Fine-Grained Performance and Cost Modeling and Optimization for FaaS Applications

Changyuan Lin, Nima Mahmoudi, *Student Member, IEEE*, Caixiang Fan, *Student Member, IEEE*, and Hamzeh Khazaei, *Member, IEEE*

Abstract—Function-as-a-Service (FaaS) has become a mainstream cloud computing paradigm for developers to build cloud-native applications in recent years. By taking advantage of serverless architecture, FaaS applications bring many desirable benefits, including built-in scalability, high availability, and improved cost-effectiveness. However, predictability and trade-off of performance and cost are still key pitfalls for FaaS applications due to poor infrastructure transparency and lack of performance and cost models that fit the new paradigm. In this study, we therefore fill this gap by proposing formal performance and cost modeling and optimization algorithms, which enable accurate prediction and fine-grained control over the performance and cost of FaaS applications. The proposed model and algorithms provide better predictability and trade-off of performance and cost for FaaS applications, which help developers to make informed decisions on cost reduction, performance improvement, and configuration optimization. We validate the proposed model and algorithms via extensive experiments on AWS. We show that the modeling algorithms can accurately estimate critical metrics, including response time, cost, exit status, and their distributions, regardless of the complexity and scale of the application workflow. Also, the depth-first bottleneck alleviation algorithm for trade-off analysis can effectively solve two optimization problems with fine-grained constraints.

Index Terms—Cloud serverless computing, performance modeling, performance optimization, cost modeling, cost optimization

1 Introduction

The emergence of Function-as-a-Service (FaaS) and serverless computing has transformed how developers build and manage event-driven cloud-native applications, by offloading most operational responsibilities and allowing developers to focus on business logic. Most FaaS solutions are based on lightweight virtualization techniques that provide ephemeral isolated sandboxes, such as containers, unikernels, and Firecracker VMs, which have low operating overhead, fast startup, and scaling speeds [1], [2], [3]. Therefore, serverless architecture can provide cloud-native applications with significant performance, scalability, and availability boosts over traditional monolithic architecture. FaaS providers usually leverage a pay-as-you-go, near real-time, and granular billing model that is cost-effective. Also, cloud computing providers offer various fully-managed services together with FaaS, such as logging, function orchestration, messaging, API gateways, databases, and storage, facilitating the development of FaaS applications.

Generally, a FaaS application is composed of a collection of serverless functions hosted on FaaS platforms and serverless services, which are orchestrated according to the business logic. As a result, the potential benefits of FaaS applications are similar to the serverless paradigm: low operational overhead, more focus on business logic, high scalability and availability, and flexible and granular pricing models. However, such benefits do not always hold, and pain points are swirling around FaaS applications. There is a long debate over whether serverless applications are more performant and cost-effective than applications with traditional architectures or based on other cloud services [4], [5], [6], [7], [8]. Also, a recent empirical study shows that the trade-off between performance and cost is one of the critical requirements from the stakeholders in serverless computing [9]. The lack of performance and cost models for serverless applications and optimization algorithms for performance/cost trade-off is one of the root causes of such debates [5], [8], [10], [11], [12], since developers cannot easily understand the performance and cost implications in serverless applications.

Our previous work [13] presented an analytical model to get the average end-to-end response time and cost of serverless applications and an optimization algorithm for performance/cost trade-off. The structures (i.e., parallels, branches, cycles, and self-loops) in serverless application workflow are defined on a directed graph. The proposed analytical model leverages graph algorithms to process structures in the serverless application and covert the workflow into a weighted directed acyclic graph (DAG). Then, the analytical model calculates the weighted average response time of paths in DAG and the average number of invocations of functions, from which derive the output of the model, namely the average values of end-to-end response time and cost. Also, we proposed the probability refined critical path (PRCP) algorithm, which is a heuristic algorithm based on the critical path method to give an optimal memory configuration that achieves the best performance/cost under budget/

- *Changyuan Lin and Hamzeh Khazaei are with the Department of Electrical Engineering and Computer Science, York University, Toronto, ON M3J 1P3, Canada. E-mail: {chlin, hkh}@yorku.ca.*
- *Nima Mahmoudi and Caixiang Fan are with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2R3, Canada. E-mail: {nmahmoud, caixiang}@ualberta.ca.*

Manuscript received 5 May 2022; revised 29 September 2022; accepted 2 October 2022. Date of publication 14 October 2022; date of current version 16 November 2022.
(Corresponding author: Changyuan Lin.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TPDS.2022.3214783

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

performance constraints. While the proposed model and algorithm made the first attempt to tackle the problem of performance and cost modeling and optimization for serverless applications, it has several limitations:

1) The analytical model could only predict the average end-to-end response time and cost. However, the average value may not be a good performance/cost indicator [14]. For example, developers may focus on the exit status of the application and the tail latency and high-percentile cost (e.g., 90-th percentile of response time) to satisfy the fine-grained service level agreement.
2) Similarly, the optimization algorithm solves the performance/cost trade-off based on the average end-to-end response time and cost, which may not be effective optimization constraints for fine-grained trade-off analysis.
3) The performance model could not work on the applications with dynamic parallelism introduced by map states or applications with multiple exit points. Also, the analytical model does not scale well and could not work on applications with nested cycles and unconditional jumps, limiting the types of FaaS applications that can be modeled and optimized.
4) The billing model on which the previous work is based has changed. AWS introduced a billing model with the granularity of 1ms and 1MB [15], which could considerably improve the functions' cost effectiveness but diminish the models' accuracy in our previous work.

Such limitations and evolution of the FaaS platform motivate us to revisit the performance and cost modeling and optimization of FaaS applications. In order to address the limitations, we propose the conditional stochastic Petri net (CSPN) for abstracting FaaS applications, a fine-grained performance and cost modeling algorithm, and the depth-first bottleneck alleviation (DFBA) algorithm for solving performance/cost trade-off. The main contributions of this work are as follows:

1) We propose CSPNs to effectively model concurrency, parallelism, synchronization, randomness, and billing in FaaS applications or any software systems with similar patterns. We present modeling rules to convert states, actions, and structures in a FaaS application into CSPNs. Also, we provide ready-to-use modules to help define FaaS applications with CSPNs.
2) We design a performance and cost modeling algorithm based on CSPNs sampling, which enables the fine-grained estimation of the distribution of the response time, cost, and exit status of FaaS applications.
3) We propose the DFBA algorithm that achieves fine-grained performance and cost tuning for FaaS applications by solving two optimization problems with accurate performance/cost constraints (e.g., tail latency and high-percentile cost).
4) The evaluation results demonstrate that the performance and cost modeling algorithm delivers an average accuracy of over 97%. The DFBA algorithm achieves an accuracy of over 99%, making a significant improvement over the algorithms in our previous work [13].

TABLE 1
Definition of Notations

Notation	Definition
P	a set of places; p is a place
T	a set of transitions; t is a transition;
${}^\bullet t, t^\bullet$	${}^\bullet t$ is the preset of t; $t^\bullet$ is the postset of t
I	$I : T \to \mathbb{N}^P$ maps transitions to input vectors
O	$O : T \to \mathbb{N}^P$ maps transitions to output vectors
W	$W : T \to \wp(\mathbb{R}_{\geqslant 0})$ maps transitions to sets of firing delays
Pr	$Pr : T \to [0, 1]$ is a transition probability function
U	$U : P \to \wp(\mathbb{R}_{\geqslant 0})$ defines ages of tokens in each place
$\boldsymbol{m}$	a marking of the net; $\boldsymbol{m_0}$ is the initial marking
F	a set of serverless functions, f is a function
$\mathbb{V}$	$\mathbb{V} : F \to \wp(\mathbb{N})$ maps F to sets of viable memory options
$vDom$	a set of functions and their viable memory options
RT	$RT : v\,Dom \to \wp(\mathbb{R}_{\geqslant 0})$ performance profile of functions
D	$D : F \to \wp(\mathbb{R}_{\geqslant 0})$ maps functions to sets of delays
C	$C : \mathbb{R}_{\geqslant 0} \times \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ cost of a serverless function
mem	a memory size in megabytes
K	number of iterations
BC	a budget constraint (cost per 1M executions in US dollars)
PC	a performance constraint (end-to-end response time in ms)
k	k-th percentile defined in the BPBC and BCPC problems
S	a set of structures and functions
$\wp$	power set
$\coprod$	disjoint union
$\mapsto$	map an element in the domain, defined in Equation (1)

The rest of this paper is organized as follows: Section 2 discusses preliminaries about SPNs and presents the construction of CSPNs. Section 3 discusses modeling FaaS applications with CSPNs. Section 4 presents the fine-grained performance and cost modeling algorithm. Section 5 presents the fine-grained performance and cost optimization algorithm (DFBA). Section 6 evaluates the effectiveness and accuracy of the proposed modeling and optimization algorithms. Section 7 discusses the latest literature. Section 8 summarizes and concludes the work.

2 Preliminaries

In this section, we briefly review the stochastic Petri nets (SPNs) and introduce conditional stochastic Petri nets tailored for abstracting FaaS applications. We introduce the following notations used in this paper: (1) $\mathbb{R}_{\geqslant 0}$ is the set of non-negative real numbers. (2) Unless specified, pointwise subtraction and addition between functions (vectors) are considered. (3) The symbol $\mapsto$ represents mapping an element in the function domain, defined as

$$f[x_0 \mapsto t] := \begin{cases} t & x = x_0 \\ f(x) & otherwise \end{cases} \tag{1}$$

For convenience, Table 1 summarizes notations used throughout the paper.

2.1 Stochastic Petri Nets

Formally, a stochastic Petri net is defined as a 6-tuple

$$\mathrm{SPN} = (P, T, I(\cdot), O(\cdot), W(\cdot), \boldsymbol{m_0}) \tag{2}$$

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

where P is a finite set of places, T is a finite set of transitions such that $T \cap P = \emptyset$, $I : T \rightarrow \mathbb{N}^P$ is a function mapping transitions to input vectors, $O : T \rightarrow \mathbb{N}^P$ maps transitions to output vectors, $W : T \rightarrow \mathbb{R}_{\geqslant 0}$ maps transitions to a set of non-negative real numbers representing the rate of the probabilistic firing delay of transitions, which is determined by a random variable following an exponential distribution, and $\boldsymbol{m_0} \in \mathbb{N}^P$ is the initial marking of the net.

