Reachability Analysis of Function-as-a-Service Scheduling Policies

Giuseppe De Palma[1,2], Saverio Giallorenzo[1,2], Jacopo Mauro[3], Matteo Trentin[1,2,3], and Gianluigi Zavattaro[1,2]

[1] Università di Bologna, Italy{giuseppe.depalma2,saverio.giallorenzo2, matteo.trentin2,gianluigi.zavattaro}@unibo.it
[2] OLAS research team, INRIA, France
[3] University of Southern Denmark, Denmark mauro@imada.sdu.dk

Abstract. Functions-as-a-Service (FaaS) is a Serverless Cloud paradigm where a platform manages the execution (e.g., scheduling, runtime environments) of stateless functions. Recently, domain-specific languages like APP and aAPP have been developed to express per-function scheduling policies, e.g., enforcing the allocation of functions on nodes that enjoy low data-access latencies thanks to proximity and connection pooling.
Reachability analysis of FaaS scheduling policies is fundamental to check quality-of-service properties, like preventing the scheduling of functions on workers which cannot sustain expected performance levels, or verifying security properties, e.g., that safety-critical functions cannot run on nodes with untrusted ones. We investigate the complexity of reachability analysis for APP and aAPP– the latter extends APP with constraints for placing functions according to the absence/presence of other (anti-)affine functions on workers. We show that reachability analysis has linear time complexity in APP, while the addition of affinities (in aAPP) makes reachability PSPACE. Given the computational complexity correspondence between reachability analysis in aAPP and automatic planning problems (both are in PSPACE), we investigate the exploitability of planners, i.e., tools specialised for solving planning problems for the realisation of static analysers of reachability-like problems for aAPP.

1 Introduction

Functions-as-a-Service (FaaS) is a programming paradigm supported by the Serverless Cloud execution model [30]. In FaaS, developers implement a distributed architecture by composing stateless functions and delegate concerns like execution runtimes and resource allocation to the serverless platform, thus focusing on writing code that implements business logic rather than worrying about infrastructure management. The main cloud providers offer FaaS [5,25,38] and open-source alternatives exist too [6,42,27,22].

A common denominator of these platforms is that they manage the allocation of functions over the available computing resources, also called *workers*, following opinionated policies that favour some performance principle. Indeed, effects like *code locality* [27] – due to latencies in loading function code and runtimes – or *session* *locality* [27] – due to the need to authenticate and open new sessions to interact with other services – can substantially increase the run time of functions. The breadth of the design space of serverless scheduling policies is witnessed by the growing literature focused on techniques that mix one or more of these locality principles to increase the performance of function execution, assuming some locality-bound traits of functions [45,41,35,1,47,55,40,31,53,50,51,54,48,12,32,9,10,29,33,52]. Besides performance, functions can have functional requirements that the scheduler shall consider. For example, users might want to ward off allocating their functions alongside "untrusted" ones – common threat vectors in serverless are limited function isolation and the ability of functions to (surreptitiously) gather weaponisable information on the runtime, the infrastructure, and the other tenants [8,57,4,15].

Although one can mix different principles to expand the profile coverage of a given platform-wide scheduler policy, the latter hardly suits all kinds of scenarios. This shortcoming motivated the introduction of a domain-specific, platform-agnostic, declarative language for expressing *Allocation Priority Policies* (which inspired the name of the language APP), for specifying custom function allocation policies [21,17]. Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions. APP has been validated by implementing a serverless platform as an extension of the open-source Apache OpenWhisk project – the APP-based variant outperforms vanilla OpenWhisk in several locality-bound scenarios [21,17].

Recently, De Palma et al. [20] presented an extension of APP for expressing *affinity-aware* policies for serverless function scheduling (the addition of affinity-awareness to APP inspired the name aAPP for this language extension). Specifically, aAPP extends APP with the possibility to impose that one function can be placed on a worker only if another function is present (affinity) or absent (anti-affinity) on that worker. These constraints are useful to avoid the co-location of functions (e.g., trusted functions cannot be placed on a worker which is executing untrusted ones) or to guarantee their co-location (e.g., a function accessing a database can be placed on a worker only if it hosts a function able to open a connection with that database).

