



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cppless: A single-source programming model for high-performance serverless

Bachelor Thesis

L. Möller

September 8, 2022

Advisors: Prof. Dr. T. Hoefler, M. Copik

Department of Computer Science, ETH Zürich

Abstract

Serverless functions have lately been getting traction in the world of high-performance applications where the dynamic scheduling features that serverless cloud environments exhibit can be used to offload CPU-intensive work to the cloud. This is especially advantageous for workloads where dynamic parallelism is required. However, using serverless platforms for this purpose remains difficult in languages like C++ which is traditionally used for high-performance application. To solve this problem we introduce cppless, a single-source programming model for high-performance serverless applications. Cppless allows users to write serverless functions together with the code that uses them to allow transparent offloading. This allows us to provide a common abstraction layer for serverless platforms and enables composable, modular architectures that make use of serverless functions. We evaluate Cppless using several high-performance problems. The results show that Cppless provides a low-overhead interface for serverless applications.

Contents

Contents	iii
1 Introduction	1
1.1 Organization of the Thesis	2
2 Background	3
2.1 Prior Work	3
2.1.1 Lithops	3
2.1.2 Crucial	4
2.1.3 Kappa	5
2.2 Comparison to Existing Frameworks	6
2.3 Serverless Environment	7
2.3.1 AWS	7
2.4 C++	8
2.4.1 Lambda Expressions	8
3 Design	9
3.1 Language Extensions	11
3.1.1 Alternative Entry Points	11
3.1.2 Lambda functions	16
3.1.3 Identification	17
3.1.4 Serialization	18
3.2 Low-level dispatcher API	19
3.2.1 Dispatcher	20
3.2.2 Task	21
3.2.3 User-facing API	22
3.3 High-level graph API	23
4 Implementation	25
4.1 Language extensions	25

CONTENTS

4.1.1	Alternative entry points	26
4.1.2	Lambda Functions	29
4.1.3	Identification	31
4.2	User-space library	32
4.2.1	Tasks	32
4.2.2	Graph Interface	33
4.3	Cloud Provider Support	33
4.3.1	Local Dispatcher	33
4.3.2	AWS Lambda	34
5	Evaluation	39
5.1	Benchmark Methodology	40
5.2	Fibonacci	40
5.3	Floorplan	41
5.4	Knapsack	43
5.5	N-Queens	44
5.6	CPU-Raytracer	48
5.6.1	Overhead analysis	50
5.6.2	Serialization	53
5.7	Pi-Estimation Benchmark	53
5.7.1	Overhead analysis	54
5.8	Micro-Benchmarks	55
5.8.1	Serialization	55
5.8.2	AWS Lambda Client	57
6	Discussion	59
6.1	Further Work	59
6.1.1	Partial Serialization	59
6.1.2	Advanced Communication Patterns	60
6.1.3	Detached Execution	60
6.1.4	Advanced high-level API	61
7	Conclusion	63
	Bibliography	65

Chapter 1

Introduction

Serverless computation is a new cloud paradigm focused on the execution of stateless and short-running functions in dynamically allocated execution environments such as containers. The main focus is on abstracting away the infrastructure required for execution and allowing transparent scaling.

Serverless has gained significant traction as a backend for websites, data analytics platforms, machine learning inference serving, and all tasks that can benefit from elastic resource allocation and pay-as-you-go billing. Lately, serverless has also gained attention in the world of high-performance applications where the dynamic scaling features that serverless environments exhibit can be used to offload CPU-intensive work to dynamically allocated workers. This is especially advantageous for workloads where the amount of parallelism changes throughout its execution or where the required amount of parallelism is not known ahead of time. For these use cases, serverless functions implementing parts of the algorithm can be deployed to the cloud provider of your choice.

As serverless functions typically follow a request-response model for data interchange it is the user's responsibility to serialize and deserialize all data that is required for remote execution. Furthermore, multiple separate applications have to be written which read and write the payload from the cloud provider, these different programs have to be compiled and deployed. This limits modularity and composability making it difficult for libraries to use serverless functions for acceleration. Lastly, the cloud provider has to be interfaced with to schedule invocations of the serverless function - this interface differs for each serverless platform resulting in the user being locked into a specific provider. These issues result in a high barrier of entry for serverless functions in the world of high-performance applications.

Cppless intends to solve these problems by providing a unified way in which generic serverless functions can be defined and invoked in the same code

base as the code using these serverless functions. This brings along a variety of advantages: Functions can be implemented without relying on the interface of a single cloud provider, cppless abstracts away the interface using an elegant high-level API allowing transparent usage of standard library container types to transfer data. Furthermore, functions, once deployed, can be invoked seamlessly through a familiar interface that is both generic and extensible, but at the same time provides opportunities for optimization. This allows users of the cppless framework to use serverless platforms easily without having to care about the details of how the data is transferred. At its core, this creates another level of abstraction around execution where the same code can be executed using different serverless platforms. Furthermore, the proposed framework makes accelerated programs more modular and composable, allowing libraries to use serverless functions for acceleration.

1.1 Organization of the Thesis

We start by analyzing the prior work in this area of research in section 2.1. We then discuss the main challenges that Cppless has to face, comparing our solution to existing work in section 2.2. In chapter 3 we go over the design of the cppless: Because cppless is by itself split into two parts, one that integrates into the compiler implements language extensions and one that builds on top of these language extensions, this chapter along with its succeeding one is split similarly. Once the design is discussed, chapter 4 covers the implementation of the cppless framework. Next, we discuss the performance and efficiency of the cppless framework in chapter 5. Here, we also discuss the limitations and outline further work in this area of research in section 6. Finally, we present the conclusions of this thesis in chapter 7.

Background

2.1 Prior Work

2.1.1 Lithops

Lithops, first presented in [9], is an ‘extensible, multi-cloud framework for transparently scaling regular and multiprocess Pythonic programs in the cloud’. At its core Lithops allows users to write programs utilizing serverless functions in a massively parallel fashion. It provides users with a familiar interface, similar to that of the built-in multiprocessing library. Once an action that normally would spawn a new process is invoked by the user, the executor part of Lithops will analyze the module tree of the current program and serialize all modules that the function which shall be offloaded needs to execute. This package of modules is uploaded together with the required data to the storage backend. The Code is then fetched together with its associated data by a serverless worker which communicates with the Lithops host using a Redis server. Once the data is deserialized, the function is executed and the data is written back into the storage backend. The host is notified using the Redis server once the result is uploaded.

Interprocess communication objects provided by Python’s multiprocessing module are reimplemented through implementations backed by the common Redis instance. It should be noted that Lithops advocates for using a generic worker: A single serverless function is deployed ahead of time, this worker will then download the required code from the storage backend once it is notified of a new invocation. This eliminates the potential overhead of function registration at runtime. Furthermore, the authors argue that this mitigates cold starts because the generic worker function can be used for different offloaded python functions because it downloads the relevant code itself from the storage backend.

Figure 2.1.1 demonstrates the usage of Lithops to compute an estimation

2. BACKGROUND

```
1  from lithops.multiprocessing import Pool
2  import random
3
4  def is_inside(n):
5      count = 0
6      for i in range(n):
7          x = random.random()
8          y = random.random()
9          if x * x + y * y < 1:
10             count += 1
11     return count
12
13 np, n = 96, 15000000000
14 part_count = [int(n/np)] * np
15 pool = Pool(processes=np)
16 count = pool.map(is_inside, part_count)
17 pi = sum(count)/n*4
18 print("Estimated Pi: {}".format(pi))
```

Figure 2.1: A simple example of how to use the lithops multiprocessing module

of pi using a classic Monte-Carlo approach: The function `is_inside` is offloaded to a serverless worker and executed in parallel. The results are then aggregated and the final result is computed. The interface of Lithops is transparent and elegant: It doesn't require any changes to the code that is offloaded. The user can directly offload functions without having to worry about the underlying infrastructure. This is especially useful for users who are not familiar with serverless platforms.

2.1.2 Crucial

Crucial, presented in [1], is "a system for the development of stateful distributed applications with serverless architectures". Function invocations are abstracted as threads that execute `Runnables`, similarly to how built-in Java Threads execute `Runnable` instances. Just like in Lithops, a generic serverless function is used, which, when invoked, initializes the `Runnable` class and runs the code with the supplied parameters. Due to the limitation that only `Runnable` is supported no data can be returned directly from the offloaded task, instead synchronization mechanisms have to be used.

Crucial uses a distributed shared object (DSO) layer for managing mutable, shared state. DSO objects are identified by a shared key and can be modified using atomic operations, allowing imperative algorithms to be ported seamlessly. Method calls on DSO objects are executed on DSO servers, thus allowing fine-grained updates.

Shared data, in the case, that it is immutable, can also be shared using object storage, the authors mention that this is especially relevant for the case of

input data which is often immutable.

```

1 public class PiEstimator implements Runnable {
2     private final static long ITERATIONS = 100_000_000;
3     private Random rand = new Random();
4     @Shared(key = "counter")
5     crucial.AtomicLong counter = new crucial.AtomicLong(0);
6
7     public void run() {
8         long count = 0;
9         double x, y;
10        for (long i = 0; i < ITERATIONS; i++) {
11            x = rand.nextDouble();
12            y = rand.nextDouble();
13            if (x * x + y * y <= 1.0) count++;
14        }
15        counter.addAndGet(count);
16    }
17 }
18
19 List<Thread> threads = new ArrayList<>(N_THREADS);
20 for (int i = 0; i < N_THREADS; i++) {
21     threads.add(new CloudThread(new PiEstimator()));
22 }
23 threads.forEach(Thread::start);
24 threads.forEach(Thread::join);
25 double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);

```

Figure 2.2: A simple example of how to use of Crucial

Figure 2.1.2 demonstrates the usage of Crucial, once again by computing an estimation of pi. The Crucial library provides a shared atomic long which is used to aggregate the results of the offloaded tasks. The Crucial library is a bit more verbose than Lithops, but it provides a more powerful interface.

2.1.3 Kappa

Kappa, presented in [13], allows users to utilize serverless functions using a familiar concurrency API by providing implementations for common primitives such as tasks and futures which offload work to a serverless environment. A process called checkpointing allows the use of long-running tasks on time-bounded serverless platforms. For this purpose, Kappa uses a compiler to add code that automatically commits checkpoints. These save a continuation of the current frame to the storage solution of the cloud service provider. Concurrency is provided using an API modeled after Python’s built-in multiprocessing package which thus should be familiar to users.

2.2 Comparison to Existing Frameworks

The major difference between Cppless and the prior work presented here is the programming language supported by the framework. Both Python and Java provide built-in mechanisms with which the internal byte-code of functions and classes can be extracted, which facilitates remote execution as the required metadata can simply be extracted, serialized, and send over to the cloud provider. Furthermore, the dependencies such as globals and libraries can be analyzed using runtime reflection features. The cross-platform compatible byte-code can thus easily be transferred at runtime from the host machine to the serverless environment.

This mechanism can't be ported directly to C++: The compiled machine code isn't portable and offloading all functions can lead to code-size issues: Different functions might require different dependencies and different cloud provider configuration options. Thus in cppless, functions that are offloaded at runtime have to be annotated in the source code indirectly such that the required code can be compiled for the target architecture of the serverless environment. This means that in cppless the execution packages for the offloaded code can be produced ahead of time, requiring the use of compile-time programming features to select functions for remote execution. However, this limits the behavior of Cppless compared to the other frameworks mentioned here but is expected due to the less dynamic nature of C++.

The compile-time integration of the architecture results in Cppless supporting a diverse set of host and offloading architecture combinations.

Furthermore, due to raw pointers and other low-level data structures, data in C++ programs can't be serialized and deserialized generically. Thus the task of providing serialization methods is left up to the user of the framework. The serialization library used for Cppless already provides implementations for most standard library types, thus these types can be used straightforwardly. Cppless also provides a generic serialization method for closure types which allows these types to be used without any additional effort.

Similar to Lithops, cppless models remote invocations in a similar way to how threads are represented locally. Given a sufficient invocable, task invocations can be started and waited upon. However, in contrast to the three frameworks presented here, cppless only provides shared-nothing parallelism so far. Adding support for shared operations should be possible using the existing language extensions and is left to further work.

In addition to the thread-style low-level interface, Cppless also provides a task-graph interface with which multiple task invocations can be connected at runtime using an elegant interface.

2.3 Serverless Environment

The general concept that we present here can be applied to a large majority of serverless environments and only minimal restrictions are enforced. To provide an overview of the inner workings of a serverless environment, we will go over the architecture and interface of AWS. Most other serverless environments follow a similar architecture and interface, thus the concepts presented here can be applied to them as well.

2.3.1 AWS

AWS offers a serverless environment through AWS Lambda. Developers can upload code to AWS Lambda in multiple languages which AWS supports itself, but they also support the use of a custom runtime where all programming languages which are supported on the target architecture can be used. Specifically, the user can upload a ZIP file (either directly or through AWS S3) that contains an independent executable. Once uploaded an AWS Lambda function can be invoked through an HTTP endpoint where the request data of the invocation is passed to the code of the Lambda function. Once the code of the Lambda terminates the return value of the invocation is sent back to the API client as the HTTP response.

Internally, AWS allocates a dynamic number of micro-VMs in which the code is executed. Due to the overhead of creating such a VM, instances are reused and kept alive for this matter. The amount of instances existing in parallel depends on the concurrency of the incoming requests. In general, we distinguish between warm and cold invocations: warm invocations are invocations that are handled by VMs which were created previously, cold invocations on the other hand exhibit a higher latency because a new micro-VM has to be started.

Runtime

AWS doesn't execute the uploaded executable each time a request has to be processed instead, the code has to interface with the runtime interface: The process is kept alive for the full duration that the instance is active. When new requests need to be handled an invocation-id is transferred to the user process through the standard input. The user process is responsible for retrieving the payload and sending the response by interfacing with an HTTP API available on a local endpoint.

Because the process itself survives multiple invocations the value of global variables persists as well in some cases. This process can also be used to our advantage in some cases.

2.4 C++

We generally assume high-level knowledge of C++ and thus don't go into detail about the language itself. The following section will explain some details about lambda expressions which should clear up the notation used in the following sections.

2.4.1 Lambda Expressions

Lambda expressions in C++ work in a similar way to how they work in many other languages, allowing users to defer the execution of a block of code while allowing the block to use variables from the outer scope. For this thesis specifically, some terminology and their internal representation are of importance. The result of a lambda expression is an object on which an overload of the call operator is defined. The body of the overloaded operator is the body as defined in the lambda expression itself. Internally the lambda expression, therefore, creates a stack-allocated instance of an anonymous record declaration which is called the closure type of the lambda expression.

Lambda expressions can reference variables from the outer context, through a process called capturing and there are two options for how this capturing should be done: Either by value or by reference - in both cases, a hidden, inaccessible field is created in the closure type which is used to store that value or reference. When constructed through the lambda expression the values are either copied into these fields or references to the local variables are created. By default all fields are const, but they can be collectively made non-const by applying the mutable operator to the lambda expression.

Although the closure type is anonymous it is possible to refer to the underlying closure type through the use of `decltype`, however, the closure type can only be default-constructed if no variables are captured and no default-capture is specified.

Lambda functions can also be generic: If this is the case the method that overloads the call operator for the closure type is generic.