In SPNs, a transition t is enabled by a marking $\boldsymbol{m} \in \mathbb{N}^P$ such that $\boldsymbol{m} - I(t) \geqslant \boldsymbol{0}$, where $\boldsymbol{0}$ is a constant function that maps any place to zero. When the transition t fires, the tokens specified by $I(t)$ are consumed, and those specified by $O(t)$ are produced, changing the current marking to $\boldsymbol{m} - I(t) + O(t)$. The occurrence of firing t is denoted as $\boldsymbol{m} \xrightarrow{t} \boldsymbol{m}'$, where $\boldsymbol{m}' = \boldsymbol{m} - I(t) + O(t)$. Evolved out of classical Petri nets, SPNs can model systems with concurrency and conflicts. Considering two occurrences of firing transitions under a marking $\boldsymbol{m}$, denoted as $\boldsymbol{m} \xrightarrow{t_i} \boldsymbol{m}'$ and $\boldsymbol{m} \xrightarrow{t_j} \boldsymbol{m}''$, $t_i \in T$ and $t_j \in T$ are in a concurrency relation such that

$$\begin{aligned} \boldsymbol{m}' - I(t_j) &\geqslant \boldsymbol{0} \\ \boldsymbol{m}'' - I(t_i) &\geqslant \boldsymbol{0} \end{aligned} \tag{3}$$

Transitions t_i and t_j are under conflict such that

$$\boldsymbol{m}' - I(t_j) < \boldsymbol{0} \tag{4}$$

Also, the preset of a transition t, denoted as ${}^\bullet t$, is defined as ${}^\bullet t = \{p \in P : I(t)(p) > \boldsymbol{0}\}$. The postset of t, denoted as $t^\bullet$, is defined as $t^\bullet = \{p \in P : O(t)(p) > \boldsymbol{0}\}$. We refer to places in the preset and the postset of a transition as its input and output places, respectively. Furthermore, the preset of a place p is defined as ${}^\bullet p = \{t \in T : p \in t^\bullet\}$. The postset of p is defined as $p^\bullet = \{t \in T : p \in {}^\bullet t\}$. For ease of description, if there is only one element (i.e., place or transition) in the preset or the postset of a transition or a place, we use the same symbol to denote that element.

2.2 Conditional SPNs for FaaS Applications

While SPNs are efficient in modeling performance and functionalities of distributed concurrent systems [16], they are not effective for modeling FaaS applications due to that 1) the response time of serverless functions does not necessarily follow an exponential distribution (e.g., f_3 shown in Fig. 9); 2) SPNs cannot solve the conflict introduced by the if-elif-else logic in FaaS applications; 3) SPNs cannot model the cost of FaaS applications that depends on the firing time and the probabilistic firing delay (response time).

Therefore, we construct a new model named conditional stochastic Petri nets (CSPNs) to abstract FaaS applications, or any software systems with similar execution/coordination/billing patterns, defined as an 8-tuple

$$\text{CSPN} = (P, T, I(\cdot), O(\cdot), W(\cdot), Pr(\cdot), U(\cdot), \boldsymbol{m_0}) \tag{5}$$

such that the following conditions hold true

1) $|p^\bullet| \leqslant 1 \vee ({}^\bullet(p^\bullet) = \{p\})$, for all $p \in P$
2) $\max(I(t)) \leqslant 1 \wedge \max(O(t)) \leqslant 1$, for all $t \in T$
3) $(|p^\bullet| \leqslant 1 \vee |{}^\bullet p| \leqslant 1) \wedge \sum_{t \in p^\bullet} \Pr(t) = 1$, for all $p \in P$

where $Pr : T \rightarrow [0, 1]$ is a transition probability function that identifies the probability of firing a transition under conflict, $W : T \rightarrow \wp(\mathbb{R}_{\geqslant 0})$ is a function mapping transitions to the power set of non-negative real numbers that represent the possible values of the firing delay, which are not necessarily exponentially distributed. Also, each token in CSPNs has an age that represents the amount of elapsed time on the token. The function $U : P \rightarrow \wp(\mathbb{R}_{\geqslant 0})$ defines ages of tokens in each place, such that $\boldsymbol{m}(p) = |U(p)|$ always holds.

The first condition in the definition ensures that $\forall t_i, t_j \in T$, Equations (3) and (4) cannot be simultaneously satisfied. In other words, there can be both concurrency and conflicts in CSPNs, but not at the same time. The second condition guarantees that each transition can only get or give at most one token from and to a place, whereas the third condition indicates the probabilistic firing rule under conflict. The enabling rule for a transition and the definition of conflicting transitions are defined in the same way as for SPNs.

When firing a transition in CSPNs, as defined by $I(\cdot)$ and $O(\cdot)$, tokens from input places are consumed. Then, the transition produces tokens and puts them in output places. The age of produced tokens is defined as the sum of the maximum age of consumed tokens and the firing delay. Namely, let us consider the occurrence of firing transition t, denoted as $\boldsymbol{m} \xrightarrow{t} \boldsymbol{m}'$, with a firing delay $\delta \in W(t)$. For each input place $p \in {}^\bullet t$, the transition consumes a token with the age of τ_p, that is, we have $U[p \mapsto U(p) \backslash \tau_p]$, and the set of ages of all consumed tokens is denoted as Γ. Then, for each output place $p \in t^\bullet$, a token with the age of $\max\{\Gamma\} + \delta$ is placed, namely we have $U[p \mapsto U(p) \bigcup (\max\{\Gamma\} + \delta)]$. In case there are conflicting transitions, let us consider n groups of conflicting transitions $CT = \{T_1, T_2, \ldots, T_n\}$ under a given marking $\boldsymbol{m}$, such that all transitions in the group T_i have the same preset p_i. Namely, we have

$${}^\bullet t = p_i, \forall t \in T_i, T_i \in CT$$

and

$$\boldsymbol{m}' - I(t_v) < \boldsymbol{0}, \forall t_u, t_v \in T_i$$

where $\boldsymbol{m} \xrightarrow{t_u} \boldsymbol{m}'$. In this case, only one transition can fire for each group of conflicting transitions, and the probability of selecting t from a group of conflicting transitions to fire is $Pr(t)$. Thus, the conflict is resolved in a probabilistic way.

3 Modeling FaaS Applications With CSPNs

In this section, we first introduce actions, structures (flows), and states in FaaS applications, and define rules for modeling such components using net structures in CSPNs. For each component, we give an example of applying such modeling rules to convert it into a CSPN model.

For clarity, besides those given in Section 2, we introduce the following notations and definitions for describing FaaS applications: (1) f is a serverless function, integrated cloud service, action, or state; F is the set of functions in the application; (2) the directed edge in the application workflow specifies the order of execution. For instance, $f_u \rightarrow f_v$ represents f_v will be triggered after f_u is complete; (3) $\mathbb{V} : F \rightarrow \wp(\mathbb{N})$ denotes viable memory options that are sufficiently large for each function; (4) $vDom := \coprod_{f \in F} \{\{f\} \times \mathbb{V}(f)\}$ is a

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

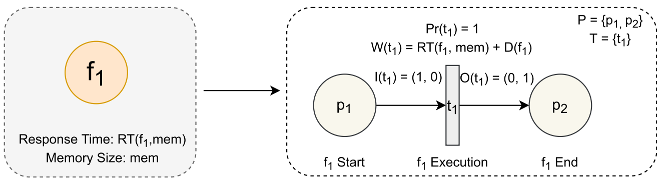


Fig. 1. Model a serverless function with CSPNs. The grey shaded box represents the original application workflow. The white box contains the converted net structures and definitions.

set of serverless functions and their corresponding viable memory options; (5) $RT : v\,Dom \rightarrow \wp(\mathbb{R}_{\geqslant 0})$ is a function mapping serverless functions and their viable memory options to a set of non-negative real numbers representing the execution duration of functions in milliseconds, and $RT(f, mem)$ is the set of the execution duration of f when allocated mem megabytes of memory. We also refer to RT as performance profile; (6) $D : F \rightarrow \wp(\mathbb{R}_{\geqslant 0})$ is a function mapping serverless functions to a set of non-negative real numbers representing the delay of functions in milliseconds.

3.1 Functions

Serverless functions are basic units of deployment and computation that abstract parts of the business logic in serverless applications. As shown in Fig. 1, a function can be modeled as a timed transition between two places that represent the start state and the end state of the function execution. The response time of a function is composed of the duration of the actual execution and the delay (e.g., latency incurred by cold start, orchestration method, or communication protocol that is typically not billed). In the CSPN model, the response time of a function is equal to the firing delay of a transition, which is determined by the sum of two random variables (actual execution duration and delay). The transition requires one token from the start place and gives one token to the end place when fired. Thus, the firing of a transition is equivalent to executing a function. Besides functions, serverless applications may have other types of actions, such as fully-managed cloud services (e.g., message queuing service and cloud transcoding task) that usually have a probabilistic firing delay, *Pass* states (pass without performing work) that is equivalent to an immediate transition, and *Wait* states (suspend execution for a given amount of time) with a deterministic firing delay. As such actions are similar to functions in terms of execution, we apply the same modeling rules to them.

For ease of modeling and optimization, when converting a function into net structures, we label the transition as a serverless function and record non-performance-related parameters, including the name and memory size. For calculating the cost of functions, we use the billing model of AWS Lambda [15] in this work. The cost of a serverless function per invocation, denoted as $C : \mathbb{R}_{\geqslant 0} \times \mathbb{N} \rightarrow \mathbb{R}_{\geqslant 0}$, can be calculated as Equation (6) for a function with the execution duration of rt milliseconds and the memory size of mem megabytes, where *PGS* (\$0.0000166667) is the price per GB-second and *PPI* (\$0.0000002) is the price per invocation. We obtain the cost from users' input for functions that represent other cloud services with different pricing models. The cost is zero for functions that represent the *Pass* or *Wait* state.

$$C(rt, mem) = \frac{\lceil rt \rceil}{1000} \cdot \frac{mem}{1024} \cdot PGS + PPI \qquad (6)$$

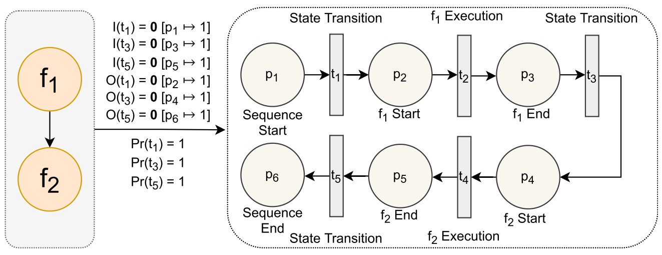


Fig. 2. Model a sequence with CSPNs.

