École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Spiderman: Software Prefetching and Request Multiplexing to Mask Memory Accesses in B+trees

Author:

Thang Long Vu (thang-long.vu@polytechnique.edu), École Polytechnique

Advisor:

Antonios Katsarakis (antonios.katsarakis@huawei.com), Huawei Edinburgh

Academic year 2022/2023

Abstract

In-memory B+trees are ubiquitous data structures that are essential for many applications, including modern file systems, key-value stores and databases. To meet the performance demands of these applications, state-of-the-art B+tree designs rely on multi-threading and efficient synchronization schemes to increase request concurrency. However, these B+tree designs stall their request processing on every request each time they encounter memory accesses on their B+tree traversal.

In this thesis, we introduce Spiderman, a traversal mechanism that couples software prefetching and request multiplexing to improve the throughput of B+tree operations. To achieve that, the Spiderman technique does "meaningful work" during the costly waiting time of memory access. More specifically, instead of waiting for a node memory access for a request, it issues an asynchronous prefetch of the memory data of the requested node (hence the phrase **"software prefetching"**) to the caches, and switches to other requests to make progress on while waiting for the data to move from main memory to caches (hence the phrase **"request multiplexing"**), before returning back to process the original request.

We evaluate the effectiveness of the Spiderman technique on the current state-of-the-art B+tree design, and our results demonstrate up to 30% higher throughput. Our findings highlight the potential of the Spiderman technique for enhancing the performance of in-memory B+trees, which are critical for many modern applications.

Acknowledgement

I would like to thank my mentor, Antonis Katsarakis, for guiding me on my Bachelor thesis. I am more than immensely grateful for his patience in answering my questions, for his helpfulness and invested time in our daily work, and for his valuable lessons in knowledge and programming. During these past 2 months, I have learned a lot from Antonis, and I am inspired by his passion, discipline, and positivity in everyday work. I am amazed by his vast, deep technical knowledge and his creative ideas. Overall, I feel really lucky to have the opportunity to work with Antonis and be under his mentorship for my Bachelor thesis. Collaborating with Antonis makes my internship at Huawei Edinburgh so much more enjoyable.

Besides my mentor, I would also like to thank the database team's manager, professor Nikos Ntarmos, for his helpful advice and tips, and for guiding me on choosing the thesis topic. I am also grateful for the nice, seamless recruiting and onboarding experience in my journey to Huawei Edinburgh for my Bachelor Thesis by HRBP Bo Pang and Junior HR Specialist Natalia Pangeiou.

Contents

1 Introduction

1.1 Motivation

In-memory B+trees [12, 13] are ubiquitous data structures that are essential for many applications, including modern file systems, key-value stores and databases. To meet the performance demands of these applications, some state-of-the-art B+trees (e.g., B+tree OLC [9], Masstree [21], OpenBw-Tree [9]) designs rely on multi-threading, and efficient synchronization schemes to increase request concurrency. However, these B+tree designs stall their request processing on every request each time they encounter memory accesses, which leaves performance on the table. This is due to poor memory locality when randomly traversing a B+tree (since nodes in B+trees are randomly allocated), and the data structure consists of multiple levels of nodes that typically do not fit in the cache. A straightforward idea to mitigate the overhead of memory accesses is to overlap work and enable higher utilization, as done by the CPU scheduler that switches to other tasks after initiating an I/O operation.

In this thesis, we introduce Spiderman, a traversal mechanism that couples software prefetching and request multiplexing to improve the throughput of B+tree operations. Briefly, this traversal mechanism, on a suspected memory access during a request traversal it prefetches the targeted address and context switches to the traversal of other operations. As such, it overlaps the waiting time of costly memory accesses of a request with useful work on other requests. When we come back to make progress on the original request, we face less (or no) memory access latency since part (or all) of the memory access work has been overlapped with useful work.

We evaluate the effectiveness of the Spiderman technique on the current state-of-the-art B+tree (B+tree OLC), and our results demonstrate significant performance improvements. Our findings highlight the potential of the Spiderman technique for enhancing the performance of B+trees, which are critical for many modern applications.

1.2 Contributions of the Thesis

In short, the main contributions of this thesis are:

1. We highlight that B+trees leave performance on the table as the request processing stalls (on each request) due to memory accesses.
2. We introduce Spiderman, a technique that overlaps work during a potential memory access using the combination of software prefetching and request multiplexing.
3. We implement and evaluate Spiderman over the state-of-the-art B+tree variant using workloads that follows the YCSB [7] mix of operations and show up to 30% throughput improvement.
4. We provide limitations and future work of the technique to be addressed.

1.3 Thesis Structure

The thesis is organized as follows:

- Chapter 2 provides background knowledge on the memory hierarchy, B+tree and related work to B+tree.
- Chapter 3 explains the Spiderman optimization idea (Software Prefetching & Request Multiplexing) and its benefits.
- Chapter 4 details our evaluations of Spiderman over the state-of-the-art B+tree.
- Chapter 5 discusses the limitations and the future work of the technique.
- Chapter 6 summarizes the contributions of the thesis and concludes the thesis.

2 Background and Related Work

2.1 CPU access costs on different memory levels

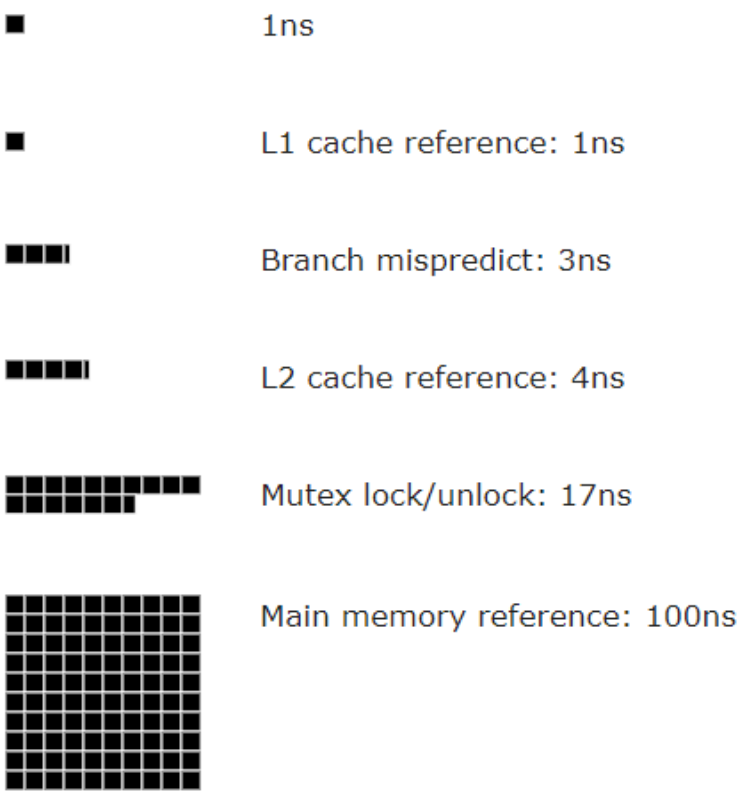

Figure 1: Visualization of different storage levels latency (adapted from [11]).

