Eunomia: Scaling Concurrent Search Trees under Contention Using HTM

Xin Wang*†, Weihua Zhang*†, Zhaoguo Wang§, Ziyun Wei*†, Haibo Chen‡, Wenyun Zhao†¶

* Software School, Fudan University
†Shanghai Key Laboratory of Data Science, Fudan University
¶School of Computer Science, Fudan University
‡Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§Computer Science Department, New York University
{xin_wang,zhangweihua,weizy14,wyzhao}@fudan.edu.cn zhaoguo@nyu.edu haibochen@sjtu.edu.cn

Abstract

While hardware transactional memory (HTM) has recently been adopted to construct efficient concurrent search tree structures, such designs fail to deliver scalable performance under contention. In this paper, we first conduct a detailed analysis on an HTM-based concurrent B+Tree, which uncovers several reasons for excessive HTM aborts induced by both false and true conflicts under contention. Based on the analysis, we advocate Eunomia, a design pattern for search trees which contains several principles to reduce HTM aborts, 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 true conflicts, and adaptive concurrency control. To validate their effectiveness, we apply such designs to construct a scalable concurrent B+Tree using HTM. Evaluation using key-value store benchmarks on a 20-core HTM-capable multi-core machine shows that Eunomia leads to 5X-11X speedup under high contention, while incurring small overhead under low contention.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

Keywords Hardware Transactional Memory, Concurrent Search Tree, Opportunistic Consistency

1. Introduction

The emergence of hardware transactional memory (HTM) [19] provides a new opportunity to construct concurrent tree 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 locking. For this reason, there have been plenty of efforts to provide concurrent tree structures (e.g., binary search tree and B-Tree) using HTM [10, 32, 34], which was shown to achieve comparable or higher performance than lock-based schemes under low contention.

However, HTM, as an opportunistic scheme, can have a high abort rate under contention, where access conflicts among multiple cores may collapse the performance of an application. Yet, there are a number of scenarios in which applications may face high contention [21, 22, 33], due to an increased number of processor cores [37], a skewed distribution of key accesses [17], or the contention on shared entities in databases [22]. Consequently, current HTM-based concurrent search tree structures fail to deliver stable and scalable performance when workloads exhibit high contention.

This paper attempts to answer a natural question: with the assistance of HTM, can we construct a concurrent search tree structure that delivers high and scalable performance even under high contention? To answer this paper, we first present a detailed analysis of the performance of a recent concurrent HTM-based B+Tree used in several in-memory databases [10, 32, 34]. We choose the B+Tree [3] because it is a representative concurrent search tree structure widely adopted in databases and file systems. Our analysis uncovers several key issues leading to non-scalable performance under contention.

First, conventional search tree structures store keys in sorted order, which incurs severe false sharing under HTM due to a coarse-grained (i.e., cache line) conflict checking. Second, conventional search tree structures intrinsically contain pervasive shared metadata to maintain semantics, and

PPoPP '17, February 04-08, 2017, Austin, TX, USA

ACM 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018752

accesses to shared variables usually cause conflicts among transactions. Third, most HTM conflicts occur in common parts, which will abort the entire transaction and incur an expensive retry, leading to excessive re-execution overhead.

Based on the analysis, we present Eunomia, a design pattern that attempts to tackle the above issues with the following design guidelines. First, based on the observation that most HTM conflicts happen in the leaf layer, we partition a monolithic HTM region into multiple parts and protect the atomicity of different parts using HTM respectively. A version-based scheme is utilized to guarantee overall consistency at the boundary of different HTM regions. With this scheme, most conflicts only cause retry within the partitioned transaction pieces, instead of the entire monolithic transaction. Second, to eliminate false conflicts incurred by consecutive data layout and metadata accesses, Eunomia refactors the tree structure in a partitioned way, which dispatches concurrent requests to different segments. Third, to throttle the true 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 a concurrent B+Tree on Intel's Restricted Transactional Memory (RTM), called Euno-B+Tree. Experimental results using the YCSB benchmark to evaluate key-value store performance show that under high contention, Euno-B+Tree can yield 5X-11X speedup over conventional HTM-based B+Tree and 1.6X speedup over fine-grained lock-based B+Tree, with less than 6% overhead under low and modest contention. The space overhead is lower than 5%.

In summary, this work makes the following contributions.

- A comprehensive analysis of an HTM-based concurrent search tree structure (i.e., a B+Tree) under high contention.
- A design pattern with four design guidelines for scalable concurrent HTM-based tree structures.
- A scalable and concurrent B+Tree that applies the above design pattern, yielding high performance and scalability with contended workloads.

2. Background and Motivation

This section describes the necessary background regarding HTM and uses B+Tree, a widely used concurrent search tree structure, to illustrate the issues of using HTM to construct concurrent search tree structures under contention.

2.1 HTM Semantics and Concurrent B+Tree

With the commercial availability of IBM z- and p-Series [9, 30] and Intel Haswell [12] processors, HTM [19] has been widely available to the mass market. Here we use Intel's RTM[1] as an example to illustrate the semantics and quirks of HTM.

RTM provides *xbegin* and *xend* primitives to enclose a critical region which should be executed transactionally. Memory addresses read and written within an RTM region constitute the read-set and write-set accordingly. 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 [6]. 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 fallback handler when an RTM transaction retries a predefined threshold. In practice, the fallback 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 [13]. Hence, the performance of an RTM transaction will fall back to a coarse-grained lock scheme with additional cost of RTM aborts.

2.2 HTM-based B+Tree

Algorithm 1 HTM-Based Put Interface

```
1:  procedure PUT(key, newVal)
2:  XBEGIN()
3:      node = root
4:      d = depth
5:      //1. traversing internal nodes
6:      while d ≠ 0 do
7:          node = node.findChild(key)
8:          d = d - 1
9:      //2. traversing leaf nodes
10:     leaf = node
11:     record = leaf.findRecord(key)
12:     if record ≠ null then
13:         record.value = newVal;
14:     else
15:         newNode = leaf.insert(key, newVal)
16:         //3. propagating splits upwards
17:         while newNode ≠ null do
18:             parent = newNode.getParent()
19:             newNode = parent.insertNode(newNode)
20: XEND()
```

The promising features of HTM like strong atomicity and processor-assisted conflict detection have stimulated the

[1] We will use RTM and HTM interchangeably in this paper

use of HTM to construct concurrent search tree structures. Here, we use an HTM-based B+Tree from DBX [32], an in-memory database, as an example. A B+Tree is a B-Tree in which internal nodes only store keys, and only leaves are associated with values [3]. The 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 [10, 34]. Since most HTM-B+Tree operations share the major process of accessing B+Tree, here we use the *put* operation as an example to illustrate the access algorithm of HTM-B+Tree in Algorithm 1, which comprises the following steps. For brevity, we omit the structural changes and rebalance operations.

