Performance of Coroutines in Hash Tables

Justin Yiu Ming Mok
Simon Fraser University
Burnaby, BC, Canada
jymok@sfu.ca

Don Van
Simon Fraser University
Burnaby, BC, Canada
dva13@sfu.ca

ABSTRACT

Working with database systems involves an abundant amount of pointer-based data structures like hash tables and binary search. Previous works have shown that CPU stalls can be caused by cache misses which force the CPU to wait for main memory accesses which is a significant source of overhead and have proposed techniques to reduce their impact. In this paper, we will compare and contrast different prefetching approaches based on runtimes. We will be presenting our analysis and experimental evaluations of the effects of the different approaches on hash table probing. This approach will allow us to have a deeper understanding of how coroutines compares to the naive, Asynchronous Memory Access Chaining (AMAC), and Group Prefetching (GP) techniques to create the most efficient database structure.

1 INTRODUCTION

Current database systems (DBMS) predominantly rely on main memory for data storage, management, and manipulation. The main memory is stored in low-latency volatile DRAM which allows for the removal of the disk I/O bottleneck which has been the major problem that was the cause for significant latency in databases in the past. Even though the disk I/O bottlenecks have been removed, main memory still suffers from higher latency than CPU caches which means there is now a memory-access bottleneck instead.

Data structures that are usually employed by in-memory database systems are pointer-based structures like hash tables and binary search. During an database operation, these pointers dereference several other pointers which quickly becomes an issue known as pointer chasing. These dereferences are able to stall the CPU due to the likelihood of cache misses if there is too much data is to be held in cache. Our goal is to study approaches that utilizes prefetching to address this bottleneck. [1] Our paper can be summarized with the following. We start by presenting an in-depth analysis of some prefetch-based approaches to hide memory access latency. We implemented a simple data structure, the hash table, while using coroutines and then compare its performance with other techniques, such as Group Prefetching (GP) and Asynchronous Memory Access Chaining (AMAC). Overall, this paper compares the performance of coroutines and two other prefetching techniques in the scope of read only hash table operations. This investigation allows us to better understand how coroutines can be utilized to build more efficient systems.

In this paper, we will provide backgrounds on software based prefetching techniques. Afterwards, we will provide a brief explanation on hash tables and the analysis of our results. Finally, we will conclude about our paper, primarily highlighting our goal and what the results showed us.

2 BACKGROUND AND RELATED WORK

Most in-memory database systems commonly utilize pointer-based data structures instead of a page-based indirections that are found in other database systems. These types of indexes are commonly hash-based which better for point lookups. Past experiments have shown that CPU stalls are primarily caused by main-memory access bottlenecks for pointer-based data structures. In order to address this issue, techniques have been developed in the past few years to address these bottlenecks.

The main idea of these techniques is hide CPU due to cache misses and memory accesses with prefetching. Each operation would give a software prefetch that will be accessed, but prior to fetching the data, it performs a context-switch to the next operation instead of waiting for the data to arrive in the CPU cache. As a result, the prefetched memory will be present in the cache by the time a thread returns to work on the first operation, which hides the cost of the CPU stall.

The rest of this section will present the these techniques that have been created to avoid CPU stalls. We will be implementing Hash Index Probing with these various techniques to analyze this problem.

2.1 Group Prefetching (GP)

Group Prefetching (GP) is a loop-transformation approach that exploits inter-tuple parallelism to overlap cache miss latencies of one tuple with computations and miss latencies of other tuples. The main advantage of GP is that the code is written in a "semi synchronous manner". This type of coding allows for operations to be executed in parallel rather than one at a time. Unfortunately, the two main disadvantages of GP is that: (1) the number of pre-defined code stages is needed in advance, and (2) if one operation terminates early, GP cannot execute another operation in its place because the new operation would be executed on a different code stage than others. The second disadvantage reveals "no-op" operations into the GP pipeline that affects overall performance when the workload is skewed. [1]

Algorithm 1 is pseudo code that executes a GP-based Hash Index Probe. The algorithm begins with the input to the algorithm which is a set of keys, size of index and the hash index itself. When executing GP-based Hash Index Probe, there are two stages: (1) From Lines 5 to 8, the algorithm runs the hash function to get the base node. (2) From Lines 10 - 23, the algorithm compares the N keys with the prefetched nodes.

