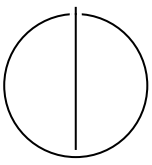
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

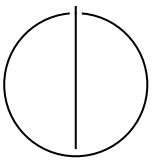
TECHNISCHE UNIVERSITÄT MÜNCHEN

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 [2].

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

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().

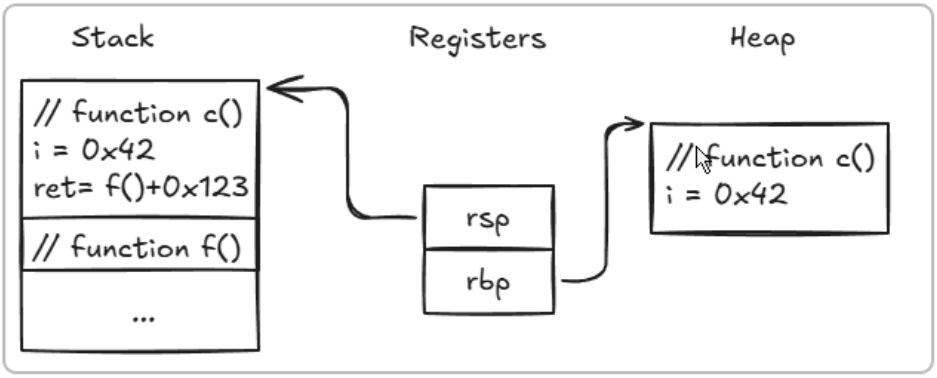


Figure 2.1: Calling a Coroutine - Heap Allocation

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

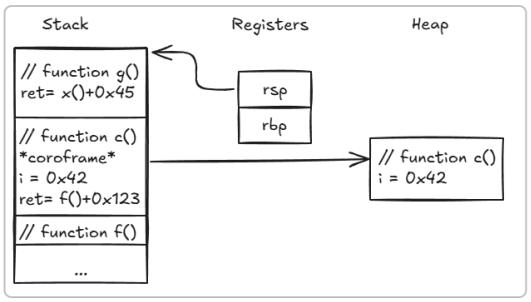


Figure 2.2: Coroutine calling a normal Function g()

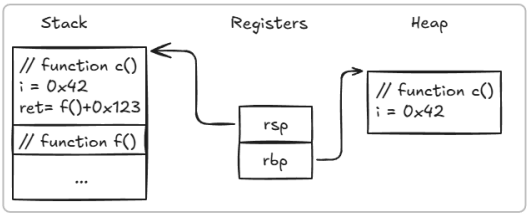


Figure 2.3: Normal Function g() returns

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).

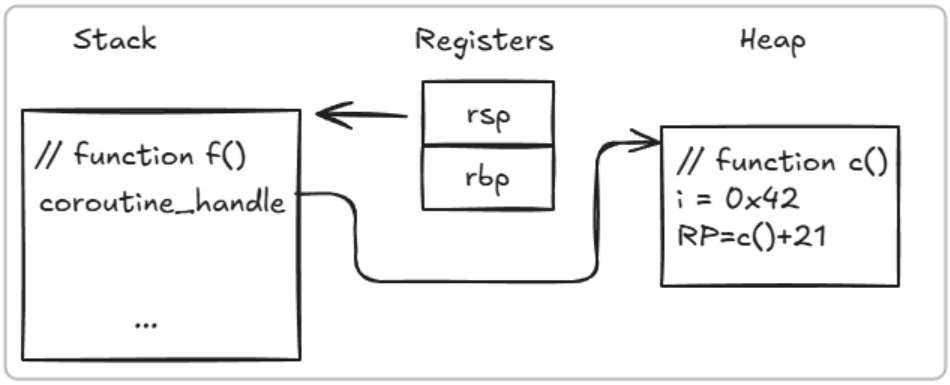


Figure 2.4: Coroutine hits Suspension Point

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.

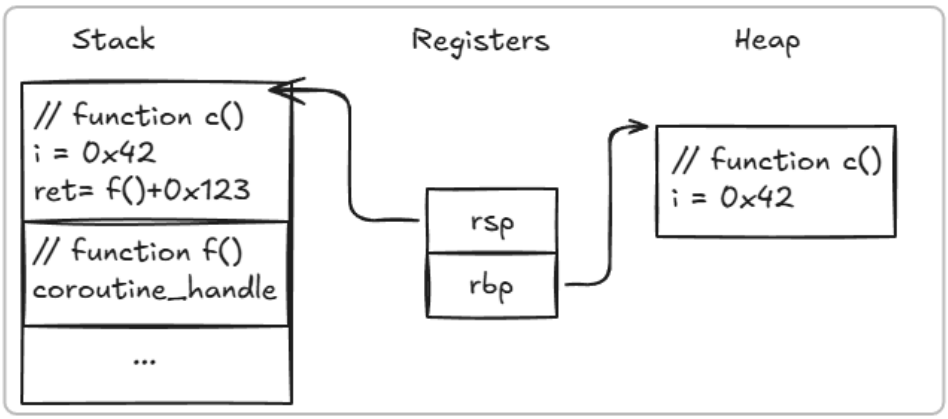


Figure 2.5: Resumption of a Coroutine

[1]

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.

