance as HTM-B+Tree. This is because Euno-B+Tree uses adaptive concurrency control to reduce most overhead under low contention. Meanwhile, the throughput of Euno-B+Tree is about 36.85 percent higher than Masstree since Masstree's fine-grained synchronization needs to execute additional instructions. According to our analysis, when $\theta = 0.5$, the number of instructions executed by Masstree is about 2.10X that of Euno-B+Tree. The extra instructions primarily come from the "before-and-after" version checking mechanism in Masstree (Section 4.6 of [21]). For example, when $\theta = 0.5$, a put operation in Masstree needs on average to check and manipulate a version number about 15 times while traversing the tree. Under high contention ($\theta > 0.6$), Euno-B+Tree can achieve 11X speedup over HTM-B+Tree (18.6 M versus 1.7 M Ops/s with $\theta = 0.99$). This is because Eunomia eliminates most aborts compared with HTM-B+Tree (Fig. 11): 60.3 versus 1.9 aborts per Op under extremely high contention. Besides, compared with Masstree, it has 65 percent better performance even under high contention. This is because Euno-B+Tree executes around 40 percent fewer instructions under high contention. On the other hand, the performance

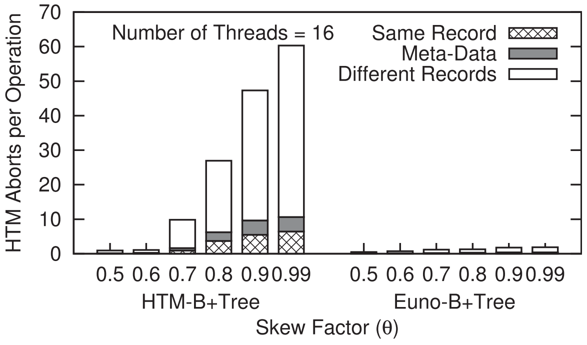


Fig. 11. Comparison of HTM aborts in concurrent B+Trees incurred by different reasons.

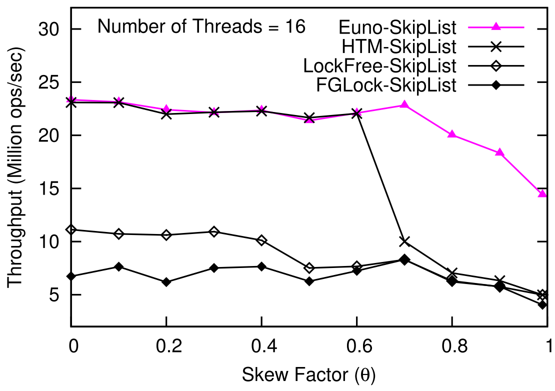


Fig. 12. Throughput of concurrent skip lists under different contention rates.

of the HTM-Masstree is worse than Masstree under both low and high contention. This is because the HTM-based Masstree has more shared variables than that of B+Tree, e.g., bits indicating whether the node is being split/inserted; its performance is low even under small skew factor and drops little with the increase of contention factor. This shows that, even for a highly optimized concurrent B+Tree, it is still hard to directly take advantage of HTM.

Furthermore, the throughput of skip list is shown in Fig. 12. Our evaluation illustrates that under low contention, Euno-SkipList outperforms FGLock-SkipList and LockFree-SkipList by about 3.47X and 2.10X. The FGLock-SkipList is a highly-optimized implementation using fine-grained locks, while LockFree-SkipList uses CAS instructions to maintain atomicity. However, in our workload, FGLock-SkipList exhibits a relatively low performance. It is because the batch of keys processed by a thread interleave with each other, making it more likely for a thread to be blocked. Meanwhile, the HTM-based implementation can take advantage of higher concurrency, leading to fewer instructions and simpler execution logic. Under low contention, Euno-SkipList does not show lower performance than HTM-SkipList, which indicates that Eunomia design does not incur much overhead to HTM-SkipList. As θ grows from 0.7 to 0.99, the performance of HTM-SkipList drops dramatically, while the performance of Euno-SkipList remains relatively stable. Under high contention, Euno-SkipList can obtain performance about 2.89X higher than HTM-SkipList. This is because Euno-SkipList eliminates most aborts compared to HTM-SkipList (Fig. 13). According to our analysis, under high contention, Euno-SkipList reduces about 95 percent aborts compared to HTM-SkipList.

