Future Generation Computer Systems 153 (2024) 403–418

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Serverless application composition leveraging function fusion: Theory and algorithms

János Czentye *, Balázs Sonkoly

High Speed Networks Laboratory, Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Magyar tudosok krt. 2, H-1117 Budapest, Hungary
MTA-BME Network Softwarization Research Group, Muegyetem rkp. 3, H-1111 Budapest, Hungary
HUN-REN-BME Cloud Applications Research Group, Muegyetem rkp. 3, H-1111 Budapest, Hungary

ARTICLE INFO

Keywords:
Serverless
FaaS
Cloud native
Function fusion
Tree partitioning

ABSTRACT

Serverless computing is a novel cloud computing paradigm enabling flexible, cost-efficient and fine-granular development of cloud-native applications, although without any guarantees on scheduling or execution times. Thus, various high-level solutions have been proposed in recent years to find proper configurations of individual FaaS functions, while considering user-given QoS requirements. However, the ever-increasing complexity of invocation patterns among stateless functions, externalized management of intermediate states, and diverse public cloud resources pose new challenges to the composition of highly data-intensive serverless applications. In this paper, we fill this gap by proposing novel algorithms based on the emerging function fusion technique, along with the related cost/performance models of composite functions supporting implicit instance parallelization and internal state propagation. We prove the NP-completeness of the underlying latency-constrained tree partitioning problem, and design a bicriteria approximation scheme and a greedy heuristic to derive cost-efficient deployment configurations in polynomial time. With the help of extensive simulations using synthetic call graphs generated from public cloud traces, we demonstrate the applicability and superior runtime performance of our proposed methods compared to state-of-the-art solutions. In addition, we showcase that further cost-reduction of up to 3–6 % can be achieved compared to the optimal partitioning with the allowance of tolerable latency violations.

1. Introduction

Serverless computing and the closely related Function-as-a-Service (FaaS) model have become a popular cloud computing paradigm in recent years, where developers implement their cloud native application in the form of ephemeral, stateless, and decoupled functions and leave all the operational burden on the shoulders of cloud providers [1,2]. Today's serverless frameworks, e.g., AWS Lambda [3], offer modular deployment and provisioning of software components relying on traffic load and invocation frequency-based autoscaling, and on the pay-as-you-go pricing model to achieve cost-efficient operation.

However, these advantages are provided at the costs of arbitrary function scheduling, introduced instance creation and initialization overheads, known as cold start, and the lack of user-given QoS guarantees, e.g., response time. Despite the apparent advantages of finer software granularity, developers still need to implement and bundle up deployable software packages, i.e, artifacts of the cloud application, and specify the sufficient amount of computational resources, such as operative memory demands and associated processing times, which is by no means a trivial task.

With these drawbacks in mind, various solutions have been published in the literature to improve the performance of serverless applications by paying particular attention to mitigated cold start [4,5], enhanced performance [6], reduced data management overheads [7,8], better resource utilization [9] or guaranteed function execution times [10]. Generally, low-level enhancements propose structural modification or extension to the serverless framework. In contrast, higher-level optimization approaches aim to find the optimal memory configuration for the deployed artifacts, i.e., cost-efficient flavors allocating adequate amounts of resources, while typically considering user-given service level objectives (SLO), such as an upper limit on the total response time.

However, the QoS-aware right sizing of individual artifacts poses additional challenges on the calculation of optimal flavor assignments due to varying function characteristics, nondeterministic cost-performance

* Corresponding author at: High Speed Networks Laboratory, Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Magyar tudosok krt. 2, H-1117 Budapest, Hungary.
E-mail addresses: czentye.janos@vik.bme.hu (J. Czentye), sonkoly.balazs@vik.bme.hu (B. Sonkoly).

https://doi.org/10.1016/j.future.2023.12.010
Received 30 June 2023; Received in revised form 3 November 2023; Accepted 9 December 2023
Available online 14 December 2023

relations, and the volatile nature of underlying cloud infrastructure [11, 12]. Although flavor selection methods can be leveraged in public cloud environments without the need of architectural changes, the severe performance degradation factors of communication overheads and cold start remain unresolved challenges.

In recent years, a different approach has emerged to reduce operational cost and improve the performance of serverless application, in which basic building block functions are grouped together and assembled in the form of composite functions that can be seamlessly managed by the cloud framework as regular serverless functions. This deployment adaptation technique, called function fusion [13], allows for higher-level control over the application's fundamental scalable units, while mitigating the hindering effects of inter-function invocations [14, 15] and helps reducing additional costs charged by platform-supported workflow management services after each state transition step [16].

Additionally, the skillful combination of standalone functions also enables us to comply with user-defined SLO requirements based on improved execution times [17], reduced cold start [4], or facilitated data management [6]. However, diverse invocation rates and complex inter-function communication patterns, which typically form a dynamic call graph with tree-like structure [18], make the cost-optimal function composition and related resource assignment problems more difficult. Moreover, standard fusion approaches that assume subsequent executions do not take full advantage of resource flavors providing multiple vCPU cores. On top of that, the group-internal handling of subresults can offer performance benefits compared to straightforward externalization of intermediate states through a platform-managed in-memory cache service, e.g., Redis [19].

In order to fill this research gap, in this paper we propose dedicated models and novel algorithms, which are capable of calculating the cost-optimal and QoS-aware deployment structure of serverless applications with a call graph of asynchronous invocations, while diminishing state transmission overheads and utilizing multiple cores with implicit function instance parallelization with the help of function fusion. Our main contribution is three-fold.

First, we elaborate cost and performance models for cloud native composite functions, while considering a wrapper-assisted execution model with parallel execution of function instances and internal state management. We also formalize the underlying optimization problem and rigorously prove its NP-completeness.

Second, based on these models, we develop exact algorithms to find feasible deployment configurations using linear and dynamic programming techniques, along with their assessed complexities and limitations. Building on these results, we also provide efficient heuristic and approximation methods to overcome the intractable number of examined subcases.

Third, we conduct extensive simulations to demonstrate the applicability and advantages of our proposed methods using synthetic call graphs (trees) that are generated according to the characteristics of data-parallel job and cloud native applications monitored in public clouds [18,20]. To facilitate future benchmarking for the research community, we released the implementations of our algorithms, test harness suit and data generators as an open source software [21].

The rest of the paper is organized as follows. In Section 2, the related works are summarized. Our defined serverless models and the related optimizations problem are detailed in Section 3. In Section 4 and Section 5, we provide a detailed description of our proposed algorithms and their applicability, while Section 6 evaluates the performed simulations. In Section 7 we discuss the implications of our results and in Section 8 we conclude our paper.

2. Related work

Optimal resource configuration of cloud-based applications has become an active research field in recent years [1]. In the context of serverless computing, the primary objective is to find the right sizing of FaaS functions in a cost-effective way while complying with user-given service level objectives (SLO). For instance, researchers in [22] recommend fine-grained performance and cost models for FaaS applications relying on conditional stochastic Petri nets. They demonstrate the NP–hardness of the emerging optimization problem and propose heuristic algorithms for the addressed cost/performance trade-off with respect to different invocation patterns. Although this technique can provide accurate estimations, the low-level function modeling prevents its general application with 3rd-party components.

However, the Sizeless framework published in [23] allows for predicting the optimal flavors of black-box functions using dedicated neural network-based models trained on public-cloud measurements of synthetic functions with pre-selected memory configurations. Despite its straightforward multi-target regression model can derive cost-optimal memory sizes of standalone functions from their extracted feature sets, it lacks consideration of call graphs and inter-function invocation patterns.

In contrast, researchers in [12] design a tool called SLAM that is capable of tracing relationships between FaaS functions, estimating their execution times, and derive optimal memory configuration for the given application regarding an upper limit on the overall execution time as the SLO. It applies a closed-loop reoptimization structure extended with function instrumentation to generate workloads and analyze trace logs for reconstructing the call graph, and uses an iterative elimination algorithm to find the first SLO-feasible configuration with the lowest total cost with the intrinsic assumption that performance and cost of one function is reversely proportional to each other.

Similarly, the COSE framework from [9] also offers a dynamic, feedback-loop based memory optimization service for serverless applications with delay constraint, although without any assumptions on the performance of the components. It uses statistical learning techniques with Bayesian optimization to predict cost and execution times of a chain of functions with the ability to adapt to changes in function executions or in the underlying infrastructure, on-the-fly. Likewise, the Aquatope tool [24] also uses Bayesian models to provide QoS-aware resource management for multiphase serverless workflows relying on the prior monitoring loop design, with the extension of tuning the number of prewarmed containers to mitigate cold start.

From the perspective of data-intensive analytics, the Astra tool [25] performs autonomous memory configuration of QoS-aware FaaS jobs with special attention to the hindering effects of intermediate data management. It realizes graph-based algorithms to enhance application performance and meet execution time constraint by reducing its primary optimization problem to the NP–hard constrained shortest path problem (CSP).

The aforementioned works aim to find the cost-optimal memory (flavor) selection of the given functions, regardless of the fundamental composition of these deployable serverless components. However, function fusion based techniques provide more flexible and cost-effective solutions in terms of eliminated invocation overheads and resource utilization of functions with different resource demands [4,26]. In our related article [17], we propose polynomial-time algorithms to minimize the overall cost of a tree-shaped serverless application with latency constraint, while focusing on the fusion of functions on subchains using multicore flavors.

In a similar work of [15], researchers develop a fusion-based optimization framework called Fusionize to dynamically eliminate double billing by greedily partitioning connected functions, i.e, subtrees of the application's call graph into deployable artifacts. However, these solutions omit the calculation of memory configurations and rely on strict restrictions or greedy decisions to achieve fast algorithmic runtimes for their (re)deployment control loops.

In addition to these works, the Costless tool [27] proposes a unified approach to both fusion functions and also finds sufficient resource assignments to the composed groups at the same time, while considering a latency threshold and also edge–cloud placement decisions

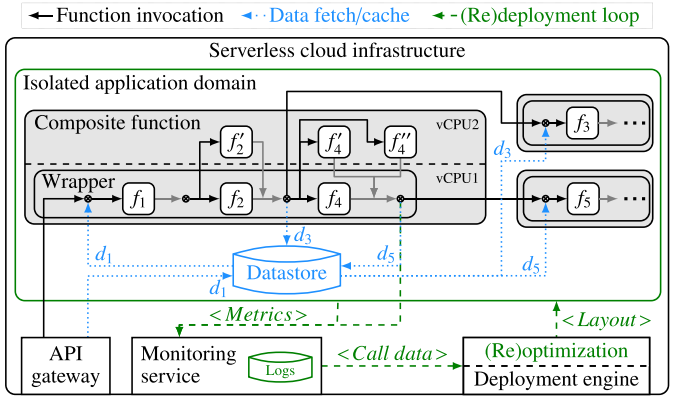


Fig. 1. Schematic architecture of a composite function leveraging external in-memory datastore for managing intermediary states between composed blocks.

by modeling them with dedicated flavors. Although this approach combines prior achievements, its applicability is limited, as it relies on the NP–hard CSP problem and requires input call graphs as chains or series-parallel graphs.

Unfortunately, none of these works exploit the full potential of composite functions as they consider neither integrated state management nor take advantage of multicore flavors.

3. Serverless layout optimization

3.1. Platform computation model

In contrast to platform-provided workflow orchestration solutions, in which serverless functions are loosely connected according to a higher-level orchestration descriptor file [1], function fusion requires tighter integration of encompassed software modules. To avoid the unfavorable drawbacks of function composition stated as the "serverless trilemma" in [13], while ensuring maximal flexibility and reusability, we consider simple, stateless, and short-lived functions as basic black boxes that can invoke each other through their single and well-defined interfaces in an asynchronous way.

Accordingly, these functions can be grouped together into separate composite functions based on their relations of call dependency and communication data type, as one function's output format must match the dependent function's call syntax and input data format. Since a function can be invoked by its predecessor more than once, the resulting function instances in a group have to be implicitly initiated and managed, independently of the platform-provided autoscaling capabilities. However, this implicit invocation management also allows for much finer control over the executions of encompassed software components, achieving not only mitigated invocation and cold start overheads [4], but also improved response times [7,14,28], better resource utilization in multicore environments [17,26], and ultimately lower cost.

For these reasons, we assume that each assembled function group is extended with a lightweight wrapper that is responsible for (i) mimicking the API of the front-end function, (ii) dispatching function invocations regarding internal function instances as well as external composite functions in a platform-specific way, (iii) managing intermediate data transfer for local invocations, (iv) fetching and caching subresults among composite functions using a platform-managed in-memory storage service, e.g., Redis [19], and (v) optionally reporting function-level metrics to the platform's monitoring service, e.g., Amazon CloudWatch [29] or Prometheus [30], for advanced application management, as it has been evaluated in practice in our prior works [14,17] over Amazon's cloud, as well as in other researches [15]. The schematic architecture of a composite function with its instance management and communication patterns is shown in Fig. 1.

In the general case of multilingual software code bases, the standalone wrapper can instantiate instances from the encompassed function executables as separated internal sub-processes [14,15], either directly or leveraging efficient techniques of pre-initialized runtimes or preloaded binaries [31,32], and perform intermediate state exchange locally using standard inter-process communication (IPC) techniques instead of accessing to an expensive remote storage service [6,7]. However, in the specific case of a homogeneous and modular code base, where basic functions as compatible software modules or standalone shared libraries can be assembled and built together into a single executable artifact similarly to the works in [5,15], function instances can be initialized by the integrated wrapper following the fork-join model [5,28].

Nevertheless, in our approach we assume the former, more general, and wrapper-governed approach of instance initiation that is not influenced by the multithreading limitations of specific runtimes, e.g., Python's global interpreter lock (GIL). Thus, we consider the overhead of direct function invocations to be negligible compared to the indirect invocation methods provided by serverless cloud frameworks, which can take up from tens to hundreds of milliseconds [11]. Moreover, the parallel execution of local function instances also helps reduce operational cost due to better vCPU utilization, and to mitigate the grouped functions' cold start by reducing the underlying microVM and container creations to a single one [4].

Based on this wrapper-supported fusion technique, we also assume that functions encompassed in one group are executed in a predefined sequential order of call dependency relations starting with the front-end function invoked outside the group similarly to the solutions in [5,7,14,17,28,33]. This straightforward function chaining approach along with the aforementioned implicit instance management method allows us to realize a simple and lightweight workflow coordination middleware with predictable resource consumption, and most importantly without any in-place decision, instance configuration, or internal scheduling logic. Moreover, the overall execution time of these composite functions can be kept manageable by prioritizing the most important functions in the execution sequence, which can be performed at design phase. Accordingly, fusion-based application structures can easily comply with any user-defined latency constraint defined on a critical path. Basically, careful grouping allows us to circumvent hindering performance factors of platform-provided invocations, state externalizations, and premature executions of noncritical functions.