2.2 Asynchronous Memory Access Chaining (AMAC)

Asynchronous Memory Access Chaining (AMAC) is a new approach for exploiting inter-lookup parallelism to hide the memory access

Algorithm 1 GP-based Hash Index Probe

```
struct GP_state node;
procedure HASH-PROBE-GP(input[],n,table)
    init value [n]
    init stateArr [n]
    for i = 0; i < n; i++ do
        stateArr[i].node = hash.get(input[i])
        prefetch stateArr[i].node
    end for
    init num_finished = 0
    while num_finished < input.length do
        for i = 0; i < n; i++ do
            if stateArr[i].node == null then
                value[i] = -1
                num_finished++
                continue
            else if input[i]==stateArr[i].node->data->key then
                value[i] = state.node->data->value
                num_finished++
            else
                stateArr[i].node = stateArr[i].node->next
                prefetch stateaArr[i].node
            end if
        end for
    end while
    return value[]
end procedure
```

latency. Specifically, AMAC can achieve its dynamism by maintaining the state of each in-flight lookup separately compared to other lookups. However, the main disadvantage of AMAC is that it transforms a synchronous operation into a state machine, which requires a complete rewrite of the code. This results with the code that has no similarities to the original code or "synchronous" version which sacrifices code readability and maintainability. Simply, AMAC fully utilizes available architectural resources and generates the maximum number of memory access allowed by both single and multi-thread execution nodes within its hardware. [2]

Algorithm 2 reveals the pseudo code for our AMAC approach for probing a hash index. In this algorithm, the group_ parameter represents the amount of concurrent operations. This prefetching technique stores the information of each operation in a circular buffer. When this algorithm is executed, it has two main stages: (1) From Lines 10 to 14, it will start a new probe for the next key that is going to be probed. (2) From Lines 15 to 25, this part of the algorithm compares the key with the prefetched node ones. [1]

2.3 Coroutines

A coroutine is a function that can suspend execution to be resumed at a later time. These functions are typically stackless meaning they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack. This type of execution allows for sequential code to be asynchronously executed and supports algorithms on lazy-computed infinite sequences and other uses. [1, 3]

Algorithm 2 AMAC-based Hash Index Probe

```
 1: struct AMAC_state key; node; value; stage;
 2: procedure HASH-PROBE-AMAC(input[],n,table, group_size)
 3:     init value [n]
 4:     AMAC_circular_buffer buff(group_size)
 5:     init num_finished = 0
 6:     init i = num_finished
 7:     init j = i
 8:     while num_finished < n do
 9:         state = buff.next_state()
10:         if state -> stage == 0 then
11:             state.key = input[i++]
12:             state.node = hash.get(state.key)
13:             state.stage = 1
14:             prefetch state.node
15:         else if state.stage == 1 then
16:             if state.node == null then
17:                 state.value = -1
18:                 state.stage = 2
19:                 value[j++] = state.value
20:                 num_finished++
21:             else if state.key == state.node->data.key then
22:                 state.value = state.node->data.value
23:                 state.stage = 2
24:                 value[j++] = state.value
25:                 num_finished++
26:             else
27:                 state.node = state.node->next
28:                 prefetch state.node
29:             end if
30:         end if
31:     end while
32: end procedure
```

Algorithm 3 Coroutines-based Hash Index Probe

```
function HASH-PROBE-CORO (table, key)
node = table.get(key)
prefetch node
co_await suspend_always
while node do
    if key == node -> key then co_return node->value
    else
        node = node->next
        prefetch node
        co_await suspend_always
    end if
end while
co_return null
end function
```

During execution, each coroutine is associated with: (1) A promise object manipulates the coroutine from the inside where it submits its result or an exception through this object. (2) The coroutine handle manipulates the coroutine from the outside. This coroutine

Table 1: Experimental Setup

DRAM	384GB
OS	Linux
Compilers	Clang

is a non-owning handle that is used to resume execution or to destroy the coroutine frame. (3) The coroutine state is an internal, heap-allocated, object that contains the promise object, parameters, some representation of the current suspension point, a local variable, and temporaries. At the beginning of a coroutine execution, it will allocate the coroutine state object using "operator new" and it would then copy all of the function parameters to the coroutine state.

Generally, using the coroutine approach is similar to AMAC since we are interleaving multiple coroutines at the same time. The difference between the two is that instead of transforming the operation ourselves, the compiler does most of the heavy lifting with coroutines allowing us to just add suspension points to the otherwise synchronous execution of the code after every prefetch.

Algorithm 3 reveals a pseudo code for a coroutine based hash index probe. This concept is almost identical to AMAC where a circular buffer is set up to maintain each operation's state. Therefore, the state would be the coroutine which does not require a transformation to become a state machine. Coroutines usually relies on the compiler to preserve each state during its suspension. At last, when the coroutine has finished its execution, it would allow the caller replace it with a new coroutine for the next key probe.