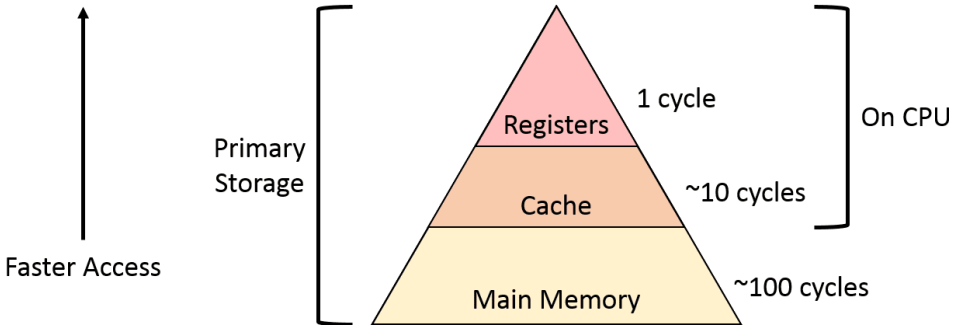

Figure 2: The memory hierarchy (adapted from [6]).

As shown in Figures 1 and 2, it is pretty clear that the latency of memory access is huge compared to that of caches in the memory hierarchy. In the famous visualization "Latency numbers every programmer should know" [11], we can see that a main memory reference is 25x times bigger than an L2 cache reference, and 100x times bigger than an L1 cache reference. Having a lot of memory accesses in a program can hurt its performance quite considerably. Accessing the caches for data is comparable to a cycle of CPU work (1ns), but accessing the main memory for data is not and is quite visible.

Software prefetching is a way to mitigate this visible latency of memory access by loading the cache lines asynchronously in advance before it is actually needed. This is done by the programmers manually inserting prefetch instructions into the programs through the GNU GCC's built-in function `__builtin_prefetch` [5].

2.2 Concurrent in-memory ordered key-value stores

A key-value store holds a collection of key-value pairs, in which the keys are uniquely stored. The data type of the keys and the values can be anything. A typical example of a key-value store in real life is a dictionary, where a word is associated with its meaning, and the dictionary stores unique words.

An ordered key-value store keeps the keys in sorted order (e.g., words in the dictionary are sorted by lexicographic order). This is important to efficiently support range queries (also known as scans), as we don't have to traverse the whole collection to find all the key-value pairs such that the keys lie in a given interval. Instead, keys in a given interval lie next to each other. Consequently, the time complexity of a scan is only proportional to the number of keys in the query interval instead of the number of keys in the key-value store.

The *in-memory* property tells us that the size of the stored collection of key-value pairs fits inside the limit of RAM storage of the system.

The *concurrent* property means multiple threads can operate with the data structure, while the data structure is able to synchronize and has no data race.

2.3 B+tree

In this work, we focus on B+trees as they are the most popular ordered key-value stores. Thus, we describe B+trees next.

2.3.1 Definition and motivating problem

B+trees are widely used in main-memory database systems and file systems [14].

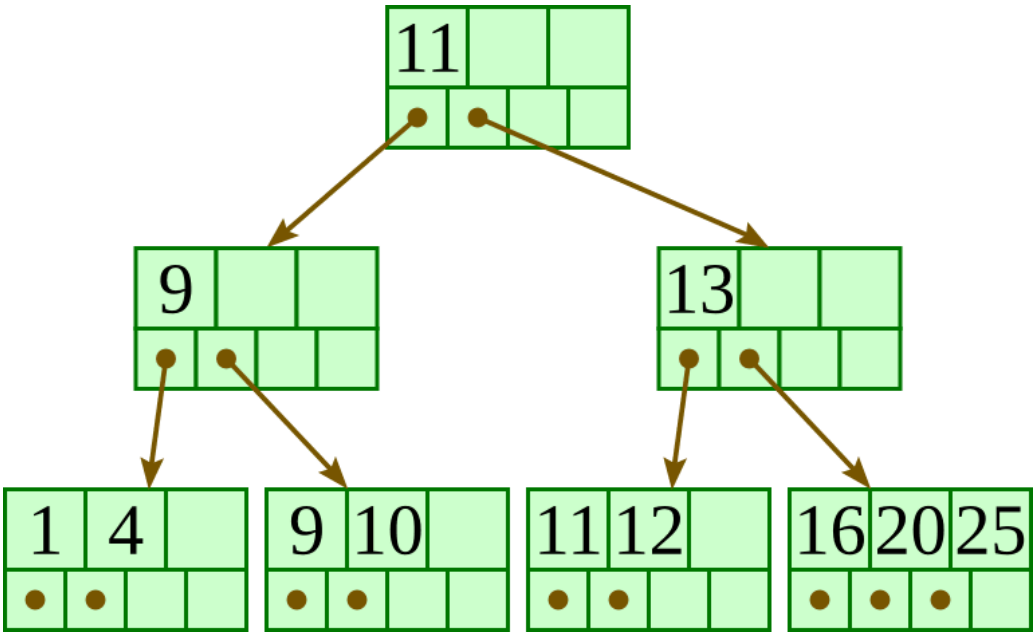

Figure 3: An example of a B+tree with maximum 4 children and 3 levels (adapted from [3]).

B+tree is an ordered key-value store, a self-balancing m-ary tree (one can think of B+tree as a generalization of a binary search tree, a B+tree with 2 children), m is fixed a priori. The data structure in Figure 3 is a B+tree with m equals to

4. It supports insert, delete, get, and range query operations. There are inner nodes and leaf nodes. Inner nodes can have up to *m* pointers to children. The number of keys that nodes have is the number of children minus one. The leaves are similar, but instead of storing children, they store values. B+trees are highly used in file systems and databases because they have a high fan-out. Having many children reduces the number of layers of the tree, hence reducing the total DRAM latency due to cache misses from accessing random memory addresses of nodes. A node of a B+tree spans multiple consecutive cache lines and holds information about the type (inner or leaf), the keys array, and the children/payloads array. Although its high fanout alleviates the memory accesses, typical B+trees are large; thus, they still encounter memory accesses during their traversal.

2.3.2 B+tree's node structure

Below is the list of data that a node of a typical B+tree may contain:

1. $atomic < uint64\_t > typeVersionLockObsolete$, this variable contains the lock and the version counter of the node, which is used for synchronization by the Optimistic Lock Coupling technique (section 2.3.4).
2. $uint8\_t\ type$, type of the node, either a inner node ($type = 1$) or a leaf node ($type = 2$).
3. $uint16\_t\ count$, number of children/values of the node.
4. $static\ const\ PageType\ typeMarker$, a constant variable equals to the inner node type ($type = 1$) in the $struct$ of inner node and equals to the leaf node type ($type = 2$) in the $struct$ of a leaf node, used to set the value of variable $type$.
5. $static\ const\ uint64\_t\ maxEntries$, maximum number of entries the node can have.
6. $KeyType\ keys[maxEntries]$, the array of key values of the node.
7. $Node^*\ children[maxEntries]$ or $PayloadType\ payload[maxEntries]$, the array of children or the array of payloads of the node.

Note that the header of the node ($typeVersionLockObsolete$, $type$, $count$) lies in the first cache line of the node (cache line size is 64B in a typical modern computer environment). This will be utilized in the Spiderman technique.

2.3.3 Traversal in B+tree

Operations in B+tree have a common traversal pattern. That is, first, they check if the type of the node is an inner node or a leaf node. In both cases, a binary search takes place, based on the given search key on the node's sorted key array, to find the corresponding child node in the case of the inner node, or the corresponding payload (also known as value) in the case of the leaf node. In the case of the inner node, the same process is repeated for the child node until a leaf node is reached. This is illustrated in Algorithm 1 below.