Contributions. APP and its extension aAPP have been validated experimentally [21,17,20] via FaaS benchmarks and case studies. However, there is no formal study nor comparison between the two. We cover this gap by presenting the aAPP-calculus, in Sec. 2, which is an abstract formal model of function scheduling in FaaS systems where functions are scheduled following aAPP scripts. Since aAPP is an extension of APP, we identify a fragment of the aAPP-calculus corresponding to APP, that we name APP-calculus. We use our formal model to investigate whether aAPP is *more expressive* than APP by showing that static verification of properties of policies, relevant in the context of function scheduling, is *more complex* in aAPP than in APP. Namely, we formally define, in Sec. 5, the properties of *reachability*, i.e., checking if a function f can be deployed on a targeted worker, and of *co-occurrence*, i.e., checking if two functions f and g can be co-located, so they can run simultaneously on a targeted worker.

In Sec. 4, we start by investigating reachability and co-occurrence in the APP-calculus, and we prove that both problems have linear-time complexity. Then, in Sec. 5, we show that the problems become PSPACE-complete when we move to the aAPP-calculus – for presentation, here we discuss proof strategy insights while detailed proofs can be found in the companion paper [19]. These results formally witnesses the increase in expressiveness from APP to aAPP, given that the latter can specify scheduling policies much harder to formally verify. In Sec. 6, we further solidify the above results by showing that the jump in complexity comes from affinity constraints. Indeed, we show that in aAPP *without affinity* reachability and co-occurrence preserve a linear complexity while in aAPP *without anti-affinity* the problems become NP-hard.

The proof of PSPACE-completeness of the reachability problems in the aAPP-calculus formalises a complexity correspondence with automated planning problems [24]. In the light of the rich literature on automated planning and the availability of *planners* (tools for solving such problems), in Sec. 7, we introduce a static analyser for aAPP scripts which leverages planners. In particular, our analyser translates both the semantics of aAPP (scripts) and the reachability problems of interest into PDDL [23], the de-facto standard input language for planners. In this way, the analyser can use any planner capable of solving the problems we express in *PDDL with numeric fluents* – the latter are necessary for modelling function occurrences on workers and their resource usage.

We position our contributions in Sec. 8 and discuss future steps in Sec. 9.

2 Preliminaries: the APP Language and the aAPP Extension

We present APP [21,17] and its affinity-aware extension aAPP [20].

Since aAPP is a syntactic extension of APP, we directly report in Fig. 1 the syntax of aAPP, which APP shares, except for the `affinity` clause in the syntax of *block*s. From here on, we indicate syntactic units in *italics*, optional fragments in *grey*, terminals in `monospace`, and lists with $\overline{bars}$. The syntax draws inspiration from YAML [58], a renowned data-serialisation language for configuration files—e.g., many modern tools use the format, like Kubernetes, Ansible, and Docker.[4] The idea is that an APP/aAPP script defines a scheduling policy where functions have associated a tag that identifies their scheduling information. Essentially, a *tag* corresponds to a list of *block*s that indicate either some collection of workers, each identified by a worker *id*, or the universal `*`. The scheduling of a function works by retrieving its tag and then its scheduling information, which includes one or more blocks of possible workers. Worker selection happens by iteratively checking the blocks from top to bottom, picking the first block with a non-empty list of valid workers and selecting one worker of that list according to the strategy defined by the block (described later).

In case all the blocks associated with a tag fail, the `followup` clause specifies what to do next: either `fail`, to terminate the attempt to schedule the function, or `default`, to try scheduling the function as if associated with the special `default` tag. More precisely, we consider `default` as a special tag name with which we can associate generic scheduling policies which are independent of the function peculiarities; if the specific function scheduling policies fail, the `followup` clause set to `default` indicates that the generic `default` policy can be used.

