Concurrent MLP Index

Efraim Adadi
efraimadadi@mail.tau.ac.il

Hod Badichi
hodbadihi@mail.tau.ac.il

ABSTRACT
In this paper, we introduce an optimistic concurrent and memory-efficient hash trie structure designed to support the storage of ranges, leveraging the principles of memory level parallelism (MLP) and derived from the MLP Index [6]. To evaluate its performance, we conduct a comparative benchmark against Linux's Maple tree [4], a widely-used data structure. Our findings indicate that our trie implementation performs on par with the Maple tree on lookup operations, but lacks on insertions. However we assume that this discrepancy in insertions is primarily due to the fact that our structure is not yet fully optimized.

Author Keywords
Memory Level Parallelism; Range trees; Maple tree; Trie

INTRODUCTION

Virtual Memory Mapping

Storing non-overlapping ranges in an efficient manner is useful in modern systems. For example, address spaces in a modern operating systems might be using 1000 distinct memory regions in each process [1]. On Linux, the fundamental concept is Virtual Memory Address (VMA) that represents a contiguous range of virtual memory addresses in a process's address space. The VMAs are used to map parts of the process's virtual address space to physical memory or to other resources such as files on disk.

There are two main operations that are performed on VMAs: lookups and modifications [3]. Lookups are used to find actual memory and modifications are used when new memory is needed or old memory is no longer needed.

Linux original solution for managing the VMAs was using `rbtrees` combined with two optimizations. The first, is having the leaves being linked, and the second is tracking the gaps between the ranges.

The Maple Tree

Modern computer systems are distinguished by two key characteristics: They involve a significant trade-off between accessing the small, close, and fast cache memory and the slower, more distant, and larger main memory. They are also characterized by having multiple cores.

In these settings, the original solution in the Linux kernel for managing VMAs (using `rbtrees` as mentioned earlier) has limitations. The requirement to maintain tree balance complicates multi-threaded access to the data structure. Additionally, since its nodes were embedded within the VMA structs (`vm_area_struct`), it was not an efficient solution in terms of cache utilization [3].

In order for having a data structure that allows tracking gaps, storing ranges and avoids using locks, Linux is transitioning into using Maple Trees- a range based B trees designed to be Read-Copy-Update safe. I.e. leaves in the tree are being copied and updated without readers being affected by writing operations. As a Maple Tree node consists of 2 cache lines, it is also more cache-efficient than the original `rbtrees`, whose nodes consist of 3 cache lines.

Memory Level Parallelism

Historically, performance of a computer was measured on ILP - instructions level parallelism - units. Since there is a dozens times multiplier (around 70 [6]) gap between executing an instruction and reading a value from main memory, Andrew Glott proposed the following thought experiment to justify the concept of MLP [2]:

> Imagine that a cache miss takes 10000 cycles to execute. For such a processor, instruction level parallelism is useless, because most of the time is spent waiting for memory. Branch prediction is also less effective, since most branches can be determined with data already in registers or in the cache; branch prediction only helps for branches which depend on outstanding cache misses

Therefore, MLP is the ability of a computer system to perform multiple memory-related operations simultaneously or in parallel, and improving it is done in order to utilize any inherent parallelism in memory access patterns in order to improve performance. Among of the features related to MLP we can denote:

- **Cache Hierarchy**. The cache goal is to to store "highly likely to be accessed" data (whether in the perspective of temporal locality or spatial locality) and therefore it allows the processor perform better if it memory access patterns are being in favour by cache
- **Prefetcher**. Predicting future memory accesses and fetching them into the cache is a feature that hides memory access latency. In addition, programmers may use a "prefetch" instruction that advise the processor to fetch an address to cache
- **Single Instruction Multiple Data**. SIMD is a type of instructions that are used to perform single instruction on multiple data elements simultaneously, including memory access operations

The MLP Index algorithm stands out from the main approach towards leveraging memory level parallelism since it is designed from bottom up to support and leverage MLP on any query [6]. On the following sections we will further discuss the baseline implementation in the MLP Index section, present

the concurrent optimistic implementation in the Concurrent MLP Index section. We will later present how we were able to support efficient range storing in the Support ranges insertion section. Finally, in the Evaluation section we benchmark the concurrent MLP Index against the baseline implementation and against the Maple tree.