Data: $traverseWithKey$
Result: Nothing
$curNode \leftarrow root$;
while $curNode.type$ *is innerNodeType* **do**
  $curNode \leftarrow curNode.children[curNode.keyBinarySearch(traverseWithKey)]$;
end
$leafValue \leftarrow curNode.payload[curNode.keyBinarySearch(traverseWithKey)]$;

Algorithm 1: Traversal in B+tree.

2.3.4 Making B+ Tree concurrent: Optimistic Lock Coupling (OLC) technique

The optimistic lock coupling (OLC) method [19] is a practical alternative to traditional locking techniques and lock-free approaches. It has high scalability, as the lock-free approaches, while being as easy to implement and apply as the traditional locking technique. In short, the idea behind optimistic lock coupling is that each node has a lock and a version

counter. Instead of acquiring locks eagerly, read operations validate version counters (and restart if the counter changes), while write operations grab the locks, modify the data of the node as well as the version number, and then unlock. This technique has been used to make the in-memory concurrent tree data structures such as ART [18] and B+tree [9] work simply and efficiently with multiple threads. The idea of the method is illustrated in Figure 4 below.

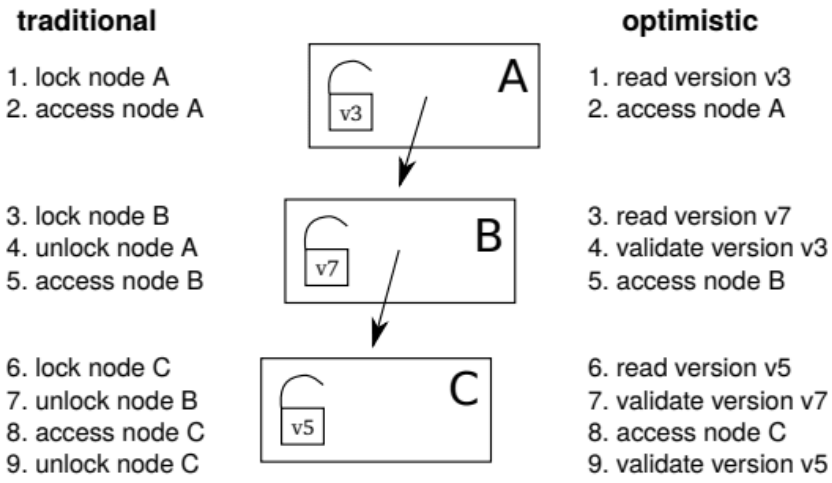

Figure 4: Comparison of a lookup operation in a 3-level tree using traditional lock coupling (left-hand side) vs. optimistic lock coupling (right-hand side) (adapted from [19]).

2.4 Observations on state-of-the-art concurrent in-memory ordered key-value stores

	Bw-Tree	OpenBw-Tree	SkipList	Masstree	B+Tree
L1 Miss	5.0	3.2	14	2.2	4.3
L3 Miss	2.0	1.4	6.3	0.68	1.7

Figure 5: For B+trees (Bw-Tree [15], OpenBw-Tree and B+tree), each request sees more than one memory access (adapted from [9]).

State-of-the-art implementations like B+tree OLC still suffers from accessing main memory during their operations, as depicted in Figure 5 that summarizes the results of the microbenchmark of L3 cache misses performed by the OpenBw-Tree paper [9] causes the program to stall. They can not hide the memory access latency effectively. Each and every B+tree operation imposes equivalent overhead and thus leaving performance on the table. In this paper, we address this memory access waiting time problem by overlapping meaningful work with the memory access, thus effectively hiding its costly long latency.

3 Spiderman: Software Prefetching and Request Multiplexing

In this section, we detail Spiderman, our solution to the memory access issue in B+tree traversals.

3.1 Core idea of the technique

Each thread is given a number of operations on the tree to execute. Each operation on the tree (insert, get, scan,...) traverses through several nodes of the tree, which we can imagine as a sequence of node memory accesses. In search tree data structures, the address of the next node is random and sometimes results in cache misses. An L3 cache miss can cause a memory access, which is very slow compared to a cache access. Instead of just waiting for this memory access and doing nothing while accessing a node, we do meaningful work. Our idea is as follows, we process a group of incoming requests simultaneously for a thread once at a time (also known as a batch of requests, the batch size is the number of requests in a batch that we choose).

The algorithm is divided into 2 phases (also shown in Algorithm 3):

1. In the first phase, we follow the path of the tree that the request normally follows but in a clever manner in order to mitigate the effect of memory access waiting time and to fetch all the information about the nodes that we want in the caches. We will discuss in more detail what this clever manner is in subsection 3.1.2.

2. In the second phase, we process each request in the batch sequentially as normal, but this will be much faster given that the required data of the nodes (node type, node's key array for binary search,...) that each request will most likely access are already in the caches as they were fetched in the first phase.

3.1.1 Prefetch function

Before we dive into the first phase of the algorithm, we will talk about an important function of the technique that we will use heavily, `__builtin_prefetch` [5], the main argument is the starting memory address of a cache line that we want to fetch from. The function then fetches this cache line in the background.

Normal memory accesses are blocking, which means we wait until the memory accesses is finished for the program to move forward while `__builtin_prefetch` is non-blocking. The process of fetching the memory content is done in the background. This is very useful since, while waiting for the memory content to be fetched, we can do other work and then come back to access the memory content when it has been fetched. If it is not finished (e.g. the memory content fetching process is not 100% done), we just need to pay the cost for the remainder of the fetching process, not from scratch, which is still a win. For instance, let's say fetching the content of memory address A takes 200 ns, the `__builtin_prefetch` was called on A, and it has been 150 ns, then when we access the memory address A, we only need to take an additional 50 ns, not 200 ns.

The only downside of the function is that it comes with a little overhead. Also, it is also not easy to check whether the right cache line that we want is fetched, so we need to be careful when using it in our program.

Since `__builtin_prefetch` only fetches a cache line, we can create an auxiliary function `Prefetch`, as shown in Algorithm 2, given an address and a number of cache lines, fetch all the consecutive cache lines starting from the address.

Data: $addr, numPrefetchCacheLines$
for $i \leftarrow 0$ **to** $numPrefetchCacheLines - 1$ **do**
  $cacheLine \leftarrow static\_cast < const\ uint8\_t* > (addr) + (CACHE\_LINE\_SIZE * i)$;
  $\_\_builtin\_prefetch(cacheLine)$;
end

Algorithm 2: `Prefetch` function.

3.1.2 The first phase of the algorithm

We traverse the path of each request in a similar fashion as the "round-robin" CPU scheduler. We prefetch the data of the current node on the traversal path of the first request, switch to the next request and prefetch the next request's current node, and so on. Once we come back to the first request, then we access the data of the first request's current node, as usual, to determine the next node, which is the child node of the first request's current node and then prefetch that child node which is set to be the first request's new current node, and so on.

Similarly note that the first phase is a best effort algorithm that doesn't modify the tree, we can safely remove the part of the code that handles synchronization from the OLC technique, as in the worst-case we would prefetch the wrong addresses but will not compromise the correctness of the B+tree.

The traversed path of a request in the first phase might be different from the traversed path of a request in the second phase since, in the second phase, any insert operations will be taken into account and modify the tree after the operations are executed, so operations after the modification operations may go down a different path than in previous phase. Since this is unlikely, we ignore this problem for now and assume that the traversed path in the first phase and second phase share many common nodes.