Chapter 3

Design

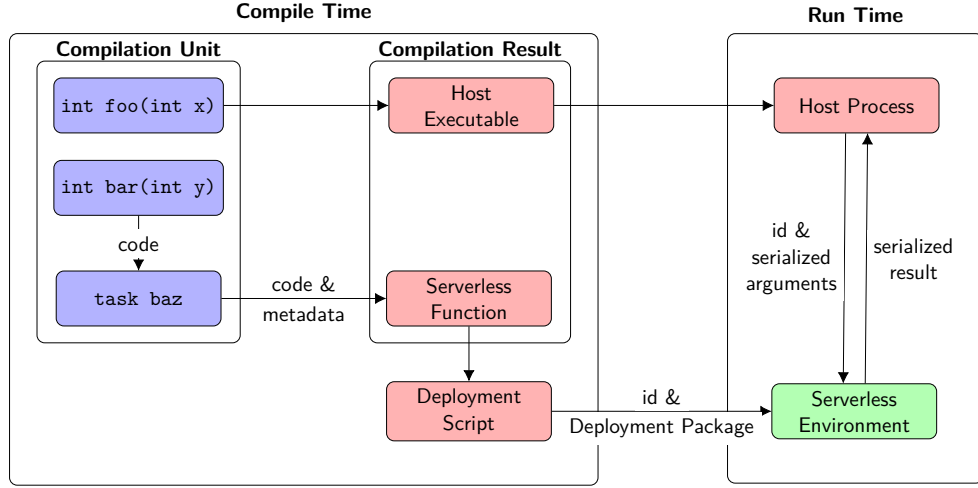
The concept of serverless is at its core the following: Users can upload a program to the cloud provider and that program can then be invoked with different parameters. Usually, the code is submitted directly, some cloud providers also allow containers or executables to be uploaded. This program is associated with some form of id, which later can be used to communicate to the cloud provider that you want the function to be executed with a specific set of inputs. Once the execution terminates the caller of the function gets the result back and the execution environment is destroyed eventually, destroying the state of the program.

Cppless is a single-source programming model for serverless computing, which means that both the serverless functions themselves and the host code calling them are authored in the same set of source files. A simple example can demonstrate this:

```
1  int bar(int x) {
2      return x + 42;
3  }
4
5  int foo(int y) {
6      cppless::task baz = [=](int x) { bar(x); };
7      return cppless::dispatch(baz, y);
8  }
```

To indicate that the function should be offloaded, the function `bar` is wrapped in a task object. Each task type corresponds to a single serverless function, which is deployed at compile time. Calls to `cppless::dispatch` are translated into function invocations on the serverless platform. The argument `x` and the return value are transparently serialized and deserialized between the host and serverless platform. The overall process is depicted in figure 3.1: `foo` is compiled to an executable using a standard C++ compiler. The task wrapping `bar` is compiled to an executable for the serverless platform

Figure 3.1: Overview of the architecture



and is deployed. `foo`, when called computes the id of the serverless function for `baz` and invokes it using the API of the serverless platform. The result is deserialized and returned to `foo`.

Cppless has a two-part design: It consists of a set of proposed language extensions, implemented as changes to the `llvm` compiler infrastructure as presented in [6], and a user-space interface written as a header-only library that uses the language extensions to provide an elegant API. These two parts are complemented by a set of tools that aid in the development of software using `cppless`. The language extensions are vital to the communication between the user-space library and outside tools and thus are discussed first.

Language extensions are used to achieve automatic compilation of serverless functions once they are declared in the source code as such, furthermore, the proposed language extension enable seamless serialization and one-way communication between the source code and the deployment script. The runtime library is used to provide serialization support and implements abstractions over cloud-provider APIs.

Compared to similar work for other programming languages, `cppless` does more work at compile-time, thus reducing the overhead that is incurred at runtime. This requires the analysis of which functions have to be offloaded at compile time. Cppless does this by requiring the user to wrap offloaded work in a task type: Wrapping the work that we want to offload in a task type indicates that we want to create a serverless function from it, to which we can then dispatch invocations.

3.1 Language Extensions

The main goal of cppless is that the process of offloading work should be seamless and easy. It becomes evident rather quickly that C++ itself doesn't offer an interface suitable to implement transparent offloading, thus language extensions are required. First of all, a way to annotate a function to be offloaded is required such that the corresponding resources for the serverless environment can be created after compilation. Furthermore, a method for handing over metadata from the C++ executed during compile time to the deployment script is required, this enables us to generate ids and configuration parameters using the same source code. As a result of the architecture of cppless, the same identifier for a function has to be generated both at compile time when deploying the function and at run time to invoke the function using the cloud provider API. Finally, the data required for the execution has to be serialized and deserialized between the host and the serverless platform. To make this seamless for certain types a language extension is required.

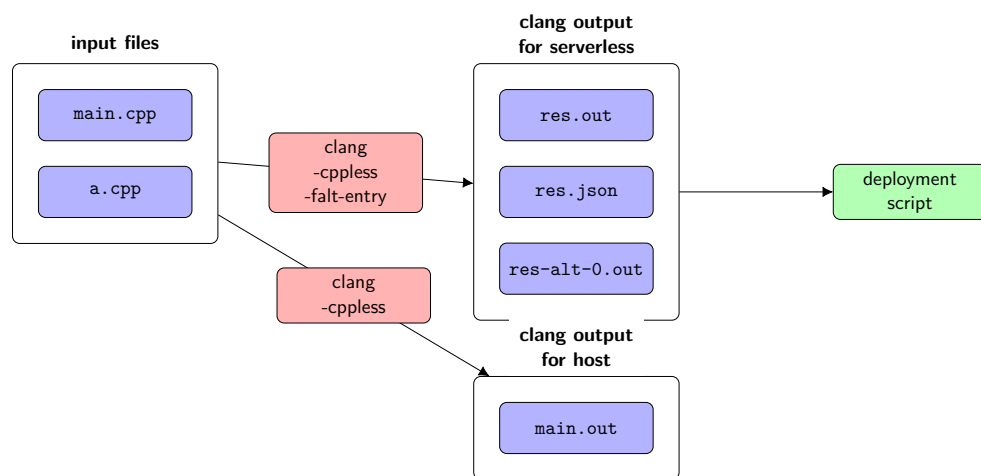
3.1.1 Alternative Entry Points

A major novelty of cppless is that it is a single-source programming model, i.e., the fact that a user can define serverless functions in the same compilation unit in which your host code is defined, allowing transparent integration. This poses the question of how this code is separated and packaged into units that can be deployed to a serverless function provider. Cppless solves this problem by introducing the concept of *alternative entry points*: Regular programs only have one entry point from which the program starts executing. A function, usually called `main`, is the entry point of a program and only one `main` function is allowed. On the other hand, programs making use of cppless have multiple entry points, one for the host program and multiple alternative entry points which the serverless functions will start once they are invoked. That means that effectively multiple programs are defined in a single compilation unit. Each compilation unit can have an arbitrary number of alternative entry points, but the completed program must only have a single `main` function. The proposed language extension allows functions to be annotated with an attribute `__attribute__((entry))`, marking the annotated function as an alternative entry point.

Whereas the compilation of regular programs only results in a single executable, the result of the compilation of programs using cppless is a collection of multiple executables, one for each alternative entry point in addition to the main executable. The alternative executables are an exact clone of the main executable, but with the main function replaced by the alternative entry point which the executable was generated for.

Figure 3.2 shows the compilation process for two compilation units. `main.o` is the output when the two source files are compiled for the host system where additional entry points are ignored. When applying a compiler flag additional entry points are respected and two additional files are generated. These can then be read by a separate deployment tool to deploy the serverless functions. In addition to that the compiler also emits a file that we will refer to as a *manifest* file - the specific format of this file is discussed later in this section.

Figure 3.2: Overview of the compilation process with an alternative entry point



Compilation

On the language level, alternative entry points are globally declared functions or static methods in your code that result in a separate executable binary. In their bare form, alternative entry points are declared as specially annotated functions, annotating a function has the following implications: The function is compiled when its surrounding template context is instantiated even if the static function itself is not used. This is necessary because the alternative entry point in most cases isn't called in the host program directly, but emitting it when the closest template context is instantiated allows other methods of the same class to assume that the alternative entry point is available. Furthermore, alternative entry points are treated as the main function and as such have to return an `int` and have two parameters acting as `argc` and `argv` with their respective types. These parameters are used to start the serverless function, the actual payload data is usually available using an API through which rich data can be exchanged. For serverless platforms which don't allow users to provide an executable directly, workarounds can be used where a shared library is generated and the main function of that library is called by the serverless platform through a custom wrapper that

is added by a deployment script.

Lastly, when the source file is compiled and the cppless options are enabled, an additional object file is created for each one of the alternative entry points. These object files have their original main function - if any - replaced by the function which was declared as an alternative entry point.

A minimal application making use of the alternative entry point feature could look like this:

```
1  __attribute__((entry)) auto alt_entry(int argc, char* argv[]) -> int
2  {
3      std::cout << "Hello alt_entry" << std::endl;
4      return 0;
5  }
6
7  auto main(int argc, char* argv[]) -> int
8  {
9      std::cout << "Hello main" << std::endl;
10     return 0;
11 }
```

This compilation unit has two entry points: One main entry point denoted by the main function and one alternative entry point defined by a function called alt_entry. Once compiled this would result in an additional binary file called a.alt_0.out in addition to a.out. This object file, once linked and executed, would output "Hello alt_entry" while running a.out would output "Hello main".

The resulting object files are numbered consecutively. This poses another question: How can we identify the different alternative entry points once they are in their compiled form? To solve this problem the compiler emits a manifest file in addition to the main object file and the enumerated object files of the alternative entry points. This manifest file which encodes metadata about each alternative entry point. This metadata is simply a list of objects containing the path of the emitted object file and the mangled name of the function that was annotated. Furthermore, because some data can't be fully determined by the function name itself, the cppless language extensions also provide users with the ability to pass opaque text through to the metadata file.

For this purpose we introduce a second annotation: metadata(data) - the metadata attribute can only be applied to alternative entry points and sets the opaque user-defined metadata for the entry point. data can be an arbitrary expression that is evaluated at compile time. The result has to be a fixed-string - a string with static storage containing a sequence of bytes. This byte-sequence could also encode structured data, but as the data is treated in an opaque way this detail is left to the user-space implementation.

An example of a manifest file could look like this:

```
1  {
2    "entry_points": [
3      {
4        "original_function_name": "_ZN7cppless4tash...E4mainEiPPc",
5        "filename": "renderer.cpp_alt_0.o",
6        "user_meta": "BAAAAAQCAAAAEWVwa...gX1pUUzhidmhfbm9kZT4="
7      }
8    ]
9  }
```

The key `original_function_name` contains the name of the function which was annotated as an alternative entry point in the mangling format of the target used to compile the compilation unit and thus cannot be used to identify a function across different architectures or compilation units.

In the above example `user_meta` is the binary encoding for the following JSON structure, encoding configuration options for an AWS lambda that should be created for the alternative entry point:

```
1  {
2    "ephemeral_storage": 64,
3    "memory": 2048,
4    "timeout": 10,
5    "identifier": "./benchmarks/custo..._ZTS4tile, _ZTS8bvh_node>"
6  }
```

This user-defined metadata is propagated through the compilation process and can be used by the deployment tool to create the required cloud resources.

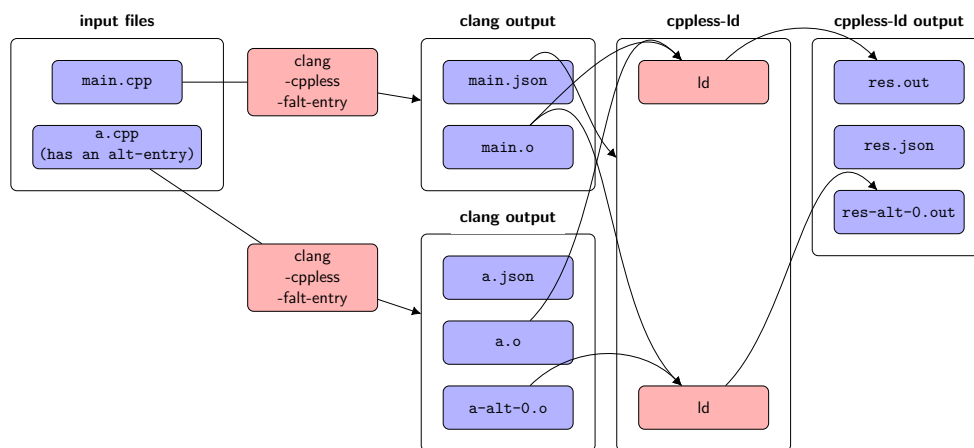
Linking

Complex C and C++ programs are mostly made up of multiple different compilation units, i.e., sets of source files that were compiled independently from each other only to be linked together either into a library or into an executable. This helps with incremental compilation and parallelism as compilation units are independent of each other and thus don't necessarily need to be recompiled all at once. Cppless thus needs to integrate into this process in a platform-independent way. As a result of the compiler output changing from a single output file into multiple output files, the linker input file format has to be changed too. To be compatible with all linkers which clang already supports cppless introduces a meta-linker called `cppless-ld`. It accepts all the command line arguments that the compiler driver accepts and in addition to that supports a few custom ones which can be used to configure `cppless-ld` itself. Cppless-ld is automatically selected as part of the compiler toolchain whenever the alternative entry point feature is enabled using the respective flag.

Cppless-ld takes the output of multiple compiler invocations with the alt-entry option being applied and merges them together into a new set of output files. It produces one regular output file, one additional output file for each alternative entry point included in the input set, and outputs a concatenated version of all of the manifest files included in the set of inputs.

The output of cppless-ld is - depending on the configuration - a set of binaries or a set of object files that can once again be used by cppless-ld again. Cppless-ld uses the linker as a black box and doesn't try to understand the output of the linker. This makes the process portable across different platforms and linkers.

Figure 3.3: Overview of the linking process of two files: main.cpp and a.cpp.



An example is visualized in figure 3.3: Two files main.cpp and a.cpp are linked together using cppless-ld, a.cpp defines one alternative entry point. The result is that three output files are produced: One containing the main executable, one manifest file, and one executable for the alternative entry point. Internally two linker invocations are performed: One for the main executable and one for the alternative entry point.

Deployment

Once all compilation units are compiled and linked together we end up with one set of output files: The main executable, a set of executables for the alternative entry points, and a manifest file containing metadata about the alternative entry points. This data can now be used to deploy the executable to a cloud provider, but it is also versatile enough to be used in other contexts too.

Deployment is not integrated into the compilation flow itself. It is the users responsibility to write or invoke a deployment script for the serverless envi-

ronment that they wish to use. The deployment script can simply be invoked on the output files using a build management tool like make or CMake. The deployment script can then use the manifest file to identify the alternative entry points and create the required cloud resources.

3.1.2 Lambda functions

In modern C++, lambda functions are everywhere: They make expressing callback functions easy by allowing users to define them inline with the function call that the callback should be passed to. As offloading work to a serverless platform also requires the user to provide a function that represents the item of work that should be executed, it is essential that lambda functions can be used seamlessly. Because serverless functions don't share memory with the host, data that is needed for execution has to be serialized and deserialized to be able to execute the code in the execution environment. To allow lambda functions to define items of work, they must be serializable together with their arguments. Once both the task and the arguments with which it shall be executed are serialized, they can be sent off, deserialized again, and the task callback can be invoked when the serverless function itself is triggered.