Our Contributions

1. **Concurrent MLP Index**. Optimistic lock free implementation
2. **Range support**. Efficient range storing in the MLP index
3. **Userspace Maple Tree**. We modified Linux kernel Maple tree to work in user space
4. **Evaluation**. We evaluated the concurrent MLP Index w.r.t Maple tree.

MLP INDEX

Overview

The MLP Index is an ordered index implementation for a key universe of size M. It consists of a path-compressed trie with a depth of D (D is chosen to maximize hardware capabilities for tracking parallel memory accesses). In this trie structure, the nodes are managed within a hash table to avoid pointer-chasing during trie traversal. Each node has a fan-out of $M^{1/D}$ children, which are managed recursively in a similar data structure held by the node.

In practice, we are primarily interested in the final, optimized implementation for a universe size of $M = 2^{64}$. In this optimized version, choosing $D = 8$ results in a fan-out of $M^{1/D} = 256$ nodes for each parent node. Consequently, the children of each node can be efficiently managed using a bit map, rather than a heavier data structure found in more naive implementations.

This optimization not only conserves space but also reduces memory access times. This is illustrated in Figure 1. A similar optimization, driven by the same rationale, involves holding the first three levels of the trie in memory rather than within individual nodes.

The tree provides support for three main operations: `insert`, `lookup`, and `lower-bound`. Each of these operations relies on the `queryLCP` function, which, when given a key, returns the longest common prefix of the key that is stored within the data structure

MLP Index nodes

The MLP Index uses Cuckoo Hashing with `24byte` slot entries defined as shown in Listing 1.

The index of a node in the hash table is its `prefix`, i.e. the string represented by the node's parent concatenated with the string that represents the edge leading from the parent to that node. For example, in Figure 1 the leaf node index in the hash table does not include the compressed `00...00` part.

Since top levels of the data structure are not part of the hash table, the hash table contains nodes whose prefixes are of length of at least `24bit`. Therefore, a further optimization is

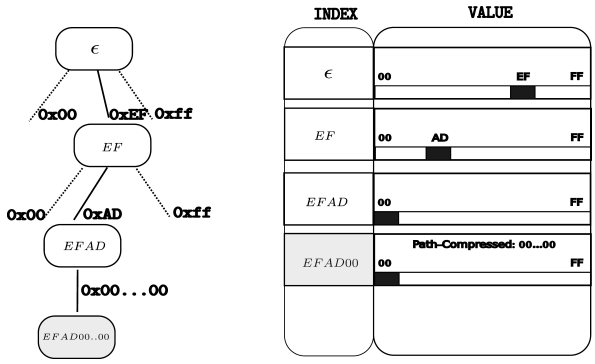

Figure 1. On the left side, you can see the abstract structure of a path-compressed trie with a fan-out of 256. On the right side, there's the same structure, but it is managed using a hash table. With a hash table, each node only contains information about whether a child of it exists or not, eliminating the need to hold a direct pointer to the child. To locate a child, you can search the hash table by concatenating the parent's index with the edge that leads to the child. An additional optimization, not shown in the figure, is that the first three levels are kept in memory due to their relatively small size in terms of memory usage.

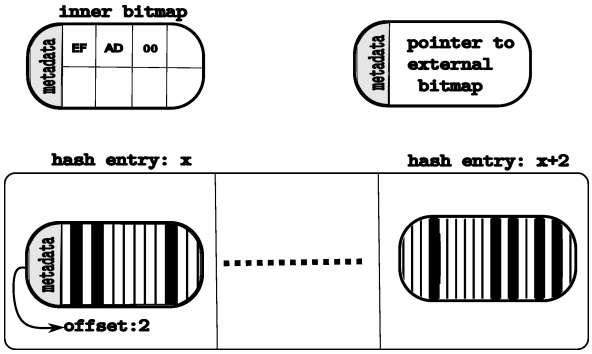

Figure 2. There are different types of nodes in this structure. In the upper left corner, you'll find a small node capable of holding a maximum of 8 children under the `childMap` field. In the bottom example, a neighbor slot is 'borrowed' to complete the node's bit map. In cases where there are no available adjacent empty nodes to borrow from, we resort to allocating external memory to store the bitmap, as illustrated in the top right example.

to store a short hash (`18bit`) of the key and optimistically use it during the queries instead of in the longer full prefix (the `header.meta_data.tag` field inside the node in Listing 1).

