Serverless Scheduling Policies based on Cost Analysis

Giuseppe De Palma[1], Saverio Giallorenzo[1,2], Cosimo Laneve[1],
Jacopo Mauro[3], Matteo Trentin[1,3], Gianluigi Zavattaro[1,2]
[1]Università di Bologna, Italy
[2]Sophia Antipolis, INRIA, France
[3]University of Southern Denmark

Current proprietary and open-source serverless platforms follow opinionated, hardcoded scheduling policies to deploy the functions to be executed over the available workers. Such policies may decrease the performance and the security of the application due to locality issues (e.g., functions executed by workers far from the databases to be accessed). These limitations are partially overcome by the adoption of APP, a new platform-agnostic declarative language that allows serverless platforms to support multiple scheduling logics. Defining the "right" scheduling policy in APP is far from being a trivial task since it often requires rounds of refinement involving knowledge of the underlying infrastructure, guesswork, and empirical testing.

In this paper, we start investigating how information derived from static analysis could be incorporated into APP scheduling function policies to help users select the best-performing workers at function allocation. We substantiate our proposal by presenting a pipeline able to extract cost equations from functions' code, synthesising cost expressions through the usage of off-the-shelf solvers, and extending APP allocation policies to consider this information.

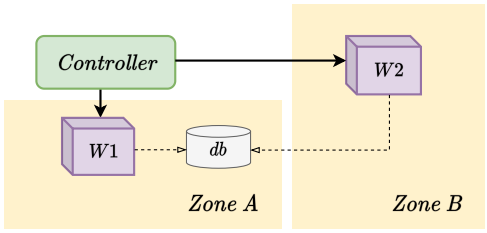
1 Introduction

Serverless is a cloud-based service that lets users deploy applications as compositions of stateless functions, with all system administration tasks delegated to the platform. Serverless has two main advantages for users: it saves them time by handling resource allocation, maintenance, and scaling, and it reduces costs by charging only for the resources used to perform work since users do not have to pay fur running idle servers [7]. Several managed serverless offerings are available from popular cloud providers like Amazon AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions, as well as open-source alternatives such as OpenWhisk, OpenFaaS, OpenLambda, and Fission. In all cases, the platform manages the allocation of function executions across available computing resources or workers, by adopting platform-dependent policies. However, the execution times of the functions are not independent of the workers since effects like *data locality* (the latencies to access data depending on the node position) can increase the run time of functions [6].

We visualise the issue by commenting on the minimal scenario drawn in Figure 1. There, we have two workers, W1 and W2, located in distinct geographical *Zones A* and *B*, respectively. Both workers can run functions that interact with a database (*db*) located in *Zone A*. When the function scheduler — the *Controller* — receives a request to execute a function, it must determine which worker to use. To minimise response time, the function scheduler must take into account the different computational capabilities of the workers, as well as their current workloads, and, for functions that interact with the database, the time to access the database. In the example, since W1 is geographically close to *db*, it can access *db* with lower latencies than W2.

APP [3, 2] is a declarative language recently introduced to support the *configuration of custom function-execution scheduling policies*. The APP snippet in Figure 1 codifies the (data) locality principle of the

Maurice H. ter Beek and Clemens Dubslaff (Eds.):
Proceedings of the First Workshop on Trends
in Configurable Systems Analysis (TiCSA'23)
EPTCS 392, 2023, pp. 40–52, doi:10.4204/EPTCS.392.3




```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
```

Figure 1: Example of function-execution scheduling problem and APP script.

example. Concretely, in the platform, we associate the functions that access *db* with a tag, called `db_query`. Then, we include the scheduling rule in the snippet to specify that every function tagged `db_query` can run on either `W1` or `W2`, and the `strategy` to follow when choosing between them is `best_first`, i.e., select the first worker in top-down order of appearance (hence giving priority to the worker `W1` if available and not overloaded).

By featuring customised function scheduling policies, APP allows one to disentangle from platform-dependent allocation rules. This opens the problem of finding the most appropriate scheduling for serverless applications. The approach currently adopted by APP is to feature only a few generic well-established strategies, like the foregoing `best_first`. The policies are selected *manually*, when the APP script is written, based on the developer's insights on the behaviour of their functions.

In this paper, we propose the adoption of automatic procedures to define function scheduling policies based on information derived with a static analysis of the functions. Our approach relies on three main steps: (*i*) the definition of code analysis techniques for extracting meaningful scheduling information from function sources; (*ii*) the evaluation of scheduling information by a(n off-the-shelf) solver that returns cost expressions; (*iii*) the extension of APP to support allocation strategies depending on such expressions. In particular, we discuss the applicability of our approach on a minimal language for programming functions in serverless applications.

We start in Section 2 by defining our minimal language called miniSL (standing for mini Serverless Language) which includes constructs for specifying computation flow (via `if` and `for` constructs) and for service invocation (via a `call` construct). Then, by following [5, 8], we describe in Section 3 how to exploit a (behavioural) type system to automatically extract a set of equations from function source codes that define meaningful configuration costs. In Section 3 we also discuss how equations can be fed to off-the-shelf cost analyser (e.g., PUBS [1] or CoFloCo [4]) to compute cost expressions quantifying over-approximations of the considered configuration costs. These expressions are then used in Section 4 to define scheduling policies in an extension of APP, dubbed cAPP. Finally, in Section 5 we draw some concluding remarks.

