SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Hiding Cache-Misses with Coroutines across different Hardware Architectures

Daniel Gruber

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Hiding Cache-Misses with Coroutines across different Hardware Architectures

Verbergen von Cache-Misses mittels Coroutinen auf unterschiedlichen Hardwarearchitekturen

Author:	Daniel Gruber
Examiner:	Alexander Beischl
Supervisor:	Prof. Dr. Thomas Neumann
Submission Date:	22.12.2025

I confirm that this master’s thesis is my own work and I have documented all sources and material used.

Munich, 22.12.2025 Daniel Gruber

Abstract

Contents

1 Introduction

1.1 Section

Citation test [4].

How can Coroutines hide latencies for hashtable/bptree lookups? What is the performance gain generally? What is the performance gain in duckdb?

How does the performance gain differ between various architectures? (Intel x86 vs ARM vs NUMA) What about Threading and Coroutines? Hyperthreading/NUMA

1.2 Related Work

boost coro, other async concepts, other works regarding coroutine implemenmtation a lot of work on coroutines microbenchmarking on one machine with one compiler but not acrosos machines and compilers

1.3 Structure of this thesis

2 Coroutines and Cache Misses Fundamentals

2.1 C++ Coroutines

2.1.1 Introduction to Coroutines

C++ Coroutines are available in the std namespace from the C++20 standard on, and from the C++23 standard there is a generator implementation based on the C++ 20 coroutine concept.

A coroutine is a generalization of a function, being able additionally to normal functions to suspended execution and resumed it later on. Any function is a coroutine if its definition contains at least one of these three keywords:

- co_await - to suspend execution
- co_yield - to suspend execution and returning a value
- co_return - to complete exeuction and returning a value

C++ Coroutines are stackless, meaning by suspension and consecutively returning to the caller, the data of the coroutine is stored separately from the stack, namely on the heap. This allows sequential code to be executed asynchronously, without blocking the thread of execution and supports algorithms to be lazily computed, e.g. generators. However, there are some restrictions to coroutines: They cannot use variadic arguments, plain return statements or placeholder return types likes auto or Concept. Also consteval, conexpr and the main function as well as constructors and destructors cannot be coroutines.

To illustrate the difference between coroutines and functions, this paragraph provides a background and general information of function calls and return: A normal function has a single entry point - the Call operation - and a single exit point - the Return operation. The Call operation creates an activation frame, suspends execution of the caller and transfers execution to the callee, where the caller is the invocating function and the callee is the invocated function. The Return operation returns the value in the return statement to the caller, destroys the activation frame and then resumes execution of the caller. These operations include calling conventions splitting the responsibilites of the caller and callee regarding saving register values to their activation frames. The activation frame is also commonly called stack frame, as the functions state (parameters, local variables) are stored on the stack. Normal Functions have strictly nested lifetimes, meaning they run synchronously from start to finish, allowing the stack to be a highly efficient memory allocation data-structure for allocation and freeing frames. The pointer pointing at the top of the stack is the **rsp** register on X86-64 CPU Architectures.

Coroutines have, additionally to the call and return operation, three extra operations, namely suspend, resume and destroy. As coroutines can suspend execution without destroying the activation frame, as it may be resumed later, the activation frames are not strictly nested anymore. This requires that after the creation of the coroutine on the stack, the state of the coroutine is saved to the heap, like illustrated in Figure 2.1 where a normal function f() calls a coroutine function c().

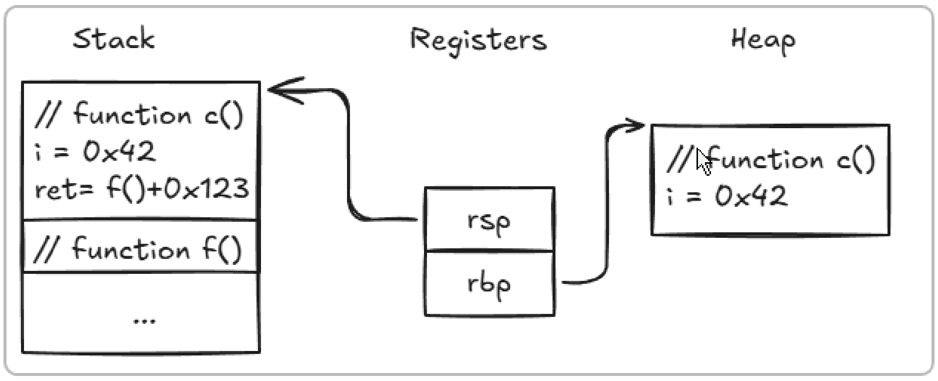

Figure 2.1: Calling a Coroutine - Heap Allocation

If the coroutine calls another normal function g(), g() 's activation frame is created on the stack and the coroutine stack frame points to the heap allocated frame, as illustrated in Figure 2.2. When g() returns, it destroys its activation frame and restores c()'s activation frame.