3.1.3 The second phase of the algorithm

At this point the majority of the nodes have their important data in the cache (fetched by the first phase), we process each request in order. To do this, we have a struct that contains the important metadata about the request (the type of request, the key of the request, etc.), and store also in the struct, the outcome of that request, if any (a get request will return the value, a scan request will return the array of values, etc.). The request array is passed by reference, so we return the results of requests by modifying the request array.

```
Data: reqs (batch of requests), numReqs (number of requests)
Result: Nothing
// Phase 1 of the algorithm
for i ← 0 to numReqs - 1 do
    traversalDone[i] ← false;
    curNode[i] ← root;
end
numTraversalDone ← 0;
curReq ← 0;
while numTraversalDone < numReqs do
    if traversalDone[curReq] is true then
        curReq ← (curReq + 1) mod numReqs;
        continue;
    if curNode[curReq].type is innerNodeType then
        curNode[curReq] ←
          curNode[curReq].children[curNode[curReq].keyBinarySearch(reqs[curReq].key)];
        // Prefetching the whole node
        Prefetch(curNode[curReq], ⌈sizeof(BTreeNode) / CACHE_LINE_SIZE⌉);
    else
        destinationNodeIndex ← curNode[curReq].keyBinarySearch(reqs[curReq].key);
        traversalDone[curReq] ← true;
        numTraversalDone+ = 1;
    end
    curReq ← (curReq + 1) mod numReqs;
end
// Phase 2 of the algorithm
for i ← 0 to numReqs do
    req ← reqs[i];
    if req.type is lookup then
        req.ret.success, req.ret.value ← lookup(req.key);
    end
    if req.type is insert then
        insert(req.key, req.value);
    end
    if req.type is scan then
        req.ret.count, req.ret.outputArray ← scan(req.key, req.range);
    end
end
```

Algorithm 3: The Spiderman technique, prefetching the whole node.

3.1.4 Signs of potential performance increase of the technique

Before implementing the Spiderman technique, for a batch of requests, we investigated that phase 1 has the majority of cache misses of the requests, which creates many work overlapping opportunities. We speculate that most of the cache misses happen in phase 1, and most of these cache misses are from the path traversal of requests. To confirm this, for the batching function, before executing each request, we traverse the predicted paths of the requests. By predicted path, we mean traversing the path as if the tree is not modified by any operations in the batch. In the second phase, we execute the requests as usual sequentially. We used the Linux `perf` [8] to profile the cache misses of the program to see which functions are causing the majority of cache misses during the whole duration of executing a workload of 50M inserts and 500M reads (also known as lookups). As suspected, shown in Figure 6, the number one source of cache misses comes from the traversal function of phase 1 with second place goes to the function's cache misses. In the traversal functions of phase 1, most of the cache misses happen in the lower-bound function on the key array of leaf nodes, and accessing the types of the nodes to see whether it is an inner node or leaf node. Both of these actions have high likelihood of causing a cache miss, because accessing the type of the node is the first data we access of a node if there is no locks or version counters, and leaf node layer has the highest cardinality of all layers in term of nodes and at the lowest level of the B+tree.

The `perf` commands that we used are

1. *sudo perf record -c1373 -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRE.L3_MISS/ppp*, to record the cache misses.
2. *sudo perf report -M intel –no-children*, to generate the report of the location of cache misses.

```
Samples: 2M of event 'MEM_LOAD_RETIRE.L3_MISS', Event count (approx.): 3491621380
Overhead  Command         Shared Object        Symbol
  48.01%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTreeLeaf::binarySearchInFirstPhaseTraversal
  17.41%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTree::lookup
  14.16%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTree::firstPhaseTraversal
   7.12%  workload-debug  libc-2.31.so         [.] __memmove_avx_unaligned_erms
   4.36%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTreeLeaf::lowerBoundInLeafInsert
   3.12%  workload-debug  [kernel.kallsyms]    [k] change_pte_range
   2.62%  workload-debug  workload-debug       [.] exec(int, int, int, int, std::vector<unsigned long, std::allocato
   1.40%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTree::insert
   0.77%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::OptLock::readLockOrRestart
   0.29%  workload-debug  [kernel.kallsyms]    [k] do_numa_page
   0.28%  workload-debug  workload-debug       [.] exec(int, int, int, int, std::vector<unsigned long, std::allocato
   0.12%  workload-debug  [kernel.kallsyms]    [k] copy_user_enhanced_fast_string
   0.08%  workload-debug  [kernel.kallsyms]    [k] rmqueue
   0.05%  workload-debug  workload-debug       [.] ModifiedBTreeOLC::BTreeInner::binarySearchInFirstPhaseTraversal
   0.04%  workload-debug  [kernel.kallsyms]    [k] free_pcppages_bulk
   0.03%  workload-debug  [kernel.kallsyms]    [k] zap_pte_range.isra.0
   0.03%  workload-debug  [kernel.kallsyms]    [k] copy_page
   0.02%  workload-debug  workload-debug       [.] GetTxnCount<true>
```

Figure 6: *perf* results of the location of the program's cache misses.

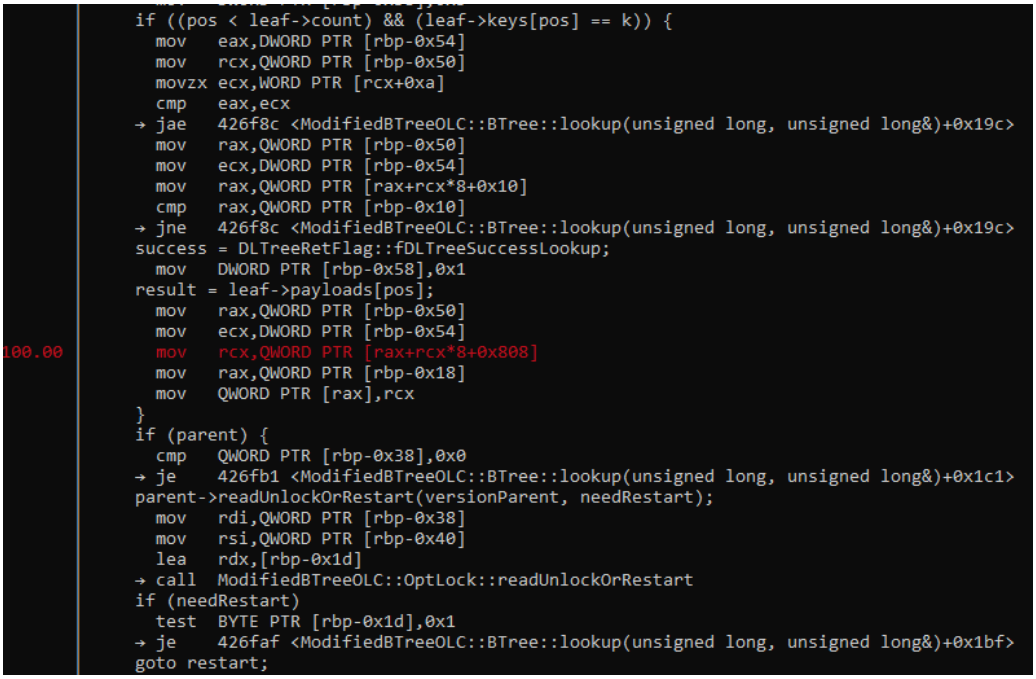

Figure 7: Cache misses inside the `lookup` function.