6.3 Scalability

Besides the skew factor, an increase in threads number also intensifies the contention rate. Here we evaluate scalability with increasing threads under different levels of contention. As to Euno-B+Tree, we set θ by referencing previous research [5], [38]: 0.2 to simulate low contention; 0.6 for modest contention; 0.9 for high contention. We also set θ to 0.99 to simulate extremely high contention. Fig. 14 shows the results. Owing to the adaptive control, under low contention (Fig. 14a), Euno-B+Tree scales smoothly and is very close to HTM-B+Tree. This indicates that the adaptive control can reduce most performance cost under low contention. However, since Masstree needs to execute more instructions for synchronization (40 percent more instructions per thread), this overhead is amplified by adding more threads. As a result, Eunomia is 52-63 percent better than Masstree under high contention. HTM-Masstree fails to scale after eight cores under low contention. Under modest contention (Fig. 14b), the performance of HTM-B+Tree begins to collapse after four threads due to the increase in abort rate. Masstree still has stable performance as false conflicts in Masstree is not as severe as that in HTM-based implementations. Under high or even extreme contention (Figs. 14c and 14d), Euno-B+Tree still has reasonable scalability and performs better than Masstree (21.9 M versus 13.1 M Ops/s with 20 threads) for extremely high contention. This benefit is still from the fact that HTM simplifies the algorithm which makes it execute fewer instructions than Masstree.

For skip list, we evaluate four conditions, namely low contention (θ is 0), medium contention (θ is 0.5), high contention (θ is 0.7) and extremely high contention (θ is 0.99). Fig. 15 shows the results. Under low contention, both Euno-SkipList and HTM-SkipList scale smoothly. Under modest contention, the scalability of Euno-SkipList is barely affected. However, under high contention, HTM-SkipList fails to scale after eight cores due to the data conflicts brought by monolithic HTM region and consecutive memory layout. On the other hand, Euno-SkipList still scales smoothly even under extremely high contention, while HTM-SkipList, FGLock-SkipList and LockFree-SkipList fail to scale after four cores. Both HTM-SkipList and Euno-SkipList outperforms FGLock-SkipList and LockFree-SkipList under each circumstance due to more concurrency achieved by leveraging HTM. Compared to FGLock-

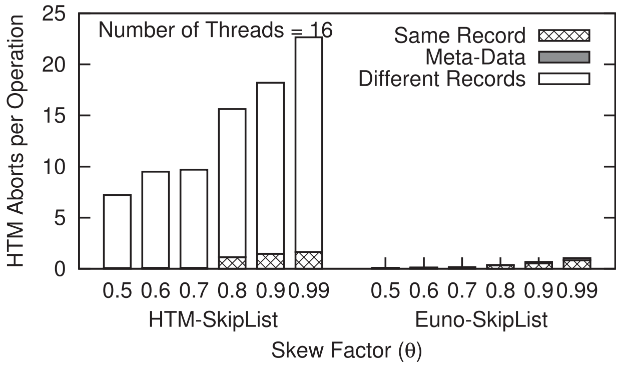
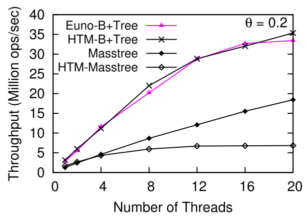
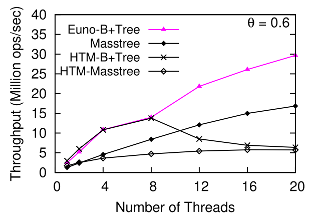


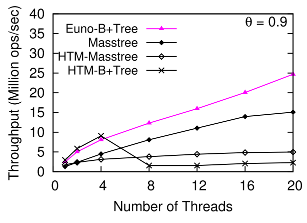
Fig. 13. Comparison of HTM aborts in concurrent skip lists incurred by different reasons.



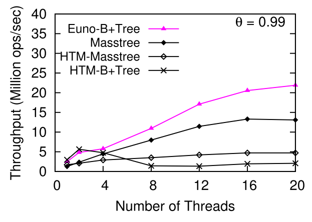
(a) Low Contention



(b) Modest Contention



(c) High Contention



(d) Extremely High Contention

Fig. 14. Performance scalability under different contention levels.