[4] While aAPP scripts are YAML-compliant, for presentation, we slightly stylise the syntax to increase readability. For instance, we omit quotes around strings, e.g., `*` instead of `"*"`.

$$
\begin{array}{lll}
 & \multicolumn{2}{c}{\mathit{id} \in \mathit{Identifiers} \qquad n \in \mathbb{N}} \\
\mathit{app} & ::= & \overline{-\mathit{tag}} \\
\mathit{tag} & ::= & \mathit{id} : \overline{-\ \mathit{block}} \quad \boxed{\texttt{followup} : \mathit{f\_opt}} \\
\mathit{block} & ::= & \texttt{workers} : \mathit{w\_opt} \quad \boxed{\texttt{strategy} : \mathit{s\_opt}} \\
 & & \boxed{\texttt{invalidate} : \overline{-\ \mathit{i\_opt}}} \quad \boxed{\texttt{affinity} : \overline{-\ \mathit{a\_opt}}} \\
\mathit{w\_opt} & ::= & \texttt{*} \mid \overline{-\ \mathit{id}} \\
\mathit{s\_opt} & ::= & \texttt{any} \mid \texttt{best\_first} \\
\mathit{i\_opt} & ::= & \texttt{capacity\_used}\ n\% \mid \texttt{max\_concurrent\_invocations}\ n \\
\mathit{a\_opt} & ::= & \mathit{id} \mid !\mathit{id} \\
\mathit{f\_opt} & ::= & \texttt{default} \mid \texttt{fail}
\end{array}
$$

Fig. 1. aAPP syntax (APP omits the `affinity` clause).

```
- f_tag:
  - workers:
      - local_w1
      - local_w2
    strategy: best_first
    invalidate:
      - capacity_used 80%
    affinity: g_tag ,!h_tag
  - workers:
      - public_w1
  followup: fail
```

Fig. 2. Example aAPP script.

Each block can define a `strategy` for worker selection (`any` selects one of the available workers non-deterministically from the list; `best_first` selects the first available worker in the list), a list of constraints that `invalidate`s a worker for the allocation (`capacity_used` invalidates a worker if its resource occupation reaches the set threshold; `max_concurrent_invocations` invalidates a worker if it hosts more than the specified number of functions). The aAPP extension includes the `affinity` clause, which carries a list containing affine tag identifiers *id* and anti-affine tags, represented by negated tag identifiers *!id*. This clause introduces an invalidation condition determined by the absence (affinity) or presence (anti-affinity) of functions on workers.

As an example, Fig. 2 shows an aAPP script carrying the scheduling information for functions tagged `f_tag`, which includes two blocks. The first restricts the allocation of the function on the workers labelled `local_w1` and `local_w2` and the second on `public_w1`. The first block specifies as invalid (i.e., unable to host the function under scheduling) the workers that reach a memory consumption above 80%. Since the strategy is `best_first`, we allocate the function on the first valid worker; if none are valid, we proceed with the next block. The function has affinity with *g_tag* and anti-affinity with *h_tag*. Hence, a worker is valid if it hosts at least a function with tag *g_tag* and no functions with tag *h_tag*. If both the first and second blocks did not find a valid worker, the scheduling of the function `fail`s.

As a final remark, note that the (anti-)affinity relation in aAPP is "directional"—similarly to the one introduced by Microsoft in its IaaS offering [37]. In particular, it does not impose any properties like symmetry on affinity or anti-affinity. The reason for avoiding these additional constraints is to allow aAPP to capture as many useful scenarios as possible, which imposing properties like symmetry would prevent.[5]

3 The aAPP-calculus and its Fragment APP-calculus

We introduce the aAPP-calculus to formally model function scheduling in FaaS systems governed by aAPP scripts. We represent the states of the FaaS systems with pairs

[5] For example, if we had symmetric anti-affinity, it would not capture typical scenarios where e.g., `init` is a seeder for a database and `query` manipulates that data. We need `init` to run before `query` but not where `query` is already running, while `query` shall run where `init` is present, i.e., `init` is anti-affine with `query` but `query` is affine with `init`.

$\langle N, \Delta \rangle$ where N is a *network of workers* and Δ is a *scheduling policy*. The calculus defines labelled transitions $\langle N, \Delta \rangle \xrightarrow{\alpha} \langle N', \Delta \rangle$ that indicate that the state $\langle N, \Delta \rangle$ can change into $\langle N', \Delta \rangle$ as effect of the occurrence of the event α. The events ranged by α are either $f{\downarrow}w$, denoting the scheduling of function f on worker w, and $f{\uparrow}w$, representing the removal of function f from worker w because f terminated its execution.

The definition of the calculus is parametric w.r.t. the workers and the functions.