When profiling the execution duration and the delay of a serverless function, a sequence of requests for function invocation should be constructed properly. The most common method is to generate a sequence based on the execution logs or trace files of a serverless application deployed in the production environment, which record the sequence of actual events that happen within the application, the frequency distribution of invocations for each function, and the real inputs of functions. Another method is to assume probability distributions for the input size and the invocation interval based on the execution logs or empirical knowledge and use a random generator to generate a sequence of requests that follow such distributions. Either way, we can construct a sequence of requests with valid payloads and intervals, send them to the serverless function, and collect the logs to obtain the performance profile of a function.

3.2 Sequences

When multiple functions are chained together and executed in a row, and each function takes the output of the previous function as its input, these functions are constructed as a sequence. Fig. 2 demonstrates the modeling rule of a sequence with two functions, where each function follows the conversion rules presented in Section 3.1. The interaction between functions is equivalent to the state transition connecting the end place of the previous function to the start place of the next function. Since the interaction may have a delay due to communication latency depending on the orchestration method, the state transition may also have a probabilistic firing delay. Also, the state transitions between functions/structures may incur fixed or variable costs that are subject to the billing model of the serverless application orchestration service [17]. Besides functions, sequences may consist of other structures, such as parallels, choices, and maps, which will be discussed in the following subsections.

3.3 Parallels

Parallels introduce parallelism (fan-out pattern) into serverless applications. The parallel is a structure that allows functions (actions) in multiple sequences to be executed in parallel. When executing a parallel structure, the first action of each parallel branch will receive the same input and be triggered simultaneously. The execution of the parallel is complete only after all of its branches finish execution. Fig. 3 illustrates the rules of modeling a parallel with two parallel branches. The parallel invocation is abstracted as a state transition connecting the start place of the parallel structure to the start place of each parallel branch, which is the start state of the function in this case. The transition

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

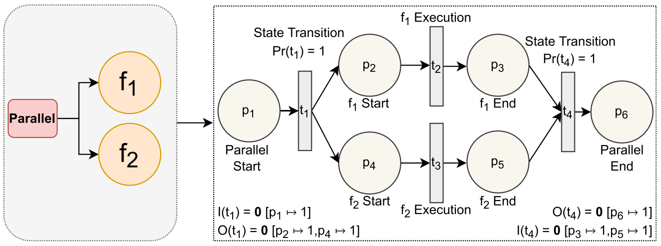


Fig. 3. Model a parallel with CSPNs.

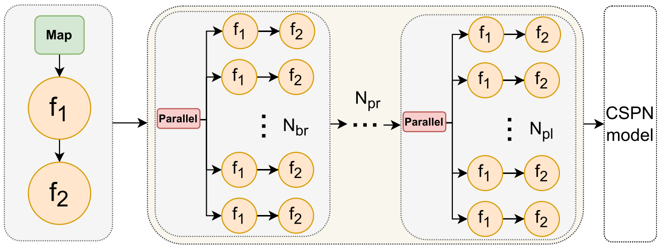


Fig. 5. Model a map with CSPNs.

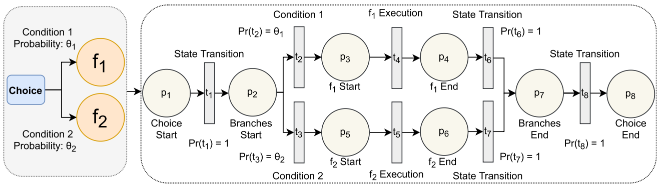


Fig. 4. Model a choice with CSPNs.

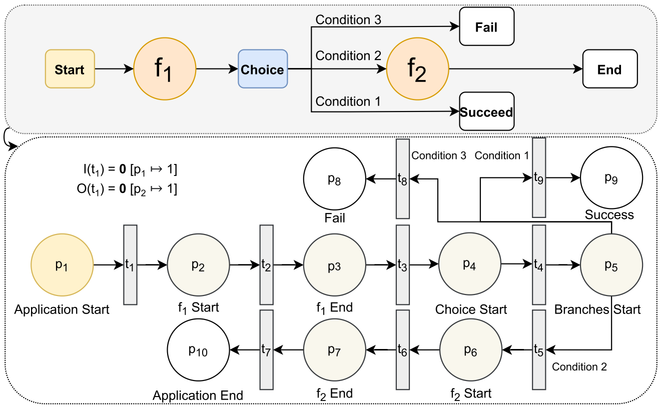


Fig. 6. An example of a serverless application with two functions and three terminal states and its model in CSPNs.

takes one token as its input and gives one token to the start place of each following action as its output, representing that all parallel branches start executions at the same time. The synchronization of the concurrent execution of parallel branches is captured by the state transition between the end place of each branch and the end place of the parallel structure, which takes one token from each branch and gives one token to the end place. In this case, transition t_4 is enabled only when both p_3 and p_5 receive a token, which guarantees the execution of the parallel is complete only after the execution of all parallel branches is complete.

3.4 Choices

Choices add if-else logic to serverless applications. Each choice has multiple branches, which are sequences composed of functions and structures. Only one branch (sequence) in the choice will receive the input and be executed depending on the satisfied condition, which can be described using probabilities defined by business logic and derived from execution logs or empirical knowledge. Fig. 4 demonstrates the rules for modeling the choice structure with CSPNs. Two conditions are equivalent to two state transitions between the places that represent the start state of the branches and the following action in different branches, whose input and output are one token. Namely, we have $I(t_2) = I(t_3) = \mathbf{0}[p_2 \mapsto 1]$, $O(t_2) = \mathbf{0}[p_3 \mapsto 1]$, and $O(t_3) = \mathbf{0}[p_5 \mapsto 1]$. As it may take some time to verify conditions, t_2 and t_3 may have a probabilistic firing delay.

When executing the choice, there is one token in p_2, enabling both t_2 and t_3 and causing a conflict. In this case, the conflict is resolved through the probabilistic firing rule, in which only one transition connecting to a branch is fired with the probability defined by Pr. When the branch completes its execution, p_7 will receive a token. Then, after t_8 fires, there will be one token in p_8, standing for the end state of the choice execution. Choices do not necessarily have an end place when modeled by CSPNs, and branches in choices may not be a sequence. Figs. 6 and 7 present applications with such choices. When constructing cycles/loops or unconditional jumps, choices act like the *goto* statement, which means the transition to another action or state. The choice only consists of two start places and transitions representing different conditions in this case. Also, the goto-like choice can only introduce transitions to actions and structures that have the same parent as it does. For example, a goto-like choice in a sequence cannot invoke functions in a branch of another parallel. We will further discuss the hierarchical relationship in Section 5.2.2.

3.5 Maps

Maps apply a function or a sequence to each element of the input array, and such functions or sequences are executed in parallel to process each element. Maps introduce dynamic parallelism into serverless applications, as the number of elements in the input array varies depending on the request payload and the business logic. A map is fundamentally equivalent to a sequence composed of a number of parallels, and each parallel branch contains the function or the sequence of actions to which the input data is mapped. Fig. 5 gives an example of a map structure, where each input element is applied to a sequence consisting of two functions. Since maps process input dynamically, the shape of the sequence is subject to the number of elements and the maximum concurrency level, which typically follows

$$N_{pr} = \begin{cases} 1 & N_{mc} = \infty \\ \left\lceil \frac{N_{el}}{N_{mc}} \right\rceil & otherwise \end{cases}$$

$$N_{br} = \begin{cases} N_{el} & N_{mc} = \infty \\ \min\{N_{el}, N_{mc}\} & otherwise \end{cases}$$

$$N_{pl} = \begin{cases} N_{el} & N_{mc} = \infty \\ N_{el} \bmod N_{mc} & otherwise \end{cases}$$

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

where N_{pr} is the number of parallels in the sequence, N_{br} is the number of branches in each parallel except for the last one in the sequence, N_{pl} is the number of branches in the last parallel, N_{el} is the number of elements in the input array, and N_{mc} is the maximum concurrency level that limits the number of concurrent actions running at one time.

Then, we can model the map structure by following the modeling rules presented in Sections 3.1 to 3.4. As the map structure assigns each input element to the same function/sequence, each parallel branch is "copied from" the function/sequence that defines the map. For example, in Fig. 5, there are in total $N_{br}N_{pr} + N_{pl}$ transitions that represent functions in the CSPN model, which have the exact same definition (e.g., W, Pr) and configuration (e.g., memory size) as the transition converted from f_1. Therefore, when modifying the configuration or definition of a function in a map, those "copies" in the application and the CSPN model are changed correspondingly. This is also applicable to updating memory sizes and marking optimized functions/structures in a map when optimizing applications, which will be presented in Section 5.2.

3.6 Start and Terminal States

Like other computer programs, serverless applications have one entry point for a given request, which is a point (i.e., function, structure) where the execution of the application begins and to where the original input data (payload) is passed. Serverless applications may have multiple exit points. The execution of the application stops immediately when any exit point is reached, and the application may exit with a status. When modeling serverless applications, we refer to the entry point as the start state and the end point as the terminal state. The location of the entry point may vary for different requests since a serverless application can usually start from any function (structure). With the CSPN model, we can easily adjust the location of the start state to reflect changes in the entry point.

As shown in the example in Fig. 6, the serverless application starts with f_1, which is the entry point of the application. There are three exit points. After the execution of f_1 is complete, the application will either trigger f_2 and then complete the execution, terminate with a status signifying failure, or exit with a success status. Therefore, when modeling serverless applications, there is always a place that stands for the start state and there are one or more places that represent the terminal state. For instance, in Fig. 6, p_1 is a start state and p_8, p_9, and p_{10} are terminal states. If there is a token in the start place of the application, it suggests the execution of the application has not yet started. The execution of the application is complete when any terminal state receives a token. Therefore, the initial marking of CSPNs modeling FaaS applications should be $\boldsymbol{m_0} = \mathbf{0}[p_S \mapsto 1]$, and the age of the token should be zero in most cases, namely $U(p_S) = \{0\}$, where p_S is the start state. In case there is initial latency (e.g., initialization latency/scheduling overhead incurred by the orchestration service), the initial age of the token may not be zero. Also, the delay incurred by the trigger of the serverless application can be captured by the probabilistic firing delay of the transition following the start state.

4 Performance and Cost Modeling

In this section, we propose the algorithm for performance and cost modeling of FaaS applications, which could obtain the distribution of the end-to-end response time and cost by simulating CSPNs and sampling delayed transitions. Then, we evaluate the efficiency of the proposed algorithms.