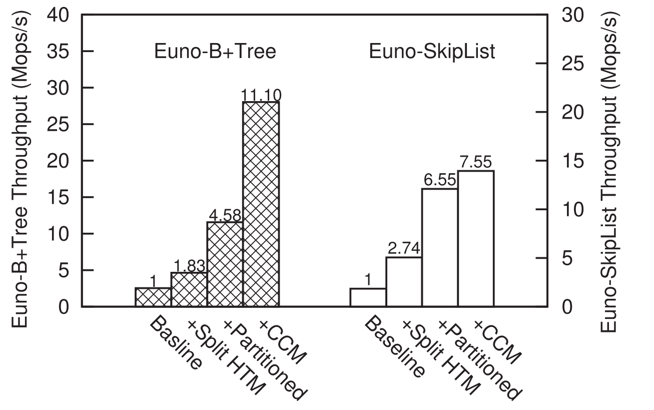


Fig. 16. Impact of different design choices. The relative performance is labeled on the top of each column.

SkipList and LockFree-SkipList, implementations based on HTM show higher performance under different circumstances.

6.4 Impact of Different Design Choices

To understand the performance gain and cost from various design aspects, we here present an analysis of multiple factors in Fig. 16. The benchmark is the YCSB with Zipfian input distribution with 20 threads under high contention (θ is 0.9).

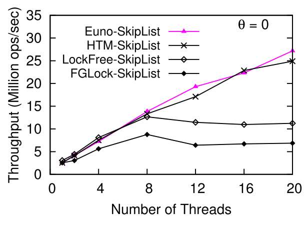
Baseline refers to an HTM-based index structure using a single HTM region to protect the entire operation. *+Split HTM* means splitting the monolithic HTM region into parts and using version numbers to protect consistency. *+Partitioned* refers to partitioning the data layout to avoid false conflicts in B+Tree and skip list. *+CCM* refers to adopting the CCM. The speedups are generated from continuously accumulating these design choices.

From the results, we can observe that all design choices bring substantial speedup to an HTM-based B+Tree. First, splitting the monolithic HTM region gets 1.83X speedup under high contention. The split HTM region not only reduces the overhead of retries, but also lowers the possibility of data conflicts since by shrinking the HTM execution time. Second, partitioned leaf nodes (*+Partitioned*) generates 4.58X speedup under high contention, as it decreases the false conflict rate caused by accessing the same leaf node. By further reducing true and false conflicts, conflict control module (*+CCM*) gets 11.10X speedup under high contention. In the case of skip list, splitting the monolithic HTM region (*+Split HTM*) gets 2.74X speedup under high contention. The speedup derives from the elimination of the time wasted in transaction retry. The employment of partitioned segments (*+Partitioned*) reduces most false conflicts, and introduces 6.55X performance gain. Then, the CCM (*+CCM*) brings 7.55X speedup since it reduces the probability of both true and false conflicts.

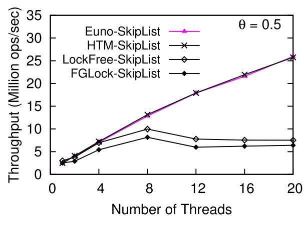
Furthermore, we collect the reduction on execution time brought by different design choices in B+Tree and skip list (Fig. 17). The multiple design choices are also added accumulatively. Splitting the monolithic HTM region (*+Split HTM*) contributes 50 percent of the execution time reduction in B+Tree and 70 percent of the execution time reduction in skip list. Split HTM region is the first design choice to apply, and it significantly reduces the original execution time by reducing both the retry overhead and conflict probability. Partitioned data structure (*+Partitioned*) contributes 25-35 percent of the execution time reduction. Conflict control module (*+CCM*) is the last design choice to apply. It further brings about 15 percent of the original time reduction for B+Tree. The impact of conflict control module in skip list is not significant since the conflicts in skip list are largely resolved by the former two design choices.