(1) Traversing the internal nodes (Lines 6-8). In this stage, the request traverses tree edges from the root to the target leaf node; (2) Traversing the leaf nodes (Lines 10-15). The request first detects if there are duplicate keys in the target leaf. If so, the put operation changes into an update; otherwise it will insert a new record; (3) Propagating splits upwards (Lines 17-19). For a put operation, if the target leaf node is already full, then insertion triggers splits, and propagates the split upwards until encountering an internal node with empty slots.

The three stages are included in a monolithic HTM region marked by *xbegin* and *xend* primitives; 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, it was shown to have much better performance compared to a state-of-the-art B+Tree (i.e., Masstree [20]) under low to modest contention [32].

2.3 Issues under High Contention

While the HTM-based concurrent tree structure has high performance under low and modest contention, its performance may collapse under high contention. To illustrate this, we evaluate the throughput of HTM-based B+Tree using the YCSB benchmark with the Zipfian input distribution [17, 25]. We adjust the skew coefficient θ in the Zipfian distribution to simulate different levels of contention. We test it on a 20-core platform with Intel's TSX [12] support. All the performance results are collected using 16 threads (a few cores are reserved for controlling threads). Threads are distributed equally on two sockets (detailed experimental setup in section 5.1).

As shown in Figure 1, with low contention rate (i.e., skew coefficient $\theta < 0.6$), the HTM-based B+Tree achieves high and stable performance. However, when the contention rate increases (e.g., $\theta > 0.6$), performance of an HTM-based B+Tree drops sharply. When $\theta = 0.9$, the performance decreases to lower than 3 million ops/s. To understand the 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

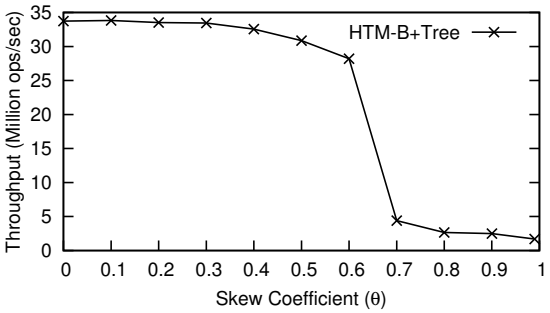


Figure 1: Performance under different contention rates.

that the performance with HTM counters deviates little from that without counters. As shown in Figure 2, the HTM abort rate increases sharply with the contention rate; the HTM abort rate for $\theta = 0.9$ is around 47X higher than that for $\theta = 0.5$. The collected CPU cycles also show that frequent HTM aborts and retries waste more than 94% of the total CPU cycles when $\theta = 0.9$.

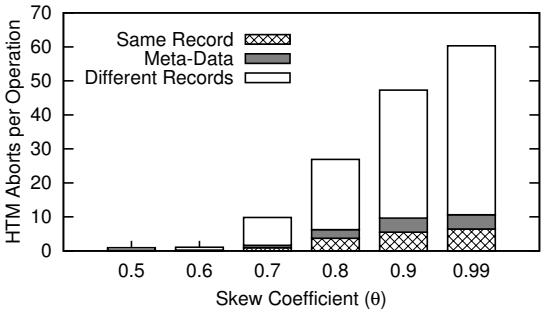


Figure 2: HTM aborts incurred by different reasons.

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

- **High retry cost due to monolithic transactions.** While using a monolithic transaction for a critical section provides consistency with trivial effort, it also causes increased abort rates. Even worse, a retry from a leaf node would waste a lot of useful work, causing high retry cost. Our analysis finds that the distribution of conflicts is non-uniform in the B+Tree: more than 90% of conflicts occur in the leaf level. In this case, a conflict in leaf nodes will abort the entire tree traversal from root to leaf, even though there is no conflict in internal nodes.
- **False conflicts.** *False Conflicts* are conflicts incurred by requests accessing different records. False conflicts stem from two major reasons. The first one is *cache line sharing of consecutive records*. B+Tree arranges keys stored in a node in a consecutive manner to provide an ordered store. However, 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 metadata in the B+Tree*. A conventional B+Tree inherently contains pervasive shared variables to maintain tree structure invariants (e.g., number of layers and version number of nodes). We categorize conflicts incurred by shared metadata as false conflicts since their target records are actually different. Since it is difficult to directly measure the exact percentage of false conflicts, 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 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 metadata, 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 Figure 2, 87%-90% of conflicts are caused by requests to different keys, which is the primary reason for the growth of abort rate. Besides, the conflicts incurred by shared metadata contribute 6%-10%, which is also a non-negligible source.

- **True conflicts.** *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 Figure 2, we can observe that 9%-12% of conflicts are incurred by requests to the same records.

3. Eunomia Design Pattern

Based on the above analysis, this section describes the Eunomia design pattern, which comprises four design guidelines to scale concurrent search tree structures under contention using HTM:

Splitting large HTM transactions with opportunistic consistency validation. Since retrying a large HTM transaction causes high retry cost, an intuitive way is to decompose a large HTM transaction into a few smaller HTM transactions. Based on the observation that the distribution of conflict aborts may be non-uniform, the decomposition can be guided by the distribution of aborts to reduce the retry code path, e.g., separating the internal nodes from the leaf nodes. However, one key remaining challenge is that there is no longer an atomicity guarantee among the decomposed transactions, which causes inconsistency for a concurrent data structure. To this end, Eunomia uses a version-based opportunistic consistency validation. The idea is based on the observation that for concurrent search trees, the interior tree structure is updated less frequently than the leaf nodes. Eunomia can use a version number to represent the tree's generation and only update the version upon a split or rebalance operation. The leaf nodes can then check the version to see if it needs to retry from the root (rare case) or just from the leaf nodes (common case). In this way, Eunomia can ensure the consistency while notably reducing the retry cost.