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

Awaitable and Promise Type concepts
Two interfaces:

- Promise Type
- Awaitable / Awaiter

co_await;
AWAITABLE The standard library defines two trivial awaitables: std::suspend_always and std::suspend_never. 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

Pseudo Code for deciding which awaiter/awaitable is used.

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

func get_awaiter(Awaitable&& awaitable)
{
  if Awaitable has member operator co_await
    return static_cast<Awaitable&&>(awaitable).operator co_await();
  else if Awaitable has non-member operator co_await
    return operator co_await(static_cast<Awaitable&&>(awaitable));
  else
    return static_cast<Awaitable&&>(awaitable);
}
```

Listing 2.2: awaitable type

```
struct Task {
  struct promise_type; // forward declaration
  std::coroutine_handle<promise_type> h{}; // coroutine handle
  // constructor, move constructor, destructor
  explicit Task(promise_type* p) : h{std::coroutine_handle<promise_type>::from_promise(*p)
  Task(Task&& rhs) : h{std::exchange(rhs.h, nullptr)}{}
  ~Task() {if (h) {h.destroy();}}
  // promise type
  struct promise_type {
      int _val{};
      Task get_return_object() { return Task{this}; }
      std::suspend_never initial_suspend() { return {}; }
      std::suspend_always final_suspend() noexcept { return {}; }
      std::suspend_always yield_value(int value) {_val = value; return {};}
      void unhandled_exception() { }
  };
```

Listing 2.3: awaitable type

await_transform example

```
// #include <coroutine> #include <iostream> #include <utility>
struct Chat {
    struct promise_type {
        std::string _msgOut{}, _msgIn{};

        void unhandled_exception()noexcept{};
        Chat get_return_object() {return Chat{this};}
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(std::string msg) {
          _msgOut = std::move(msg); return {};}
        void return_value(std::string msg) noexcept {
          _msgOut = std::move(msg); }
        auto await_transform(std::string) noexcept {
            struct awaiter {
                promise_type& pt;
```

```
            constexpr bool await_ready() const noexcept {return true;}
            void await_suspend(std::coroutine_handle<>)const noexcept{}
            std::string await_resume() const noexcept {
              return std::move(pt._msgIn); }
        };
        return awaiter{*this};
      }
  };
  std::coroutine_handle<promise_type> _hdl;
  explicit Chat(promise_type* p) : _hdl{
    std::coroutine_handle<promise_type>::from_promise(*p)}{}
  Chat(Chat&& rhs) : _hdl{std::exchange(rhs._hdl, nullptr)} {}
  ~Chat() {if (_hdl) {_hdl.destroy();}}

  std::string listen() {
      if(not _hdl.done()) {_hdl.resume();}
      return std::move(_hdl.promise()._msgOut);}
  void answer(std::string msg) {
      _hdl.promise()._msgIn = msg;
      if(not _hdl.done()) {_hdl.resume();}}
};

Chat Fun(){
  co_yield "Hello\n";
  std::cout << co_await std::string{};
  co_return "Here!\n";}
int main(){
  Chat chat = Fun();
  std::cout << chat.listen();
  chat.answer("Where␣are␣you?\n");
  std::cout << chat.listen();
}
```

Listing 2.4: awaitable type

co_yield expression returns a value to the caller and suspends the current coroutine: it is the common building block of resumable generator functions. equivalent to co_await promise.yield_value(expr), but additionally you need to specify yield_value() function in the promise type.

2.1.4 Hands on example

2.1.5 Comparison to other alternative solutions to hide cache misses

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

2.2 Cache Misses

2.2.1 What are cache misses?

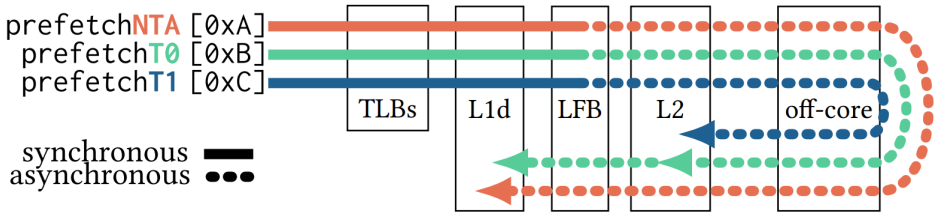


Figure 2.6: Prefetch Lifecycle through memory subsystem

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 in different DataStructures for hiding cache-misses

3.1 Different coroutine approaches and their trade-offs

Citation test [2].

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 [2].

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. *Coroutine Theory*. https://lewissbaker.github.io/2017/09/25/coroutine-theory. Accessed: 2025-10-30. Sept. 2017.

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