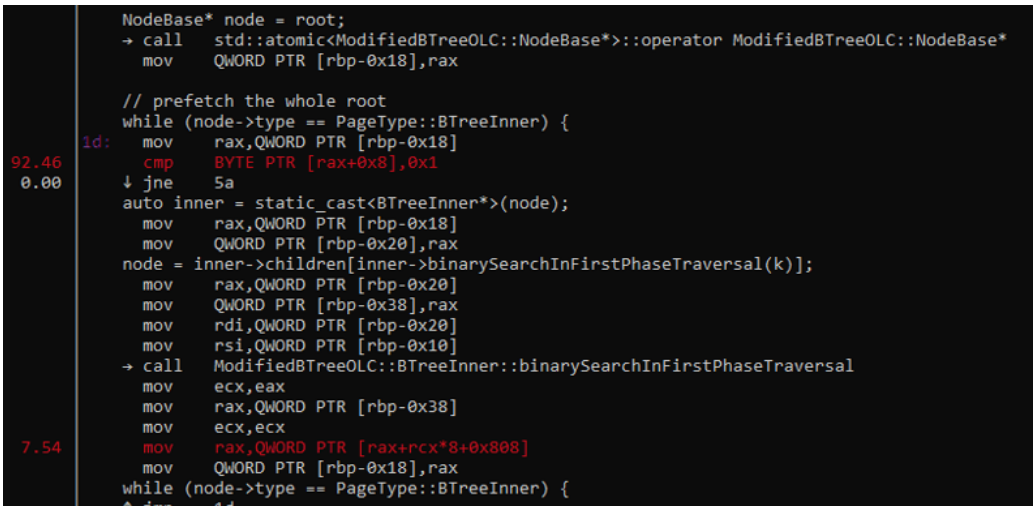

Figure 8: Cache misses inside the `traversal` function of first phase.

As shown in Figure 6 and 7, lookup function of phase 2 after running through the read-only workload causes cache misses entirely from accessing a leaf node's payloads, this is most likely because in the traversal function of phase 1 we only access the key array of nodes but never touch the leaf node's payloads.

As shown in Figure 8, in the traversal function of phase 1, apart from binary searching in the leaf node's key array, a majority of the cache misses come from accessing the type of the node during the traversal process. This is because we don't use locks and version numbers in the first phase's traversal function, hence accessing type of the node is the first time the node's data is accessed.

3.2 Benefits of Spiderman

1. Increase in operations throughput due to potential reduction in meaningless waiting time.
2. This technique requires no functionality changes to the actual functions performing the operations. Notice that the phase 2 part of the algorithm uses the existing code of the tree. Consequently, Spiderman is fairly general, easy to integrate, while avoiding extra correctness issues.
3. Operations are executed sequentially as usual in phase 2 of the algorithm, with the relevant content of the nodes in cache lines, which means less time is spent on holding locks for update operations since most likely the nodes' lock and version counter are in the cache. Therefore, this technique potentially reduces the contention time of waiting for lock release in a multithreaded environment.

3.3 Tuning the software prefetcher

There are several optimization options that we can enable, which can be imagined as toggles; each can be turned on/off individually.

1. Prefetch the header of the node that contains important information such as the lock, the version counter, the type (inner node or leaf node), the number of children, etc.
2. Only prefetch nodes after the first 2 layers of the tree, as the first two layers are accessed frequently enough and fit in caches.
3. Prefetch the middle key and then use the value of the middle key to determine which half of the key array to prefetch (called "middle key's optimization" in Algorithm 4).
4. Prefetch the whole node.
5. Prefetch the child address.
6. Prefetch payload of the leaf node if the request is a lookup.

We can even combine some of the optimization options together:

- Option 1, 3, 5, 6 can be combined with any other options except option 4.
- Option 2 can be combined with any other options.
- Option 4 can only be combined with option 2.

Algorithm 4 shows a finer prefetching strategy, which potentially improves the Spiderman technique since only the most likely used data of a node is kept in the cache.

```
Data: reqs (batch of requests), numReqs (number of requests)
for i ← 0 to numReqs do
    traversalDone[i] ← false;
    curNode[i] ← root;
    middleKeyOpPhase[i] ← 3;
    numTraverseLayers[i] ← 0;
end
numTraversalDone ← 0;
curReq ← 0;
while numTraversalDone < numReqs do
    if traversalDone[curReq] is true then
        curReq ← (curReq + 1) mod numReqs;
        continue;
    if curNode[curReq].type is innerNodeType then
        // Middle key's optimization phase 3: Do the normal memory access
        if middleKeyOpPhase[curReq] is 3 then
            curNode[curReq] ←
              curNode[curReq].children[curNode[curReq].keyBinarySearch(reqs[curReq].key)];
            // Only prefetch nodes after the first 2 layers of the tree
            if numTraverseLayers[curReq] >= 2 then
                // Only prefetch the header information of the node
                Prefetch(curNode[curReq], 1);
            numTraverseLayers[curReq]+ = 1;
            middleKeyOpPhase[curReq] ← 1;
        end
        // Middle key's optimization phase 1: Prefetch the middle key
        if middleKeyOpPhase[curReq] is 1 then
            mid ← curNode[curReq].count/2;
            // Fetch the cache line that contains the middle key
            if numTraverseLayers[curReq] >= 2 then
                Prefetch(..., ...);
            middleKeyOpPhase[curReq] ← 2;
        end
        // Middle key's optimization phase 2: Prefetch the correct half of key array based on the middle key
        if middleKeyOpPhase[curReq] is 2 then
            mid ← curNode[curReq].count/2;
            if reqs[curReq].key < curNode[curReq].key[mid] then
                // Prefetch first half since our key lies before mid
                if numTraverseLayers[curReq] >= 2 then
                    Prefetch(..., ...)
            else
                // Prefetch second half since our key lies after mid, similar as above
                ...
            end
            middleKeyOpPhase[curReq] ← 3;
        end
    else
        destinationNodeIndex ← curNode[curReq].keyBinarySearch(reqs[curReq].key);
        traversalDone[curReq] ← true;
        numTraversalDone+ = 1;
    end
    curReq ← (curReq + 1) mod numReqs;
end
```

Algorithm 4: The Spiderman technique, with a finer prefetching strategy for phase 1 of the technique, using prefetching option 1, 2, and 3.

4 Evaluation

We evaluate the effectiveness of the Spiderman technique on the state-of-the-art B+tree OLC codebase [1] in a similar fashion as how the authors of OpenBw-Tree [9] evaluated their work.

Our experiments are conducted on a server PowerEdge R740x, with two Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz processors and 125GB DRAM. Each processor has:

- 18 cores
- 36 threads
- 1152KiB L1 cache
- 18MiB L2 cache
- 24MiB L3 cache

We compiled our framework using clang++ (v12) [2], with the following flags [17]:

- -std=c++20
- -O3
- -march=native
- -finline-functions
- -fno-threadsafe-statics

We used the `mimalloc` library [4] for memory allocation along with the following environment options to ensure that outcome of the experiments is less affected by page faults:

- MIMALLOC_EAGER_COMMIT_DELAY=0
- MIMALLOC_PAGE_RESET=0
- MIMALLOC_RESERVE_HUGE_OS_PAGES=8
- MIMALLOC_RESERVE_HUGE_OS_PAGES_AT=0
- MIMALLOC_VERBOSE=0

Threads are pinned, and their memory allocations are restricted to the same NUMA node (node 0) on a single CPU socket in our multithreading experiments.