In order to further reduce the CPU footprint and initialization overhead of function compositions, we assume that all initialization data of the function merged together are preloaded by the wrapper and shared with the initiated function instances using efficient techniques, such as memory mapping [5]. This approach can significantly reduce the instances' execution time due to the elimination of redundant and on-demand data fetching, although at the expense of increased cold start overhead and operational memory demand of the whole function group. However, prefetching of configuration data is of paramount importance in use cases where some functions require an exceedingly large amount of initialization data. For instance, complex and large neural network models offer fast inference but require model weights, which can have the size of several hundreds of megabytes that is possibly larger than the platform's deployment size quota (250 MB in AWS [3]), to be loaded into memory from an external datastore. This means that dynamic model loading could lead to outstandingly large overheads.

Therefore, the CPU-memory trade-off introduced by the preloading of configuration data needs to be taken into account during function fusion, as platform flavors fundamentally specify an upper limit on the accessible operative memory of deployable units [3]. The aforementioned platform-related limitations accompanied by user-define QoS limitations, such as the overall processing delay of an input, anticipate the need for an efficient and cost-optimal composition of serverless applications.

Table 1
Mathematical notations used in the paper.

Notation	Description
$T(F,E)$	Application model as a rooted, directed and weighted tree
$t_f, m_f \in \mathbb{N}_+$	Node weights of runtime [ms] and memory demand [MB]
$r_f, d_f \in \mathbb{N}_+$	Edge weights of invocation rate [1/s] and data caching [ms]
$\wp \in F$	Dedicated node denoting the serverless framework's gateway
$\phi \in \Phi$	Platform resource flavor as a triplet of $\langle M_\phi, N_\phi, \gamma_\phi \rangle$
$M_\phi, N_\pi \in \mathbb{N}_+$	Available memory [MB] and vCPU cores of flavor ϕ
$\gamma_\phi \in \mathbb{Q}_+$	Cost multiplier of flavor ϕ
$\mathcal{P}$	Power set operator
$P_T \subset \mathcal{P}(F)$	Node partitioning as disjoint sets of nodes of T
$p_b \in P_T$	Partition block of partitioning P_T with barrier node $b \in F$
$B_{P_T} \subseteq F$	Set of barrier nodes identifying the partitioning P_T
$C_{P_T} \subseteq E$	Set of multicut edges identifying the partitioning P_T
$\varphi : P_T \mapsto \Phi$	Flavor assignment of partition blocks
$n^\phi_{f,b} \in \mathbb{N}_+$	Billed number of instances of $f \in p_b$ assigned to flavor ϕ
$E^+_{p_b} \subset E$	Set of out-edges regarding partition block $p_b \in P_T$
$\mathrm{c}_\phi(p_b) \in \mathbb{Q}_+$	Operational cost [GBs] of block $p_b \in P_T$ assigned to ϕ
$\mathrm{m}_\phi(p_b) \in \mathbb{N}_+$	Memory demand of partition block $p_b \in P_T$ with flavor ϕ
$k^\phi_{f,b} \in \mathbb{N}_+$	Largest consecutive instances count of $f \in p_b$ with flavor ϕ
$f_\pi \in F$	User-given leaf node of T marking the end of critical path π
$\pi = [f_1 \to f_\pi]$	Critical path of T from its first function f_1 to a leaf node f_π
$L_\pi \in \mathbb{N}_+$	End-to-end latency constraint [ms] for T's critical path π
$\delta \in \mathbb{N}_+$	Platform invocation delay among partition blocks
$\pi_b = [p_b \cap \pi]$	Sub-path of p_b's topological ordering restricted by π
$B_\pi = \pi \cap B_{P_T}$	Barrier nodes of partition blocks restricted by π
$b^*_\pi \in B_\pi$	Barrier node of the π-limited block called by a function in p_b
$\mathrm{l}_\phi(p_b) \in \mathbb{N}_+$	Expected latency of the partition block $p_b \in P_T$ with flavor ϕ
$x^\phi_p \in \mathbf{X}$	Decision variables of partitioning operations from $\mathcal{P}(F) \times \Phi$
T_b	Subtree of T induced by the root node b and all its descendant
$\mathbf{C}[v,i]_{l,m}$	Minimal cost subcase of node v and its first i children

3.2. Layout composition with function fusion

In general, the deployment of any serverless application requires (i) the collection of software modules and dependencies forming the deployment artifacts, and (ii) the belonging resource configurations, i.e., flavors, from the user. Regarding function fusion, we generalize the description of a deployment configuration, called the *serverless application layout*, by extending the deployment units, considered as composite functions, with the information of encompassed basic functions. Hence, the main goal of any serverless application composition or optimization service is to derive a suitable layout, that is, the groups of building block functions and assigned resource flavors, based on the dependency relations, available flavors, and the predefined platform and user constraints. In the following, we provide the formal definition of serverless layout composition assuming our wrapper-based execution model detailed in Section 3.1. Notations introduced in this paper are summarized in Table 1.

For the application description model, we assume a rooted, directed, and weighted tree $T(F,E)$ consisting of basic black-box functions F and invocation relations E, which is either provided by the developer beforehand or can be extracted from the records of a platform monitoring service. Each function $f \in F$ is characterized by its overall execution time $t_f \in \mathbb{N}_+$ measured on a reference configuration with one vCPU core [11], including all implicit I/O operation delays and language-specific interpreter/runtime overheads (as cold start), and its peak memory demand $m_f \in \mathbb{N}_+$, while each ingress edge $e_f = (g,f) \in E$ has the weights of aggregated invocation rate $r_f \in \mathbb{N}_+$, i.e., the number of invocations of f triggered by a single invocation of T, and the data externalization overhead $d_f \in \mathbb{N}_+$, i.e., the intermediate data transmission delay assuming the usage of a platform-managed in-memory cache service. In this context, we consider that the transmission delay is the same for both R/W operations as caching g's output and fetching f's input require transmitting the same intermediate state, that is, the same amount of data. To model the application's initial call rate and input data size, we introduce a dedicated root node $\wp$ representing the serverless platform itself and the related edge $(\wp, f_1) \in E$, where f_1 marks the first/entry point function.

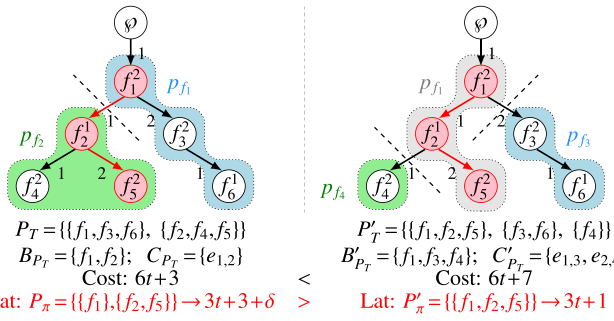


Fig. 2. Feasible partitionings of nodes f^m_i of a tree T having specific memory demands m and edge data weights, with uniform runtimes t and invocation rates of size 1, while assuming critical path $\pi = \{f_1, f_2, f_5\}$ and a single flavor $\langle 4, 1, 1 \rangle$.

Platform-provided resource configurations Φ are defined as a set of triplets $\phi = \langle M, N, \gamma \rangle$ that specify the amount of available operative memory $M_\phi \in \mathbb{N}_+$ and vCPU cores $N_\phi \in \mathbb{N}_+$ as upper bounds, and the dimensionless coefficient $\gamma_\phi \in \mathbb{Q}_+$ as the relative cost ratio of flavor prices with respect to one billing unit of execution time.

Fundamentally, a deployment layout of T is the set of non-empty, pairwise disjoint, and connected sets of functions F, i.e., the node partitioning P_T of tree T, along with the flavor assignment $\varphi : P_T \mapsto \Phi$ that maps a resource flavor $\phi \in \Phi$ to each partition block $p \in P_T$. Formally, P_T must meet the following requirements: (i) $P_T \subset \mathcal{P}(F)$ where $\mathcal{P}$ marks the power set operator, (ii) $\emptyset \notin P_T$, (iii) $\bigcup_{p \in P_T} p = F$, (iv) $\forall p_a \neq p_b \in P_T : p_a \cap p_b = \emptyset$, and (v) $\forall p \in P_T$ must be a connected subtree of T having only a single ingress edge, i.e., $|\{(g,b) \in E \ : g \notin p \wedge b \in p\}| = 1$.

Node $\wp$ belongs to no partition block, hence e_1 is always a cut. As an equivalent formalization, partitioning of T is the task of dividing T into separated subtrees by designating the set of subtree roots $B \subseteq F$, called the barrier nodes, or interchangeably a multicut $C \subseteq E$. It is easy to see that in tree graphs, where one node has only one predecessor, barrier nodes can unambiguously determine a multicut and vice versa. Moreover, a multicut bisects edges into internal and external (intra-block) subsets that correspond to the locally implemented and platform-supported invocations, respectively.

Since barrier nodes provide a compact way to track and encode partition blocks, as well as a straightforward approach to calculate distinct layouts from the discrete set of edges E, we leverage this concept in the context of tree partitioning and define two actions of *cut* and *merge* for an edge $(f,g) \in E$ to denote the partition decision of separating or keeping together the related two functions f and g. Accordingly, we utilize the notation of $p_b \in P_T$ to denote the unique partition block of a given partitioning P_T with front-end function $b \in F$, which also forms a subtree of T with root node $b \in B_{P_T}$. Each subtree block cut by C_{P_T} is rooted in a barrier node, while its leaves are either original leaves of T or a predecessor node of another barrier node in B_{P_T}. In Fig. 2, we depict illustrative examples for multicut tree splitting with the related barrier nodes.

3.3. Cost and constraint models

Based on our wrapper-based computation model, each block p of a given partition P_T sequentially initiates the encompassed functions according to a topological ordering of the related cut subtree T_b started with its barrier node b. The internal instances of each function $f \in p$ are scheduled according to the available vCPU cores N_ϕ of the considered flavor ϕ, while the number of instances depends on the function's invocation rate r_f compared to the partition block's invocation rate given by r_b. Since cloud-based monitoring services typically measure aggregated metrics for a period of time, we assume no explicit information about any function's egress call distribution.

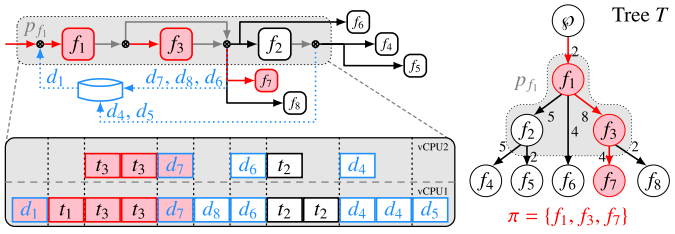


Fig. 3. Illustrative example of block-internal scheduling in a partition block $p_{f_1} \in P_T$ of the given tree T with explicit invocation rates and two vCPU cores.

Thus, without loss of generality, we assume that functions are simple and single-threaded building boxes developed for consistently perform a dedicated task without any significant variance in the fan-out invocation rate or execution performance. Hence, the total execution makespan of an encompassed function $f \in p_b$ comes from the block-relative number $\frac{r_f}{r_b}$ of its instances and their low-level scheduling on the available vCPU cores N_ϕ, whereas the total execution time of a partition block comes from the aggregated function execution times, input fetching and state caching overheads, as depicted in Fig. 1.

First, to get the overall billed number of instances $n^\phi_{f,b}$ of function $f \in p_b$ regarding all platform-initiated instances r_b of the composite function p_b and the assigned flavor $\phi \in \Phi$, we use the following formula

$$n^\phi_{f,b} = \underbrace{(r_f \bmod r_b)\left\lceil \left\lceil \frac{r_f}{r_b} \right\rceil \frac{1}{N_\phi} \right\rceil}_{\substack{\text{Saturated instance executions} \\ \text{utilizing all vCPU cores}}} + \underbrace{(r_b - r_f \bmod r_b)\left\lceil \left\lfloor \frac{r_f}{r_b} \right\rfloor \frac{1}{N_\phi} \right\rceil}_{\text{Number of straggler instances}}, \tag{1}$$

that separately calculates the number of function instances fully utilizing the available number of vCPU cores and the straggler instances. The main connection between the parts of billed instance count and the wrapper-based direct invocation management is visualized in Fig. 3 that also demonstrates the assumed distribution of egress function calls in cases of varying function behaviors. In this example, f_3 has saturated instances and f_1 has one straggler instance, whereas instances of f_2 are contributed from both.

While the input data d_b of the first function b in p_b need be loaded only once by default as $n^\phi_{b,b} \triangleq r_b$, each outgoing invocation introduces additional state externalization overhead. Thus, let $E^+_{p_b}$ represent the set of out-edges of block p_b. Based on the general pay-as-you-go billing model [11], we define the execution cost of a partition block in Eq. (2) relying on the billed instance execution times and data fetching and caching overhead, and accordingly the aggregated operation cost of a given partitioning as $\sum_{p \in P_T} \mathrm{c}_{\varphi(p)}(p)$. This formula applies only to one call of T and actually provides a provider-agnostic cost metric value in units of milliseconds rather than in any currency.

$$\mathrm{c}_\phi(p_b) = \gamma_\phi \Big[\underbrace{r_b d_b}_{\substack{\text{Input fetching delay} \\ \text{of the barrier node}}} + \sum_{f \in p_b} \Big[\underbrace{n^\phi_{f,b} t_f}_{\substack{\text{Instance} \\ \text{executions}}} + \sum_{e_u \in E^+_{p_b}} \underbrace{n^\phi_{u,b} d_u}_{\substack{\text{State caching} \\ \text{delays}}} \Big] \Big]. \tag{2}$$

Therefore, different partitionings of an application result in different costs due to the chosen block flavors and the data transmission delays of the external invocations. These formulas also reveal that each state externalization is considered twice (except for the application's initial invocation), once to fetch input data at a barrier node and once to cache sub-results. For leaf nodes of T, caching the final result is included in their execution times. In our illustrative example of Fig. 3, the input fetching overhead is included as d_1, while state externalization overheads of p_{f_1}'s egress edges are incorporated as d_4–d_8, all shown in blue.

However, for a deployable application layout, we must only consider composite functions that satisfy predefined constraints. Accordingly, a given partitioning P_T is a *feasible partitioning* if it satisfies the platform's memory constraints M_ϕ and the end-to-end latency requirement $L_\pi \in \mathbb{N}_+$ defined on the application's critical path $\pi = [f_1 \to f_\pi]$, i.e., the subchain of T between its first/root node f_1 and a user-given leaf node f_π.

Memory constraints are applied independently to each partition block $p \in P_T$ based on flavor assignment φ. These hard limits must not be exceeded, otherwise an under-provisioned component could stall and as a consequence its execution would be suspended by the framework. Regarding data sharing techniques discussed in Section 3.1, we define the highest memory demand $\mathrm{m}_\phi(p_b)$ of a partition block $p_b \in P_T$ in Eq. (3), which relies on the total prefetched function data and the operative memory usage based on the number ($\leq N_\phi$) of instances running in parallel. Then, for any feasible partitioning P_T, it must hold that $\forall p \in P_T : \mathrm{m}_{\varphi(p)}(p) \leq M_{\varphi(p)}$.

