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

Acknowledgments

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

2 Coroutines, Data Structures and Cache Misses Fundamentals

2.1 C++ Coroutines

C++ Coroutines are available from the C++20 standard on, and from the C++23 standard there is a generator based on coroutines.

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.

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 the coroutines state is saved to the heap, like illustrated in Figure 2.1 where a function f() calls a coroutine c().

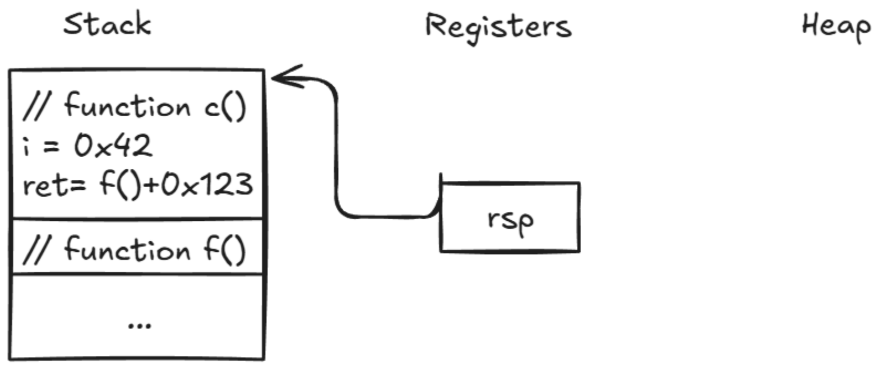

Figure 2.1: Calling a Coroutine

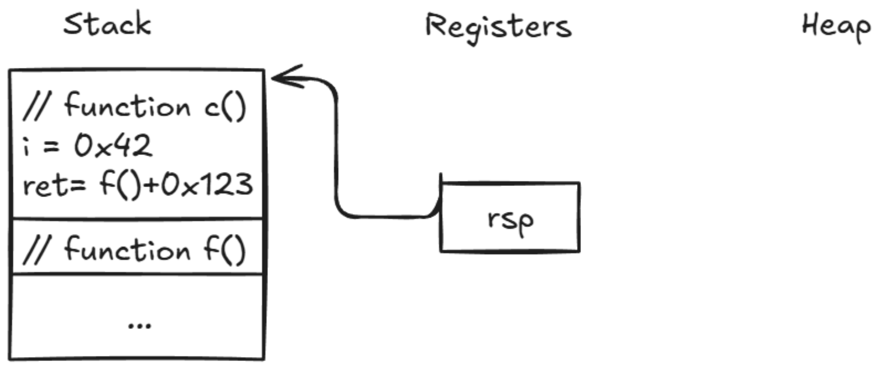

Figure 2.2: Calling a Coroutine

AWAITABLE An Awaitable has to provide the following three methods:

```
bool await_ready();

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

```
T await_resume();
```

Listing 2.1: awaitable type

PROMISE TYPE
[1] [3] [2]

2.1.1 The co_await, co_return and co_yield operators

co_await
awaitable (and awaiter) concept and promiseType
co_return
co_yield

2.1.2 Example of Coroutines

Simple co_await example

2.1.3 Simple generator (co_yield) example

Advanced Example

2.1.4 Tricks and Pitfalls

2.2 Computer Architecture, Cache Misses

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

2.2.2 Introduction to computer achitecture and storage layout

2.2.3 What are cache misses?

2.2.4 When can they happen?

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

2.3 Data Structures

2.3.1 Hashtable

Chaining vs Open Addressing (Linear Probing)

2.3.2 B+ Tree

3 Using Coroutines 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 Coroutines in chaining Hashtable

3.2.1 Implementation

3.2.2 Benchmarking and Measurements

3.2.3 Results

3.3 Coroutines in linear probing hashtable

3.3.1 Implementation

3.3.2 Measurements

3.3.3 Results

3.4 Coroutines in B+ Tree

3.4.1 Implementation

3.4.2 Measurements

3.4.3 Results

4 Integration of Coroutines in DuckDB

4.1 DuckDB Introduction

4.2 Understanding the implemented linear Hashtable

Citation test [4].

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 Results of timings

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.