SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution

Jovan Stojkovic, Tianyin Xu, Hubertus Franke†, Josep Torrellas
University of Illinois at Urbana-Champaign †*IBM Research*
{jovans2, tyxu}@illinois.edu, frankeh@us.ibm.com, torrella@illinois.edu

***Abstract*—Serverless computing has emerged as a popular cloud computing paradigm. Serverless environments are convenient to users and efficient for cloud providers. However, they can induce substantial application execution overheads, especially in applications with many functions.**

In this paper, we propose to accelerate serverless applications with a novel approach based on software-supported *speculative execution* of functions. Our proposal is termed *Speculative Function-as-a-Service* (*SpecFaaS*). It is inspired by out-of-order execution in modern processors, and is grounded in a characterization analysis of FaaS applications. In SpecFaaS, functions in an application are executed early, speculatively, before their control and data dependences are resolved. Control dependences are predicted like in pipeline branch prediction, and data dependences are speculatively satisfied with memoization. With this support, the execution of downstream functions is overlapped with that of upstream functions, substantially reducing the end-to-end execution time of applications. We prototype SpecFaaS on Apache OpenWhisk, an open-source serverless computing platform. For a set of applications in a warmed-up environment, SpecFaaS attains an average speedup of 4.6×. Further, on average, the application throughput increases by 3.9× and the tail latency decreases by 58.7%.

***Index Terms*—Cloud computing, Serverless computing, Function-as-a-Service**

I. Introduction

Serverless computing is an attractive cloud computing paradigm, where users upload application code and the cloud provider secures all the libraries, runtime environment, and system services needed to run it. The basic unit of execution is a function, which runs in an ephemeral, stateless container or micro virtual machine (VM) created and scheduled on demand in an event-driven manner. Applications are then composed of multiple interdependent functions. Compared with traditional monolithic cloud applications, function-based serverless applications can attain elasticity, fine-grained billing, and high resource utilization. All major cloud providers offer such Function as a Service (FaaS) environment, including AWS Lambda [9], Microsoft Azure [56], IBM Cloud Functions [41], and Google Cloud Functions [34]. There are also many open-source frameworks for serverless experimentation [1]–[4], [18], [39], [80].

Despite these attractive benefits, current serverless workloads suffer from multiple overheads, including cold start [6], [48], [52], [80], virtualization, RPC or HTTP invocation, and the need to persist outputs to global storage. Importantly, in applications with many functions, where cross-function control and data dependences are common, such overheads accumulate. The result is long application response times, despite using state-of-the-art frameworks such as AWS Step Functions [14], Azure Durable Functions [54], IBM Cloud Composer [40], or Google Cloud Workflows [35].

In this paper, we aim to fundamentally accelerate the execution of function-based serverless applications. We want to go beyond current work which, while effective, either focuses on cold-start effects (e.g. [6], [7], [20], [24], [27], [29], [32], [36], [48], [52], [59], [60], [66], [69], [72], [80]) or targets one type of overhead such as cross-function communication [8], [22], [44], data access latency [43], [62], [65], [74], [82], [85], or RPC invocation [61], [76]. What we seek is a novel way to execute applications in this paradigm.

An analysis of serverless applications suggests a way to accelerate this environment. Specifically, we find that the outcomes of the branches that encode cross-function control dependences are fairly predictable. Moreover, in this environment where functions are stateless by definition, cross-function data dependences are often predictable. Indeed, functions often produce the same outputs every time that they are invoked with the same inputs. Hence, the data that a function will generate for the subsequent function is typically predictable.

With these insights, we propose to accelerate serverless applications using software-supported *speculation*. Our proposal is termed *Speculative Function-as-a-Service* (*SpecFaaS*). It is inspired by the out-of-order execution of modern processors. In SpecFaaS, functions in an application are executed early, speculatively, before their control and data dependences are resolved. Control dependences are predicted with a software-based branch predictor, while data dependences are speculatively satisfied with memoization. With this support, the execution of downstream functions is overlapped with that of upstream functions, substantially reducing the end-to-end execution time of applications.

While a function execution is speculative, SpecFaaS prevents its buffered outputs from being evicted to global storage. When the dependences are resolved, SpecFaaS proceeds to validate the function. If no dependence violation is detected, the function commits. Otherwise, the buffered speculative data is discarded and the offending functions are squashed and re-executed. SpecFaaS provides policies to configure the degree of speculation based on control/data dependence predictability.

We prototype SpecFaaS on Apache OpenWhisk [18], an open-source serverless computing platform. SpecFaaS runs transparently to the applications. We evaluate SpecFaaS using

three application suites, namely *Alibaba* [50], *TrainTicket* [87], and *FaaSChain*. They have a total of 16 FaaS applications of, on average, 12 functions each. In a warmed-up environment, SpecFaaS attains an average speedup of 4.6×. Further, on average, the application throughput increases by 3.9× and the tail latency reduces by 58.7%.

Overall, this paper makes the following contributions:

- SpecFaaS, a novel approach to accelerate serverless applications with speculative function execution.
- A characterization of dependences and overheads in large serverless applications.
- An implementation and evaluation of SpecFaaS on the OpenWhisk platform.

II. Background

A. Serverless Applications

Large, complex serverless applications are organized as workflows of multiple interdependent functions. To construct these workflows, FaaS platforms typically have composition frameworks, such as AWS Step Functions [14], Azure Durable Functions [54], IBM Cloud Composer [40], or Google Cloud Workflows [35]. These frameworks allow developers to specify control- and data-flow dependences between functions, such as producer-consumer relationships, control branches, loops, parallel execution, and error handling.

In this paper, we implement our infrastructure on top of OpenWhisk [18], an open-source serverless framework that has been used in many prior studies [19], [57], [68], [86]. Listing 1 shows the code snippet of a smart home serverless application that is described in [45], [77], using OpenWhisk Composer [17]. This application is composed of seven functions and Figure 1 shows the workflow.

```
import composer
def main():
    return composer.when('Login',
        composer.sequence(
            'ReadTemp',
            'Normalize',
            composer.when('CompareTemp',
                'TurnAir'),
            'Done'),
        'Fail')
```

Listing 1: Code snippet that implements a smart home FaaS application using OpenWhisk Composer.

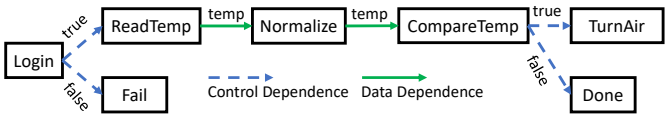

Fig. 1: The workflow of the FaaS application in Listing 1.

The application specifies two types of dependences using the `when` and `sequence` directives. The `when` directive is a branch statement, which specifies a control dependence: `when(br_cond, true_target, false_target)`. If `br_cond` returns `true`, we execute `true_target`, otherwise, we execute `false_target`. The `sequence` directive expresses data dependence: `sequence(source, destination)`. The output of the `source` function will be used as the input of the `destination` function. Applications can also use loop structures, which are specified with `while` and `do_while` directives. These directives are compiled to the same code as `when`, and so we will not consider them separately. It is also possible to execute multiple functions in parallel with the `parallel` directive. Such directive is currently not supported by OpenWhisk Python Composer, and thus we have implemented it ourselves.

An application's workflow is exposed to the FaaS platform. However, the code of each function may not be visible. Therefore, we treat every function as a blackbox.

B. Serverless Workflow Execution

The execution of the functions in an application's workflow needs to satisfy all data and control dependences. For example, in Figure 1, Functions `ReadTemp` and `Fail` do not execute until `Login` finishes and returns the condition that determines which function to execute next. Similarly, `Normalize` does not execute until `ReadTemp` finishes and produces its output.

Serverless platforms schedule each function to execute as soon as it reaches the ready state. Typically, function scheduling is done by a *Controller* component, which keeps track of the state of the workflow graph. In OpenWhisk, after a function completes, the controller calls a helper function called *Conductor*. The conductor picks the next function to execute. Then, the controller initializes a *Worker* that first encapsulates the function in an execution environment (a container [53] or a micro VM [7], [51]) and then launches the function.

C. Implicit Workflows

The workflows described so far are called *Explicit*, in that the developer explicitly specifies the whole graph topology and the controller knows the next function to execute. However, workflows can also be created in an implicit manner. In *Implicit* (or multi-tier) workflows, functions invoke other functions as subroutines. Specifically, a function calls another function, waits on the results, and then continues with its own execution, possibly to call other functions. There are gather functions that invoke multiple simple services and then aggregate the obtained data. An application example from Alibaba [50] is shown in Figure 2. Since the FaaS platform may not know the code of functions, it does not know ahead of time which functions a given function can invoke.