4.1 Algorithm Design

The pseudocode of the performance and cost modeling algorithm is presented in Algorithm 1. The input is the CSPNs that model the FaaS application and the number of iterations, denoted as CSPN and K, respectively. The algorithm runs K iterations, in each of which the CSPN model is first initialized by placing one token in the start place and removing all tokens from all other places (lines 3 to 8), i.e., making the initial marking of the net as $\boldsymbol{m_0} = \mathbf{0}[p_S \mapsto 1]$ and $U(p_S) = \{0\}$. Then, the algorithm simulates transition firing in CSPNs. Following definitions presented in Section 2, the algorithm obtains all enabled transitions under the current marking. Then, enabled transitions without any conflicts will be directly fired. Conflicting transitions are divided into different groups depending on the input place. The algorithm selects one enabled transition to fire from each group by weighted random sampling through the $wgt\_rand()$ method, and the probability is defined by Pr.

The transition firing is processed through the $fire()$ method, of which the implementation follows the steps presented in Section 2.2. Before the transition fires, its firing delay is set by random sampling based on the function W. When firing a transition, tokens from input places are consumed, and tokens with new ages are produced and placed in output places. Also, the transition keeps a record of its last firing time (LFT), which is the elapsed time between the last firing and the start of the application execution, namely $\max\{\Gamma\}$. Occurrences of firing transitions that model functions or cloud services incur costs, and the cost is returned by the $get\_cost()$ method. For transitions that represent functions, the cost is calculated by Equation (6). The algorithm obtains the cost from users' input for transitions representing services with different pricing models.

The algorithm continues to check and fire enabled transitions until a terminal place receives a token and no transitions were fired during the previous iteration (line 9), which means the execution of the modeled application is complete. The end-to-end response time of the application for this round of simulation is the minimum age of all tokens in terminal states. The exit status is the status label of the terminal state that has the token with the minimum age. The list $cost\_list$ records the costs incurred by the fired transitions and the point in time when the transition is fired. Only the costs incurred (i.e., transition fired) before the execution of the application is complete are used to calculate the total cost of the application. The output of the algorithm comprises three sets, each containing K values that represent the end-to-end response time, cost, and exit status of the FaaS application in K iterations, respectively. If K is large enough (e.g., 10000), the posterior distribution of the end-to-end response time, cost, and exit status derived by the algorithm becomes stable and could be considered as the accurate distribution. We could calculate the

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

tail latency and high-percentile cost based on the derived distribution.

Algorithm 1. $profile(\mathrm{CSPN}, K)$ Performance and Cost Modeling Algorithm for FaaS Applications

```
Data: model and number of iterations $\mathrm{CSPN}, K$
Result: $\mathrm{ERT}, \mathrm{COST}, \mathrm{STATUS}$
1:  $\mathrm{ERT} \leftarrow [], \mathrm{COST} \leftarrow [], \mathrm{STATUS} \leftarrow []$
2:  $p_S \leftarrow$ the start state of the application
3:  for $i$ from 1 to $K$ do
4:      for each $p \in P$ do
5:          $p.holdings \leftarrow []$
6:      end
7:      $p_S.holdings.add($ new $Token(age = 0))$
8:      $cost \leftarrow 0, cost\_list \leftarrow []$
9:      while $!isTerminated(\mathrm{CSPN})$ do
10:         $conflictedT \leftarrow$ new $HashMap\langle P, \wp(T)\rangle()$
11:         for each $t \in T$ do
12:             if $t.isEnabled()$ then
13:                 if $t.isConflicted()$ then
14:                     $conflictedT.get({}^{\bullet}t).add(t)$
15:                 else
16:                     $t.fire()$
17:                     $cost\_list.add((t.LFT, t.get\_cost()))$
18:                 end
19:             end
20:         end
21:         for each $p \in conflictedT.keySet()$ do
22:             $t \leftarrow wgt\_rand(conflictedT.get(p))$
23:             $t.fire()$
24:             $cost\_list.add((t.LFT, t.get\_cost()))$
25:         end
26:     end
27:     $ert \leftarrow min\{$age of tokens in terminal states$\}$
28:     $st \leftarrow$ status of terminal state with min age token
29:     for each $t, c$ in $cost\_list$ do
30:         if $t \leqslant ert$ then
31:             $cost \leftarrow cost + c$
32:         end
33:     end
34:     $\mathrm{ERT}.add(ert), \mathrm{COST}.add(cost), \mathrm{STATUS}.add(st)$
35: end
36: return $\mathrm{ERT}, \mathrm{COST}, \mathrm{STATUS}$
```

4.2 Algorithm Analysis

The time complexity of the performance and cost modeling algorithm depends on the number of iterations, the application workflow, and the implementation of the function W, which is generally bounded by $O(KL|T|(|P|+Q))$, where K is the number of iterations given in the input data, L is the number of iterations over the net structures before the execution of the application is complete, $|T|$ is the number of transitions, $|P|$ is the number of places, and Q is the overall time complexity of the partial function (callable) that implements the random firing delay generation. For valid serverless workflows, the number of places in their CSPN models should satisfy $|T| \leqslant |P| < 2|T|$, and there is generally a linear relationship between the number of serverless functions in the application and the number of transitions in the CSPN model. Also, the time complexity of a random sampling algorithm to generate probabilistic firing delay is usually $O(1)$ [18]. Therefore, the overall time complexity is $O(KLn^2)$, where n is the number of functions (actions/states) in the FaaS application.

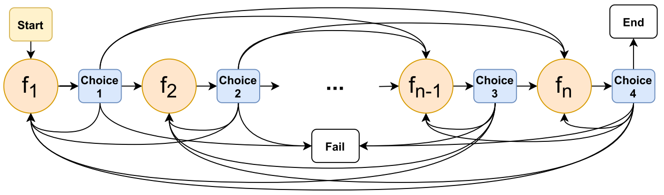


Fig. 7. Worst-case scenario.

For applications that do not contain cycles or self-loops in their workflows, or in which the number of cycles is small and the probability of entering cycles is low, there is usually a linear relationship between the number of functions and the number of iterations over the net structures, and the overall time complexity is $O(Kn^3)$. As the application workflow varies with the actual business logic that is unknown to outsiders in most cases, it is unreasonable to assume an average workflow and an average time complexity. However, we empirically study the most applications in the AWS serverless application repository [19], which is a public repository maintained by AWS to host serverless application projects. We find that the vast majority of serverless applications satisfy such conditions. Also, a recent empirical study shows that 72% of serverless applications have a small-scale workflow with less than ten functions [20].

For applications with a large number of cycles and self-loops, the magnitude of the coefficient L generally depends on the number of cycles/loops. While such workflows are highly unlikely to exist in real FaaS applications due to practical limitations (e.g., restrictions on duration, number of actions, and concurrency level), we analyze the efficiency of the modeling algorithm by considering the worst-case scenario shown in Fig. 7. The workflow is composed of n serverless functions that form a strongly connected component (SCC), since any function can be triggered after the execution of a function. For the worst-case scenario with $n = 220$, $K = 10000$, and $Pr(t) = \frac{1}{n+1}$ for transitions converted from choices, the modeling algorithm can give results in less than 100 seconds on two 2.10GHz Intel Xeon E5-2620 v4 processors. This shows the applicability of the modeling algorithm for now and in the foreseeable future.

5 Performance and Cost Optimization

In this section, we propose the algorithm for performance and cost optimization of FaaS applications, which can solve the trade-off between the end-to-end response time and cost by answering two optimization problems.

5.1 Problem Statement

We follow the problem statement in our previous work, but with some modifications to be consistent with the new motivations, notations, and the proposed modeling algorithm in this work. Let us consider a FaaS application composed of a set of functions F with viable memory options and response time defined by $\mathbb{V}$ and RT. We define the following two optimization problems.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

Best Performance under Budget Constraint (BPBC): For a given budget constraint BC (cost per 1 million executions in US dollars), or a given budget constraint BC and a number $k \in \{x \in \mathbb{Z} : 0 < x < 100\}$, we need to find a memory size for each function $f \in F$ from $\mathbb{V}(f)$ such that the FaaS application has the minimum average end-to-end response time while its average cost or the k-th percentile of its cost is less than or equal to BC.

Best Cost under Performance Constraint (BCPC): For a given performance constraint PC (end-to-end response time in milliseconds), or a given performance constraint PC and a number $k \in \{x \in \mathbb{Z} : 0 < x < 100\}$, we need to find a memory size for each function $f \in F$ from $\mathbb{V}(f)$ such that the FaaS application has the minimum average cost while its average end-to-end response time or the k-th percentile of its response time is less than or equal to PC.

Algorithm 2. $DFBA(\mathrm{CSPN}, S, BC, PC, k)$ Depth-First Bottleneck Alleviation Algorithm for Solving the BPBC and BCPC Problems

```
Data: CSPN model, structures and functions, budget/performance constraint, percentile $\mathrm{CSPN}, S, BC, PC, k$
Result: optimal memory configuration of each function $OMC : F \to \mathbb{N}$
1: $OMC \leftarrow$ new $HashMap\langle F, \mathbb{N}\rangle()$, $reoptimizeF \leftarrow []$;
2: get the avg rt map $avgRT : vDom \to \mathbb{R}_{\geqslant 0}$;
3: get the avg cost map $avgC : vDom \to \mathbb{R}_{\geqslant 0}$;
4: get the avg rt cost ratio map $ratio : vDom \to \mathbb{R}_{\geqslant 0}$;
5: $infeasible\_options \leftarrow$ new $HashMap\langle F, \wp(\mathbb{N})\rangle()$;
6: $initialize\_mem\_configurations()$;
7: $\mathrm{ERT}, \mathrm{COST}, \mathrm{STATUS} \leftarrow profile(\mathrm{CSPN}, 10000)$;
8: $f_B \leftarrow DFS(\mathrm{CSPN}, S, None)$;
9: while $f_B != None$ do
10:     $new\_f_B \leftarrow alleviate(f_B, \mathrm{CSPN}, S, BC, PC, k)$;
11:     if $new\_f_B == f_B$ then
12:         $f_B \leftarrow DFS(\mathrm{CSPN}, S, None)$;
13:     else
14:         $f_B \leftarrow new\_f_B$;
15:     end
16: end
17: $reoptimize(reoptimizeF)$;
18: for each $f \in F$ do
19:     $OMC.put(f, f.mem)$;
20: end
21: return $OMC$;
```

5.2 Algorithm Design

As mentioned in our previous work, BPBC and BCPC problems are more complex variations of the multiple-choice knapsack problem (MCKP), a well-known NP-complete problem [21], and we leveraged the PRCP algorithm to heuristically solve the trade-off. As discussed in Section 1, the PRCP algorithm is not effective for the BPBC and BCPC problems in this work. Hence, we propose the Depth-First Bottleneck Alleviation (DFBA) algorithm that comprises three phases: (1) a depth-first search (DFS) phase to identify the function with the most impact on overall performance and cost of the application in a depth-first manner, (2) a bottleneck alleviation phase to optimize the bottleneck function identified in the DFS phase with the optimal memory option, and (3) a profiling phase to check if the configuration satisfies the constraints and updates the firing logs. Algorithm 2 presents the pseudocode of the DFBA algorithm.

