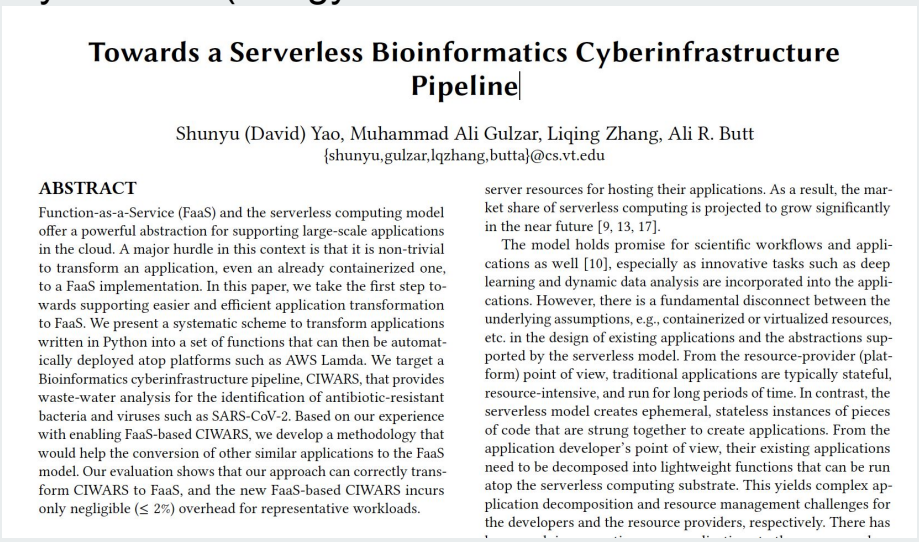
Towards a Serverless Bioinformatics Cyberinfrastructure Pipeline

Presented by: Team 3 (Bhagyashree Aras and Dhruvi Kaswala)



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.

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

Agenda

- ▷ Problem Introduction
- ▷ Proposed Solution
- ▷ Related Work
- ▷ Summary of Technology Approach
- ▷ Experimental Evaluation
- ▷ Conclusion
- ▷ Critique (Strengths, Weaknesses, and Evaluation)
- ▷ Q&A

Problem Introduced in the Paper

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.

How the problem is solved?

The paper proposed a systematic scheme to transform applications written in Python into a set of functions that can then be automatically deployed on platforms such as AWS Lambda.

Related Work

Requiring identification of 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:

Related Work: Serverless Performance Analysis

Wang et al measured the architectural, resource scheduling, and performance isolation characteristics of AWS Lambda, Azure Functions, and Google Cloud Functions.
Issues:

1. None of the three platforms completely hide tenants' runtime information from each other; exposing potential vulnerabilities to dedicated attacks. But, container warming technique can help reduce the cold start overhead and resource utilization.

Related Work : Serverless Performance Analysis(contrast)

Yu et al. proposes an all-in-one benchmark for serverless platforms. The function splitting strategies here are
crucial:

1. Applications can be decomposed based on periods of consistent resource consumption to avoid pre-configured resources being wasted > splitting parallelizable regions into different functions
2. sequential chaining of function instances > requires less resource and execution time than nested chaining.
3. Saving implicit states (runtime information, code, etc) of the instances of one function > optimize the overall performance

Related Work : HPC FaaS Platforms

Issues:

1. Startup and communication latency among functions from the same application
2. Isolating function invocations of the same application with processes instead of the stronger isolation through containers is viable
3. Implements a hierarchical message bus > the work does not provide a solution for isolation among different users
4. Current serverless platforms do not integrate the HPC resources well and the reliance on Docker requires superuser privileges, creating security concerns.

HPC FaaS Platforms - 2

Faasm argues that the problems with current serverless computing mainly arise from data access latency and resource footprint. Thus, proposed 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.

1. Faaslets also employ a two-tier state architecture:
    a. The first tier being the local sharing
    b. The second tier being a distributed state sharing across hosts.

Related Work : HPC FaaS Platforms - 3

Advantage: This model reduces cold-start latency through firing up new instances from snapshots of pre-run Faaslets functions.

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.

Related Work : Serverless Function Decomposition

FaaSter proposes the idea of splitting functions based on the potential timeout cutoff of a function.
The work also introduces the categorization of different levels of FaaSification:

1. Shallow FaaSification, splitting the application into the units of functions
2. Medium FaaSification, splitting the application into code snippets
3. Deep FaaSification, splitting the application into the units of instructions

Related Work : Serverless Function Decomposition

Spillner propose Lambada, a tool to automate the transformation of cloud applications to
be lambda-ready.

1. 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.
2. Functions are transformed into remote functions with stub functions at the local as the entry point.