To serialize and deserialize lambda functions, their captured variables have to be serialized and deserialized too. This poses a challenge that cannot be overcome in C++ as it is specified right now because the hidden lambda capture fields are inaccessible. Thus we propose a language extension that can be used to write a serialization/deserialization method for arbitrary closure types. The underlying record declaration of the closure type is extended with two methods: A static constexpr method which can be used to access the number of captures that lambda functions of this type have and a second overloaded method, templated with a non-type-template parameter of integer type which can be used to access l-value references of the underlying fields. The non-type-template parameter acts as an index into a sequence of capture fields. These accessor methods can be used to both read and write the fields.

The following example visualizes these language extensions:

```
1  auto main(int argc, char* /*argv*/[]) -> int
2  {
3      double m = 12;
4      auto lambda = [argc, m](int /*q*/()
5      { std::cout << argc << " " << m << std::endl; });
6
7      constexpr int capture_count = decltype(lambda)::capture_count();
8      auto first_capture = lambda.capture<0>();
9      auto second_capture = lambda.capture<1>();
10
```

```
11  std::cout << "first: " << first_capture << std::endl;
12  std::cout << "second: " << second_capture << std::endl;
13
14  lambda.capture<0>() = 13;
15
16  lambda(5);
17
18  return 0;
19 }
```

The lambda function defined in this example captures two variables from the surrounding context: `argc` and `m`. Thus `::capture_count` returns the integer 2 as a constant expression. The accessor `.capture<i>()` returns the l-value reference to the underlying fields in a stable, but unspecified, order, thus two different outputs are viable for this example.

3.1.3 Identification

These alternative entry points can now be deployed to a cloud provider of your choice, but the problem of how different functions can be identified remains. Once we have an executable that shall be deployed, we need a name for the function, specifically an identifier is required that the host program can also compute, as it needs to know which function to dispatch invocations to. The architecture which the compiled functions use might not be the same as what the host system is using, thus using the natively generated mangled name on each platform can result in mismatches. The most straightforward solution to this problem is using the same mangling scheme for all platforms to generate this identifier. For this purpose, we introduce a macro-like function called `__builtin_unique_stable_name` which can be applied to a type and returns a literal-like string containing the mangled name of the type under the Itanium mangling scheme.

The mangled names are, however, still not always unique in the resulting binary, to illustrate this we can consider a simple example:

```
1  // a.cpp
2  static inline auto get_task() {
3      return []() { return 13; };
4  }
5  auto get_task_a() {
6      return __builtin_unique_stable_name(decltype(get_task));
7  }
8  // b.cpp
9  static inline auto get_task() {
10     return []() { return 12; };
11 }
12 auto get_task_b() {
```

```
13  return __builtin_unique_stable_nam(decltype(get_task));  
14 }
```

This program will compile and link successfully because both `get_task` declarations will at most result in a symbol with local / internal linkage the return value of both `get_task_a` and `get_task_b` is the same. The solution is once again straightforward: We simply have to prepend the compilation unit that was used as an entry point to the compiler invocation to the mangled name, making the resulting expression unique in most cases. There are still some obscure cases where a single source file is compiled several times with different compilation options and then linked together with itself, but as there is no generic way of solving this problem we are not handling these cases for now.

Inline namespaces pose a problem as different implementations of a library might be chosen depending on the host system. This specifically becomes evident when standard library types like `std::string` are used, which resolves to different templated classes depending on the C++ standard library implementation used. To avoid ABI problems, standard library implementations usually nest public symbols in an inline namespace, allowing libraries compiled against different C++ standard library implementations to be linked together. However, this also means that classes containing standard library types in their template instantiation, result in different mangled names depending on the standard library implementation used. To avoid this problem we use a slightly modified version of the Itanium mangling scheme where inline namespaces are ignored. This change only applies to the output of `__builtin_unique_stable_name`.

3.1.4 Serialization

All captured variables and the set of parameters passed to a function invocation have to be serialized to facilitate remote execution. As previously noted, there is no general way of serializing structured data in C++, especially because pointers might point to sub-objects of heap-allocated instances.

For example, in the following listing, we would need to serialize the entire array because it is legal to refer to the preceding element, but there's no way of finding all parent objects:

```
1  auto main() -> int {  
2      std::array<int, 3> x;  
3      int *b = x[1];  
4      auto fn = [&]() {  
5          return b[-1];  
6      };  
7      auto w = offload(fn, {});  
8  }
```


The captured type is a plain `int *`, but accessing elements that occurred in preceding elements of the parent array is valid.

To solve this problem, serialization is deferred to the user: Cppless uses the Cereal [4] serialization library, which includes serialization support for most standard library container types. However, support for custom types has to be added by the user to be able to use them in remotely-executed tasks.

Because the serialization functions of closure types are created using meta-programming features, we can require that the serialization functions of captured types are accessible. This works in an analogous way for arguments too: The user is thus alerted at compile-time when they try to use non-serializable types as a parameter or capture one in a task created from a lambda. This is a clear advantage over runtime solutions for single-source solutions which allows the compiler to catch errors early.

3.2 Low-level dispatcher API

The dispatcher interface provides a low-level abstraction to an underlying serverless platform. It only puts minimal constraints on the abilities of the serverless platform and allows the user to define the actual dispatching mechanism.

The dispatcher interface can be used as follows, assuming that `instance` was initialized previously:

```
1  int a = -1;
2  auto t0 = [=]() { return a + 3; };
3  int t0_result;
4  cppless::dispatch(instance, t0, t0_result, {});
5  instance.wait_one();
6  // t0_result can be accessed
```

The creation of an alternative entry point is abstracted away: It is created automatically because `cppless::dispatch` is instantiated with the concrete closure type. Internally, when a closure is provided, the default task type of the dispatcher is instantiated, that task type then exposes an alternative entry point. As all captured variables are serializable, a serialization and deserialization function is synthesized automatically, archiving the captured variable `a`. The dispatch function itself serializes the argument and both serialized parts are passed to the serverless environment. There, the task is executed and the result is transferred back to the host. The result is deserialized into `t0_result` directly and is available once the task has finished execution.

This interface is already generic over the specific serverless platform in use and thus allows the definition of high-level abstractions without the need to implement the low-level details.

3.2.1 Dispatcher

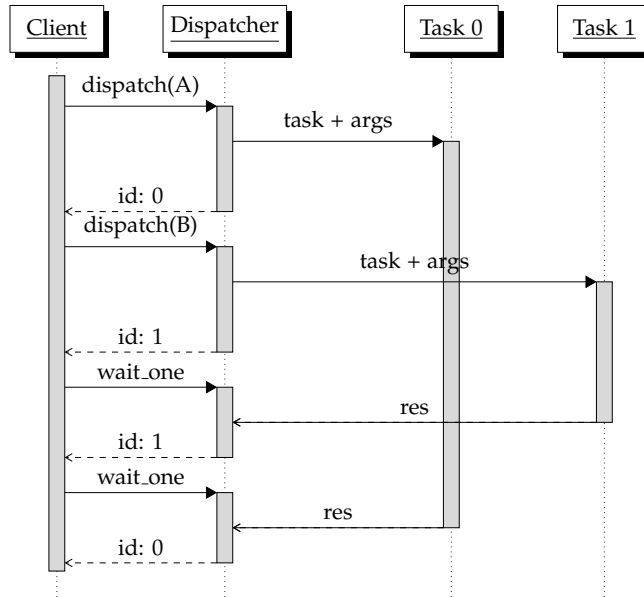
Dispatchers act as factories that can be used to create dispatcher instances, which can be used to offload the actual work and are intended to do the actual networking. Requiring this factory pattern brings several benefits: The underlying state of an instance might be opaque and can contain OS-specific handles that are not transferrable. The dispatcher itself however is only supposed to store the data which can be used to create connections to the serverless platform. This distinction allows users to copy, serialize and transfer the dispatcher itself while creating instances locally. Making the dispatcher a factory also means that the user can utilize it for creating different namespaces and to parallelize work by creating multiple instances providing their own separate thread-safety guarantees.

Once an instance is created, compatible tasks can be dispatched by calling the `dispatch` method of an instance, providing it with arguments for the tasks. Each task invocation, once dispatched, is associated with an identifier unique in the context of that instance. The user is also responsible for providing the storage space into which the result shall be written once the task invocation is completed. To await tasks to complete, instances provide a `wait_one` method, which blocks until an arbitrary task finishes, returning the id of each dispatched task exactly once.

Once a task invocation id was returned by the dispatcher instance, the result can be read from the storage space provided by the user. The interface of the `wait_one` method allows a diverse set of waiting behaviors, such that for example, a user could wait for the first one of two tasks to finish. Furthermore, this event-loop style allows the user to react to every task completion individually, which is useful for example when a task invocation is associated with a callback.

The dispatcher interface is supposed to provide an interface for reliable execution: Thus it is expected to handle errors and retry the task if the error indicates that the task invocation shall be retried. It should also be able to handle an arbitrary number of concurrently running task invocations. Furthermore, tasks may be invoked dynamically, even after the `wait_one` method has been called.

An example of a dispatcher interaction is depicted in Figure 3.4: A client is dispatching two task invocations using the `dispatch` method of an instance. The dispatcher is responsible for serializing the data and communicating with the serverless platform. Each invocation is associated with a unique id, which is returned to the client. The client then waits for the first invocation to finish using the `wait_one`, in this case task 1 finishes first. This is indicated by the return value of the `wait_one` call. Afterward, the client waits for the remaining invocation to finish.

Figure 3.4: A sequence diagram depicting an interaction with the dispatcher interface

3.2.2 Task

Tasks are an abstraction that provides a uniform interface to an underlying dispatcher. Tasks have to ensure for themselves that an alternative entry point is created, but have to delegate the actual execution to the dispatcher. They are the analogue to the `std::function` type in C++: `std::function` instances don't do anything themselves, but callables can be converted into `std::function` instances to retarget the `std::function` to the callable. The same holds for `cppless::task`: Sendable tasks can be converted into `cppless::task` instances, but the specific behavior is determined by the implementation.

Tasks can both be sent and received. To allow them to be sent, an implementation of the `serialize` method has to be provided, which will be used by the dispatcher to serialize the task. Furthermore, it has to provide a method returning the `id` of the task, this `id` shall be unique for each different type of task and is used by the dispatcher to invoke the task using the API of the serverless platform.

Task implementations are responsible for using the dispatcher to define an alternative entry point that the corresponding deployment script can process. The task implementation has to provide a configuration type, a receivable type that is callable, and an `id`. These values are passed to the dispatcher which will wrap the receivable variant of the task using the serverless platform API and will serialize the `id` together with the configuration into the user-defined metadata of the alternative entry point.

We provide a `lambda_task` class that, given a dispatcher type, a compile-time

configuration, and a lambda function implements the task interface. The serialization and deserialization methods are synthesized automatically using the aforementioned lambda-reflection mechanism. To create `lambda_task` instances without too much overhead a `lambda_task_factory` can be used which creates instances of the correct type. This indirection allows the user to write code that is generic over the dispatcher and the configuration.

3.2.3 User-facing API

To use a dispatcher directly the dispatcher type has to be instantiated and an instance has to be created.

A user-facing interface is provided using global overloads of a `cppless::dispatch` function, it handles automatic conversion of lambda functions into tasks and allows specifying config options which are passed to the lambda function factory. This means that a user can simply call `cppless::dispatch` with a lambda function instance, a reference to a result location, and a set of parameters, and under the hood, a task will be created transparently. Once tasks are dispatched the results can be awaited using `cppless::wait(dispatcher, n)`, which will wait until `n` tasks finish.

An example using the low-level dispatcher API is shown below.

```
1  const int n = 100000000;
2  const int np = 128;
3  double pi_estimate(int);
4  int main(int, char*[])
5  {
6      cppless::aws_dispatcher dispatcher;
7      auto aws = dispatcher.create_instance();
8
9      std::vector<double> results(np);
10     auto fn = [=] { return pi_estimate(n / np); };
11     for (auto& result : results)
12         cppless::dispatch(aws, fn, result);
13     cppless::wait(aws, np);
14
15     auto pi = std::reduce(results.begin(), results.end()) / np;
16     std::cout << pi << std::endl;
17 }
```

First, we create an instance of the default dispatcher for the AWS platform. Once we create an instance of the dispatcher we can dispatch tasks to it. The call to `cppless::dispatch` is instantiated with the closure type and a task is created. This task is then invoked `np` times, the results are stored in a vector and the `cppless::wait` function is called to wait for all tasks to finish.

Note that a lambda instance is passed to `cppless::dispatch` and not simply a function pointer. This is because function pointers aren't serializable and

thus cannot be sent to the serverless platform directly.

3.3 High-level graph API

In addition to the low-level dispatcher API, cppless offers a high-level graph API where a graph can be dynamically scheduled at runtime. It allows the user to build a graph where nodes consist of either tasks or initial nodes, these nodes are then connected to each other using edges representing data movement. The graph interface is split up into two parts: A builder interface used to construct the task graph and an executor interface used to execute the task graph once it is built. As the executor might want to store some state in the node or edge objects the builder wraps around the executor interface providing both a base layer on top of which the executor can build upon, and also provides an interface to the creator such that common behavior can be written generically and be reused.

Once a graph builder instance is created, initial nodes can be created using the `schedule(builder)` function. This creates a `sender<void>` which can be used as a timing dependency. Task nodes can be created using `then` which takes a set of dependencies that are passed to the task as arguments once all of them are completed.

An example of a graph builder is shown below.

```
1 auto a = schedule(builder);
2 auto b = then(a, []() { return 12; });
3 auto c = then(b, [](int m) { return m + 1; });
4 auto f = c->future();
5 builder.await_all();
6 std::cout << f.value() << std::endl;
```

This example creates an initial node `a` and then creates two task nodes `b` and `m`. The `c` node is dependent on `b`. The result of `b` is passed as an argument to `c` and the result of `c` is written to the future of `c`.

Once the graph is executed the `f` future will be ready and the result of `c` will be printed to the console.

By default, a `host_controller_executor` is provided which uses a central dispatcher instance to execute the task graph using the best achievable concurrency. However, the graph builder interface would also allow more specialized executors to be used, which for example could use specialized products of a cloud provider to do its job more efficiently.

Chapter 4

Implementation

We provide an open-source implementation of the previously presented design: We implement the language extensions on top of the `clang` compiler infrastructure and implement a separate user-space library. Both components are published as GitHub repositories. Programs using `cppless` have to be compiled using the modified `clang` compiler to ensure that the language extensions are supported, otherwise the program will not compile correctly. The language extensions are however designed to optimize the code-editing experience by ensuring that an unmodified `clangd` language server can provide code completion and other IDE-level features.

4.1 Language extensions

The language extensions are implemented as direct modifications to the `clang` compiler. The LLVM-project git repository was forked and modified. The modifications are required due to the restriction that `clang` doesn't provide any interface with which the existing AST can be modified, thus the plugin API can't be utilized. Furthermore, some subtleties required modifications of some core parts of the `clang` infrastructure. Specifically, the alternative entry point feature requires modification to the way the backend compilation process is invoked, which plugins can't modify. Furthermore, this allows for a simple, straightforward interface to the language extensions through the standard command line interface of the `clang` driver.

New flags were added to the `clang` driver using the existing infrastructure for adding new flags. The `clang` driver also uses flags to communicate with the `clang` compiler itself (`cc1`). A flag called `-cppless` allowed both as an argument to the driver and `cc1`, enables `cppless` mode in which the feature detection macro is enabled and some language extensions are enabled. A driver flag `-falt_entry` controls whether alternative entry points are emitted together with their manifest file. Internally, a different flag,

called `-falt_entry_output`, is used to communicate between the driver and `cc1`. The flag determines the location at which the manifest file and alternative entry points are emitted. The flag is only used internally and is not exposed as a driver flag.

