Binary search tree with SIMD bandwidth optimization using SSE

Bowen Zhang, Xinwei Li

1.ABSTRACT

In-memory tree structured index search is a fundamental database operation. Modern processors provide tremendous computing power by integrating multiple cores, each with wide vector units. There has been much work to exploit modern processor architectures for database primitives like scan, sort, join and aggregation. However, unlike other primitives, tree search presents significant challenges due to irregular and unpredictable data accesses in tree traversal.

In [1], the authors presented FAST, an extremely **F**ast **A**rchitecture **S**ensitive layout of the index **T**ree, which is a binary tree logically organized to optimize for architecture features like page size, cache line size, and SIMD width of the underlying hardware. FAST eliminates impact of memory latency, and exploits thread-level and datalevel parallelism on both CPUs and GPUs to achieve 50 million (CPU) and 85 million (GPU) queries per second, 5X (CPU) and 1.7X (GPU) faster than the best previously reported performance on the same architectures.

In our project, we are focusing on implementing and analyzing the performance of the binary tree with SIMD width optimization on CPUs.

2. INTRODUCTION

Tree structured index search is a critical database primitive, used in a wide range of applications. In today's data warehouse systems, many data processing tasks, such as scientific data mining, network monitoring, and financial analysis require handling large volumes of index search with low-latency and high-throughput. As memory capacity has increased dramatically over the years, many database tables now reside completely in memory, thus eliminating disk I/O operations. Modern processors integrate multiple cores in a chip, each with wide vector (SIMD) units. Although memory bandwidth has also been increasing steadily, the bandwidth to compute ratio is reducing, and eventually memory bandwidth will become the bottleneck for future scalable performance. We are using Streaming SIMD Extensions (SSE) to optimize the query operation on binary search tree which could utilize the use of SIMD to achieve better performance.

3.ARCHITECTURE SENSITIVE TREE

3.1Motivation

Given a list of (key, rid) tuple sorted by the keys, a typical query involves searching for tuple containing a specific key (key_q). Tree index structures are built using the underlying keys to facilitate fast search operations – with run-time proportional to the depth of the trees. Typically, these trees are laid out in a breadth first fashion, starting from the root of the tree. The search algorithm involves comparing the search key to the key stored at a specific node at every level of the tree, and traversing a child node based on the comparison results. Only one node at each level is actually accessed, resulting in ineffective cache line utilization, to the linear storage of the tree.

Although blocking for disk/memory page size has been proposed in the past [2], the resultant trees may reduce the TLB miss latency, but do not necessarily optimize for effective SIMD utilization/ Recently, 3-ary trees were proposed to exploit the 4-element wide SIMD of CPUs. They rearranged the tree nodes in order to avoid expensive gather/scatter operations. In order to efficiently use the compute performance of processors, it is imperative to eliminate the latency stalls, and store/access trees in a SIMD friendly fashion to further speedup the run-time.

3.2 Hierarchical Blocking

We are building binary trees (using the keys of the tuple) as the index structure, with a layout optimized for the SIMD architecture. In order to cooperate with the feature we rearrange the nodes of the binary index structure and blocking in a hierarchical fashion. Before explaining our hierarchical blocking scheme in detail, we first define the following notation:

E : Key size (in bytes).
K : SIMD width (in bytes).
N : Total Number of input keys.
NK : Number of keys that can fit into a SIMD register.

dN : Tree depth of Index Tree.
dK : Tree depth of SIMD blocking.

In order to simplify the computation, the parameter **NK** is set to be equal to the number of nodes in complete binary sub-trees of appropriate depths 3. Consider Figure 1 where we let **N** = 31, **dN** = 5 and **dK** = 2. Figure 1(a) shows the indices of the nodes of the binary tree, with the root being the key corresponding to the 15**th** tuple, and its two children being the keys corresponding to the 7**th** and 23**rd** tuples respectively, and so on for the remaining tree. Traditionally, the tree is laid out in a breadth-first fashion in memory, starting from the root node. For our hierarchical blocking, we start with the root of the binary tree. The first **NK** elements are laid out in a breadth-first fashion. Thus, in Figure 1(b), the first three elements are laid out, starting from position 0. Each of the **(NK + 1)** children sub-trees (of depth **dK**) are further laid out in the same fashion, one after another. This corresponds to the sub-trees (of depth 2) at positions 3, 6, 9 and 12 in Figure 1(b). Our framework for architecture optimized tree layout preserves the structure of the binary tree, but lays it out in a fashion optimized for efficient searches.