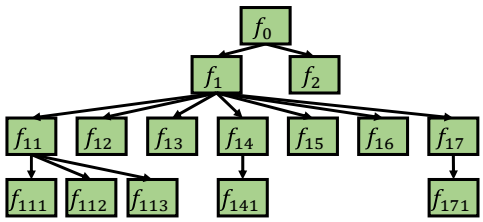

Fig. 2: Example of an application with implicit workflow.

Studies by Alibaba [50] and Facebook [75] show that the multi-tier paradigm is popular in production-level microservice architectures. The largest open-source serverless applications [33], [87] are also built in this way. Note that the caller function is blocked while waiting for the results from a callee.

III. FaaS Environment Characterization

Our work is driven by the characteristics of FaaS application workflows. In this section, we analyze three FaaS application suites running on OpenWhisk. These suites are *Alibaba* [50], *TrainTicket* [87], and *FaaSChain*, which we detail in §VII. Alibaba and TrainTicket use implicit workflows, while *FaaSChain* uses explicit workflows. Table I characterizes these application suites. For each suite, we show the number of applications and, on average per application: the number of functions, the number of cross-function branches, the number of cross-function data dependences, the number of callees per function with calls, the maximum application DAG depth, and the application execution time in a warmed-up environment. For *TrainTicket* and *FaaSChain*, the execution time is obtained by running the applications on AMD EPYC 7402P servers as described in Section VII; for *Alibaba*, the execution time is obtained from the traces.

TABLE I: FaaS application suites considered.

Characteristic	Alibaba	TrainTicket	FaaSChain
Workflow Type	Implicit	Implicit	Explicit
# of Applications	5	5	6
Per-appl. metrics:			
Avg # Functions	17.6	11.2	7.8
Avg # Branches	N/A	1.8	2.5
Avg # Data Deps.	3.4	4.8	2.7
Avg # Callees/Func.	3.4	4.8	N/A
Max DAG Depth	5	3	10
Avg Exec. Time (ms)	387.2	268.8	160.0

Our analysis reveals the following observations:

Observation 1: *Even under warmed-up conditions, function execution per se constitutes less than 1/2 of the time taken to invoke and run an FaaS function.*

Figure 3 shows the average response time of a function invocation under cold start conditions in each of the three application suites. Each bar is broken down into five categories, each with a number expressing their duration in ms. The bottom-most category is *Container Creation*, which includes creating the container and network stack, and connecting to the network. This category is the one taking the longest by far (1500ms), and is shown broken into two pieces in the figure. Next is *Runtime Setup*, which involves injecting the function code and starting the docker proxy. This category is also large and, together with the previous one, constitutes the *cold-start overhead*.

Platform Overhead is the time for the communication between different FaaS platform components such as front-end, controller, and worker when the new request comes. *Transfer Function Overhead* is the time between when a function completes and its successor starts execution. For implicit

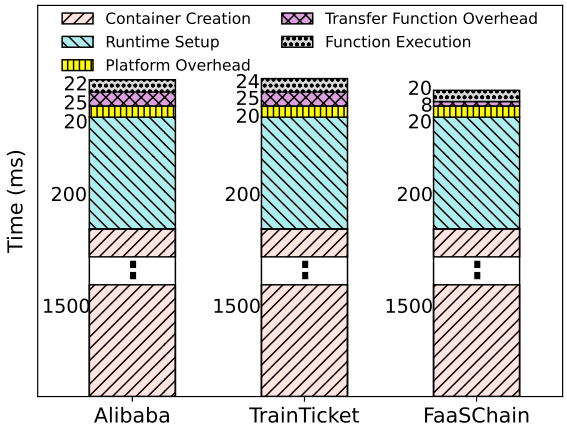

Fig. 3: Breakdown of a function's response time.

workflows, this time corresponds to an HTTP or RPC call; for explicit workflows, it corresponds to additional communication between worker and controller, and the execution of the conductor function. Finally, *Function Execution* is the actual function execution.

From the bars, we get a sense of the overheads: function execution only accounts for 33-42% of the total function response time in warm-up conditions (or 1% in cold-start conditions). Our goal is to minimize *Platform Overhead* and *Transfer Function Overhead*, and overlap *Function Execution* across functions. In addition, if the system runs under cold-start conditions, we want to overlap *Container Creation* and *Runtime Setup* across functions.

Observation 2: *The sequence of functions executed in an FaaS application is highly deterministic.*

The sequence of functions that an application executes can change across invocations of the application. The reason is that some functions conclude with a branch condition that can transfer execution to different functions. In addition, in implicit workflows, some functions call other functions conditionally. In this experiment, we measure, for a given application, the sequence of functions that it executes from beginning to end. We record how many times we observe each of the possible different sequences. We then select the most popular sequence.

On average, the most popular sequence accounts for 90% of the total invocations of the application in Alibaba and 98% in TrainTicket. We do not report the result for FaaSChain because its control dependences use synthetic data (§VII). Based on this result, our goal will be to develop *software branch predictors* to pick functions to execute early, speculatively, before knowing whether they will need to execute.

Observation 3: *Most FaaS functions do not read from writable global state; many do not even write to global state.*

While FaaS functions are stateless by definition, in addition to taking inputs and producing outputs, they may read and write global state. In this experiment, we first measure the fraction of functions that either do not read global state or, if they do, they read read-only state. For example, this fraction is 75.8% for TrainTicket and 85.1% for FaaSChain

across the runs. These functions are guaranteed to produce the same outputs every time that they are invoked with the same inputs. Given these large fractions, our goal will be to maintain memoization tables that record pairs of {inputs, outputs} observed for functions. These tables will allow us to predict the outputs of functions in advance, hence allowing the early, speculative execution of successor functions.

An interesting subset of these functions are those that, in addition, do not modify the global state. These functions produce the same outputs for the same inputs and have no side effects. The fraction of such functions is 57.6% for TrainTicket and 61.7% for FaaSChain. These functions can not only be memoized, but their execution can be skipped altogether!

Observation 4: *Remote storage is not frequently updated.*

We analyze traces of the blob accesses in Microsoft's Azure Functions [55], [65] and observe that write operations are not common. Specifically, out of 40M accesses, only 23% are writes. Moreover, two thirds of blobs are read-only. Out of the writable blobs, 99.9% are written less than 10 times overall. Finally, the time between a write and a subsequent read to the same storage location is more than 1s in 96% of the times, and more than 10s in 27% of the times. This means that reads and writes to the same location are separated from each other, and are rarely issued in the same function invocation.

Observation 5: *FaaS functions have few types of side-effects.*

To confirm that the behavior of our application suites is representative, in this experiment, we take another 110 open-source serverless functions from various benchmark suites [11], [25], [42], [47], [84] and analyze their side effects. Our analysis shows that, in agreement with the previous sections, a major portion of the functions (63.4%) does not have any side-effects. The rest have only three types of side-effects: writes to global storage, writes to temporary local files, and HTTP requests.

Observation 6: *CPUs are not fully utilized in the cloud.*

Recent studies on resource utilization of cloud and datacenter systems [26], [28], [38], [49], [63] report that computing resources are commonly over-provisioned and not fully utilized. In this experiment, we extract from the Alibaba traces [50] the CPU utilization of the bare-metal nodes in the Alibaba cloud. For each node, we compute the P90 CPU utilization—i.e., the utilization U such that, for 90% of the time, the CPU is utilized U or less. We then generate the P90 distribution for all the nodes in the cluster. Figure 4 shows the CDFs for P90, P80, P70, P60, and P50. We can see that, most of the time, the CPU usage is 60-80%. Thus, the environment could support some cycles wasted due to misspeculation.

IV. SpecFaaS Overview

With SpecFaaS, we propose a new approach to accelerate serverless applications that are composed of multiple functions. The idea is to optimistically execute functions speculatively, before their control or data dependences are resolved. Later, if a mis-speculation is detected, the optimistically-executed functions are squashed and potentially re-started.

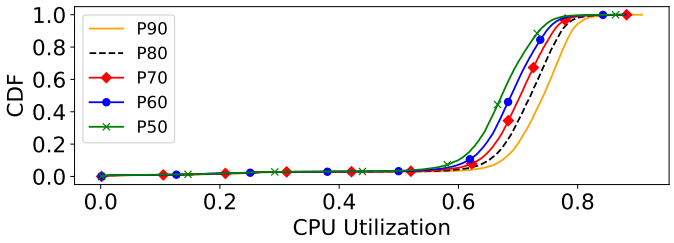

Fig. 4: P50-P90 CPU utilization of Alibaba's bare-metal nodes.