Definition 1 (Workers and Functions). *We assume given the set $\mathcal{W}$ (ranged over by w) of the* worker *identifiers and the set $\mathcal{F}$ (ranged over by f) of the* function *identifiers. For each $w \in \mathcal{W}$ we use* **cap**(w) *to denote the capacity of the worker w (i.e., a strictly positive number quantifying the total resources of the worker w) and for each $f \in \mathcal{F}$ we use* **res**(f) *to denote the resources needed by the function f (i.e., a strictly positive number quantifying the resources that the function f must acquire from the worker where it is scheduled).*

We also use two additional sets: the *selection strategies*, $\mathcal{S} \triangleq \{$`any`, `best_first`$\}$ (ranged over by s), and the set of the *invalidation conditions*, $\mathcal{I} \triangleq \{$`capacity_used` n%, `max_concurrent_invocations` n, `affine` F, `anti_affine` $F \mid n \in \mathbb{N}, F \subseteq \mathcal{F}\}$ (ranged over by i). Note that the selection strategies, `any` and `best_first`, as well as the invalidation conditions, `capacity_used` n%, `max_concurrent_invocations` n, precisely correspond to the syntactic elements of aAPP/APP (cf. Fig. 1). Contrarily to aAPP's syntax, we do not treat `affinity` as a separate entity and include it with the other invalidation conditions by adding `affine` F, meaning that at least one function in F should be present, and `anti_affine` F, requiring the absence of all the functions in F.

We start by defining the network of workers N, which intuitively associates to each worker the list of functions currently scheduled on it.

Definition 2 (Network of workers). *A* network of workers $N \in \mathcal{N} \triangleq \mathcal{W} \to \mathit{List}(\mathcal{F})$ *is a function that associates to a worker the list $\overline{f}$ of functions currently scheduled on that worker.*

In the following, $N \cup (w \mapsto \overline{f})$ denotes a network of workers obtained by extending the network N with the mapping from the worker w to the list of functions $\overline{f}$.[6]

Moving on to the definition of the scheduling policy Δ, it intuitively associates to each function its list of scheduling blocks, each carrying the list of workers, strategies, and invalidation conditions that determine the scheduling of the function.

Definition 3 (Scheduling blocks and policy). *A* scheduling block $b \in \mathcal{B} \triangleq (\mathit{List}(\mathcal{W}), \mathcal{S}, \mathit{List}(\mathcal{I}))$ *is a triple $(\overline{w}, s, \overline{i})$ where $\overline{w}$ is a list of workers, s is a selection strategy, and $\overline{i}$ is a list of invalidation conditions. A* scheduling policy $\Delta \in \mathcal{D} \triangleq \mathcal{F} \to \mathit{List}(\mathcal{B})$ *is a mapping that associates to each function $f \in \mathcal{F}$ the list $\overline{b}$ of blocks that define the scheduling of f on workers.*

As for networks, in the following, we denote with $\Delta \cup (f \mapsto \overline{b})$ a policy obtained by extending the policy Δ with the mapping from function f to the the list of blocks $\overline{b}$.[7]

[6] The notation assumes that w does not belong to the domain of N.
[7] The notation assumes that f does not belong to the domain of Δ.

$$\llbracket \overline{-\ tag} :: -\ \texttt{default:}\overline{-\ block} \rrbracket = \llbracket \overline{-\ tag} :: -\ \texttt{default:}\overline{-\ block}\quad \texttt{followup: fail} \rrbracket$$

$$\llbracket \overline{-\ tag} :: -\ \texttt{default:}\overline{-\ block}\quad \texttt{followup:}\ f\_opt \rrbracket = \bigcup_{t\ \in\ \overline{tag}} \left\{ [t]_{\overline{[block]}} \right\}$$

$$\llbracket id : \overline{-\ block} \rrbracket_{\overline{b}} = \llbracket id : \overline{-\ block}\quad \texttt{followup : default} \rrbracket_{\overline{b}} = \bigcup_{f\ \in\ id} \left\{ (f \mapsto \overline{[block]} :: \overline{b}) \right\}$$

$$\llbracket id : \overline{-\ block}\quad \texttt{followup : fail} \rrbracket_{\overline{b}} = \bigcup_{f\ \in\ id} \left\{ (f \mapsto \overline{[block]}) \right\}$$