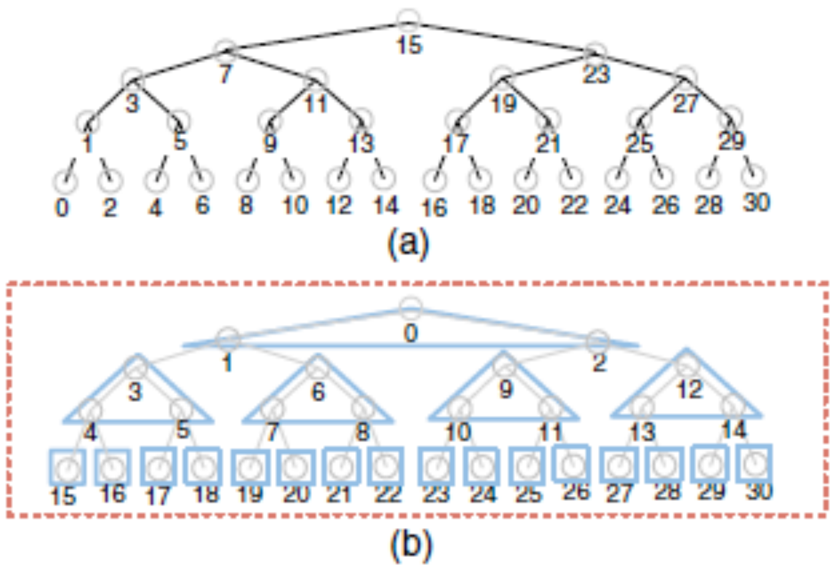


Figure 1. (a) Node indices (=memory locations) of the binary tree. (b) Rearranged nodes with SIMD blocking.

4.Streaming SIMD Extensions

The reason we are grouping three nodes into one, is that the structure could fit in Streaming SIMD Extensions (SSE).

SSE is a SIMD instruction set extension to the x86 architecture, which contains 70 new instructions, most of which work on single precision float point data. SIMD instructions can greatly increase performance when exactly the same operations are to be performed on multiple data objects. To cooperate our project with SSE. We defined key as 4bytes float numbers, and using __m128 as the node type.

__m128 is a 16 bytes vector, which could store four float numbers and do instructions on all four number at same time. A variable of type **__m128** maps to the XMM[0-7] registers. We have used following instructions in our implementation of the data structure:

void _mm_store_ps(float *p, __m128 a)	Stores four single-precision, floating-point values
__m128 _mm_loadu_ps(float * p)	Loads four single-precision, floating-point values.
__m128 _mm_load1_ps(float * p)	Loads a single single-precision, floating-point value, copying it into all four words.
__m128 _mm_cmpgt_ps(__m128 a, __m128 b)	Compares for greater than.
__m128 _mm_cmpeq_ps(__m128 a , __m128 b)	Compares for equality.
int _mm_movemask_ps(__m128 a)	Creates a 4-bit mask from the most significant bits of the four single-precision, floating-point values.

5.Implementation

5.1 Building the tree

Given a sorted input of tuples (Ti , i∈ (1..N), each having 4-byte (key, rid)), we layout the index tree (T) by collecting the keys from the relevant tuples and laying them out next to each other. We set dN = log2 (N) . In case N is not a power of two, we still build the perfect binary tree, and assume keys for tuples at index greater than N to be equal to the largest key (or largest possible number).

To build the blocking tree, we need first generate the normal tree. The data structure we used to generate is like heap, child of node i is stored at node i*2+1 and i*2+2. The algorithm we generating the normal tree is bottom-up, we have the bottom level nodes at first (tuple), we calculate keys of node in one level up by:

```
Normal_Tree_Key[i] =( Normal_Tree_Key[i*2+1]+Normal_Tree_Key[i*2+2] ) / 2
```

We keep the procedure until we got key[0]. After that we got the complete tree like figure1(a), in an array.

Then we generate blocking tree using the normal tree, each odd level node and two of their children make up and new node.