4.1 Workloads

Like in OpenBw-Tree [9], we used a set of Yahoo! Cloud Serving Benchmark (YCSB) microbenchmarks to mimic OLTP index workloads [7, 10]. We used default workloads A (Read/Insert, 50/50), B (Insert Only), C (Read Only), and D (Scan/Insert, 95/5) with random distributions. The average length of workload D's scan is 48, with standard deviation 30.12. We used 64-bit integers for both keys and values in the workloads. We populate the indexes using 50M keys and then execute 500M transactions (Read/Insert/Scan). The page size of each node of the B+tree OLC is 4096 bytes. For inner nodes, we have a maximum of 255 pairs of key and child. For leaf nodes, we have a maximum of 255 pairs of key and payload (also known as values).

4.2 Results and remarks of Spiderman

4.2.1 Spiderman's results

We compare the transaction throughput of the B+tree OLC with different batching sizes on the 4 workloads in the previous section. In detail, first we evaluate the transaction throughput of the tree without the technique. Then, we evaluate the transaction throughput of the technique applied to the tree with batching sizes of 1, 8, and 16 to see how the effectiveness changes with different batching sizes. For the list of transactions, we execute the transactions in groups of batch size. For example, if the batch size is 4, then we execute the first batch of 4 transactions, then the second batch of 4 transactions, and so on. We also vary the number of threads of the program, that is, we experiment with 1, 4, and 16 threads.

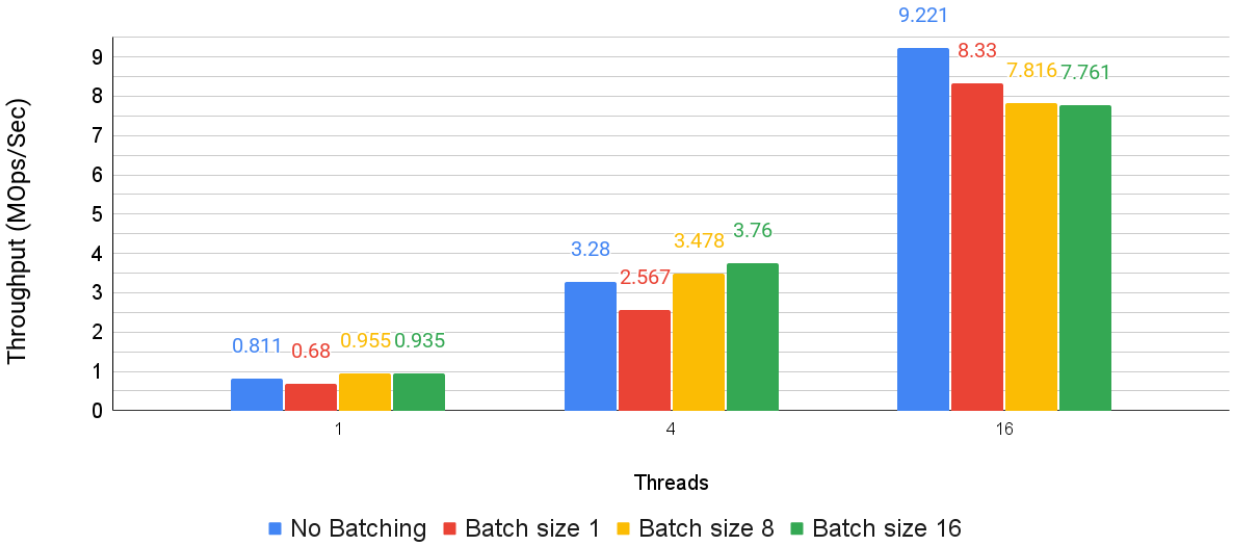

Figure 9: Spiderman technique with prefetching option 1, 2, and 3 on workload A (Read/Insert, 50/50).

As shown in the Figure 9,

- For the 1-thread program, we observe a 18% increase in throughput by using the Spiderman technique with batch size 8 and 16.
- For the 4-threads program, we notice a 15% increase in throughput by using the Spiderman technique with batch size 16.
- For the 16-threads program, we remark that there is a decrease in throughput by using the Spiderman technique with any batch sizes.

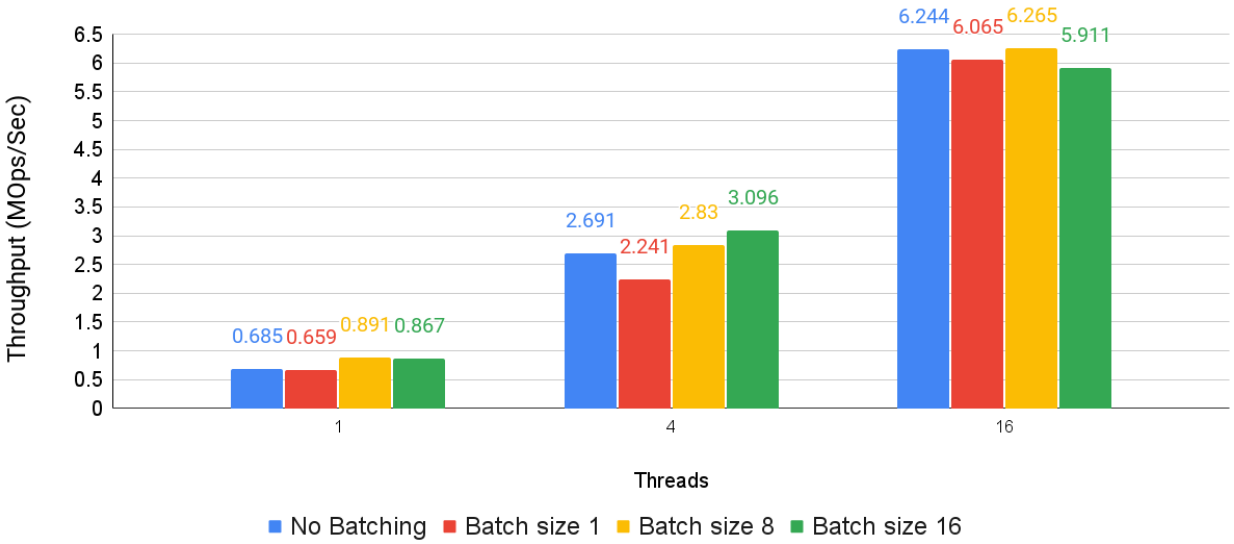

Figure 10: Spiderman technique with prefetching option 1, 2, and 3 on workload B (Insert Only).

As shown in the Figure 10,

- For the 1-thread program, we observe a 30% increase in throughput by using the Spiderman technique with batch size 8 and 16.
- For the 4-threads program, we notice a 15% increase in throughput by using the Spiderman technique with batch size 16.
- For the 16-threads program, we remark that there is a similar throughput by using the Spiderman technique with batch size 8.