2 The mini Serverless Language

The mini Serverless Language, shortened into miniSL, is a minimal calculus that we use to define the functions' behaviour in serverless computing. In particular, miniSL focuses only on core constructs to define operations to access services, conditional behaviour with simple guards, and iterations.

Function executions are triggered by events. At triggering time, a function receives a sequence of invocation parameters: for this reason, we assume a countable set of *parameter names*, ranged over by

p, p'. We also consider a countable set of *counters*, ranged over by i, j, used as indexes in iteration statements. Integer numbers are represented by n; service names are represented by h, g, $\cdots$. The syntax of miniSL is as follows (we use over-lines to denote sequences, e.g., p_1, p_2 could be an instance of $\overline{p}$):

```
F ::= (p̄) => { S }
S ::= ε | call h(Ē) S | if (G) { S } else { S } | for (i in range(0,E)){ S }
G ::= E | call h(Ē)
E ::= n | i | p | E ♯ E
♯ ::= + | - | > | == | >= | && | * | /
```

A *function* F associates to a sequence of parameters $\overline{p}$ a statement S which is executed at every occurrence of the triggering event. *Statements* include the empty statement ε (which is always omitted when the statement is not empty); calls to external services by means of the `call` keyword; the conditional and iteration statements. The guard of a conditional statement could be either a boolean expression or a call to an external service which, in this case, is expected to return a boolean value. The language supports standard expressions in which it is possible to use integer numbers and counters. Notice that, in our simple language, the iteration statement considers an iteration variable ranging from 0 to the value of an expression E evaluated when the first iteration starts.

In the rest of the paper, we assume all programs to be well-formed so that all names are correctly used, i.e., counters are declared before they are used and when we use p, such p is an invocation parameter. Similarly, for each expression used in the range of an iteration construct, we assume that its evaluation generates an integer, and for each service invocation `call` h$(\overline{E})$, we assume that h is a correct service name and $\overline{E}$ is a sequence of expressions generating correct values to be passed to that service. Calls to services include serverless invocations, which possibly execute on a different worker of the caller.

We illustrate miniSL by means of three examples. As a first example, consider the code in Listing 1 representing the call of a function that selects a functionality based on the characteristic of the invoker.

```
1 ( isPremiumUser, par ) => {
2   if( isPremiumUser ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 1: Function with a conditional statement guarded by an expression.

This code may invoke either a `PremiumService` or a `BasicService` depending on whether it has been triggered by a premium user or not. The parameter `isPremiumUser` is a value indicating whether the user is a premium member (when the value is true) or not (when the value is false). The other invocation parameter `par` must be forwarded to the invoked service. For the purposes of this paper, this example is relevant because if we want to reduce the latency of this function, the best node to schedule it could be the one that reduces the latency of the invocation of either the service `PremiumService` or the service `BasicService`, depending on whether `isPremiumUser` is true or false, respectively.

Consider now the following function where differently from the previous version, it is necessary to call an external service to decide whether we are serving a premium or a basic user.

```
1 ( username, par ) => {
2   if( call IsPremiumUser( username ) ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 2: Function with a conditional statement guarded by an invocation to external service.

Notice that, in this case, the first parameter carries an attribute of the user (its name) but it does not indicate (with a boolean value) whether it is a premium user or not. Instead, the necessary boolean value is returned by the external service `IsPremiumUser` that checks the username and returns true only if that username corresponds to that of a premium user. In this case, it is difficult to predict the best worker to execute such a function, because the branch that will be selected is not known at function scheduling time. If the user triggering the event is a premium member, the expected execution time of the function is the sum of the latencies of the service invocations of `IsPremiumUser` and `PremiumService` while, if the user is not a premium member, the expected execution time is the sum of the latencies of the services `IsPremiumUser` and `BasicService`. As an (over-)approximation of the expected delay, we could consider the worst execution time, i.e., the sum of the latency of the service `IsPremiumUser` plus the maximum between the latencies of the services `PremiumService` and `BasicService`. At scheduling time, we could select the best worker as the one giving the best guarantees in the worst case, e.g., the one with the best over-approximation.

Consider now a function triggering a sequence of map-reduce jobs.

```
1 ( jobs, m, r ) => {
2   for(i in range(0, m)) {
3     call Map(jobs, i)
4     for(j in range(0, r)) {
5       call Reduce(jobs, i, j)
6     }
7   }
8 }
```

Listing 3: Function implementing a map-reduce logic.

The parameter `jobs` describes a sequence of map-reduce jobs. The number of jobs is indicated by the parameter `m`. The "map" phase, which generates `m` "reduce" subtasks, is implemented by an external service `Map` that receives the `jobs` and the specific index `i` of the job to be mapped. The "reduce" subtasks are implemented by an external service `Reduce` that receives the `jobs`, the specific index `i` of the job under execution, and the specific index `j` of the "reduce" subtask to be executed — for every `i`, there are `r` such subtasks. In this case, the expected latency of the entire function is given by the sum of `m` times the latency of the service `Map` and of $\texttt{m} \times \texttt{r}$ times the latency of the service `Reduce`. Given that such latency could be high, a user could be interested to run the function on a worker, only if the expected overall latency is below a given threshold.

3 The inference of cost expressions

In this section, we formalise how one can extract a cost program from miniSL code. Once extracted, we can feed this program to off-the-shelf tools, such as [4, 1], to calculate the cost expression of the related miniSL code.