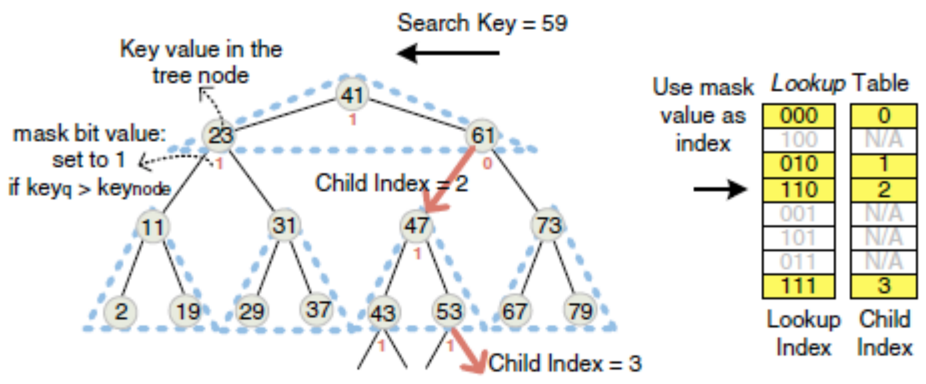
```
F[4] = {Normal_Tree_Key[i], Normal_Tree_Key[i*2+1], Normal_Tree_key[i*2+2], 0}

Fast_Tree_Key[i] = _mm_loadu_ps( &Normal_Tree_Key[i*3])
```

5.2 Query the tree

Given a search key (key_**q**), we now describe our SIMD friendly tree traversal algorithm. For a query of key_**q**, we begin by splatting key_**q** into a vector register (i.e.,replicating key_**q** for each SIMD lane), denoted by xmm_key_q. We start the search by comparing the root of the tree (assign to index) and xmm_key_q, the result is assign to a mask register. Then we compute an integer value (termed mask) from the mask register. The mask is then looked up into an lookup table, that returns the child index(assign to index). If not reach the bottom layer, repeat the procedure

```
Step 1:   xmm_key_q = _mm_load1_ps(&key_q)
Step 2:   index = root
Step 3:   xmm_mask = _mm_cmpeq_ps(xmm_key_q, V_tree[index])
Step 4:   mask =   _mm_movemask_ps (xmm_mask)
Step 5:   index = index*4 + lookup[mask]
Step 6:   if index is not reach the bottom layer of the tree back to step 3
```

Since **d**K = 2, there are two nodes on the last level of the SIMD block, that have a total of four child nodes, with local ids of 0, 1, 2 and 3. There are eight possible values of mask, which is used in deciding which of the four child nodes to traverse. Hence, the lookup table has 2^NK (= 8) entries, with each entry returning a number & [0..3]. Even using four-bytes per entry, this lookup table occupies less than one cache line, and is always cache resident during the traversal algorithm.



Lookup Index	Child Index
000	0
100	N/A
010	1
110	2
001	N/A
101	N/A
011	N/A
111	3

Figure2. Example of SIMD(SSE) tree search and the lookup table.

In Figure 2, we depict an example of our SSE tree traversal algorithm. Consider the following scenario when keyq equals 59 and (key_q > V_tree[0]), (key_q > V_tree[1]) and (key_q < V_tree[2]). In this case, the lookup table should return 2 (the left child of the second node on the second level, shown in the first red arrow in the figure), and the new index should be 2 = 0(old_index)+1(offset)+ lookup[3].   For this specific example, mask **(**[1, 1, 0] and hence index would be 1(20) + 1(21) + 0(22) = 3. Hence Lookup[3] **(** 2. The other values in the lookup table are similarly filled up. Since the lookup table returns 2, child_index for the next SSE tree equals 2. Then, we compare three nodes in the next SSE tree and Vmask **(**[1, 1, 1], implying the right child of the node storing the value 53, as shown with the second red arrow in the figure. We now continue with the load, compare, lookup and offset computation till the end of the tree is reached. After traversing through the index structure, we get the index of the tuple that has the largest key less than or equal to key_q.

As compared to a scalar code, we resolve **d**K (= log2(NK +1)) levels simultaneously. Hence theoretically, a maximum of 2X (=**d**K) speedup is possible (in terms of number of instructions).

6. Performance Analysis

We did benchmark for SIMD optimized FAST on energon1.cims.nyu.edu server, whose architectural parameters relevant to the performance are listed below.