The algorithm first calculates the average response time, average cost, and ratio between average response time and cost of all functions in the application under their viable memory configurations (lines 2 to 4). The average response time of a function f with a memory size of mem is the mean of $RT(f, mem)$, and the average cost is calculated by $RT(f, mem)$ and Equation (6). For each $f \in F, mem \in \mathbb{V}(f)$, the ratio is calculated as $\frac{avgRT(f,mem_f)-avgRT(f,mem)}{avgC(f,mem_f)-avgC(f,mem)}$ such that $mem_f = \arg\max_{mem \in \mathbb{V}(f)} avgRT(f, mem)$. Then, the application is initialized by employing the memory size that achieves the lowest average cost for each function when solving the BPBC problem (line 6). When solving the BCPC problem, the memory size that leads to the shortest response time is leveraged. When updating the function f with the memory size of mem, the firing delay of the function in the CSPN model is also updated as $W[t \mapsto RT(f, mem)]$, where t is the transition converted from f. After updating the configurations, the algorithm profiles the application (line 7) and identifies the bottleneck function (line 8). Then, the algorithm continues to alleviate the identified bottleneck until no bottleneck function is found or can be alleviated due to constraints (lines 9 to 16). For functions added to the list for re-optimizing during the bottleneck alleviation phase, the algorithm finds the memory option achieving the best performance/cost within the constraints (line 17). We will discuss each phase in detail in the following subsections.

5.2.1 Profiling Phase

Before any DFS phase, the FaaS application should go through a profiling phase to obtain the firing logs for the given application and configuration. During the profiling phase, the performance and cost of the application are profiled using the modeling algorithm proposed in Section 4. Since a more accurate profile can improve the effectiveness of the optimization algorithm, the number of iterations K should be large enough in the profiling phase, which is 10000 by default in our setting. The algorithm obtains the end-to-end response time, cost, and exit status of the application during the profiling phase. Also, for each transition fired before the execution of the application is complete, the algorithm collects its last firing time, firing delay, and incurred cost into the firing logs each time the transition fires. The firing logs will be used in the DFS phase to obtain the response time and incurred cost of functions and structures for identifying bottlenecks.

5.2.2 Depth-First Search Phase

FaaS applications are usually composed of functions and structures (i.e., sequences, parallels, choices, and maps). We use S to denote all structures and functions in an application. The functions and structures are usually hierarchically nested. A structure may contain another structure or multiple structures or functions. For example, a parallel structure consists of multiple sequences, and each sequence may comprise a number of serverless functions. The hierarchies of FaaS applications can be very complicated. For instance,

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

the level of the hierarchy for function f_1 in the application shown in Fig. 5 is at least four, in which the top level is a sequence that contains N_{pr} parallels, the second level is the parallel, and the third level is the sequence that contains f_1. Therefore, while the FaaS application workflow is a directed graph (not a tree due to loops introduced by goto-like choices), structures can be linked to each other in parent-child relationships. When a structure comprises multiple structures and functions, we define the parent of these structures and functions as this structure. For goto-like choices, the parent of structures and functions to which it jumps is always the same as its parent.

Algorithm 3. $DFS(\mathrm{CSPN}, S, u)$ Depth-First Search Algorithm for Identifying the Bottleneck Function

```
Data: model, structures and functions, and where the search
      starts CSPN, $s, u$
Result: bottleneck function $f_B$
1:  $max\_impact \leftarrow -1$, $f_B \leftarrow None$, $isOptimized \leftarrow true$;
2:  $children \leftarrow \{s : s \in S$, the parent of $s$ is $u\}$;
3:  for each $s \in children$ do
4:      if $s.optimized$ then
5:          continue
6:      end
7:      $isOptimized \leftarrow false$;
8:      if $s.type == Function$ then
9:          $impact \leftarrow sum\{$firing delay of $s$ in firing logs$\}$;
10:         $//impact \leftarrow sum\{$incurred cost of $s$ in firing
            logs$\}$;                                   /* BCPC */
11:     else
12:         $impact \leftarrow sum\{$LFT of $s.end$ - LFT of $s.start\}$
13:         $//impact \leftarrow get\_total\_cost(s)$;          /* BCPC */
14:     end
15:     if $impact > max\_impact$ then
16:         $max\_impact \leftarrow impact$, $f_B \leftarrow s$;
17:     end
18: end
19: if $isOptimized$ and $u! = None$ then
20:     $u.optimized \leftarrow true$;
21: end
22: if $f_B.type == Structure$ then
23:     return $DFS(\mathrm{CSPN}, S, f_B)$;
24: end
25: return $f_B$
```

During the DFS phase, the algorithm traverses the FaaS application workflow in a depth-first manner and identifies the bottleneck function with the most impact on the overall performance (end-to-end response time) when solving the BPBC problem and that with the most impact on the overall cost when solving the BCPC problem. The pseudocode of the DFS phase is presented in Algorithm 3. When solving the BPBC problem, the algorithm calculates the total execution duration of each structure and action whose parent is u. For serverless functions, the execution duration is the firing delay recorded in the firing logs (line 9). The execution duration of structures is the difference between the LFT of the transition that gives tokens to the end place of the structure and the LFT of the transition that requires a token from the start place of the structure (line 12). When solving the BCPC problem, we consider the cost as the impact. The algorithm can retrieve the cost of serverless functions from the firing logs (line 10). The total cost of a structure is the sum of the cost of all its child structures and functions recorded in the firing logs, which is recursively calculated by the method get_total_cost (line 13).

In the DFBA algorithm, the DFS phase starts with the top-level structures or functions without a parent. If a structure has the most impact, the algorithm will recursively analyze its child structures and functions until a function with the most impact is found (line 23). The algorithm analyzes the impact of structures and functions that have not been marked as optimized at each iteration. The structure is marked as optimized if all its child structures and functions are optimized (line 20). For the map structure, it is marked as optimized when there is any parallel structure converted from the map marked as optimized. The algorithm only marks structures during the DFS phase, while functions are marked during the bottleneck alleviation phase.

5.2.3 Bottleneck Alleviation Phase

During the bottleneck alleviation phase, the algorithm alleviates the identified bottleneck by optimizing the memory configuration of the bottleneck function. Algorithm 4 provides the pseudocode of the bottleneck alleviation phase. The algorithm first obtains the memory options that are eligible for bottleneck alleviation (line 2). These options are the memory sizes that are not marked as infeasible options and lead to lower average response time (or cost) than the current memory size does. Namely, for each eligible memory option mem for solving the BPBC problem, the conditions (1) $mem \in \mathbb{V}(f_B)$, (2) $avgRT(f_B, mem) < avgRT(f_B, pre\_mem)$, and (3) $mem \notin infeasible\_options.get(f_B)$ hold true. When solving the BCPC problem, the second condition should be $avgC(f_B, mem) < avgC(f_B, pre\_mem)$. The eligible memory options are sorted by the average response time (or the average cost) achieved by the option in ascending order. Suppose no memory options satisfy these three conditions, which means no further optimization can be done on this bottleneck function, the algorithm marks the function as optimized and ends the alleviation phase (lines 3 to 6).

For each eligible memory option, the algorithm updates the memory configuration, profiles the application, and obtains the average value/k-th percentile of the cost/end-to-end response time, depending on the problem to solve (lines 9 to 14). If the new configuration does not satisfy the cost/performance constraint, the memory option is marked as infeasible, and the algorithm works on the next memory option at the next iteration. Suppose the identified bottleneck function still has the most impact on performance/cost even with the minimum possible average response time/cost. In that case, the algorithm adopts the memory option with the optimal response time-cost ratio (line 23), which is the memory size with the largest positive ratio or the smallest negative ratio, achieves lower cost (BPBC) or shorter response time (BCPC) than the current memory size does, and satisfies constraints. Then, the algorithm adds the function to the list to be re-optimized, marks the function as optimized, and ends the alleviation phase. If there are feasible memory options making the bottleneck function not a bottleneck anymore, the algorithm selects the memory option with lowest cost (BPBC) or that with the shortest response time (BCPC), without marking the

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

function as optimized. The function is marked as optimized when there is no feasible memory option at all (line 40).

Algorithm 4. $Alleviate(f_B, \mathrm{CSPN}, S, BC, PC, k)$ Alleviate the Bottleneck Function

```
Data: the same as Algorithm 2
Result: bottleneck function after alleviation phase
1:  $pre\_mem \leftarrow f_B.mem$, $flag \leftarrow false$;
2:  $options \leftarrow get\_eligible\_mem\_options(f_B, pre\_mem)$;
3:  if $options == \emptyset$ then
4:      $f_B.optimized \leftarrow true$;
5:      return $f_B$;
6:  end
7:  $min\_val \leftarrow +\infty$; $min\_val\_mem \leftarrow pre\_mem$;
8:  for each $mem \in options$ do
9:      $update\_configuration(f_B, mem)$
10:     $\mathrm{ERT}, \mathrm{COST}, \mathrm{STATUS} \leftarrow profile(\mathrm{CSPN}, 10000)$;
11:     $cost \leftarrow mean(\mathrm{COST})$; // BPBC; $k$ not passed
12:     $//cost \leftarrow percentile(\mathrm{COST}, k)$; // $k$ is passed
13:     $//ert \leftarrow mean(\mathrm{ERT})$; // BCPC; $k$ not passed
14:     $//ert \leftarrow percentile(\mathrm{ERT}, k)$; // $k$ is passed
15:     if $cost * 1000000 > BC$ // $ert > PC$ (if BCPC) then
16:         $infeasible\_options.get(f_B).add(mem)$;
17:         continue;
18:     end
19:     $new\_f_B \leftarrow DFS(\mathrm{CSPN}, S, None)$;
20:     if $new\_f_B == f_B$ then
21:         if $!flag$ then
22:             $optimize\_with\_optimal\_ratio(f_B)$;
23:             $reoptimizeF.add(f_B)$, $f_B.optimized \leftarrow true$;
24:             return $new\_f_B$;
25:         end
26:         $min\_val\_mem \leftarrow mem$;
27:         break
28:     else
29:         $flag \leftarrow true$;
30:         if $cost < min\_val$; // $ert < min\_val$ (BCPC) then
31:             $min\_val \leftarrow cost$; // $ert$ (if BCPC)
32:             $min\_val\_mem \leftarrow mem$;
33:         end
34:     end
35: end
36: $update\_configuration(f_B, min\_val\_mem)$;
37: if $min\_val\_mem == pre\_mem$ then
38:     $f_B.optimized \leftarrow true$;
39: end
40: return $new\_f_B$;
```

6 Experimental Evaluation

We implemented the CSPN and all proposed algorithms using Python 3.9. We validated our performance and cost modeling algorithm using five serverless applications deployed on AWS, which comprise four to sixteen functions with different workload patterns. We tested the DFBA algorithm on a serverless application composed of six functions and four structures. Also, the DFBA algorithm was evaluated against the PRCP algorithm proposed in our previous work. All algorithms, scripts, and experiment results are available in the public artifact repository[1].