If c() now hits a supension point as shown in Figure 2.4, the Suspend operation is invoked where c() suspends execution and returns control to f() without destroying its activation frame. The Suspend operation interrupts execution of the coroutine at a current, well-defined point - co_await or co_yield -, within the function and potentially transfers execution back to the caller without destroying the activation frame.

This results in the stack-frame part of c() being popped off the stack while leaving the coroutine-frame on the heap. When the coroutine suspends for the first time, a return-value is returned to the caller. This return value often holds a handle to the

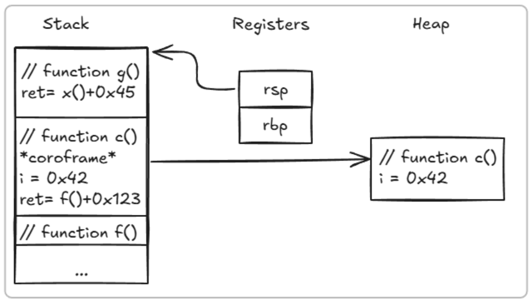

Figure 2.2: Coroutine calling a normal Function g()

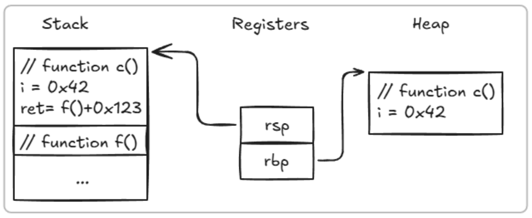

Figure 2.3: Normal Function g() returns

coroutine-frame that suspended that can be used to later resume it. When c() suspends it also stores the address of the resumption-point of c() in the coroutine frame (in the illustration called RP for resume-point).

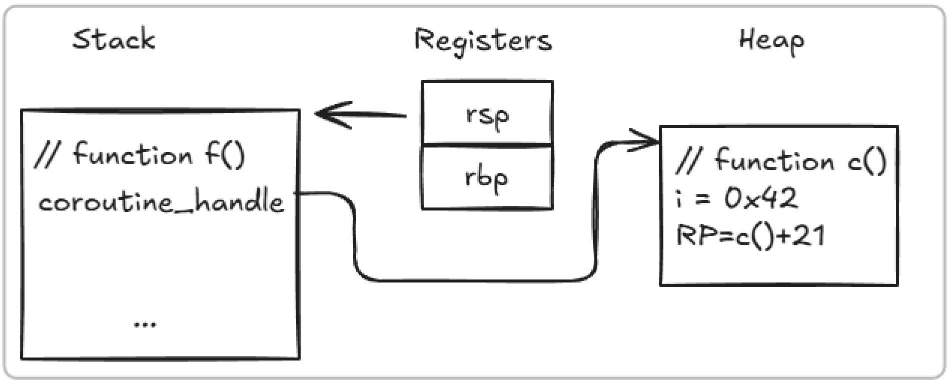

Figure 2.4: Coroutine hits Suspension Point

The Resume operation transfers execution back to the coroutine at the point it was suspended whereas the Destroy operation is the only operation that destroys the activation frame of the coroutine.

This handle may now be passed around as a normal value between functions. At some point later, potentially from a different call-stack or even on a different thread, something (say, h()) will decide to resume execution of that coroutine. For example, when an async I/O operation completes.

The function that resumes the coroutine calls a void resume(handle) function to resume execution of the coroutine. To the caller, this looks just like any other normal call to a void-returning function with a single argument.

This creates a new stack-frame that records the return-address of the caller to resume(), activates the coroutine-frame by loading its address into a register and resumes execution of x() at the resume-point stored in the coroutine-frame.

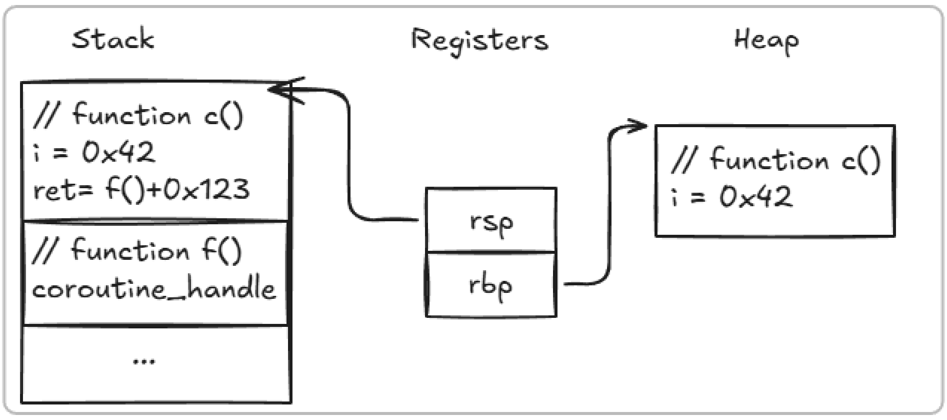

Figure 2.5: Resumption of a Coroutine

[3]

2.1.2 Execution of a Coroutine

Each coroutine is associated with a promise object, a coroutine handle and the coroutine state.