Partitioning data layout to reduce false conflicts. Consecutive memory layout pervasively exists in concurrent search trees like the leaf nodes in a B+Tree. However, such a layout incurs high false conflict rate when multiple threads access the data in the same cache line. To address this issue, Eunomia first partitions the continuous records in leaf nodes into multiple segments. Then requests to adjacent records will be randomly distributed to different segments located in different cache lines. Since requests to adjacent data are scattered to different segments randomly, the false conflict rate can be reduced. For operations requiring ordered data (e.g., range query in B+Tree), Eunomia uses transient buffers to store sorted keys gathered from multiple segments temporarily. Hence, the original ordering semantics can still be maintained.

Proactively detecting and avoiding true conflicts. The probability that multiple threads access the same record is high under high contention. Therefore true conflicts also impact performance significantly. To mitigate this, Eunomia proactively detects potential true conflicts and avoids them accordingly. Eunomia adopts two main techniques. First, when a node is nearly full or being split, requests to this node are likely to incur conflicts. Therefore, Eunomia employs a fine-grained advisory lock to serialize concurrent requests to a leaf node if it is near full or in the process of split. Second, detecting potential conflicts of every operation could be a time-consuming process. Eunomia leverages a Bloom filter [5] based mechanism to approximately detect potential conflicts at low cost. For example, with a Bloom filter attached to each leaf node, requests to a non-existent key in the segment will be eliminated, reducing the conflict rate.

Adopting 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 designs to handle high contention may bring overhead under low contention, Eunomia uses an adaptive contention control strategy to detect contention rates and bypass extra overhead when contention is low.

4. Concurrent B+Tree using Eunomia

This section describes how to apply the Eunomia's design to construct a contention-conscious concurrent B+Tree (namely Euno-B+Tree).

4.1 Design of Euno-B+Tree

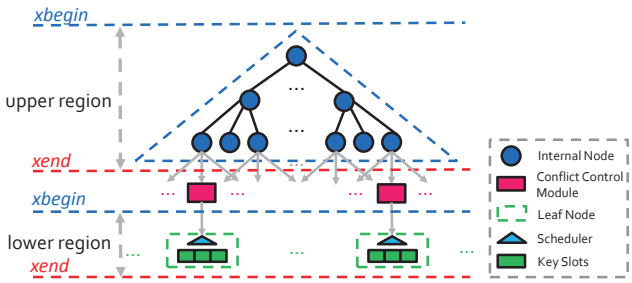


Figure 3: Overview of Euno-B+Tree structure

Splitting HTM transactions: reducing retry cost. As shown in Figure 3, Euno-B+Tree splits the original HTM region into two parts: *upper region*, protecting the atomicity of index traversing, where conflict rate is low; and *lower region*, protecting the atomicity of leaf nodes accessing, where conflict rate is high.

Trivially splitting the monolithic HTM region may introduce inconsistency issue when node splits: consider if a thread tries to insert record A, while a concurrent thread tries to read record B. 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 scans the leaf node in another HTM region. As a result, the read request is not aware of concurrent splitting.

Euno-B+Tree adopts the version-based consistency validation approach through a two-step transactional operation. The first step passes a leaf pointer to the latter one. This pointer is the "connection point" between two adjacent transactions. If the pointed leaf splits, the overall atomicity could be violated. Therefore, the overall consistency should be guaranteed by a version number tracking the split operation (Figure 4). 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, while the version number will be checked after entering the lower region (Section 4.2.1). For the above example, after the read request gets the leaf node pointer in the upper region, it also reads its version into a local variable. When the leaf node is split due to insertion, its version is updated. Therefore, 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. As a result, only when the leaf node splits will a request retry from root; otherwise a conflicting request just retries in the lower region.

Scattered leaf nodes: reducing false conflicts. As shown in Figure 4, Euno-B+Tree redesigns the leaf layer in a scattered way. Each leaf node is separated into multiple segments. Only the keys in the same segment are kept

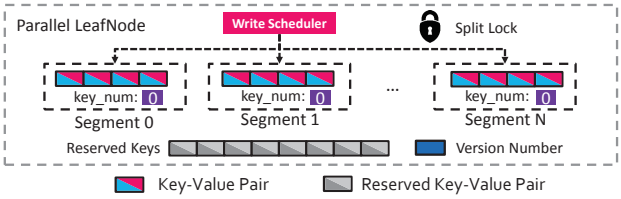


Figure 4: The data layout of leaf nodes.

sorted and stored consecutively, and the value pointers are combined with keys for the convenience of sorting and reorganization. Each segment has its own metadata to record the number of stored elements. 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 also maintains a version number to ensure the consistency of the separated HTM regions. Each leaf node maintains a lock (split lock) to serialize concurrent split operations. We directly use a per-leaf advisory lock because concurrent split operations on the same leaf node are highly likely to conflict with each other.

Besides, we set *reserved keys* in each leaf node as a transient buffer to hold the sorted keys to provide sorted results for operations requiring ordered results (e.g., range query). The *reserved keys* are only updated when it incurs split or the first scan. The *reserved keys* are dynamically allocated during a node splits or is under scan operation, and the memory space is freed after the process. Therefore, there is little extra memory consumption introduced (as analyzed in Section 5.7). Such a design sacrifices the performance of scan operations. However, since each segment is already sorted, performing a merge sort is quick for a scan operation and the sorted results can be reused for consecutive scan operations.

Example. For a *get* request, to search a specific record, Euno-B+Tree will compare the first and last elements of each segment since keys are ordered within each segment while unordered among multiple segments. For an insert operation, Euno-B+Tree uses the write scheduler to assign the operation to a random segment. With the random write scheduler, the probability that multiple requests collide within the same segment is reduced.