6.5 Memory Consumption Analysis

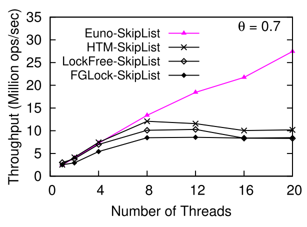
In this section, we take Euno-B+Tree and Euno-SkipList as examples to examine the memory overhead brought by Eunomia. Two structures in Euno-B+Tree would involve additional memory consumption: reserved keys and the conflict control module. The conflict control module consists of two-bit vectors for each leaf node and its memory consumption is negligible. Therefore, the main memory overhead comes from reserved keys. For Euno-SkipList, only the segments structure would involve additional memory consumption. Here, we evaluate the memory consumption overhead of Euno-B+Tree and Euno-SkipList using Valgrind



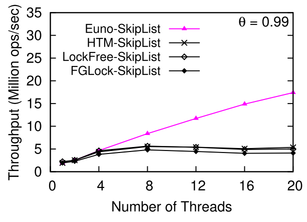
(a) Low Contention



(b) Modest Contention



(c) High Contention



(d) Extremely High Contention

Fig. 15. Performance scalability under different contention levels.

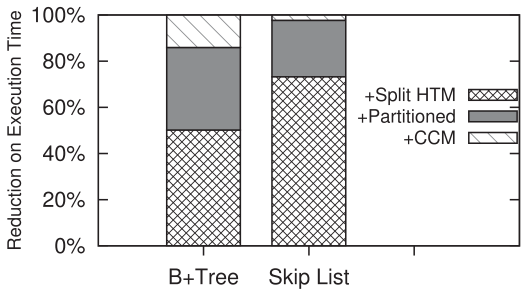
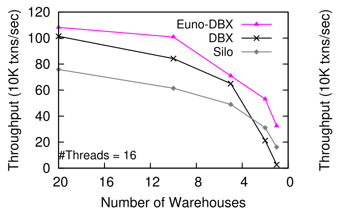
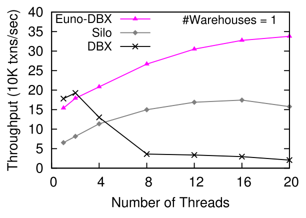


Fig. 17. Reduction on the execution time brought by each design choice.



(a) Throughput with decreasing warehouse number.



(b) Throughput with increasing thread number.

Fig. 18. TPC-C benchmark with standard mix. We adjust the number of warehouses and threads to control the contention rate.

toolset [39]. The workload includes getting and putting 10 million keys in a Zipfian distribution with 16 threads. The node fanout is set to 16 for Euno-B+Tree, and run duration is set to longer than 20 seconds to get stable performance.

1) We have measured the memory overhead under different contention rates. For Euno-B+Tree, we varied the skew factor of the Zipfian Distribution from 0.0 to 1.0. The results show that the average memory consumption overhead is 5.64 percent (1.79 GB versus 1.69 GB) (from 2.44 to 7.64 percent). For Euno-SkipList, the results show that the average memory consumption overhead is negligible (1.75 GB versus 1.75 GB), which is because the memories allocated to segments are just enough for storing pointers.
2) We have also measured the memory overhead with different get/put ratios: 0.2/0.8, 0.5/0.5, and 0.8/0.2. The results show that the average memory consumption overhead is 4.21 percent (1.62 GB versus 1.55 GB) (from 2.91 to 5.80 percent) for Euno-B+Tree, and 1.2 percent (1.65 GB versus 1.63 GB) (from 0.7 to 1.1 percent) for Euno-SkipList.

As such analysis results show, the additional memory consumption is small. The major reason is that reserved keys in B+Tree work as a buffer to hold the keys being sorted for split and scan operations. Such data structures are allocated and expanded dynamically and will not hold redundant space permanently. Besides, the segments structure introduces small memory overhead in skip list when segments are not full.

7 System Evaluation

To further study the benefit of Eunomia to database applications, we evaluate it in a mainstream database system. We here port Eunomia as the underlying index structure for an in-memory database called DBX [2] and evaluate the end-to-end performance under different contentions.

7.1 Workloads

The TPC-C benchmark [40] is specified by Transaction Processing Performance Council (TPC) and represents the current industry standard for evaluating OLTP system [40]. It comprises nine tables and five types of transactions, simulating a warehouse-centric order processing application. For database benchmarks like TPC-C, the number of entities to access (*scale_factor*) is by default equal to the number of threads in normal situation and shrinking the number of entities is a common way to increase the contention rate [9], [31], [41]. The decrease of entity number increases the probability that multiple threads write the same (or adjacent) keys [42].