This flag interface allows for two separate compilation modes: Host mod, where alternative entry points are valid but not emitted, but `cppless` feature detection and serialization are enabled, and an offloading mode where additionally alternative entry points are emitted.

4.1.1 Alternative entry points

Alternative entry points are an essential feature of the process that `cppless` uses to emit the executable binaries for serverless functions. They are identified by being annotated with a special annotation which also gets passed through the code generation process as an LLVM function annotation.

Clang uses a table definition file to parse annotations, thus it is sufficient to add an entry to the annotation table definition to add a new annotation. In this case, an annotation is added which can be applied to functions declarations and `CXXMethods`.

Clang uses `Decl::isUsed(bool)` to check whether a declaration is used in the current compilation unit. This method was modified such that it also checks whether the declaration is annotated with the `AltEntry` attribute. Furthermore `ASTContext::DeclMustBeEmitted` was modified to also check for the new attribute. This ensures that the alternative entry points are emitted to the LLVM module. During LLVM code generation top-level declarations of alternative entry points are eagerly emitted using `CGM.addUsed-OrCompilerUsedGlobal(var)`;

Templates

Alternative entry points are especially useful in templated contexts: A common use case for serverless applications will be to wrap a serializable callable into a class with an alternative entry point. Templated classes are often defined with in-class definitions of their methods, thus we want to handle cases where the alternative entry point method is defined in the class itself.

In-class definitions of classes are implicitly defined inline by default and thus are emitted lazily when they are first used. As alternative entry points are mostly not called directly by the host code this leads to unexpected behavior where the alternative entry point is not emitted into the LLVM module, furthermore, the modifications mentioned previously only handle top-level-declarations. Children of templated contexts are instantiated lazily such that instantiation of the surrounding class doesn't trigger instantiation

of all of its methods. Therefore we treat method declarations annotated as alternative entry points as being used even if they are not referenced. Clang already implements an annotation called `used` which leads to this behavior, alternative entry points are treated as implicitly used. Thus, we modify the template instantiation, to eagerly instantiate methods declared as alternative entry points. In templated contexts, this results in the function being emitted once the surrounding template context is fully instantiated.

Semantic Analysis

Alternative entry points are type-checked as if they were main function definitions. We implement this change by reusing the behavior of the existing semantic analysis of main function definitions to apply the same checks to alternative entry points. For example, this means that the following code is invalid:

```
1 __attribute__((entry)) auto alt_entry(char argc) -> int
2 {
3     return 42;
4 }
```

Compiling this example will generate the following error:

```
1 first parameter of 'main' (argument count) must be of type 'int'
2 __attribute__((entry)) auto alt_entry(char /*argc*/) -> int
3 ^
4 1 error generated.
```

Furthermore, alternative entry points implicitly return 0 if no return value is specified. This behavior is implemented to stay consistent with the behavior of main function definitions.

LLVM CodeGen

The LLVM code generation process is modified to support alternative entry points. Once it is decided which functions are supposed to be emitted, the expression associated with the metadata attribute of alternative entry points can be evaluated as a constant expression: The expression node is evaluated using `EvaluateAsConstantExpr`, the result is an `APValue` which we inspect. We expect the result to be a `fixed_string`¹ of char values, thus we rely on its internal structure to read out the string value. Internally, `fixed_string` instances are stored as struct instances with a single fixed-sized array member, thus this member is accessed and the result is converted at compile-time to an `std::string` available to the compiler program.

¹`fixed_string` is a string implementation backed by an array of compile-time known size, this allows them to be used in `constexpr` contexts more easily.

This process only needs to be done in the case that alternative entry points are enabled. The resulting binary string is attached to the LLVM function to which the current function is emitted, through an LLVM function attribute. The general fact that a function was generated from an alternative entry point is also denoted using an attribute.

BackEnd CodeGen

During backend code generation we ensure that the fact that a function is an alternative entry point is propagated through the compilation pipeline. The LLVM function created from the instantiated method is annotated with an LLVM attribute to ensure that it can be treated as such in the backend pass setup. Once the LLVM module is generated by the CodeGen module, the module is cloned for each alternative entry point, the entry point function itself is renamed to `main` and the original `main` function is removed in the case that there is one. We trigger code generation for each one of these modules, resulting in separate object files in the target object file format for each alternative entry in addition to an original, unmodified binary. At this point, we also output the manifest file.

Linking

The main interface to `clang` is its driver, which provides a command line interface to the underlying compiler. Its main job is to parse the command line arguments and invoke the appropriate tools to perform the desired task, it does this in a two-step process: At first, the arguments are passed and a list of tool invocations is computed, afterwards, the tool invocations are executed in topological order. The driver is both used to compile C / C++ files to object files, but also to link object files into executables. Build tools will usually use the driver to link the compiled files due to the uniform interface which it provides, thus the driver planning process has to be modified to produce multiple output files when the input file contained alternative entry points. Because the specific linker invocations which are required are only known once the compiler invocation is completed, we have to defer the process of planning the linking phase by delegating the task to a separate tool.

To implement alternative entry points we introduce a new tool called `cppless-ld` which acts as a cross-platform wrapper for the underlying linker, linking multiple alternative entry output files together. This executable is represented internally as a linker tool and it is used by the driver to link when the appropriate command line argument is passed.

Internally in `clang`, different platforms are supported by selecting between toolchains, toolchains are a list of tools that provide different functionality

for each required step, the most important ones being compilation, assembling, and linking. To provide support for cross-platform linking `cppless-ld` accepts the same command line interface as the `clang` driver itself, thus the driver will simply proxy its arguments to `cppless-ld`, adding several additional flags to instruct `cppless-ld` what to do. `cppless-ld`, implemented as a separate executable, starts by reading the manifest files of the input files and proceeds by invoking the original `clang` driver to link the different entry points together. One regular output file is produced which is the result of linking together the main object files of each input argument. In addition to that, one additional output file is produced for each alternative entry point in the set of inputs. Furthermore, the manifest files are merged and written to the output file.

The method for implementing alternative entry points presented here is focused on correctness and platform compatibility. Because the object files are treated as opaque data and no new data is added to them, the process should be compatible with any platform which `clang` supports. This means that host-platform executables like Mach-O and ELF are supported and tested, but in theory, WebAssembly and LLVM-Bitcode files² should also work with this approach.

4.1.2 Lambda Functions

As proposed in the design chapter, we introduce a compile-time reflection mechanism for inspecting and accessing the captured variables of closure types. We implement the proposed mechanism by adding methods to the anonymous record declaration of lambda expressions. Internally, after parsing a lambda function the method `BuildLambdaExpr` is called with information from the parser. This method creates a new anonymous record declaration in which the captured variables are saved as fields. This record type previously only had an operator overload for the `()` operator, which is used to invoke the lambda expression. In addition to that, several conversion operators are conditionally added, allowing some closure types to be converted to function pointers.

An example of the AST returned previously is shown below, the AST shown here is simplified:

```
1 TranslationUnitDecl
2 | -VarDecl cinit
3 |   -ExprWithCleanups
4 |     -CXXConstructExpr noexcept' elidable
5 |       -MaterializeTemporaryExp xvalue
6 |         -LambdaExpr
7 |           -CXXRecordDecl
```

²`clang` uses bitcode files as its output format to implement link-time-optimization

4. IMPLEMENTATION

```
8 |           `...
9 |   `~CXXRecordDecl
10 |     |-DefinitionData lambda
11 |       `~CXXMethodDecl operator() 'int () const' inline
12 |         `~CompoundStmt
13 |           `~ReturnStmt
14 |             `~IntegerLiteral
```

The anonymous record declaration is constructed in `BuildLambdaExpr`. For `cppless` two methods were added: A static `capture_count()` method which returns the number of captured variables and a `capture<I>()` method which returns the captured variable with the given index.

`capture_count()` is implemented as a static method on the record type which contains a compiler-generated body returning an integer constant. The method body is made up of a return statement that returns the number of captured variables as determined by `BuildLambdaExpr`.

`capture<I>()` is implemented as a function template declaration with a single non-type template parameter named `I`, returning `auto`. This return type of `auto` is required because the different overloads might return different types. The template parameter is used to indicate the index of the captured variable to be returned. For each capture variable, an explicit specialization is added to the template declaration, these overloaded specializations return an l-value reference to the captured variable by referring to the field of the anonymous record declaration.

We'll demonstrate this using the following example:

```
1  int main(int argc, char* argv[])
2  {
3      int a = 5;
4      double b = 4;
5      auto x = [=](int q) {
6          return a + b + q;
7      };
8      std::cout << x(42) << std::endl;
9      x.capture<0>() = 4;
10     std::cout << x(42) << std::endl;
11 }
```

The lambda expression is treated internally similar to the following C++ code:

```
1  struct lambda
2  {
3      lambda(int a, double b): m_a(a), m_b(b) {}
4      auto operator()(int q) {
5          return m_a + m_b + q;
6      }
```

```
7     constexpr static int capture_count() {
8         return 2;
9     }
10    template<int I>
11    auto& capture();
12    template<>
13    auto& capture<0>() {
14        return m_a;
15    }
16    template<>
17    auto& capture<1>() {
18        return m_b;
19    }
20    private:
21        int m_a;
22        double m_b;
23    };
24
25    int main(int argc, char* argv[])
26    {
27        int a = 5;
28        double b = 4;
29        lambda x(a, b);
30        std::cout << x(42) << std::endl;
31        x.capture<0>() = 4;
32        std::cout << x(42) << std::endl;
33    }
```

Similar code-generation methods are already used for the implementation of lambda expressions, thus this way of implementing the language extension comes naturally. Please note that explicit specializations are not valid in non-namespace scopes per language specification - clang however accepts this code without issuing any warnings. An alternative approach could use `constexpr` if statements to implement the `capture<I>()` methods in case the generated code becomes invalid in the future.

4.1.3 Identification

As proposed in the design chapter, a way to identify types across the different platforms is required to allow seamless offloading.

A macro-like function taking a type as an argument and returning a literal-like string used to identify the type was added. Internally, this function is backed by the clang implementation of `__builtin_sycl_unique_stable_name`, a feature added for the `sycl` support of clang, which has a similar use-case. A keyword named `__builtin_unique_stable_name` was introduced which is activated when the `cppless` extension is enabled. When it is parsed, a special expression AST node is created using `ActOnSYCLUniqueStableNameExpr`.

The AST node computes the mangling of the type and returns it as a string. A slightly modified mangling scheme is implemented by modifying the interface of the Itanium mangler. Specifically a boolean parameter `IgnoreInlineNamespaces` is added `mangleCXXRTTIName: void mangleCXXRTTIName(QualType T, raw_ostream &, bool IgnoreInlineNamespaces);`. The change to the mangling scheme itself is implemented in `manglePrefix` where inline namespaces are skipped, removing them from the mangling prefix.

4.2 User-space library

The user-space library is set up as a CMake-project which requires the custom version of clang to be built. Several benchmarks and examples are included in the repository. The library is set up as a header-only library, due to the heavy reliance on template metaprogramming. Dependencies are managed using the CMake integration of the Conan package manager.

4.2.1 Tasks

The task interface is implemented as a wrapper around a `std::unique_ptr` to an object extending the `task_base` abstract base class. `task_base` defines the abstract interface which task implementations have to provide: A serialization method used for serializing the task when sending it from the host to the serverless platform and a method identifying the task type. This interface is implemented by the `lambda_task` class which allows wrapping an instance of a capture class into a class providing the task interface. A `lambda_task` can thus be converted into a task which removes the generic dependence on the capture class type. This implementation is similar to the implementation of the STL `std::function` class, although less optimized.

Most functions in `cppless` take a task as an argument, which comes with the overhead of a virtual function call. An additional non-virtual layer could have been added in cases where the virtual function call can be avoided, but this would result in additional complexity.

Tasks interface with the dispatcher when serializing their configuration options and when defining an alternative entry point. For this purpose, dispatcher types have to expose a certain type-level interface providing access to the archive used for serialization and methods for wrapping the alternative entry point and for serializing the config. The configuration object has to be handled as opaque data: Tasks have to pass a valid configuration type to the serialization method of a dispatcher.

4.2.2 Graph Interface

The graph interface provides a high-level interface for building task graphs dynamically. The API to the graph interface is available through graph builder instances. Graph builders use executors to execute the graph once it is built. The general internal structure of the graph is as follows: The graph is represented as a vector of nodes that extend the `node_core` abstract base class. Nodes can be senders, which inherit from `sender_core` and receivers, which inherit from `receiver_core`. Receivers contain a tuple of `receiver_slot` instance into which senders can connect. Senders can connect to multiple receiver slots, the receiver slots a sender is connected to are stored in a vector. Senders can be turned into a future, if a future is extracted from a sender the result has to be sent to the host.

This basic graph interface is implemented as an extensible assortment of classes: For each of the above classes, a base class is defined which a graph executor implementation can inherit from. Generic wrappers for these classes are provided which allows the graph builder interface to depend on a common interface.

Two types of nodes are defined by default: `source_node` which is a sender of type `void`, providing an entry point into the graph, and `task_node` which takes a tuple of variadic dependencies and produces a single value. New nodes can be defined by executors by inheriting from the base classes and implementing the required methods.

A host controller executor is implemented which has a compile-time dependency on a dispatcher: The dispatcher creates lambda tasks for each task node. The graph is executed using the generic dispatcher interface. For each node, the number of remaining dependencies is tracked. Once a node has no remaining dependencies, the task is executed and the result is sent to the host. Successor nodes are triggered once the result is received. Once all nodes have been executed, the execution is terminated.

4.3 Cloud Provider Support

So far cloud provider support is limited to AWS Lambda. Furthermore, for local testing, a dispatcher which executes tasks in a separate operating system process is provided.

4.3.1 Local Dispatcher

The local dispatcher serves as the reference implementation for the dispatcher interface. When created, the manifest file is read from the disk and the dispatcher is initialized with the information contained in the manifest.

When a task is submitted, the dispatcher creates a new operating system process using the `fork` system call, using the executable that is specified in the manifest. Furthermore, pipes are created such that they can be used to communicate with the child process. Once the child process is initialized, the task data along with the arguments is serialized and sent through the pipe to the standard input of the child process. The child process deserializes the task data, executes the task, serializes the result and sends it back to the parent process. The parent process waits for the child process to terminate, deserializes the result and returns it to the caller.

Internally a new thread is created in the host process for each task that is dispatched to watch the child process. This allows for several tasks to be executed in parallel.

The local dispatcher has little use in a production environment, but it is useful for testing and debugging.

4.3.2 AWS Lambda

To add support for a cloud provider two main things have to be implemented: A dispatcher and a deployment tool. The dispatcher has to provide a wrapper around a generic callable implementation, furthermore, a dispatcher has to provide an implementation of the `dispatch_impl` method which is used for invoking an offloaded task.

The offloading wrapper for AWS Lambda is built on top of the `aws-lambda-cpp` project maintained by AWS itself. It provides a straightforward interface to deploy AWS Lambda functions using C++ by implementing the custom runtime interface which the executable uses to communicate with the AWS Lambda runtime. The request and response objects can be used to pass data to and from the function.