Cost programs are lists of *equations* which are terms

$$f(\overline{x}) \;=\; \mathfrak{e} + \sum_{i\in 0..n} f_i(\overline{\mathfrak{e}_i}) \qquad\qquad [\,\varphi\,]$$

where variables occurring in the right-hand side and in φ are a subset of $\overline{x}$ and f and f_i are (cost) function symbols. Every function definition has a right-hand side consisting of

- a *Presburger arithmetic expression* $\mathfrak{e}$ whose syntax is

$$\mathfrak{e} \;::=\; \quad x \;\mid\; q \;\mid\; \mathfrak{e}+\mathfrak{e} \;\mid\; \mathfrak{e}-\mathfrak{e} \;\mid\; q*\mathfrak{e} \;\mid\; max(\mathfrak{e}_1,\cdots,\mathfrak{e}_k)$$

  where x is a variable and q is a positive rational number,
- a number of *cost function invocations* $f_i(\overline{\mathfrak{e}_i})$ where $\overline{\mathfrak{e}_i}$ are Presburger arithmetic expressions,
- the *Presburger guard* φ is a *linear conjunctive constraint*, *i.e.*, a conjunction of *constraints* of the form $\mathfrak{e}_1 \geq \mathfrak{e}_2$ or $\mathfrak{e}_1 = \mathfrak{e}_2$, where both $\mathfrak{e}_1$ and $\mathfrak{e}_2$ are Presburger arithmetic expressions.

The intended meaning of an equation $f(\overline{x}) \;=\; \mathfrak{e} + \sum_{i\in 0..n} f_i(\overline{\mathfrak{e}_i}) \;\; [\,\varphi\,]$ is that the cost of f is given by $\mathfrak{e}$ and the costs of $f_i(\overline{\mathfrak{e}_i})$, when the guard φ is true. Intuitively, $\mathfrak{e}$ quantifies the specific cost of one execution of f without taking into account invocations of either auxiliary functions or recursive calls. Such additional cost is quantified by $\sum_{i\in 0..n} f_i(\overline{\mathfrak{e}_i})$. The *solution of a cost program* is an expression, quantifying the cost of the function symbol in the first equation in the list, which is parametric in the formal parameters of the function symbol.

For example, the following cost program

$$\begin{array}{lcll} f(N,M) & = & M + f(N-1,M) & [N \geq 1] \\ f(N,M) & = & 0 & [N = 0] \end{array}$$

defines a function f that is invoked $N+1$ times and each invocation, excluding the last having cost 0, costs M. The solution of this cost program is the *cost expression* $N \times M$.

Our technique associates cost programs to miniSL functions by parsing the corresponding codes. In particular, we define a set of (inference) rules that gather fragments of cost programs that are then combined in a syntax-directed manner. As usual with syntax-directed rules, we use *environments* Γ, Γ', which are maps. In particular,

- Γ takes a service $\mathtt{h}$ or a parameter name p and returns a Presburger arithmetics expression, which is usually a variable. For example, if $\Gamma(\mathtt{h}) = X$, then X will appear in the cost expressions of miniSL functions using $\mathtt{h}$ and will represent the cost for accessing the service. As regards parameter names p, $\Gamma(p)$ represents values which are known at function scheduling time,
- Γ takes counters i and returns the type `Int`.

When we write $\Gamma + i : \mathtt{Int}$, we assume that i does not belong to the domain of Γ. Let C be a sum of cost of function invocations and let Q be a list of equations. Judgments have the shape

- $\Gamma \vdash \mathsf{E} : \mathfrak{e}$, meaning that the value of the *integer expression* E in Γ is represented by (the Presburger arithmetic expression) $\mathfrak{e}$,
- $\Gamma \vdash \mathsf{E} : \varphi$, meaning that the value of the *boolean expression* E in Γ is represented by (the Presburger guard) φ,
- $\Gamma \vdash \mathsf{S} : \mathfrak{e} \;;\; \mathsf{C} \;;\; \mathsf{Q}$, meaning that the cost of S in the environment Γ is $\mathfrak{e} + \mathsf{C}$ given a list Q of equations,

- $\Gamma \vdash \mathsf{F} : \mathsf{Q}$, meaning that the cost of a function F in the environment Γ is the list Q of equations.

We use the notation $var(\mathbb{e})$ to address the set of variables occurring in $\mathbb{e}$, which is extended to tuples $var(\mathbb{e}_1, \cdots, \mathbb{e}_n)$ with the standard meaning. Similarly $var(\sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}))$ is the union of the sets of variables $var(\overline{\mathbb{e}_0}), \cdots, var(\overline{\mathbb{e}_n})$.

The inference rules for miniSL are reported in Figure 2. They compute the cost of a program with respect to the calls to external services (whose cost is recorded in the environment Γ). Therefore, if a miniSL expression (or statement) has no service invocation, its cost is 0. Notice that in the rule [IF-EXP] we use the guard $[\,\neg\varphi\,]$, to model the negation of a linear conjunctive constraint φ, even if negation is not permitted in Presburger arithmetic. Actually, such notation is syntactic sugar defined as follows:

- let $\neg\varphi$ (the *negation* of a Presburger guard φ) be the *list* of Presburger guards

$$\begin{array}{rcl} \neg(\mathbb{e} \geq \mathbb{e}') & = & \mathbb{e}' \geq \mathbb{e} + 1 \\ \neg(\mathbb{e} = \mathbb{e}') & = & \mathbb{e} \geq \mathbb{e}' + 1 \;;\; \mathbb{e}' \geq \mathbb{e} + 1 \\ \neg(\mathbb{e} \wedge \mathbb{e}') & = & \neg\mathbb{e} \;;\; \neg\mathbb{e}' \end{array}$$

  where ; is the list concatenation operator (the list represents a *disjunction of Presburger guards*),