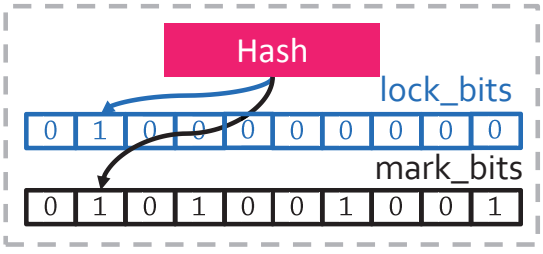


Figure 5: The hash function of conflict control module.

Conflict control with fine-grained locking: reducing true conflicts. To proactively detect and eliminate true con-

flicts, Euno-B+Tree adds a conflict control module above each leaf node. The conflict control module uses fine-grained atomic locks to detect potential conflicts and avoid two conflicting operations from accessing the same record simultaneously. Figure 5 shows the basic layout of the conflict control module for a single leaf node. It has a hash function and two bit vectors: mark bits and lock bits. The target key of a request will be hashed to a bit in the vector. The lock bits function as fine-grained atomic 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 (put vs. get, put vs. put) to enter the HTM region simultaneously. The mark bits vector is used to indicate the existence of the according key, resembling the working mechanism of Bloom filter. If a request searches for a nonexistent key, the mark bits will eliminate 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% while the false positive rate is kept under 6%.

Example. Consider two concurrent operations on the same record: read and update. In a conventional HTM-B+Tree, the read operation will abort as the concurrent put will update the value of the same record. In contrast, with the conflict control module, the read request will set the lock bit first before entering the HTM region, then the put request will be blocked until the read request finishes. As a result, true conflicts in the HTM region are avoided.

Adaptive concurrency control. To reduce the overhead for low contention rate, we use a *contention detector* in conflict control module, 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 time period, its contention rate is considered to be low. In such a condition, the new incoming request will bypass the conflict control module and leaf locks, skipping the overhead brought by these modules. Note that, the adaptive concurrency control is done at a per-leaf node basis so that each leaf node can choose its own scheme according to its contention level.

4.2 Algorithms

Based on the above design, we further describe how Euno-B+Tree handles common B+Tree operations, including get, put (it will be changed into a simple update if the target key exists; otherwise it will be changed into an insertion), and range query.

4.2.1 Get/Put Interfaces

Algorithm 2 shows the traversal procedure shared by both *get* and *put* operations. In Euno-B+Tree, the traversal procedure is divided into two phases. The atomicity of each phase

Algorithm 2 Get/Put Interface (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:     seqno = leaf.seqno
27:     ccm = leaf.CCModule //get the conflict control module
28: XEND()
29:     slot = hash(key)
30:     while !CAS(ccm[slot].lockBit, 0, 1) do
31:         spin()
32:     exist = ccm[slot].marked
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].marked, 0, 1)
39:             if insert and leaf.isNearFull() then
40:                 leaf.lock() //hold the lock for split
41: XBEGIN() //lower region
42:     if 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
```

is protected by an HTM region, one for the upper region (Lines 23-28) and one for the lower region (Lines 41-51) accordingly. The fallback strategy is similar to that used in DBX [32] and DrTM [34]. We set different thresholds for different types of aborts, where an HTM region will enter the fallback path according to the thresholds. We omit the fallback path here for brevity. The conflict control module is used to prevent conflicting requests from entering the lower region simultaneously (Lines 29-40).

To process a put or a get request:

1. Euno-B+Tree first traverses the tree index 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, it reads the current version number into a local variable (Line 26).

2. Before entering the lower region, Euno-B+Tree uses the conflict control module 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 locks to protect the atomicity of each byte in the bit vector. After a request sets the lock bit successfully, it will check 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 return a $null$ value; for a *put* request, it will set its bit in the mark bits vector and begin an insert procedure. For an insertion operation, if the leaf node needs to be split due to the capacity limitation, it will try to acquire the split lock before splitting (Line 40).
3. Scanning the leaf node is protected in the *lower region* (Lines 41-51). At the beginning, it needs to check the version number of the leaf node. If it has 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 should retry from the root and search for the latest proper leaf node. Otherwise, the leaf node is still valid; the request will search 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).

4.2.2 Insertions

When a *put* request attempts to update a non-existent record, it will include an insertion to the tree structure. The target leaf node will be split if it is full, and the split will propagate upwards. As discussed in Section 2, highly-contended workloads usually result in a large amount of highly concurrent operations, and intensive put operations incur highly concurrent insertions. Intensive insertions, combined with consecutive layout of leaf nodes, often generate false conflicts. Retaining concurrency while reducing false conflicts is the starting point of our insertion algorithm.

Based on the partitioned leaf nodes introduced in Section 4.1, algorithm 3 illustrates concurrent insertions. The important steps are shown in Figure 6.

1. The write scheduler will randomly distribute incoming requests to different segments (Lines 60-66). The random function records the index it generated last time, and guarantees that the current index is not duplicated with the last one (Line 60). If all insertions are distributed to different available segments (segments with empty slots), then multiple insertions can be processed concurrently. Each insertion affects only the local variables within the segment, so that no false conflicts will be triggered by data shifts or shared variables. If the target segment is full, the scheduler will retry the distribution attempt (Figure 6a).

Algorithm 3 Insertion and Split