Due to the working set limitation, it is impossible to use HTM region to protect the correctness of the entire business transaction [43]. Therefore, current HTM-based database systems adopt a two-layer design [2], [3], [4]. That is, building the database out of two independent components: a shared in-memory store layer and a transaction layer. The former exposes a key-value access; the latter builds atop the former and protects the consistency of business transaction, which contains multiple key-value accesses. Here, we also use such a strategy. We adjust the number of warehouses and working threads to control the contention rate. We test TPC-C with standard mixed transactions, and all logging functionalities are disabled.

Here we reuse the concurrency control strategy of DBX [2]. We replace the underlying HTM-B+Tree with Eunomia (namely Euno-DBX). We compare the performance with two state-of-the-art OLTP systems:

1) DBX [2], an in-memory database using HTM to protect the consistency of a single-threaded B+Tree;
2) Silo [41], which is an in-memory database adopting a Masstree-inspired [21] tree structure for underlying indexes. Since the key length is fixed to 8 bytes, the underlying Masstree can represent a highly-optimized fine-grained locked B+Tree.

7.2 End-to-End Performance

Fig. 18 shows the throughput with different numbers of warehouses and threads. From the data in Fig. 18a, we can observe that with the shrink of warehouse number, the throughput of all the systems decreases due to the increasing contention rate. Euno-B+Tree achieves 1.07X-12.14X (3.60X on average) speedup over DBX and 1.42X-2.01X (1.65X on average) speedup over Silo. From the data in Fig. 18b, we can observe that Euno-B+Tree scales well with thread number under high contention rate (warehouse number is fixed at 1), and achieves 337 K transactions per second (TPS) throughput with 20 threads. The performance of Euno-B+Tree is on average 1.60X-16.26X (9.07X on average) higher than the collapsed performance of HTM-based B+Tree in DBX, and on average 1.78X-2.35X (1.99X on average) higher than Masstree-based Silo.

To further understand the source of speedup, we collect the runtime data of the five main transaction types. Among the five types of transactions in TPC-C, NEW-ORDER (45 percent out of five transaction types) transaction is the backbone of the entire transaction system, and the contention rate of sequentially inserting new orders is the highest among the five transaction types. Therefore the throughput of original HTM-B+Tree in DBX collapses due to frequent aborts incurred by true/false HTM conflicts. Eunomia removes the contention incurred by sequential insertion, thereby unleashes the concurrency and achieves high scalability. The performance of Silo is on average 50 percent that of Eunomia, which is consistent with the key-value store performance as analyzed in Section 6.

8 Related Work

HTM is an emerging hardware feature enabling the combination of high performance and low programming complexity. HTM is gaining more and more attention in designing concurrent data structure. However, it could also exhibit pathological performance if misused. Dice et al. [44] note that memory allocators could incur certain pathological cases. Unlike them, who use HTM intuitively, we figure out more subtle design to achieve scalability under high contention. Brown et al. [45] find that multi-socket architecture could have a critical influence on the behavior of HTM, as cross-socket cache access lengthens the time to complete a transaction. In our research, we have also noticed the impact of NUMA architecture. However, NUMA architecture only magnifies the impact of transaction conflicts. Our research attempts to find a way to eliminate conflicts, thus solve the problem from the source.

An extensive body of research has discussed techniques to improve HTM efficiency by splitting monolithic transactions. Hassan et al. [46] propose optimistic transactional boosting (OTB), which divides each operation of concurrent data structures into three steps (traversal, validation, and commit). Afek et al. [47] propose consistency oblivious programming (COP), which splits concurrent code to boxes, and allows sections of code that meet certain criteria to execute without checking for consistency. Xiang et al. [48] propose software partitioning of hardware transactions (ParT). It splits HTM operations into a non-atomic *planning phase* and an HTM-protected *completion phase* with the compiler support. These methods try to reduce conflicts by shrinking transaction size, and our design differs from these works in two aspects: first, we focus on highly-contented workloads; second, we attempt to reduce both true and false conflicts.