As discussed above, node manages the existence of any of its child in a bit map. In our settings, the universe of children that each node needs to manage is small enough ($M^{1/D} = 256b = 32B$) to justify storing it directly in nodes instead of in a recursive data structure. To support that we define two types of nodes (see Figure 2):

1. For nodes with at most 8 children, explicitly indicate those children at the `childMap` field. Those are called *small nodes*

2. For nodes with more than 8 children use `childMap` as the first `8B` of a `32B` bitmap, and use "other memory" for the rest of `24B` of the bitmap. Such nodes are called *large nodes*

There are two options regarding of what exactly is the "other memory" that can be used to store the rest of the bitmap. Optimistically, since the load factor of the used Cuckoo Hash is 50%, and since the distribution of occupied entries can be considered as uniform, there is high probability[1] to have an empty slot in a prefetched `128B` chunk of entries. Putting this in another words, assuming one prefetched `128B` data of entries (which is essentially a bit more than 5 hash slots), with one of these slots is a small node that should to be changed to a large node, then there is a high probability that one of the other 4 slots is empty. In that case, we would use that empty slot (since it leverages MLP). We call sch nodes *stolen nodes*, and we mark the offset from node to it's stolen complement in the node's metadata.

The other option for having more memory to store the rest of the children bitmap of large node is allocating external memory. In that case, the field `childMap` is used as a pointer to external allocated `32B` of memory.

The final field in a node is `minKey`, the minimal key in the subtree of the node. Since the metadata holds the `indexLen` of a node (corresponding to the index of the prefix that defines this node in the trie) and holds the `pcLen` (corresponding to the length of the string this node represents in the trie, including the last path compressed string that is concatenated to its index) we have all needed information to identify a node[2].

Listing 1. Hash table node

```
typedef union NodeMetaData {
    uint32_t value;
    struct MetaData {
        /* indicates that slot is: available,
            normal or bitmap(stolen) */
        uint32_t occupyFlag : 2;
        /* length of the indexing part of the
            key */
        uint32_t indexLen   : 3;
        /* full length of key (inc. it's
            compressed part) */
        uint32_t pcLen      : 3;
        /* indicate that the node is small,
            uses external mem, or uses stolen
            slot */
        uint32_t nodeType   : 3;
        /* if node is small stores num of
            child, if uses stolen slot keeps 2
             bits of bitmap */
        uint32_t numChild   : 3;
        uint32_t tag        : 18;
    } meta_data;
} NodeMetaData_t;

struct CuckooHashTableNode {
    NodeMetaData_t header;
    uint32_t minvOffset;
    uint64_t minKey;
    uint64_t childMap;
}
```

`queryLCP`

For a given `key`, `queryLCP` starts by querying the hash table for any of the 8 prefixes that corresponds to `key`. I.e. it query the hash table for `key[:24]` , `key[:32]`, ... ,`key[:56]`. These queries are parallel with respect to memory as there is no pointer chasing in this algorithm, and that is a key factor for leveraging MLP.

After the queries resolved, the hash entries resides in cache, `queryLCP` looks for the longest prefix that matches the key. It first does so by checking the `tag` as we noted before, but in case that there is hash collision it takes the slow path and compares any of the prefixes to the key.

After finding the longest common prefix node, `queryLCP` use the `minKey` of the LCP and compares it to `key` in order to find the true LCP length. For example, in Figure 1, the LCP length for `EFAD000011223344` is 4 even though the length of the prefix node is 2 due to the path compression optimization.

Given `queryLCP` logic, implementing the `lookup` operation is straight forward and accomplished by calculating the length of the LCP. The queried key is in the tree if and only if the length of it's LCP is 8.

`Insert`

Recall that the top levels of the trie are bit map arrays. Thus, to insert a new key to the trie, the specific bits in these bitmaps that match the top bytes in the key are set to 1.

Then, to insert an actual node to the hash table, we start with finding the LCP node, which is queried from the hash table. There are two possible cases:

1. If the new key completely matches the path compressed part of the LCP, then we just need to add the new key to the hash table as it is not different than any of the other nodes in the subtree of the LCP node.
2. But, if it shares only part (or none) of the path compressed part of the LCP node we need to split the path compressed part with another node that marks the splitting point in the path compressed string.