In fact, it is a pessimistic approach as prefetched data are not always as large as instance demands. Since M is a hard limitation we leverage the same value m_f for both to prevent overbooked and stalling containers. It is worth mentioning that other function demands and block-related upper constraints, such as the artifact's maximum execution time, ephemeral storage size, I/O bounds, or deployment size [3], can be added to our model in a similar way. Nevertheless, without loss of generality, we consider $\mathrm{m}_{\varphi(p)}(p)$ and $M_{\varphi(p)}$ to be scalars.

$$\mathrm{m}_\phi(p_b) = \max\left\{ \underbrace{\sum_{f \in p_b} m_f}_{\substack{\text{Total size of prefetched} \\ \text{data in the wrapper}}}, \max_{f \in p_b} \left\{ \underbrace{\min\left\{ \left\lceil \frac{r_f}{r_b} \right\rceil, N_\phi \right\} m_f}_{\substack{\text{Operative demands} \\ \text{of parallel instances}}} \right\} \right\}. \tag{3}$$

Unlike memory constraints that can be verified and enforced locally for each partition block, the latency limit L_π restricting a subchain that spans the entire application T can affect multiple partition blocks and only some subset of their grouped functions, which cannot be enforced block by block. Moreover, the expected execution delays of function instances differ from the total billed execution times leveraged in Eq. (2), as it is only considered for a block's subpath coinciding with the critical path π and only for the longest instance execution paths, as visualized in Fig. 3. Nevertheless, we can calculate the largest number of consecutive instances $k^\phi_{\pi,f}$ for a nonbarrier function $f \in p_b$, $f \neq b$, on the restricted subpath $\pi_b = [p_b \cap \pi]$ of a π-limited block ($b \in \pi$) using the following recursive formulas.

$$k^\phi_{f,b} = \left\lceil \frac{r_f}{r_g N_\phi} \right\rceil k^\phi_{g,b}, \quad k^\phi_{b,b} = 1. \tag{4}$$

In Eq. (4), g marks the block-internal predecessor of node f, such that $(g, f) \in \pi_b$. Since each barrier function b has only one local instance due to $\frac{1}{r_b} n^\phi_{b,b} = 1$, the number of b's latency-influencing instances $k^\phi_{b,b}$ equals to 1, as well. Basically, $k^\phi_{f,b}$ derives the accumulated number of function instances within a block following the block-internal invocation scheduling shown in Fig. 3. In addition, to introduce the hindering effects of cut edges with respect to latency, we define the average external invocation delay $\delta \in \mathbb{N}_+$ that describes the platform's management overhead, that is, scheduling delays of deployed artifacts within a VPC. Moreover, let b^*_π designate the barrier node following b on π related to the restricted out-going invocation of p_b, that is, the single π-limited barrier node invoked from p_b.

Similarly to the cost calculation in Eq. (2), we define the partition block latency $\mathrm{l}_\phi(p_b)$ introduced on T's critical path π in Eq. (5), and the overall latency L_{P_T} as $\sum_{p \in P_T} \mathrm{l}_{\varphi(p)}(p) + \delta(|B_\pi| - 1)$, where $B_\pi = \pi \cap B_{P_T}$ marks the unique set of latency-limited partition blocks. Here, we consider that critical functions are prioritized in π-restricted blocks even in favor of block-external data caching, as illustrated in red in Fig. 3. Moreover, we calculate the external invocation delays between blocks (excluding the first block) explicitly based on the number $|B_\pi|$ of barrier nodes that also belong to π. For this, we leverage that a given block p_b is restricted by π iff $b \in \pi$. Additionally, our model can be easily extended with a supplementary delay constant representing

the application's triggering delay ($\wp \to f_1$) or the transmission delay between the user and the platform's gateway. Consequently, for any feasible partitioning P_T, it must also hold that $L_{P_T} \leq L_\pi$.

$$\mathrm{l}_\phi(p_b) = \begin{cases} 0, & \text{if } b \notin \pi, \\ d_b + \sum_{f \in \pi_b} k^\phi_{f,b} t_f, & \text{if } f_\pi \in p_b, \\ d_b + \sum_{f \in \pi_b} k^\phi_{f,b} t_f + k^\phi_{b^*_\pi, b} d_{b^*_\pi}, & \text{otherwise.} \end{cases} \tag{5}$$

It is important to mention that our serverless application layout modeled in Eq. (1)–(5) relies on consistent function behavior to leverage multiple vCPU cores for efficient application deployment without the knowledge of exact execution time or call rate distributions. However, in special cases of $N_{\phi^*} = 1$, such as having one single flavor ϕ^* providing just enough vCPU cores for the (even multithreaded) basic functions, our formulas are reduced to simpler forms, i.e., $n^{\phi^*}_{f,b} = r_f$ and $\mathrm{m}_{\phi^*}(p_b) = \sum_{f \in p_b} m_f$.

Fundamentally, these relations describe the fully serialized execution of encompassed function instances that does not rely on or require assumptions of function behavior, which makes our model resilient against any variation of fan-out distributions.

3.4. Problem definition and complexity analysis

Relying on the cost and constraint models of serverless applications introduced in Section 3.3, the main purpose of serverless layout optimization is to find a cost-optimal feasible layout, that is a viable deployment configuration of the application's basic functions, which satisfies the platform and user-given QoS limitations. First, we formulate the optimization problem of *Serverless Application Composition* (SAC) in Definition 1.

Definition 1. SAC(T, Φ, π, L_π): find a node partitioning $P_T \subset \mathcal{P}(F)$ and a related flavor assignment $\varphi : P_T \mapsto \Phi$ of the serverless application $T(F, E)$, such that (i) $\forall p \in P_T$ the calculated block memory demand does not exceed the platform limit $M_{\varphi(p)}$, (ii) the calculated latency of T's critical path π does not exceed the user-given limit L_π, and (iii) the total cost of the deployment layout P_T is minimal among all feasible partitionings.

Since all node and edge weights are typically provided by a platform-managed monitoring service as discrete values, there can exist multiple minima for the SAC problem. Unfortunately, greedy approaches to our problem are intractable since examining all possible partitionings requires generating all combinations of multicuts, which is equal to $\sum_{c=0}^{|E|} \binom{|E|}{c} = 2^{n-1}$ for a tree of size n.

In essence, our problem consists of function partitioning and flavor assignment parts, the latter of which is proven to be NP-hard for a latency-limited chain of standalone functions assuming no linear relation between flavors and function performance [34]. In the following, we demonstrate the NP-hardness of the SAC problem by focusing on the former partition-based subproblem while reducing it to the well-known NP-complete *Graph Partitioning Problem* (GPP) [35,36] defined in Definition 2.

Definition 2. GPP(T, ω, ρ, B): Suppose a graph $G(V, E)$ with node weights $\omega : V \mapsto \mathbb{N}_0$ and edge values $\rho : E \mapsto \mathbb{N}_0$, and a knapsack capacity $B \geq \max\{\omega(v) : v \in V\}$, find a disjoint node partitioning $P_G \subset \mathcal{P}(V)$, such that (i) $\forall p \in P_G$ the aggregated weight $\sum_{f \in p} \omega(p) \leq B$, and (ii) the summed value $\sum_{e \in E'} \rho(e)$ of uncovered edges $E' = \{e_{uv} \in E : \{u, v\} \not\subseteq p \in P_G\}$ is minimized.

Theorem 1. *The (decision version of) SAC problem is* NP*-complete.*

Proof. Obviously, SAC $\in$ NP, since any given partitioning P_T can be verified in linear time if it is feasible regarding M, L_π, and a decision cost bound by calculating all metrics $\forall p \in P_T$. To prove NP-hardness, we use a reduction from GPP. First, we construct a specific instance of SAC by ignoring the latency limit L_π and considering a tree T' with uniform call rates $r_f \equiv 1$, a single flavor σ of $B \geq \max\{m_f : f \in F\}$, and $N = \gamma = 1$, that is, $\Phi = \{\langle B, 1, 1\rangle\}$.

Hence, the flavor assignment φ becomes trivial and block metric calculations in SAC$(T', \Phi, \emptyset, \infty)$ simplify to $\mathrm{c}_\sigma(p_b) = r_b\, d_b + \sum_{f \in p_b} r_f\, t_f + \sum_{e_i \in E^+_{p_b}} r_i\, d_i$, and $\mathrm{m}_\sigma(p_b) = \sum_{f \in p_b} m_f$ as stated in Section 3.3. Thus, the main objective is reduced to minimize the data transfer of externalized states detailed in Eq. (6).

$$\min \sum_{p \in P_T} \mathrm{c}_\sigma(p) = \min \Big\{ \sum_{b \in B_{P_T} \setminus \{f_1\}} 2 r_b d_b + \sum_{f \in F} r_f t_f \Big\} \overset{r_b = 1}{=} \min \sum_{b \in B_{P_T}} d_b. \tag{6}$$

Regarding the transformations $\omega(f) = m_f$ and $\rho(e_{uv}) = d_v$, it is easy to see that GPP$(T, \omega, \rho, B) \equiv$ SAC$(T', \Phi^*, \emptyset, \infty)$. Thus, a partitioning is a solution to SAC iff it is a solution to GPP. ■

By these means, the optimization of a serverless application layout with memory (and latency) constraints cannot be solved in polynomial time even with only one flavor, unless P = NP. Therefore, in this paper, we propose efficient exact and approximation algorithms for solving variants of our SAC problem.

4. Optimal solution with linear programming

Linear programming (LP) is one of the well-established approaches to solve hard problems resource efficiently. According to our cumulative models defined in Eq. (1)–(5) and their intrinsic relationship to the Generalized Assignment Problem (GAP), the combined optimization of function fusion and flavor assignment can be expressed as an Integer Linear Program (ILP) [37].

As a baseline model, we provide the straightforward formulation of our combined SAC problem, called *configuration ILP model* (CfgILP). Let $x^\phi_{bc} \mapsto \mathcal{P}(F) \times \Phi$ be a decision variable denoting the partition block $p_{b,c} \in \mathcal{P}(F)$ with barrier node $b \in F$, inducing multicut $c \in \mathcal{P}(E)$, and assigned flavor $\phi \in \Phi$, called a unique configuration of tree T. Consequently, the optimization problem lies in the cost-minimal selection of disjoint configurations that forms a feasible partitioning of T regarding all constraints. Let c^ϕ_{bc}, m^ϕ_{bc}, and l^ϕ_{bc} be shorthand notations for block metrics $\mathrm{c}_\phi(p_{b,c})$, $\mathrm{m}_\phi(p_{b,c})$, and $\delta + \mathrm{l}_\phi(p_{b,c})$, respectively. Considering that $\forall x^\phi_{bc} : m^\phi_{bc} \leq M_\phi$ (M-feasibility), CfgILP can be formulated as

$$\Big\{ \min \sum c^\phi_{bc} x^\phi_{bc} : \sum l^\phi_{bc} x^\phi_{bc} \leq L_\pi + \delta;\ \sum_{b,c\,:\, f \in p_{b,c}} x^\phi_{bc} = 1;\ x^\phi_{bc} \in \{0, 1\} \Big\}, \tag{7}$$

i.e., to minimize the cost while meeting the implicit memory, latency, and all feasibility constraints $\forall f \in F$, such that one function must belong to only one block and the binary decision variables evaluated to 1 designate the selected partition blocks and their assigned flavors. This simple formalization allows us to utilize arbitrary methods to calculate block attributes without any linearity restriction. Unfortunately, this approach is highly inefficient due to the intractable $\mathcal{O}(2^{n-1})$ number of decision variables representing all possible function combinations.

In contrast to set covering problems [37], our main problem does not stipulate the final number of partition blocks, however, groups are required to be connected, that is, to form a connected subtree of T. Thus, to keep the size of our model under control, we propose a more detailed, function-based modeling approach that considers lower-level merge actions instead of higher-level group selections at the cost of a larger number of constraints.

Suppose an n-size tree T, let $\mathbf{X}$ be an $n \times n$ matrix of binary variables, in which $x_{fb} \in \mathbf{X}$ represents the partition decision of merging node $f \in F$ to the block $p_b \in \mathcal{P}(F)$ starting with barrier node $b \in F$, i.e., $x_{fb} = 1 \Leftrightarrow f \in p_b$. By definition, a node cannot be assigned to any of its descendant nodes, i.e., $\forall d \in F : x_{fd} \triangleq 0$ if there exists a directed chain $[f \to d]$ in T, while the set of barrier nodes can be identified as $B_{P_T} = \{b \in F : x_{bb} = 1\}$. With this approach, columns can be rearranged so that all interested variables are set in the lower triangular part of $\mathbf{X}$

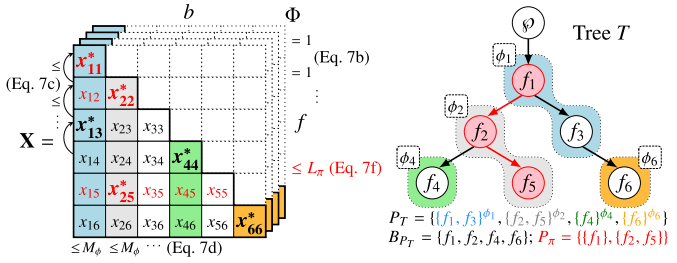


Fig. 4. Illustrative example of one feasible partitioning of tree T with respect to our matrix ILP model. Nonzero variables in $\mathbf{X}$ are marked with an asterisk.

as exemplified in Fig. 4, while all designated barrier nodes are located in the diagonal of matrix $\mathbf{X}$. The number of utilized variables is at most $\frac{1}{2}n(n+1)$, which depends on the tree's basic structure.

This structure enables the convenient and flexible formalization of the ILP model's constraints and overall cost, as these relations need to be defined either in a per-function (rows) or a per-block (columns) basis. Moreover, block-dependent variables help to circumvent issues of symmetric solutions typically encountered with set-coverage problems, since each block p_b differs according to its barrier node b. To generalize our model, we introduce a third dimension to $\mathbf{X}$, such that $x^{\phi}_{fb} \in \mathbf{X}$ determines the assigned flavor ϕ of the given partition block p_b, as well.

For this problem formalization, called the *matrix ILP model* (MtxILP), we utilize the shorthand notations defined previously. However, we also rely on two essential characteristics of our serverless composition formulas in Eq. (1)–(5), which at the same time specify the intrinsic limitations of this ILP approach. First, the per-function merge decisions imply that block metric calculations must rely on the linear relationship of grouped functions. Second, in order to formalize our problem as a linear program, merge operations must be independent from the encompassed functions of the partition block, i.e., from the prior merge decisions. Clearly, our block metrics satisfy these two criteria. Besides the assigned flavor ϕ, instance execution time of a merged function f depend only on the block invocation rate r_b given by the barrier node $b : f \in p_b$, whereas data externalization overheads are defined by the merged edge e_f, solely.