2.3.1 Coroutines in C++. Although coroutines existed for a period of time, this approach has only been a standard of C++20 for only a few years, however this standard is still considered an experimental feature. So, in this paper, coroutines are available on Clang 5.X/6.X and g++ compilers. When coroutines are executed in C++, the compiler would contain any keywords like co_yield, co_return or co_await into a coroutine. [1, 3] Co_return is roughly functionally equivalent to return. Co_await will optionally suspend he coroutine, as directed by the object being awaited. These two keywords is what every developer is required to start with coroutines. The compiler would create the code to allocate and manage a coroutine frame for each call in which the library tracks. Since coroutines are stackless, allocating frames for each call is slow and the cost could be insignificant.

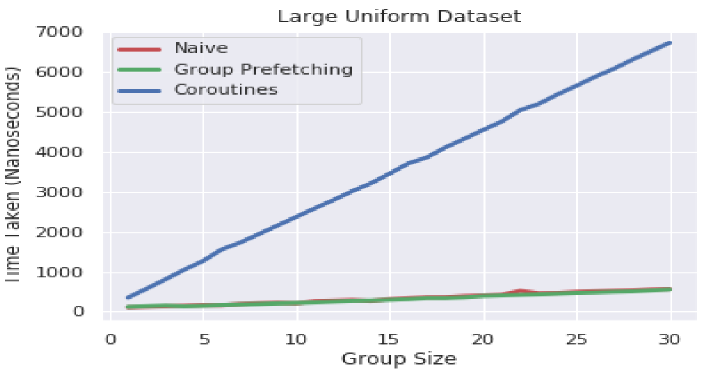
3 SIMPLE DATA STRUCTURES

This section would describe our experiments and analysis when comparing GP and Coro approaches to hide memory stall latency for hash table probing. Primarily, our focus is to have a quantitative comparison using throughput, by comparing the run-times under a n number of runs for each technique.

These experiments will be executed on a Linux platform with specific hardware shown in Table 1. Since coroutines are still considered to be a experimental feature on the C++20 standard, we will be using the Clang 6.0 compiler to compare the different runtimes.

Algorithm 4 Naive-based Hash Index Probe

```
procedure HASH_PROBE(input[],n,table)
    init value[n]
    for i = 0; i < n; i++ do
        init index = hash.get(input[i])
        curr = table.nodes[index]
        while curr != null do
            if curr.key == input[i] then
                value[i] = curr.value
                break curr = curr.next
            end if
        end while
        if curr == null then value[i] = -1
        end if
    end for
    return value
end procedure
```



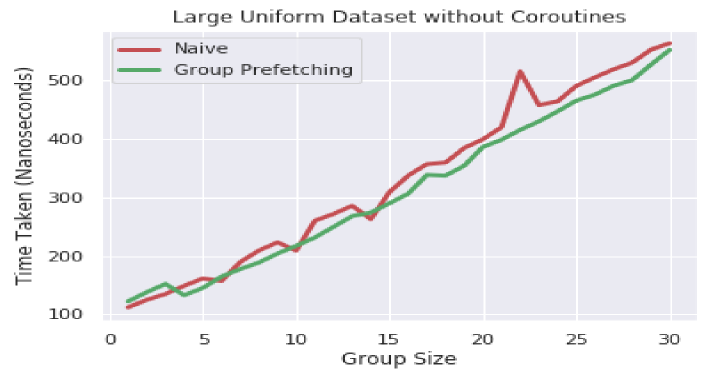
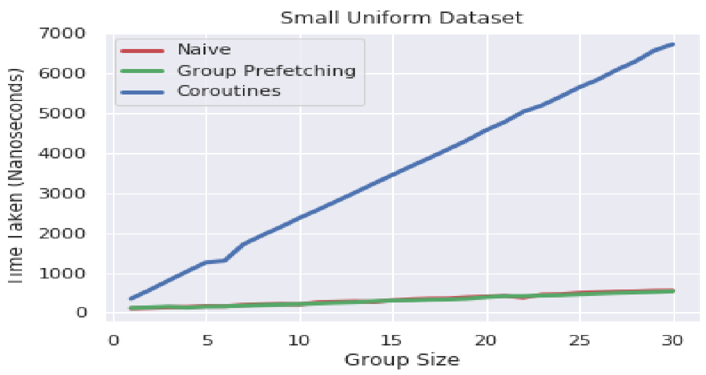