Related Work : Serverless Function Decomposition

3. 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.
4. The transformation proposed in Lambda is useful as a skeleton reference for general transformation from cloud application to FaaS function

Approach used in this paper

Summary of approach used

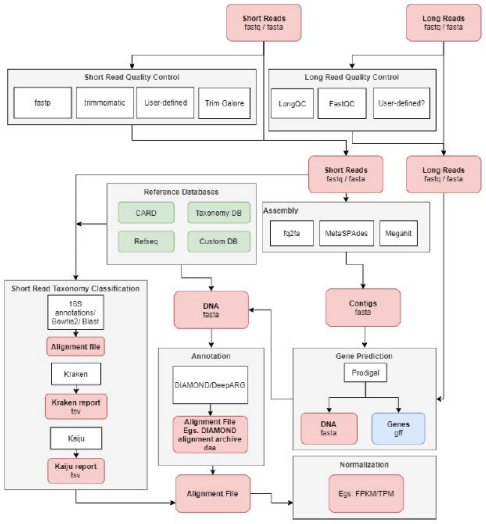


Figure 1: The CIWARS cyberinfrastructure pipeline, which employs tools such as MetaCompare [22], MetaStorm [7], and DeepARG [8]. Note that given the monolithic nature of individual components, some functionalities such as short reads and annotations are repeated in the components.

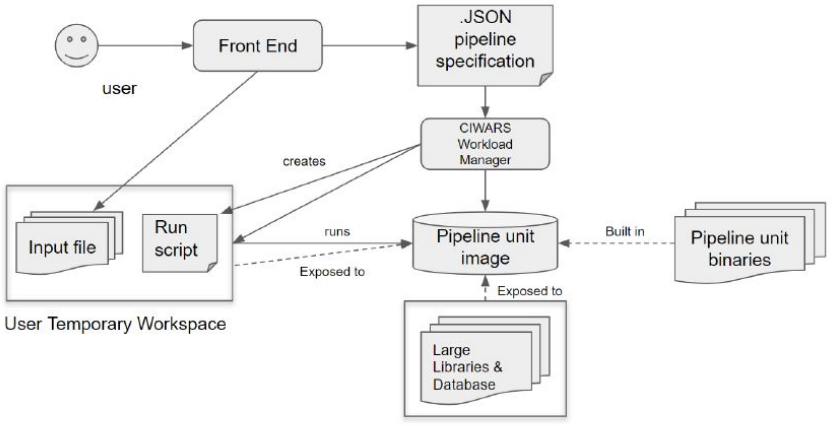


Figure 2: The proposed framework for instantiating decomposed functions created by our approach atop AWS Lambda.

Summary of approach used - 2

1. Decomposition set represents a set of locations in the original pipeline code where the suggested decomposition boundaries can be established.
2. Generation represents the number of stages of decomposition that have been applied to the current decomposition set.
3. Code snippet is a piece of code to be decomposed, and one generation can contain multiple code snippets.

Baseline Granularity

Utilised shallow level of Faasification with functions as the atomic unit of the decomposition. 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.

Issue: However, relying on the user to define decomposition locations in a program would lead to many missed opportunities to improve performance.

Solution: incorporate static and dynamic program analysis techniques in our approach to fully explore such parallelizable components.

Summary of approach used

```
1.  kl = len(kernel)
2.  ks = (kl-1)/2
3.  imx = len(matrix)
4.  imy = len(matrix[0])
5.  for i in range(imx):
6.    for j in range(imy):
7.      acc = 0
8.      for ki in range(kl):
9.        for kj in range(kl):
10.         if 0 <= i-ks <= kl:
11.           acc = acc + (matrix[i-ks+ki][j-ks+kj] * kernel[ki][kj])
12.     matrix[i][j] = acc
13.
```

Control-flow-based Decomposition

```
acc = 0
for ki in range(kl):
 for kj in range(kl):
  if 0 <= i-ks <= kl:
   acc = acc + (matrix[i-ks+ki][j-ks+kj] * kernel[ki][kj])
return acc
```

Data-flow-based Analysis

```
def decomposed(i,j,kl,matrix, kernel):
 acc = 0
 for ki in range(kl):
  for kj in range(kl):
   if 0 <= i-ks <= kl:
    acc = acc + (matrix[i-ks+ki][j-ks+kj] * kernel[ki][kj])
 matrix[i][j] = acc
```


Control-flow-based Decomposition

Employed existing static analysis, control flow analysis, to generate a control-flow graph (CFG) of a given pipeline.

Why this approach?