```
59: procedure INSERT(leaf, key)
60:     idx = RandomScheduler()
61:     while leaf.segs[idx].isFull() and retries < threshold do
62:         idx = RandomScheduler() //retry with a new idx
63:         retries += 1
64:     if !leaf.segs[idx].isFull() then
65:         record = leaf.insertSegment(idx, key)
66:     else
67:         if leaf.hasSpace() then  //the node has sufficient space
68:             //move the keys to reserved space
69:             leaf.moveToReserved()
70:             leaf.shrinkSegs()
71:             record = leaf.insertSegment(idx, key)
72:         else if !leaf.isFull() then
73:             leaf.moveToReserved()
74:             record = leaf.insertSegment(idx, key)
75:         else //node is really full
76:             //move the keys to reserved space
77:             leaf.moveToReserved()
78:             leaf.sortKeys()
79:             newNode = leaf.split()
80:             leaf.seqno += 1
81:             //which one is to insert
82:             toInsert = compareKeys(leaf, newNode)
83:             record = toInsert.insert(key)
84:             while newNode != null  do
85:                 parent = newNode.getParent()
86:                 newNode = parent.insertNode(newNode)
87:     return record
```

2. 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* (Figure 6b), then clean the segments to accommodate new concurrent insertions. If the retries are incurred by an 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 (Figure 6c), the scheduler continues to distribute concurrent insertions to different segments (Lines 67-71).
   (b) If there is not sufficient room for new keys, the advisory lock is acquired for a further split operation (Lines 75-86).

By this means, the leaf node allocates concurrent insertions to different segments, and the shared metadata is naturally divided into several parts. False conflicts are mainly due to put operations (including updates and insertions) to consecutive records. By scattering the accesses to multiple segments, the probability of multiple requests accessing adjacent records is reduced; thus the false conflict rate can be decreased. The contiguous *reserved keys* are used to store

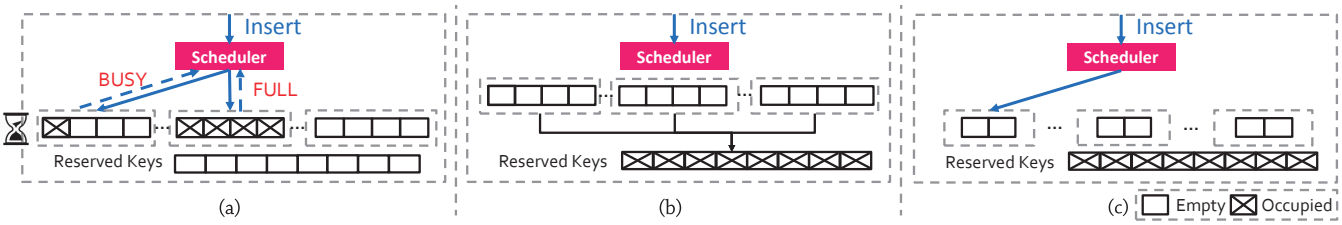


Figure 6: Concurrent insertions to a leaf node

key-value pairs for scan or split operation, which will not be updated and inserted frequently; thus *reserved keys* do not increase the false conflict rate.

4.2.3 Splits

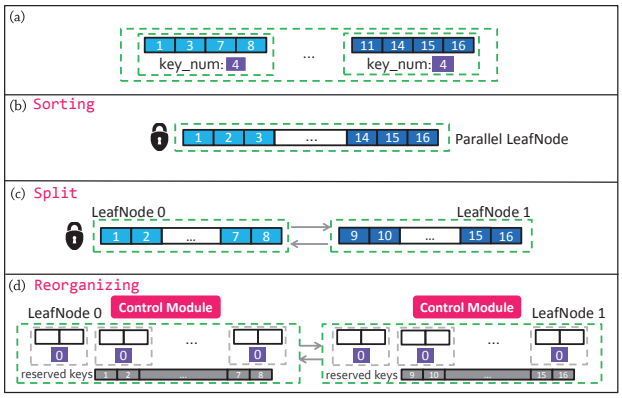


Figure 7: Splitting a node

When a node in a 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. The procedure of the node split is presented in Lines 75-86 of Algorithm 3.

1. When a node splits, Eunomia 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 (Figure 7b and Lines 76-78).
3. The original node is split according to the normal scheme of a B+Tree, while the parent node also should be adjusted accordingly. (Figure 7c and Lines 83-86).
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 store new records, eliminating the overhead of moving old records when inserting new records. Moreover, such a design can also avoid resorting them when generating the reserved keys for scan or split operation (Figure 7d).

In such a *sorting-split-reorganizing* way, we can constrain the randomness of keys within the leaf node layer. The new keys inserted to parent nodes are ordered, and thus the internal nodes are still arranged in an ordered way.

4.2.4 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 advisory 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.

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. The tree will not re-balance instantly. Euno-B+Tree reuses the deletion and garbage collection scheme in DBX [32] to clean up the unused nodes.

Re-balance. Instead of re-balancing the tree on every deletion instantly, we do the re-balance when the number of delete operations exceeds a threshold. Previous research has proved that such a re-balance scheme has theoretical and empirical advantages [27] and has been adopted by many prior systems [20, 32].

4.3 Proof Sketch

This section discusses the correctness of our algorithm with a proof sketch. The main proof obligation is to show the lower region is always consistent with upper region. More specifically, we need to prove that 1. the copy of leaf node in the upper region is mutual consistent with the copy used in the lower region. 2. the introduction of conflicting control module does not affect the consistency. 3. all the acquired locks will be eventually released.

First, as the sequence number is the key to provide the mutual consistency of the leaf node. By comparing the sequence number at the beginning of lower HTM region (line XXX), we ensure that the leaf node can not be changed since the upper HTM region commits. Second, adding the conflicting control module does not affect the overall correctness. However, to prevent the false negative

of the mark bits, the bit lock is used to ensure the mark bits (line 38) is atomically updated with the inert operation. To argue all acquired locks will be eventually released, we first need to argue there is no deadlock among concurrent operations. This is ensured by letting all operations acquire different locks in the same order. We also need to show every operation make forward progress. As we use a fallback handler for HTM to ensure it will not abort infinitely, the operation will eventually execute the codes at line 53 and line 54 which releases the bit lock and leaf lock.

5. Tree Evaluation

Our implementation of Euno-B+Tree consists of approximately 600 lines of code in C++. By evaluating the prototype, we try to answer the following questions:

- Does Eunomia solve all the issues discussed in section 2.3?
- Can Eunomia deliver better performance than a fine-grained locking B+Tree even under high contention?