SpecFaaS' rationale is our observations that the sequence of functions executed in an application is highly predictable and that individual functions, when passed the same inputs, typically produce the same outputs. As a result, with SpecFaaS, we can substantially reduce the end-to-end execution time of serverless applications.

SpecFaaS is inspired by the out-of-order execution of instructions in modern processors—a function in SpecFaaS is analogous to an instruction in processors. Consider Figure 5(a), which shows one possible conventional execution of the smart-home application of Figure 1. All functions are executed in sequence. There are two types of cross-function dependences that prevent any concurrent execution of functions: control (e.g., between *Login* and *ReadTemp*) and data (e.g., between *ReadTemp* and *Normalize*). If SpecFaaS correctly predicts the cross-function control dependences, the execution timeline will look like Figure 5(b); if SpecFaaS correctly predicts both control and data dependences, the timeline will look like Figure 5(c).

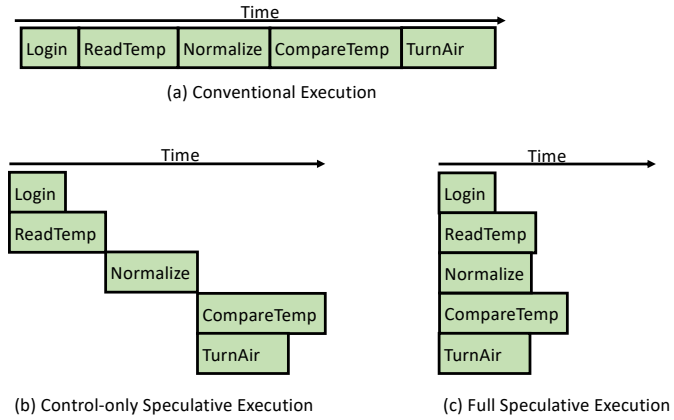

Fig. 5: Possible execution of the smart-home application of Figure 1: conventionally (a) and with speculation (b and c).

To predict control dependences, SpecFaaS adds to the FaaS controller a *Branch Predictor* software table with an entry for each function that may invoke different functions. The entry contains probability information to decide on the next function to invoke. When a function is about to execute, the branch predictor is queried. If an entry is found, the outcome with the highest probability is identified and the corresponding function is also invoked, speculatively.

To predict data dependences, the FaaS controller is augmented with a *Memoization* software table for each function that creates outputs. For a given function, the table contains the

pairs of {input, output} values that the function has taken and produced, respectively, in the past. When a function is about to execute with certain input values, its memoization table is queried. If an entry with such input values is found, the corresponding output values are retrieved and the next function of the application in sequence is concurrently invoked, speculatively, taking the retrieved values as inputs.

A function may read data values beyond those that it explicitly takes as inputs. Moreover, it may write other variables beyond those declared as outputs. Consequently, as a function executes speculatively, its global writes are buffered in a *Data Buffer* and not merged with the global state until the function's speculative execution is validated. The Data Buffer is shared by all the concurrently-running functions of a given application invocation. Similarly, global reads first access the Data Buffer, to check if the Data Buffer contains the desired data, as the current function or earlier, less-speculative functions may have generated it. The Data Buffer is also used to detect data dependence violations—i.e., a speculative function that has read data that is later updated by a predecessor function.

On a misspeculation, the FaaS controller squashes the mispredicted function and all its successors, and invalidates the corresponding data in the Data Buffer. In case of a control misspeculation, the controller then launches the correct-path functions (Figure 6(a)); in a data misspeculation, the controller re-launches the successor functions, now with the correct input values and correct Data Buffer state (Figure 6(b)).

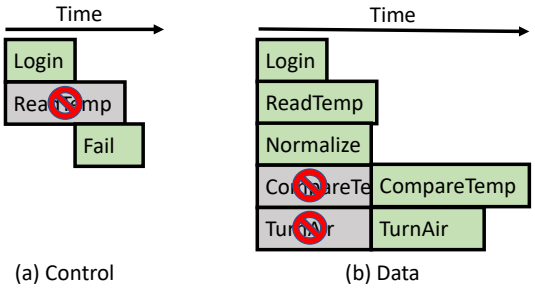

Fig. 6: Control (a) and data (b) misspeculations.

To reduce execution overheads, SpecFaaS also keeps in the FaaS controller node a software table with the sequence of functions that the application needs to execute. This *Sequence Table* is created at the application compile time. For each function, it records the next function to execute. For functions that may invoke one of multiple possible functions, the entry incorporates the branch predictor entry described above. The Sequence table allows the controller to immediately identify the next function to call without the need to invoke a component like the Conductor in OpenWhisk, hence minimizing the Transfer Function Overhead (§ III).

SpecFaaS is a generic design for modern serverless infrastructures. It works with both implicit and explicit application workflows (§ II). Moreover, while in this paper it targets warmed-up environments, it is also effective if the FaaS infrastructure is not equipped with any of the previously-proposed optimizations that remove the function cold-start overheads shown in Figure 3. Indeed, consider Figures 5(a) and 5(c). Each of the bricks shown may or may not include the function start-up overheads; in either case, SpecFaaS can reduce the execution time of the application substantially. In this paper, we will assume by default that all the function start-up overheads have been removed by previously-proposed optimizations. Finally, while we implement SpecFaaS in the OpenWhisk serverless framework [18], it can be implemented in other frameworks as well.

V. SpecFaaS Design

Figure 7 shows an overview of the SpecFaaS system. For a given application, the controller node maintains a software structure representing the pipeline of not-yet-committed functions of the application (*Function Execution Pipeline*). Functions are ordered in program order and tagged based on whether they are speculative or not, and whether they have completed or not. In addition, the controller keeps the application's Sequence table (which includes the Branch Predictor), the application's Data Buffer and, for each function in the application, the Memoization table. The Sequence and Memoization tables can remain in the controller across invocations of the application and, at every invocation of the application, are augmented with more execution information.

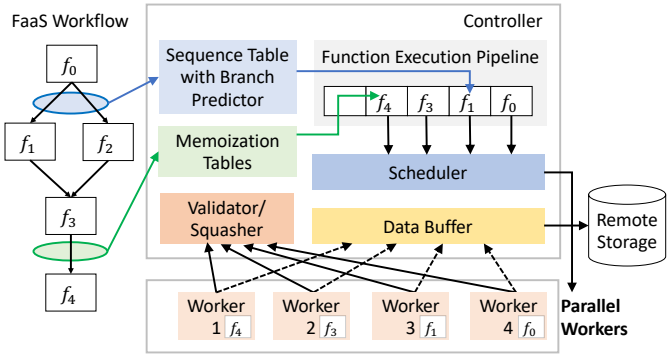

Fig. 7: Overview of the SpecFaaS system.

During execution, the controller repeatedly picks the next function to execute (speculatively or not) from the Sequence table, launches it in one of the nodes, and detects any misspeculations. If a function misspeculates, the controller squashes and potentially re-launches the function and its successors; otherwise, it eventually commits the function. In the process, the controller updates the application's Function Execution Pipeline, Sequence table, Data Buffer, and Memoization tables. We now describe the main structures.

A. Sequence Table and Branch Prediction

The Sequence table lists the ordered sequence of functions to be executed—like the sequence of instructions in a program. It enables the controller to pick the next function to launch with minimal overhead. However, like programs, FaaS applications have branches that SpecFaaS wants to predict in advance. Hence, the entries of the Sequence table for functions with branches are augmented with a branch predictor entry.

Figure 8(a) shows an application where function f_2 can be followed by either f_3 or f_6. Figure 8(b) shows the application's

Sequence table. The entry for f_2 includes a pointer to the entry for f_6 and a branch predictor entry. The latter contains state that determines whether the controller should take the branch to f_6 or just proceed to the next entry, which is f_3's.

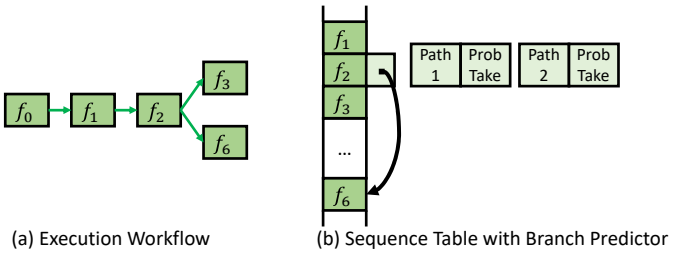

Fig. 8: Example of execution workflow (a) and corresponding Sequence table with Branch Predictor (b).