- let $\neg\varphi = \varphi_1 \;;\; \cdots \;;\; \varphi_m$, where φ_i are Presburger guards, then

$$\left(f(\overline{x}) = \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i})\right)[\,\neg\varphi\,] \quad \stackrel{\text{def}}{=} \quad \left\{ f(\overline{x}) = \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \quad [\,\varphi_j\,] \quad | \quad j \in 1..m \right\}.$$

We now comment on the inference rules reported in Figure 2.[1]

Rule [CALL] manages invocation of services: the cost of `call` $\mathtt{h}(E)$ S is the cost of S plus the cost for accessing the service `h`.

Rule [IF-EXP] defines the cost of conditionals when the guard is a Presburger arithmetic expression that can be evaluated at function scheduling time. We use a corresponding cost function, if_ℓ, whose name is fresh,[2] to indicate that the cost of the entire conditional statement is either the cost of the then-branch or the else-branch, depending on whether the guard is true or false. As discussed above, the use of the guard $\neg\varphi$ generates a list of equations.

Rule [IF-CALL] defines an upper bound of the cost of conditionals when the guard is an invocation to a service. At scheduling time it is not possible to determine whether the guard is true or false – *c.f.* the second example in Section 2. Therefore the cost of a conditional is the maximum between the cost $\mathbb{e}' + \mathsf{C}$ of the then-branch and the one $\mathbb{e}'' + \mathsf{C}'$ of the else-branch, plus the cost $\mathbb{e}$ to access to the service in the guard. However, considering that the expression $max(\mathbb{e} + \mathsf{C}, \mathbb{e}' + \mathsf{C}')$ is not a valid right-hand side for the equations in our cost programs, we take as over-approximation the expression $max(\mathbb{e}, \mathbb{e}') + \mathsf{C} + \mathsf{C}'$.

As regards iterations, according to [FOR], its cost is the invocation of the corresponding function, for_ℓ, whose name is fresh (we assume that iterations have pairwise different line-codes). The rule adds the counter i to Γ (please recall that $\Gamma + i : \mathtt{Int}$ entails that $i \notin dom(\Gamma)$). In particular, the counter i is the first formal parameter of for_ℓ; the other parameters are all the variables in $\mathbb{e}$, in notation $var(\mathbb{e})$ plus those in the invocations C (minus the i). There are two equations for every iteration: one is the case when i is out-of-range, hence the cost is 0, the other is when it is in range and the cost is the one of the body *plus* the cost of the recursive invocation of for_ℓ with i increased by 1.

The cost of a miniSL program is defined by [PRG]. This rule defines an equation for the function *main* and puts this equation as the first one in the list of equations.

[1] We omit rules for expressions E since they are straightforward: they simply return E if E is in Presburger arithmetics.

[2] We assume that conditionals have pairwise different line-codes and ℓ represents the line-code of the if in the source code.

$$\text{[EPS]}\quad \Gamma \vdash \varepsilon : 0 \;;\; \emptyset \;;\; \emptyset$$

$$\text{[CALL]}\quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q}}{\Gamma \vdash \mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}})\ \ \mathsf{S} : \mathbb{e} + \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q}}$$

$$\text{[IF-EXP]}\quad \frac{\Gamma \vdash \mathsf{E} : \varphi \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \;;\; \mathsf{C}' \;;\; \mathsf{Q}' \qquad \mathit{if}_\ell \ \mathit{fresh} \qquad \overline{w} = \mathit{var}(\mathbb{e}, \mathbb{e}', \mathbb{e}'') \cup \mathit{var}(\mathsf{C}, \mathsf{C}') \qquad \mathsf{Q}'' = \begin{bmatrix} \mathit{if}_\ell(\overline{w}) = \mathbb{e}' + \mathsf{C} & [\,\varphi\,] \\ \mathit{if}_\ell(\overline{w}) = \mathbb{e}'' + \mathsf{C}' & [\,\neg\varphi\,] \end{bmatrix}}{\Gamma \vdash \mathtt{if}\ (\mathsf{E})\ \{\,\mathsf{S}\,\}\ \mathtt{else}\ \{\,\mathsf{S}'\,\} : 0 \;;\; \mathit{if}_\ell(\overline{w}) \;;\; \mathsf{Q}, \mathsf{Q}', \mathsf{Q}''}$$

$$\text{[IF-CALL]}\quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \;;\; \mathsf{C}' \;;\; \mathsf{Q}'}{\Gamma \vdash \mathtt{if}\ (\mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}}))\ \{\,\mathsf{S}\,\}\ \mathtt{else}\ \{\,\mathsf{S}'\,\} : \mathbb{e} + \mathit{max}(\mathbb{e}', \mathbb{e}'') \;;\; \mathsf{C} + \mathsf{C}' \;;\; \mathsf{Q}, \mathsf{Q}'}$$