Note that since nodes are stored in a Cuckoo Hash Table, `Insert` should make sure that there is an available slot for the new key. Specifically, if both of possible slots are occupied, there is a need to invoke a chain of displacements of hash slots.

Moreover, since some slots in the hash table, contain bitmaps (and not normal nodes, see Figure 2), bitmap nodes (a "stolen" slot) might be moved to any other close slot or to an external memory bitmap.

Finally, there is the case where a new key is being added to a small node that already contains 8 children. In that case, the insertion must also allocate a slot to hold its bitmap. As explained before, if there's an empty neighbor slot it will be stolen, but o.w. an external memory is allocated.

`lower_bound`

The `lower_bound` method returns the minimum key in the data structure that is no smaller than the input key. The first step is to find `lower_bound` for the LCP.

If the `lower_bound` does not exist in the LCP subtree, we need to take one step upward in the trie hierarchy to be exposed

[1]The probability for all other 4 slots to be occupied is 2^{-4}

[2]While also saving the memory required for the key of the node itself

to larger domain of keys that are currently in the trie. We look on the children of the LCP parent, and search for a child that is larger than the LCP. If there is no such child we take another step upwards (shorter prefix) in the trie hierarchy and repeat.

If the input key does not match the compressed part of the LCP, then there are two possibilities. It might be larger than the whole LCP subtree or smaller than the whole LCP subtree. If it is larger we have to find the lower bound in the parents of the LCP as explained before, and otherwise the minimal key in the LCP subtree is the lower bound.

CONCURRENT MLP INDEX

The concurrent MLP Index employs optimistic lock-based concurrency control within the memory model of single writer and multiple readers. It assigns version numbers to the hash slots that write operations update during their execution. Readers rely on these version numbers for two purposes: first, to detect an inconsistent state, which is indicated by an odd version number, and second, to identify situations where a modification was successfully completed while the reader was actively processing its operation. This is signaled by an even version number that differs from the value the reader initially collected at the start of its operation.

We have made the observation that a version only matters for slots that store `small nodes` or slots that store the main part of `large node`. Since any reader that accesses the second part of any `large node` is also accessing the main part of it (which we call `owner`).

In addition to the versions that are assigned to slots, there are 3 additional versions that are assigned to the top levels of the trie. Assigning versions to the top levels of the trie is not contradicting the goal of having fine granularity level of concurrency, since `lookup` is ignoring the top levels, and it is only the `lower bound` operation that might use these bit maps if it need to find the lower bound in the parents of the LCP node.

In practice, we made a one-byte expansion to the hash table node structure to accommodate the version number. This modification does not adversely affect the Memory-Level Parallelism (MLP) optimizations that allow the prefetcher to fetch 5 nodes. The slot size has shifted from `24B` to `25B`, resulting in a change in the 5-slot chunk size from `120B` to `125B`. This size remains smaller than the memory unit size (`128B`) that the prefetcher accesses during its invocation.

SUPPORT RANGES INSERTION

The motivation behind the effort to efficiently store ranges was to enhance the competitiveness of the MLP Index compared to the Maple Tree. Consequently, we introduced support for range storage within the storeRange API. Our primary objective in implementing this feature was to minimize the necessity for frequent insert operations.

We begin with the observation that for a given range of keys `(low, high)` there is a shared prefix of length at least zero bytes. This prefix corresponds to a node that is the `range root`. We call the index of the last byte that `low` shares with `high` a `splitting byte (sb)` so that the splitting byte is zero if low differs from high on the first byte, and the splitting byte is eight if keys are equal.

Perfect range

We also present the notion of `perfect range`. A perfect range satisfies that after the splitting byte, the range low bound is padded with $0x00$ and the high bound is padded with $0xff$. The perfect is in sense that the range root subtree is complete.

For example, consider the range

$$0x11|22|33|44|55|00|00|00,\ \ 0x11|22|33|44|55|ff|ff|ff$$

The 6th byte is the first different byte and the range is perfect. This means that the subtree that is represented by the prefix $0x11|22|33|44|55$ is a complete subtree.