- Can Eunomia achieve performance scalability under different contention levels?
- How does the workload (get/put ratios or input distributions) affect Eunomia's performance?
- How does each design choice affect the performance?

5.1 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, 32KB 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 Restricted Transactional Memory to implement atomic regions.

We compare Euno-B+Tree with three different concurrent B+Tree implementations: (1) An HTM-based B+Tree adopted by many database systems [10, 32, 34], which uses HTM to protect the atomicity of the entire operation. We reuse the fallback strategy and retry policy in DBX [32]. (2) A highly optimized concurrent B+Tree implementation derived from Masstree [20]. It uses fine-grained locks to achieve good scalability. However, we still use the term "Masstree" for simplicity. (3) An HTM version of Masstree, denoted as HTM-Masstree. It uses HTM region to protect the entire Masstree operation (as in (1)), subsuming multiple elided locks.

We here adopt the Yahoo! Cloud Serving Benchmark (YCSB) [11], 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%. The key range is set to 100 million. The average tree depth is 6 and run duration is set longer than 20 seconds to get stable performance. Unless otherwise specified, we use Zipfian as the default input distribution, private to each thread (intra-thread locality). The Zipfian distribution has a skew coefficient θ, and the probability of accessing a key k is given by

$$P(k) \propto (\frac{1}{k})^{\theta} \quad (1)$$

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

5.2 Throughput

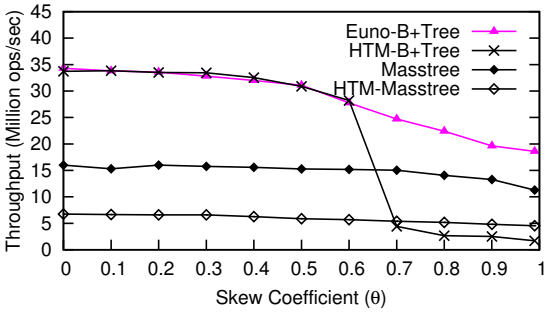


Figure 8: Throughput under different contention rates (Thread number is 16).

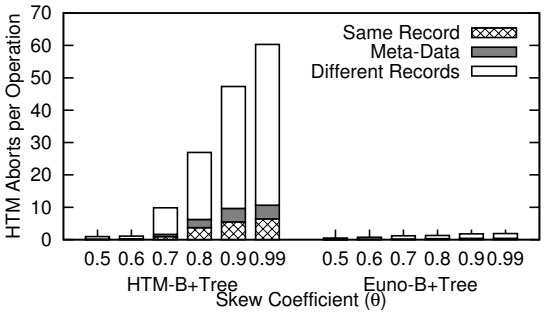
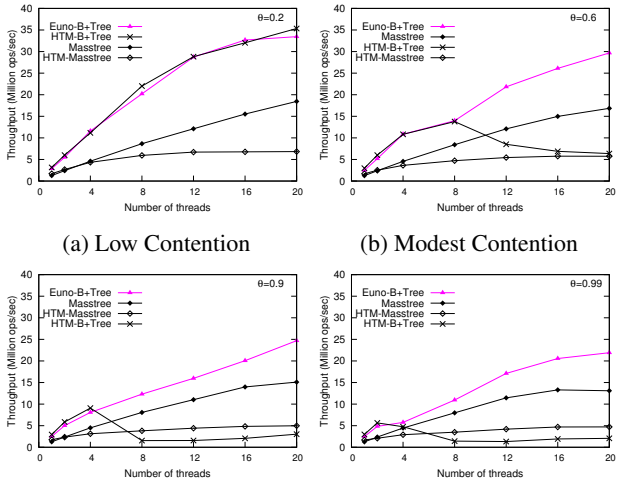


Figure 9: Comparison of HTM aborts incurred by different reasons (Thread number is 16).

To see whether Eunomia solves the issues discussed in Section 2.3, we repeat the experiment in Figure 1; the data are shown in Figure 8. When the contention is low or even modest ($\theta < 0.6$), Euno-B+Tree can obtain similar performance as HTM-B+Tree about 36.85% higher than Masstree. This is because Euno-B+Tree uses adaptive concurrency control to reduce most overhead under low contention, while Masstree's fine-grained synchronization needs to execute additional instructions. According to our analysis, when θ = 0.5, the amount 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 (§4.6 of [20]). For example, when θ = 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 vs. 1.7 M Ops/s with θ = 0.99). This is because Eunomia eliminates most aborts compared with HTM-B+Tree (Figure 9): 60.3 vs. 1.9 aborts per Op under extreme high contention. Compared with Masstree, it has 65% better performance even under high contention. This is because Euno-B+Tree executes around 40% fewer instructions under high contention. Performance of the HTM-based Masstree is worse than Masstree under both low and high contention. This is because HTM-based Masstree has shared variable accesses which incurs frequent HTM aborts. This shows that, even for a highly optimized concurrent B+Tree, it is still hard to directly take advantage of HTM.

5.3 Scalability



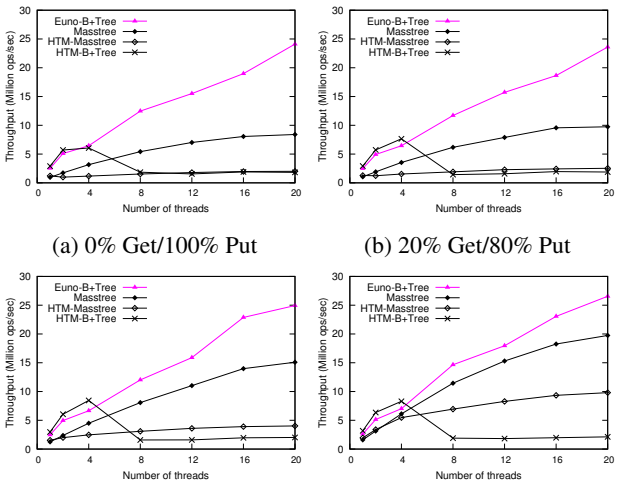
(a) Low Contention (b) Modest Contention

(c) High Contention (d) Extremely High Contention

Figure 10: Performance scalability under different contention levels.

Besides the skew coefficient, an increase in the number of threads also intensifies the contention rate. Here we evaluate scalability with increasing threads under different levels of contention. We set θ by referencing previous research [15, 37]: 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. Figure 10 shows the results. Thanks to the adaptive control, under low contention (θ is 0.2), 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% more instructions per thread), this overhead is amplified by adding more threads. As a result, Eunomia is 52%-63% better than Masstree under high contention. HTM-Masstree fails to scale after 8 cores. Under modest contention (Figure 10b), the performance of HTM-B+Tree begins to collapse after 4 threads due to the increase in abort rate. Masstree still has stable performance as false conflicts do not waste CPU cycles. Under high or even extreme contention, Euno-B+Tree still has reasonable scalability and performs better than Masstree (21.9 M vs. 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.

5.4 Get/Put Ratio



(a) 0% Get/100% Put (b) 20% Get/80% Put

(c) 50% Get/50% Put (d) 70% Get/30% Put

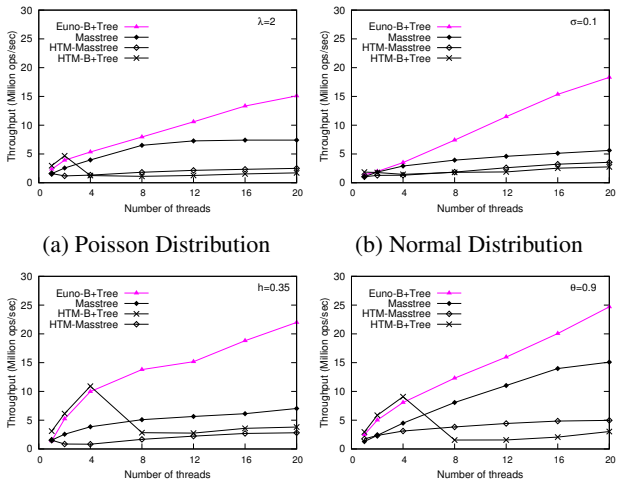
Figure 11: Performance under different get/put ratios in Zipfian Distribution. Euno-B+Tree achieves scalable performance with various get/put ratios under high contention (θ = 0.9).

Get and *put* are the fundamental operations of a key-value store. Interleaved reads and writes often exacerbate contention. Here we measure the impact of the get/put ratio on overall performance. We consider ratios of (1) 0% get/100% put, (2) 20% get/80% put, (3) 50% get/50% put, and (4) 70% get/30% put.

The default ratio of YCSB is 50% get/50% put [11], and we adjust the percentage of get and put to simulate read-heavy and write-heavy situations. Since Eunomia is designed for concurrent tree modifications, we do not consider the all-get case here.

We measure the Zipfian distribution with high contention (θ = 0.9) under these get/put ratios. The performance is shown in Figure 11. From the results, we can observe that Euno-B+Tree can achieve near-linear scalability under various get/put ratios. The speedup increases with put ratio, and the advantage of Eunomia is most obvious with 100% puts (Figure 11a). This is because the put operations introduce more conflicts, in which case Euno-B+Tree is much better than other alternatives. The performance of Masstree also scales smoothly with threads, but is still 25% lower than Euno-B+Tree on average.