CPU Name: Intel Xeon L5320
Number of Cores: 4
SIMD width: 4
Cache Size: 4096KB
Cache Line Size: 64B
Page Size: 4096B

A list of tuples are generated with 32-bit key and value. The tuples are sorted according to the key. Then a normal tree and a FAST are constructed based on it. The size of tuple varies from 2^6 to 2^22, corresponding to tree size that varies from 1KB to 64MB. One hundred thousand searches are run for each tree. Search keys are generated randomly to avoid coherence between subsequent searches.

We run the benchmark for 10 times and average the time consumed. Each benchmark involves 10^5 random searches applied on both trees. The result of benchmark is shown in table 1. Search performances are compared between the normal tree and on the optimized FAST in Figure 3.

Tree Size	1KB	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
Normal Tree Elapsed Time (s)	0.0293	0.0356	0.0389	0.0440	0.0533	0.0591	0.0761	0.0936	0.163
FAST Tree Elapsed Time (s)	0.0228	0.0267	0.0330	0.0420	0.0480	0.0547	0.0739	0.1032	0.165

Table 1. Time consumed by searches on trees of different sizes

The benefit of SIMD optimization on search is more noticeable for small trees than large trees. The reason is that small trees do not suffer as much latency of memory I/O as large trees. We could observe, from table 1, that, for tree size of 1KB and 4KB, searches on FAST are 22% and 25% faster than normal tree, respectively. When tree size is smaller than page size (4KB), the number of accessed memory pages is minimized. There is little memory latency. Thus, the data-level parallelism exploited by SIMD optimization gains performance by 20%-30%. This performance improvement shows agreement with what addressed in [1]. However, with tree size increasing, the performance difference between FAST and normal tree gradually shrinks and finally disappears. Probably it results from that search on small trees is compute bound while search on large trees is not. When tree size become larger than page size or even cache size, it is necessary to bring in data from memory after only a few computations. Access to memory also need more address computation for large trees. In this situation, the latency coming from cache or memory access becomes performance bottleneck. Then search on large trees switches from compute bound to latency bound, thus it does not exploit additional compute resources provided by SIMD instructions.

To verify the above hypothesis about performance, we ran the benchmark again with Linux profiling tool “perf” and tried to find out the primary reason of the performance decrease for large trees. 10^7 searches were carried out in the benchmark to amplify the possible effect caused by the hardware events. A variety of hardware events were analyzed, including instructions, cache misses and page faults. Finally, it turned out that elapsed time showed similar trend with number of bus cycles, as shown in table 2 and