1. https://github.com/pacslab/SLAppMdlOpt

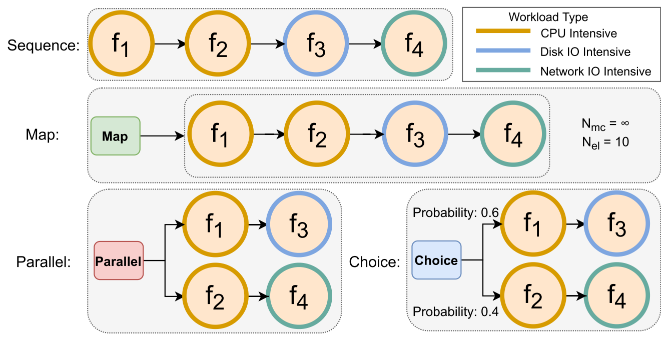


Fig. 8. Four applications (structures) used for evaluating the modeling algorithm.

6.1 Performance and Cost Modeling

6.1.1 *Experimental Design*

We first evaluated the performance and cost modeling algorithm on a set of serverless applications that contain one structure, namely sequence, parallel, choice, and map, where each is composed of four serverless functions with three types of workload as shown in Fig. 8. The CPU intensive workload involves arithmetic calculations and hashing. The hard disk IO intensive workload writes and reads several files to and from the hard disk drive, and the network intensive workload retrieves several files from an AWS S3 bucket located in the same region. We deployed four functions on AWS Lambda [22], specified an appropriate allocated memory size, and profiled them by sending requests to invoke them ten thousand times. For each function, we collected the execution logs from Amazon CloudWatch [23]. We removed the first and the last five percent of invocations from the logs to avoid any transient fluctuations in performance and adopted the remaining nine thousand execution durations as the performance profile. Then, to evaluate the accuracy of the modeling algorithm when dealing with applications with nested and complex structures, we followed the same procedures and conduct experiments using an application with sixteen functions and all types of structures. Fig. 10a demonstrates the definition of the application. The definition of the other four applications and the performance profile of the functions are presented in the artifact repository.

We defined the aforementioned five applications using Amazon States Language [24] and deployed them on AWS Step Functions [25]. Each application was invoked five thousand times with an interval of ten seconds. Application executions were monitored, and logs were collected from Amazon CloudWatch. Then, we defined the five applications using our proposed CSPN model and executed the performance and cost modeling algorithm with the obtained performance profile and the parameter $K = 100000$.

6.1.2 *Experiment Results*

We evaluated the accuracy of the modeling algorithm by comparing the distributions of the end-to-end response time and cost reported by AWS and those derived from the algorithm. The accuracy was evaluated from two aspects: the relative difference between the value given by the modeling algorithm and that reported by AWS for each statistic and

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

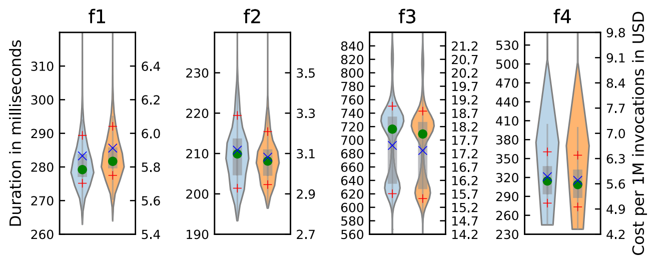


Fig. 9. The performance profile of the functions in Fig. 8. For each function, the left violin plot (with the left y-axis) illustrates the distribution of the function execution duration in milliseconds, and the cost per 1 million executions in US dollars is shown on the right (with the right y-axis). The markers $+$, $\times$, and $\bullet$ represent the 90th/10th-percentiles, the average, and the median, respectively.

the first Wasserstein distance between the distribution given by the modeling algorithm and that reported by AWS [26]. Fig. 11 shows the distributions of the response time and cost of five serverless applications, in which each plot on the left depicts the distribution predicted by the algorithm while the distribution reported by AWS is on the right. The number below each plot is the first Wasserstein distance between two distributions. Table 2 compares several statistics of the response time and cost of the application with sixteen functions. As is evident from Fig. 11 and Table 2, the distribution given by the algorithm is very close to the distribution from AWS, regardless of the complexity of the application workflow. Table 2 also gives the average accuracy (AA) of the predicted statistics over five serverless applications. The overall accuracy of the algorithm is above 97% for all four statistics. Such results demonstrate the high accuracy of the proposed performance and cost modeling algorithm.

6.2 Performance and Cost Optimization

6.2.1 Experimental Design

The performance and cost optimization algorithm was first evaluated on a serverless application with six functions and four types of structures, which is illustrated in Fig. 10b. We first profiled the six functions with different viable memory sizes. As response time becomes insensitive with large memory sizes, memory is allocated in larger increments when the memory size is larger while profiling functions. Specifically, the viable memory sizes of functions range from 128 MB to 1,024 MB in 64 MB increments, from 1024 MB to 2,048 MB in 128 MB increments, from 2,048 MB to 4,096 MB in 256 MB increments, and from 4,096 MB in 10,240 MB in 512 MB increments, resulting in 43 viable memory

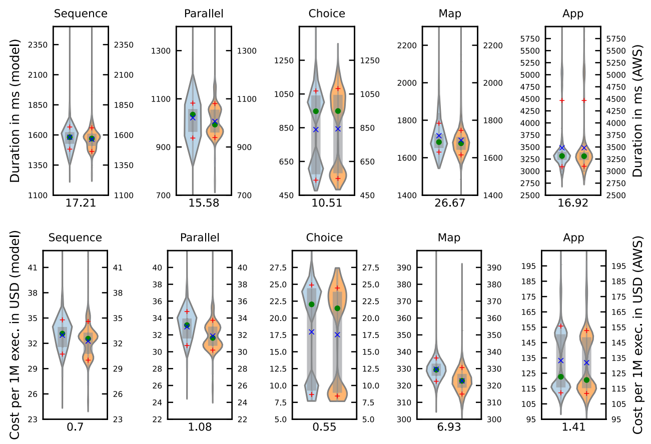


Fig. 11. The comparison between the distribution of the application response time and cost derived by the proposed algorithm (each plot on the left) and that reported by AWS (each plot on the right). The legend is the same as Fig. 9.

options for each function and 6,321,363,049 different memory configurations for the application. Fig. 12 shows the performance profile of the six functions with three types of workloads. The cost-memory size curve becomes smoother with the more granular billing model, suggesting the improved cost effectiveness of serverless functions.

We defined the application with CSPNs and ran the DFBA algorithm to solve four types of optimization problems, including the BPBC/BCPC problem with the average cost/performance as the constraint and that with the 90-th percentile of the cost/performance as the constraint. For each type of optimization problem, we chose fifty equidistant values as the constraints. The lower bound of the constraint is the minimum possible cost/performance (average value/90-th percentile). Also, we evaluated the DFBA algorithm against the PRCP algorithm. We ran the DFBA algorithm to solve the BPBC/BCPC problems (average cost/performance as the constraint) using the performance profile, application definition (App6), constraints, and pricing model in our previous work, and compared the optimization results with the best answers derived by four greedy strategies of the PRCP algorithm.

6.2.2 Experiment Results

The results of the DFBA algorithm solving the BPBC problem are shown in Fig. 13. The response time of the application is decresing along with the growth of the budget. While the maximum cost of the application is around \$1,016 on

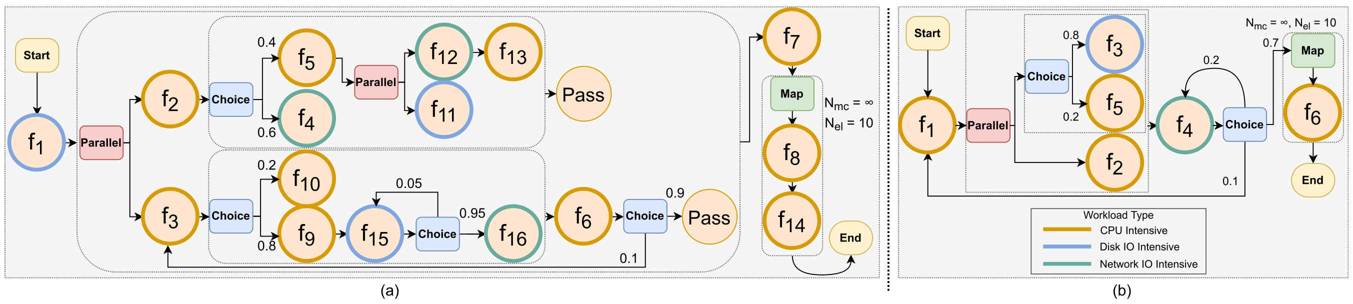


Fig. 10. (a) The serverless application with sixteen functions for evaluating the modeling algorithm. (b) The serverless application for evaluating the optimization algorithm. The performance profile of the functions is depicted in Fig. 12.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
The Experimental Evaluation Results for the Performance and Cost Modeling Algorithm

	Statistics	Model	AWS	Diff	AA
Duration	mean	3481.38	3482.39	-0.03%	99.09%
	median	3313.21	3309.00	+0.13%	98.90%
	10th %tile	3089.89	3100.00	-0.33%	99.08%
	90th %tile	4466.10	4466.09	0.00%	99.19%
Cost	mean	133.26	131.89	+1.04%	97.78%
	median	122.67	120.57	+1.74%	97.36%
	10th %tile	112.22	111.81	+0.37%	98.07%
	90th %tile	155.75	152.83	+1.91%	98.19%

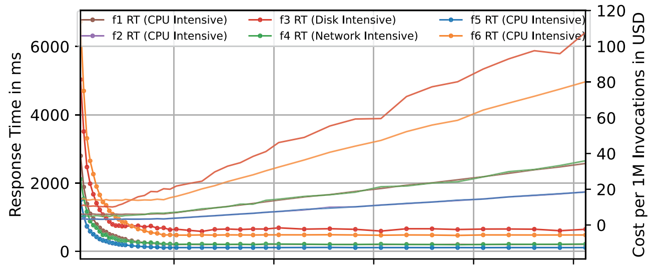


Fig. 12. The performance profile of the six functions.