$$\text{[FOR]}\quad \frac{\Gamma \vdash \mathsf{E} : \mathbb{e} \qquad \Gamma + i : \mathtt{Int} \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \overline{w} = (\mathit{var}(\mathbb{e}, \mathbb{e}') \cup \mathit{var}(\mathsf{C})) \setminus i \qquad \mathit{for}_\ell \ \mathit{fresh} \qquad \mathsf{Q}' = \begin{bmatrix} \mathit{for}_\ell(i, \overline{w}) = \mathbb{e}' + \mathsf{C} + \mathit{for}_\ell(i+1, \overline{w}) & [\,\mathbb{e} \geq i\,] \\ \mathit{for}_\ell(i, \overline{w}) = 0 & [\,i \geq \mathbb{e} + 1\,] \end{bmatrix}}{\Gamma \vdash \mathtt{for}\ (i\ \mathtt{in}\ \mathtt{range}(\mathtt{0}, \mathsf{E}))\{\,\mathsf{S}\,\} : 0 \;;\; \mathit{for}_\ell(0, \overline{w}) \;;\; \mathsf{Q}, \mathsf{Q}'}$$

$$\text{[PRG]}\quad \frac{\Gamma \vdash \mathsf{S} : \mathbb{e} \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \overline{w} = \mathit{var}(\overline{p}, \mathbb{e}) \cup \mathit{var}(\mathsf{C}) \qquad \mathit{main}\ \mathit{fresh} \qquad \mathsf{Q}' = \mathit{main}(\overline{w}) = \mathbb{e} + \mathsf{C} \quad [\,]}{\Gamma \vdash (\overline{p})\ \texttt{=>}\ \{\,\mathsf{S}\,\} : \mathsf{Q}', \mathsf{Q}}$$

Figure 2: The rules for deriving cost expressions

As an example, in the following, we apply the rules of Figure 2 to the codes in Listings 1, 2 and 3. Let $\Gamma(\mathtt{isPremiumUser}) = u$, $\Gamma(\mathtt{PremiumService}) = P$ and $\Gamma(\mathtt{BasicService}) = B$. For Listing 1 we obtain the cost program

$$\begin{array}{rll} \mathit{main}(u,P,B) = & \mathit{if}_2(u,P,B) & [\,] \\ \mathit{if}_2(u,P,B) = & P & [\,u = 1\,] \\ \mathit{if}_2(u,P,B) = & B & [\,u = 0\,] \end{array}$$

For Listing 2, let $\Gamma(\mathtt{IsPremiumUser}) = K$. Then the rules of Figure 2 return the single equation

$$\mathit{main}(K,P,B) = \quad K + \mathit{max}(P,B) \qquad [\,]$$

For 3, when $\Gamma(\mathtt{m}) = m$, $\Gamma(\mathtt{r}) = r$, $\Gamma(\mathtt{Map}) = M$ and $\Gamma(\mathtt{Reduce}) = R$, the cost program is

$$\begin{array}{rll} \mathit{main}(m,r,M,R) = & \mathit{for}_2(0,m,r,M,R) & [\,] \\ \mathit{for}_2(i,m,r,M,R) = & M + \mathit{for}_4(0,r,R) + \mathit{for}_2(i+1,m,r,M,R) & [\,m \geq i\,] \\ \mathit{for}_2(i,m,r,M,R) = & 0 & [\,i \geq m+1\,] \\ \mathit{for}_4(j,r,R) = & R + \mathit{for}_4(j+1,r,R) & [\,r \geq j\,] \\ \mathit{for}_4(j,r,R) = & 0 & [\,j \geq r+1\,] \end{array}$$

The foregoing cost programs can be fed to automatic solvers such as Pubs [1] and CoFloCo [4]. The evaluation of the cost program for Listing 1 returns $\mathit{max}(P,B)$ because u is unknown. On the contrary, if u is known, it is possible to obtain a more precise evaluation from the solver: if $u = 1$ it is possible to

ask the solver to consider $main(1,P,B)$ and the solution will be P, while if $u = 0$ it is possible to ask the solver to consider $main(0,P,B)$ and the solution will be B. The evaluation of $main(K,P,B)$ for Listing 2 gives the expression $K + max(P,B)$, which is exactly what is written in the equation. This is reasonable because, statically, we are not aware of the value returned by the invocation of `IsPremiumService`. Last, the evaluation of the cost program for Listing 3 returns the expression $m \times (M + r \times R)$.

4 From APP to cAPP

We now discuss the extension of APP that we plan to realise, where function scheduling policies could depend on the costs associated with the possible execution of the functions on the available workers.

Before discussing the extensions towards cAPP, we briefly introduce the APP syntax and constructs, reported in Figure 3, as found in its first incarnation by De Palma et al. [3]

The APP Language

An APP script is a collection of tagged scheduling policies. The main, mandatory component of any policy (identified by a *policy_tag*) are the `workers` therein, i.e., a collection of labels that identify on which workers the scheduler can allocate the function. The assumption is that the environment running APP establishes a 1-to-1 association so that each worker has a unique, identifying label. A policy, associate to every function a list of one or more *block*s, each including the `worker` clause to state on which workers the function can be scheduled and two optional parameters: the scheduling `strategy`, followed to select one of the workers of the block, and an `invalidate` condition, which determines when a worker cannot host a function. When a selected worker is invalid, the scheduler tries to allocate the function on the rest of the available workers in the block. If none of the workers of a block is available, then the next block is tried. The last clause, `followup`, encompasses a whole policy and defines what to do when no *block*s of the policy managed to allocate the function. When set to `fail`, the scheduling of the function fails; when set to `default`, the scheduling continues by following the (special) `default` policy.