As partially illustrated in Figure 2.1, when a coroutine begins execution, it performs the following: - allocates the coroutine state object using operator new and copies all function parameters to the coroutine state. - calls the constructor for the promise object. - calls promise.get_return_object() and stores the result in a local variable, which will be returned to the caller on the first suspension point of the coroutine. - calls promise.initial_suspend() and co_awaits its result. Typically returns std::suspend_always (lazily-started) or std::suspend_never (eagerly-started). - when co_await promise.initial_suspend() resumes, starts executing the body of the coroutine.

```
{
  co_await promise.initial_suspend();
  try
  {
    <body-statements>
  }
  catch (...)
  {
    promise.unhandled_exception();
  }
FinalSuspend:
  co_await promise.final_suspend();
}
```

Listing 2.1: coroutine execution

When a coroutine reaches a suspension point, co_await or co_yield, the return object obtained earlier is returned to the caller/resumer. When encountering the co_return statement, it calls either promise.return_void() or promise.return_value(expr) depending on whether an expression is provided and whether this expression is non-void. Furthermore, all variable with automatic storage duration are destroyed in reverse order of their creation and finally calling promise.final_suspend() and co_awaiting its result. When falling off the end of the coroutine, meaning there is no co_return statement, it is equivalent to a co_return; statement.

2.1.3 Implementation details/concepts of a coroutine

The following is a minimum coroutine example using co_await, illustrated in Listing 2.2. The Task struct is a the coroutine wrapper holding the coroutine handle and the promise_type struct defining the promise type and thus, the behavior of the coroutine. The function Task Coroutine(int num_steps) is the coroutine function containing a co_await expression suspending the coroutine for num_steps times. And as explained in previous subsection, also see Listing 2.1, when another function, e.g. main(), calls the coroutine, the coroutine frame is allocated on the heap, the promise object is constructed and initial_suspend() is called. When the coroutine hits the co_await expression, it suspends execution and returns control to the caller until it is resumed again. The coroutine results in the output of "Coroutine at step i" and "Resuming coroutine" for num_steps times, in this case from 0 to 4, as initial_suspend() is set to suspend_never and thus, the coroutine starts executing immediately when called. If it it set to suspend_always, the coroutine initially pauses before starting work (lazily computed coroutine) and results in "Resuming coroutine" being printed first followed by "Coroutine at step i" num_steps times.

```
struct Task {
  struct promise_type {
      Task get_return_object() { return Task{this}; }
      std::suspend_always initial_suspend() { return {}; }
      std::suspend_never final_suspend() noexcept { return {}; }
      void unhandled_exception() { }
      void return_void() { }
    };

    std::coroutine_handle<promise_type> h{}; // coroutine handle

    explicit Task(promise_type* p) : h{std::coroutine_handle<promise_type>::from_promise(*
    Task(Task&& rhs) : h{std::exchange(rhs.h, nullptr)} { }
    ~Task() { if (h) { h.destroy(); } }

sk PrintCoroStep(int num_steps) {
  for (int i = 0; i < num_steps; ++i) {
    std::cout << "Coroutine␣at␣step␣" << i << "\n";
    co_await std::suspend_always{}; // if suspend_never, coro would never pause
  }
  co_return; // equivalent to omitting this line

t main() {
  Task task = PrintCoroStep(5);
  for (int i = 0; i < 5; ++i) {
    std::cout << "Resuming␣coroutine\n";
    task.h.resume();
  }
```

Listing 2.2: minimum coroutine co_await example

In this simple example, trivial awaitables std::suspend_always and std::suspend_never, defined in the standard library, are used to control the suspension behavior of the coroutine at the initial and co_await suspension points. The implementation of suspend_always is shown in Listing 2.3 returning false in await_ready(), indicating that the await expression always suspends and waits for a value (is not ready). The suspend_never implementation is analog with the only difference of returning true in await_ready(), indicating that the await expression never suspends (is always ready).

```
struct suspend_always {
  constexpr bool await_ready() const noexcept{return false;}
  constexpr void await_suspend(std::coroutine_handle <>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

Listing 2.3: suspend_always implementation

For more customization, more complex awaitabletypes can be implemented, requiring to implement these three methods listed in Listing 2.4.

```
bool await_ready();

// one of:
void await_suspend(std::coroutine_handle<> h);
bool await_suspend(std::coroutine_handle<> h);
std::coroutine_handle<> await_suspend(std::coroutine_handle<> h);

T await_resume();
```

Listing 2.4: awaitable type

In the *co_awaitexpr;* operator the expression is firstly converted to an awaitable type. If the promise_type type of the current coroutine has a member function await_transform, then the awaitable is obtained by calling promise.await_transform(expr), illustrated in Listing 2.5. Otherwise, the awaitable is expr as-is.