9 Conclusion

We have presented Eunomia, a design pattern for concurrent index structures under high contention. First, Eunomia provides a new strategy of partitioning monolithic transactions to reduce abort rate. Second, Eunomia scatters the original index structure to reduce false conflicts. Third, Eunomia adopts a proactive conflict control module to eliminate true conflicts and reduce conflict rate. Fourth, Eunomia adapts away the overhead under low contention. We have shown the effectiveness of Eunomia by refactoring a concurrent B+Tree and a skip list according to Eunomia design patterns.

Acknowledgments

We are grateful to support from the National Key Research and Development Program of China (No. 2017YFB0202105), the National Natural Science Foundation of China (No. 61672160 and 61370081) and Shanghai Science and Technology Development Funds (17511102200).

References

[1] T. Harris, J. Larus, and R. Rajwar, “Transactional memory,” *Synthesis Lectures Comput. Archit.*, vol. 5, no. 1, pp. 1–263, 2010.

[2] Z. Wang, H. Qian, J. Li, and H. Chen, “Using restricted transactional memory to build a scalable in-memory database,” in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 26:1–26:15.

[3] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, “Fast in-memory transaction processing using RDMA and HTM,” in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 87–104.

[4] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, “Fast and general distributed transactions using RDMA and HTM,” in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 26:1–26:17.

[5] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[6] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, ‘Quickly generating billion-record synthetic databases,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1994, vol. 23, pp. 243–252.

[7] N. Narula, C. Cutler, E. Kohler, and R. Morris, “Phase reconciliation for contended in-memory transactions,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 511–524.

[8] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, “Extracting more concurrency from distributed transactions,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 479–494.

[9] Z. Wang, S. Mu, H. Y. Yang Cui , H. Chen, and J. Li, “Scaling multicore databases via constrained parallel execution,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1643–1658.

[10] A. Wang, et al., “Evaluation of blue Gene/Q hardware support for transactional memories,” in *Proc. 21st Int. Conf. Parallel Archit. Compilation*, 2012, pp. 127–136.

[11] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, “Robust architectural support for transactional memory in the power architecture,” in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 225–236.

[12] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Santa Clara, CA, USA, 2015.

[13] C. Blundell, E. C. Lewis, and M. M. Martin, “Subtleties of transactional memory atomicity semantics,” *IEEE Comput. Archit. Lett.*, vol. 5, no. 2, Jul.-Dec. 2006.

[14] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, “Early experience with a commercial hardware transactional memory implementation,” in *Proc. 14th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2009, pp. 157–168.

[15] J. D. Ullman, H. Garcia-Molina, and J. Widom, *Database Systems: The Complete Book*. London, U.K.: Pearson Education, 2002.

[16] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984, pp. 47–57.

[17] J.-S. Ahn, D. Kang, D. Jung, J.-S. Kim, and S. Maeng, “μ*-tree: An ordered index structure for NAND flash memory with adaptive page layout scheme,” *IEEE Trans. Comput.*, vol. 62, no. 4, pp. 784–797, Apr. 2013.

[18] Y. Sasaki, W.-C. Lee, T. Hara, and S. Nishio, “SKY R-tree: An index structure for distance-based top-k query,” in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2014, pp. 220–235.

[19] W. Pugh, “Skip lists: A probabilistic alternative to balanced trees,” *Commun. ACM*, vol. 33, pp. 668–676, 1990.

[20] M. K. Aguilera, W. Golab, and M. A. Shah, “A practical scalable distributed B-Tree,” *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[21] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage,” in *Proc. Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[22] A. Sultana, H. A. Cameron, and P. C. Graham, “Concurrent B-trees with lock-free techniques,” *Masters Abstracts Int.*, vol. 46, no. 4, p. 2171, 2008.

[23] A. Natarajan and N. Mittal, “Fast concurrent lock-free binary search trees,” in *Proc. ACM Symp. Principles Practice Parallel Program.*, 2014, pp. 317–328.

[24] A. Braginsky and E. Petrank, “A lock-free B+Tree,” in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2012, pp. 58–67.

[25] A. Ramachandran and N. Mittal, “Improving efficacy of internal binary search trees using local recovery,” in *Proc. ACM Symp. Principles Practice Parallel Program.*, 2016, pp. 42:1–42:2.

[26] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007, Art. no. 5.

[27] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Proc. ACM Symp. Principles Practice Parallel Program.*, 2012, pp. 151–160.