Arbitrary range

Our idea was that given a perfect range we don't need to store any of its children but only the range root and mark all its child map bits with 1. Then, for an arbitrary range it is clear what to do: First convert it to the best perfect range possible, and then recursively handle the possible leftovers.

For example, consider the range

$$(0x11|22|33|44|54|ab|00|00,\ \ 0x11|22|33|44|55|ff|ff|ff)$$

Note that the best possible way to handle this range is to handle the perfect range

$$(0x11|22|33|44|55|00|00|00,\ \ 0x11|22|33|44|55|ff|ff|ff)$$

and then recursively handle

$$(0x11|22|33|44|54|ab|00|00,\ \ 0x11|22|33|44|54|ff|ff|ff)$$

which leads to 2 node insertions in total.

A remark: If the splitting byte is 7, then low differs from high on the 8th byte. On this case, the best possible solution is to just insert the leaves as singletons, but instead we have chosen to assume that low and (high+1) are divisible by 256. Thus, low ends with $0x00$ and high with $0xff$. This assumption is reasonable, but as explained before it can be removed by simply inserting the remaining leaves. In case that one chooses to support every range, the highest number of separating leaves that one could insert is 510.

Range Node

Recall that each bit in a node's child map corresponds to a key prefix derived from concatenating the node's string with the bit's value. In the baseline implementation, if a bit was set in a node's child map, it implied the existence of a node in the hash table that represented the corresponding prefix within the trie.

In our implementation, we do not store any inner nodes in the hash table, except for the `range root`. As a result, there may be nodes with set bits, but no corresponding nodes in the hash table that reflect these prefix bits. We refer to such nodes as 'range nodes' because they indicate the presence of a complete subtree for any of the set bits that are not represented in the hash table.

Implementation

We modified `queryLCP` to identify a range node. It is done by checking the bit in the child map that corresponds to the next byte of the key. If the bit is set we reached a range node (otherwise, we should have been able to find a longer prefix!).

For example, if `queryLCP` searches the LCP of

$$0x11|22|33|44|55|66|77|88$$

and finds the longest prefix in the hash table is

$$0x11|22|33|44|55$$

but also the $0x66$ bit in its child map is set, then `queryLCP` concludes that the subtree of the prefix

$$P = 0x11|22|33|44|55|66$$

is complete and exist in the trie (since otherwise, it should found the prefix P in the hash table).

Large ranges are stored in the top levels bit map arrays in a similar way. If no prefix exists in the hash table (`queryLCP` returns 2) but all bits which corresponds to the first 3 bytes of the key are set then the subtree of those 3 bytes.

Consider the range

$$(0x11|00|00|00|00|00|00|00,\ \ 0xee|ff|ff|ff|ff|ff|ff|ff)$$

For `Insertion` we should set the following bits in the top levels arrays:

1. bits $0x11$ to $0xff$ in the root bitmap array
2. all bits that correspond to the child maps of previous bits on level 1 bitmap
3. all bits that correspond to the child maps of previous bits on level 2 bitmap

The same concept applies to the lower bound operation as well. Whenever the lower bound operation identifies a prefix as a range node, it can deduce that the minimal key (obtained by padding the prefix with $0x00$) within that range exists. This situation happens when the lower bound operation scans a node's bitmap or when it ascends to a higher level of the trie to search for the lower bound

EVALUATION

We divide this section to two comparisons. First, we compare the concurrent MLP Index to the baseline implementation, and then we compare the concurrent MLP Index to Linux Maple Tree which were introduced earlier at subsection 1.2. On our machine we were able to assign up to 28 threads, each to a different core, where there are two sockets, so each socket holds 14 threads. Threads were assigned to sockets incrementally, when there are more than 14 threads it means both sockets are assigned with threads, which should harm performance. In addition, we used `huge pages` to enhance MLP, as needed.

Note that we also measured the concurrent MLP algorithm running with a single thread, so we were able to compare one-threaded-concurrent-MLP-Index to the baseline MLP Index to see the effect of our additions to the baseline implementation. Each attempt was averaged over 5 attempts.