5.5 Different Input Distributions



(a) Poisson Distribution (b) Normal Distribution

(c) Self-Similar Distribution (d) Zipfian Distribution

Figure 12: Performance with different input distributions under high contention.

Since the input keys may follow various kinds of distributions in real use cases, we measure performance with the following mainstream input distributions. Requests with different types of contended keys constitute different input distributions and thus impact the performance of B+Tree systems in different ways.

- **Poisson**: A common distribution in probability and statistics, Poisson expresses the probability of a given number of events occurring in a fixed interval of time or space if such events happen with a known average rate and independently of the time since the last event. The skew coefficient is determined by the expected value λ. Here we set contention such that the 10% hottest records are accessed by 70% of the requests.
- **Normal**: Here the requests are chosen to follow a Normal distribution with mean of N/2, where N is the set to cover a moving range of leaf nodes. The standard deviation is set to 1% of the mean to simulate a skewed situation. Here we set contention such that the 10% hottest records are accessed by 67% of the requests.
- **Self-Similar**: Common in network traffic, the requests in this distribution follow an 80-20 rule [17]. Within any range of the distribution, the skew coefficient is the same as in any other region. Here we set contention such that the 10% hottest records are accessed by 66% of the requests.

The get/put ratio is set to the default 50%/50% in each distribution. The results are shown in Figure 12 (the Zipfian distribution is tested with θ = 0.9). From the figure, we can observe that Euno-B+Tree can achieve scalable performance under various input distributions, higher than HTM-B+Tree and lock-based Masstree. In the Poisson distribution, Self-Similar distribution, and Zipfian distribution, the performance of HTM-B+Tree collapses when threads exceed 2 or 4, due to the CPU time wasted by HTM aborts and retries. Under the normal distribution, the input distributes densely around the expected value; thus, the performance of HTM-B+Tree keeps at a low level, without any obvious trend increasing. The performance of Masstree can scale stably under Zipfian distributions since its contention is comparatively modest among the four input distributions, yet is 38%-51% (40% on average) lower than Euno-B+Tree due to the number of executed instructions.

5.6 Impact of Different Design Choices

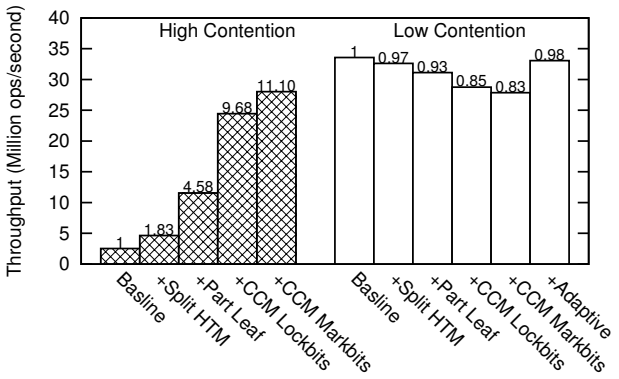


Figure 13: Impact of Different Design Choices. The relative performance is labeled on the top of each column.

To understand the performance improvement and cost from various design aspects, we here present an analysis of multiple factors in Figure 13. The benchmark is the YCSB with Zipfian input distribution with 20 threads under both high contention (θ is 0.9) and low contention (θ is 0.2).

Baseline refers to an HTM-B+Tree using a single RTM region to protect the entire operation. *+Split HTM* means splitting the monolithic HTM region into two parts, and using version numbers to protect consistency. *+Part Leaf* refers to partitioning the leaf nodes to avoid false conflicts. *+CCM lockbits*/*+CCM markbits* refer to adopting the lock_bits/mark_bits in the conflict control module. *+Adaptive* means adopting adaptive concurrency control under low contention.

First, splitting the monolithic HTM region gets 1.83X speedup under high contention, as the overhead of rollback is reduced. Under low contention, it introduces 3% overhead, as it doubles the number of RTM instructions for each operation. Second, partitioned leaf nodes (*+Part Leaf*) generates 4.58X speedup under high contention, as it decreases the false conflict rate caused by accessing the same leaf node. But it incurs 4% performance slow down under low contention by introducing extra overhead on searching. By further reducing false conflicts, *+CCM lockbits* and *+CCM markbits* get 9.68X and 11.10X speedup under high contention. They also introduce overhead (8% and 2%) under low contention due to extra computation.

With *+Adaptive*, the overhead introduced by the system is successfully bypassed. As a result, we only have 2% performance overhead under low contention, which is brought by extra HTM instructions and version checking.

5.7 Memory Consumption Analysis

In the design of Euno-B+Tree, two structures 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. Here, we evaluate the memory consumption overhead of Euno-B+Tree using Valgrind toolset [24]. The workload includes getting and putting 10 million keys in a Zipfian distribution with 16 threads. The node fanout is set to 16, 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. We varied the skew coefficient of the Zipfian Distribution from 0.0 to 1.0. The results show that the average memory consumption overhead is 5.64% (1.79GB v.s. 1.69GB) (from 2.44% to 7.64%).
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% (1.62GB v.s. 1.55GB) (from 2.91% to 5.80%).
3. We have also measured the memory overhead with different input distributions. The input distributions include Self-Similar Distribution, Poisson Distribution, and Uniform Distribution. The results show that the average memory overhead is 2.20% (1.81GB v.s. 1.77GB), 6.91% (1.81GB v.s. 1.70GB), and 2.25% (1.77GB v.s. 1.74GB) respectively.