Accordingly, per-function merge decisions can be generally considered as the combination of partition blocks formed by standalone functions where the included fetching and caching overheads need to be eliminated for each merged edge.

$$\min \sum_{\phi\in\Phi}\sum_{b\in F'}\sum_{f\in T_b} \gamma_\phi \Big[n^{\phi}_{f,b}\big(S^f_b\, d_f + t_f\big) + \sum_{(f,u)\in E} n^{\phi}_{u,b} d_u\Big] x^{\phi}_{fb}, \tag{8a}$$

$$\text{s.t.} \sum_{\phi\in\Phi}\sum_{b:\, f\in T_b} x^{\phi}_{fb} = 1 \qquad \forall\, f \in F', \tag{8b}$$

$$x^{\phi}_{ib} - x^{\phi}_{jb} \geq 0 \qquad \forall\, \phi \in \Phi, b \in F', (i,j) \in E(T_b), \tag{8c}$$

$$\sum_{f\in T_b} m_f x^{\phi}_{fb} \leq M_\phi \qquad \forall\, \phi \in \Phi, b \in F', \tag{8d}$$

$$\min\left\{\left\lceil \frac{r_f}{r_b}\right\rceil, N_\phi\right\} x^{\phi}_{fb} \leq M_\phi \qquad \forall\, \phi \in \Phi, b \in F', f \in T_b, \tag{8e}$$

$$\sum_{\phi\in\Phi}\sum_{b\in\pi}\sum_{\substack{f\in T_b\cap\pi \\ v:(f,v)\in\pi}} \Big[\delta I^f_b + k^{\phi}_{f,b}\big(S^f_b\, d_f + t_f\big) + k^{\phi}_{v,b} d_v\Big] x^{\phi}_{fb} \leq L_\pi, \tag{8f}$$

$$x^{\phi}_{fb} \in \{0,1\} \qquad \forall\, \phi \in \Phi, b \in F', f \in T_b. \tag{8g}$$

Relying on these aspects, we define our matrix ILP model in Eq. (8), where S^f_b marks the signum function yielding 1 if $b = f$, otherwise -1, and I^f_b represents an identity function returning 1 if $f = b$, otherwise 0 (and in our case, also if $f = f_1$ to ignore the application's triggering delay). In Eq. (8a), our *objective* is to minimize the aggregated cost of partition blocks, for which the candidate functions are limited to the barrier node b and all its descendant, i.e., the induced subtree T_b with root b. By definition, we only consider basic functions $F' = F \setminus \{\wp\}$, and use S^f_b to include input fetching overheads for barrier nodes and also to remove previously included data caching overheads of merged functions. In Eq. (8b), we restrict all functions to belong to one partition block (*feasibility constraints*) as in the configuration model, whereas *connectivity constraints* defined in Eq. (8c) explicitly ensure that partition blocks must be connected subtrees. Basically, we rely on the observation that in connected blocks a nonbarrier function is grouped together with its single predecessor. Eqs. (8d)–(8e) define the block *knapsack constraints*, where the operational memory demands of prefetched function data and parallel instances are separately restricted according to the linear decomposition of Eq. (3). However, Eq. (8e) can also be used to exclude unavailing (must-zero) variables before providing our model to the ILP solver. The critical path's *latency limit* L_π is ensured by Eq. (8f), where the introduced block delays are calculated similarly to Eq. (2), while leveraging I^f_b to incorporate platform invocation delay δ, and S^f_b to derive fetching/caching overheads. Finally, Eq. (8g) defines the *binary constraints*.

Compared to the configuration LP model, our matrix model needs to take special care for the connectivity of partition blocks and the elimination of included data overheads in addition to the apparent memory, latency, and feasibility constraints. Although it is more complex than our former configuration model, its size is polynomially bounded with respect to the tree structure and available resource flavors, as it requires $\mathcal{O}\big(\frac{1}{2}n(n+1)|\Phi|\big)$ number of decision variables and $\mathcal{O}\big(n^2|\Phi|\big)$ number of constraints.

5. Tree partitioning algorithms

In contrast to LP formalization that imposes an implicit linearity requirement on the model formulas in Eq. (1)–(5), iterative algorithms based on tree traversal techniques can be exploited to obtain node partitioning without verifying explicit feasibility, connectivity, or linearity constraints.

However, the multimodal nature of layout optimization that incorporates two distinct NP-hard subproblems makes our joint SAC problem even more difficult. Regarding that flavor assignment basically introduces an additional dimension to the function fusion problem's subcases as stated in the proof of Theorem 1, in the rest of the paper we examine the tree partitioning problem with the default, user-given, single flavor $\sigma = \langle M, N, 1\rangle \in \Phi$.

5.1. Recursive tree partitioning

Tree partitioning algorithms relying on iterative generation of minimal-cost partitioning subcases allow us to calculate the optimal partitioning, where in each iteration only the relevant subset of the main problem's exponentially large state space is maintained to reduce calculation times [38,39]. These approaches rely on the recursive decomposition of the optimal partitioning into optimal sub-solution based on the edge cut assignments. Briefly, if an edge $(f, b) \in E$ is a cut of the optimal partitioning P_T it holds that P_T consists of the cost-optimal subsolutions of subtrees T_b and $T \setminus T_b$. In the opposite case, P_T can be obtained as the combination of two subcases merged along the given edge, that is, $P_T \triangleq P_A \backslash p_a + P_B \backslash p_b + p_a \cup p_b$, where $f \in p_a \in P_A \in \mathcal{P}(T \backslash T_b)$, as well as $b \in p_b \in P_B \in \mathcal{P}(T_b)$.

Since any tree cut also results in (sub)trees, subcases of a subtree can be further decomposed recursively until trivial partition blocks of standalone functions are reached. Accordingly, starting from single leaf nodes all feasible sub-partitioning can be generated by merging or concatenating previously derived subcases, while the upper constraints M and L_π are validated in each iteration. The optimal substructure of our tree partitioning problem along with carefully chosen dominance rules allow us to use *dynamic programming* (DP) techniques to derive the optimal layout. In the following we propose DP-based tree partitioning algorithms tailored to our serverless layout model defined in Eq. (1)–(5) and discuss their applicability with respect to our platform computation model's block-internal parallelization capability.

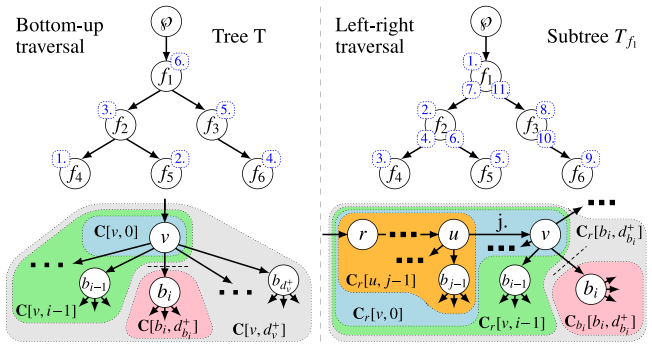


Fig. 5. Tree traversal methods used in our BUTP and LRTP algorithms with node visiting orders (blue marks) and subtree cases considered in each iteration.

5.1.1. Dynamic programming approach

The straightforward approach to process the tree is to choose one node $v \in F$ for which all its successors A_v have already been processed, form a trivial block $p_v = \{v\}$ and combine p_v with all subcases of its children $b \in A_v$, while considering both cutting and merging actions for the outgoing edges $(v, b) \in E$ as described in [38,40]. This approach leverages that merging only requires modifying the topmost partition blocks of the subcases, i.e., merging block p_v with the related block p_b for all subcase combinations of adjacent nodes v and b, whereas edge cut actions involve no modification.

This also reveals that only these topmost blocks can influence the merged subcase properties, such as the memory demand or introduced latency on the critical path π. To avoid maintaining an intractable number of sub-partitionings, we rely on two specific calculation reduction techniques. First, we observe that from two subcases with the same properties regarding their topmost blocks, the one with the lower cost can eventually be part of T's optimal partitioning, that is, it dominates the subcase with higher cost. Consequently, this dominance relation also holds for all the subcases generated from them. Hence, any subcase proved to be inferior can be eliminated on-the-fly to avoid redundant calculations in later steps.

However, the overall number of subcase combinations to be examined for all (v, b) edges still remains intractable. These calculations can also be reduced based on the observation that the objective and constraints are monotonic functions [41]. By this way, we can derive subcases with the lowest cost for each node v by iterating over its successor nodes $b_i : i = 1, 2, \dots, |A_v|$ in an arbitrary order and match b_i's subcases with v's subcases that are derived from the prior successors b_1–b_{i-1}. In formal terms, suppose a node v, a topmost block p_b, and the topmost block p_v of their merged subcases, our serverless model must satisfy that $\mathrm{c}_\sigma(p_b) \leq \mathrm{c}_\sigma(p_v)$, $\mathrm{m}_\sigma(p_b) \leq \mathrm{m}_\sigma(p_v)$ and $\mathrm{l}_\sigma(p_b) \leq \mathrm{l}_\sigma(p_v)$, respectively. For the memory demand, this always holds. But for the latency and cost metrics, we rearrange our formulas in Eqs. (2) and (5) by omitting the input fetching parts and explicitly adding the related externalization overheads to the related blocks of cut edges as well as the invocation delay δ to ensure monotonicity.

We also leverage that critical path can restrict only one successor's subcases in case of $v \in \pi$, otherwise it remains unchanged. However, ensuring latency for a path is challenging in the sense that it cannot be validated locally on topmost blocks. Thus, we propose tracking the latency metrics l and l_π for topmost block p_v and critical subpath covered by the subtree, i.e., $\pi \cap T_v$, separately. By this means, we can easily derive latency values from each other, while our basic dominance rule remains applicable.

Unfortunately, block-internal parallelization may break our problem's optimal substructure as in each merge step we actually replace the topmost block's barrier node from b_i to v. This could result in altered instance assignments to the vCPU cores in previously processed subcases since our model formulas fundamentally depends on the barrier node. However, in case of serialized executions, the crucial substructure precondition holds and this standard tree partition technique remains applicable.

$$\mathbf{C}[v,i]^-_{l,l_\pi,m} = \min_{\substack{0<m\leq M\\ 0\leq l'_\pi, l''_\pi \leq L_\pi}} \left\{ \mathbf{C}[v,i-1]_{l,l'_\pi,m} \;+\; \mathbf{C}[b_i, d^+_{b_i}]_{l'_\pi, l''_\pi, l_\pi - l''_\pi - d^\pi_{b_i} - \delta^\pi_{b_i}, 0} \right\}, \tag{9a}$$

$$\mathbf{C}[v,i]^+_{l,l_\pi,m} = \min_{\substack{0<m_v\leq M\\ 0\leq l', l'', l'_\pi, l''_\pi \leq L_\pi}} \left\{ \mathbf{C}[v,i-1]_{l',l'_\pi,m_v} \;+\; \mathbf{C}[b_i, d^+_{b_i}]_{l-\Delta^+_l, l_\pi - \Delta^+_\pi, m-\Delta^+_m} \;+\; \Delta^+_c \right\}, \tag{9b}$$

$$\mathbf{C}[v,i]_{l,l_\pi,m} = \begin{cases} \min\left\{\mathbf{C}[v,i]^-_{l,l_\pi,m},\ \mathbf{C}[v,i]^+_{l,l_\pi,m}\right\}, & \text{if } i > 0, \\ \mathrm{c}_\sigma(\{v\}), & \text{if } i = 0, m = \mathrm{m}_\sigma(\{v\}), l = l_\pi = \mathrm{l}_\sigma(\{v\}), \\ \infty, & \text{otherwise.} \end{cases} \tag{9c}$$

In summary, we formalize our dynamic programming-based tree partitioning with fully serialized executions ($N = 1$) as follows. Let $\mathbf{C}[v,i]_{l,l_\pi,m}$ represent the minimal-cost partitioning of node v and subtrees T_{b_i} of v's first i children, such that its topmost block p_v has $\mathrm{m}_\sigma(p_v) = m$, $\mathrm{l}_\sigma(p_v) = l$, and achieves latency l_π on the processed subpath $\pi \cap T_v$. We also explicitly store the optimal solution from all possible memory options with $m = 0$, such that $\mathbf{C}[v,i]_{l,l_\pi,0} = \min_m \mathbf{C}[v,i]_{l,l_\pi,m}$. Edge cut scenarios are formulated in Eq. (9a), in which the additional delay introduced on π stems from the conditional parameters of platform invocation delay $\delta^\pi_{b_i}$ and data overhead $d^\pi_{b_i}$ (yields 0 if $b_i \notin \pi$), whereas block metrics l and m remain unchanged.

For each combination we consider the minimal-cost subcases from v's prior $i-1$. child node and the optimal subcases of regarding subtree T_{b_i}, which is stored in b_i's last child node indexed by its out-degree $d^+_{b_i}$. In contrast, merge scenarios formalized in Eq. (9b) must derive the altered metrics of the topmost block, which are formalized by the dedicated Δ^+ variables. Due to the limitation of $N = 1$, the difference regarding the memory demand and partitioning cost are $\Delta^+_m = m_v$ and $\Delta^+_c = -2r_{b_i} d_{b_i}$ as it is stated in the proof of Theorem 1.

For the latency calculation, we rely on the simplified relation of $\mathrm{l}_\sigma(\{v\} \cup p_{b_i}) \triangleq \mathrm{l}_\sigma(\{v\}) + \left\lceil \frac{r_{b_i}}{r_v} \right\rceil \mathrm{l}_\sigma(p_{b_i})$ based on Eq. (4)–(5) and $N = 1$. In this respect, Δ^+_l and Δ^+_π can be derived from the latency metrics l'' and l''_π belonging to b_i's subcase, in constant time. In Eq. (9c) we integrate the two scenarios, as well as define trivial solutions for the initial cases $\mathbf{C}[v,0]_{l,l_\pi,m}$ of single nodes without subtrees and for T's leaf nodes ($i = 0$). In order to guarantee that all children nodes are preprocessed during the dynamic programming iterations, we apply postorder DFS traversal that processes the tree's nodes in a bottom-up fashion.

Thus, the cost of T's optimal solution equals $\min_{l,l_\pi} \mathbf{C}[f_1, d^+_{f_1}]_{l,l_\pi,0}$. Partition blocks can be managed simultaneously with costs based on the tracking of the barrier nodes. Provided that $\mathbf{B}_v$ and $\mathbf{B}_{b_i}$ denote the barrier nodes of combined subcases, edge cut results in $\mathbf{B}_v = \mathbf{B}_v \cup \mathbf{B}_{b_i}$, whereas subcase merging uses the adjusted formula $\mathbf{B}_v = \mathbf{B}_v \cup \mathbf{B}_{b_i} \setminus \{b_i\}$. In conclusion, the complexity of our Bottom-Up Tree Partitioning algorithm (BUTP) is defined in Theorem 2.

Theorem 2. *The BUTP algorithm's complexity is* $\mathcal{O}(nM^2L_\pi^2)$.

Proof. Regarding the DFS traversal and the combinatorial iteration of each node's children, the algorithm visits all $n-1$ edges only once. For each edge, we combine merge and cut subcases based on the different memory demands of the topmost blocks, which can lead to $M \times M$ comparisons in the worst case, assuming integer block properties.

In contrast, latency metrics impose only L_π comparisons as there exists only one π-restricted (v, b_i) edge for each node, where v's subcases have the same value for l and l_π, while b_i's subcases can take L_π different values. In other combinations or in case of $v \notin \pi$, both metrics are 0. Hence, for each edge there could be $M^2 \times L_\pi \times L_\pi$ subcase combinations in the worst case for both cut and merge scenarios. In conclusion, BUTP's computational complexity sums up to $\mathcal{O}(nM^2L_\pi^2)$. ■

The advantage of this recursive algorithm lies in that block metrics can be calculated based on block nodes with any monotonic function as in the CfgLP model, but without any linearity restriction as in the Mtx-ILP model. However, BUTP's complexity is pseudo-polynomial, i.e., it depends on the upper limits M and L_π that are not bounded by the input size n. Moreover, it can only be applied to fully serialized models. Therefore, we propose enhancements to this approach to improve the performance and bypass the limitation of serialized executions.