Serialization

AWS Lambda restricts the data that is passed to lambda invocations using the request body to valid JSON. This leaves two main options for serialization: Either the data can be directly serialized to structured JSON or the data can be serialized to some other string and then base64 encoded. Both options are implemented and can be reused for other dispatchers, by default the binary archive is used, but the user can choose to use structured serialization if desired for debugging purposes.

The lambda invocation response can be an arbitrary string of bytes, thus any archive can be used to serialize and deserialize the response. Once again, the binary archive is used as the default, but the user can choose to use other provided serialization methods.

Dispatching

Two dispatchers for AWS Lambda are implemented: Either an HTTP/2-based implementation building on `nghttp2` [12] or an HTTP/1.1-based implementation using the `boost.beast` [3] library. Both have different trade-offs: HTTP/2 is generally faster and more efficient in cases where many different requests are to be sent at the same time while the `boost-beast`-based implementation is more flexible and portable. Furthermore, the overhead of HTTP/1.1 is negligible compared to HTTP/2 in the case that only a few requests are to be sent, which makes it more useful for recursive invocations.

The `nghttp2`-based dispatcher uses round-robin scheduling to assign requests to a pool of HTTP/2 connections to the AWS Lambda API. This allows for a high number of concurrent requests, allowing it to use the available concurrency. Furthermore using a pool of connections decreases the probability of head-of-line blocking problems that HTTP/2 can exhibit. Furthermore, a naive resending strategy is used.

The `boost-beast`-based implementation is rather straightforward and is issuing a TCP-backed HTTP request for each invocation. This means that the number of concurrent requests is limited by the number of file descriptors available to the process.

Deployment

To deploy a lambda function, a ZIP file containing the executable itself and its required dependencies has to be created. The executable itself has to be compiled to the architecture of the lambda function, currently, `x86_64` and `arm64` are supported. Because the target architecture for the lambda function isn't necessarily the host architecture, cross-compilation is needed. This requires a toolchain with a linker, assembler, and a `sysroot` for the target architecture. The `sysroot` is extracted from a docker image which is built from a `Dockerfile` in which essential packages are installed. This allows for portable cross-compilation.

When building on a Linux system we also provide support for native compilation, leveraging the default `sysroot` and toolchain,

Once a compiled binary is available, a python script is used to create the deployment packages and upload them to AWS. At first, the manifest file is inspected and the different binaries are located. The `ldd` tool is used to find the libraries which are dynamically linked against and the binary, together with the libraries are copied into an in-memory zip file. The zip file is then uploaded to AWS. In the case that the binary is cross compiled, the commands can be run in the docker container from which the toolchain was extracted to ensure that tools like `ldd` and `strip` work correctly on the binary. This behavior can be controlled using a command line flag. For

Linux systems, we generally recommend the native versions to be used, to maximize compilation speed.

The deployment script is integrated into the build system using a set of CMake functions. The application has to be built effectively twice: Once for the host architecture and once for the target architecture. The flags and configuration values used for these two different targets might differ, thus different CMake configurations are created. This is achieved by having a separate build configuration for each target which does the cross-compilation. Specifically, the root configuration created by the user adds itself as a dependency to a special target by using `ExternalProject.Add`. These details are abstracted away from the user using a CMake function `aws_lambda_serverless_target($target_name)`, given a target name it will create an additional target `serverless_${target_name}` which when built will trigger a cross-compilation build and afterwards deploy the result to AWS. This allows users to easily integrate cppless into their build system. An example could look like this:

```
1 add_executable("some_target" main.cpp)
2 target_link_libraries("some_target" PRIVATE cppless::cppless)
3 aws_lambda_target("some_target")
4 aws_lambda_serverless_target("some_target")
```

First, a target is created from a source file, the target is linked against the cppless library. Furthermore cppless-specific options are added using the `aws_lambda_target($target_name)` function and the special serverless target is created by using `aws_lambda_serverless_target$target_name`. This allows the user to trigger a full build of both the host program and the offloaded parts by building the target `serverless_${target_name}`. When changes only occur in the host program, the target `${target_name}` can be built directly.

Configuration

AWS Lambda functions can be configured to run with different resource limits and timeouts, our dispatcher implementation allows these values to be specified for each task individually at compile time, meaning that heterogeneous task configurations become possible while sane defaults are applied. To pass the config from the language level to the deployment script the user metadata feature of alternative entry points is used. The lambda dispatcher uses a custom binary format to serialize structured compile-time data into a flat binary fixed-string buffer. This fixed-string is then base64 encoded and attached to the alternative entry point using the metadata annotation. After linking this user-meta is read once again, deserialized, and then used to configure the lambda function.

An example of a lambda with a non-default configuration is:

```

1  int result = 0;
2  auto task = [] { return fib(42); };
3  using config = lambda::config<lambda::with_memory<512>,
4                      lambda::with_ephemeral_storage<64>>;
5  cppless::dispatch<config>(instance, task, result);

```

Once deployed this will create a lambda function with a memory limit of 512 MB and an ephemeral storage limit of 64 MB.

The result of `lambda::config` is a record declaration type with public `constexpr` static members, resembling the following:

```

1  struct config
2  {
3      constexpr static unsigned int memory = 512; // MB
4      constexpr static unsigned int ephemeral_storage = 64; // MB
5      constexpr static unsigned int timeout = 10; // seconds
6  };

```

Due to the timeout not being set in the creation of the config using `lambda::config` this value is set to the default as configured in `dispatcher::default_config`.

To be able to distinguish between different functions with different configuration values the configuration values are hashed together with the name of the task to create a unique identifier for the function. Furthermore, the lambda function names share a prefix which is useful for identifying which program a lambda function belongs to. By default, the AWS CMake library sets this prefix to the name of the target which builds the application.

Due to the limitations of `fixed_string`, a standard encoding like JSON or BSON is difficult to implement, thus a custom encoding is used which was developed with these constraints in mind. The encoding is made up of different values, each one representing either an unsigned integer, a signed integer, an array, a dictionary, or a string. Arrays can contain heterogeneous values and maps map a key of arbitrary type to a value of arbitrary type. In C++ the values are represented using variadic data containers, resulting in their total size being known at compile-time, thus strings also have to be of type `fixed_string`. The resulting data structure can then be encoded to a single `fixed_string` by applying the `serialize` function. The result is a binary string and thus has to be base64 encoded. For this purpose, the base64 encoding method of the `beast` library was ported to support the custom `fixed_string` implementation.

An elegant `constexpr`-safe interface is provided, which makes it simple to construct values of this custom format. The following is an excerpt of the meta-data serialization code of the dispatcher implementation for AWS Lambda.

4. IMPLEMENTATION

```
1  template<unsigned int N>
2  constexpr static auto serialize(basic_fixed_string<char, N> identifier)
3  {
4      using namespace cppless;
5      return encode_base64(serialize(
6          map(kv("ephemeral_storage", Config::ephemeral_storage),
7              kv("memory", Config::memory),
8              kv("timeout", Config::timeout),
9              kv("identifier", identifier))));
10 }
```

Metrics

Traces utilizing the AWS Lambda dispatchers contain the request-id of the invocation, the function name, and the function version. This allows users to inspect the metrics of all invocations made by a program using cppless. To aid in the analysis of metrics we also provide a tool called `aws_trace` which given a trace file will extract the invocation ids, function names, and function versions to retrieve the AWS reports from the log files. This can be used to analyze the cost of invocations.

Chapter 5

Evaluation

Cppless was evaluated using a variety of benchmarks, mainly targeting the performance of the runtime implementation. This was done both using micro-benchmarks targeting the overhead of specific parts of the runtime and also using scientific benchmarks, which are closer to real-world applications.

The set of problems to which a serverless accelerator framework can be applied and performance improvements are to be expected is quite specific: First of all, the problems have to be massively parallelizable, while at the same time the algorithm shouldn't exhibit too much data movement as communication results in additional overhead. The benchmarks presented here are supposed to cover a diverse set of problem types, demonstrating different use cases for cppless.

We start with a recursive Fibonacci implementation that offloads invocations to the cloud recursively, i.e., the body of the serverless function invokes other serverless functions. Although not practical, this is an interesting example of how the framework can be used. Next, we will analyze the implementation of a solver for floorplan, an NP-hard optimization problem. Afterward, we will discuss the implementation of the classic Knapsack problem. We will also analyze the performance of an implementation of the N-Queens problem.

Next, we will analyze the implementation of a distributed CPU-raytracer and use that benchmark to provide an in-depth analysis of the impact of different parts of the runtime on the performance of the benchmark. Finally, we will discuss a pi-estimation algorithm, which is a good example of a problem that is well suited for parallelization.

The benchmark suite presented here is primarily based on the Barcelona OpenMP Task Suite (BOTS) as described in [2].

5.1 Benchmark Methodology

All benchmark data points presented here were produced on an isolated virtual machine on AWS. The benchmarks were run with the following settings: A `t3.medium` EC2 instance was used, which features 4 GiB of RAM and 2 VCPUs, furthermore, they feature a network interface with a 5 Gigabit connection. Both the EC2 instance and the deployed AWS Lambda functions are located in the `eu-central-1` region of AWS.

Each benchmark was compiled as an isolated binary, using the `O3` optimization setting of `clang`. Neither the compilation time nor the time that it takes to upload the AWS Lambda deployment packages to the cloud was taken into account. The resulting binary, once it runs, executes one iteration of the benchmark. The default AWS Lambda dispatcher configuration is used, which configures the memory limit of a Lambda function to be 1 GiB. An exception to this is the N-Queens benchmark where the memory limit is set to 2 GiB.

A command line tool called `hyperfine` [7] was used to measure the execution time of these binaries, and its JSON output option was used to get access to the individual measured times. Generally, 3 warmup runs were performed, which did not contribute to the measured times - this ensures that the disk pages containing the executable are loaded into the filesystem cache of the virtual machine. Furthermore, for the benchmarks using `cppless` it ensures that warm instances of the serverless function are created. Following these warmup runs, timed runs were performed until a statistically significant amount of timing data was collected. The time measured is ‘real time’ and not ‘CPU time’, when referring to ‘user time’ or ‘system time’ we will explicitly refer to this fact.

The benchmark JSON data was used for post-processing, which was done using several Python scripts. The raw data and the analysis results are available on the GitHub repository of this thesis.

5.2 Fibonacci

The Fibonacci benchmark aims to demonstrate the basics of recursive task invocations using `cppless`, but has little practical use. The benchmark program consists of a single task that recursively invokes itself twice, mimicking a naive recursive implementation of the classic Fibonacci problem. Once the recursive tasks are dispatched the function waits for both tasks to complete and then returns the sum of the two results.

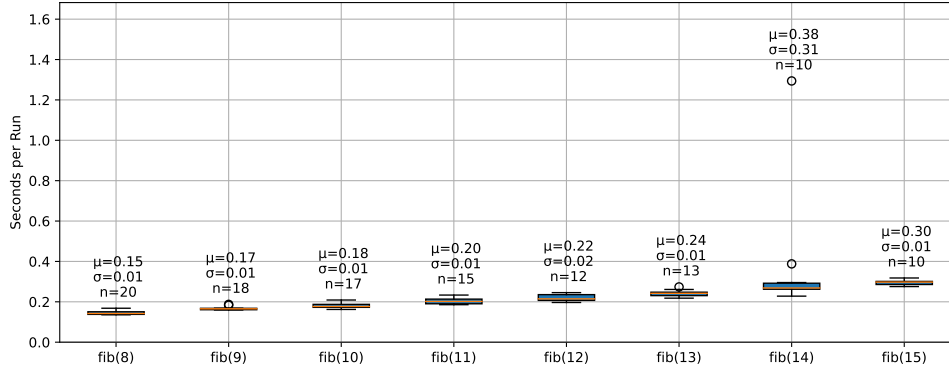


Figure 5.1: Execution times for different values of the parameter n for the Fibonacci benchmark.

The benchmark program was executed for different parameter values, and the results of the benchmark are represented in figure 5.1. As can be seen, the execution times roughly follow a linear trend, although the total computational complexity for this implementation follows the Fibonacci sequence. This shows one of the advantages of the framework: Due to the scalability of the serverless platform, it is possible to scale spontaneously to a large number of tasks, which up to a certain bound can lead to improved time complexity.

5.3 Floorplan

This benchmark, derived from the BOTS suite of parallel benchmarks, is concerned with the placement of boxes in a two-dimensional grid, minimizing the area of the bounding box.

The input data is a set of boxes that are to be placed in the grid. The single-threaded implementation is a simple brute-force algorithm that uses a branch and bound search to place the individual boxes: It starts with the first box, placing it in a corner and succeeds by placing the next box adjacent to one of the boxes which are already placed on the grid. Once all boxes are placed, the area of the bounding box is calculated. To speed up the computation, search tree pruning is used to avoid unnecessary computations: If the area of a partial solution is already larger than the area of the best complete solution found so far the algorithm aborts the search in the current branch and continues.

The cppless-based implementation calculates a list of prefixes on the host machine and offloads a task invocation for each prefix to AWS Lambda. Here, AWS Lambda instances with 1024 MiB of memory are used, and the available vCPU count scales linearly with this memory limit. The results

5. EVALUATION

of these invocations are awaited and the optimal result of the search is returned. The individual task invocations rely on search tree pruning in their respective subtree, but no communication is used to share intermediate results between the invocations.

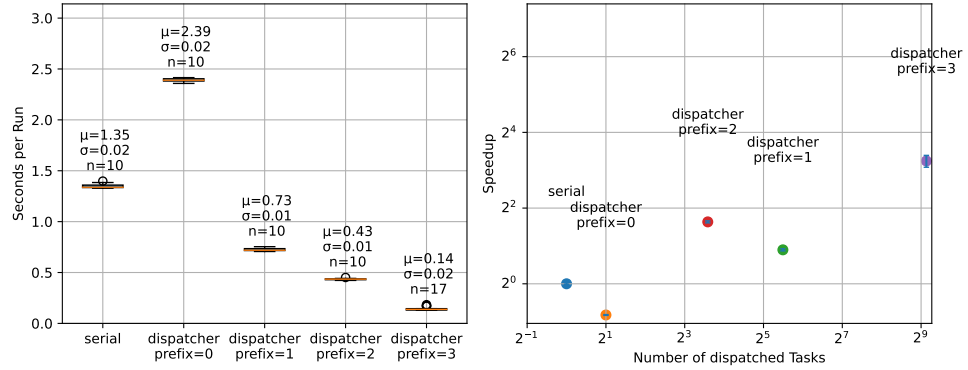


Figure 5.2: Execution times for different configurations of the floorplan benchmark for an input of size 15.

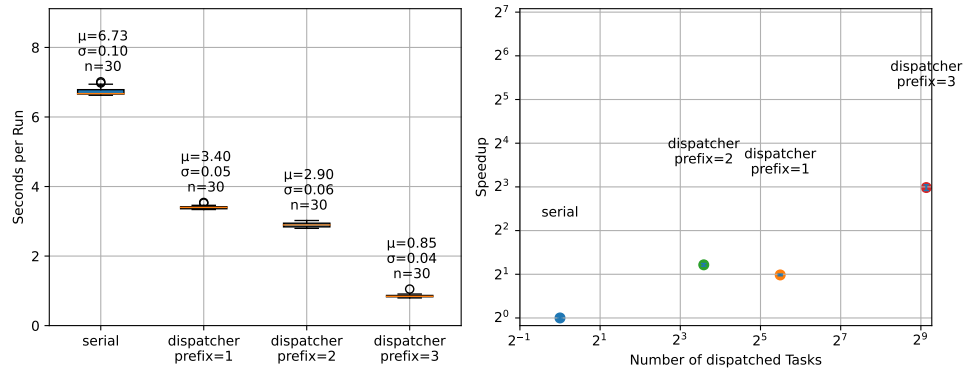


Figure 5.3: Execution times for different configurations of the floorplan benchmark for an input of size 15.