$$[\texttt{workers:}\ w\_opt\quad \bullet] = \begin{cases} ([w\_opt],s,i) & \text{if } (s,i) = [\bullet] \\ ([w\_opt],\texttt{any},\varepsilon) & \text{otherwise} \end{cases} \qquad [\texttt{strategy : } s\_opt\quad \bullet] = \begin{cases} (s\_opt,i) & \text{if } (-,\overline{i}) = [\bullet] \\ (s\_opt,\varepsilon) & \text{otherwise} \end{cases}$$

$$\llbracket \texttt{invalidate : } \overline{-\ i\_opt}\quad \bullet \rrbracket = \begin{cases} (\texttt{any},\overline{i\_opt} :: \overline{i}) & \text{if } (-,\overline{i}) = [\bullet] \\ (\texttt{any},\overline{i\_opt}) & \text{otherwise} \end{cases} \qquad \llbracket \texttt{affinity:}\ \overline{a\_opt} \rrbracket = (\texttt{any}, \llbracket \overline{a\_opt} \rrbracket)$$

$$\llbracket a\_opt :: \overline{a\_opt} \rrbracket = \begin{cases} \texttt{affine}\ id :: \llbracket \overline{a\_opt} \rrbracket & \text{if } a\_opt = id \\ \texttt{anti\_affine}\ id :: \llbracket \overline{a\_opt} \rrbracket & \text{if } a\_opt = !id \end{cases} \qquad \begin{array}{l} \llbracket \varepsilon \rrbracket = \varepsilon \qquad \llbracket \overline{-\ id} \rrbracket = \overline{id} \\ [*] = w_1 :: w_2 :: \cdots :: w_n \quad \text{with } \mathcal{W} = \{w_1, w_2, \cdots, w_n\} \end{array}$$

Fig. 3. aAPP syntax encoding.

From aAPP to Δ The straightforward encoding $[\ ]$ – defined inductively in Fig. 3 – determines the relationship between an aAPP script and its formal representation within a policy Δ in the aAPP-calculus. The encoding formalises the translation from aAPP to Δ, assuming a sorting of aAPP scripts where the `default` tag is the last element – to keep the definition in Fig. 3 compact, we use $\bullet$ as a pattern-matching variable. We slightly abuse the notation so that a tag `f_tag` denotes also the set of functions tagged with `f_tag`. Moreover, we extend the notation of lists, denoting with ε the empty list and with $e :: l$ a list obtained by adding the element e in front of the list l.

For instance, $f \in$ `f_tag` means that f has tag `f_tag`. For each $f \in$ `f_tag` we simply add in Δ a mapping from f to the list of blocks associated with `f_tag` in the aAPP script. We implement the semantics of the `followup` clause – where the policy follows the logic of the `default` blocks if none of the tag's blocks led to a successful scheduling – by appending the list of blocks of the `default` tag to the tag's blocks. This concatenation happens only when the `followup` value of the tag is set to `default`.

Formally, for each block in the aAPP script, we consider a corresponding aAPP-calculus block $(\overline{w},s,\overline{i})$, where $\overline{w}$ is the list of the specified workers (all the workers, in case of `*`), s is the selection strategy, and $\overline{i}$ contains all the invalidation conditions in the aAPP script. These conditions include the sequence of affinity-based invalidations of the form `affine g_tag` and `anti_affine h_tag` resp. for each `g_tag` and `!h_tag` occurring in the `affinity` clause – `affine g_tag` indicates affinity with functions tagged `g_tag` and `anti_affine h_tag` indicates anti-affinity with functions tagged `h_tag`.

Operational semantics We are ready to define the operational semantics of aAPP-calculus. Formally, we define a labelled transition system on the set of states $States \triangleq \{\langle N,\Delta\rangle \mid N \in \mathcal{N}, \Delta \in \mathcal{D},\}$ (which contains the set of possible configurations of the modelled FaaS system represented as pairs $\langle N,\Delta\rangle$ where N is a network of workers and Δ is a scheduling policy) and the set of labels $Labels \triangleq \{f{\downarrow}w, f{\uparrow}w \mid f \in \mathcal{F}, w \in \mathcal{W}\}$ (where $f{\downarrow}w$ represents the scheduling of an instance of function f on worker w while $f{\uparrow}w$ represents the removal of f from w, occurring when the execution of f on w terminates).