5.1.2. Left–right tree traversal

In order to circumvent the major challenge of barrier node replacements in BUTP, we leverage a left–right traversal method of the tree's nodes, which was originally proposed in [35] for reducing the number of comparisons during node iterations. With this technique, however, we are able to derive subcase merging by extending the topmost block based on one of its egress edges, instead of the block's single ingress edge.

More specifically, we rephrase our partitioning problem by considering a node v and iteratively merging the topmost block p_v with one of its direct successor nodes along a block-egress edges from $E^+_{p_h}$, whereas for edge cuts we still rely on subtree partitions derived a priori. By this way, we can realize a top-down direction of block extension where the barrier node is fixed, opposite to the bottom-up direction of merging one block to the barrier node's successor in BUTP. This allows us to fully leverage our platform computation models with internal instance parallelization along with the benefits of simplified merging and reduced iterations.

To integrate the left–right tree traversal with our serverless model formulas, we propose a dedicated procedure for calculating the required subcase parameters on-the-fly while traversing the tree's edges. For this purpose, we apply the notation $\mathbf{T}_r[v,i]$ based on [35] to denote the subtree consisting of node $v \in T_r$, the subtrees $T_{b_1},\dots,T_{b_i}$ of v's first i children and each predecessor of v until node r. In Fig. 5 we also highlight the different subtrees considered for an iteration of our proposed algorithms. In our adjusted traversal, the following precedence rules are implemented:

(i) $\mathbf{T}_r[v,i] \prec \mathbf{T}_r[v,i+1]$,
(ii) $\mathbf{T}_r[b_i,d^+_{b_i}] \prec \mathbf{T}_r[v,i]$,
(iii) $\mathbf{T}_r[v,i-1] \prec \mathbf{T}_r[b_i,0]$,
(iv) $\mathbf{T}_v[v,d^+_v] \prec \mathbf{T}_u[u,d^+_u]$.

Here, u and b_i denote v's predecessor and one of its sorted successors, respectively. Fundamentally, rules (i) and (iii) designate the direction of block expansion, while rule (ii) defines backtrack steps in subtree T_r. Our added consistency rule (iv) guarantees that for a node r each dependent subtree $T_v : v \in T_r$ has been processed beforehand to support edge cut calculations. The traversal steps are presented in Algorithm 1, where we exploit that rule (iv) basically corresponds to the postorder DFS traversal and rule (ii) can be matched with the specific backward steps in a DFS.

$$\mathbf{C}_r[v,0]_{l,m} = \mathbf{C}_r[u,j-1]_{l-\Delta_l,m-\Delta_m} + \Delta_c, \tag{10a}$$

$$\mathbf{C}_r[v,i]_{l,m} = \min_{\substack{0<m\le M\\ 0\le l\le L_\pi}} \left\{ \mathbf{C}_r[v,i-1]_{l-d^\pi_{b_i}-\delta^\pi_{b_i}-l_{b_i},m} + \mathbf{C}_{b_i}[b_i,d^+_{b_i}]_{l_{b_i},0},\ \mathbf{C}_r[b_i,d^+_{b_i}]_{l,m} \right\}. \tag{10b}$$

Similarly to BUTP, we apply the notation $\mathbf{C}_r[v,i]_{l,m}$ to mark the minimal cost of subtree $\mathbf{T}_r[v,i]$'s optimal partitioning. In Eq. (10a) we formalize the merge action of extending the topmost block p_r with node v along the edge $(u,v) \in E(T_r) : u \in p_r$. Hence, the values Δ_l, Δ_m and Δ_c represent the regarding increments based on the fixed barrier node r. It is important to mention that only the topmost block's latency value l needs to be maintained, as in every iteration we compare two subcases that actually cover the whole π-restricted subpath of T_v.

In Eq. (10b) we consider the different scenarios with respect to edge $(v,b_i) \in T_r$. For edge cut we concatenate the cost-minimal subcase of $\mathbf{T}_r[v,i-1]$ with the optimal solution of subtree T_{b_i} cached with $m=0$, otherwise a processed subcase is assessed in which node b_i belongs to the topmost partition p_r (based on the definition of $\mathbf{T}_r[b_i,d^+_{b_i}]$). The trivial subcase of a given subtree T_r, i.e., the singleton block of its barrier node r is initialized as $\mathbf{C}_r[r,0]_{\mathrm{l}_\sigma(\{r\}),\mathrm{m}_\sigma(\{r\})} = \mathrm{c}_\sigma(\{r\})$, while the optimal solution's cost is equal to $\min_l \mathbf{C}_{f_1}[f_1,d^+_{f_1}]_{l,0}$. The runtime complexity of our enhanced Left-Right Tree Partitioning (LRTP) is concluded as follows.

Algorithm 1 Left-Right (Hybrid) Tree Traversal

```
procedure LeftRightTreeTraversal(T, root)
    Initialize traversed parameter list params
    for all r ∈ PostorderDFS(T, root) do
        Initialize LIFO data stack with default item ⟨Null, Null, 0, r, Null⟩
        while stack is not empty do
            u_{j-1}, b_{i-1}, i, v, b_i ← stack.pop()
            params.add(⟨u_{j-1}, b_{i-1}, v, b_i⟩)        ▷ Store iteration parameters
            if i ≤ d_v^+ then                              ▷ Step to next existing child
                stack.push(⟨u_{j-1}, b_i, i+1, v, b_{i+1}⟩)   ▷ Re-push stepped item v
                stack.push(⟨b_i, Null, 0, b_{i+1}, Null⟩)     ▷ Push new item b_{i+1}
    return params
```

Theorem 3. *The LRTP algorithm's complexity is* $\mathcal{O}(n^2 M L_\pi)$.

Proof. According to our left–right traversal method applied to every subtree of T, the required steps can be bounded by $\mathcal{O}(n^2)$. In each iteration, we either extend block p_r by merging $M \times L_\pi$ subcases with node b_i (Eq. (10a)), or compare block p_r introducing one latency value regarding the subpath $\pi \cap [r \to v]$ with $M \times L_\pi$ different optimal partitioning of subtree T_{b_i} (Eq. (10b)). Therefore, the number of LRTP's required steps equals $\mathcal{O}(n^2 M L_\pi)$. ■

Although LRTP reduces the performed steps regarding the unbounded variables M and L_π, it remains pseudo-polynomial. To mitigate our algorithm's sensitivity to these parameters, we also propose the following two methods.

First, we can observe that in spite of the serialized processing of (subtree) nodes, subcases of disjoint subtrees can be calculated in parallel. Relying on the recursive substructure of both tree partitioning problems, we can apply our algorithms on separate parts of T in a multithreaded or multiprocessed manner where partial solutions can be synchronized among the algorithm instances on-demand. In addition, our formulas in Eqs. (9) and (10) enable the implementations to only keep subcases of the related successors for each traversed node to diminish their memory footprints. However, this enhancement highly depends on the parallelization capabilities (CPU cores) and T's exact structure (the wider, the better).

From a different viewpoint, the cached subcases can be truncated in place with a generalized dominance rule to reduce the algorithms' calculation steps. In general, a subcase is also dominated if there exists another one for that all stored cost, memory and latency metrics are lower than or equal to the former. In our added algorithmic step referred as *bidirectional elimination*, all previously calculated subcases are revalidated with this general dominance rule for each subcase admitted by the principal rule. Although these additional checks can introduce a quadratic factor into the asymptotic runtimes, we argue that distinguishing and eliminating unavailing subcases is beneficial in most cases.

5.1.3. Bicriteria approximation scheme

Our recursive problem formulations and dynamic programming approaches in Sections 5.1.1–5.1.2 allow for an efficient construction of polynomial-time approximation algorithms by eliminating the unbounded runtime parameters that make our BUTP and LRTP algorithms prone to state space explosion [42]. This can be achieved by utilizing different combinatorial state trimming techniques, e.g., value rounding

or interval partitioning, to merge cached subcases according to the dominance rule, while keeping the introduced additive errors under control [43,44].

Unfortunately, due to the multimodal nature of our runtime complexities containing M and L_π along with the possibility of multidimensional block constraints described in Section 3.3, these techniques cannot be used directly. However, we can leverage that these constraints need to be enforced differently. While flavors pose exact memory limits that cannot be violated otherwise overbooked components are terminated by the serverless framework, user-defined latency constraints based on average measurements of the volatile cloud platform might be exceeded to a tolerable extent.

In this respect, we propose a bicriteria fully polynomial-time approximation scheme based on our LRTP algorithm with serialized instance executions ($N = 1$), which approximates the optimal solution's objective with a fixed factor ε and may violate the latency limit L_π with a fixed factor λ, while its runtime complexity is polynomial in the input size n as well as in the approximation ratios $\frac{1}{\varepsilon}$ and $\frac{1}{\lambda}$. More precisely, we transform our relaxed problem into an equivalent maximization variant that derives a partitioning with objective value at most $(1-\varepsilon)$–times the optimal solution and with critical path latency at most $(1+\lambda)L_\pi$, such that $0 < \varepsilon < 1$ and $0 < \lambda$.

First, we construct the maximization variant of our relaxed problem. Based on the proof of Theorem 1, the objective is equivalent to minimize the aggregated data transmission overhead of cut edges regarding $N = 1$, that is, $\min \sum_{b \in B_{P_T}} r_b d_b$. We can observe that this can be reformulated to the maximization of eliminated data overheads, i.e., $\max \sum_{f \notin B_{P_T}} r_f d_f$, since edges are basically bisected into block-internal and external calls. Additionally, we also rearrange our primary dominance rule by swapping the cost with the topmost block's memory metric. By this way, we keep the subcase with smaller memory $\mathbf{M}_r[v,i]_{w,l}$ provided two subcases have the same data weights w (instead of cost) and latency value l within an iteration. It is easy to see that this formulation still results in an optimal solution, while it also allows for tracking multidimensional block constraints along with memory and applying trimming methods based on both w and l, simultaneously.

For latency values, we utilize rounding on demand, using the scaling factor $\frac{n}{\lambda L_\pi}$ for each derived latency metric l to calculate the rounded value $\hat{l} = \lceil \frac{nl}{\lambda L_\pi} \rceil$, similarly to the approaches in [45]. For the covered weights, we apply interval partitioning based on [43]. In each iteration of $\mathbf{M}_r[v,i]$, the highest weight value $W_r^{v,i}$ is obtained and used to divide the interval $[0, W_r^{v,i}]$ into $\lceil \frac{n}{\varepsilon} \rceil + 1$ uniform sub-intervals (except the last one) based on the adjusted sub-interval size $\frac{\varepsilon}{n} W_r^{v,i}$. Subsequently, our primary dominance rule (possibly with our bidirectional elimination) is applied to all subcases in each $\hat{w}$. subinterval having the same $\hat{l}$ value to eliminate all but one of them.

Hence, we restrict the number of subcases by keeping only one per subintervals, i.e., $\mathbf{M}_r[v,i]_{\hat{w},\hat{l}}$, while we introduce bounded rounding errors in our cumulative models in each iteration with respect to the interval sizes $\frac{\lambda}{n} L_\pi$ and $\frac{\varepsilon}{n} W_r^{v,i}$. The runtime complexity of our Bicriteria Fully Polynomial-Time Approximation Scheme (BiFPTAS) is defined as follows.

Theorem 4. *The BiFPTAS algorithm's complexity is* $\mathcal{O}(\frac{n^4}{\varepsilon\lambda})$.

Proof. Latency rounding can introduce at most $e_b^l = \frac{\lambda}{n} L_\pi$ error for both cut and merge decisions of an edge $(v,b) \in E$ according to the increments $d_{b_i}^\pi$, $\delta_{b_i}^\pi$ and Δ_l in Eq. (10), while it guarantees that the subcases can take only $\lceil \frac{n}{\lambda} \rceil + 1$ distinct latency values.

Similarly, interval partitioning can make $e_{b_i}^w = \frac{\varepsilon}{n} W_r^{v,i}$ additive error in each iteration, while it manages only $\lceil \frac{n}{\varepsilon} \rceil + 1$ distinct values. Suppose that the optimal solution's weight is W^*, for the aggregated errors of the derived solution it holds that $\sum_{|\pi|} e_b^l \leq \lambda L$ and $\sum_{b \in T} e_b^w \leq \varepsilon W^*$ since $\forall (v,b_i) \in T : W_r^{v,i} \leq W^*$.

Algorithm 2 Greedy Tree Partitioning Algorithm (GrTP)

```
procedure GreedyTreePartitioning(T(F, E), π, M, L_π, σ)
    Initialize empty partitioning and multicut sets P = C = ∅
    for i ∈ |π| do                                   ▷ Perform critical path splitting
        Initialize array l⃗ for π's nodes with values of ∞
        for all c ∈ π \ C do                         ▷ Iterate over viable cuts
            l⃗[c] ← GetPathLatencyFromFeasibleMultiCut(π, M, σ, C ∪ {c})
        C ← C ∪ {argmin {l⃗}}                        ▷ Get min-latency multicut
        if min {l⃗} ≤ L_π then
            break                                    ▷ Feasible π-multicut
    if min {l⃗} ≥ L_π and |C| = |π| then
        return ∅                                     ▷ Infeasible solution
    B ← {f_1} ∪ C                                    ▷ Set initial barrier nodes
    Assign new edge labels as e_f ∈ E ↦ r_f d_f
    while B ≠ ∅ do                                   ▷ Perform M-bounded partitioning
        b ← B.pop()
        p_b ← {b};  neighbs ← E+_{b} \ C             ▷ Filter out π-cuts in C
        while neighbs ≠ ∅ do
            v ← GetNodeWithMaxIngressLabel(neighbs)
            if m_σ(p_b ∪ {v}) > M then               ▷ Overloaded block
                break
            p_b ← p_b ∪ {v};  neighbs ← neighs ∪ E+_{v} \ C   ▷ Update block
        P ← P ∪ {p_b};  B ← B ∪ E+_{p_b}              ▷ Update partitioning
    return P
```

In summary, each node iteration has to perform at most $\frac{n^2}{\varepsilon\lambda}$ combinations, resulting in $\mathcal{O}(\frac{1}{\varepsilon\lambda} n^4)$ algorithmic steps due to the quadratic left–right tree traversal. ■

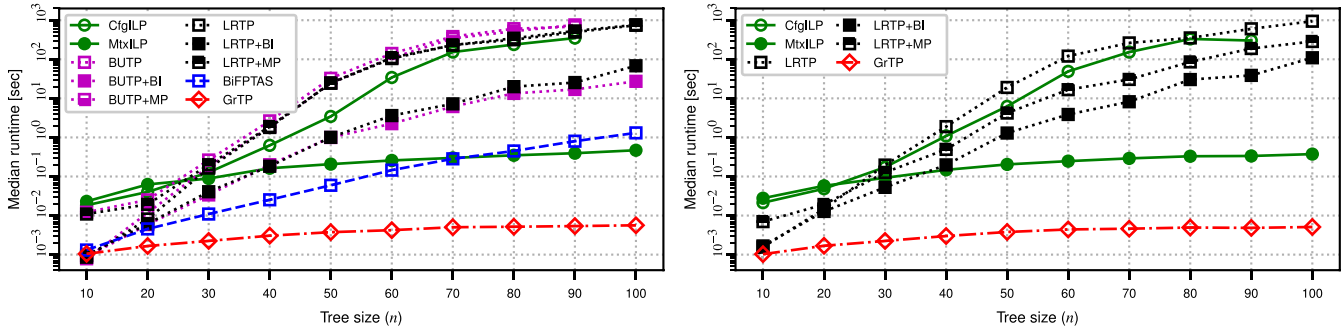
Our BiFPTAS algorithm relies on serialized instance executions to be able to reformulate the objective function for subcase trimming. However, our bicriteria approach could also be leveraged for $N > 1$ provided an FPTAS is constructed from our models and minimized objective. We leave this for future work.