```
func get_awaitable(promise_type& promise, T&& expr){
  if P has member function await_transform:
    return promise.await_transform(expr);
  else
    return expr;
}

func get_awaiter(awaitable){
  if awaitable has member operator co_await:
    return awaitable.operator co_await();
  else if awaitable has non-member operator co_await:
    return operator co_await(awaitable);
  else:
    return awaitable;
}
```

Listing 2.5: Pseudo Code for deciding which awaiter/awaitable is used

Then, the awaiter object is obtained like illustrated in func get_awaiter in Listing 2.5. If no operator co_await is defined, the awaitable itself is the awaiter which implicates that a type can be an awaitable and an awaiter type simultaneously. For the simple example, in Listing 2.2, the expression std::suspend_always after the co_await is the expression which already is the awaitable as-is as it is an awaitable and has no await_transform()

member. The awaiter is also std::suspend_always as it has no operator co_await defined. Thus, std::suspend_always is both the awaitable and the awaiter type.

Then, awaiter.await_ready() is called , The coroutine is suspended awaiter.await_suspend(handle) is called, where handle is the coroutine handle representing the current coroutine. Inside that function, the suspended coroutine state is observable via that handle, if await_suspend returns void, control is immediately returned to the caller/resumer of the current coroutine (this coroutine remains suspended), otherwise if await_suspend returns bool, the value true returns control to the caller/resumer of the current coroutine the value false resumes the current coroutine. if await_suspend returns a coroutine handle for some other coroutine, that handle is resumed (by a call to handle.resume()) Finally, awaiter.await_resume() is called (whether the coroutine was suspended or not), and its result is the result of the whole co_await expr expression.

If the coroutine was suspended in the co_await expression, and is later resumed, the resume point is immediately before the call to awaiter.await_resume().

Additionally to suspending the coroutine, the co_yield expression returns a value to the caller and is equivalent to co_await promise.yield_value(expr). However, to define the type to return, the promise_type has to implement the yield_value() function.

Besides the awaiter/awaitable concept explained in the previous paragraphs, there is also the promise type concept. These are the two main interfaces, defined by the coroutine Type Specification (TS), for customizing the behavior of coroutines and co_await expressions:

- Awaiter / Awaitable: specifies methods that controls the semantics of co_await expression. When a value is co_awaited, the awaitable object allows to specify whether to suspend, execute some logic after suspensions (for asynchronously completed operations) and/or execute some logic after the coroutine resumes.
- Promise Type: specifies methods for customising the behavior of a coroutine, e.g. the behavior of any co_await or co_yield expression inside the coroutine body.

The promise_type struct has to be implemented in the coroutine wrapper type, which can be named arbitrarily, e.g. Task in Listing 2.2. It has to follow the scheme illustrated in Listing 2.6.

```
struct promise_type{
  // required methods
  ReturnType get_return_object();
  std::suspend_always initial_suspend();
  std::suspend_always final_suspend() noexcept;
  void unhandled_exception();
  // depending on ReturnType
  void return_void(); // if ReturnType is void
  void return_value(T value); // if ReturnType is T