$$[\mathit{start}]\ \frac{\overline{b} \rhd (w,s,\overline{i})::\overline{b'} \qquad \mathbf{valid}(f,w,\overline{i},\overline{f})}{\langle N\cup(w\mapsto\overline{f}),\ \Delta\cup(f\mapsto\overline{b})\rangle \xrightarrow{f\downarrow w} \langle N\cup(w\mapsto f::\overline{f}),\ \Delta\cup(f\mapsto\overline{b})\rangle}$$

$$[\mathit{rm}]\ \frac{f\in\overline{f}}{\langle N\cup(w\mapsto\overline{f}),\ \Delta\rangle \xrightarrow{f\uparrow w} \langle N\cup(w\mapsto\overline{f}\setminus f),\ \Delta\rangle}$$

$$[\mathit{next}]\ \frac{\langle N,\ \Delta\cup(f\mapsto\overline{b'})\rangle \xrightarrow{f\downarrow w} \langle N',\ \Delta'\rangle \qquad \overline{b} \rhd (w',s,\overline{i})::\overline{b'} \qquad \neg\mathbf{valid}(f,w',\overline{i},N(w'))}{\langle N,\ \Delta\cup(f\mapsto\overline{b})\rangle \xrightarrow{f\downarrow w} \langle N',\ \Delta\cup(f\mapsto\overline{b})\rangle}$$

$\rhd$ relation

$$\frac{(s=\texttt{any} \wedge w\in\overline{w})\vee(s=\texttt{best\_first} \wedge \overline{w}=w::\overline{w'})}{(\overline{w},s,\overline{i})::\overline{b} \rhd (w,s,\overline{i})::(\overline{w}\setminus w,s,\overline{i})::\overline{b}} \qquad \frac{\overline{b} \rhd (w,s,\overline{i})::\overline{b'}}{(\varepsilon,s',\overline{i'})::\overline{b} \rhd (w,s,\overline{i})::\overline{b'}}$$

valid predicate

$$\mathbf{valid}(f,w,i_1::\cdots::i_n,\overline{f}) \quad = \quad \mathbf{occ}(f::\overline{f})\leq\mathbf{cap}(w) \ \wedge \bigwedge_{j\in\{1,..,n\}} \mathbf{check}(f,w,i_j,\overline{f})$$

$$\mathbf{check}(f,w,i,\overline{f})=\begin{cases} \mathsf{true} & \text{if } i=\texttt{capacity\_used}\ n\% \wedge 100*\mathbf{occ}(f::\overline{f})/\mathbf{cap}(w)\ \leq n \\ \mathsf{true} & \text{if } i=\texttt{max\_concurrent\_invocations}\ n \wedge |f::\overline{f}|\leq n \\ \mathsf{true} & \text{if } i=\texttt{anti\_affine}\ F \wedge \forall g\in F.g\notin\overline{f} \\ \mathsf{true} & \text{if } i=\texttt{affine}\ F \wedge \exists g\in F.g\in\overline{f} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathbf{occ}(\overline{f})=\sum_{f\in\overline{f}}\mathbf{res}(f)$$

Fig. 4. aAPP semantics.

Definition 4 (aAPP-calculus and APP-calculus). *The aAPP-calculus models function scheduling in FaaS systems by means of a labelled transition system (States × Labels × States) defined as the minimal transition system satisfying the rules in Fig. 4. We call APP-calculus the fragment of the aAPP-calculus where the invalidation conditions* `affine` F *and* `anti_affine` F *are not used.*

Before discussing the formal definition of the semantics of the calculus, we extend the notation for lists so that: $e\in l$ means that e belongs to l ($e\notin l$ is its negation), $l\setminus e$ is a list obtained by removing the first occurrence of e from l, and $|l|$ is the length of l.