5.2. Greedy heuristic

For comparison purposes, we provide a greedy heuristic algorithm that can derive a feasible low-cost function partitioning with respect to the memory and latency constraints M and L_π and to the block-internal parallelization of function instances by iteratively appending edges to a feasible multicut. These greedy edge-cut decisions can rely on the general observation that cost-optimal layouts typically have a small number of external edges due to the introduced data and latency overheads. Therefore, an initial partitioning can be improved by merging external edges having the largest data load with respect to the extended block's memory.

In our approach, we begin with the fix barrier node f_1 (T's root node) and the related topmost block p_{f_1}, and successively merge block-egress edges with the largest data load until the given block's memory $\mathrm{m}(p_{f_1}) \leq M$. We store the non-extendable block p_{f_1}, mark its egress edges as cuts, and continue with an arbitrary barrier node made by the cut edges. Fundamentally, this describes a *Greedy Best-first Search* (GBS) algorithm, where the highest-weight unseen edge is always considered first. Since in GBS each edge is examined only once, we can obtain a memory-limited partitioning in $\mathcal{O}(|E|)$ steps.

However, guaranteeing the latency limit on the critical path requires additional precalculations. In our case, we iteratively assign a cut to each possible edge on π and recalculate the whole path's latency. If we find a feasible setup, the critical path cuts are used as predetermined barrier nodes for the memory-limited partitioning. If no cut assignment meets L_π and M, we select the cuts achieving the lowest latency and repeat the calculations with an additional edge. As the multicut size increases in each iteration, we either find a proper π-multicut at most $\frac{1}{2}|\pi|(|\pi|-1)$ steps or consider the given problem unsolvable due to



(a) Runtime performance of models with serialized function execution ($N = 1$). (b) Runtime performance of models with block-internal parallelization ($N = 2$).

Fig. 6. Median execution times of our proposed models in Table 2, measured with our Random tree dataset.

the too strict constraint L_π. Our joint algorithmic steps are provided as pseudocode in Algorithm 2. In summary, our greedy tree partitioning algorithm (GrTP) can find a feasible, low-cost tree partitioning, although at the cost of suboptimal merge and cut decisions. Its runtime complexity is specified in Theorem 5.

Theorem 5. *The GrTP algorithm's time complexity is* $\mathcal{O}(n^2)$.

Proof. It follows directly from the linear M-bounded tree partitioning and the quadratic path-splitting subproblems, assuming that all block metrics can be derived in $\mathcal{O}(1)$ time. ■

Although our GrTP algorithm provides a feasible and low-cost partitioning without explicit cost calculation, it guarantees neither optimality nor an upper bound on the approximation ratio. For the sake of comparability, we summarize the properties, benefits, and limitations of our proposed solutions in Table 2.

6. Evaluation

In this section, we evaluate our proposed serverless layout optimization algorithms with synthetically generated inputs and compare their performance to each other and to state-of-the-art solutions. Tests are conducted on AMD EPYC-Rome 2.4 GHz CPUs and 64 GB memory, with Python 3.11 and CPLEX 22.11.

6.1. Test case configuration

During the experiments we automatically aborted test cases after 1000 s and considered the tested algorithm insufficient for the given tree size. Accordingly, median values of the tracked performance metrics are recorded to avoid biased estimations. To unify the interested algorithmic parameters M and L_π, whose applicability depend on the actual node/edge parameters and the underlying tree structure itself, we define two ratios $\rho_M, \rho_{L_\pi} \in [0, 1]$ to designate the upper limits of block memory and critical path latency on tree-specific intervals calculated for each input tree T, separately.

By definition, memory constraints are calculated based on the theoretical lowest and highest values, i.e., $M = \lceil (1 - \rho_M) M_T^{min} + \rho_M M_T^{max} \rceil$, such that $M_T^{min} \triangleq \max_{f \in F} m_f$ marks the lowest memory value of singleton blocks and $M_T^{max} \triangleq \sum_{f \in F} m_f$ marks the memory demand of the all-merged subcase ($N \geq 1$). Latency limits are derived in the same way, but instead of calculating exact interval bounds we estimate L_π^{max} and L_π^{min} by considering the all-merged and all-cuts subcases of path π, i.e., $L_\pi^{max} = \mathrm{l}_\sigma(\pi)$ and $L_\pi^{min} = \sum_{v \in \pi} \left\{ \mathrm{l}_\sigma(\{v\}) + \delta \right\}$.

The following default settings are defined for each test cases: $\rho_M = \rho_{L_\pi} = 0.2$ (moderately strict limits), $N = 1$ for serialized executions and $N = 2$ for paralleled function executions, $\varepsilon = \lambda = 0.5$ (permissive approximation ratios), and tree size $n = 40$.

6.2. Test input generation

To be able to conduct comprehensive experiments with arbitrary tree sizes, but also to comply with particular characteristics of microservice/serverless applications published in the literature, we implemented three different input tree generation methods. The following techniques can generate synthetic trees bearing high resemblance to public-cloud measurements both in their function/invocation metrics and the call graph structure. For each test case, we randomly generated 100 input trees to cover the wide spectrum of diverse application characteristics.

Random trees are constructed from randomized Prüfer sequences of size $n - 2$, while node/edge attributes are uniformly drawn from preconfigured intervals, such that $t_f \in [10, 500]$ ms, $m_f \in [64, 512]$ MB, $d_f \in [10, 100]$ ms, and $r_f \in [1, 10]$ $\frac{1}{s}$. By this way, random trees equally contain wide and deep trees as well as slight and large data overheads with varying frequencies.

Job trees are generated based on the findings of data-parallel job executions published in [46], where researchers investigated anonymous call traces recorded in Alibaba cloud and proposed a tool to create dependency DAGs based on the original records extended with oversampling. We rely on this generator to create trees by iterative duplicating subgraphs having more than one ingress call and distribute the call rate accordingly, whereas task completion times and memory demands are reused directly.

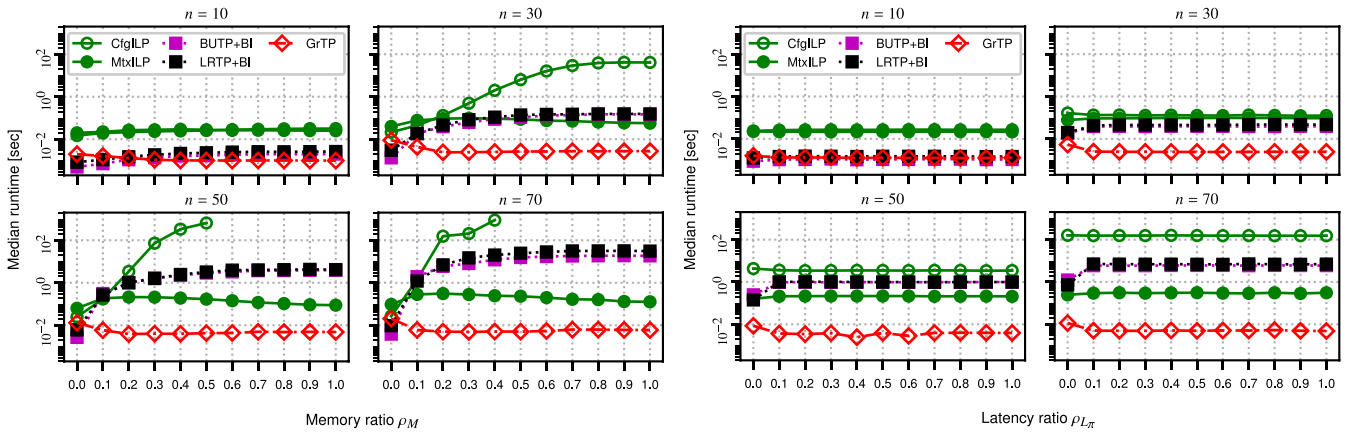
For data overhead, we derive values from an exponential distribution, where its expected value is proportional to the average task runtime by a configurable factor (0.2 in our case), according to the heavy-tailed performance of in-memory datastores depicted in [11].

Hence, our job tree generator constructs trees that sufficiently describe the data-parallel execution of scientific and IoT applications. These converted trees consist of duplicated chains of functions due to the DAG-to-tree conversion, which typically results in deep trees with relatively small out-degrees.

FaaS trees are assembled based on fitted and empirical distributions of call graph degrees and node/edge parameters published in the literature. For the tree's architecture, we leveraged the invocation model from [47], which extends Barabási-Albert random graphs with power-law degree distributions. According to our requirements and parameters evaluated in that paper, in our implementation we used the following configuration $m = 1$, $\alpha = 0.9$, and $a = 3.25$.

For node attributes, we leveraged best-fit distributions of average function execution times (Log-Normal with $\mu = -0.38$, $\sigma = 2.36$) and average memory demands (Burr with $c = 11.652$, $k = 0.221$, $\lambda = 107.083$) as revealed in [20].

In case of edge attributes, we extracted dedicated empirical distributions from cloud native call traces published in [18] to obtain knowledge on invocation frequencies specifically within a call graph and on R/W overheads of in-memory datastore usage.



(a) Median execution times measured with increasing memory limits M.

(b) Median execution times measured with increasing latency limits L_π.

Fig. 7. Sensitivity analyses of our primary models based on varying unbounded algorithm parameters, executed with our Random tree dataset.

Table 2
Summary of our proposed serverless layout optimization solutions.

Alg.	Sec.	Method	Runtime	Optimal	Limitations
CfgILP	4	ILP	Solver dep.	✓	Intractable problem size
MtxILP	4	ILP	Solver dep.	✓	Fully linear model
BUTP	5.1.1	DP	$\mathcal{O}(nM^2L_\pi^2)$	✓	Monotonicity, $\|\Phi\|$=1, N=1
LRTP	5.1.2	DP	$\mathcal{O}(n^2ML_\pi)$	✓	Monotonicity, $\|\Phi\|$=1
BiFPTAS	5.1.3	DP	$\mathcal{O}(n^4\frac{1}{\lambda\varepsilon})$	–	Monotonicity, $\|\Phi\|$=1, N=1
GrTP	5.2	GBS	$\mathcal{O}(n^2)$	–	No approx. ratio, $\|\Phi\|$=1

By this joint model, we can generate synthetic serverless trees that characterize microservice and serverless applications typically having wider structures and higher out-degrees than job trees. It is noteworthy that these serverless tree generators are also made open source as part of our software package.

6.3. Performance validation experiments

In this section, we evaluate the runtime performance of our algorithms listed in Table 2 with their proposed improvements. In CfgILP we utilize the same reversed DFS and subcase combination method as in BUTP to only consider decision variables with respect to memory limit M, whereas MtxILP models are assembled systematically from Eq. (8). Consequently, their executions include the Python-based model creation and the external solution substeps, the latter of which is performed with the CPLEX solver.

Both pseudo-polynomial models (BUTP and LRTP) are configured with multiprocessed (MP) and bidirectional elimination (BI) enhancements detailed in Section 5.1.2. For the MP extension, we use a bottom-up, brute-force tree traversal approach to divide input trees into k disjoint subtrees for the initiated subprocesses. In our tests, $k = \sqrt{n}$ empirically proved to have the best results. In BiFPTAS, bidirectional elimination is integrated into the ε-based subcase trimming step by default.

In Fig. 6, we summarize the runtime measurements of both the fully serialized and block-internal parallelization test cases. Based on the slope of runtime curves in Fig. 6(a), our algorithms can be divided into four identifiable groups. The best-scaling result is provided by our greedy heuristic (GrTP), deriving function groups in several milliseconds even for trees of size $n = 100$, while MtxILP and BiFPTAS models also offer scalable performance with subsecond execution times. Although on small input MtxILP has the worst performance due to its non-negligible model creation time, it improves quickly ($n \geq 30$) compared to other models as the input size increases and the benefits of ILP solvers start to prevail.

In contrast, pseudo-algorithm variations and CfgILP have worse performance that originates from their exponentially large state space. However, BI enhancements proved to be beneficial despite the added elimination steps, reducing computational overhead by an order of magnitude. In the majority of test cases, distributed layout calculations (MP) cannot achieve any gain that can compensate for the introduced IPC overheads of transferring the separately derived subresults.

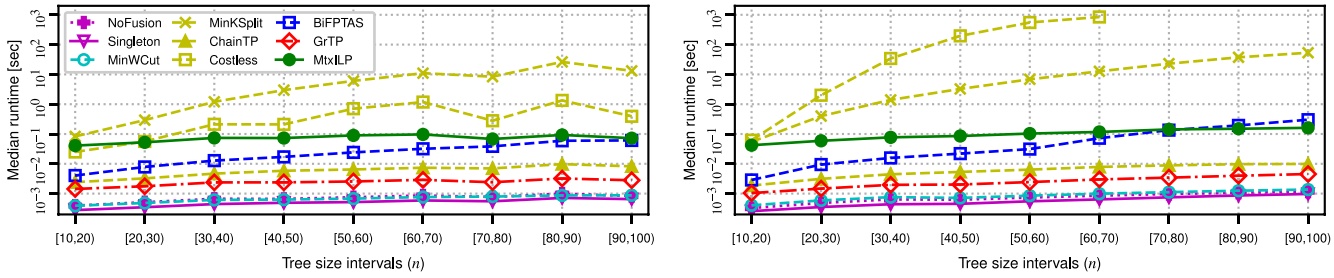
On top of that, the basic nonenhanced models (CfgILP, BUTP, and LRTP) even fail in successfully calculating any solution of the largest inputs ($n = 100$) in our given time deadline. These observations are completely in line with the related model complexities and also hold for models with block-internal parallelization ($N \geq 1$) capability, whose results are shown in Fig. 6(b).

To further assess computational performance with respect to user and platform constraints, we also performed random tree experiments with varying algorithmic parameters M and L_π by iterating over the related intervals $\rho_M \in [0, 1]$ and $\rho_{L_\pi} \in [0, 1]$ at different tree sizes n. These test results are depicted in Fig. 7, where we focus on the well-performing algorithms of the prior tests and apply bidirectional elimination in each setup.