Concurrent MLP Index vs Baseline MLP Index

We employed benchmarks that closely resembled the original MLP paper [6], making slight adjustments to enable multi threading capabilities. In all benchmarks the tire was initialized with 200K elements. The benchmark then performs 2M operations according to the benchmark type. Keys were distributed such that 6 significant bytes were of limited range to encourage dense tree.

In terms of `Exist` and `LowerBound` operations (Figures 4, 5) we demonstrated successfully how the performance improves as a function of the number of threads.

In terms of the `Insert` operation (Figure 3), first recall that our model is a single writer model, which we implemented using a global lock that a writer must acquire. We opted for a spin lock instead of mutex because acquiring and releasing a lock incurred a significant overhead.

The single threaded had 30% overhead compared to the original algorithm, 0.8ms on our algorithm and approx. 0.62ms in the baseline one.

Incrementing the number of threads harmed the Insertion performance as can be seen at Figure 3.

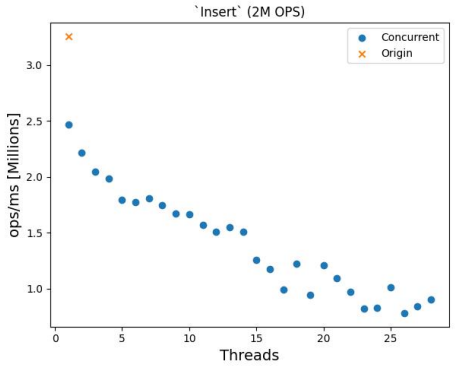

Figure 3. Insert comparison between the concurrent MLP Index and the baseline implementation

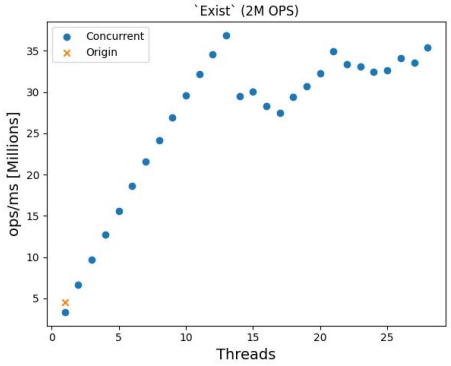

Figure 4. Exist comparison between the concurrent MLP Index and the baseline implementation

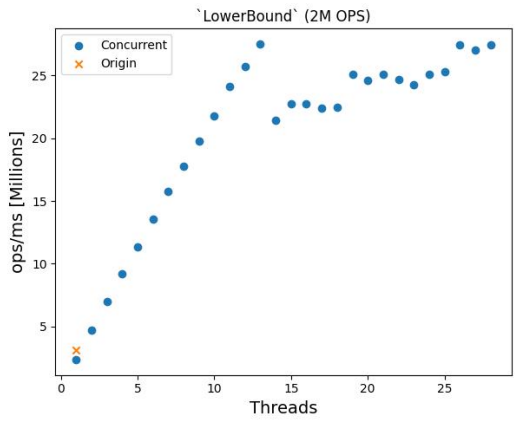

Figure 5. Lower Bound comparison between the concurrent MLP Index and the baseline implementation

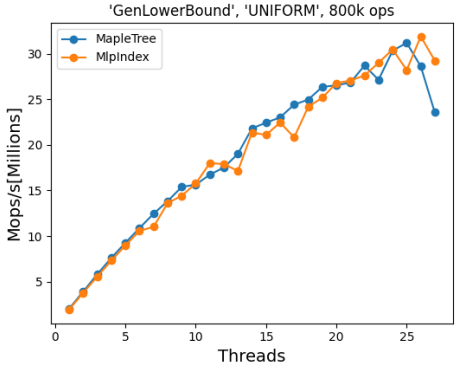

Figure 7. LowerBound comparison between the concurrent MLP Index and the Maple Tree over uniform key distribution

Concurrent MLP Index vs Maple tree

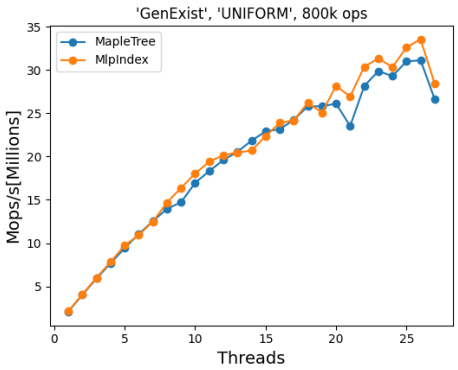

Figure 6. Exist comparison between the concurrent MLP Index and the Maple Tree over uniform key distribution