In the definition, we use the auxiliary $\rhd$ relation and **valid** predicate, specified in Fig. 4. The relation $\rhd$ is defined on lists of scheduling blocks with the aim of extracting, from a given list, one worker to be considered for scheduling. Namely, $\overline{b} \rhd (w,s,\overline{i})::\overline{b'}$ means that w is one of the workers that can be selected from the first (non-empty) scheduling block of $\overline{b}$, and s and $\overline{i}$ are the selection strategy and the invalidation conditions of the block from which w has been extracted. The list $\overline{b'}$ contains the remaining workers and scheduling blocks. The predicate $\mathbf{valid}(f,w,i_1::\cdots::i_n,\overline{f})$

checks whether a function f can be scheduled on w assuming invalidation conditions $i_1 :: \cdots :: i_n$, and that the worker w currently hosts the functions $\overline{f}$, i.e., $N(w) = \overline{f}$. In the definition of the predicate, **occ** returns the occupancy of a list of functions (using **res** to obtain the occupancy of a single function), **cap** calculates the capacity of a worker, and **check** verifies whether a condition i the scheduling of a function f on a worker w, given a list of functions $\overline{f}$ already present on w.

We are now ready to discuss the operational semantics rules. Rule [*start*] states that if the worker w (extracted from the list of scheduling blocks of f via the $\triangleright$ relation) is valid for the scheduling of the function f (validity is checked with the **valid** predicate) then the current state can evolve by adding an instance of f to the list of functions currently scheduled on w. Rule [*rm*] simply removes an instance of function f from the list of functions currently scheduled on a worker w. This rule models the removal of a function from a worker when its execution has been completed. Lastly, rule [*next*] models the case when an attempt to schedule a function f fails (due to the invalidation conditions) and it allows the remaining workers/scheduling blocks to schedule f.

We present a couple of examples of derivations which involve the rules [*start*] and [*next*] (rule [*rm*] has a rather straightforward application). We consider the function f with consumed resources $\textbf{res}(f) = 5$ and scheduling policy $\Delta = (f \mapsto b)$, where $b = (w :: w', \texttt{best\_first}, \texttt{capacity\_used}\ 50\%)$, and initial network $N = (w \mapsto \varepsilon) \cup (w' \mapsto \varepsilon)$, with workers w and w' with capacity $\textbf{cap}(w) = \textbf{cap}(w') = 10$.

First, we can schedule f on w using the rule [*start*] (for brevity, we use f to denote the list $f :: \varepsilon$ in networks).

$$[\mathit{start}]\ \frac{\begin{array}{c} b \triangleright (w, \texttt{best\_first}, \texttt{capacity\_used}\ 50\%) :: (w', \texttt{best\_first}, \texttt{capacity\_used}\ 50\%) \\ \textbf{valid}(f, w, \texttt{capacity\_used}\ 50\%, \varepsilon) \end{array}}{\langle (w \mapsto \varepsilon) \cup (w' \mapsto \varepsilon),\ (f \mapsto b) \rangle \xrightarrow{f \downarrow w} \langle (w \mapsto f) \cup (w' \mapsto \varepsilon),\ (f \mapsto b) \rangle}$$

Let N' be the reached network $(w \mapsto f) \cup (w' \mapsto \varepsilon)$. If we try to schedule f in $\langle N',\ (f \mapsto b) \rangle$, we will allocate it on w' since w cannot host f without exceeding the $\texttt{capacity\_used}\ 50\%$ condition. This is formalised by the transition below, where $s = \texttt{best\_first}$ and $i = \texttt{capacity\_used}\ 50\%$, for brevity.

$$[\mathit{next}]\ \frac{b \triangleright (w, s, i) :: (w', s, i) \qquad [\mathit{start}]\ \dfrac{b \triangleright (w', s, i) :: \varepsilon \qquad \textbf{valid}(f, w', i, \varepsilon)}{\langle N',\ (f \mapsto (w', s, i)) \rangle \xrightarrow{f \downarrow w'} \langle (w \mapsto f) \cup (w' \mapsto f),\ (f \mapsto (w', s, i)) \rangle} \qquad \neg\textbf{valid}(f, w, i, f)}{\langle N',\ (f \mapsto b) \rangle \xrightarrow{f \downarrow w'} \langle (w \mapsto f) \cup (w' \mapsto f),\ (f \mapsto b) \rangle}$$

In the remainder, we use the following notation: given the sequence of labels $\overline{\alpha} = \alpha_1 \ldots \alpha_n$, we use $\langle N, \Delta \rangle \overset{\overline{\alpha}}{\hookrightarrow} \langle N_n, \Delta \rangle$ to denote that $\langle N, \Delta \rangle \xrightarrow{\alpha_1} \langle N_1, \Delta \rangle \ldots \xrightarrow{\alpha_n} \langle N_n, \Delta \rangle$.