It can be observed that flavor selection, including block constraint M, has greater influence on the runtime performance of our models than changes in the user-given latency limit L_π. More precisely, our exact algorithms show significant sensitivity to the increase of M, which becomes more severe as the input tree size grows (Fig. 7(a)). This originates from the fact that the higher the upper block limit M, the more subcases have to be taken into account. In case of CfgILP, this limitation lies in its tree traversal-based model creation subtask. Interestingly, MtxILP shows a minor decline in larger trees, which comes from the better efficiency of the underlying LP solver on larger models.

For the latency limit L_π, the same phenomena can be observed in Fig. 7(b) but only to a lesser extent, as the size of the critical path π depends more on the tree's structure than on its size. In contrast, GrTP has a larger runtime overhead with stricter constraints ($\rho_M, \rho_{L_\pi} \leq 0.2$). This stems from that the smaller the upper limits, the more greedy iterations must be performed in general to find a feasible multicut of π and also to generate M-saturated blocks, whose number is inversely proportional to M (Section 5.2).

In addition, we also performed sensitivity tests of algorithm-specific parameters regarding the two approximation factors ε and λ. We experienced runtime increase only in cases of strict λ values where the critical path's length was large with respect to the graph structure and size (deep and narrow trees). This confirms our general argument that tolerating latency violation even for a small degree can be advantageous in reducing the complexity of our layout optimization problem. Moreover,



(a) Runtime of examined methods measured with our Job tree dataset ($n = 40$). (b) Runtime of examined methods measured with our FaaS tree dataset ($n = 40$).

Fig. 8. Runtime performance of our best-performing models compared to baseline and state-of-the-art solutions, executed on our Job and FaaS tree datasets.

we also evaluated the performance of our block-internal parallelization models on different numbers of vCPU cores ($N \geq 1$). Based on our results, the core count N has no effect on performance.

In conclusion, our efficient algorithms GrTP and BiFPTAS deliver good performance even on large inputs without suffering from runtime degradations introduced by unbounded algorithm parameters M, L_π and N. However, our MtxILP model can also be a suitable choice for large trees assuming that internal layout metrics of the considered serverless framework remain linear.

6.4. Cost reduction experiments

To highlight the cost-reduction capabilities of our optimization algorithms, we conduct comparative experiments with baseline solutions and the most relevant techniques from the literature. First, we consider two trivial cases of no function fusion (NoFusion) and the combination of all functions into one single block (Singleton). For a straightforward partitioning without any constraints, we implemented a greedy tree splitting method into subchains (MinWCut) based on [33]. It iteratively merges a node with only one of its children with the largest ingress data weight, similar to GrTP, to build subchain blocks with minimized data externalization.

For state-of-the-art assessments, we select and reimplement different solutions that approach our main SAC problem from different perspectives. The MinKSplit algorithm uses a min-weight, k-split tree clustering algorithm from [48] with $\mathcal{O}(n^3)$ complexity, to greedily iterate over k-size subtree partitionings with $k \in [n-1, \dots, 0]$ that meets the latency constraint L_π, although without any memory limit. We also validate one of our tree partition algorithms from [17]. The chain-based tree partitioning algorithm (ChainTP) can derive cost-minimal subchain blocks satisfying both M and L_π in $\mathcal{O}(n^3)$ time, although without regarding data externalization.

For these tests, we alter the input trees to incorporate data overhead in function runtimes according to Eq. (2), i.e, applying implicit data externalization for all the functions. Furthermore, we validate the performance of the Costless tool detailed in Section 2. In order to circumvent its chain limitation and apply it on trees, we re-implement its NP-hard constrained shortest path (CSP)-based model leveraging our metrics in Eq. (1)–(5), and iteratively execute it on all topological orderings of input trees. To obtain the exact solution, we use an CSP solver based on a specific bidirectional labeling algorithm [49].

These methods are compared with our non-exact algorithms BiFPTAS and GrTP based on the optimal solution of MtxILP. It is noteworthy that NoFusion, Singleton, and MinWCut always return a solution regardless whether it is feasible or not, whereas MinKSplit, ChainTP, and GrTP may prematurely declare the given partitioning problem unsolvable.

We compare the aforementioned techniques with each other based on their runtime performance along with cost and latency deviations of examined solutions from the optimal partitioning. Each test case is evaluated with our dedicated datasets of data-parallel Job and serverless FaaS trees. Since our Job dataset is assembled from real-cloud call graph traces with varying number of functions, we group our measurements based on intervals of tree sizes.

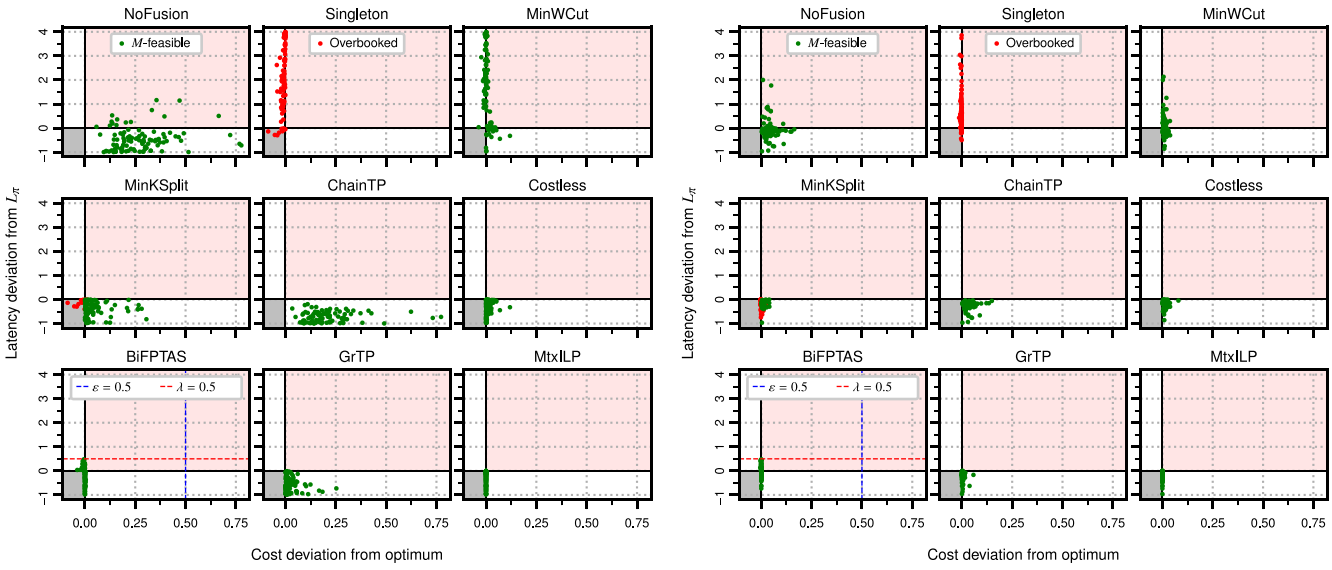
In Fig. 8, we present the measured execution times for both tree types. Besides the baselines and our tree traversal-based methods (BiFPTAS, GrTP), only ChainTP can achieve better (subsecond) performance than the exact MtxILP model. The other two state-of-the-art methods MinKSplit and Costless scale poorly in our FaaS tests in Fig. 8(b), which comes directly from their iterative substructure where multiple variations of the same partitioning problem are solved to comply with the upper limits or address all possible function combinations.

However, it can be observed in Fig. 8(a) that BiFPTAS, MinKSplit, and especially Costless take advantage of the characteristics of Job trees that typically contain long, parallel subchains and relatively small number of branches. These results also testify that GrTP and MtxILP models show no sensitivity to the input tree structure as these methods do not rely on consecutive combinations of numerous subcases. In this respect, BiFPTAS performs well in eliminating subcases as only negligible degradations can be measured in the runtimes of FaaS tree experiments compared to Job trees, with respect to the trimming factors ϵ and λ. Based on these runtime results, we can conclude that in general layout optimization becomes more challenging when the input is more complex in terms of the tree's node degree distribution.

In Fig. 9, test cases with valid outcomes are visualized in a two-dimensional coordinate system based on the signed deviations of resulted cost/latency values compared to the minimal-cost solution (MtxILP) and latency limit L_π, respectively. By definition, results in the positive quadrant (light red) violate the latency limit L_π, whereas the negative quadrant (light gray) cannot contain feasible solutions, as it would be a cheaper partitioning than the optimal with suitable latency. We also plot results violated memory constraint M with red markers, while M-feasible solutions are depicted in green. The most deviating results are given by the two baselines.

Essentially, NoFusion's results illustrate the efficiency of the function fusion technique itself, as careful grouping of functions improves the deployment layout in all cases, reaching up to 75% cost reduction with Job trees. On the contrary, grouping all functions in the Singleton model gives better results than the optimum in terms of cost, although with latency violation up to four times the limit L_π. Considering that NoFusion results can be enforced with $\rho_M = 0$ and Singleton results can be achieved only when $\rho_M = 1$, NoFusion results are mostly feasible partitionings, whereas Singleton solutions are infeasible due to our default setting $\rho_M = 0.2$. Interestingly, MinWCut provides quite similar, but M-feasible results, despite the lack of any constraint validation.

Other more elaborated models successfully satisfy the configured latency constraint. However, without implicit memory validation, significant part of MinKSplit's results are infeasible. The remaining two



(a) Measured deviations of derived partitionings with Job trees ($n = 40$).

(b) Measured deviations of derived partitionings with FaaS trees ($n = 40$).

Fig. 9. Cost–latency comparison of baseline, state-of-the-art, and our proposed methods compared to the optimal solution (MtxILP).

state-of-the-art models result in acceptable partitions, which also exemplify the trade-off between algorithmic runtimes and cost efficiency. Although ChainTP calculates more expensive partitioning than Costless, especially with Job trees, due to its internal subchain block constraint, it constructs partitioning multiple orders of magnitudes faster in polynomial time, i.e., in a more scalable manner.

Comparing our proposed algorithms with these results, in the vast majority of test cases GrTP provides similar or cheaper results that ChainTP or Costless with superior running time performance. However, the best cost-performance is achieved by our BiFPTAS compared to the optimal solutions of MtxILP, with more scalable runtimes than ChainTP or Costless. Moreover, BiFPTAS is able to derive partitioning even better than optimum within the limitation of our latency violation ratio $\lambda = 0.5$ (depicted in the upper left quadrant) as the only model among the examined methods.

Furthermore, we also evaluate cost–latency results of Job trees in multicore setups ($N = 2$) in Fig. 10. Similarly to prior experiments, par-Costless provides partitionings closest to cost-optimal solutions derived by (par)MtxILP, but also considers more partition problems unsolvable incorrectly than other models. In contrast, parGrTP can achieve similar (FaaS) or better (Job) performance than parChainTP, despite the fact that it calculates tree multicuts based on only data overheads instead of relying on a cost function utilizing N as in the case of parChainTP and parCostless.

Additionally, we can observe that all the examined methods provide closer-to-optimal solutions with FaaS trees. This stems from that model limitations, i.e., greedy node merging (GrTP), subchain blocks (ChainTP), topological ordering (Costless), can involve sub-optimal decisions at branches, which cause greater cost deviations on the decisively longer subchains of Job trees.

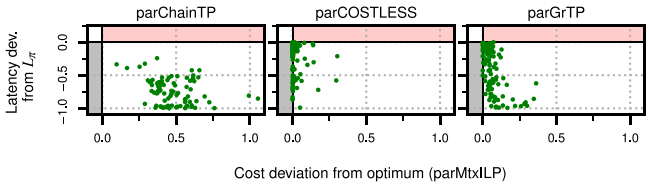


Fig. 10. Results of multicore partitionings ($N = 2$) with Job trees ($n = 40$).

7. Discussion

In our paper, we proposed models and algorithms for serverless application composition while making several assumptions by considering only a subset of platform features (async calls), deployment limitations (memory), and application characteristics (tree structure). Since these are all restrictive considerations by nature, the proved NP-hardness of our specific SAC problem implies, as one of our major findings, that the best we can hope to find efficient algorithms for more complex composition problems are well-designed heuristic approaches (unless P = NP).

Nevertheless, our algorithms proposed in this paper can be applied in many emerging serverless use cases despite the specified restrictive assumptions. For example, cost-efficient operation of QoS-aware IoT applications can benefit from our algorithms, in which video and other types of sensor data streamed to the cloud need to be extracted, transformed, and loaded (ETL) back to an intermediate data storage by different stateless functions at each stage [14,17]. In these cases, fluctuating call rates due to diurnal usage patterns and application-specific metrics can be tackled by our adaptive reoptimization loop.

Similar cost and QoS challenges arise in the serverless adaptation of compute and data-intensive scientific applications [50,51] and analytical jobs [25], where the cost-efficient deployment of chain and tree-shaped workflows is affected by the memory demands and intermediate data sizes of the stages. Similarly, offline construction of cloud-based AI inference applications can benefit from our algorithms in edge serverless environments, where the main issue is how to divide the stateless layers of oversized neural models into bounded deployable artifacts [28,52].

However, our proposed solution can form the basis for extensions along the assumptions to be applicable for more general serverless use cases, such as standard web applications. To support synchronous calls [15], idle times have to be taken into account in formulas where callers must wait for the called components to finish. Here our recursive tree algorithms prevail in contrast to ILP models, as all dependent subgraphs have been processed at each node iteration, making this feature straightforward to implement. To incorporate multiple constraints,

e.g., I/O bound, network bandwidth, maximum execution time, or deployment size, block constraint calculation $\mathrm{m}(p)$ and flavor limits M can be generalized to multidimensional vectors [17].

Although these constraints are clearly additive (deployment size) or monotonic functions of the number of parallel instances (I/O, network bandwidth, maximum execution time, ephemeral storage), they increase the complexity of our problem and may hinder the formulation of an efficient FPTAS. Regarding the separate stateless functions, another extension could be to consider a more integrated, multithreaded instance management instead of our general process-based execution design. This would require extra effort to derive the minimal total execution of function instances in a composite group, which poses the well-known NP-complete problem of task scheduling, also necessary to tackle. However, the interaction of instances, e.g., due to race conditions on shared hardware, can be easily added to our formulas.

Additionally, extending our composition problem to DAGs significantly increases its complexity. The serverless design essentially makes it possible to transform multi-rooted DAGs into trees by iteratively duplicating subgraphs along the nodes with multiple ingress edges in polynomial time, which we actually applied to construct Job trees in Section 6.2.

Nevertheless, another approach to tackle DAGs can be the adaptation of general graph partitioning heuristics with upper block bounds. Unfortunately, the vast majority of partitioning solutions target balanced partitioning with external edge weight minimization, such as variants of the Kernighan–Lin method, or aim to minimize the cardinality of partitioning with upper/lower bounds, or simply require the number of blocks beforehand [53]. Additionally, these approaches do not take into account any latency constraint or explicit instance execution model with fan-out invocation patterns. Hence, these heuristics cannot be applied directly to our SAC problem.

Currently, all of these aforementioned challenging extensions are in the main focus of our future work.