The branch predictor entry contains the probability to take the branch. Such probability is obtained by recording the outcome of previous invocations of f_2. If the probability is higher than a threshold, the controller will speculatively launch f_6; if it is lower than another, lower threshold, it will speculatively launch f_3. In our case, we find that branches in FaaS applications are highly biased. More specifically, we find that the *path* of functions executed from the beginning of the application until the branch typically determines the branch outcome. For example, it may be that if f_2 is reached from f_0 and f_1, the branch is typically taken, but if reached from f_0 and f_{11} (not shown in the picture), it is typically not taken. Hence, as shown in Figure 8(b), a branch prediction entry has a sub-entry for each possible path that reaches f_2.

If the branch at a given function *f* can have *N* targets, then the Sequence table entry for *f* has *N*-1 pointers and a branch predictor entry with the probabilities of invoking each of the targets (for each path reaching *f*). While OpenWhisk does not support more than two targets, other frameworks do [14], [35].

The controller keeps a record of the sequence of functions executed so far. On a branch, it checks the appropriate predictor and decides which speculative path to follow. Once the branch resolves non-speculatively, the controller updates the predictor and, if the prediction was incorrect, squashes the mispredicted functions and executes the correct path.

B. Function Memoization

A function often produces outputs that are consumed as inputs by its successor function. To speed-up execution, SpecFaaS executes the successor function speculatively without waiting for the predecessor function to complete. To accomplish this, the controller maintains a per-function Memoization table. The table contains input values that a function has taken-in in the past, and the corresponding output values that the function has produced. With this support, when the controller launches a function f with a given set of inputs, it checks f's Memoization table. If the specific input values are found, the controller retrieves the output values from the table and speculatively launches the successor function of f passing the retrieved values as inputs. When f commits, the controller validates f's execution.

Note that, even if f is invoked with inputs present in its Memoization table, we still need to execute *f*. This is because many FaaS functions also read and write global (i.e., non-private) data beyond their inputs and outputs. As a result, f may read global data that causes f to produce unexpected outputs. Further, f may issue updates to global state, which cannot be skipped. In addition, f may get squashed before completing execution. For example, the Data Buffer may detect a dependence violation (§V-C), where f reads a global variable that is later written by a predecessor of *f*. Because of all these cases, f must execute and validate at its commit time that it produces the expected outputs. If f generates unexpected outputs, when f commits, f's Memoization table is updated.

Some FaaS functions are *pure*, meaning that they do not read or write any global state. Therefore, the inputs that they take fully determine the outputs that they will generate, and they do not have side effects. To speed-up execution, SpecFaaS allows programmers to declare pure functions using the `pure-function` annotation (§VI). When the controller is about to launch a function and finds from its Memoization table that the function is pure, it skips the execution of the function and launches the successor function with the outputs retrieved from the table.

If a function takes input values that are not yet in its Memoization table, the successor is not launched until the predecessor completes. Later, when the predecessor completes and commits, the pair of input-output values are saved in a new row of the Memoization table.

Our measurements of real-world datasets show that Memoization table sizes are relatively modest. The combined tables for all the functions in one application use 100 to 1K entries, which consume 1.5KB to 30KB.

C. Data Buffering

In serverless environments, nodes may have software caches that temporarily store remote data accessed by a locally-running function [43], [62], [65], [74], [82]. While different designs are possible, the goal of the caches is for functions to be able to re-access previously-accessed data with low latency.

In SpecFaaS, because some functions are executed speculatively, we need one additional level of data buffering per application invocation. A new buffer, called the Data Buffer, exists in the node running the controller for the application invocation. The Data Buffer can receive requests from all the nodes that are currently running functions of the application invocation. Its goal is to detect and manage data dependences between two concurrently-executing functions of the application: (i) a non-speculative and a speculative function, or (ii) a speculative and a more speculative function. If an in-order RAW dependence is detected, the requested data is forwarded from the Data Buffer to the node running the successor function; if the RAW dependence is out-of-order, the controller sends a squash signal to the successor function (and to the successor's successors, recursively). The Data Buffer is also able to manage in-order and out-of-order WAW and WAR dependences without squashes.

At a high level, the Data Buffer is used as follows. When a function updates a record, the runtime system, in addition to updating the local cache, it sends the update to the Data Buffer. The Data Buffer stores the update and checks if any successor function has prematurely read the record. If so, the successor function (and its successors) are squashed and re-started.

When a function reads a record that the function has not accessed before, the runtime system sends the request to the Data Buffer, which provides the record. The record is also stored in the local cache. Note that the Data Buffer is only accessed if the read is *exposed*—i.e., the function has not yet read or written the record. If the read is not exposed, the read gets the data from the local cache.

In more detail, the Data Buffer is a table with a row for each record accessed by the in-progress functions of the application—i.e., the functions that are being executed or that have just executed but not yet committed. Each row has the address of the record plus a circular buffer with as many columns as the maximum number of in-progress functions supported, ordered by their program order. Each column has Valid (V), Read (R), and Write (W) bits, and space to store the updated record. Figure 9 shows a Data Buffer with two rows and three columns per row. Assume that the non-speculative function uses the leftmost column and, as successor functions are executed speculatively, they take the second and third columns in order. When a write or a read to a record from function *i* reaches the Data Buffer, the controller accesses the row for the record and performs the following operations.

Address	*Function* $i-1$				*Function* i				*Function* $i+1$			
	V	R	W	Data	V	R	W	Data	V	R	W	Data
Record 1	1		1	Value 1								
Record 2									1	1		Value 2

Fig. 9: Data Buffer for an application invocation. Each row corresponds to a record, and each column to an in-progress function of the application invocation.

Write Operation. The controller scans the R bit of all the successor functions of *i* in order. The scanning ends at (and includes) the first column that has the W bit set. If any column has the R bit set, the controller sends a squash message to the corresponding function and all its successors. Then, the columns for the squashed functions are invalidated and re-assigned to a new speculative execution of the squashed functions. Moreover, the local cache of the squashed functions is invalidated. The updated record is stored in Function *i*'s column and the W bit is set.

Read Operation. The controller scans the W bit of all the predecessor functions of *i* in reverse order. As soon as a set W bit is found, the data in that column is read and provided to the requester. If no W bit is set, the data is requested from the global storage and provided to the requester. In either case, the R bit in Function *i*'s column is set.

When a Function *i* that executes non-speculatively completes, it can commit. In the Data Buffer, committing involves writing back to global storage the records in Column *i* that have the W bit set, clearing out Column *i*, and re-assigning it to a new, most speculative function. The immediate successor function becomes non-speculative.

As an example, consider Figure 9. If Function *i* issues a write to *Record 2*, an out-of-order RAW dependence is detected and the controller will squash Function *i+1*. If Function *i* issues a read to *Record 1*, an in-order RAW dependence is detected and the Data Buffer will provide *Value 1* generated by Function *i-1*.

The logic described seamlessly handles WAR ($R_1 \rightarrow W_2$) and WAW ($W_1 \rightarrow W_2$) dependences. Indeed, assume out-of-order dependences: W_2 occurs first; later, when the other access (R_1 or W_1) occurs, it will neither read from W_2 nor squash W_2's function. Assume now in-order dependences: when W_2 occurs last, it affects neither R_1 nor W_1.

Minimizing the frequency of squashes. It is possible that a communication over remote storage between two functions of the same application triggers a squash in many of the application's invocations. To avoid this case, we augment the controller as follows. When the controller detects that a function is frequently squashed due to prematurely reading a given record that a predecessor function later updates, the controller remembers the producer-consumer function pair and the record causing the data dependence. The next time that the consumer tries to read the record, if the record is not yet updated by the producer, the controller stalls the consumer. The consumer remains stalled until the producer either updates the record or completes its execution. With this support, we minimize the number of squash operations.

D. Speculating Implicit Workflows

The presence of applications with implicit workflows requires some extensions to SpecFaaS. To describe them, we use as an example the workflow of Figure 10(a), where function f_1 can call subroutine functions f_2 and f_3.

As indicated in § II-A, FaaS frameworks do not typically know the internals of functions and, therefore, do not know the static call graph of implicit workflows. Consequently, SpecFaaS may be able to augment the Sequence table, Data Buffer, and Memoization tables with information for f_2 and f_3 *only* after one or more dynamic invocations of f_1. After those, the structures are augmented as follows.