  // optional methods
  auto await_transform(U&& value); // customize co_await behavior
  auto yield_value(V value); // customize co_yield behavior
}
```

Listing 2.6: Promise Type

The Promise type is determined by the compiler from the return type of the coroutine using std::coroutine_traits.

As mentioned in Subsection 2.1.2, each coroutine is associated with a coroutine handle. This handle is a non-owning reference to the coroutine frame and can be used to resume execution of the coroutine or to destroy the coroutine frame.

[1] [2]

2.2 Cache Misses and Software Prefetching

This section provides a background on cache misses, why they cause latencies in modern computer architectures by accessing data and how they can be potentially be hidden.

2.2.1 Definition of Cache Misses

To explain cache misses, first the underlying concept of caches in modern computer architectures has to be explained briefly. Caches are small, fast memory units located close to the CPU cores, designed to store frequently accessed data and instructions to reduce the average time to access memory. When the CPU needs to read or write data, it first checks if the data is present in the cache (a cache hit). Typically, there are three levels of caches (L1, L2, L3 or called LLC) with L1 being the smallest and fastest, and L3 being the largest and slowest. If the data is not found in the caches (a cache miss), it has to be fetched from the slower main memory, which incurs a significant performance penalty.

Cache misses are typically categorized into three main types: compulsory misses, capacity misses, and conflict misses. Compulsory misses occur on the first access to a block, as the data has not yet been loaded into the cache regardless of cache design. Capacity misses happen when the cache can not hold all the data actively used by a program during execution, meaning the working set exceeds the cache size. Conflict misses, also known as interference misses, occur when multiple memory addresses map to the same cache set, leading to evictions of useful data even though the cache is not full. For a multi-processor system this 3Cs group of cache misses can be extended to 4Cs. The 4th C are coherence misses arising from a cache line being invalidated due to modifications made by other processors (for L3 Cache as it is typically shared across cores) or threads (for L1, L2 and L3 Cache).

2.2.2 Cause of Cache Misses Data Structures

Modern database engines utilize various in-memory data structures that are directly addressed by virtual memory pointers. Memory blocks used by these data structures are typically allocated from the heap and chained together using pointers. For example, the nodes of an in-memory B+-tree are dynamically allocated and deallocated as the tree grows and shrinks. To traverse such a tree from its root node to the target leaf node, the accessing thread must dereference multiple pointers sequentially from the root to the leaf, forming a random access pattern.

This pointer chasing behavior poses a significant challenge for hardware prefetchers. Such access patterns are very difficult, if not impossible, for hardware prefetchers to predict accurately, leading to a high likelihood of last-level cache misses upon each pointer dereference. As a result, in main-memory systems, data stalls often dominate total CPU execution time. Beyond tree-based indexes, version chains in multi-versioned concurrency control systems can also cause a significant amount of data stalls when the accessing thread searches for a particular version. In some systems, over 50% of the total CPU cycles can be spent waiting for data to be loaded from memory to CPU caches. Hiding such stalls can potentially lead to much higher overall performance.

2.2.3 Techniques to counter Cache Misses

The literature presents various software techniques to address cache misses. Based on their impact on the number of cache misses and the incurred penalty, these techniques can be categorized into three main approaches.

- Eliminate cache misses by increasing spatial and temporal locality.
- Reduce the cache miss penalty by scheduling independent instructions to execute after a load.
- Hide the cache miss penalty by overlapping memory accesses.

The first approach aims to eliminate cache misses by increasing spatial and temporal locality. Locality can be improved through several methods: eliminating indirection by designing cache-conscious data structures, matching the data layout to the access pattern of the algorithm such that data accessed together is stored in contiguous space, or reorganizing memory accesses to increase locality through techniques like array blocking and tree blocking. However, these optimizations often require fundamental changes to data structure design and may not be applicable when the data structure layout is already optimized or when modifications would negatively impact other operations.

The second approach focuses on reducing the cache miss penalty by scheduling independent instructions to execute after a memory load. This approach increases instruction-level parallelism and leads to more effective out-of-order execution. To reduce the main-memory access penalty, a non-blocking load must be introduced early enough to allow independent instructions to execute while fetching data. This can be achieved through simple prefetching within an instruction stream or by exploiting simultaneous multithreading with helper threads that prefetch data. However, in scenarios like index lookups where one memory access depends on the previous one with few independent instructions in between, these techniques provide limited benefits.

The third approach seeks to hide the cache miss penalty by overlapping memory accesses. Modern memory systems can serve multiple memory requests in parallel (typically around 10 concurrent requests in current Intel CPUs), and exploiting this memory-level parallelism increases memory throughput. However, overlapping requires independent memory accesses that can be issued simultaneously. In the access chain of an index lookup, where each step depends on the result of the previous one, such independent accesses are not naturally available.

These traditional approaches provide limited benefits for individual index lookups with dependent memory accesses. To effectively hide cache misses in such scenarios, techniques that can create parallelism across multiple independent operations are needed, such as processing operations in groups with interleaved execution.

Hand-Crafted Approaches. The mechanism we just described fits naturally with many loop-based operations. Group prefetching and software pipelined prefetching [5] overlap multiple hash table lookups to accelerate hash joins. After a prefetch is issued, the control flow switches to execute the computation stage of another operation. Asynchronous memory access chaining (AMAC) [26] is 432 a general approach that allows one to transform a heterogeneous set of operations into state machines to facilitate switching between operations upon cache misses. A notable drawback of these approaches is they require developers hand-craft algorithms. The resulted code is typically not intuitive to understand and hard to maintain. This limits their application to simple or individual data structures and operations (e.g., tree traversal). Recent approaches tackle this challenge using lightweight coroutines, described next.

2.2.4 Software Prefetching

Hardware manufacturers have implemented various hardware prefetching techniques to automatically detect and pre-load data into caches before they are accessed by the CPU. These hardware prefetchers monitor memory access patterns and attempt to predict future accesses based on historical data. However, the reliance on hardware prefetchers has limitations, especially for applications with irregular access patterns where predictions may be inaccurate or insufficient.

Typically, these hints are communicated via specific instructions embedded within modern ISAs such as x86 and ARMv8. Especially in the context of trees and hash tables, many studies

The solution to overcome this is Software-based prefetching. It enables applications to proactively hint to the CPU about upcoming data accesses which for the hardware is hard or even impossible to predict. These hints, specific instrucctions in modern ISAs such as x86 and ARMv8, allow the hardware to move data asynchronously into the cache hierarchy, thereby reducing access latency by overlapping data transfers with computation.

With Software prefetching, proactively hinting the CPU what data to preload into the cache, timing of these prefetches is crucial for effectiveness. When software executes the prefetching instruction too early before the actual access, the data may have already been evicted from the cache when needed. Conversely, if prefetched too late, the data might not be transferred to the cache completely, potentially causing the CPU to stall. The presence of intricate memory systems like NUMA and high-bandwidth memory, as well as the increasing sophistication of CPUs with out-of-order execution and superscalar architectures, further complicate achieving precise timing.

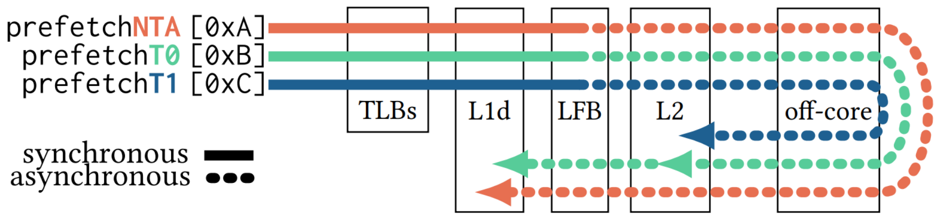

Figure 2.6: Lifecycle of prefetch instructions through the memory subsystem

Modern platforms offer several prefetch instructions to target different levels of the cache hierarchy, with each instruction initiating the transfer of one cache line. As visualized in Figure 2.6, when the software executes a prefetch, the logical address is first translated into a physical one. This procedure happens synchronously—if the translation is not cached within the Translation Lookaside Buffer (TLB), the CPU contacts a Page Miss Handler (PMH) before continuing execution of the prefetch instruction.

Once the physical address is known and the request misses the L1 data cache (L1d), the hardware triggers the transfer from memory to caches asynchronously. The L1d cache requests the cache line through the Line Fill Buffer (LFB), which acts as a buffer-like structure interfacing between the L1d and L2 cache to communicate requests on a cache-line granularity. For every cache line that misses the L1d, the CPU allocates a slot in the LFB. This nature makes it straightforward to implement asynchronous prefetches: the desired cache line does not cause the instruction to wait until the data is transferred into the cache but only until the miss is communicated to the LFB. Furthermore, the LFB enables the CPU to handle several pending requests simultaneously without blocking, supporting out-of-order execution and performing micro-optimizations like merging multiple requests to the same cache line.

The specific software prefetch instruction dictates how close the data is moved to the CPU. In the x86 ISA, most prefetch instructions clearly specify the cache level target for the fetched data: `prefetcht1` stores cache lines in the L2 cache, `prefetcht0` moves data into the L1d, and `prefetcht2` is generally aimed at the LLC (though on recent Intel platforms since Skylake with non-inclusive victim caches, data is placed into the L2 instead). The non-temporal `prefetchnta` instruction does not specify a particular cache target but aims to reduce cache pollution for data that will be accessed non-recurringly, with these cache lines being prioritized for quicker eviction.

Modern platforms exhibit two notable limitations in the implementation of software prefetches. First, the execution of a single prefetch instruction stalls until the virtual address is translated into a physical address. Although the TLB might accelerate this process, prefetched addresses are often not present in the TLB since software prefetching is primarily used when the application accesses scattered data objects. Consequently, the latency of the prefetch instruction is dominated by the latency of the PMH performing a page-table walk to translate the address. However, this restriction only applies to the initial prefetch operation within a memory page if several prefetches are performed consecutively.

Second, due to the limited number of LFB entries (typically between 3 and 24), the hardware can accommodate only a small number of outstanding memory requests. If an excessive volume of prefetch requests is issued rapidly, the LFB becomes saturated and cannot accept further requests. Under this circumstance, the CPU will either stall until an LFB slot becomes available or drop new prefetch requests. This limitation necessitates careful management of prefetch intensity to balance memory-level parallelism with hardware constraints.

3 Approach to hide cache misses with coroutines

3.1 Existing Techniques

Group Prefetching and AMAC

3.2 Why Coroutines and cache prefetching?

3.3 scheduler for multiple coroutines