As far as the `strategy` is concerned, it allows the following values: `platform` that applies the default selection strategy of the serverless platform; `random` that allocates functions stochastically among the workers of the block following a uniform distribution; `best-first` that allocates functions on workers based on their top-down order of appearance in the block. The options for the `invalidate` are instead: `overload` that invalidates a worker based on the default invalidation control of the platform; `capacity_used` that invalidates a worker if it uses more than a given percentage threshold of memory; `max_concurrent_invocations` that invalidates a worker if a given number of function invocations are already currently executed on the worker.

Towards cAPP

Our proposal to extend APP to handle cost-aware scheduling policies entails two major modifications: (*i*) extending the APP language to express cost-aware scheduling policies, (*ii*) implementing a new controller that selects the correct worker following the cost-aware policies.

As far as (i) is concerned, we discuss at least two relevant ways in which costs can be used. The first one is a new selection strategy named `min_latency`. Such a strategy selects, among some available workers, the one which minimises a given cost expression. The second one is a new invalidation condition named `max_latency`. Such a condition invalidates a worker in case the corresponding cost expression is greater than a given threshold.

$$
\begin{array}{lcl}
\textit{policy\_tag} & \in & \textit{Identifiers} \cup \{\texttt{default}\} \qquad \textit{worker\_label} \in \textit{Identifiers} \qquad n \in \mathbb{N} \\
\textit{app} & ::= & \overline{\textit{tag}} \\
\textit{tag} & ::= & \textit{policy\_tag} : \overline{\texttt{-}\ \textit{block}}\ \textit{followup}? \\
\textit{block} & ::= & \texttt{workers:}\ [\ \texttt{*} \mid \overline{\texttt{-}\ \textit{worker\_label}}\ ] \\
 & & (\texttt{strategy:}\ [\ \texttt{random} \mid \texttt{platform} \mid \texttt{best\_first}\ ])? \\
 & & (\texttt{invalidate:}\ [\ \texttt{capacity\_used} : n\% \\
 & & \qquad \mid \texttt{max\_concurrent\_invocations:}\ n \\
 & & \qquad \mid \texttt{overload}\ ])? \\
\textit{followup} & ::= & \texttt{followup:}\ [\ \texttt{default} \mid \texttt{fail}\ ]
\end{array}
$$

Figure 3: The APP syntax.

We dub cAPP the cost-aware extension of APP and illustrate its main features by showing examples of cAPP scripts that target the functions in Listings 1–3.

```
- premUser:
 - workers:
     - wrk: W1
     - wrk: W2
   strategy: min_latency
```

Listing 4: cAPP script for Listings 1 and 2.

```
- mapReduce :
  - workers:
    - wrk: W1
    - wrk: W2
  strategy: random
  invalidate:
    max_latency: 300
```

Listing 5: cAPP script for Listing 3.

In Listing 4, we define a cAPP script where we assume to associate the tag premUser to both the functions at Listing 1 and 2. In the script, we specify to follow the logic min_latency to select among the two workers, W1 and W2 listed in the workers clause, and prioritises the one for which the solution of the cost expression is minimal.

To better illustrate the phases of the min_latency strategy, we depict in Figure 4 the flow, from the deployment of the cAPP script to the scheduling of the functions in Listings 1 and 2. When the cAPP script is created, the association between the functions code and their cAPP script is specified by tagging the two functions with //tag:premUser. In this phase, assuming the scheduling policy of the cAPP script requires the computation of the functions cost, the code of the functions is used to infer the corresponding cost programs. When the functions are invoked, i.e., at scheduling time, we can compute the solution of the cost program, given the knowledge of the invocation parameters. For instance, for the function in Listings 1, it is possible to invoke the solver with either $main(1,P,B)$ or $main(0,P,B)$ depending on the actual invocation parameter. Figure 4 illustrates this last part with the horizontal "request" lines found at the bottom. In particular, when we receive a request for the function at Listing 1, we take its cost program (represented by the intersection point on the left) and its corresponding cAPP policy to implement the expected scheduling policy. We can implement this behaviour in two steps. First, the solver solves the cost programs (depicted by the gear); then, we compute the obtained cost expression for each of the possible workers (in this case, W1 and W2) by instantiating the parameter representing the cost of invocation of the external services, with an estimation of the latencies from the considered workers. In this case, given the min_latency strategy, the worker that minimises the latency to contact PremiumService will be selected. This last step regards the second point (*ii*) mentioned at the beginning of this section, i.e., the