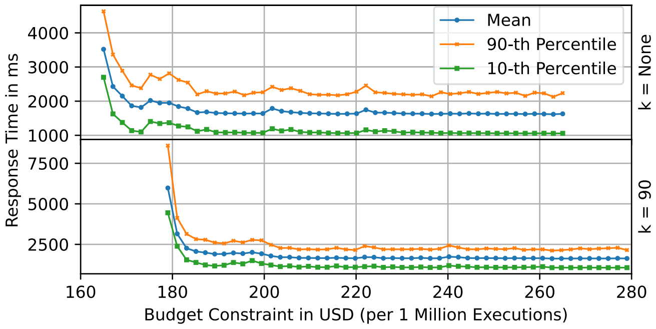


Fig. 13. The evaluation results of the DFBA algorithm (BPBC).

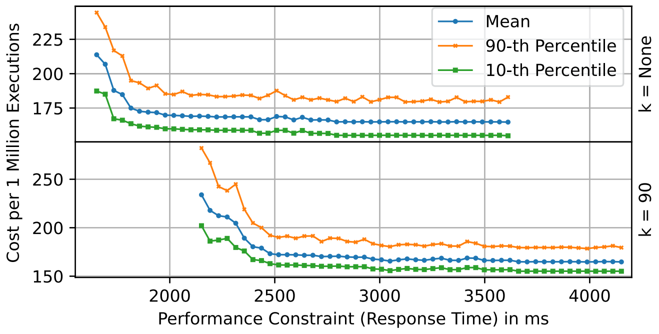


Fig. 14. The evaluation results of the DFBA algorithm (BCPC).

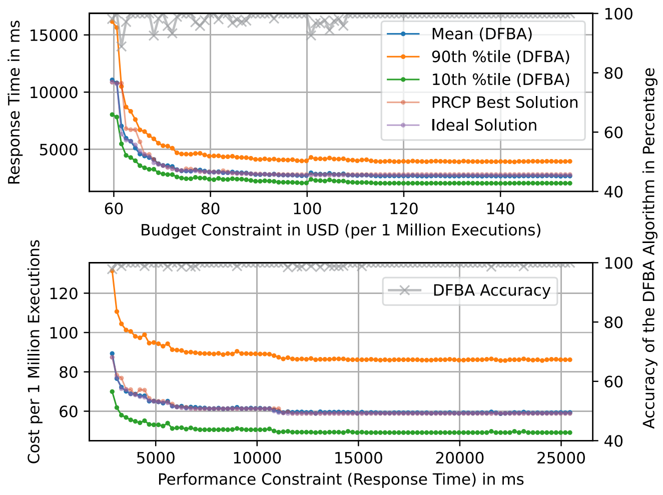


Fig. 15. Comparison between the DFBA algorithm ($k = None$) and the PRCP algorithm solving the BPBC/BCPC problems.

average among 6,321,363,049 configurations, the DFBA algorithm can obtain the optimal memory configurations achieving the minimum possible average end-to-end response time with the average cost at \$193.57 and the 90-th percentile of the cost at \$216.60 per 1 million executions when $k = 90$. Fig. 14 shows the DFBA algorithm solving the BCPC problem. Similarly, there is a clear downward trend in cost with increasing end-to-end response time. Despite the maximum average response time being 17981.03 ms, the algorithm can achieve the minimum average cost with the average response time at 2794.67 ms and the 90-th percentile of the response time at 3661.20 ms. Such results demonstrate the effectiveness of the DFBA algorithm in solving four types of optimization problems on FaaS applications with various structures and large state spaces.

Fig. 15 illustrates the comparison among the optimization results derived by the DFBA algorithm, the best solution given by the PRCP algorithm with four greedy strategies, and the ideal solution obtained by exhaustive search. The optimal solution given by the DFBA algorithm is very close to the ideal solution. The DFBA algorithm outperforms the PRCP algorithm, especially when the constraint is relatively small. We calculated the accuracy of the DFBA algorithm by the difference between the response time/cost given by the algorithm and the deterministic ideal value. As the response time and cost derived by the performance and cost modeling algorithm are probabilistic, we only considered significant differences greater than 1%. The average accuracy of the DFBA algorithm is 99.09% for the BPBC problem and 99.74% for the BCPC problem, which are higher than the PRCP algorithm (97.40% and 99.63%). Such results verify the high accuracy of the DFBA algorithm. Compared to the PRCP algorithm, the DFBA algorithm does not have to run different strategies to get the best answer and can work on FaaS applications with more types of structures, multiple exit points, and optimization constraints with percentiles. Also, the CSPN model can give accurate distributions of the response time and cost. Thus, the performance and cost modeling and optimization algorithms are a significant improvement over those proposed in our previous work.

7 Related Work

In recent years, serverless computing has attracted much attention from the research community. In this section, we will discuss some of the studies contributing to the body of knowledge in this field.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

Performance, cost, and availability have been listed among the top 10 obstacles towards the adoption of cloud services [27]. The serverless computing paradigm holds great promises for the future of cloud computing. However, one of the key factors holding the adoption of this paradigm back is its variable and sub-optimal performance. Many studies in the literature have focused on analyzing, modeling, and optimizing the performance and cost of serverless computing workloads. Wang et al. [28] performed multiple experiments on mainstream serverless computing platforms and gave insights into how each provider handles the workload introduced to their systems. Figiela et al. [29] analyzed the performance, cost, and life-cycle of a serverless workflow on four FaaS platforms. Their results shed some light on the management of the serverless platforms and the performance implications of different management decisions made by providers.

Many studies have analyzed benchmark applications on public FaaS platforms to highlight performance and cost implications. For example, Eyk et al. [10], [30] investigated the performance challenges in the current generation of FaaS platforms. They found the most critical challenges for serverless computing adoption are the sizable computational overhead, unreliable performance, absence of benchmarks, and new forms of performance-cost trade-off that reduces the predictability of services based on FaaS. The introduction of a reliable performance model could overcome some of the shortcomings. Manner et al. [31] analyzed the factors influencing the cold start performance of serverless functions, including the programming language, deployment package size, and memory size. Kaffes et al. [32] introduced a core-granular and centralized scheduler for serverless functions. They found serverless applications have several unique characteristics, including burstiness, short and variable execution time, statelessness, and single-core execution. Also, they found the scalability of the existing serverless platforms is sub-optimal. Bortolini et al. [33] investigated the factors related to the performance and cost of serverless computing platforms and found the low cost predictability is one of the most significant drawbacks of FaaS platforms. Lloyd et al. [34] performed a comprehensive investigation into five factors influencing the performance of FaaS platforms. Hellerstein et al. [35] pointed out the main gaps in the first-generation serverless computing platforms and the anti-patterns. They found that the lack of a global state and the inability to address individual function instances are the most important shortcomings. Singhvi et al. [36] introduced a scalable low-latency serverless platform named Archipelago. Their studies showed that current serverless schedulers are limited in handling very short-lived tasks, tasks with unpredictable arrival patterns, and tasks that require an expensive setup of sandboxes. Pelle et al. [37] looked into the suitability of FaaS platforms for latency-sensitive applications. They found the complexity of accurately predicting the application performance for a given task is one of the main drawbacks of serverless computing. Bardsley et al. [38] analyzed the possibility of deploying low-latency high-availability applications on AWS Lambda and argued that developers need a fair understanding of the underlying infrastructure. Mathew et al. [39] investigated the cost and performance benefits of serverless applications through experiments on AWS Lambda and AWS Step Functions. They found express workflows are economical for short-lived tasks with many transitions.

Some recent studies tried to address the lack of predictability in FaaS using performance modeling and simulation. For example, Boza et al. [40] proposed a model-based simulation analysis for cloud budget planning to allow cost simulation for using reserved VMs, on-demand VMs, and bid-based VMs in conjunction with FaaS platforms for the same computing task. Hence, the resulting serverless computing model is overly simplistic and does not capture many important aspects of the performance of these platforms. Abad et al. [41] developed a SimPy-based simulation for their proposed scheduling method for serverless computing platforms. Jeon et al. [42] introduced a CloudSim extension focused on Distributed Function-as-a-Service (DFaaS) on edge devices. However, the resulting simulator does not allow simulation for mainstream serverless computing platforms like AWS Lambda. Mahmoudi et al. [43], [44] analyzed autoscaling and found three categories in existing FaaS platforms. They proposed steady-state and transient solutions to performance modeling for scale-per-request autoscaling in FaaS. Also, the authors argued that the presence of an accurate and tractable performance model could improve the predictability of cost and performance and facilitate the adoption of serverless computing. Later, they introduced SimFaaS [45], which is a performance simulator designed specifically for FaaS platforms to allow single-function performance simulations. Manner et al. [46] proposed a local simulation approach to find the optimal configuration for individual serverless functions. Eismann et al. [47], [48] presented frameworks to predict the response time of serverless functions, cost of serverless workflows, and the optimal memory size of functions using machine learning models. They focused on predicting the optimal memory size for each individual function without considering the application-level constraints (i.e., end-to-end response time and cost). Also, their solutions couldn't predict the accurate distribution of the end-to-end response time and exit status of serverless applications and couldn't work on applications with various structures and non-DAG workflows.

There have been some works on empirical study and improving DevOps practices for serverless applications. For example, Leitner et al. [8] performed empirical studies about serverless use cases and FaaS cloud offerings. Eismann et al. [20] conducted a comprehensive study on 89 open-source and proprietary serverless applications and analyzed 16 characteristics. Wen et al. [12] identified current challenges of serverless application development based on Stack Overflow posts. Borges et al. [49] studied the integration of tracing in serverless applications to improve the observability of faults. Cordingly et al. [50] developed a reusable framework for inspecting performance, resource, and infrastructure metrics of serverless functions deployed on commercial and open-source FaaS platforms. Alpernas et al. [51] proposed a framework for runtime monitoring of serverless applications to facilitate the process of bug identification.