The Sequence table is augmented to support implicit workflows as shown in Figure 10(b). The entry for f_1 has as many pointers as functions it can call. The pointers have a Call (C) bit set, to indicate that this is a call. In the example, the f_1 entry has pointers to the entries for f_2 and f_3. Each of these entries has a Return (R) bit that, when set, tells the controller to return to the caller on completion. In addition, the entry for f_1 has one Branch Predictor entry for each of the functions it can call. As in Figure 8, a Branch Predictor entry has as many sub-entries as possible paths that may reach f_1. With this design, when the controller launches f_1, it checks the two Branch Predictor entries and decides whether to speculatively launch any combination of f_2 and f_3—possibly in addition to the function that follows f_1 in sequence.

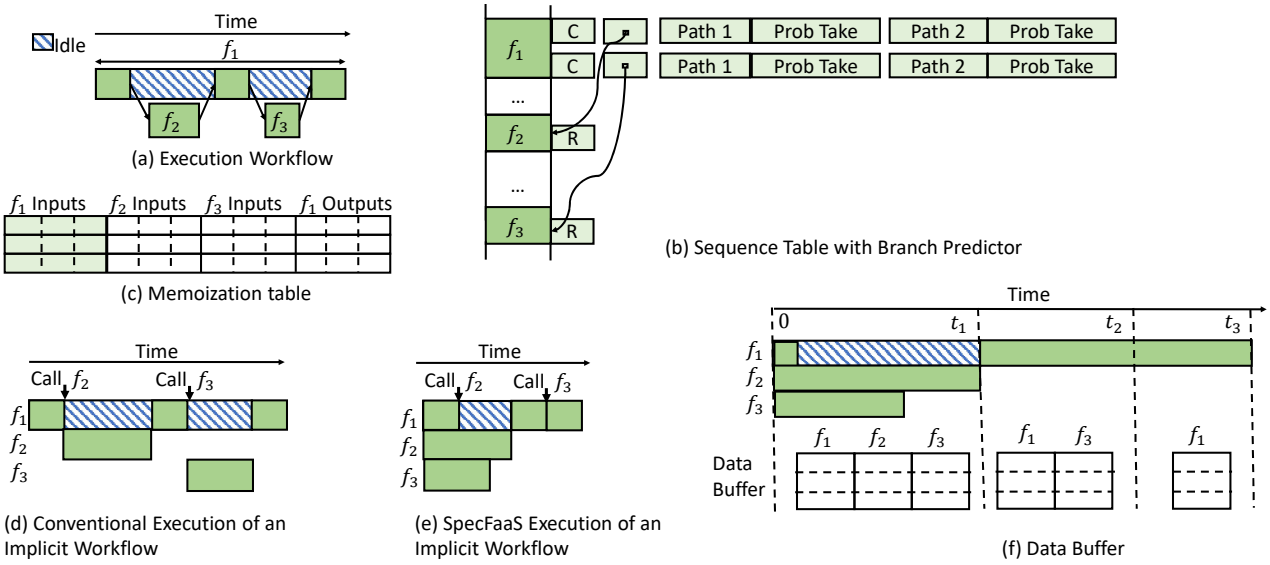

Fig. 10: Extensions to the SpecFaaS structures to support implicit workflows.

The Memoization table of f_1 is also augmented to support implicit workflows. As shown in Figure 10(c), each row of the f_1 table contains, in addition to the usual f_1 input and output values, the expected input values for f_2 and f_3. With this support, when the controller launches f_1 (speculatively or otherwise) with the input values from one row of the Memoization table, it can also speculatively launch f_2 and f_3 with their corresponding input values from the same row.

With these structures, we can now examine the execution of the f_1, f_2, and f_3 functions. Figure 10(d) shows the conventional execution of this implicit workflow, assuming that both f_2 and f_3 are executed and that we have three cores (one for each row of Figure 10(d)). We see that f_1 stalls while f_2 and f_3 execute, keeping the core idle. In contrast, Figure 10(e) shows the execution under SpecFaaS. The three functions execute concurrently, with at least f_2 and f_3 executing speculatively. When the core running f_1 reaches the point where it needs to call f_2, if f_2 has not yet completed, it stalls until f_2 completes; then, it resumes executing f_1. We choose to stall because the continuation of f_1 is often data-dependent of f_2 and, if these dependences are violated, the whole f_1 has to be squashed. The same policy is used for f_3. However, as shown in Figure 10(e), when f_1 reaches the point of calling f_3, f_3 has already completed. Therefore, f_1 does not need to stall.

Our policy of stalling the caller until the callee has finished executing speculatively is also key to simplifying the operation of the Data Buffer. With this policy, before a function f calls a subroutine s, both f and s have columns in the Data Buffer and f is a predecessor of s. When s returns, the Data Buffer column of s is merged into the column for f.

Figure 10(f) shows the Data Buffer for our example workflow. Between times 0 and t_1 (when f_2 completes), the Data Buffer has columns for f_1, f_2, and f_3—as usual, columns to the right are more speculative. At t_1, the f_2 column is merged into f_1's. Between t_1 and t_2 (when f_1 is about to call f_3 and finds that f_3 has already completed), the Data Buffer has two columns. At t_2, the f_3 column is merged into f_1's. At any time, reads and writes always manipulate the Data Buffer with the algorithm of §V-C.

E. Implications on Scalability and Security

SpecFaaS does not introduce scalability bottlenecks. In particular, the controller is not a bottleneck, as a machine has many independent controllers spread across different nodes—each controller managing a set of invocations of the same or different applications. The same is true for the controllers in current systems.

In current systems, the controller for an application invocation already keeps track of the function chain's progress and receives results from all the functions in the chain. With SpecFaaS, the controller additionally keeps the sequence table with the branch predictor, memoization tables, and Data Buffer for the chain. These structures are relatively small. The sequence table with the branch predictor and the memoization tables are kept on a per-application basis, the Data Buffer is kept on a per application invocation basis. The Data Buffer is destroyed at the end of the application invocation.

SpecFaaS can guarantee that executing and squashing speculative functions does not result in information leakage as follows. First, structures like branch predictors and memoization tables are never updated with speculative data. Second, on a squash, any software structure that holds speculative state (e.g., the Data Buffer) is invalidated. Finally, any state that a speculative function loads into microarchitectural processor structures (e.g., caches or TLBs) can be handled with known defenses (e.g., partitioning the structure, flushing the structure on context-switch or squash, etc.). We implement the first two but not the last one. The reason is that, for the microarchitecture to be safe from speculative attacks, one would also have to implement known defenses against branch miss-speculation attacks (e.g., Spectre), which would slow-down both SpecFaaS and the baseline configuration substantially.

VI. SpecFaaS Implementation Aspects

We have implemented SpecFaaS on top of Apache OpenWhisk [18], a serverless cloud platform. SpecFaaS runs transparently to serverless applications. We modify the OpenWhisk controller to support speculative execution, the software structures required, and the function runtime to intercept remote storage operations. The implementation takes about 1K lines of Scala and 500 lines of Python. In this section, we describe some of the implementation challenges.

Minimizing Squash Cost. When SpecFaaS detects a misspeculation or dependence violation, it needs to squash the incorrectly-executed functions. There are three ways in which SpecFaaS can implement the squashing mechanism. First, SpecFaaS could allow the squashed functions to complete their execution in the background but never propagate the functions' global state updates. This approach allows container reuse for subsequent invocations but wastes CPU cycles. Second, SpecFaaS could squash the functions by terminating the containers as soon as the mis-speculation or violation is detected. This approach frees the CPUs from useless execution, but it takes a long time to stop a container ($\approx$10s). Moreover, this approach disallows the reuse of the containers for subsequent invocations. Finally, our chosen mechanism is to kill only the processes executing the functions inside the containers while leaving the containers alive. This approach frees CPU cycles, is very fast ($\approx$1ms), and allows safe container reuse for subsequent invocations.

We modify the function runtime. Instead of having a single process perform the function initialization and serve all invocation requests, the SpecFaaS runtime has two types of processes: the *Initializer* process that performs the function initialization and waits on new requests, and *Handler* processes that serve requests. When the initializer receives a request, it forks a handler process and forwards the input values to it. The handler executes the function, returns the results, and dies. Every request is served by a new handler process. In this way, squashing a function is cheap, as it only involves terminating the handler process. The container and its initializer process remain active and can serve subsequent requests.