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

Issues:

1. may overlook some trivial parallelizing opportunities

Data-flow-based Analysis

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

Advantage:

1. May improve performance.
2. 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.

How fine-grained decomposition is achieved for an application code after several generations of control-flow-based decomposition and data-flow-based analysis

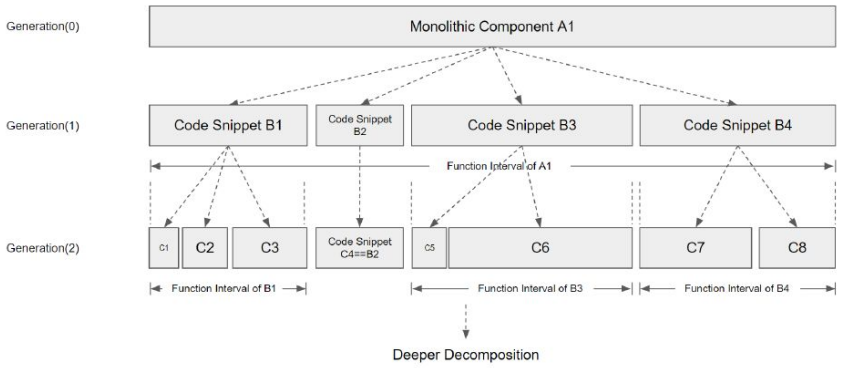


Figure 4: A sample evolution of the decomposition on a monolithic application. Each code snippet is decomposed into a set of multiple smaller functions. If the overall performance of the resulting set is better than the preceding set, then we keep the

Just-in-Time Analyzer

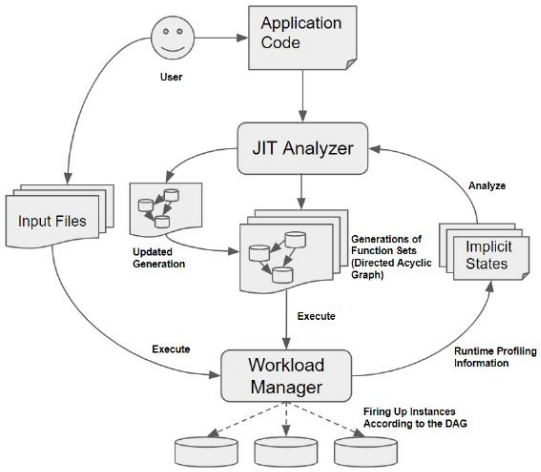


Figure 5: Overall structure of our decomposition runtime.

Experimental Evaluation

Does our decomposition implementation preserve correctness, and what are the performance loss and potential gain exposed by our decomposition approach?

Experimental Evaluation

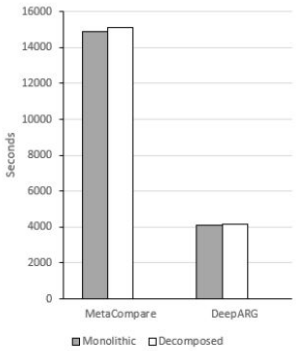


Figure 6: Execution time for Monolithic Vs. Decomposed implementations.

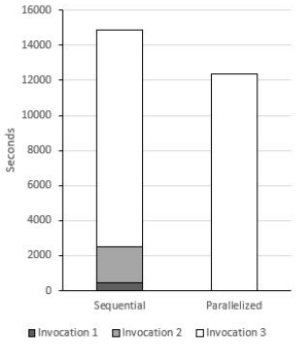


Figure 7: Execution time with Sequential vs. Parallelized library invocations in MetaCompare.

Conclusion

The author has provided a general framework for decomposing the monolithic app through an example, and explained various approaches used in the framework, with a goal to maintain the correctness and not to increase the performance overhead.

Static and dynamic flow of the code has been leveraged to separate the functions.

The paper contains the solution of decomposition (of CIWARS) automating the existing FAAS implementation too

Strengths of the Paper

Manually FaaSifying the existing monolithic application shows the great insight(such as common libraries and meta data requirement) and problems(which needs to be addressed while automating the process for similar application.

Dividing the decomposition into two methods, the static analysis and dynamic data flow analysis, leveraging them was a great technique used in this paper.

Profiling for the resource usage(memory, CPU etc) for the decomposed function to make sure the performance is not degraded was one of the strength of this paper.

Weakness of the Paper

The decomposition of the monolithic application into independent functions depends entirely on the accuracy of the MetaCompare's and DeepArgs's component.

Assuming the custom limited user input data, for design purpose can be misleading as the code flow are entirely dependent on the input variables.

Control flow based decomposition missing the cyclic dependencies, which can be present in monolithic applications, and needs to be addressed.

Critique: Evaluation

Data-flow based analysis could include different combinations of input variables(the user input, configurations, etc)

It would be great to see cost analysis and find the relationship between decomposition degree to the platform cost.

The compiler optimization may be lost while decomposing the functions, which needs to be considered as well.

Thank you for listening!
Q&A session