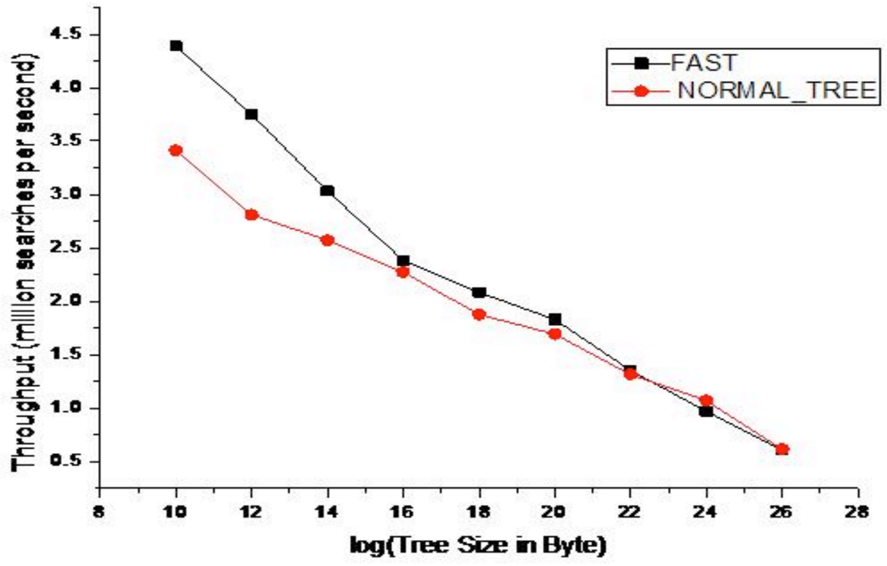
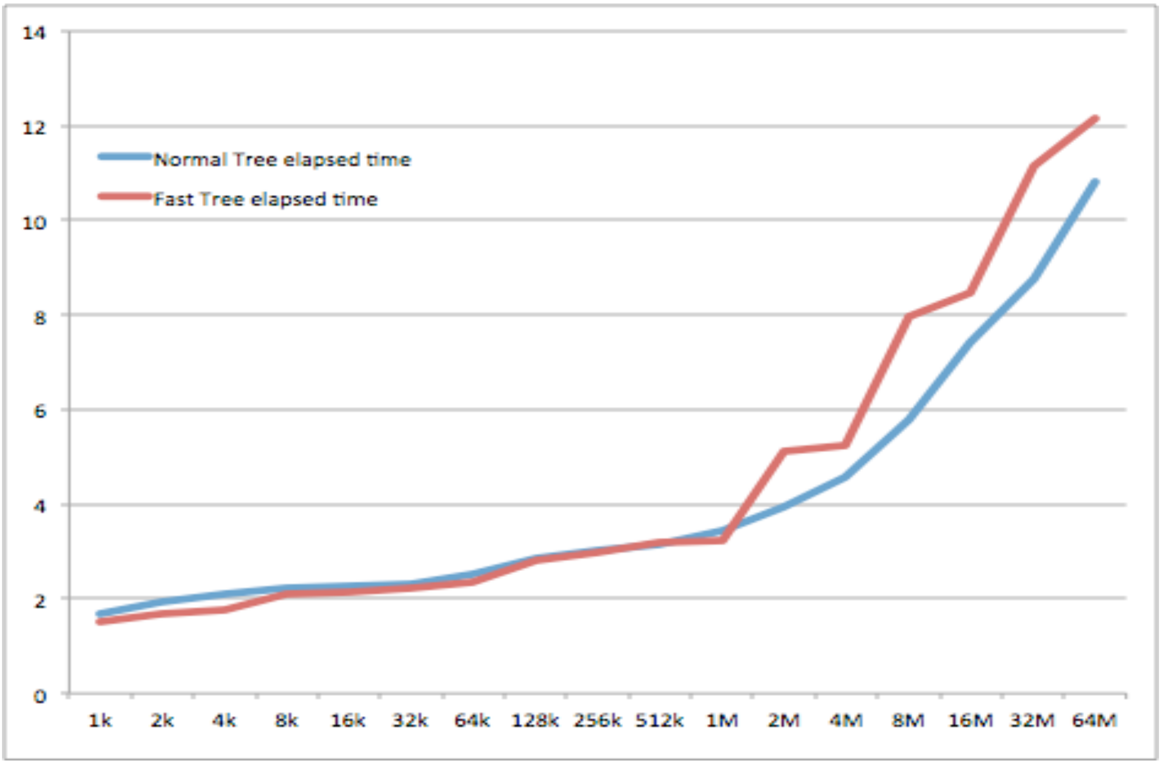


Figure 1. Throughput (million searches per second) of Normal Tree and Fast Tree

tree size	Normal Tree bus cycle	Fast Tree bus cycle	Normal Tree elapsed time	Fast Tree elapsed time
1k	430,502,606	382,000,321	1.663833406	1.503150901
2k	460,804,589	425,113,382	1.951052939	1.668005499
4k	498,377,832	451,082,235	2.115594736	1.766830354
8k	535,990,005	497,048,714	2.240976779	2.098979892
16k	574,567,859	514,587,931	2.256231695	2.158809217
32k	615,431,845	595,555,330	2.330204714	2.246055118
64k	660,218,516	656,586,118	2.515943729	2.36383442
128k	708,707,604	701,287,043	2.869222885	2.826380779
256k	761,954,913	715,888,286	3.030701338	2.988343545
512k	799,936,717	813,487,022	3.130861292	3.19864613
1M	897,151,742	831,481,712	3.448993999	3.226389534
2M	993,904,764	1,268,572,088	3.952918245	5.102762455
4M	1,187,185,549	1,356,143,392	4.588148258	5.261571795
8M	1,509,162,112	2,036,208,446	5.792797421	7.968207739
16M	1,852,235,495	2,186,256,315	7.441925196	8.467376134
32M	2,091,114,732	2,842,514,412	8.743061693	11.13028249
64M	2,837,744,858	3,120,498,274	10.83473015	12.14902355

Table 2. Elapsed Time and number of Bus Cycles generated by benchmark with "perf stat event=bus cycle"

figure 3. Because the SIMD optimization search requires that four tree nodes are computed simultaneously, the processor needs to read four tree nodes from memory for every two levels. For normal tree search algorithm, it is necessary to read two tree nodes from memory for every two levels. Therefore, SIMD optimization tree search have to read more tree nodes from memory than normal tree search. When tree size is small, the whole tree may be located in one or a few pages, not many bus cycles are needed even SIMD optimization search requires more tree nodes. However, when tree size increases, tree nodes are distributed very dispersedly in memory. Thus, the number of bus cycles called by SIMD optimization search is increasing rapidly and surpassing normal search.