Side-effect Handling. As observed in §III, FaaS functions have three common types of side effects: writes to global storage, writes to temporary local files, and HTTP requests. SpecFaaS handles writes to global storage with the Data Buffer mechanism (§V-C). Writes to temporary local files are handled by a scheme similar to copy-on-write for memory pages. As long as a handler process only reads from a file, it can use the shared initial file. However, when it tries to update the file, it creates its own temporary file, with a unique name. From this moment on, all reads and writes by the handler are directed to the new temporary file. When the handler completes execution, all of its temporary files are discarded. We implement this functionality by intercepting file-related syscalls and redirecting the operations to the appropriate files.

SpecFaaS detects when a speculative function tries to issue an HTTP request that is not a function call or a storage access. In this case, it delays the operation until the function turns non-speculative. This is implemented by intercepting *sendto* socket syscalls and checking flags to see the origin of the call and whether the caller function is speculative. When a stalled speculative function turns non-speculative, if the speculation is validated, the *sendto* operation in performed; otherwise the function is squashed without performing the *sendto* operation.

In the applications we measure in this paper, functions do not have other side effects. However, SpecFaaS handles a wide range of other potential side effects. It handles them in a manner similar to how it handles the *sendto* operation. Specifically, all globally-visible side effects need to invoke syscalls. Some syscalls are safe, while others are not. For example, the getters syscalls (i.e., get PID, UID, time, capabilities, file-status, etc.) account for about 50 syscalls and are safe. SpecFaaS maintains a list of unsafe syscalls and, whenever a speculative function issues one such call, SpecFaaS intercepts it and suspends the function until the function turns non-speculative.

Storage Request Interception. SpecFaaS is transparent to the application. Its runtime intercepts read and write operations issued by functions to the global storage. Specifically, we modify the function runtime to intercept the `get` and `set` operations to the Redis [5] key-value store. The runtime redirects operations first to the OpenWhisk controller, which may update the Data Buffer. When a function commits, its buffered updates are flushed to global storage; when a function is squashed, its buffered updates are discarded. Given that the key-value interface is the main storage interface for FaaS [32], [33], [43], [70], [82], our interception is generically applicable to other storage services.

Function Annotations. SpecFaaS gives programmers the option to specify custom speculation policies for functions in a workflow. We implement this capability with OpenWhisk's function annotation support [16]. SpecFaaS currently supports two annotations. The first one is `non-speculative`, which specifies that the function should not be executed speculatively. In this case, the controller does not launch the function until all its predecessor functions are committed. We use this annotation when we know that the function has dependences that would typically induce squashes. The second annotation is `pure-function`, which specifies that the function is pure and, therefore, the controller should skip its execution if it finds a matching input in the function's Memoization table.

Configurability. SpecFaaS provides configurations to tune the level of speculation based on the workload type and the load of the machine. First, SpecFaaS does not perform branch speculation for branches whose current probability of being taken is within a configurable, short range around 50%. Second, SpecFaaS reduces the depth of speculation (i.e., the number of functions that are speculative at a time) to a configured threshold when the load of the machine is above a certain other threshold.

VII. Experimental Methodology

Platform. We run our experiments on five AMD EPYC 7402P servers. Each server has one socket with 24 cores (each 2-way

TABLE II: Applications used in the evaluation.

FaaSChain – *6 real-world FaaS applications with explicit workflows*	
Login	Login user and send profile info (3 functions)
Banking [13]	Withdraw money from account (6 functions)
FlightBook [12]	Book flight, hotel and car for trip (11 functions)
HotelBook [33]	Book the best available nearest hotel (7 functions)
SmartH [84]	Turn A/C if temp is above threshold (7 functions)
OnlPurch [77]	Buy an item from the closest store (13 functions)
TrainTicket – *5 open-source FaaS applications with implicit workflows*	
TcktApp	Get all tickets for a given trip (15 functions)
TripInfApp	Get information about the trip (24 functions)
QueryTrvl	Get travel-specific information (8 functions)
GetLeftApp	Get unsold tickets for given time frame (5 functions)
CancelApp	Cancel an order (4 functions)
Alibaba – *5 implicit workflow applications from production-level traces*	

multi-threaded), a 128MB Last Level Cache (LLC) and 128GB of DRAM. The OS is Ubuntu 20.04.2 LTS.

Our evaluation focuses on warmed-up scenarios, where functions and containers are available in main memory. Before starting our measurements, we first run the functions, and do not evict any containers from memory—our servers have sufficient memory to host all the containers in memory. Therefore, our evaluation does not count cold-start effects, which have been the main focus of much prior work (e.g., [8], [32], [57], [60], [65], [73], [80], [83], [86]). Note that SpecFaaS is effective in both warmed-up and cold-start scenarios (§IV).

We set the following values for the configurable parameters. First, a RAW dependence that causes the squash of a given function three times in a row triggers the controller to stall the function. Second, branches whose current probability of being taken is 60% or higher, 40% or lower, or between 40-60% are speculatively taken, speculatively not taken, and not speculated, respectively. Finally, for machine loads between 60-70%, between 70-80%, and above 80%, the maximum speculation depth is 6, 5, and 4 functions, respectively.

Application Suites. To comprehensively evaluate SpecFaaS, we use three application suites: *FaaSChain*, *TrainTicket* [87], and *Alibaba* [50]. Table II lists the applications in each suite, and Table I characterizes these suites.

FaaSChain. We developed this new FaaS application suite that has six real-world FaaS applications. The applications have different characteristics, with chain lengths varying from 2 to 10. Functions are implemented in Python and use explicit workflow, following the best practices in AWS [10].

TrainTicket. We select five representative applications from the serverless TrainTicket suite [31]. Applications are composed with implicit workflows, mainly because the code is ported from a microservice-based implementation.

Alibaba. We select five representative implicit application workflows from Alibaba's production microservice traces [50]. Traces provide the call graphs of each application (from which we infer the workflow) and the execution time of each function of the application. However, the function code is unavailable. For space reasons, in the evaluation, we show data only for the average across the five applications.

Application Input Data Sets. For *FaaSChain*, we use real-word data sets from repositories in the web [15], [58], [79]. In some cases, the data set does not provide enough information to determine the outcomes of control dependences, such as for login information in the workflow of Figure 1. In such cases, we use synthetic data for the outcomes of branches. Specifically, we assume a 90% hit rate for the branch predictor, which is the average hit rate observed in Alibaba's traces. We discuss the impact of branch prediction accuracy in §VIII-E.

For *TrainTicket*, we lack a sizable input data set of train tickets. Hence, as a high-fidelity input set, we use a real-world dataset of three million airline tickets purchased in 2021 from the Bureau of Transportation Statistics [23].

Like in prior research on serverless systems [8], [20], [36], [68], [71], [78], [86], we use the Poisson distribution to model the request inter-arrival time. In the evaluation, we use *Low*, *Medium*, and *High*, to refer to load levels of 100, 250 and 500 application requests per second (RPS), respectively.

VIII. Evaluation

A. Response Time and Speedups

We measure the end-to-end response time of applications, from the moment the client sends a request to the point it receives the result. Using this response time, we compute the speedup of SpecFaaS over the OpenWhisk baseline. Recall that all our experiments are performed under warmed-up conditions. Figure 11 shows the average speedup of each application for different loads. The figure also shows bars for the average of each application suite.

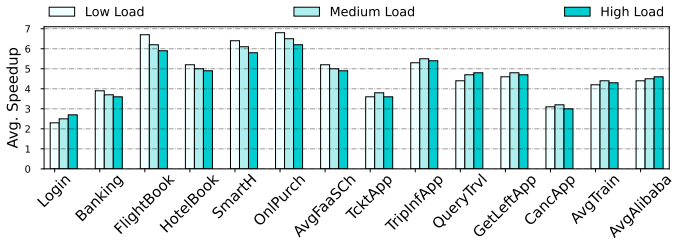

Fig. 11: Speedup of SpecFaaS over the baseline for different request loads.

SpecFaaS effectively reduces the response times and, as a result, delivers speedups across all applications and load levels. On average, the speedup is 4.6×.

Consider the FaaSChain applications. On average, SpecFaaS delivers speedups of 5.2×, 5.0×, and 4.9× in low, medium, and high load, respectively. Typically, the speedups slightly decrease with higher load because of the higher resource utilization induced by parallel function execution in SpecFaaS. However, for short-running applications like Login, speedups increase. The reason is that, as the load increases, Platform and Transfer Function overheads (§III) account for a larger fraction of the execution time, and SpecFaaS reduces them.