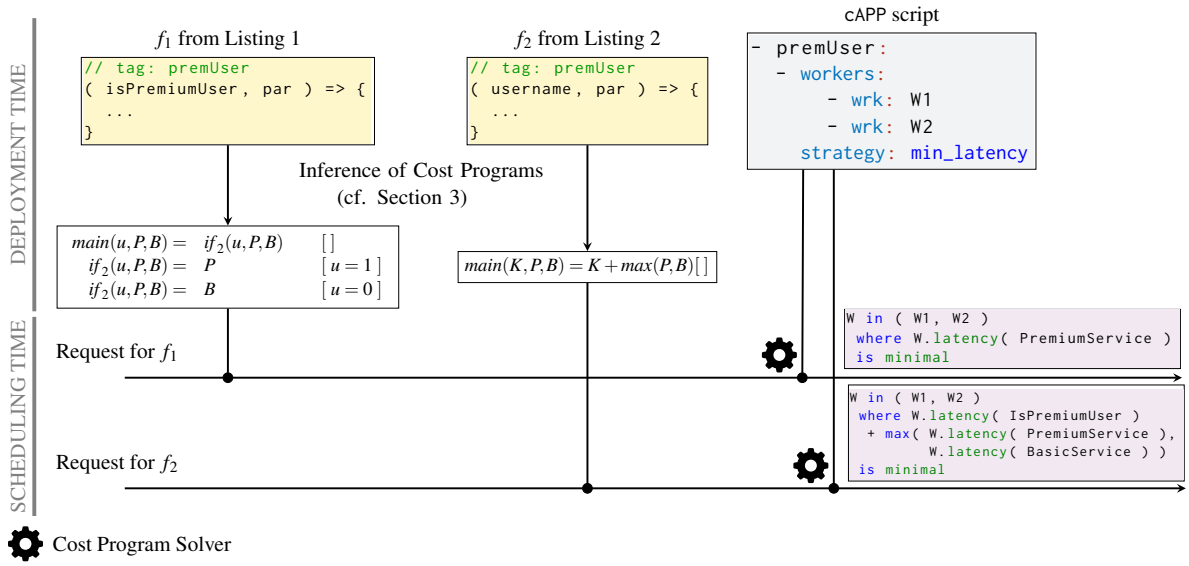


Figure 4: Flow followed, from deployment to scheduling, of the functions at Listings 1 and 2.

modifications we need to perform on the controller to let it execute the newly introduced cost-aware strategies at scheduling time.

For `max_latency`, once a worker is selected using a given strategy, its corresponding cost is computed in order to check whether the selection is invalid (i.e., if we can consider the worker able to execute the function, given the invalidation constraints of the script). To illustrate this second occurrence, we look at the cAPP code we wrote for the map-reduce function in Listing 5, and we illustrate it using Figure 5. As seen above, we start (top-most box) from the deployment phase, where we tag the function (`//tag: mapReduce`) and we proceed to compute its cost program, obtaining the associated cost expression. Then, when we receive a request for that function, we trigger the execution of the cAPP policy, which selects one of the two workers `W1` or `W2` at `random` and checks their validity following the logic shown at the bottom of Figure 5, i.e., we solve the cost program and then compute the corresponding cost expression by replacing the parameters `m` and `r` with the `latency` to contact the `Map` and `Reduce` services from the selected worker, and possibly invalidate it if the computed value is greater than `300`. In the function's code, for simplicity, we abstract away the coordination logic between `Map` and `Reduce` (which usually performs a multipoint scatter-gather behaviour) by offloading it to external services (e.g., a database contacted by the functions).

These new `strategy` and `invalidate` parameters added for cAPP interact with the cost-inference logic presented in Section 3. As shown in Figure 4, the definition of the `strategy` and `invalidate` parameters, as well as the cost inference, happen independently, when the cAPP script is deployed. A strategy indeed (e.g., `min_latency`) is not tied to any specific cost expression. For example, the user can define the `premUser` policy (see the cAPP script on the right-hand side of Figure 4) before having deployed any function with that tag. When functions are deployed on the platform (centre and left-hand side of Figure 4), the cAPP runtime performs the inference of programs' costs. When instead a request for the execution of a function reaches the platform, the cAPP use the cost expressions and create the logic of selection/invalidation down to its runtime form. For instance, in Figure 4, the scheduling of function f_1 compiles the `min_latency` logic using the reduced form P (the cost of accessing service `PremiumService`) since at scheduling time the parameter `isPremiumUser` (represented by the variable u

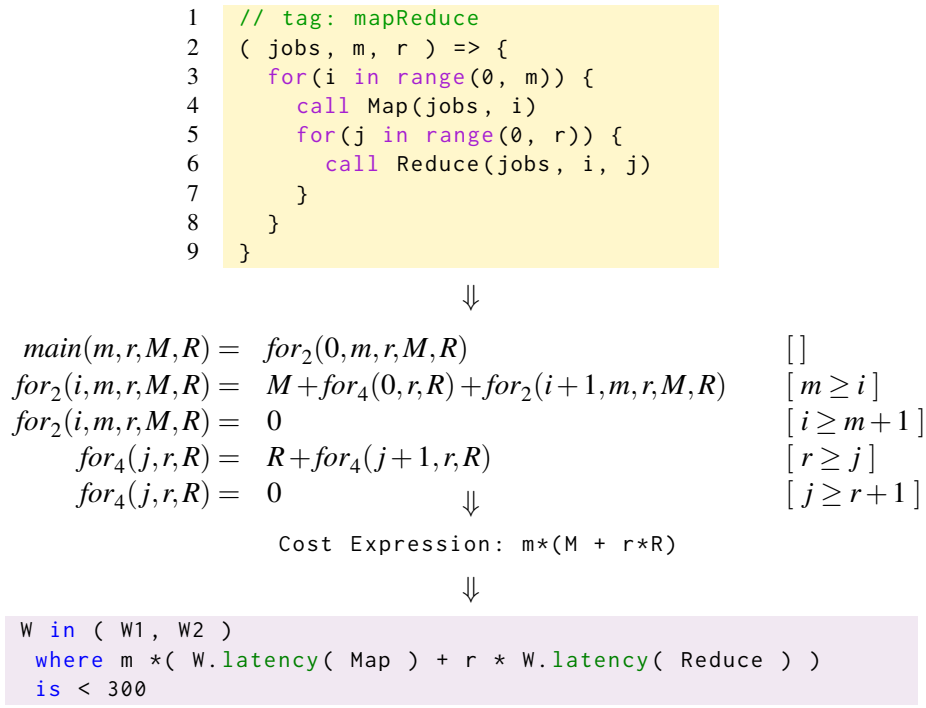


Figure 5: The map-reduce function, its cost analysis, and scheduling invalidation logic.

in the related cost equations in Figure 4) is known, which in the example we value to 1 (i.e., the request is from a premium user). From the reduced cost expression we can obtain the cAPP selection logic on the right-hand side of Figure 4: select that worker, among the one provided in the cAPP block, that minimises (`is minimal`) the `latency` of interaction with the `PremiumService` service.