```
template<const size_t numCoroutines>
  std::vector<typename ht_primary_key::Hashtable::Entry *>
  lookupHTEntries(const ht_primary_key::Hashtable &ht, const std::span<const size_t> &look
      // Compute the nr. of lookups for each coroutine
      // Initialize the coroutines
      std::vector<typename ht_primary_key::Hashtable::Entry *> entries;
      entries.reserve(lookups.size() / 2);
      std::array<Silent, numCoroutines> coroutines =
              ([&ht, &lookups, &entries]<size_t... Is>(std::index_sequence<Is...>) {
                  return std::array{lookup<numCoroutines>(ht, lookups, entries, Is)...};
              })(std::make_index_sequence<numCoroutines>{});
      // entries.reserve(lookups.size());
      //std::cout << "Starting lookup with " << numCoroutines << " coroutines." << std::en

      while (true) {
          size_t activeCoroutines = 0;
          for (auto &c: coroutines) {
              // std::cout << "Coroutine " << activeCoroutines << " is active." << std::end
              if (!c.h_.done()) {
                  activeCoroutines++;
                  c.h_.resume();
```

```
                }
            }
            if (activeCoroutines == 0) {
                break;
            }
        }

        return std::move(entries);
    }
```

Listing 3.1: Scheduler for lookup

3.4 Different coroutine approaches and their trade-offs

Citation test [4].

First approach was simple: create a coroutine each time from scratch for lookup. Bad performance, re-use coroutines so the creatio overhead is minimal and maybe create 5-20 coroutines and reuse them for all the lookups. First with co_yield concept and caller has to resume the coroutine, but for caller it is complex to distinguish if

3.5 Why Coroutines and cache prefetching?

not amac and gp

3.6 Cache Prefetching

3.7 testing on different machines and compilers and data structures

3.8 Benchmarking Setup

doing this on different compilers and machines

4 Using Coroutines and Cache Prefetching in different DataStructures for hiding cache-misses

4.1 Coroutines in chaining Hashtable

4.1.1 Implementation