This paper differs from the previous work at least in the following aspects: 1) We propose a formal model to abstract FaaS applications with various structures and the rules for converting real-world FaaS applications into the proposed

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

model. 2) We achieve the fine-grained performance and cost modeling for FaaS applications by predicting the accurate distribution of the end-to-end response time, cost, and exit status. 3) We design an optimization algorithm to solve the performance-cost trade-off with application-level constraints in a fine-grained manner.

8 Conclusion

This work presents fine-grained modeling and optimization algorithms that predict the distribution of the performance and cost of FaaS applications and solve two types of optimization problems based on performance and cost constraints. We proposed CSPNs for abstracting FaaS applications or systems with similar patterns, which can effectively model concurrency, parallelism, synchronization, and randomness in cloud-native software systems with serverless architecture and defined modeling rules to convert FaaS applications into CSPNs. The performance and cost modeling algorithm is based on CSPNs sampling, which predicts the end-to-end response time, cost, and exit status of a FaaS application. The efficiency of the modeling algorithm was verified in the worst-case scenario with a large application workflow that is strongly connected. We evaluated the performance and cost modeling algorithm on five FaaS applications deployed on AWS. The accuracy of the modeling algorithm is over 97%, regardless of the complexity of the application workflow and the predicted statistics. We proposed the DFBA algorithm that achieves fine-grained performance and cost tuning for FaaS applications. The DFBA algorithm could find the optimal memory configuration that leads to the best performance under a budget constraint or the best cost under a performance constraint. The experimental evaluation demonstrates that the DFBA algorithm can effectively solve four optimization problems on a real application deployed on AWS with 6.32 billion different memory configurations. We also compared the DFBA algorithm with the PRCP algorithm. The evaluation results show that the DFBA algorithm outperforms the PRCP algorithm, with an average accuracy of over 99% for solving two optimization problems.

References

[1] I. E. Akkus et al., "{SAND}: {Towards High-Performance} serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 923–935.
[2] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.
[3] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 419–434.
[4] A. Eivy and J. Weinman, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, Mar./Apr. 2017.
[5] I. Baldini et al., "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Berlin, Germany: Springer, 2017, pp. 1–20.
[6] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 884–889.
[7] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," 2019, *arXiv:1902.03383*.
[8] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *J. Syst. Softw.*, vol. 149, pp. 340–359, 2019.
[9] S. Eismann et al., "A review of serverless use cases and their characteristics," 2020, *arXiv:2008.11110*.
[10] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG cloud group's vision on the performance challenges of faas cloud architectures," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 21–24.
[11] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, pp. 1–34, 2021.
[12] J. Wen et al., "An empirical study on challenges of application development in serverless computing," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 416–428.
[13] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, Mar. 2021.
[14] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *J. Syst. Softw.*, vol. 170, 2020, Art. no. 110708.
[15] "Serverless computing – AWS lambda pricing – amazon web services," 2021. Accessed: Aug. 24, 2021. [Online]. Available: https://aws.amazon.com/lambda/pricing/
[16] M. K. Molloy, "Performance analysis using stochastic petri nets," *IEEE Trans. Comput.*, vol. 31, no. 09, pp. 913–917, Sep. 1982.
[17] AWS step functions pricing – serverless microservice orchestration – amazon web services, 2021. Accessed: Aug. 24, 2021. [Online]. https://aws.amazon.com/step-functions/pricing/
[18] K. Bringmann and K. Panagiotou, "Efficient sampling methods for discrete distributions," in *International Colloquium on Automata, Languages, and Programming*. Berlin, Germany: Springer, 2012, pp. 133–144.
[19] Application search - AWS serverless application repository, 2022, Accessed: Jan. 16, 2022. [Online]. Available: https://serverlessrepo.aws.amazon.com/applications/
[20] S. Eismann et al., "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4152–4166, Oct. 2022.
[21] H. Kellerer, U. Pferschy, and D. Pisinger, "The multiple-choice knapsack problem," in *Knapsack Problems*. Berlin, Germany: Springer, 2004, pp. 317–347.
[22] Serverless computing - AWS lambda - amazon web services, 2022. Accessed: Jan. 08, 2022. [Online]. Available: https://aws.amazon.com/lambda/
[23] Amazon cloudwatch - application and infrastructure monitoring, 2022. Accessed: Jan. 08, 2022. [Online]. Available: https://aws.amazon.com/cloudwatch/
[24] Amazon states language, 2021. Accessed: Dec. 16, 2021. [Online]. Available: https://states-language.net/
[25] AWS step functions – serverless microservice orchestration – amazon web services, 2022. Accessed: Jan. 08, 2022. [Online]. Available: https://aws.amazon.com/step-functions/
[26] A. Ramdas, N. G. Trillos, and M. Cuturi, "On wasserstein two-sample testing and related families of nonparametric tests," *Entropy*, vol. 19, no. 2, 2017, Art. no. 47.
[27] M. Armbrust et al., "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[28] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.
[29] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency Comput. Pract. Experience*, vol. 30, no. 23, 2018, Art. no. e4792.
[30] E. van Eyk and A. Iosup, "Addressing performance challenges in serverless computing," in *Proc. ICT. OPEN*, 2018, pp. 6–7.
[31] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion*, 2018, pp. 181–188.
[32] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 158–164.
[33] D. Bortolini and R. R. Obelheiro, "Investigating performance and cost in function-as-a-service platforms," in *Proc. Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, 2019, pp. 174–185.
[34] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An investigation of factors influencing microservice performance," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2018, pp. 159–169.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.

[35] J. M. Hellerstein et al., “Serverless computing: One step forward, two steps back,” 2018, *arXiv:1812.03651*.

[36] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, “Archipelago: A scalable low-latency serverless platform,” 2019, *arXiv:1911.09849*.

[37] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, “Towards latency sensitive cloud native applications: A performance study on AWS,” in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 272–280.

[38] D. Bardsley, L. Ryan, and J. Howard, “Serverless performance and optimization strategies,” in *Proc. IEEE Int. Conf. Smart Cloud*, 2018, pp. 19–26.

[39] A. Mathew, V. Andrikopoulos, and F. J. Blaauw, “Exploring the cost and performance benefits of AWS step functions using a data processing pipeline,” in *Proc. IEEE/ACM 14th Int. Conf. Utility Cloud Comput.*, 2021, pp. 1–10.

[40] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, “Reserved, on demand or serverless: Model-based simulations for cloud budget planning,” in *Proc. IEEE 2nd Ecuador Tech. Chapters Meeting*, 2017, pp. 1–6.

[41] C. L. Abad, E. F. Boza, and E. Van Eyk, “Package-aware scheduling of FaaS functions,” in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 101–106.

[42] H. Jeon, C. Cho, S. Shin, and S. Yoon, “A cloudsim-extension for simulating distributed functions-as-a-service,” in *Proc. IEEE 20th Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, 2019, pp. 386–391.

[43] N. Mahmoudi and H. Khazaei, “Performance modeling of serverless computing platforms,” *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2022.3169619.

[44] N. Mahmoudi and H. Khazaei, “Temporal performance modelling of serverless computing platforms,” in *Proc. 6th Int. Workshop Serverless Comput.*, 2020, pp. 1–6.

[45] N. Mahmoudi and H. Khazaei, “SimFaaS: A Performance Simulator for serverless computing platforms,” in *Proc. Int. Conf. Cloud Comput. Serv. Sci.*, 2021, pp. 1–11.

[46] J. Manner, M. Endreβ, S. Böhm, and G. Wirtz, “Optimizing cloud function configuration via local simulations,” in *Proc. IEEE 14th Int. Conf. Cloud Comput.*, 2021, pp. 168–178.

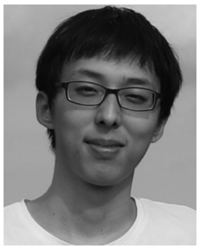
[47] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, “Predicting the costs of serverless workflows,” in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2020, pp. 265–276.

[48] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, “Sizeless: Predicting the optimal size of serverless functions,” in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 248–259.

[49] M. C. Borges, S. Werner, and A. Kilic, “Faaster troubleshooting-evaluating distributed tracing approaches for serverless applications,” in *Proc. IEEE Int. Conf. Cloud Eng.*, 2021, pp. 83–90.

[50] R. Cordingly, N. Heydari, H. Yu, V. Hoang, Z. Sadeghi, and W. Lloyd, “Enhancing observability of serverless computing with the serverless application analytics framework,” in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2021, pp. 161–164.

[51] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, “Cloud-scale runtime verification of serverless applications,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 92–107.



Changyuan Lin received the MSc degree in software engineering and intelligent systems from the University of Alberta, Edmonton, AB, Canada in 2021. He is a research associate with the Department of Electrical Engineering and Computer Science at York University, Toronto, ON, Canada. He worked on performant, secure and optimized microservice-based distributed systems. His research interests include cloud computing, serverless computing, performance modelling, and empirical software engineering.

Nima Mahmoudi (Student Member, IEEE) received the BS degrees in electronics and telecommunications and the MS degree in digital electronics from the Amirkabir University of Technology, Tehran, Iran in 2014, 2016, and 2017 respectively, and the PhD degree in software engineering and intelligent systems from the University of Alberta, Edmonton, AB, Canada in 2022. He is a data scientist with TELUS Communications Inc. His research interests include serverless computing, cloud computing, performance modelling, applied machine learning, and distributed systems.

Caixiang Fan (Student Member, IEEE) received the bachelor’s degree from the University of Electronic Science and Technology of China in 2012 and the MSc degree in software engineering and intelligent systems from the University of Alberta in 2019. He is currently working toward the PhD degree with the Electrical and Computer Engineering Department, University of Alberta, under co-supervision of Dr. Hamzeh Khazaei and Dr. Petr Musilek. His current research interests include blockchain, serverless computing, performance evaluation and modelling.

Hamzeh Khazaei (Member, IEEE) received the PhD degree in computer science from the University of Manitoba, where he extended queuing theory and stochastic processes to accurately model the performance and availability of cloud computing systems. He is an assistant professor with the Department of Electrical Engineering and Computer Science, York University. Previously he was an assistant professor with the University of Alberta, a research associate with the University of Toronto and a research scientist with IBM, respectively. His research interests include performance modelling, cloud computing and engineering distributed systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on December 28,2022 at 02:00:02 UTC from IEEE Xplore. Restrictions apply.