The TrainTicket and Alibaba applications show similar speedups. For TrainTicket, SpecFaaS attains average speedups of 4.2×, 4.4×, and 4.3× in low, medium, and high loads. For Alibaba the average speedups are 4.4×, 4.5×, and 4.6×.

As indicated in §IV, SpecFaaS is effective in both cold and warmed-up environments. We repeat the experiments in

Figure 11 without warming-up the environment and see similar average speedups across all loads: 5.2×, 4.5×, and 4.7× for FaaSChain, TrainTicket, and Alibaba, respectively.

B. Speedup Breakdown

We attribute the speedups of SpecFaaS to three main components: branch prediction, data memoization, and squash optimization. Squash optimization was described in Section VI: rather that using the naive approach of terminating the containers on mis-speculation or violation (second approach in that section), SpecFaaS terminates processes but not containers. To assess the impact of each component, we apply one technique at a time. Figure 12 shows the speedups for the different applications averaged across all loads. We apply, in order and cumulatively, branch prediction, data memoization, and squash optimization.

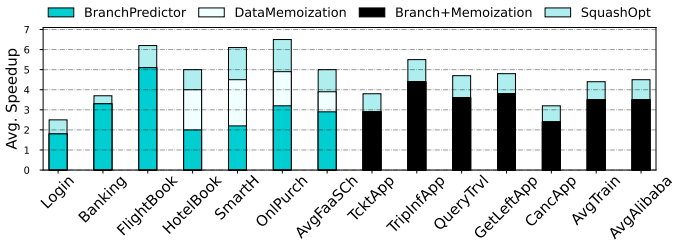

Fig. 12: Breakdown of SpecFaaS speedups.

Three of the FaaSChain applications (Login, Banking, Fight Booking) do not have data dependences; consequently, their bars have only two categories. Moreover, for implicit workflows, as used in TrainTicket and Alibaba, the branch predictor and the memoization table optimizations cannot work without each other. First, without branch prediction as shown in Figure 10(b), we cannot even build a memoization table, like the one in Figure 10(c). Second, consider a function *f* that calls subroutines. Without a memoization table for *f*, branch prediction cannot help because we cannot speculatively call the subroutines since we do not know their inputs. In contrast, in explicit workflows, if *f* finishes with a branch, we know the input of its successor functions because they take the same input as *f*. Therefore, for TrainTicket and Alibaba, a single category combines branch prediction and memoization.

The figure shows that all three techniques contribute substantially to the speedups. Consider the branch predictor first. Without it, SpecFaaS would not be able to speculate on control dependences. With it, SpecFaaS attains an average speedup of 2.9× in FaaSChain. The branch predictor of SpecFaaS obtains an average branch prediction hit rate of 98% and 90% in TrainTicket and Alibaba, respectively. For FaaSChain, we assume a 90% hit rate (§VII).

Consider now memoization. Without it, SpecFaaS would not be able to speculate on data dependences. With it, SpecFaaS attains substantial additional speedups. The combination of branch prediction and memoization delivers average speedups of 3.9×, 3.5×, and 3.5× for FaaSChain, TrainTicket, and Alibaba, respectively. A modest-sized memoization table suffices. For example, a 50-entry memoization table obtains an average 96% hit rate in TrainTicket applications. In FaaSChain applications, which vary more, the average hit rate with 50 entries ranges from 65% to 98%. In addition, many functions are pure and, therefore, SpecFaaS could use the memoization table to avoid executing them completely. For example, this is the case for more than 57.6% of the function invocations in TrainTicket. However, to be conservative in the evaluation, we do not perform this additional optimization. The Data Buffer size is also modest: it has at most 12 columns and 4 rows, for a total of 3KB.

Finally, the squash optimization is also effective. By applying it on top of the other two optimizations, SpecFaaS attains average speedups of 5.0×, 4.4×, and 4.5× for FaaSChain, TrainTicket and Alibaba, respectively.

C. Throughput Improvement

We define effective throughput as the maximum number of requests per second that are serviced without QoS violation. A QoS violation occurs when the average response time of the requests is more than 2× the response time when the system only serves one single request. Table III shows the average effective throughput of each application suite for baseline and SpecFaaS, and the improvement obtained by SpecFaaS. From the table, we observe that SpecFaaS improves the effective throughput substantially. On average across application suites, SpecFaaS improves the throughput by 3.9×.

TABLE III: Effective throughput in requests per second.

Application Suite	Baseline	SpecFaaS	Improvement
FaaSChain	118.3	485.0	4.1×
TrainTicket	90.3	346.0	3.8×
Alibaba	81.6	304.2	3.7×
Average	96.7	378.4	3.9×

D. Tail Latency

SpecFaaS also reduces tail latency significantly. Figure 13 shows the P99 (99th percentile) response time of SpecFaaS normalized to the baseline for the three application suites and different loads. On average across loads, SpecFaaS reduces the tail latency by 62%, 56% and 58% for FaaSChain, TrainTicket, and Alibaba, respectively. The average tail latency reduction across loads and applications is 58.7%.

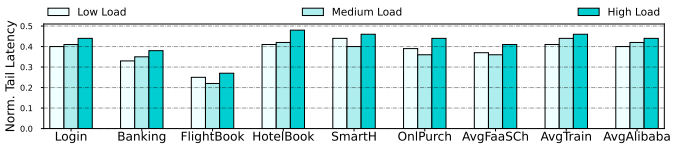

Fig. 13: Tail latency of SpecFaaS normalized to the baseline tail latency for varying system loads.

The reason for the high tail latency in baseline is the sequential allocation of resources and execution of functions, and the high transfer function overhead. The system waits for a function to complete before allocating resources for the following function. Instead, SpecFaaS allocates resources and executes functions early on.

E. Effect of Branch Prediction

We analyze the effect of branch prediction hit rates on overall performance. Figure 14 shows the speedup of SpecFaaS over the baseline for the FaaSChain applications and averaged across all loads, as we vary the branch prediction hit rates. We show data for 100%, 90%, 70%, and 50% hit rates.

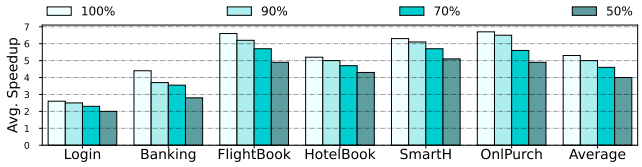

Fig. 14: Speedup of SpecFaaS over the baseline for different branch prediction hit rates.

As we decrease the hit rate from 100% (a perfect predictor) to 90% (the hit rate observed with Alibaba's trace [50]), the average speedup decreases by 5.7%. As the hit rate continues to decrease, the speedups decrease substantially. The impact of branch misprediction depends on where the branches are in the application workflow. If they are in the front-end, more work is likely to be discarded. Although not shown, the difference between the perfect and imperfect predictors becomes larger when the load is high because misspeculation is relatively more costly in such environments.

F. Resource Utilization

SpecFaaS uses more resources than the baseline because some speculative work is squashed. In this section, we measure the contribution of the squashed work to the CPU utilization. We take the FaaSChain applications and vary the hit rate of speculation (i.e., how frequently control and data speculation succeed) from 100% to 50%. Table IV shows the average CPU utilization for two speculative environments: *LazySquash* (where mis-speculated functions are allowed to complete their execution in the background before squashing) and SpecFaaS (where mis-speculated functions are immediately squashed). The CPU utilizations are normalized to the baseline, which is also shown in the table with a utilization of one. The last column of the table shows the average speedup of SpecFaaS over baseline. The data corresponds to the average of all loads.

TABLE IV: Normalized average CPU utilization for FaaSChain for different speculation hit rates.

HitRate	Baseline	LazySquash	SpecFaaS	Speedup
100%	1	1	1	5.2×
90%	**1**	**1.09**	**1.03**	**5.0×**
70%	1	1.24	1.08	4.6×
50%	1	1.43	1.15	4.0×

The 90% row is bolded because it corresponds, approximately, to the average speculation hit rate attained by SpecFaaS. We see that, for this hit rate, SpecFaaS increases CPU utilization by only 3%, while achieving a 5× speedup. This is a tolerable cost, as the CPUs are typically busy only 60-80% of the time (§III). Moreover, squashing mis-speculated functions immediately saves substantial CPU cycles.

IX. Related Work

Most prior works address performance bottlenecks when executing a *single* function. They use lightweight containers specialized for serverless environments [7], [60], snapshotting techniques to reduce VM boot time [24], [29], and techniques that provide isolation for multi-tenancy [85]. Moreover, to reduce cold start latency, serverless platforms keep function containers alive for a grace period [32], [59], [69], and researchers propose advanced techniques [6], [48], [52], [66], [80]. SpecFaaS speeds-up multi-function applications.