The benchmark program was executed for different input sizes, and the results of the benchmark are represented in figure 5.2 and 5.3. The number of tasks created is not monotonic in the specified prefix length as inviable solutions are already eliminated in the host program.

As can be seen, the configurations relying on cppless only achieve a speedup of around 8 in both cases, even when creating around 500 tasks that execute in parallel. This is to be expected, as the branch-and-bound implementation that was used here requires communication between different tasks to

eliminate search paths that are not promising. The original BOTS implementation relied on a shared variable between tasks which was used to store the best solution found so far. Cppless currently only supports the request-and-response model which makes it difficult to implement this kind of communication.

5.4 Knapsack

The knapsack problem is a classic optimization problem: Given a set of items, each with a weight and a value, the task is to determine which items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. The problem is NP-hard and requires little data to be transferred, therefore was deemed to be a good candidate for the framework.

We use a naive recursive implementation based on a backtracking approach for the serial version of the benchmark. The items are sorted by decreasing value to weight ratio. This allows for efficient search tree pruning by computing an upper bound on the optimal result in a certain subtree, thus making this a branch and bound algorithm.

For cppless we follow a similar approach for splitting the problem into smaller tasks as in the floorplan benchmark: We use a simple recursive implementation that uses creates task invocations at a certain recursion depth. Pruning is applied locally in each task invocation, but the different workers don't communicate with each other.

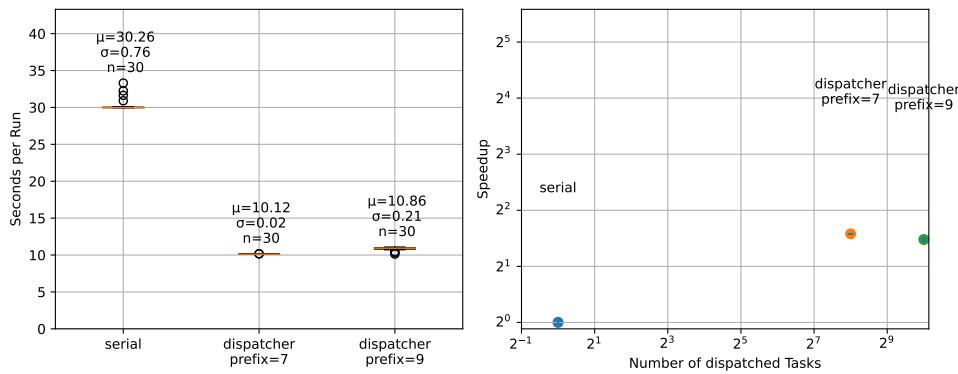


Figure 5.4: Benchmark results for the knapsack benchmark applied on input with 40 items.

The benchmark program was executed for different input sizes, and the results for an input of size 40 are represented in figure 5.4. As can be seen, the speedup that is achieved with the cppless-based implementation is in-

5. EVALUATION

significant. This can be attributed once again to the missing communication between the workers, resulting in fewer subtrees being eliminated early on.

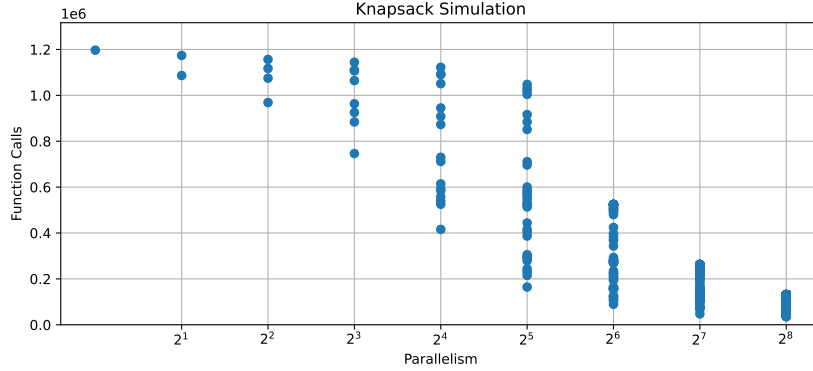


Figure 5.5: Function calls to a recursive `knapsack` function with pruning enabled when processing the knapsack item of size 24 from the bots suite.

This behavior was simulated using a Python script for certain cases, the results of which can be used to illustrate the problem as depicted in figure 5.5. Here, the number of function calls that occur in each subtask is plotted against the parallelism used. These function calls relate directly to the latency of the task invocations, thus we can use this to make statements about the performance of the serverless tasks. Specifically, we can conclude that although the average work per worker is reduced when the amount of parallelism is increased, the overall speedup is still limited by the longest-running task. For the problem size used here, the maximum achievable speedup using this algorithm only begins to get significant after around 64 tasks are executed in parallel. Even then, the speedup is still limited to a factor of around 2, leading to subpar performance.

We can thus conclude from this benchmark that although this benchmark seemed like a possible candidate for the framework, it is not. The reason here is that the benefits of early pruning overweight the benefits of parallelism, making parallelism without communication ineffective.

5.5 N-Queens

The N-Queens problem is concerned with placing N queens on an NxN chessboard, specifically, we are interested in the number of different solutions for a given problem size N. The problem is NP-hard, meaning that there are no known polynomial time algorithms for solving it. This variant of the N-Queens algorithm is part of the BOTS benchmark suite, however here we are using a modified implementation resulting in a considerable per-

formance improvement. N-Queens is typically implemented using a backtracking approach where the queens are placed incrementally row by row, once a queen can no longer be placed in a given row, the algorithm backtracks and tries to place the previous queens at different positions.

Sequentially, this can be done by representing the board as an array of length N , where each element stores an integer representing in which column the queen is placed at that row. The algorithm then has to look at the different indices of the array to determine the positions where the next queen can be placed. In practice, this data representation is thus rather inefficient, as determining where a queen can be placed requires a linear search over the entire array. Instead, we want to make sure that the locations where a queen cannot be placed are available in a performant way, which makes it easy to find the location where a queen can indeed be placed. Richards introduced an efficient backtracking algorithm in [8] which elegantly solves the problem using bit patterns. Instead of denoting the positions of the queens, we represent the current state of the board using three different bitmasks: One for tracking vertical collisions and one for each of the two diagonal directions, i.e. one for the upper left direction and one for the upper right direction. A valid solution is found once all bits in each bitmask are set.¹

The core of the serial implementation is the following:

```

1  using u64 = unsigned long;
2
3  template <unsigned char n>
4  void nqueens(u64 &res, u64 min_diag, u64 maj_diag, u64 vertical) {
5      constexpr u64 mask = (1 << n) - 1;
6
7      if (vertical == mask) {
8          res++;
9          return;
10     }
11     u64 bitmap = mask & ~(min_diag | maj_diag | vertical);
12     while (bitmap) {
13         u64 bit = (~bitmap + 1) & bitmap;
14         bitmap ^= bit;
15         u64 new_min_diag = (bit | min_diag) >> 1;
16         u64 new_maj_diag = (bit | maj_diag) << 1;
17         u64 new_vertical = bit | vertical;
18         nqueens<n>(res, new_min_diag, new_maj_diag, new_vertical);
19     }
20 }
```

This recursive version of the algorithm first checks if the current state is a

¹There exist optimized implementations which do not rely on recursion but manage a fixed-size stack directly. However, no appropriately licensed version was found.

5. EVALUATION

valid solution, if that is the case it increments the result counter, otherwise, it recursively tries to place a queen in each of the available positions. The bitmap is used to track the remaining valid solutions, that is, the positions which are not out of bounds (mask) and which do not collide with any other queen. We extract the highest bit in the bitmap and use it to place a queen at that position. The bitmap is then updated to remove the bit which was used to place the queen. A recursive call is performed with the updated masks.

To parallelize the algorithm we create tasks where the location of the first p queens is fixed. We call this a prefix. This method of splitting up an instance of the N-queens problems into subtasks is commonly used in high-performance computing as seen in [5]. These prefixes are computed using a serial recursive implementation, similar to the one above. For each of these prefixes, the resulting masks are computed.

Cppless is utilized by locally computing prefixes up to a specific depth and creating task invocations from each of these prefixes. These invocations are offloaded to the cloud where the serial implementation is executed to determine the number of possible solutions in the part of the search tree the task is responsible for. The partial results are then accumulated on the host machine. The length of the prefix determines the number of tasks that are offloaded and thus the parallelism that is achieved. The overhead of calculating the prefixes is linear to the number of tasks that are offloaded, making this problem embarrassingly parallel.

The same approach is used for the multi-threaded implementation, here the prefix is chosen such that at least a task is created for each thread of the machine. For the t3.medium instance, this is equivalent to using a prefix length of 1.

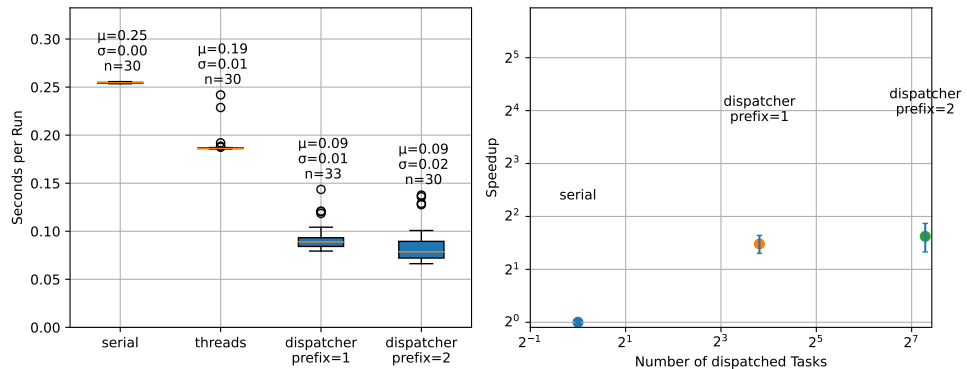


Figure 5.6: N-queens benchmark results, computing nqueens(14)

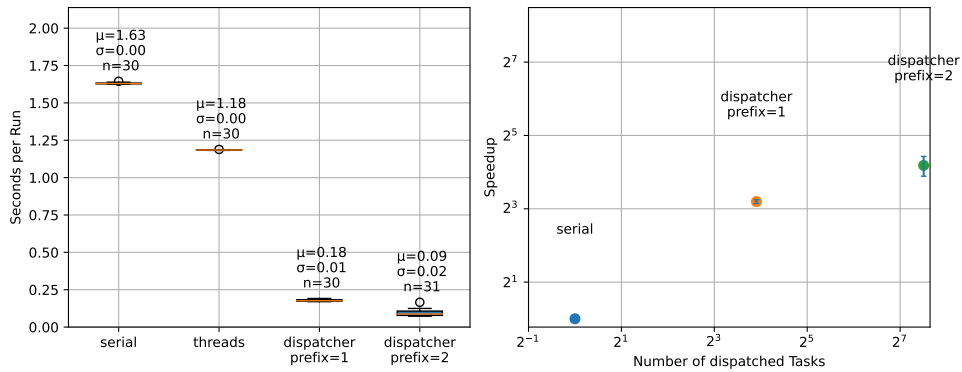


Figure 5.7: N-queens benchmark results, computing nqueens(15)

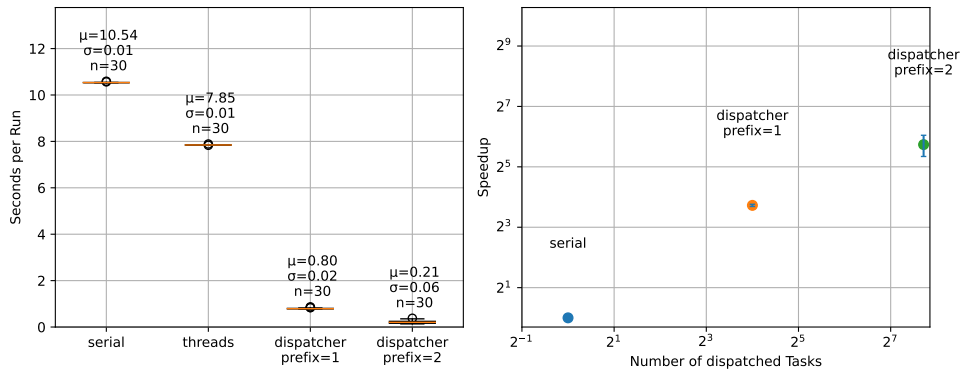


Figure 5.8: N-queens benchmark results, computing nqueens(16)

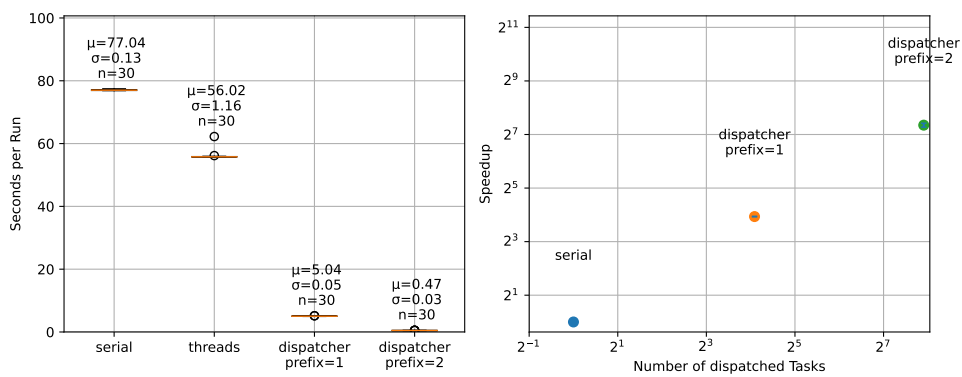


Figure 5.9: N-queens benchmark results, computing nqueens(17)

5. EVALUATION

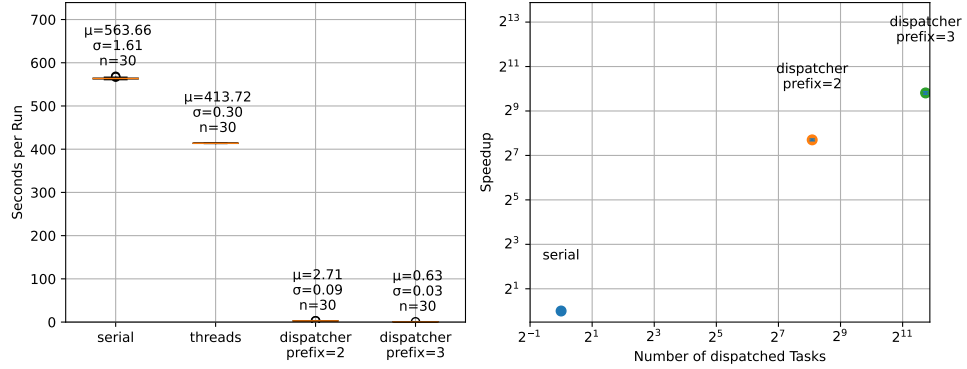


Figure 5.10: N-queens benchmark results, computing nqueens(18)

The results for the N-queens benchmark are depicted in figure 5.6, 5.7, 5.8, 5.9 and 5.10. We can see that the speedup is indeed significant, but isn't always in the order of magnitude of the parallelism used to achieve the speed. We can see that the speedup achieved per dispatched task decreases significantly with increased parallelism. This can be attributed to the initial connection latency, which stays significant, especially for larger values of p . The initial connection latency also explains the poor speedup achieved for nqueens(14).