8. Conclusion

In this paper, we address the challenging task of cost-efficient serverless application composition with user-given QoS requirements. First, we proposed a wrapper-assisted execution model for a group of functions that enables the high-level composition of FaaS functions into deployable components, known as function fusion. In addition, we considered two performance enhancement extensions, i.e., block-internal management of intermediate states for reduced data transmission overhead and parallel execution of function instances within a block for better utilization of multicore resource flavors.

Based on our cost and performance models and inherent platform constraints, we defined the emerging optimization problem of minimizing operational cost of serverless applications with a tree structure, while also fulfilling the user-given latency constraint. We proved its NP-hard complexity and proposed exact algorithms to solve it utilizing integer linear programming and also dynamic programming with recursive tree traversal methods. To tackle our main problem efficiently, we also designed a greedy heuristic method and also a bicriteria approximation scheme that allows tolerable latency violations to achieve polynomial runtime.

Based on extensive simulation results, we can conclude that our greedy heuristic and bicriteria approximation scheme can achieve near-optimal cost reduction with superior runtime performance compared to state-of-the-art solutions. Moreover, we could further reduce deployment cost without any runtime degradation by allowing latency violation within a tolerable factor. Due to the transparent instance management of black-box serverless functions, our optimization methods can be applied in various use-cases, such as data-processing IoT applications, scientific workflows, or distributed AI/ML pipelines, just to mention a few. Furthermore, our dedicated execution models also pave the way for fast reoptimization capabilities, either implemented by the application developer as an integrated software component or offered by the cloud provider as a value-added service.

CRediT authorship contribution statement

János Czentye: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Balázs Sonkoly:** Conceptualization, Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Reference implementations of evaluated algorithms along with the used test inputs are made open source as a standalone software package [21].

Acknowledgments

This work was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects (i) no. 135074 under the FK_20 funding scheme, Hungary, (ii) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme, Hungary, (iii) 2019-2.1.11-TÉT-2020-00183 under the 2019-2.1.11-TÉT funding scheme, Hungary, and (iv) 2021-1.2.4-TÉT-2021-00058 under the 2021-1.2.4-TÉT funding scheme, Hungary.

On behalf of our work, we are also grateful for the possibility to use ELKH Cloud [54] that helped us achieve all the results published in this paper.

References

[1] J. Wen, Z. Chen, X. Jin, X. Liu, Rise of the planet of serverless computing: A systematic review, ACM Trans. Softw. Eng. Methodol. (2023) http://dx.doi.org/10.1145/3579643.

[2] S. Eismann, J. Scheuner, E.v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C.L. Abad, A. Iosup, The state of serverless applications: Collection, characterization, and community consensus, IEEE Trans. Softw. Eng. 48 (10) (2022) 4152–4166, http://dx.doi.org/10.1109/TSE.2021.3113940.

[3] AWS lambda, 2023, https://aws.amazon.com.(Accessed 15 June 2023).

[4] S. Lee, D. Yoon, S. Yeo, S. Oh, Mitigating cold start problem in serverless computing with function fusion, Sensors 21 (24) (2021) 8416, http://dx.doi.org/10.3390/s21248416.

[5] H. Yu, H. Zhang, J. Shen, Y. Geng, J. Wang, C. Miao, M. Xu, Serpens: A high performance faas platform for network functions, IEEE Trans. Parallel Distrib. Syst. (2023) 1–15, http://dx.doi.org/10.1109/TPDS.2023.3263272.

[6] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, A. Corradi, DIFFUSE: A distributed and decentralized PlatForm enabling function composition in serverless environments, Comput. Netw. 210 (C) (2022) http://dx.doi.org/10.1016/j.comnet.2022.108993.

[7] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, S. Bagchi, SONIC: Application-aware data passing for chained serverless applications, in: USENIX Annual Technical Conference, 2021, pp. 285–301.

[8] Z. Xu, L. Zhou, W. Liang, Q. Xia, W. Xu, W. Ren, H. Ren, P. Zhou, Stateful serverless application placement in MEC with function and state dependencies, IEEE Trans. Comput. (2023) 1–14, http://dx.doi.org/10.1109/tc.2023.3262947.

[9] N. Akhtar, A. Raza, V. Ishakian, I. Matta, COSE: Configuring serverless functions using statistical learning, in: IEEE Conference on Computer Communications, 2020, pp. 129–138, http://dx.doi.org/10.1109/infocom41043.2020.9155363.

[10] M. Szalay, P. Mátray, L. Toka, Real-time FaaS: Towards a latency bounded serverless cloud, IEEE Trans. Cloud Comput. 11 (2) (2023) 1636–1650, http://dx.doi.org/10.1109/TCC.2022.3151469.

[11] I. Pelle, J. Czentye, J. Dóka, B. Sonkoly, Towards latency sensitive cloud native applications: A performance study on AWS, in: IEEE 12th International Conference on Cloud Computing, 2019, pp. 272–280, http://dx.doi.org/10.1109/CLOUD.2019.00054.

[12] G. Safaryan, A. Jindal, M. Chadha, M. Gerndt, SLAM: SLO-aware memory optimization for serverless applications, in: IEEE 15th International Conference on Cloud Computing, 2022, pp. 30–39, http://dx.doi.org/10.1109/cloud55607.2022.00019.
[13] I. Baldini, P. Cheng, S.J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: Function composition for serverless computing, in: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2017, pp. 89–103, http://dx.doi.org/10.1145/3133850.3133855.
[14] I. Pelle, J. Czentye, J. Dóka, A. Kern, B.P. Gerő, B. Sonkoly, Operating latency sensitive applications on public serverless edge cloud platforms, IEEE Internet Things J. 8 (10) (2021) 7954–7972, http://dx.doi.org/10.1109/JIOT.2020.3042428.
[15] T. Schirmer, J. Scheuner, T. Pfandzelter, D. Bermbach, Fusionize: Improving serverless application performance through feedback-driven function fusion, in: IEEE International Conference on Cloud Engineering, 2022, pp. 85–95, http://dx.doi.org/10.1109/IC2E55432.2022.00017.
[16] B. Sigurleifsson, N. Ahmed, A. Verdet, M. Hamdaqa, M. Sabri, I. Pelletier, An approach for modeling the operational requirements of faas applications for optimal deployment, Inf.and Software Technol. 161 (2023) 107242, http://dx.doi.org/10.1016/j.infsof.2023.107242.
[17] J. Czentye, I. Pelle, B. Sonkoly, Cost-optimal operation of latency constrained serverless applications: From theory to practice, in: IEEE/IFIP Network Operations and Management Symposium, 2023, pp. 1–10, http://dx.doi.org/10.1109/NOMS56928.2023.10154412.
[18] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, C. Xu, An in-depth study of microservice call graph and runtime performance, IEEE Trans. Parallel Distrib. Syst. 33 (12) (2022) 3901–3914, http://dx.doi.org/10.1109/TPDS.2022.3174631.
[19] Redis, 2023, https://redis.io/. (Accessed 15 June 2023).
[20] M. Shahrad, R. Fonseca, Í.n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, in: Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, 2020, pp. 205–218.
[21] SLAMBUC, 2023, https://pypi.org/p/SLAMBUC. (Accessed 15 June 2023).
[22] C. Lin, N. Mahmoudi, C. Fan, H. Khazaei, Fine-grained performance and cost modeling and optimization for FaaS applications, IEEE Trans. Parallel Distrib. Syst. 34 (1) (2023) 180–194, http://dx.doi.org/10.1109/tpds.2022.3214783.
[23] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, S. Kounev, Sizeless: Predicting the optimal size of serverless functions, in: Proceedings of the 22nd International Middleware Conference, 2021, pp. 248–259, http://dx.doi.org/10.1145/3464298.3493398.
[24] Z. Zhou, Y. Zhang, C. Delimitrou, AQUATOPE: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows, in: 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 1, 2022, pp. 1–14, http://dx.doi.org/10.1145/3567955.3567960.
[25] J. Jarachanthan, L. Chen, F. Xu, B. Li, Astra: Autonomous serverless analytics with cost-efficiency and qos-awareness, in: IEEE International Parallel and Distributed Processing Symposium, 2021, pp. 756–765, http://dx.doi.org/10.1109/ipdps49936.2021.00085.
[26] L. Toka, The shape of your cloud: How to design and run polylithic cloud applications, IEEE Access 10 (2022) 97971–97982, http://dx.doi.org/10.1109/ACCESS.2022.3206433.
[27] T. Elgamal, A. Sandur, K. Nahrstedt, G. Agha, Costless: Optimizing cost of serverless computing through function fusion and placement, in: IEEE/ACM Symposium on Edge Computing, 2018, pp. 300–312.
[28] M. Yu, Z. Jiang, H.C. Ng, W. Wang, R. Chen, B. Li, Gillis: Serving large neural networks in serverless functions with automatic model partitioning, in: IEEE 41st International Conference on Distributed Computing Systems, 2021, pp. 138–148, http://dx.doi.org/10.1109/icdcs51616.2021.00022.
[29] Amazon CloudWatch, 2023, https://aws.amazon.com/cloudwatch/. (Accessed 15 June 2023).
[30] Prometheus, 2023, https://prometheus.io/. (Accessed 15 June 2023).
[31] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, et al., SOCK: Rapid task provisioning with serverless-optimized containers, in: USENIX Annual Technical Conference, 2018, pp. 57–70.
[32] I.E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, V. Hilt, SAND: Towards high-performance serverless computing, in: USENIX Annual Technical Conference, 2018, pp. 923–935.
[33] J. Czentye, I. Pelle, A. Kern, B.P. Gero, L. Toka, B. Sonkoly, Optimizing latency sensitive applications for Amazon's public cloud platform, in: IEEE Global Communications Conference, 2019, pp. 1–7, http://dx.doi.org/10.1109/GLOBECOM38437.2019.9013988.
[34] C. Lin, H. Khazaei, Modeling and optimization of performance and cost of serverless applications, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2021) 615–632, http://dx.doi.org/10.1109/TPDS.2020.3028841.
[35] D.S. Johnson, K.A. Niemi, On knapsacks, partitions, and a new dynamic programming technique for trees, Math. Oper. Res. 8 (1) (1983) 1–14, http://dx.doi.org/10.1287/moor.8.1.1.
[36] R. Schrader, Approximations to clustering and subgraph problems on trees, Discrete Appl. Math. 6 (3) (1983) 301–309, http://dx.doi.org/10.1016/0166-218x(83)90083-5.
[37] T. Öncan, A survey of the generalized assignment problem and its applications, INFOR: Inf. Syst. Op. Res. 45 (3) (2007) 123–141, http://dx.doi.org/10.3138/infor.45.3.123.
[38] J.A. Lukes, Efficient algorithm for the partitioning of trees, IBM J. Res. Dev. 18 (3) (1974) 217–224, http://dx.doi.org/10.1147/rd.183.0217.
[39] P.A. Jensen, Optimum network partitioning, Oper. Res. 19 (4) (1971) 916–932, http://dx.doi.org/10.1287/opre.19.4.916.
[40] J. Misra, R. Tarjan, Optimal chain partitions of trees, Inf. Process. Lett. 4 (1) (1975) 24–26, http://dx.doi.org/10.1016/0020-0190(75)90057-5.
[41] Y. Perl, Y. Shiloach, Efficient optimization of monotonic functions on trees, in: Lecture Notes in Comp. Science, Springer, 1981, pp. 332–339, http://dx.doi.org/10.1007/3-540-10828-9-73.
[42] G.J. Woeginger, When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? INFORMS J. Comput. 12 (1) (2000) 57–74, http://dx.doi.org/10.1287/ijoc.12.1.57.11901.
[43] S. Sahni, General techniques for combinatorial approximation, INFORMS Oper. Res. 25 (6) (1977) 920–936, http://dx.doi.org/10.1287/opre.25.6.920.
[44] O.H. Ibarra, C.E. Kim, Fast approximation algorithms for the knapsack and sum of subset problems, J. ACM 22 (4) (1975) 463–468, http://dx.doi.org/10.1145/321906.321909.
[45] A. Khan, E. Sharma, K.V.N. Sreenivas, Approximation algorithms for generalized multidimensional knapsack, 2021, ArXiv arXiv:2102.05854.
[46] H. Tian, Y. Zheng, W. Wang, Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud, in: Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 139–151, http://dx.doi.org/10.1145/3357223.3362710.
[47] V. Podolskiy, M. Patrou, P. Patros, M. Gerndt, K.B. Kent, The weakest link: Revealing and modeling the architectural patterns of microservice applications, in: Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering, 2020, pp. 113–122.
[48] M. Maravalle, B. Simeone, R. Naldini, Clustering on trees, Comput. Stat. Data Anal. 24 (2) (1997) 217–234, http://dx.doi.org/10.1016/s0167-9473(96)00062-x.
[49] C. Tilk, A.-K. Rothenbächer, T. Gschwind, S. Irnich, Asymmetry matters: Dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster, Eur. J. Oper. Res. 261 (2) (2017) 530–539, http://dx.doi.org/10.1016/j.ejor.2017.03.017.
[50] C.Q. Wu, X. Lin, D. Yu, W. Xu, L. Li, End-to-end delay minimization for scientific workflows in clouds under budget constraint, IEEE Trans. Cloud Comput. 3 (2) (2015) 169–181, http://dx.doi.org/10.1109/TCC.2014.2358220.
[51] C. Gou, A. Benoit, L. Marchal, Partitioning tree-shaped task graphs for distributed platforms with limited memory, IEEE Trans. Parallel Distrib. Syst. 31 (07) (2020) 1533–1544.
[52] J. Jarachanthan, L. Chen, F. Xu, B. Li, AMPS-Inf: Automatic model partitioning for serverless inference with cost efficiency, in: 50th International Conference on Parallel Processing, 2021, pp. 1–12, http://dx.doi.org/10.1109/ipdps49936.2021.00085.
[53] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, C. Schulz, Recent advances in graph partitioning, in: Alg. Eng., Springer, 2016, pp. 117–158, http://dx.doi.org/10.1007/978-3-319-49487-6_4.
[54] M. Héder, E. Rigó, D. Medgyesi, R. Lovas, et al., The past, present and future of the ELKH cloud, Inf. Társadalom 22 (2) (2022) 128, http://dx.doi.org/10.22503/inftars.xxii.2022.2.8.

János Czentye is currently a Ph.D. candidate at the Budapest University of Technology and Economics, under the supervision of Dr. Balázs Sonkoly. He received his M.Sc. from BME in Networks and Services with highest honors in 2014. He worked at the HSNLab participating in European research projects (FP7 UNIFY, H2020 5GEx) and gained wide knowledge about SDN/NFV, microservice and cloud technologies. His current research focuses on modeling, composition and provisioning of cloud-native applications in future networked systems, with a particular interest in IoT and AR/VR.

Balázs Sonkoly received the M.Sc. and Ph.D. degrees in computer science from BME in 2002 and 2010, respectively, where he is an Associate Professor and the Head of MTA-BME Network Softwarization Research Group. He has participated in several EU projects and he was the demo Co-Chair of ACM SIGCOMM in 2018, EWSDN in 2014 and 2015, IEEE HPSR 2015. His current research activity focuses on cloud/edge computing, NFV, SDN, and 5G.