Function workflows have recently gained great attention. However, prior work primarily focuses on addressing the cascading cold start via proactive and just-in-time resource provisioning [20], [27], [36], [72]. These solutions bring the next function to execute into warm state and thus, can only mitigate the cold start effect. SpecFaaS addresses both cold and warmed-up invocations through function overlap.

Netherite [22] is a distributed execution engine that allows a function to pass global updates to its successor function before the updates make it to global storage—a process they call speculation. SpecFaaS uses a similar form of data forwarding in the Data Buffer, plus speculation of function execution.

SAND [8] and Faastlane [44] reduce latency and improve resource efficiency of function chaining by executing all the functions of a workflow in the same container. The techniques are orthogonal to and can be combined with SpecFaaS.

Prior work proposed different local and remote in-memory caching techniques to bring data closer to processing units in FaaS systems [43], [62], [65], [74], [82]. SpecFaaS is orthogonal to these techniques and, additionally, uses caching to buffer speculative data.

Many works proposed memoization techniques for dataflow jobs [21], [30], [37], [46], [64], [67], [81]. Contrary to SpecFaaS, these systems are not speculative. Hence, they do not need mechanisms to detect wrong memoization, squash incorrect executions, or restart from the correct state.

X. Conclusion

This paper proposes to accelerate serverless environments with a novel approach based on *speculation*. Our proposal, *SpecFaaS*, executes functions in an application early, speculatively, before their control and data dependences are resolved. The execution of downstream functions is overlapped with that of upstream ones, substantially reducing the end-to-end execution time of applications. Our evaluation on Apache OpenWhisk showed that SpecFaaS is very effective. On average, it attains an application speedup of 4.6× in a warmed-up environment. Further, the application throughput increases by 3.9× and the tail latency reduces by 58.7%.

Acknowledgments

This work was supported by the IBM-Illinois Discovery Accelerator Institute, and by NSF grant CNS-1956007.

REFERENCES

[1] "Fission: Open source Kubernetes-native Serverless Framework," https://fission.io/.

[2] "Fn Project," https://fnproject.io/.

[3] "Kubeless: The Kubernetes Native Serverless Framework," https://kubeless.io/.

[4] "OpenFaaS," https://docs.openfaas.com/.

[5] "Redis," https://redis.io/.

[6] A Cloud Guru, "How long does AWS Lambda keep your idle functions around before a cold start?" https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810.

[7] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.

[8] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[9] Amazon AWS, "AWS Lambda," https://aws.amazon.com/lambda/.

[10] Amazon AWS, "AWS Orchestration Documentation," https://docs.aws.amazon.com/lambda/latest/operatorguide/orchestration.html.

[11] Amazon AWS, "AWS Samples: AWS Serverless Workshops," https://github.com/aws-samples/aws-serverless-workshops/.

[12] Amazon AWS, "AWS Serverless Airline Booking," https://github.com/aws-samples/aws-serverless-airline-booking.

[13] Amazon AWS, "AWS Serverless Workshops," https://github.com/aws-samples/aws-serverless-workshops.

[14] Amazon AWS, "AWS Step Functions," https://aws.amazon.com/step-functions/.

[15] N. Antonio, A. de Almeida, and L. Nunes, "Hotel Booking Demand Datasets," *Data in Brief*, 2019.

[16] Apache Software Foundation, "Annotations on OpenWhisk assets," https://github.com/apache/openwhisk/blob/master/docs/annotations.md.

[17] Apache Software Foundation, "Open Whisk Composer," https://github.com/apache/openwhisk-composer-python.

[18] Apache Software Foundation, "OpenWhisk," https://openwhisk.apache.org/.

[19] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The Serverless Trilemma: Function Composition for Serverless Computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017.

[20] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[21] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proceedings of the VLDB Endowment*, 2010.

[22] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Serverless Workflows with Durable Functions and Netherite," *CoRR*, vol. abs/2103.00033, 2021. [Online]. Available: https://arxiv.org/abs/2103.00033

[23] Bureau of Transportation Statistics, "Origin and Destination Survey," https://www.transtats.bts.gov/Fields.asp?gnoyr_VQ=FHK.

[24] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip Redundant Paths to Make Serverless Fast," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, 2020.

[25] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," in *Proceedings of the 22st International Middleware Conference (Middleware '21)*, 2021.

[26] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[27] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.

[28] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014.

[29] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[30] A. Fuchs and D. Wentzlaff, "Scaling Datacenter Accelerators with Compute-Reuse Architectures," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[31] Fudan SE Lab, "Serverless Train Ticket," https://github.com/FudanSELab/serverless-trainticket.

[32] A. Fuerst and P. Sharma, "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[33] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[34] Google Cloud, "Google Cloud Functions," https://cloud.google.com/functions.

[35] Google Cloud, "Workflows," https://cloud.google.com/workflows.

[36] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling Resource Underutilization in the Serverless Era," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.

[37] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[38] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces," in *Proceedings of the International Symposium on Quality of Service (IWQoS '19)*, 2019.

[39] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

[40] IBM Cloud, "IBM Cloud Composer," https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer.

[41] IBM Cloud, "IBM Cloud Functions," https://cloud.ibm.com/functions/.

[42] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019.

[43] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[44] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service Workflows," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[45] F. Lardinois, "Platform9's Fission Workflows makes it easier to write complex serverless applications," https://techcrunch.com/2017/10/03/platform9s-fission-workflows-makes-it-easier-to-write-complex-serverless-applications/?guccounter=1, 2017.

[46] W. Lee, E. Slaughter, M. Bauer, S. Treichler, T. Warszawski, M. Garland, and A. Aiken, "Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, 2018.

[47] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[48] P. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *CoRR*, vol. abs/1903.12221, 2019. [Online]. Available: http://arxiv.org/abs/1903.12221

[49] Q. Liu and Z. Yu, "The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace," in *Proceedings of the 2020 ACM Symposium on Cloud Computing (SoCC '20)*, 2018.

[50] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[51] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[52] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *Proceedings of 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018.

[53] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, 2014.

[54] Microsoft Azure, "Azure Durable Functions," https://docs.microsoft.com/en-us/azure/azure-functions/durable//.

[55] Microsoft Azure, "Azure Public Dataset," https://github.com/Azure/AzurePublicDataset.

[56] Microsoft Azure, "Microsoft Azure Functions," https://azure.microsoft.com/en-gb/services/functions/.

[57] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*, 2019.

[58] National Centers for Environmental Information, "Weather Data," https://www.ncei.noaa.gov/pub/data/uscrn/products/subhourly01/2021/.

[59] G. Neves, "Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues," https://www.serverless.com/blog/keep-your-lambdas-warm.

[60] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[61] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.

[62] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[63] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*, 2012.

[64] H. Rito and J. Cachopo, "Memoization of Methods Using Software Transactional Memory to Track Internal State Dependencies," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*, 2010.

[65] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[66] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming Serverless Functions Better with Heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[67] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014.

[68] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO' 19)*, 2019.

[69] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*.

[70] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[71] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A Scalable Low-Latency Serverless Platform," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[72] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A Scalable Low-Latency Serverless Platform," *CoRR*, vol. abs/1911.09849, 2019. [Online]. Available: http://arxiv.org/abs/1911.09849

[73] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "BabelFish: Fusing Address Translations for Containers," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[74] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, 2020.

[75] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA '19)*, 2019.

[76] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-Optimized Architecture," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[77] O. Tardieu, "Serverless Composition of Serverless Functions," https://s3.us.cloud-object-storage.appdomain.cloud/res-files/2516-debs19.pdf.

[78] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-Service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[79] University of California Irvine - Machine Learning Repository, "Online Retail Data Set II," https://archive.ics.uci.edu/ml/datasets/Online+Retail+II.

[80] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[81] V. Vassiliadis, M. A. Johnston, and J. L. McDonagh, "Fast, Transparent, and High-Fidelity Memoization Cache-Keys for Computational Workflows," in *2022 IEEE International Conference on Services Computing (SCC '22)*, 2022.

[82] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020.

[83] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," in *Proceedings of the Fourteenth EuroSys Conference (EuroSys '19)*, 2019.

[84] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[85] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the Gap Between Serverless and Its State with Storage Functions," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*, 2019.

[86] Y. Zhang, I. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[87] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking Microservice Systems for Software Engineering Research," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, 2018.