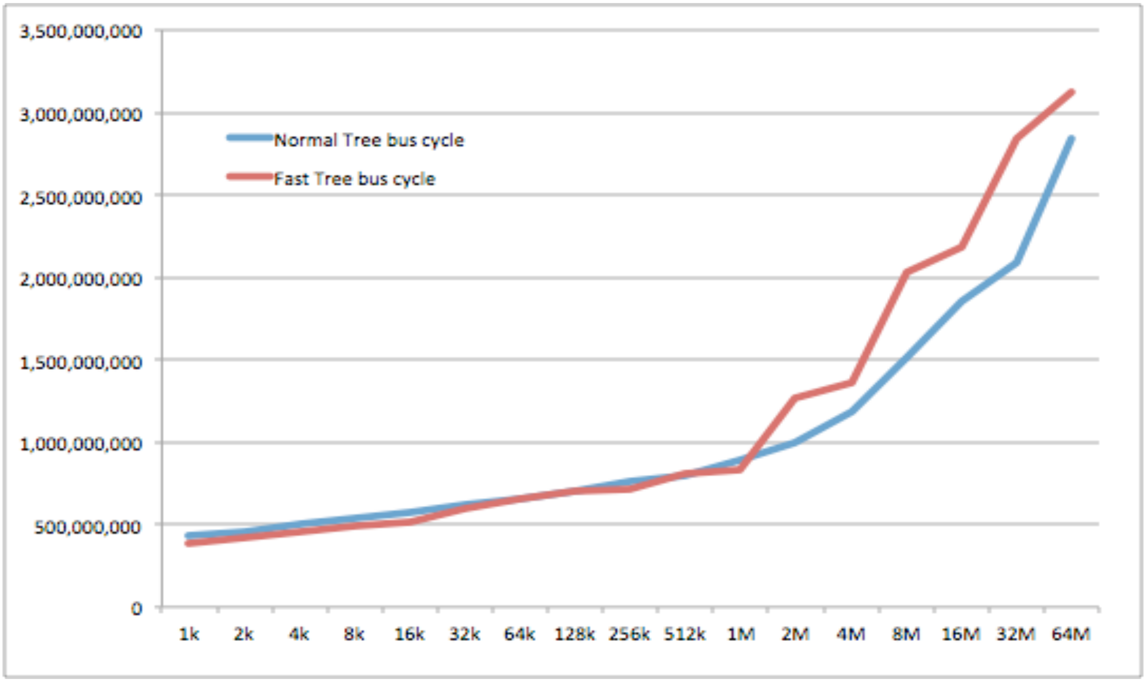


Figure 3. Elapsed Time and number of Bus Cycles (According to data from table 2)

SIMD optimization search is advantageous in hardware events other than bus cycles. For example, there are fewer instructions, CPU clocks in SIMD optimization search. Although these differences are tiny, they still help SIMD optimization search. Therefore, in conclusion, the analysis result with perf partly proves our hypothesis we raised about search performance. It clearly shows that the failure of SIMD optimization search in large trees is resulted from the large amount of memory I/O.

7 Future work.

Since Fast tree algorithm's bottleneck is memory I/O, we propose following ways to improve performance.

Because __m128 data type could store four float numbers and we only used three of them, to utilize the memory space, we could reconstruct the FAST tree that each node contains four nodes of normal tree. The total number of nodes in FAST would be reduced, hopefully that could improve memory efficiency.

The cache line blocking and page blocking techniques reported by [1] seem to be helpful to this problem. We could add more hierarchy block (cache line size, page size) to FAST to utilize the hardware.

Reference

[1] Changkyu Kim. *et al* FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs., 2010
[2] Comp D. Comer. Ubiquitous b-tree., 1979.