This method of parallelizing the algorithm is quite effective, but it can be observed that there is some variance in the workload that each task is responsible for, this fact is also demonstrated in [5] in figure 3. The execution time depicted in the benchmark results is limited by the longest-running task and the overhead in the host machine. In this case, there's little serialization overhead and the suboptimal speedup can mostly be attributed to the fact that the longest-running task has to be awaited.

However, the heterogeneous task workload should not be interpreted as a downside of the approach. Due to the pay-as-you-go pricing model, the cost of running a task is directly proportional to the amount of work that the task has to do. Thus, waiting for the longest-running task to finish does not incur any additional cost, thus there is no need to combine small tasks to have fewer tasks in total.

5.6 CPU-Raytracer

This benchmark implements a basic Monte-Carlo raytracer utilizing the CPU. The implementation is derived from the 'Ray Tracing in One Weekend' book series (see [10]). In addition to that, the bounding volume hierarchy mechanism was implemented as outlined in [11]. Modifications are implemented

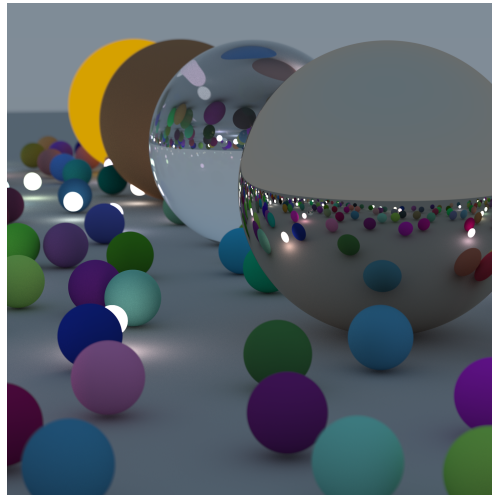


Figure 5.11: CPU-Raytracer rendered image

which increase the usage of AVX2 vector instructions as supported by the Haswell architecture². The benchmark generates a random scene³ and then renders it using a renderer backend implementation depending on a command line flag. An example of a generated image is shown in figure 5.11.

Currently, three different backends are supported: A single-threaded implementation, a multi-threaded implementation using worker threads and a cppless-backed implementation utilizing AWS lambda to offload the rendering to the cloud.

The multi-threaded renderer and the cppless-backed renderer both use tiles to split the rendering into smaller pieces. The viewport is divided into a grid of equally sized tiles. For the multi-threaded renderer, a dynamic number of worker threads is created⁴ and shared memory is used to allocate tiles to a worker thread, thus resulting in less overhead than spawning a new thread for each tile. This way of splitting up the work also optimizes the memory access pattern as the individual workers tend to stay in one region of the image while rendering a tile, thus it is more likely that the objects are already loaded in the cache.

To offload work using cppless the bounding volume hierarchy is computed once on the host machine and the image is split up into tiles. A task is created for each tile, and the tile location along with the bounding volume hierarchy and the rendering settings is passed to the cloud using captures

²For x86_64 Lambda function AWS guarantees that the AVX2 ISA extension is available: <https://docs.aws.amazon.com/lambda/latest/dg/runtimes-avx2.html>

³A seeded random generation process was used to ensure consistency between runs.

⁴8 worker threads were used for the EC2 instance.

5. EVALUATION

of a lambda function. Thus the bounding volume hierarchy along with the objects containers and the associated materials has to be serializable using the cereal library. Due to the inheritance structure of the object and material classes, the support of cereal for polymorphic types is used. This allows the entire scene to be efficiently serialized for the task invocation. The result of each task invocation is a small image, which contributes to the final image.

Cppless is especially advantageous here due to the heterogeneous nature of the individual tasks: The computation time required for a task heavily depends on the objects that are contained in the tile. As the scheduling of the tasks is left to the cloud provider the programmer doesn't have to worry about efficient, equal work distribution. Furthermore, the benchmark can make use of the massive parallelism provided by the cloud environment.

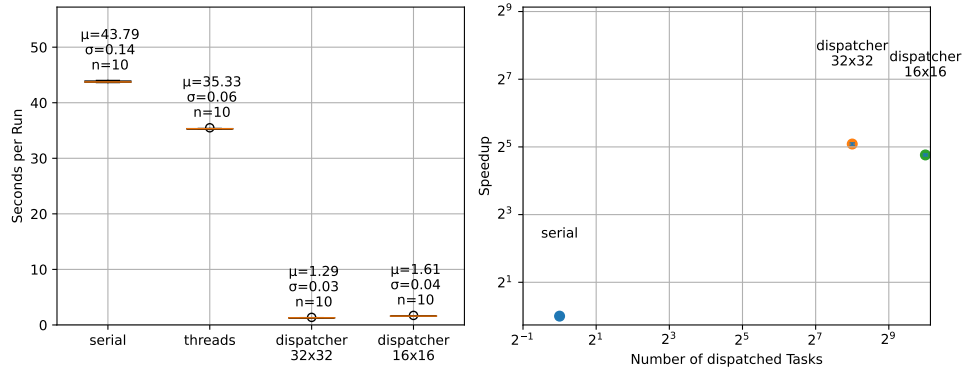


Figure 5.12: CPU-Raytracer benchmark results, rendering a 500x500 image

The benchmark results are illustrated in figure 5.12. Once again the speedup is limited by the longest-running task, which determines the achievable speedup. Analysis of the individual tasks showed that the mean task workload scales almost perfectly with the tile size. On the other hand, the maximum workload size only decreases by around 40% whenever the tile size is halved. This can be attributed to heterogeneous per-pixel workloads, resulting in the tiling process not splitting up the work evenly.

5.6.1 Overhead analysis

The raytracing benchmark was used to further analyze the overhead of the framework, both in terms of the cost of the computation, but also the performance overhead that the framework introduces. The cost overhead was estimated by running a scene configuration using different tile sizes which directly correlates with the amount of parallelism that is used. This benchmark is especially suited to analyze further as the amount of parallelism

that is used can be varied to arbitrary values. We executed the benchmark for various tile sizes resulting in 4 to 2048 lambda functions being used concurrently. The cost of the computation was analyzed using the `aws_trace` tool, which outputs the billed duration as determined by AWS. Enabling tracing support when running a cppless program slightly increases its overhead, thus different runs were analyzed to keep the timing data as accurate as possible.

The first metric that was analyzed is the total number of memory-GB seconds as billed by AWS Lambda. The amount of memory-GB seconds directly correlates with the cost, generally, there is a fixed multiplier, although discount rates are applied at some point. Each benchmark configuration was run 16 times, afterwards, the trace files were analyzed using the `aws_trace` tool to determine the cost of individual runs.

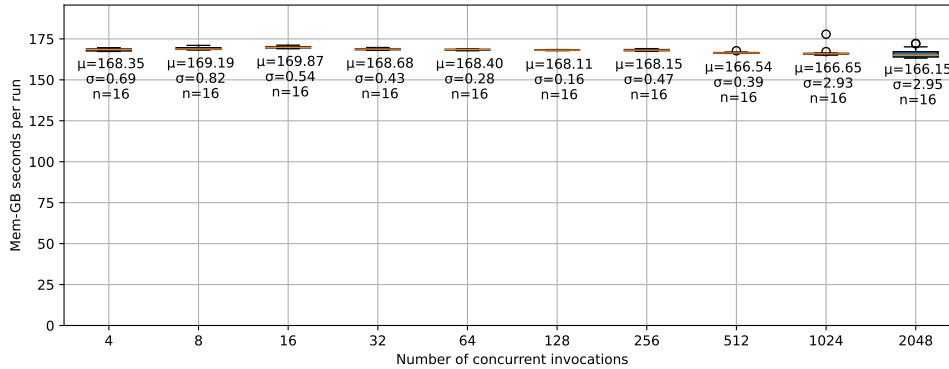


Figure 5.13: Cost dependence on the number of parallel tasks

As can be seen in figure 5.13, the cost of the computation is barely affected by the number of parallel tasks. Although the result is available more quickly, the cost stays almost constant as long as the number of tasks stays within reasonable bounds. Specifically, for this benchmark, the data showed that the task duration for the smallest tile size varied between 8ms and 150ms. Even with invocations in the milliseconds range the cost of the computation is dominated by the productive work that is done in the function invocations. This benchmark result shows that the overhead on the side of the AWS lambda function is relatively small.

In this example, however, the serial work that has to be done to split up the work is insignificant, dividing the work into tiles doesn't add much overhead. However, the host machine is also required to handle serialization and network communication, which adds additional cost.

We conclude that the main factor limiting the latency for this benchmark is

5. EVALUATION

the host platform which is responsible for offloading the work to the cloud. The host machine has to serialize the data, resulting in around 88 KiB for each invocation and transferring it to the AWS lambda API through their HTTP API. The result takes up another 32 MiB of memory, due to the high dynamic range format which is only normalized on the host machine.

Further analysis of the traces showed that for a tile size of 8x16 (resulting in 2048 tasks) around 40% of the time was spent on serialization of the BVH tree and the objects. Furthermore, around 2% of the time was spent computing the computation of the BVH tree and another 2% of the time was spent deserializing the response messages. The rest of the time on the host machine was spent on network communication, leaving around 0.91 milliseconds for networking per invocation.

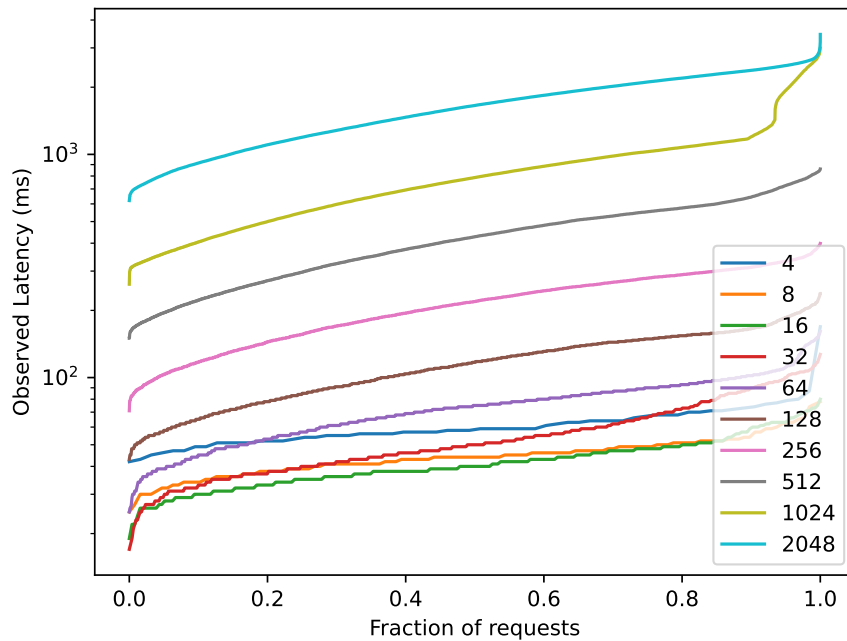


Figure 5.14: The latency overhead distribution of all invocations when a certain level of parallelism is used. The latency overhead times were sorted in each group and plotted against the number of invocations.

As the serialization of the data takes up the CPU core of the host machine, the latency overhead⁵ of the benchmark program increases as the number

⁵The latency overhead is the duration between the invocation of a task and its result, excluding the actual computation time as billed by AWS

of parallel tasks increases. The latency overhead distribution and its dependence on the parallelism in use are shown in figure 5.14. The observed latency decreases until a certain point as the overhead of the initial connection establishment dominates the latency if only a few tasks are dispatched. At some point, the result of the tasks can't be processed quickly enough anymore and the latency increases again. This explains why the speedup doesn't increase anymore at some point as the host is busy with deserializing and merging the results.

5.6.2 Serialization

Serialization results in a major overhead in this case, possibly due to the recursive layout of the bounding volume hierarchy. Furthermore, the bounding volume hierarchy stays the same for all invocations but is serialized for each invocation individually. Currently, cppless only supports serializing the entire task together with its arguments, meaning that it cannot easily reuse the results of previously serialized tasks. There are however manual optimization techniques that can solve this issue partially. Specifically, we can manually serialize the BVH tree into a buffer with contiguous memory, thus making it much easier to use that serialized result. This method also requires manual deserialization in the task body:

```
1  auto bvh_serialized = cppless::binary_archive::serialize(bvh_root);
2  // In the task body:
3  bvh_node world;
4  cppless::binary_archive::deserialize(bvh_serialized, world);
```

This change results in an end-to-end performance increase of around 7% by reducing the 'user time'⁶ from 1.56 s to 1.10 s.

5.7 Pi-Estimation Benchmark

The pi-estimation benchmark uses a classic Monte-Carlo implementation to estimate the value of pi. To get a rough estimate a Mersenne Twister pseudo-random number generator is used to generate random points in a unit square. The number of points that fall within the unit circle is counted and the ratio of points in the circle to the total number of points is used to estimate pi. The benchmark is executed for various numbers of parallel tasks, ranging from 1 to 2048.

⁶'user time' is referring to the amount of time a process spent in user mode, the time during which the process is blocked or is waiting on a system call is not included.

5. EVALUATION

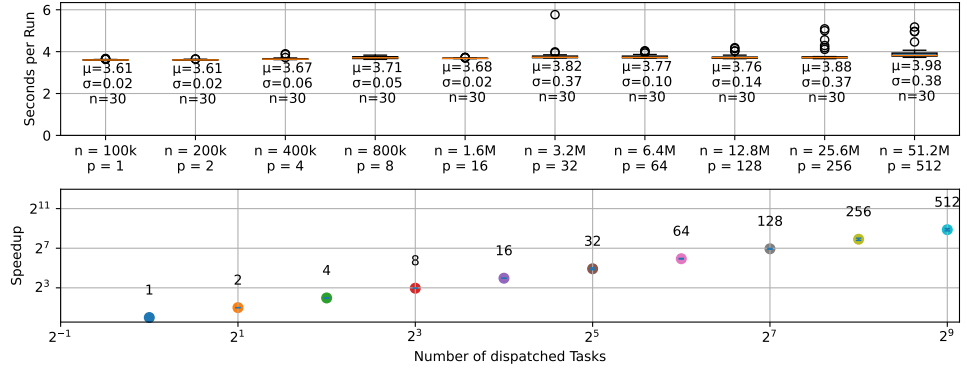


Figure 5.15: Pi-Estimation benchmark results, n specifies the number of total iterations, p specifies the number of tasks that are dispatcher.

The benchmark was parallelized with cppless using the dispatcher API. The tasks individually calculate an estimate of pi, the different values from the tasks are then combined by calculating the average. The results of the benchmark are shown in figure 5.15. The results show that near-perfect scaling is achieved. Furthermore, the results also show that the overhead on the side of the host machine is negligible as very little serialization has to be performed.

5.7.1 Overhead analysis

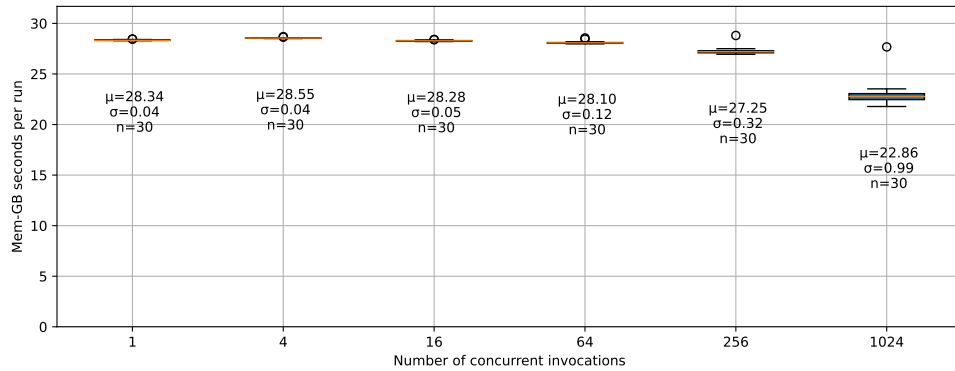


Figure 5.16: The Memory-GB seconds as billed by AWS Lambda for the pi-estimation benchmark.

The dependence of the total cost of the computation on the number of parallel tasks is shown in figure 5.16. The cost of the computation is dominated by the productive work that is done in the function invocations. The over-

head on the side of the AWS lambda function is relatively small. We can furthermore see that the cost slightly decreases as the number of parallel tasks increases. This could be explained by AWS calculating the billed time inaccurately for very short invocations, favoring the customer in this case.

These results show that cppless adds very little overhead both on the side of the host system and on the side of the cloud provider. The overhead that can be seen in the benchmark results is mostly a result of inefficient serialization and suboptimal speedup can be attributed to task workload heterogeneity and algorithmic constraints.

5.8 Micro-Benchmarks

We used Micro-Benchmarks to analyze the performance of two essential parts of the runtime part of the framework: We evaluate the performance of different serialization methods and measure the performance of the AWS lambda client that the dispatcher uses to invoke tasks.

5.8.1 Serialization

As mentioned above, serialization can become a major overhead in certain cases when a lot of data has to be serialized and deserialized. Furthermore, certain serverless environments like AWS put additional constraints on the data that can be transferred, making some serialization techniques unusable. In the serialization microbenchmark, we analyzed the additional overhead that comes along with these constraints. Specifically, AWS requires the data that is provided to an invocation of an AWS Lambda function to be valid JSON. With this constraint two different archive formats are implemented: One based on the JSON archive of cereal and a second one, based on the binary archive of cereal, the output of that archive most likely isn't valid JSON, thus base64-encoding is used to encode the data and delimiters are added to the data to make it valid JSON.

As a baseline, we compare both results against the plain binary serialization of cereal. The benchmark data presented here is the result of running a custom benchmark library, that uses the monotonic clock to measure the time spent in a function. We ensure that the result is not optimized away by issuing a volatile write. Both the decoding and the encoding of data were measured separately.

Array Serialization This benchmark measures the time spent in the serialization of an array of unsigned integers of 64 bits. An `std::vector` with 1000000 elements was repeatedly serialized and deserialized using the different serialization methods. One run of the benchmark represents one serialization/deserialization round.

5. EVALUATION

	Time per Run		Throughput
	<i>ms</i>	σ	GiB/s
binary			
Encode	5.90	0.597	1.32
Decode	3.18	0.315	2.46
binary_json			
Encode	13.03	0.162	0.60
Decode	28.63	0.192	0.27
structured_json			
Encode	462.40	6.806	0.02
Decode	144.15	1.408	0.05

Figure 5.17: Benchmark results for serializing an array of unsigned integers.

The results as shown in 5.17 show that the binary archive of cereal is much faster than the plain binary archive of cereal. The overhead of the `binary_json` format is slightly higher and can be attributed to the base64 encoding that has to be executed. On the other hand, the JSON archive performs much worse than the other two options. The JSON archive has to do much more work: Find the correct key to decode, transform the basis of numbers, dynamically allocate memory, and so on. Thus it is much slower than the binary archive. For these reasons, the `binary_json` format was used as the default format for the AWS Lambda dispatcher implementation.

The `std::vector` takes up around 8 MiB of memory, which means that the binary archive can achieve a serialization throughput of around 1.324 GiB/s and a deserialization throughput of around 2.458 GiB/s. The `binary-JSON` archive achieves a serialization throughput of around 0.600 GiB/s and a deserialization throughput of around 0.272 GiB/s. The JSON archive achieves a serialization throughput of around 0.017 GiB/s and a deserialization throughput of around 0.054 GiB/s.

Struct Serialization In this benchmark, instead of serializing an `std::vector` of unsigned integers, each element of the vector contains a struct with two signed integers and a string. The struct has a custom serialization method. The data that is serialized is similar to the first benchmark: We vary the values for the integers, but the string content is constant, but as an `std::string` instance, the memory for the string is allocated elsewhere.

	Time per Run		Throughput
	<i>ms</i>	σ	GiB/s
binary			
Encode	97.35	0.280	0.3061
Decode	81.48	0.191	0.3658
binary_json			
Encode	131.62	11.150	0.2264
Decode	158.62	0.412	0.1879
structured_json			
Encode	726.51	3.493	0.0410
Decode	650.27	3.000	0.0458

Figure 5.18: Benchmark results for serializing an array of structs.

As can be seen in 5.18, the serialization of the struct performs much worse compared to the vector of integers. This can be explained by the fact that the struct contains a string, which implies worse cache performance. Furthermore, the first benchmark benefits from a fast path in cereal: Vectors of arithmetic types are directly serialized as a binary blob. For the cereal binary archive, this means that the data pointer of the vector is used as the input to an `sputn` operation on the underlying stream. This results in a near zero-overhead serialization. As the JSON-based archive can't take advantage of this optimization it already had to do the same work in the first benchmark. This explains why the difference between the binary archive and the JSON archive isn't that exaggerated compared to the vector of integers.

5.8.2 AWS Lambda Client

In this microbenchmark, we analyzed the end-to-end latency of the custom AWS Lambda client that is used for all benchmarks presented here, the only exception being the Fibonacci benchmark⁷. The main goal of the custom client is to support a lot of parallel executions, furthermore, we wanted to avoid the use of a separate operating system thread for each task invocation, thus the client makes use of boost ASIO's operating system abstraction. We evaluated the latency for different numbers of invocations. In the configuration used for the benchmark, the same pool of 16 HTTP/2 connections that the dispatcher makes use of is used. Each of these HTTP/2 connections is limited to 100 concurrent, active requests, thus resulting in a total of 1600 possible concurrent invocations.

⁷The HTTP1 client is used for the Fibonacci benchmark, as the HTTP2 client is not supported for recursive function calls.

5. EVALUATION

The client program is targeting an AWS lambda function hosted on an account with an unreserved account concurrency of 1000. The function itself, written in Node-JS, is a simple function with a constant output that the client verifies. This limits the overhead introduced on the side of the AWS Lambda service to the minimum.

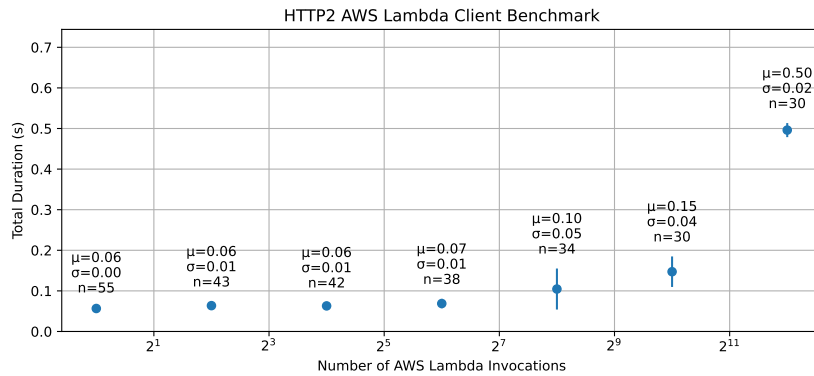


Figure 5.19: Latency of the AWS Lambda client for different numbers of invocations.

As depicted in 5.19, the latency for 1 invocation starts at around 50ms and increases linearly to around 150ms until the limit of the client itself is reached. At this point, invocations are not dispatched directly anymore, but instead, have to wait until the response of other invocations arrives. This results in a significant increase in latency.

We conclude, that the client can theoretically dispatch invocations at a rate of around 10 invocations per millisecond, with an initial latency of around 50 milliseconds, in its current configuration. However, restrictions of the AWS account that the client dispatches to limit the number of concurrent invocations, resulting in a considerable slowdown at some point. In the optimal case around 1000 to 2000 tasks should be created concurrently, resulting in a throughput of around 6 tasks per millisecond. For decently sized tasks, this eliminates the client implementation as a bottleneck.

More concurrent invocations could be dispatched by increasing the size of the HTTP/2 connection pool, but this would require manual adjustments by the user. Furthermore, the request limit for individual connections could be increased, but this results in more situations where head-of-line blocking becomes a problem.

Chapter 6

Discussion

The novel programming model presented in the thesis allows users transparently make use of serverless environments. Although there are some unique restrictions, we believe that the generic implementation through alternative entry points provides a decent trade-off between flexibility and simplicity. The low-level dispatcher interface provided a sufficient abstraction layer for most benchmarks, but might not be well-suited for more complicated scenarios. The graph interface on the other hand has a certain threshold after which it becomes useful, which is why we chose not to use it for most benchmarks.

The benchmark results suggest that there is a class of algorithms for which `cppless` indeed is advantageous. On the other hand, it also shows the need for more diverse communication methods as several benchmarks could benefit from point-to-point communication allowing for a more efficient implementation.

6.1 Further Work

In the current state, we see two main problems with `cppless`: Its inability to support something we call ‘partial serialization’ and its missing support for advanced communication patterns.

6.1.1 Partial Serialization

Currently, `cppless` serializes all data at once when the task is dispatched using an instance of the dispatcher class. This is not a problem when the data that is to be serialized is small or when only a few tasks are dispatched, but it can become a problem when the data is large, especially when the task is invoked a lot. In a lot of scenarios the task invocations, if created from the same task, will share the same data, which would allow the data to

be serialized only once and shared between all the task invocations. However, simply serializing the data when the task is created doesn't solve the problem completely either because there might small amounts of data that are not shared between the task invocations. For example, a task invocation might include a dataset itself and an index into the data set. The resulting task invocations are different, thus the serialized data could not be shared between the task invocations. What we propose would be a container class using which data can be wrapped - when constructed the wrapper would serialize the underlying data and store the serialized data in a buffer. When the task is invoked, the task would use the data that is already serialized.

However, the interface would have to ensure that the data is also only deserialized once, meaning that its state, serialized or not, is saved internally. This is not possible in the current implementation.

A utility class that implements this interface would allow for reducing the overhead that is required to serialize the data. Different nested loops could reuse the same serialized object, allowing a transparent interface for partial serialization.

6.1.2 Advanced Communication Patterns

Cppless currently supports only a naive request-response communication pattern. To speed up a variety of use-cases communication between tasks is required. To this end, we propose the introduction of a channel-based communication pattern. A channel, as made popular by the Go programming language, is a shared object allowing different threads to communicate with each other by sending and receiving messages. Channels could be created with different underlying message types that are serialized and deserialized transparently.

The implementation of channels could be platform dependent. For example, a channel could be implemented using a cloud provider product, but implementations relying on a central serverless function acting as an orchestrator could be used as well.

Channels would be added as a low-level interface that can be used to implement more advanced communication patterns. For example, shared variables using a merge operation could be implemented using a channel.

6.1.3 Detached Execution

The dispatcher interface forces the user to wait for a task to complete, keeping the current execution unit active, even for recursive implementations. In some use cases, however, it might make sense to allow the task invocation to continue execution on its own, especially when it can communicate

with other task invocations using channels. Specifically, this would entail extending the dispatcher interface to support the concept of detached execution. Detached task invocations cannot be waited for, but should execute independently from their caller.

6.1.4 Advanced high-level API

These primitives would allow for a much richer high-level API. For example, task graph executions could run independently, without the need for a host machine orchestrating the execution. We envision that an independent task graph executor could allow more data to be transported between tasks as the host machine would not be required to serialize the data, thus reducing the dependence on the host machine. This would allow more algorithms to be implemented efficiently using cppless.

Chapter 7

Conclusion

In this thesis, we have proposed a novel way of defining serverless functions in the C++ programming language and present a user-space library that facilitates this new programming model. Although the concept of defining serverless functions in the same source file as the code using it, is not new, C++ has additional constraints that had to be dealt with. Cppless allows users to elegantly define programs that offload work transparently to a serverless service, placing only minimal restrictions on the serverless platform. This makes it a viable alternative to the current state of serverless programming in the C++ programming language, lowering the barrier to entry for serverless programming.

The compilation pipeline is well-designed and allows users to specify a set of serverless function definitions at compile time which are then compiled and deployed independently for a serverless platform. The meta-programming features of C++ allow a majority of the work that had to be done at runtime in the previous work to be done at compile time. This allows for efficient offloading and reduced the serialization overhead as only the task parameters, but not the code itself has to be transferred at runtime.

The pipeline seamlessly integrates into the regular C++ compilation process and makes programs that make use of serverless function for acceleration composable: Libraries can expose templated functions that instantiate tasks which are then automatically compiled for the targeted platform. This allows high-level libraries to build on top of cppless. Furthermore, the compilation pipeline is designed to be extensible, allowing users to add support for new serverless platforms by implementing a few simple interfaces.

The benchmark results showed that different types of algorithms can make use of cppless with different degrees of effectiveness: For embarrassingly parallel programs that require little communication, cppless achieves near-perfect scaling and only adds very little overhead both to the host and to the

7. CONCLUSION

serverless worker. At the same time, we showed that even when the speedup is limited by the host process or uneven workload distribution, cppless still is a cost-effective solution.

We admit that the request-response communication model that cppless currently enforces requires careful planning to minimize the overhead. Most of these limitations can be addressed by extending upon the current model, necessary changes were outlined in the previous chapter.

Bibliography

- [1] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference, Middleware '19*, pages 41–54, New York, NY, USA, 2019. Association for Computing Machinery.
- [2] Alej Duran, Ro, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp, 2009.
- [3] Vinnie Falco. Beast: C++ http and websocket built on boost.asio. <https://github.com/boostorg/beast>, 2016.
- [4] W. Shane Grant and Randolph Voorhies. cereal - a c++11 library for serialization. <http://uscilab.github.io/cereal/>, 2017.
- [5] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. Solving the 24-queens problem using mpi on a pc cluster. *Graduate School of Information Systems, The University of Electro-Communications, Tech. Rep*, 2004.
- [6] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.
- [7] David Peter. Hyperfine: A command-line benchmarking tool. <https://github.com/sharkdp/hyperfine>, 2022.
- [8] Martin Richards. Backtracking algorithms in mcpl using bit patterns and recursion. Technical report, Citeseer, 1997.

- [9] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. Toward multicloud access transparency in serverless computing. *IEEE Software*, 38(1):68–74, 2021.
- [10] Peter Shirley. Ray tracing in one weekend, December 2020.
- [11] Peter Shirley. Ray tracing: The next week, December 2020.
- [12] Tatsuhiko Tsujikawa. nghttp2 - http/2 c library. <https://github.com/nghttp2/nghttp2>, 2013.
- [13] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, pages 328–343, New York, NY, USA, 2020. Association for Computing Machinery.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Cppless: A single-source programming model for high-performance
serverless

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Müller

First name(s):

Wlas

With my signature I confirm that

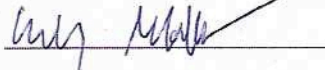
- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 08.09.2022

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.