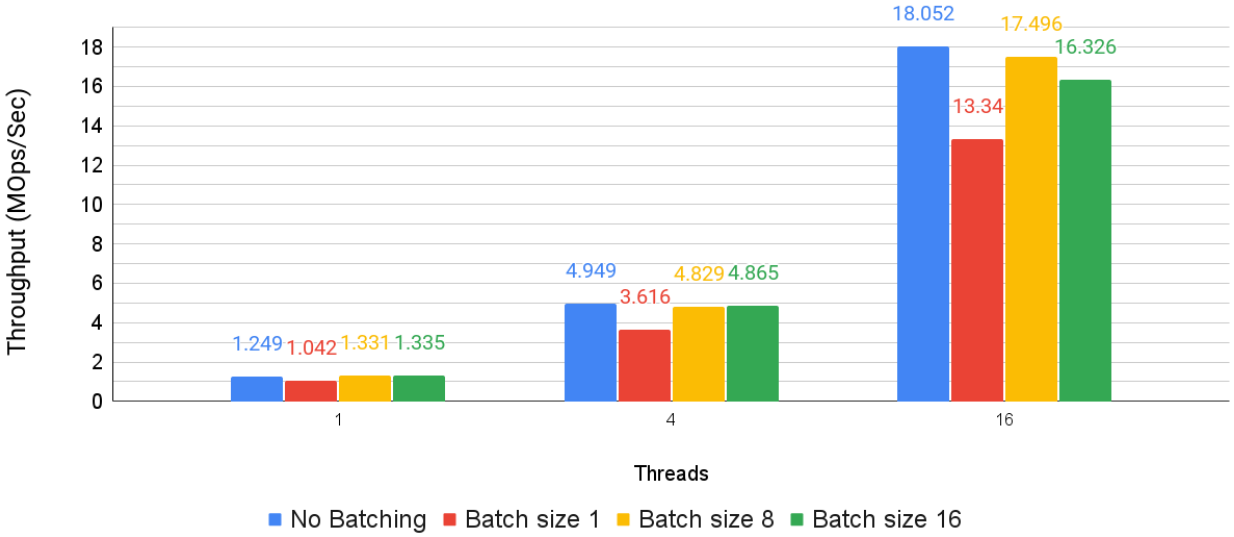

Figure 11: Spiderman technique with prefetching option 1, 2, and 3 on workload C (Read Only)

As shown in the Figure 11,

- For the 1-thread program, we observe a 6% increase in throughput by using the Spiderman technique with batch size 8 and 16.
- For the 4-threads program, we notice a similar throughput by using the Spiderman technique with batch size 8.
- For the 16-threads program, we remark that there is a slight decrease in throughput by using the Spiderman technique with any batch sizes.

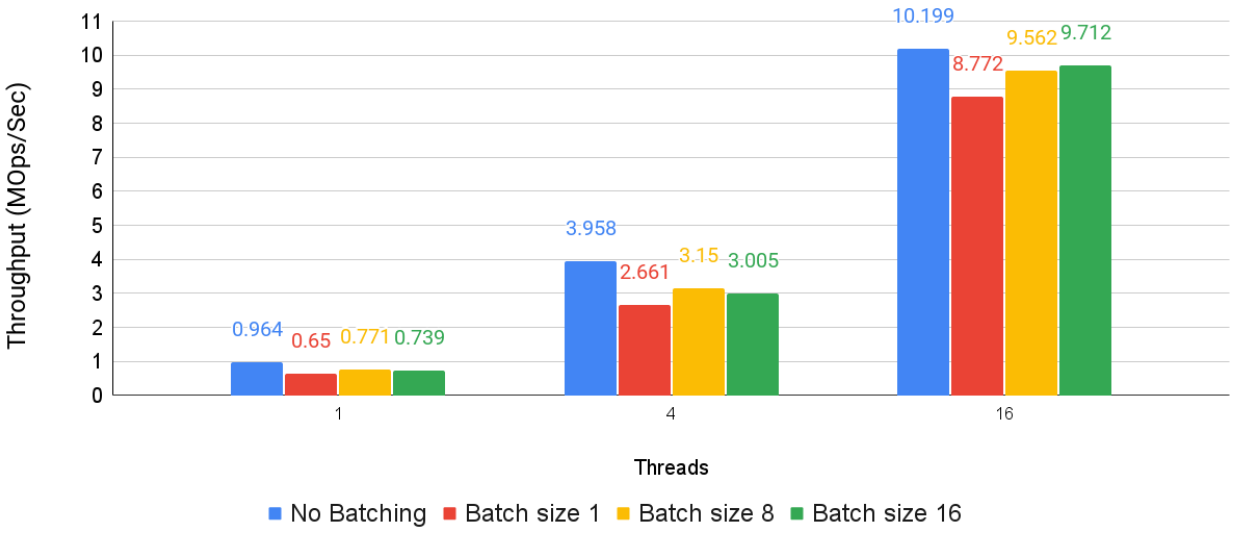

Figure 12: Spiderman technique with prefetching option 1, 2, and 3 on workload D (Scan/Insert, 95/5)

As shown in the Figure 12, we observe a decrease in throughput in all the programs by using the Spiderman technique with any batch sizes.

4.2.2 Remarks of Spiderman

Overall, the Spiderman technique shows up to 30% increase in the throughput of the B+tree OLC. While we observe performance gain in workload A and B, we notice some performance loss in workload C and D.

In all of the cases, the Spiderman technique with batching size 1 always perform worse than the original program, this is because batching of size 1 is ineffective, we don't have any other requests to switch to in order to take advantage of the memory access waiting time, while paying the cost of calling the prefetching function.

Sometimes, increasing the batch size does not prove to be more effective, this could be due to the overhead of maintaining metadata of the requests for phase 1 and phase 2 of the algorithm, and overhead from our naive implementation of the phase 1.