```
template<const size_t numCoroutines>
   Task lookup(const ht_primary_key::Hashtable &ht, const auto &lookups, auto &results,
             const size_t offset) noexcept {
      auto idx = offset; size_t key; uint64_t hashIdx;

      for (; idx < lookups.size(); idx += numCoroutines) {
         key = lookups[idx];
         hashIdx = hashKey(key) & ht.mask;
         PREFETCH(&(ht.ht[hashIdx])); // Initial prefetch of the first value
         co_await std::suspend_always{}; // Suspend the coroutine until resumed

         auto result = ht.ht[hashIdx]; // Read the previously cached value

         while (!((result == nullptr) || (result->key == key))) {
            PREFETCH(result->next); // Prefetch the next entry in the linked list
            co_await std::suspend_always{};
            result = result->next;
         }
         if (result != nullptr) results.push_back(result);
      }
      co_return;
   }
```

Listing 4.1: coroutine lookup in chaining hashtable

4.1.2 Benchmarking and Measurements

4.1.3 Results

4.2 Coroutines in linear probing hashtable

4.2.1 Implementation

```
template<const size_t numCoroutines>
    Task lookup_ht_entry_t(const linear_probing_ht_duckdb::LinearProbingHashTable<uint64_t
                            const auto &lookups, auto &results, const size_t offset) noexcep
        auto idx = offset;
        size_t key;
        uint64_t hashIdx;
        uint64_t hash;
        linear_probing_ht_duckdb::LinearProbingHashTable<uint64_t, uint64_t>::EntryData *da

        for (; idx < lookups.size(); idx += numCoroutines) {
            key = lookups[idx];
            hash = ht.ComputeHash(key);
            uint64_t slot = hash & ht.bitmask;
            uint64_t salt = ht.ExtractSalt(hash);

            // Find SLOT
            linear_probing_ht_duckdb::ht_entry_t entry;
            while (true) {
                PREFETCH(&(ht.entries[slot])); // Initial prefetch of the first value
                co_await std::suspend_always{}; // Suspend the coroutine until resumed
                entry = ht.entries[slot]; // Read the previously cached value
                if (!entry.IsOccupied()) {
                    // Empty slot found
                    break;
                }
                if (ht.use_salts) {
                    // Check salt first (bloom filter)
                    if (entry.GetSalt() == salt) {
                        // Salt matches, need to check actual key
                        data = reinterpret_cast<
                                linear_probing_ht_duckdb::LinearProbingHashTable<uint64_t, ui
                            entry.GetPointer());
                        if (data && ht.key_equal(data->key, key)) {
                            entry = ht.entries[slot]; // Found matching key
                            break;
                        }
                    }
                } else {
                    // No salt optimization, check key directly
                    data = reinterpret_cast<
                            linear_probing_ht_duckdb::LinearProbingHashTable<uint64_t, uint64
                            entry.GetPointer());
                    if (data && ht.key_equal(data->key, key)) {
                        entry = ht.entries[slot]; // Found matching key
                        break;
                    }
                }

                // Continue linear probing
                data = nullptr;
                linear_probing_ht_duckdb::IncrementAndWrap(slot, ht.bitmask);
            }

            if(data == nullptr) {
                continue; // Not found
            }
            // Found an entry, retrieve all values in the chain
            linear_probing_ht_duckdb::LinearProbingHashTable<uint64_t, uint64_t>::EntryData
            while (current) {
                results.push_back(current);
                PREFETCH(current->next);
                co_await std::suspend_always();
                current = current->next;
            }
        }

        co_return;
    }
```

Listing 4.2: linear probing hashtable coroutine lookup

4.2.2 Measurements

4.2.3 Results

4.3 Coroutines in B+ Tree

4.3.1 Implementation

```
template<typename Key_t, typename Value_t, const size_t numCoroutines>
    Task lookup(const bptree::BPTree<Key_t, Value_t> &t, const auto &lookups, auto &result
                 const size_t offset) noexcept {
        auto idx = offset;
        uint64_t key;

        for (; idx < lookups.size(); idx += numCoroutines) {
            key = lookups[idx];

            auto node = t.root;
            while (!node->isLeaf()) {
                size_t i = 0;
                while (i < node->nr_keys && key >= node->keys[i]) {
                    i++;
                }
                PREFETCH(std::move(dynamic_cast<bptree::PtrNode<Key_t, Value_t> *>(node)->cl
                co_await std::suspend_always{};
                node = dynamic_cast<bptree::PtrNode<Key_t, Value_t> *>(node)->children[i];
            }
            bptree::LeafNode<Key_t, Value_t> *leafNode =
                    std::move(dynamic_cast<bptree::LeafNode<Key_t, Value_t> *>(node));

            // here I could do better search algorithm (binary search) but for now I just d
            for (size_t i = 0; i < leafNode->nr_keys; i++) {
                if (leafNode->keys[i] == key) {
                    results.push_back(std::make_tuple(std::move(key), std::move(leafNode->va
                    break; // Stop searching if key is found
                }
            }
        }
    co_return;
}
```

Listing 4.3: B+ Tree lookup coroutine

4.3.2 Measurements

4.3.3 Results

5 Integration of Coroutines in DuckDB

5.1 DuckDB Overview

DuckDB is a relational embeddable analytical DBMS that focuses on supporting analytical query workloads (OLAP). Similar to SQLite, DuckDB prioritizes simplicity and ease of integration by eliminating external dependencies for compilation and run-time. DuckDB's columnar-vectorized query execution engine reduces the CPU cycles expended per individual value, processing large batches of values in one operation as a vector. This design choice optimizes DuckDB for analytical queries, and in embedded scenarios, it enables high-speed data transfer to and from the database. To provide transactional guarantees (ACID properties), DuckDB employs its custom bulk-optimized multi-version concurrency control (MVCC) and enables data to be stored in persistent, single-file databases. Additionally, DuckDB supports secondary indexes to speed up queries trying to find a single table entry.

JOINS DuckDB supports several join algorithms, including hash join, sort-merge join, and index join. The system also implements join ordering optimization using dynamic programming and a greedy fallback for complex join graphs. It performs flattening of arbitrary subqueries and has a set of rewrite rules to simplify the expression tree, including common subexpression elimination and constant folding. The result is an optimized logical plan for the query, which is then transformed by the physical planner into the physical plan, selecting suitable implementations where applicable. In addition, DuckDB introduced a join algorithm called range join, which uses the interval encoding join (IEJoin) technique to handle range queries efficiently. IEJoin leverages min-max indexes to reduce the amount of data that needs to be scanned and uses SIMD instructions to further improve performance. Finally, DuckDB also supports an out-of-core hash-join for large tables that cannot fit in memory.

QUERY COMPILATION DuckDB does not support Just-in-Time (JIT) compilation of SQL queries, but instead chooses to use a vectorized model. This is because JIT engines require large compiler libraries such as LLVM, which can create additional transitive dependencies and make the deployment and integration process more complex. DuckDB aims to simplify this process by avoiding external dependencies and using templates for code generation.

QUERY EXECUTION DuckDB uses a vectorized interpreted execution engine that supports fixed-length types like integers as native arrays and variable-length values like strings as a native array of pointers into a separate string heap. To represent NULL values, DuckDB uses a separate bit vector that is only present if NULL values appear in the vector. DuckDB supports a vectorized pull-based model that is also known as "vector volcano". In this model, query execution starts by pulling the first chunk of data from the root node of the physical plan. A chunk is a horizontal subset of a result set, query intermediate, or base table. This node will recursively pull chunks from child nodes, eventually arriving at a scan operator that produces chunks by reading from the persistent tables. The process continues until the chunk arriving at the root is empty, indicating that the query has been completed. DuckDB uses a vectorized processing model that processes batches of columns at a time, which is optimized for CPU cache locality, suitable for SIMD instructions and pipelining, and has small intermediates that ideally fit in L1 cache. The system contains an extensive library of vector operations that support the relational operators, and this library expands code for all supported data types using C++ code templates.

5.2 Understanding the Linear Hashtable in DuckDB

5.3 Integration of Coroutines in DuckDB

DuckDB integration: linear probing hashtabale where if same key it is chained.

Difficulty in understanding the hashtable and implementing coroutines in huge concept of hastable withotu knowing full picture. has to be done in Join phase of the hastable (which is divided in different datachaunks and each datachunk is handled by a thread wher ea thread cna possibly handle more datachunks). In this thread coroutines can easily integrated as they are asynchronoulsy called and ...

5.4 Benchmarking

5.4.1 Results per machine

5.4.2 Results per compiler

5.4.3 results across machines and compiler

5.4.4 results per query

6 Evaluation

6.1 B+ Tree evaluation

6.2 Hashtable evaluation

6.3 Linear Probing Hashtable evaluation

6.4 DuckDB Evaluation

6.5 Coroutine Approaches evaluation

6.6 Coroutine in different Data structures evaluation

6.7 Coroutine in DuckDB evaluation

6.8 Summarized evaluation of coroutines in C++ for hiding cache misses

7 Conclusion

Abbreviations

List of Figures

List of Tables

Bibliography

[1] L. Baker. *C++ Coroutines: Understanding operator co_await*. `https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await`. Accessed: 2025-10-30. Nov. 2018.

[2] L. Baker. *C++ Coroutines: Understanding the promise type*. `https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type`. Accessed: 2025-10-30. Sept. 2018.

[3] L. Baker. *Coroutine Theory*. `https://lewissbaker.github.io/2017/09/25/coroutine-theory`. Accessed: 2025-10-30. Sept. 2017.

[4] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.