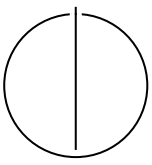
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

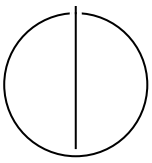
TECHNISCHE UNIVERSITÄT MÜNCHEN

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

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().

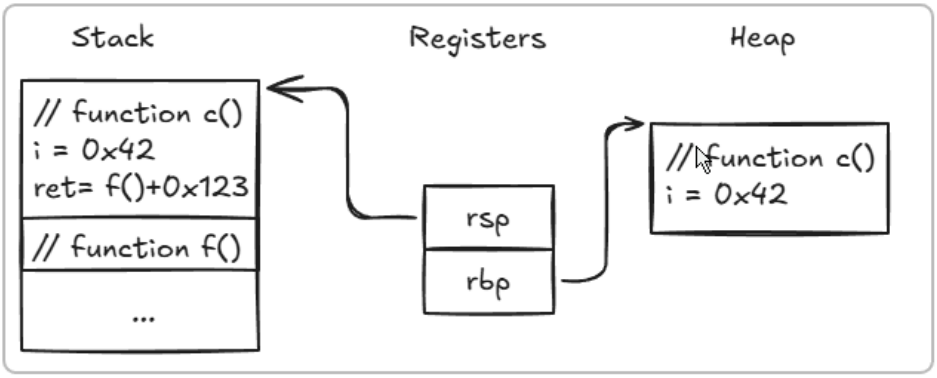


Figure 2.1: Calling a Coroutine - Heap Allocation

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

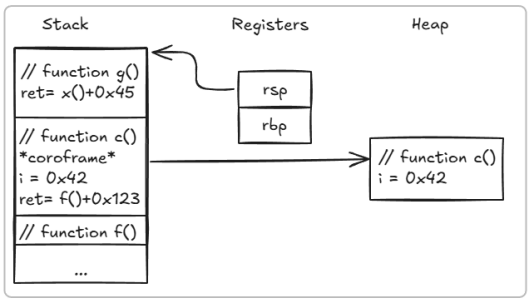


Figure 2.2: Coroutine calling a normal Function g()

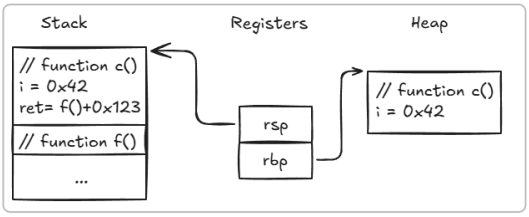


Figure 2.3: Normal Function g() returns

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).

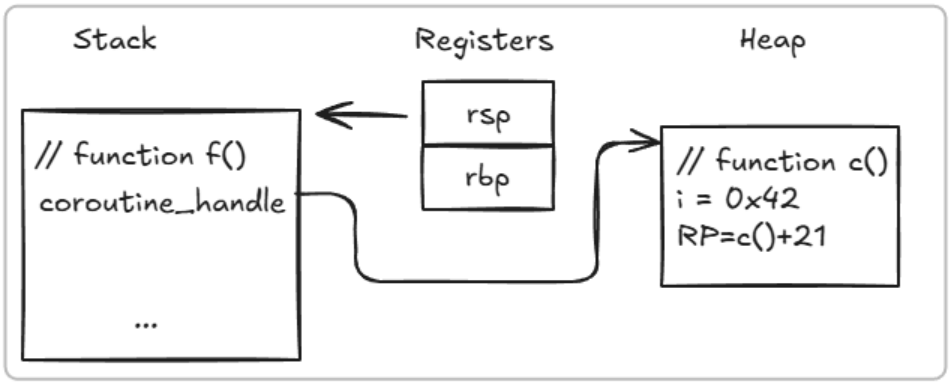


Figure 2.4: Coroutine hits Suspension Point

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.

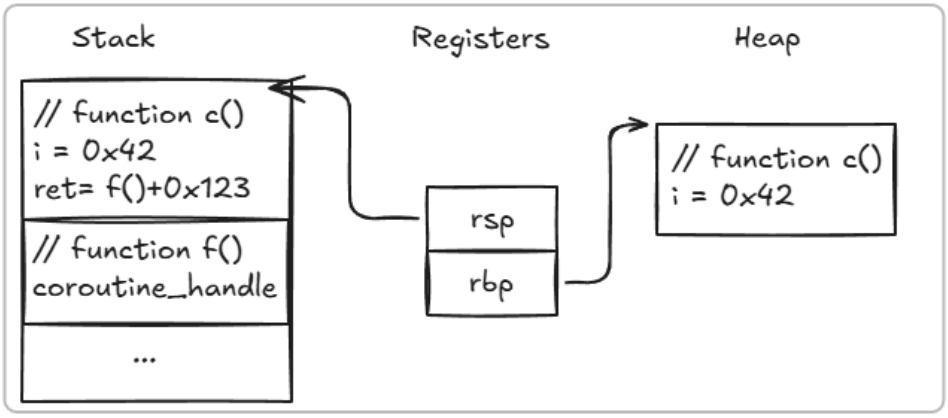


Figure 2.5: Resumption of a Coroutine

[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, the awaiter type, is firstly converted to an awaitable. 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.