Figure 1: Clang Hash Index Probe 1 Thread Large Input Size

3.1 Hash Tables

Hash Tables are a data structure that stores data in an associative manner. This type of data structure is able to access data extremely fast only if we knew the index of the desired data. To begin, we use our experimental evaluations to examine the runtimes of probing a hash table under a variety of prefetch-based approaches to hide memory latency. In order, to prove these comparisons, we would use a naive implementation that does not utilize any prefetching



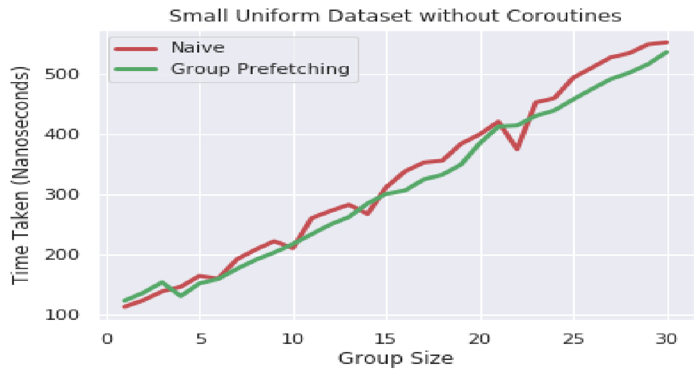


Figure 2: Clang Hash Index Probe 1 Thread Small Input Size

techniques. In this paper, we are not only trying to replicate results but using other techniques like coroutines with other compilers as well. [4]

Algorithm 4 represents the pseudo code for our naive implementation of Hash Index Probe. To perform this, we created an already used index in the array. Then by searching, it looks for the next location and looks into next cell until an empty one is found.

3.1.1 Single Threaded Varying Group Sizes. Our objective with this experiment is to analyze how group size affects the performance of prefetch based approaches to hash probing. Additional factors that we wanted to tweak to analyze the difference in performance is utilization of both a small and large key size along with the variable group size. We used 0.13M keys for our small input size and 10M keys for our large input size. When inputting these sizes into our naive, GP and Coro based implementations, we would use a maximum group size of 30 on a single thread executing 100000 and 10000 runs on a small and large input size respectively in order to calculate the overall runtimes for each respective prefetching technique. Figures 1 and 2 show the results for the Clang compiler with input sizes large and small respectively, with group size on the x-axis (0-30) and average nanoseconds on the y-axis.

For Figure 1, the two graphs show the differences in runtimes in nanoseconds in respect with its group size for a large uniform dataset when using coroutines and one that does not use coroutines. Figure 1(a) shows the distribution of runtimes when coroutines are used, it shows that GP and Naive has similar average runtimes and its optimal group size is 22. However, Coro has greater runtimes than GP and Naive due to it high overhead but its optimal group size is around 5. In contrast, Figure 1(b) shows the distribution of runtimes when coroutines are not used, it clearly shows that no matter your group size, the average runtimes are inconsistent, therefore without coroutines, it is less efficient to large uniform dataset on a hash index.

For Figure 2, the two graphs show the difference in runtimes in nanoseconds in respect with its group size for a small uniform dataset when using coroutines and one that does not use coroutines. Similar to Figure 1(a) and 1(b), it clearly shows that without coroutines, the average runtimes are inconsistent no matter what group size is used. However, when coroutines are used, the runtimes are consistent with optimal group size 25 for GP and Naive, while the optimal group size 5 for Coro.

4 CONCLUSION

In this paper, our initial goal was to implement and evaluate coroutine-based data structures to hide memory stall latency. We evaluated this by using a simple data structure call hash index probe. To show the effectiveness of coroutines we, compared it with other prefetching techniques like naive and group prefetching. We showed what happens to the average runtimes with respect to its group size on large and small uniform dataset when coroutines are used and not used. The graphs clearly shows that with coroutines the runtimes are consistent and without them the runtimes are inconsistent. Unfortunately, we had an unsuccessful porting of our g++ implementation of AMAC to Clang porting and thus we were unable to perform benchmarks on the target system.

REFERENCES

[1] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the 'Killer Nanoseconds'. *Proceedings of the VLDB Endowment* 11, 11 (Aug. 2018), 1702–1714. https://doi.org/10.14778/3236187.3236216
[2] Onur Kocberver, Babak Falsafi, and Boris Grot. [n.d.]. Asynchronous Memory Access Chaining. *Proceedings of the VLDB Endowment* 9, 4 (April [n. d.]), 252–263. https://doi.org/10.14778/2856318.2856321
[3] tutorialspoint. 2022. Coroutines (C++20) - cppreference.com. https://en.cppreference.com/w/cpp/language/coroutines/
[4] tutorialspoint. 2022. Data Structure and Algorithms - Hash Table. https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm