Towards a Serverless Bioinformatics Cyberinfrastructure Pipeline

Shunyu (David) Yao, Muhammad Ali Gulzar, Liqing Zhang, Ali R. Butt
{shunyu,gulzar,lqzhang,butta}@cs.vt.edu

ABSTRACT

Function-as-a-Service (FaaS) and the serverless computing model offer a powerful abstraction for supporting large-scale applications in the cloud. A major hurdle in this context is that it is non-trivial to transform an application, even an already containerized one, to a FaaS implementation. In this paper, we take the first step towards supporting easier and efficient application transformation to FaaS. We present a systematic scheme to transform applications written in Python into a set of functions that can then be automatically deployed atop platforms such as AWS Lamda. We target a Bioinformatics cyberinfrastructure pipeline, CIWARS, that provides waste-water analysis for the identification of antibiotic-resistant bacteria and viruses such as SARS-CoV-2. Based on our experience with enabling FaaS-based CIWARS, we develop a methodology that would help the conversion of other similar applications to the FaaS model. Our evaluation shows that our approach can correctly transform CIWARS to FaaS, and the new FaaS-based CIWARS incurs only negligible ($\leq$ 2%) overhead for representative workloads.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Serverless Computing; Function-as-a-Service; Program Decomposition

ACM Reference Format:
Shunyu (David) Yao, Muhammad Ali Gulzar, Liqing Zhang, Ali R. Butt. 2021. Towards a Serverless Bioinformatics Cyberinfrastructure Pipeline. In *Proceedings of the 1st Workshop on High Performance Serverless Computing (HiPS '21), June 25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3452413.3464787

1 INTRODUCTION

The serverless computing, or Function-as-a-Service (FaaS), model is popularized by its desirable ease-of-use and its ability to scale the deployments in a cost-effective on-demand fashion. The model reduces the time and cost to migrate applications across deployments, and relieves developers from the cumbersome task of provisioning server resources for hosting their applications. As a result, the market share of serverless computing is projected to grow significantly in the near future [9, 13, 17].

The model holds promise for scientific workflows and applications as well [10], especially as innovative tasks such as deep learning and dynamic data analysis are incorporated into the applications. However, there is a fundamental disconnect between the underlying assumptions, e.g., containerized or virtualized resources, etc. in the design of existing applications and the abstractions supported by the serverless model. From the resource-provider (platform) point of view, traditional applications are typically stateful, resource-intensive, and run for long periods of time. In contrast, the serverless model creates ephemeral, stateless instances of pieces of code that are strung together to create applications. From the application developer's point of view, their existing applications need to be decomposed into lightweight functions that can be run atop the serverless computing substrate. This yields complex application decomposition and resource management challenges for the developers and the resource providers, respectively. There has been work in converting some applications to the new serverless model [6, 11, 13, 27], but such works have typically focused on particular components only. What is needed is a set of techniques and tools that can help developers efficiently decompose their applications and automatically create serverless implementations ready for deployment.

In this paper, we take the first steps in designing a systematic approach for complete application decomposition into functions for serverless computing. Our focus application is a bioinformatics workflow pipeline, Cyberinfrastructure for Waterborne Antibiotic Resistance Risk Surveillance (CIWARS), that identifies antibiotic-resistant bacteria in wastewater. For each analyzed wastewater sample, the application assembles and annotates genetic sequences therein, and then scores the genomes for anomalous behavior. The pipeline is a large monolithic application with a number of components (data and compute-intensive) [7, 8, 22] integrated into a rigid and fixed workflow. While the application is generally executed end-to-end, users may stop the pipeline at arbitrary points between components, analyze the output, and tweak the information before forwarding it to the next component. This interruption and need to examine intermediate data makes CIWARS well-suited for a FaaS implementation.

While we are starting with CIWARS to demonstrate the efficacy of our approach, our proposed approach will be applicable to a set of similar scientific workflows. There are two reasons for this. First, many workflows are essentially represented by directed acyclic graphs (DAGs) of components as in CIWARS, and thus amenable to a FaaS implementation should the components be transformed to handle the mismatch discussed above. Second, all applications can

HiPS '21, June 25, 2021, Virtual Event, Sweden

ACM ISBN 978-1-4503-8388-2/21/06…$15.00
https://doi.org/10.1145/3452413.3464787

benefit from having the agility of light-weight FaaS functions dynamically allocated to large computing substrates. A FaaS instance can perhaps be used to fill small unused resources in the popular FCFS with back-filling schedulers, letting traditional HPC and new serverless applications co-exist, and thus provide higher efficiency for HPC resources.

We build a systematic approach to enable fully functional decomposition. For starters, we manually split CIWARS's components into individual functions, and containerize them using Docker [20]. Doing so also makes the pipeline flexible as users can now pick and choose components that should be assembled into the application pipeline, e.g., by replacing one kind of anomalous behavior detector with another, and provide customized inputs and sample any communication between components for intermediate analysis. Once the pipeline of functions is identified, it is instantiated automatically to create the end-to-end application instance. Through the decomposition of CIWARS, we also identify a number of optimizations that can be achieved in the future, such as exploiting parallelism among functions, or reducing the transactional costs between function executions by tuning decomposition granularity. We plan to also automate the decomposition process itself by compiler-assisted program analysis to identify function boundaries. For this exploration, we deployed our own FaaS backend as we needed detailed low-level runtime information to fine-tune our approach, which is not readily available in platforms such as AWS Lambda. However, as we improve our tools, we will not need a local FaaS backend and will be able to use AWS Lambda, etc.

Specifically, we make the following contributions:

- We discuss recent works in converting applications to the FaaS model, and identify techniques that can be leveraged in our tool for automatic decomposition.
- We present the design of our tool to decompose traditional application workflows into function sets that maintain correctness while preserving performance. We show that the resulting functions can be deployed on serverless platforms such as AWS Lambda.
- We evaluate manually decomposing the CIWARS Bioinformatics application into AWS-Lambda-ready function sets, comparing two versions of the application to better understand the trade-offs between performance and scalability of the proposed transformation techniques.
- We discuss future steps needed for implementing our tool and optimizing it for performance and scalability.

2 BACKGROUND

Decomposing a legacy (or traditionally developed) application into serverless ready function sets is a complex problem requiring identifying the function boundaries within an application and matching of the application characteristics to that of the target serverless platforms. A number of works have started exploring this transformation, which we categorize and discuss in the following.

2.1 Serverless Performance Analysis

Wang et al. [28] surveyed the performance of a large number of function instances on a major commercial FaaS platform (up until 2018). It systematically measured the architectural, resource scheduling, and performance isolation characteristics of AWS Lambda, Azure Functions, and Google Cloud Functions. The work found that none of the three platforms completely hide tenants' runtime information from each other, exposing potential vulnerabilities to dedicated attacks. However, container warming technique can help reduce the cold start overhead and resource utilization. Furthermore, the work showed that resource contention can be caused by failed performance isolation among instances or even tenants, those who have more pre-configured resources, mainly memory, will get better resources while in resource contention.

In contrast, Yu et al. [29] proposes an all-in-one benchmark for serverless platforms. The function splitting strategies here are crucial: applications can be decomposed based on periods of consistent resource consumption to avoid pre-configured resources being wasted; and serverless platforms tend to limit allowed concurrency in one instance. Thus, splitting parallelizable regions into different functions could help the overall performance. The work also shows that sequential chaining of function instances generally requires less resource and execution time than nested chaining. Saving implicit states (runtime information, code, etc) of the instances of one function can be later used to optimize the overall performance of all the instances of the same function.

2.2 HPC FaaS Platforms

SAND [5] identifies the main problem of applying existing commercial FaaS platforms onto functionized HPC applications as the startup and communication latency among functions from the same application. The work shows that the function calls from the same application demonstrate the pattern that the outputs of one function are usually immediately fed into another, therefore isolating function invocations of the same application with processes instead of the stronger isolation through containers is viable. *SAND* also implements a hierarchical message bus that routes the output of one function to the next if the next function is hosted on the same node, reducing the global message passing latency. However, the work does not provide a solution for isolation among different users.

The problem of scientific applications' deployment on FaaS platform is explored in [10], which recognized that current serverless platforms do not integrate the HPC resources well and the reliance on Docker requires super user privileges, creating security concerns. A FaaS platform is designed for scientific computing, which allows users to register Python code snippets as functions, takes a JSON object as inputs, invoke registered functions, and monitor execution via REST API exposed by a *funcX* service. *FuncX* uses different container technologies that fit different resource types. Singularity [19] and Shifter [16] are used to ensure better isolation on larger-scale HPC facilities, while Docker [20] works better on local and cloud deployments [10].

Faasm [24] argues that the problems with current serverless computing mainly arise from data access latency and resource footprint. The data access latency is constrained by the stateless nature of serverless computing: states have to be moved from/to computation, and between function invocations. Similarly, the resource footprint is constrained by the container technologies the current platforms embrace, cold-starting containers is still quite expensive. The work

proposes a stateful serverless abstraction, Faaslets and its runtime Faasm. All Faaslets instances on the same host are placed within one address space, and share states through shared memory regions. Faaslets also employ a two-tier state architecture, with the first tier being the local sharing, and the second tier being a distributed state sharing across hosts. This model reduces cold-start latency through firing up new instances from snapshots of pre-run Faaslets functions. One disadvantage with this model is that it requires user code changes to invoke Faasm specific APIs to fully make use of the two-tier state sharing mechanism. The work helps reinforce our thesis that the placement of function instances on hosts would affect the communication efficiency, and in turn would affect our proposed code decomposition policies.

2.3 Serverless Function Decomposition

FaaSter [26] compares FaaS computing model with monolithic application execution through conducting FaaS transformation on a series of scientific computations. It proposes the idea of splitting functions based on the potential timeout cutoff of a function. The partial functions after splitting are connected through a nested chain. This way even if the previous partial functions are timed out, the following partial functions will proceed to execute in other instances, thus preserving the semantic of the original function. The work also introduces the categorization of different levels of FaaSification: shallow FaaSification, splitting the application into the units of functions; medium FaaSification, splitting the application into code snippets; deep FaaSification, splitting the application into the units of instructions. The work is complementary to our approach in designing our decomposition strategies.

Spillner [25] offers the bases for systems such as FaaSter, and targets a problem similar to ours, i.e., how to automate the process of lambdafication on Python cloud applications? They propose Lambada, a tool to automate the transformation of cloud applications to be lambda-ready. The tool recursively scans the modules according to function dependencies and transforms them into corresponding Lambda modules, with Lambda runtime as the gateway across module boundaries. Functions are transformed into remote functions with stub functions at the local as the entry point. Classes are decomposed into functions, deployed with a similar tactic that a local proxy class and a remote proxy class exchange call arguments and function states. The transformation proposed in Lambada is useful as a skeleton reference for general transformation from cloud application to FaaS function. However, in contrast to such works, we also consider performance optimizations and general FaaS platforms, and not only a solution tailored solution AWS Lambda.

3 METHODOLOGY

The initial CIWARS pipeline is composed of a number of monolithic tools [7, 8, 22] as shown in Figure 1. The pipeline forms a DAG, but the boundaries of the individual components are only conceptually defined. The existing implementation still packs everything into one monolithic unit. The domain scientists and users of CIWARS ideally would like to have an ability to replace/exchange the pipeline components and extract intermediate data from various points in the pipeline—something that cannot be supported in the monolithic version without significant redesign—to enable analysis

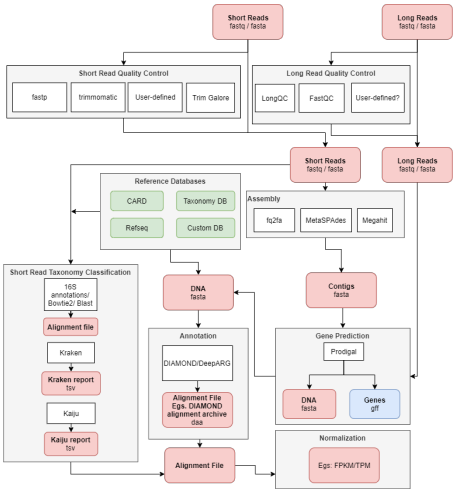

Figure 1: The CIWARS cyberinfrastructure pipeline, which employs tools such as MetaCompare [22], MetaStorm [7], and DeepARG [8]. Note that given the monolithic nature of individual components, some functionalities such as short reads and annotations are repeated in the components.

and tweaking/curating of data between components (§1). Therefore, the general guideline we follow in transforming CIWARS to a function set is to modularize all the functions with boundaries defined by the designers, i.e., as close to the conceptual DAG as possible.

We perform a shallow decomposition [26] of the CIWARS pipeline, which can help us create functions that can be used in the serverless context. The challenge we faced is that the components are all packed into a single application, which entails manual splitting. We consider the relations between the components for this purpose. For instance, MetaCompare [22] contains the "Annotation" component, as well as a component for scoring an "Alignment" File. However, this overlaps with DeepARG's [8] functionalities of "Annotation". The input, expressed as a FASTA file [23] by the "Assembly" component, can either go into DeepARG's annotation component, or MetaCompare's annotation section. The output from either of these can be scored by MetaCompare's "Alignment" component. Therefore, we need to separate the "Annotation" and "Alignment" components of MetaCompare into two functionizable code snippets. As a design rule for modularizing components, we separate a functionality from a monolithic component if the functionality can be used by multiple other components and support the same input and output formats. Thus, a modularized component can be instantiated as needed to support multiple processing pipelines. The modularized component model also provides for trade-offs in the pipeline. For instance, we can have an annotation component that is fast but not as accurate, or vice versa a very accurate but slow one. The model provides users with the ability to pick and choose to meet their overall application needs.

Once the application is modularized into a function set, the next step is to support a large-scale deployment to speed up the process and to support more users. We target AWS Lambda [15]

for this purpose, and design automatic mechanisms to deploy the functions atop the computing substrate. To coordinate among the decomposed function instances, automate the execution of the designated partial pipeline according to users' specifications, and manage the implicit and explicit states of instances [29], we create our own AWS Lambda-ready lightweight serverless architecture.

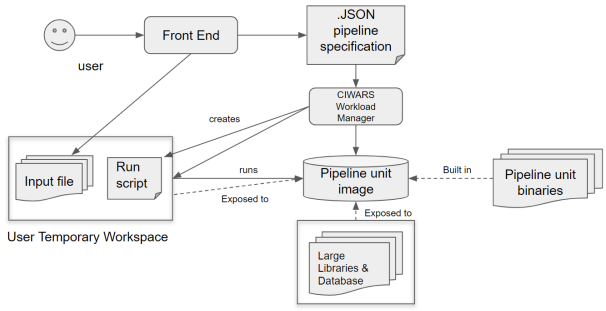

Figure 2: The proposed framework for instantiating decomposed functions created by our approach atop AWS Lambda.

Figure 2 shows the overall architecture of our approach. Users upload their input files, select the desired pipeline components, and specify the configurations for the components using a front-end[1]. The configurations are then uploaded to our server—a Django server that captures the configuration and metadata, parses them into JSON, and creates a temporary directory for this user on the server storage. A workload manager then processes the JSON files, creates a run script with user configurations in the user directory, and coordinates the designated functions to run. The workload manager manages the execution order of the functions/instances through Kubernetes [3]. The functions are pre-registered and containerized through Docker [20], function-proprietary libraries and databases are shared among functions and are exposed to the containers. The Django server will notify users through the front-end when the execution is done. Thus completing the process.

4 DESIGN

The exercise of manually faasifying a monolithic application gives us insights on how to devise an automated program refactoring tool capable of identifying code regions, at both coarse- (i.e., function or class level) and fine- (statement level) level granularity, best suited to be ported to a serverless platform. This section proposes a tool that takes as input a monolithic pipeline and a sample input data, and outputs a decomposed set of functions to be deployed on serverless platforms. While searching for possible candidates in an application for FaaS, our objective is to increase performance from undiscovered parallelism in the program and, to a great extent, utilize the available scalability of the serverless platform.

To identify candidate code regions in a given program, we first have to define the granularity at which our approach will decompose the program. We resort to a greedy approach by starting with the most coarse granularity, i.e., at the function level, and then applying finer granularity decomposition, i.e., statement-level, while preserving the sequential order of statements. This approach helps manage the search space of all possible decompositions of a program rather than experiencing search-space explosion from fine-grained-level that may arise from statement re-ordering.

Starting with the most coarse-grained level in the first stage, we construct a program decomposition at the function level. We pass the decomposition candidates to the second stage and generate a control-flow graph (CFG) at the level of function calls. Then, we identify branching locations in the CFG where the execution path splits (§4.2), leading to two possibly independent code blocks. These code blocks are considered as possible candidates for faasification. For each of those code blocks, we expand data flow boundaries by adding statements present in the hierarchy of the block's data dependency graph (DDG) but independent of the rest of the program execution (§4.3). These two decomposition steps are iteratively applied at a finer-grained level until we find the local maxima for performance. For each candidate decomposition, we perform concrete execution on the sample input to document the resulting performance gains/loss. We formalize our decomposition mechanism with the following terms:

- *Decomposition set D* represents a set of locations in the original pipeline code where the suggested decomposition boundaries can be established.
- *Generation D(X)* represents the number of stages of decomposition that have been applied to the current decomposition set. *X* represents the total rounds of control-flow decomposition and data-flow analysis that have already been applied. For example, *D(0)* represents the original application with function boundaries as decomposition boundaries. *D(1)* is the set after the first round of control-flow decomposition and data-flow analysis.
- *Code snippet* is a piece of code to be decomposed, and one generation can contain multiple code snippets.

4.1 Baseline Granularity

In our experience of transforming CIWARS pipeline §3, we observe that although the pipeline does not have user-annotated sub-pipeline boundaries, it is compartmentalized in independent functions. A similar observation was reported by FaaSter [26], which describes a shallow level of Faasification with functions as the atomic unit of the decomposition. Our insight is that such computational pipelines may already be partially modularized into functions, capturing the implicit computational boundaries intended by the developer and, hence, can easily be transformed into FaaS. As a first step, we leverage the already existing functions as a possible decomposition and deploy them as FaaS to measure the performance gains over baseline application. We define this initial baseline decomposition as D(0).

However, relying on the user to define decomposition locations in a program would lead to many missed opportunities to improve performance. Each user-defined function may contain parallelizable computation, which can be decomposed into multiple smaller FaaS. For instance, in Figure 3, the convolution operation (lines 1–13)

[1]Currently, we use manually created XML/JSON objects to specify the setup. In the future, we envision a visual tool that will help users to make this step easier.

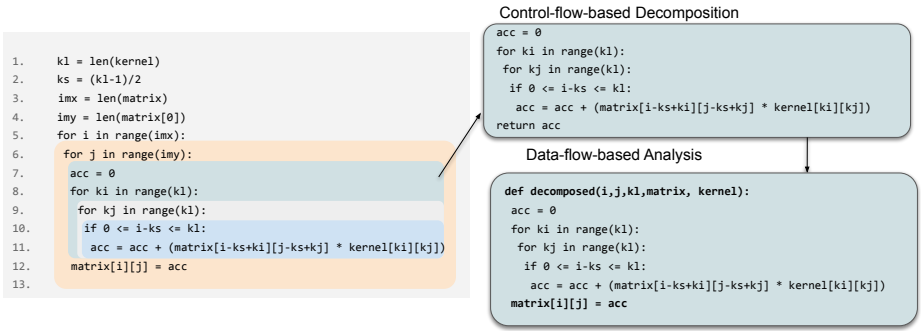

Figure 3: A sample application of control-flow-based decomposition followed by data-flow-based analysis. The grey box on the left presents a part of a monolithic Python application that performs a convolution operation on a given image, `matrix`.

can be considered as a function and transformed into FaaS. However, better decomposition locations–either of the orange, green, grey or blue boxes–would lead to more parallelism and scalability. We recognize this limitation and incorporate static and dynamic program analysis techniques in our approach to fully explore such parallelizable components.

4.2 Control-flow-based Decomposition

To further explore opportunities to parallelize, we leverage static program analysis techniques to identify code blocks independent of other code blocks, which lie at the branching locations (e.g., IF, FOR, WHILE, and SWITCH) of the program. For this purpose, we employ existing static analysis for Python, e.g., control flow analysis [21], to generate a control-flow graph (CFG) of a given pipeline. Once the CFG is generated, we traverse the graph to identify sub-paths in the program that emerge from branching conditions but do not have cyclic dependencies. Code blocks outside of these branching locations are composed in their own functions. A function invocation to a method defined within the given block also counts as a branching location. Due to our technique's recursive nature, the control-flow-based decomposition will be applied on the CFG in a top-to-bottom fashion, i.e., level-order traversal. The succeeding stages will be responsible for all other decomposition locations in the bottom or middle part of the CFG.

The insight behind using control flow as a criterion to decompose a program is that if we want to achieve deep Faasification, splitting code at the control points maintains correctness and sequential order, and is also coarse-grained. For example, in Figure 3, the FOR loop at line 6 represents such branching location where the code block (lines 6-12) will be executed if the loop-predicate is true. Thus, the orange box represents D(1), the decomposition set constructed after the first CFG based decomposition. As another example, consider an IF clause, where both true and false sides of the branches are independent of each other and can be viewed as a candidate for FaaS.

However, the decomposition of a program based solely on control flow may overlook some trivial parallelizing opportunities. Consider a program with an IF condition where the branch's true side invokes external heavy libraries, and the false side invokes lightweight computations. The computation disparity in two independent program paths may lead to Faasifying a relatively light computation instead of the one depending on large-sized library packages. This would entail profiling in the next data-flow-based analysis step.

4.3 Data-flow-based Analysis

After defining a set of decomposition boundaries, it is critical to identify the data dependencies of an extracted function. For example, in Figure 3, the extracted code in blue box depends on variables such as `kl`, `matrix`, `i`, and `j`. Similarly, the variable `acc` is used by line 12, and therefore, line 12 depends on a complete execution of the blue box. To perform a dependency analysis, we extend our static analysis to construct a data dependency graph (DDG) of the given pipeline. DDG provides fine-grained information on how a variable or a code region is formed and what variables are needed for its correct execution.

We observe that an edge in the data dependency graph can interchangeably be interpreted as a happens-before relationship between two code statements. Based on this observation, we first explore all variables used in the current FaaS candidate and then isolate their dependencies (i.e., other variable and code statements) via a DDG. This exercise also helps with constructing a legal function definition with input parameters and return type.

During the data dependency analysis, we recognize more room for improvement. Using the edges in the DDG, we can also find a code statement that satisfies the following two conditions: 1) the statement is adjacent (1-edge away) to the FaaS candidate under investigation and the candidate is data-dependent on the statement, and (2) no other code block in the sibling path depends on the statement. When such statements are presented, they are appended to the current candidate FaaS without affecting the original pipeline's sequential order.

After the data dependency analysis and function expansion, a profiling stage, as described in §4.2, will be performed to compare the two decomposition sets to distinguish the one that delivers better performance than the other. Ultimately, we expect multiple aspects of a function to be measured during this final profiling, among which execution time and memory footprint would be given heavier weights. The profiling stage will also decide whether this

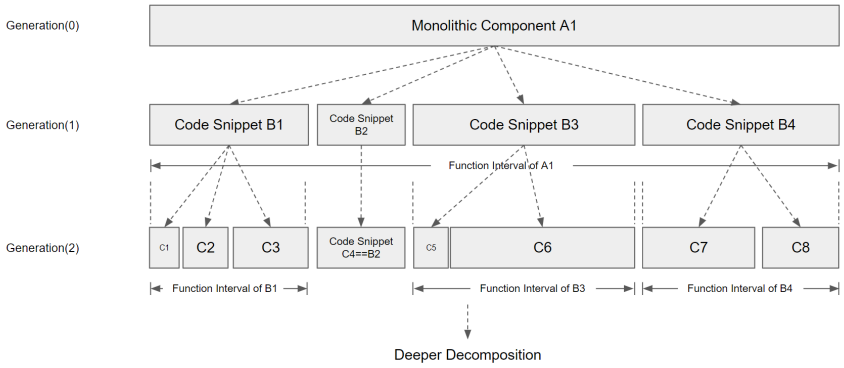

Figure 4: A sample evolution of the decomposition on a monolithic application. Each code snippet is decomposed into a set of multiple smaller functions. If the overall performance of the resulting set is better than the preceding set, then we keep the resulting set, otherwise the original set is preserved.

generation of decomposition is successful or not, if the performance of the decomposed result is better than the code prior to this generation's decomposition efforts, we keep the decomposed code as the new generation. Figure 4 shows an example of how fine-grained decomposition is achieved for an application code after several generations of control-flow-based decomposition (§4.2) and data-flow-based analysis (§4.3).

4.4 Just-in-Time Analyzer

After the control-flow and data-flow analysis within a stage, we evaluate the performance of the current generation on user input data. If the current decomposition step performs better than the previous one, we move to the next stage, with finer-grained decomposition. The profiling step helps us keep on track while searching for the best decomposition set and acts as a stopping condition.

However, the decision of this step will vary hugely based on the user's input. Different inputs will lead to different execution paths at runtime, and different input sizes might have different resource footprints and execution times. Therefore, we need to decide the decomposition of our functions dynamically. We propose a Just-in-Time Analyzer to perform all the decomposition stages on the application before running it, offloading the responsibility of profiling to a back-end workload manager (Figure 5). The configuration on a monolithic application with a single set of input will come to an asymptotic state after several generations when further decomposition no longer brings performance gain if not loss. After a set of decomposed functions is finalized, the analyzer will keep all the information of each generation, including the decomposed functions, their signature, and their dependency relations.

A challenge in this context is as users execute more and more application instances, the input data varies and its behavior will change over time. Thus a generation that offered a desirable performance in the past needs to be re-evaluated and the analyzer may select a new generation. Such re-evaluation can be done at fixed intervals, e.g., after a predetermined number of instances have been run, or if the input changes drastically, e.g., the file size varies significantly. This re-evaluation can go both forward, i.e., decompose to a new generation, or backward, i.e., retrace to a previous generation. This would be akin to going down or up the tree of Figure 4. The goal is for the analyzer to dynamically match the workload and use a generation that can provide improved performance.

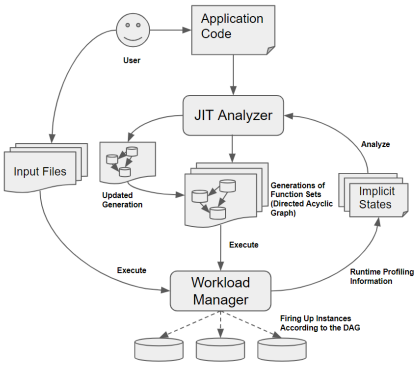

Figure 5: Overall structure of our decomposition runtime.

4.5 Discussion

In this section, we presented a vision of our tool and its overall design; however, there are several open challenges related to generalizability, completeness, efficacy, and soundness. For instance, the data dependency graph does not incorporate the cold-start and data transfer cost of a set of adjacent instructions separated by our tool. A bare bone runtime measurement does not reflect all performance characteristics essential to make decomposition decisions. Similarly, at each generation's decomposition, not all functions need to be reevaluated, and a better performance could still be achieved if we only keep decomposing the largest functions in the current set of functions or only reconsider the smallest functions that are likely suffering cold-start overhead. We look forward to addressing these challenges in the future design and implementation of our tool.

5 EVALUATION

We answer two questions: Does our decomposition implementation preserve correctness, and what are the performance loss and potential gain exposed by our decomposition approach?

5.1 Experimentation Setup

Testbed. We focus on showing correctness so our testbed consists of a single node with an 8 core Intel i7-9700K CPU, 32 GB RAM, and a 500 GB SSD. We isolate the decomposed functions using Docker 20.10.3[2], and build our own image based on the official Python3 base image [4]. We run our decomposed applications on Ubuntu 18.04.2 LTS. We test our decomposition approaches on the MetaCompare and deepARG components of the CIWARS pipeline.

Dataset. We downloaded 100 FASTA-formatted [23] metagenomic sequence read data with a uniformed size of 500 MB from a local wastewater management site in Christianburg, VA. The data is pre-processed through a gene prediction tool called Prodigal [14] to prepare it for annotation by MetaCompare. For DeepARG, we also downloaded 100 metagenomic raw FASTQ-formatted [12] sequence read data sets that come from the University of Hong Kong and range from 6 GB to 7 GB [1]. The datasets are representative of those used by the domain scientists in the field. Both MetaCompare and DeepARG take two input files for an execution.

5.2 Preserving Correctness

We decompose MetaCompare into its "Annotation" and "Alignment" components using our approach of §3. The resulting function set *{Annotation, Alignment}* are arranged such that the input is provided to the *Annotation* function, whose output is then fed to the *Alignment*, which then provide the output. The functions are run using the system shown in Figure 2. Similarly, MetaCompare is decomposed into *{Short Read, Annotation}* function set and executed. First, we manually examined using compiler analysis the key components in the original as well as the decomposed version to check the correctness of the code, and found the two versions to be achieving similar computations. Next we executed the original and decomposed FaaS versions of the tools with 100 different inputs from our datasets, and for each case compared the output files. We determined that our decomposed version matched the output for all of our tests. These experiments show that the approach is able to correctly transform the monolithic components of CIWARS into FaaS-ready function sets.

5.3 Performance Analysis

In our next experiment, we study the execution time of the original and decomposed versions of MetaCompare and DeepARG to gauge the performance of our approach. We measure the execution time of our decomposed code by placing python timing instructions at its beginning and after the completion of state persistence, and calculating the difference. For the original versions, we measure the overall execution time of the monolithic instances; for our decomposed implementation, we measure from the beginning of the first component's instance until the completion of the last component's instance. We repeat the experiment three times and report the averages.

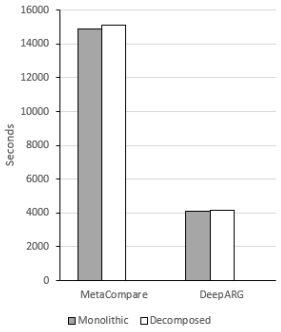

Figure 6: Execution time for Monolithic Vs. Decomposed implementations.

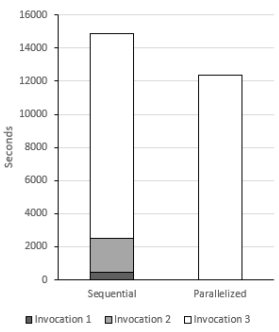

Figure 7: Execution time with Sequential vs. Parallelized library invocations in MetaCompare.

Figure 6 shows the average execution times for the two studied versions of the tools. We observe that the FaaS version incurs 1.5% and 2.0% overhead as compared to the monolithic version for MetaCompare and DeepARG, respectively. We note that while the raw one-instance comparison of the original and FaaS version of the tools incur the overhead, the FaaS version opens up the tools to benefit from parallelism and on-demand scale out.

Transactions cost. The main reason for the degradation in performance in the decomposed version is due to transactions between the functions. The functions need to utilize intermediate storage to persist state. This overhead is dependent on how the original tools persisted states between their various components. MetaCompare adopted an in-memory in the monolithic version, which added an explicit intermediate storage in the decomposed version. However, the overhead is small as the intermediate dataset for the tool is not as large (500 MB) incurring a smaller number of accesses to the storage. In contrast, DeepARG already relied on intermediate storage for communication between its components. Thus, even though it processed a much larger amount of intermediate data in our tests (27 GB), both versions of DeepARG use the intermediate storage in a similar way, resulting in only a small overhead.

We infer that while the decomposed version of a monolithic application would require intermediate storage for preserving state between components, the original version would also do the same due to smaller memory sizes compared to the very large dataset sizes. Thus, the expected overhead in real tools arises from extra invocations to external libraries and APIs in the decomposed version. By properly defining the component boundaries and by optimized implementations, this overhead can be made negligible or amortized over large dataset sizes. As observed above, our first implementation of the tool incurred $< 2.0\%$ overhead.

What we can learn from the observation is that persisting states is inevitable in original monolithic applications, both invocations to external libraries and the application itself already tend to store the states that are too big to fit in the memory on to disks. Therefore, under the assumption that the monolithic application has well-defined component boundaries, in many situations, decomposition

of existing monolithic code may not bring too much performance degradation.

Exploiting Parallelism. During the decomposition of MetaCompare, we observed that its implementation involves three external library function invocations, and that these invocations are independent of each other. In the previous experiment, our decomposed version had these external invocations grouped into a single instance. In this experiment, we also parallelize these external library function invocations using FaaS. We measure the execution time under this approach and compare it with our previous version. Figure 7 shows the result and breakdown of the time taken by each of the library invocations. In our first decomposed version, the library invocations are sequential, whereas in our parallel version the three can run simultaneously. As a result, the overall execution time for the parallel version is reduced—though still dictated by the longest invocation, i.e., #3—by 17%.

We note that even after the parallelization of the library invocations, the overall execution time is well over 30 minutes, which is much more than the average life cycle of a typical serverless instance [18]. This can be remedied by scaling out the system, i.e, instead of just one FaaS instance processing all of the input data, create a large number of instances each processing a fraction of data. However, this may significantly increase function-to-function communication. We plan to explore such optimizations and trade-offs therein in our future work.

6 CONCLUSION

Serverless computing offers great promise for legacy and emerging scientific computing applications. In this paper, we presented a systematic approach to decompose a scientific workflow into a set of functions that can be deployed on serverless platforms such as AWD Lambda. We demonstrate our approach in the context of a bioinformatics pipeline (CIWARS). Our decomposition goal is to preserve correctness of the original application without compromising performance. We also presented the design of an automation tool that can provide finer-grained FaaS functions. The tool employs a two-step approach of control-flow decomposition and data-flow analysis to transform a monolithic application into functions for the target CIWARS. The tool also provides the foundation for building general-purpose automatic tools for FaaS implementations of general monolithic applications. Our evaluation shows that our approach is able to preserve correctness, and incur only small (2.0%) overhead compared to the original application. In the future, we aim to fully automate our process, and test our solutions more rigorously and at scale.

ACKNOWLEDGEMENTS

This work is sponsored in part by the National Science Foundation under grants CCF-1919113 and OAC-2004751.

REFERENCES

[1] 2018. Actiavted sludge samples collected from Shatin WWTP, Hong Kong. https://www.ncbi.nlm.nih.gov/bioproject/PRJNA432264. Accessed: April 9, 2021.
[2] 2021. Docker Engine Release Notes. https://docs.docker.com/engine/release-notes/#20103. Accessed: April 9, 2021.
[3] 2021. Kubernetes. https://kubernetes.io/. Accessed: April 9, 2021.
[4] 2021. python - Docker Hub. https://hub.docker.com/_/python?tab=description. Accessed: April 9, 2021.
[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 923–935.
[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC*.
[7] Gustavo Arango, Gargi Singh, Lenwood Heath, Amy Pruden, Weidong Xiao, and Liqing Zhang. 2016. MetaStorm: A Public Resource for Customizable Metagenomics Annotation. *PloS one* 11 (09 2016), e0162442.
[8] G. A. Arango-Argoty, E. Garner, A. Pruden, L. S. Heath, P. Vikesland, and L. Zhang. 2017. DeepARG: A deep learning approach for predicting antibiotic resistance genes from metagenomic data. *bioRxiv* (2017). https://doi.org/10.1101/149328
[9] Paul C. Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry. *CoRR* abs/1906.02888 (2019). arXiv:1906.02888
[10] Ryan Chard, Tyler J. Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. 2019. Serverless Supercomputing: High Performance Function as a Service for Science. arXiv:1908.04907 [cs.DC]
[11] A. Christidis, R. Davies, and S. Moschoyiannis. 2019. Serving Machine Learning Workloads in Resource Constrained Environments: a Serverless Deployment Example. In *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*. 55–63.
[12] Peter Cock, Christopher Fields, Naohisa Goto, Michael Heuer, and Peter Rice. 2009. The Sanger FASTQ format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research* 38 (12 2009), 1767–71. https://doi.org/10.1093/nar/gkp1137
[13] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. 2021. Serverless Applications: Why, When, and How? *IEEE Software* 38, 1 (2021), 32–39.
[14] Doug Hyatt, Gwo-Liang Chen, Phil Locascio, Miriam Land, Frank Larimer, and Loren Hauser. 2010. Hyatt D, Chen G-L, LoCascio PF, Land ML, Larimer FW, Hauser LJ.. Prodigal: prokaryotic gene recognition and translation initiation site identification. BMC Bioinform 11: 119. *BMC bioinformatics* 11 (03 2010), 119.
[15] Amazon Web Services Inc. 2021. Amazon Lambda. https://aws.amazon.com/lambda. Accessed: April 9, 2021.
[16] Douglas M Jacobsen and Richard Shane Canon. 2015. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group* (2015), 33–49.
[17] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *Proceedings of the 26th International Conference on World Wide Web, WWW*.
[18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.
[19] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20. https://doi.org/10.1371/journal.pone.0177459
[20] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014 (03 2014).
[21] Jan Midtgaard. 2012. Control-Flow Analysis of Functional Programs. *ACM Comput. Surv.* 44, 3, Article 10 (June 2012), 33 pages.
[22] Min Oh, Amy Pruden, Chaoqi Chen, Lenwood S Heath, Kang Xia, and Liqing Zhang. 2018. MetaCompare: a computational pipeline for prioritizing environmental resistome risk. *FEMS Microbiology Ecology* 94, 7 (04 2018).
[23] William Pearson and D Lipman. 1988. Pearson WR, Lipman DJ. Improved tools for biological sequence comparison. Proc Natl Acad Sci USA 85: 2444-2448. *Proceedings of the National Academy of Sciences USA* 85 (05 1988), 2444–8.
[24] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.
[25] Josef Spillner. 2017. Transformation of Python Applications into Function-as-a-Service Deployments. *CoRR* abs/1705.08169 (2017).
[26] Josef Spillner, Cristian Mateos, and David A. Monge. 2018. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In *High Performance Computing*, Esteban Mocskos and Sergio Nesmachnow (Eds.). Springer International Publishing, Cham, 154–168.
[27] H. Wang, D. Niu, and B. Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*.
[28] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.
[29] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. 30–44.