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. In-

side 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.

The class template coroutine_handle can be used to refer to a suspended or executing coroutine.

CO_RETURN operator

[1] [2]

2.1.4 Comparison to other alternative solutions to hide cache misses

AMAC, Group Prefetching coro is better, not performance wise but easier to use, more general

2.2 Cache Misses

2.2.1 What are cache misses?

Cache misses are a fundamental concept in computer architecture, representing instances where requested data is not found in the cache, necessitating a slower access to main memory. They 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 is too small to hold all the data actively used by a program during execution, meaning the working set exceeds the cache size. This type of miss is directly related to the cache's capacity and can be reduced by increasing 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. These misses are influenced by the cache's associativity; increasing associativity can reduce conflict misses, especially in smaller caches.

The performance of a cache is often measured using metrics such as the cache miss rate, which is the percentage of memory requests that result in a cache miss, calculated as the total number of misses divided by the total number of memory requests over a given time interval. The miss rate is inversely related to the hit rate, with the sum of the two equaling 100The miss rate is a critical indicator for evaluating system

performance and can be used to guide optimizations in both hardware and software. For example, in multi-tasking environments, the last-level cache (LLC) miss rate is commonly used to assess an application's contention characteristics and sensitivity to resource competition. However, while the LLC miss rate is effective in distinguishing CPU-bound from memory-bound applications, it is less reliable for predicting the degree of contention or sensitivity in memory-bound workloads.

Research and literature have explored various strategies to mitigate cache misses. Increasing cache capacity can reduce both capacity and conflict misses, although it does not affect compulsory misses. Adjusting block size can also impact miss rates; larger blocks may reduce compulsory misses due to spatial locality but can increase conflict misses by reducing the number of sets in a fixed-size cache. Techniques such as loop nest optimization and virtual coloring are used in software to minimize conflict misses by ensuring that frequently accessed data does not map to the same cache set. Additionally, cache replacement policies like LRU (Least Recently Used) and write strategies such as write-back or write-through are designed to manage data in the cache efficiently, with the goal of minimizing the number of misses and their associated penalties.

The study of cache misses continues to be an active area of research, particularly in the context of modern multi-core and heterogeneous systems where contention and memory access patterns are complex. Performance monitoring units (PMUs) are often used to collect data on cache misses, enabling the development of contention-aware runtime systems and scheduling algorithms. Despite the availability of these tools, the challenge remains in accurately predicting the impact of cache misses on overall system performance, especially for memory-intensive applications.

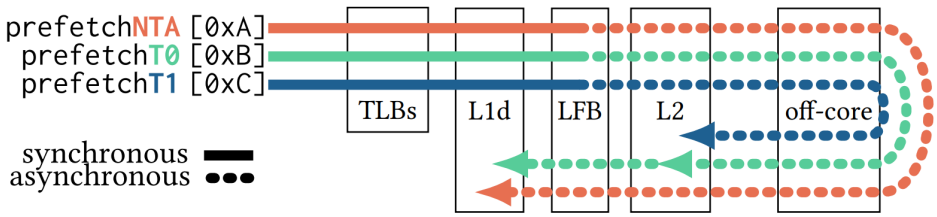


Figure 2.6: Prefetch Lifecycle through memory subsystem

2.2.2 When can they happen?

2.2.3 Why are they relevant in data bases and different index structures

2.2.4 Cache Misses in Data Structures

Chaining vs Open Addressing (Linear Probing) B+ Tree

2.3 Computer Architecture

2.3.1 Different Computer Architectures, x86-64, amd, apple, mips

2.3.2 Introduction to computer achitecture and storage layout

2.3.3 Different Compilers - Gnu vs Clang

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

3.1 Different coroutine approaches and their trade-offs

Citation test [4].

First approach was simple: create a coroutine each time from s cratch 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

3.2 Why CCoroutines and cache prefetching?

not amac and gp

3.3 Cache Prefetching

3.4 Coroutines in chaining Hashtable

3.4.1 Implementation

3.4.2 Benchmarking and Measurements

3.4.3 Results

3.5 Coroutines in linear probing hashtable

3.5.1 Implementation

3.5.2 Measurements

3.5.3 Results

3.6 Coroutines in B+ Tree

3.6.1 Implementation

3.6.2 Measurements

3.6.3 Results

4 Integration of Coroutines in DuckDB

4.1 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.

4.2 Understanding the Linear Hashtable in DuckDB

4.3 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 ...

4.4 Benchmarking

4.4.1 Results per machine

4.4.2 Results per compiler

4.4.3 results across machines and compiler

4.4.4 results per query

5 Evaluation

5.1 Coroutine Approaches evaluation

5.2 Coroutine in different Data structures evaluation

5.3 Coroutine in DuckDB evaluation

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

6 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.