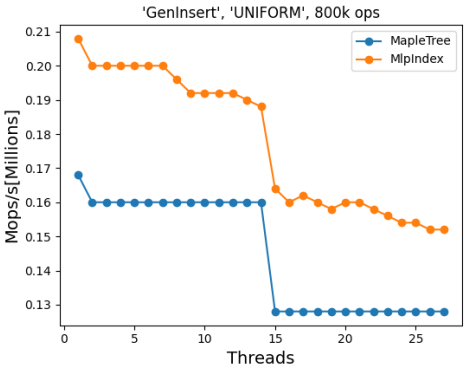

Figure 8. Insert comparison between the concurrent MLP Index and the Maple Tree over uniform key distribution

To benchmark the concurrent MLP Index against Linux's Maple tree, we initially had to extract the latter from the kernel source code and relocate it to user space. To enable the Read-Copy-Update behavior, we built the Maple tree within the user space, in conjunction with Userspace-RCU [5], and it is running on dynamically allocated memory mode.

The benchmarks encompass four distinct workloads. Three of them involve trie operations (`Exists`, `LowerBound`, `InsertRange`), while the fourth is a mixed workload consisting of 25% `InsertRange` operations and an equal distribution of `Exist` and `LowerBound` operations. Each benchmark was measured against two different key distributions: a uniform distribution and another where the four most significant bytes had a limited range (encouraging a denser tree structure). We refer to the latter distribution as "LOW32" since it primarily focuses on the lower 32 bits of the keys. All benchmarks were measured in terms of the average number of operations performed per second. The benchmarks included only did not include singletons.

The benchmarks clearly indicate that we achieved comparable results to the Maple tree in lookup operations and outperformed it in insert operations. It's worth noting, though, that the Maple tree operated in a dynamically allocated mode, which we assume may have introduced an inherent performance penalty compared to the statically allocated MLP Index. However, due to time constraints, we were unable to benchmark it using the alternative mode.

On a positive note, it's worth mentioning that the benchmark did not involve singleton insertions, which are known to be a strength of the MLP Index. Despite this omission, we still achieved comparable performance to the Maple tree.

FUTURE WORK

In summary, our observations indicate that the MLP Index consistently matches the performance of the Maple tree in lookup operations.

1. Additional work is required to enhance the synchronization mechanism within the trie. Specifically, there is an opportunity for improvement in reducing the size of the acquired write set during insertions and the size of the acquired read set during lookups. We anticipate that these efforts may help narrow the performance gap compared to the Maple tree in insertions

2. Future work may involve conducting extensive profiling the runtime behavior, quantifying the extent of memory level parallelism (MLP) within the algorithm, and making comparisons among them. Specifically we were interested to explore the impact of various MLP-related factors, such as the use of huge pages or the prefetcher's ability to retrieve nodes beforehand, on algorithm performance, and especially to try to determine which feature has the most impact on performance
3. There is room to investigate alternative implementations for the 'insert range' operation. One such approach involves inserting only the lower and upper bounds during range insertion while keeping track of whether a node was inserted as a lower bound or an upper bound, similarly to btrees pivots implementations. This approach heavily relies on the Maple tree assumption that there are no overlapping ranges
4. An additional optimization opportunity lies in the type of global lock used to synchronize write operations

REFERENCES

[1] Nickolai Zeldovich Austin T. Clements, M. Frans Kaashoek. 2012. Scalable Address Spaces Using RCU Balanced Trees. *ASPLOS 12* (2012).

[2] Andrew Glew. 1998. MLP yes! ILP no! *Intel Microcomputer Research Labs* (1998).

[3] Liam Howlett. 2021. The Maple Tree, A Modern Data Structure for a Complex Problem. *blogs.oracle.com* (2021).

[4] Linux Maple tree 2023. (2023). https://docs.kernel.org/core-api/maple_tree.html.

[5] URCU library 2023. (2023). https://github.com/urcu/userspace-rcu.

[6] Horan Xu. 2018. Efficient Data Structures Via Memory Level Parallelism. (2018).