4.3 Spiderman's prefetching optimization decomposition

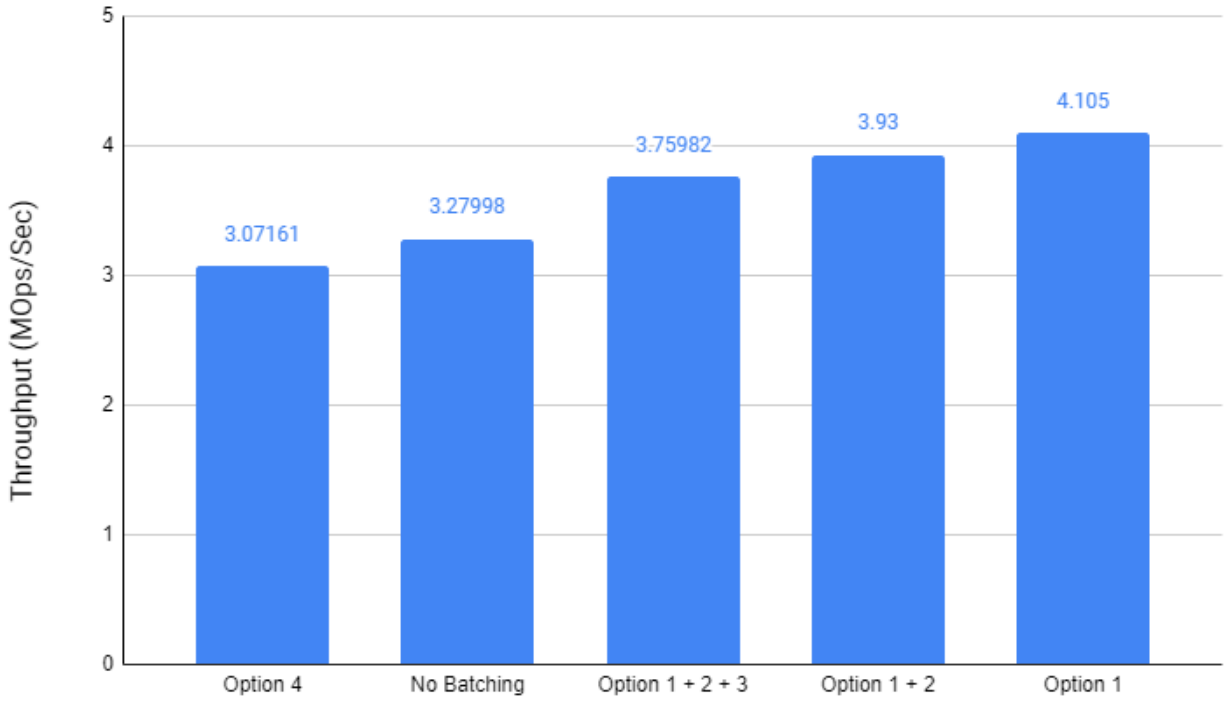

Figure 13: Spiderman's prefetching optimization decomposition

In the section above, for benchmarking, we used the Spiderman technique with prefetching options 1, 2, and 3. It would be interesting to remove each prefetching option one by one and see how it affects the performance, and also to compare the "finer-grain" strategies with the "coarse-grain" prefetching strategy of option 4, that is prefetching the whole node. Here we try different optimization strategies on Workload A (50% Read, 50% Insert) with 4 threads and batch size of 16 (if batching is enabled) to see how each prefetching option affects the performance. As shown in figure 13:

- Option 4, "coarse-grain" prefetching strategy, where we just prefetch the whole node, is worse than every other strategies, including no batching. We believe that this is due to the inefficient caching of the node, because in reality, during the execution of the operations, only some of the node's data is accessed. If we store all of the nodes in the caches, it will result in a lot of unused cache lines, thus creating many memory accesses.
- We observed that option 1 + 2 + 3 performs worse than option 1, we speculate that this might be because of the added overhead complexity of option 3. In option 1, for each level of the tree, we prefetch only once the memory

content of a node, which in this case it is the header of the node. In option 3, we prefetch the memory content of a node 3 times, in particular, the header of the node, then wait a round of batch size until the middle key is prefetched, and finally we wait another around until we prefetch a half of the key array.

5 Limitations and future work

5.1 Limitations

Here are the limitations of the technique:

1. Determining how to choose the batching size so that we get the maximum transaction throughput for a given workload and environment.
2. The prefetch function has a small overhead, which can potentially slow down the code if called many times unnecessarily.

5.2 Future Work

Currently, for a given operation, the first part of the technique will traverse through the nodes that the algorithm predicts this operation will go through. Then in the second part, where we execute the requests in the batch sequentially, we will do another traverse through the tree for the operation. The common nodes on these 2 paths will be traversed twice. Future work could explore eliminating this double traversal. We can do that by storing the destination node from the first traversal and seeing somehow if the path is not modified then we can jump directly to the destination node and eliminate the overhead of going through the path another time.

Moreover, the implementation of the phase 1 of the Spiderman technique that handles prefetching and request multiplexing is still naive, and can be more efficient. For example, even if there is only 1 request left out of 16 requests that has not finished its traversal in phase 1, we still have to go through the other 15 requests that are already done, check whether or not it is done, and skip it, something we have in mind is only maintain and switch between the traversals that are still in progress.

There are still a lot of explorations to be done on what are the combinations of prefetching that yield the best performance and other opportunities where we could leverage software prefetching and memory. For examples, we haven't tried out prefetching strategies that involve option 5 and 6, and also have the time to evaluate the promising Spiderman technique with option 1.

We believe our technique can be general enough to apply to other types of tree data structures. As such, another important future work is to adapt and integrate it on other concurrent in-memory trees (e.g., Masstree [21], ART OLC [18, 20], HOT [16], etc.).

6 Summary of contributions and conclusion

In this thesis, we presented Spiderman, a technique that can be applied to concurrent in-memory B+trees to boost their throughput. We detailed how this technique has different prefetching options that can be enabled and a configurable batching size. To verify our claims, we implemented Spiderman over a state-of-the-art B+tree design and run experiments both in single-threaded and multi-threaded versions to see the effect of this technique, demonstrating up to 30% throughput gains, which showcases its potential in pushing the limits of the state-of-the-art concurrent in-memory trees.

7 References

[1] B+tree OLC - https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC.

[2] Clang 12.0.0 - https://releases.llvm.org/12.0.0/tools/clang/docs/ReleaseNotes.html.

[3] Hendrix college's CSci 340: Database web systems - http://www.cburch.com/cs/340/reading/btree/.

[4] Microsoft's free and open-source compact general-purpose memory allocator - https://github.com/microsoft/mimalloc.

[5] GNU extension to prefetch memory __builtin_prefetch - https://www.daemon-systems.org/man/__builtin_prefetch.3.html.

[6] https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html.

[7] Yahoo! cloud serving benchmark - https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/.

[8] Linux performance analyzing tool perf - https://perf.wiki.kernel.org/index.php/Main_Page. 2009.

[9] Ziqi Wang Andrew Pavlo Hyeontaek Lim Viktor Leis Huanchen Zhang Michael Kaminsky David G. Andersen. Building a bw-tree takes more than just buzz words. *SIGMOD*, 2018.

[10] Erwin Tam Raghu Ramakrishnan Brian F. Cooper, Adam Silberstein and Russell Sears. Benchmarking cloud serving systems with ycsb. *SoCC*, 2010.

[11] Jeff Dean Colin Scott and Peter Norvig. Latency numbers every programmer should know by year on github - https://colin-scott.github.io/personal_website/research/interactive_latency.html.

[12] Douglas Comer. Ubiquitous b-tree. *Association for Computing Machinery (ACM) Computing Surveys (CSUR)*, 11, 2:121–137, 1979.

[13] Dmitry Dolgov. https://erthalion.info/2020/11/28/evolution-of-btree-index-am/. 2020.

[14] Andrew Pavlo Michael Kaminsky Lin Ma Huanchen Zhang, David G. Andersen and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. *ACM Special Interest Group on Management of Data (SIGMOD)*, page 1567–1581, 2016.

[15] Justin J Levandoski and Sudipta Sengupta. The bw-tree: A latch-free b-tree for log-structured flash storage. *EEE Data Eng. Bull*, pages 56–62, 2013.

[16] Martin Pichl Günther Specht Viktor Leis Robert Binna, Eva Zangerle. HOT: A height optimized trie index for main-memory database systems. *SIGMOD*, 2018.

[17] Caio Rordrigues. Compiler flags options - https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html.

[18] Alfons Kemper Viktor Leis and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. *ICDE*, page 38–49, 2013.

[19] Alfons Kemper Viktor Leis, Florian Scheibner and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method.

[20] Alfons Kemper Viktor Leis, Florian Scheibner and Thomas Neumann. The art of practical synchronization. *DaMoN*, 2016.

[21] Eddie Kohler Yandong Mao and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. *EuroSys*, page 183–196, 2012.

A Appendix

INSTITUT POLYTECHNIQUE DE PARIS

Statement of Academic Integrity Regarding Plagiarism

I, the undersigned.....Thang Long VU.....[family name, given name(s)], hereby certify on my honor that:

1. The results presented in this report are the product of my own work.

2. I am the original creator of this report.

3. I have not used sources or results from third parties without clearly stating thus and referencing them according to the recommended rules for providing bibliographic information.

Declaration to be copied below:

I hereby declare that this work contains no plagiarized material.

Date 19/03/2023

Signature Long

Thang Long VU