For completeness, we can draw a parallel example for the invalidation parameter by looking at Figure 5. There, once we have a request for the map-reduce function, we take the cost expression calculated at deployment time, whose (integer) values represented by `m` are `r` are known at scheduling time, and we compile the `invalidate` logic, `max_latency:300` — for the map-reduce function, the logic declares invalid any worker whose cost `m *( W.latency( Map ) + r * W.latency( Reduce ) )` exceeds the set `300` threshold.

5 Conclusion

We have presented a proposal for an extension of the APP language, called cAPP, to make function scheduling cost-aware. Concretely, the extension adds new syntactic fragments to APP so that programmers can govern the scheduling of functions towards those execution nodes that minimise their calculated latency (e.g., increasing serverless function performance) and avoids running functions on nodes whose execution time would exceed a maximal response time defined by the user (e.g., enforcing quality-of-service constraints). The main technical insights behind the extension include the usage of inference rules to extract cost equations from the source code of the deployed functions and exploiting dedicated solvers to compute the cost of executing a function, given its code and input parameters.

Growing our proposal into a usable APP extension is manyfold. The cost inference of Section 2

programs is under active development at the time of writing.[3] While the solution of the cost equations can be done by off-the-shelf tools (e.g.,CoFloCo [4]), another important component to develop is the cAPP runtime to generate cAPP rules from the cost equations when functions are scheduled and interact with the workers available in the platform to collect the measures that characterise the costs sustained by the workers (e.g., the latency endured by a worker when contacting a given service).

Implementing the cAPP runtime and proving the feasibility of cost-aware function scheduling is only the first move along the way. Indeed, in Section 4 (illustrated in Figure 4) we described a naïve approach where we solve the cost equations of an invoked function at scheduling time, but this computation step could delay the scheduling of the function. This challenge calls for further investigation. On the one hand, we shall investigate if the problem presents itself in practice, i.e., if developers would actually write functions whose cost equations take too much time for the available engines to solve. On the other hand, we envision working on models and techniques that can make the problem treatable (e.g., via heuristics and over-approximations), possibly complementing the former with architectural solutions, like the inclusion of caching systems that allows us to compute the actual cost of function invocations once and timeouts paired with sensible default strategies which would keep the system responsive.

Acknowledgement

Research partly supported by the H2020-MSCA-RISE project ID 778233 “Behavioural Application Program Interfaces (BEHAPI)” and by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim & Germán Puebla (2008): *Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis*. In María Alpuente & Germán Vidal, editors: *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, *Lecture Notes in Computer Science* 5079, Springer, pp. 221–237, doi:10.1007/978-3-540-69166-2_15. Available at https://doi.org/10.1007/978-3-540-69166-2_15.

[2] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin & Gianluigi Zavattaro (2022): *A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling*. In Claudio Agostino Ardagna, Nimanthi L. Atukorala, Boualem Benatallah, Athman Bouguettaya, Fabio Casati, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Chirine Ghedira Guegan, Robert Ward, Fatos Xhafa, Xiaofei Xu & Jia Zhang, editors: *IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022*, IEEE, pp. 337–342, doi:10.1109/ICWS55610.2022.00056.

[3] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro & Gianluigi Zavattaro (2020): *Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation*. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya & Hamid Motahari, editors: *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, *Lecture Notes in Computer Science* 12571, Springer, pp. 416–430, doi:10.1007/978-3-030-65310-1_29.

[4] Antonio Flores-Montoya & Reiner Hähnle (2014): *Resource Analysis of Complex Programs with Cost Equations*. In Jacques Garrigue, editor: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, *Lecture Notes in Computer Science* 8858, Springer, pp. 275–295, doi:10.1007/978-3-319-12736-1_15. Available at https://doi.org/10.1007/978-3-319-12736-1_15.

[3] https://github.com/minosse99/CostCompiler

[5] Abel Garcia, Cosimo Laneve & Michael Lienhardt (2017): *Static analysis of cloud elasticity*. *Sci. Comput. Program.* 147, pp. 27–53, doi:10.1016/j.scico.2017.03.008. Available at https://doi.org/10.1016/j.scico.2017.03.008.

[6] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau & Remzi H. Arpaci-Dusseau (2016): *Serverless Computation with OpenLambda*. 41. Available at https://www.usenix.org/publications/login/winter2016/hendrickson.

[7] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica & David A. Patterson (2019): *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.

[8] Cosimo Laneve & Claudio Sacerdoti Coen (2021): *Analysis of smart contracts balances*. *Blockchain: Research and Applications* 2(3), p. 100020, doi:https://doi.org/10.1016/j.bcra.2021.100020. Available at https://www.sciencedirect.com/science/article/pii/S2096720921000154.