As such analysis results show, the additional memory consumption is small. The major reason behind this is that reserved keys work as a transient buffer to hold the keys being sorted for split and scan operations. Such data structures are allocated and freed dynamically and they are allocated only before a split or a scan operation.

6. Related Work

Conventional concurrent search tree structures [16] usually adopt fine-grained locks [2, 4, 20, 29] or lock free methods [7, 23, 26, 28] to unleash concurrency and provide consistency. These methods can provide scalability, but incur high performance cost due to thread synchronization. At the same time, lock-free schemes primarily use single-word atomic instructions to coordinate multiple threads, making it difficult to argue correctness. Compared with them, we leverage HTM to simplify the implementation and provide good scalability.

Recent works [31, 32] also try to use HTM to simply the implementation of concurrent data structures. However, it could also exhibit pathological performance if misused. Dice et al. [14] 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. [8] also 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. [18] propose optimistic transactional boosting (OTB), which divides each operation of concurrent data structures into three steps (traversal, validation, and commit). Afek et al. [1] 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. [35, 36] propose software partitioning of hardware transactions (ParT). It splits common operations into read-mostly *planning phase* and write-mostly *completion phase*. 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.

7. Conclusions

We have presented Eunomia, a design pattern for concurrent search tree structures under high contention. First, Eunomia provides a new strategy of partitioning monolithic transactions to reduce abort rate. Second, Eunomia scatters the original tree structure to reduce false conflicts. Third, Eunomia adopts fine-grained advisory locks to eliminate true conflicts. Fourth, Eunomia adapts away the overhead under low contention. We have shown the effectiveness of Eunomia by refactoring a concurrent B+Tree according to Eunomia design patterns.

Acknowledgments

We thank our shepherd, Michael Scott, and the anonymous reviewers for their constructive comments. We are grateful to support from the National Key Research and Development Program of China (No. 2016YFB0800104), the National Natural Science Foundation of China (No. 61672160), Shanghai Science and Technology Development Funds (16JC1400801) and NFS award CNS-1218117.

References

[1] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.

[2] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-Tree. *VLDB*, 1(1):598–609, 2008.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262, 2002.

[4] J. Besa and Y. Eterovic. A concurrent red–black tree. *JPDC*, pages 434–449, 2013.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[7] A. Braginsky and E. Petrank. A lock-free B+Tree. In *SPAA*, pages 58–67, 2012.

[8] T. Brown, A. Kogan, Y. Lev, and V. Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *SPAA*, pages 121–132, 2016.

[9] 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 *ISCA*, pages 225–236, 2013.

[10] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *EuroSys*, pages 26:1–26:17, 2016.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.

[12] I. Corporation. Intel® 64 and ia-32 architectures software developers manual, 2015.

[13] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, pages 157–168, 2009.

[14] D. Dice, T. Harris, A. Kogan, and Y. Lev. The influence of malloc placement on tsx hardware transactional memory. *arXiv preprint arXiv:1504.04640*, 2015.

[15] J. Dittrich, L. Blunschi, and M. A. V. Salles. Dwarfs in the rearview mirror: how big are they really? *VLDB*, 1(2):1586–1597, 2008.

[16] G. Graefe. Modern B-Tree techniques. *Found. Trends databases*, pages 203–402, 2011.

[17] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickily generating billion-record synthetic databases. In *SIGMOD*, volume 23, pages 243–252, 1994.

[18] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *OPODIS*, pages 437–452, 2014.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*. ACM, 1993.

[20] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, pages 183–196, 2012.

[21] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[22] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[23] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.

[24] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[25] D. M. Powers. Applications and explanations of zipf's law. *Joint conferences on new methods in language processing and computational natural language learning*, pages 151–160, 1998.

[26] A. Ramachandran and N. Mittal. Improving efficacy of internal binary search trees using local recovery. In *PPoPP*, pages 42:1–42:2, 2016.

[27] S. Sen and R. E. Tarjan. Deletion without rebalancing in balanced binary trees. In *SODA*, pages 1490–1499, 2010.

[28] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+Trees on many-core processors. *VLDB*, 4(11):795–806, 2011.

[29] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[30] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT*, pages 127–136, 2012.

[31] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *APSys*, pages 3:1–3:7, 2013.

[32] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys*, pages 26:1–26:15, 2014.

[33] Z. Wang, S. Mu, H. Y. Yang Cui, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, pages 1643–1658, 2016.

[34] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *SOSP*, pages 87–104, 2015.

[35] L. Xiang and M. L. Scott. Composable partitioned transactions. In *Wkshp. on the Theory of Transactional Memory (WTTM)*, 2013.

[36] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *PPoPP*, pages 76–86, 2015.

[37] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *VLDB*, 8(3):209–220, 2014.