[28] R. Elmasri, *Fundamentals of Database Systems*. Noida, Delhi, India: Pearson Education, 2008.

[29] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proc. 23rd Annu. ACM Symp. Principles Distrib. Comput.*, 2004, pp. 50–59.

[30] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," in *Software Pioneers*. Berlin, Germany: Springer, 2002, pp. 245–262.

[31] Y. Wu, C.-Y. Chan, and K.-L. Tan, "Transaction healing: Scaling optimistic concurrency control on multicores," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1689–1704.

[32] D. M. Powers, "Applications and explanations of Zipf's law," in *Proc. Joint Conf. New Methods Language Process. Comput. Natural Language Learn.*, 1998, pp. 151–160.

[33] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[34] S. Sen and R. E. Tarjan, "Deletion without rebalancing in balanced binary trees," in *Proc. 21st Annu. ACM-SIAM Symp. Discrete Algorithms*, 2010, pp. 1490–1499.

[35] W. Pugh, "Concurrent maintenance of skip lists," Univ. Maryland, College Park, College Park, MD, USA, Tech. Rep. CS-TR-2222.1, 1990.

[36] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-579, 2004.

[37] 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.*, 2010, pp. 143–154.

[38] J. Dittrich, L. Blunschi, and M. A. V. Salles, "Dwarfs in the rearview mirror: How big are they really?" *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1586–1597, 2008.

[39] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2007, pp. 89–100.

[40] Transaction Processing Performance Council, "TPC-C benchmark, revision 5.11," http://www.tpc.org/tpc, 2010

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. ACM Symp. Operating Syst. Principles*, 2013, pp. 18–32.

[42] S. K. Cha and C. Song, "P* TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 1033–1044.

[43] Z. Wang, H. Qian, H. Chen, and J. Li, "Opportunities and pitfalls of multi-core scaling using hardware transaction memory," in *Proc. 4th Asia-Pacific Workshop Syst.*, 2013, pp. 3:1–3:7.

[44] D. Dice, T. Harris, A. Kogan, and Y. Lev, "The influence of Malloc placement on TSX hardware transactional memory," *arXiv:1504.04640*, vol. 5, pp. 1–6, 2015.

[45] T. Brown, A. Kogan, Y. Lev, and V. Luchangco, "Investigating the performance of hardware transactions on a multi-socket machine," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2016, pp. 121–132.

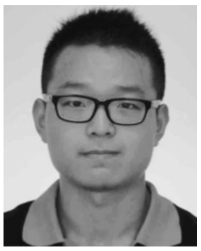
[46] A. Hassan, R. Palmieri, and B. Ravindran, "On developing optimistic transactional lazy set," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2014, pp. 437–452.

[47] Y. Afek, H. Avni, and N. Shavit, "Towards consistency oblivious programming," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2011, pp. 65–79.

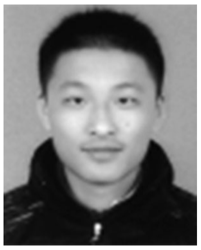
[48] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *Proc. ACM Symp. Principles Practice Parallel Program.*, 2015, pp. 76–86.

Weihua Zhang received the PhD degree in computer science from Fudan University, in 2007. He is currently an associate professor of Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, parallelization, and systems software.

Xin Wang is now working toward the graduate degree in the Software School, Fudan University and working in the Parallel Processing Institute. His work is related to computer architecture, simulation, parallel optimization and so on.



Shiyu Ji is currently working toward the undergraduate degree in the Software School, Fudan University and working in the Parallel Processing Institute. His work is related to transaction memory, parallel optimization and so on.



Ziyun Wei is working toward the undergraduate degree in the Software School, Fudan University currently and a member of Parallel Processing Institute. He is mainly focusing on GPU security, computer architecture, parallel optimization and so on.

Zhaoguo Wang received the MS and PhD degrees from Fudan University. He is a post-doc researcher in the School of Computer Science, New York University. His research area is mainly in multicore in-memory database.

Haibo Chen received the BS and PhD degrees in computer science from Fudan University, in 2004 and 2009, respectively. He is currently a professor in the School of Software, Shanghai Jiao Tong University, doing research that improves the performance and dependability of computer systems. He is a senior member of the IEEE and the IEEE Computer Society.