We close the section with the definition of the problems we consider in our development: *reachability* concerns checking whether a function f can be scheduled on a worker w starting from an initial network of workers, while *co-occurrence* checks whether two functions f and g can be both present in the list of functions scheduled on a worker w. By *initial* network of workers, we mean a network N where all workers have no functions already scheduled, i.e., $N(w)=\varepsilon$ for every available worker w.

Definition 5 (Reachability). *Given a scheduling policy Δ, an initial network N, a function f, and a worker w, the* reachability *problem* `Reach`*(Δ,N,f,w) consists of checking if there exists a sequence of labels $\overline{\alpha}$ such that $\langle N,\Delta\rangle \overset{\overline{\alpha}}{\hookrightarrow} \langle N',\Delta\rangle$ with $N'(w)=\overline{f}$ such that $f\in\overline{f}$.*

Definition 6 (Co-occurrence). *Given a scheduling policy Δ, an initial network N, two functions f and g, and a worker w, the* co-occurrence *problem* `CoOccur`*(Δ,N,f,g,w) consists of checking if there exists a sequence of labels $\overline{\alpha}$ such that $\langle N,\Delta\rangle \overset{\overline{\alpha}}{\hookrightarrow} \langle N',\Delta\rangle$ with $N'(w)=\overline{f}$ such that $f\in\overline{f}$ and $g\in\overline{f}\setminus f$.*

4 Reachability and Co-occurrence in the APP-calculus

We prove that both reachability and co-occurrence have linear time complexity in the APP-calculus.[8] The result follows from two properties formalised in Lemmas 1 and 2.

Lemma 1 formalises a sort of anti-monotonic property of scheduling: if we can schedule a function f on a worker, we can do it if the worker hosted fewer functions.

Lemma 1. *Let $\overline{i}$ be a list of invalidation conditions that does not contain invalidations of the kind* `affine` *F, N a network of workers, Δ a scheduling policy, w a worker belonging to the domain of N, and f, g functions belonging to the domain of Δ. We have that* **valid**$(f,w,\overline{i},\overline{f})$ *implies* **valid**$(f,w,\overline{i},\overline{f}\setminus f')$*, for every function f'.*

Notice that Lemma 1 does not hold if we consider also the invalidation conditions `affine` F. In fact, if $f\in F$, the **check** predicate in Fig. 4 checks whether $f\in\overline{f}$, and if there is only one occurrence of f in $\overline{f}$ we have that $f\notin(\overline{f}\setminus f)$. We can then conclude that the property stated by the above lemma does not hold in the full aAPP-calculus due to the presence of the invalidation conditions `affine` F, while it holds in the APP-calculus (and in a fragment of the aAPP-calculus which does does not use `affine` F, named negatively-polarised fragment of aAPP in Sec. 6).

Lemma 2 intuitively states that, when considering the reachability on a worker w, we can consider a simpler script with only the blocks that involve w and remove the other workers from said blocks; as defined by the auxiliary function `simple`. Formally, `simple`$(\Delta,w)=\Delta'$ where Δ' has the same domain as Δ and if $\Delta(f)=\overline{b}$ then $\Delta'(f)=$ `fltr`$(\overline{b},w)$ where `fltr`$(\varepsilon,w)=\varepsilon$ and `fltr`$((\overline{w},s,\overline{i})::\overline{b},w)=\begin{cases}(w,s,\overline{i})::\texttt{fltr}(\overline{b},w) & \text{if } w\in\overline{w}\\ \texttt{fltr}(\overline{b},w) & \text{otherwise}\end{cases}$

Intuitively, Lemma 2 holds because the other workers (not w) that have higher priority than w can be invalidated by assigning functions to them until they reach full capacity.

Lemma 2. *Given a scheduling policy Δ, an initial network of workers N, a function f, and a worker w, we have that* `Reach`*(Δ,N,f,w) iff* `Reach`*(*`simple`*$(\Delta,w),N,f,w)$.*

[8] We recall that details about the proofs of our results are in the companion paper [19]. In this paper, we give an idea of the proof structure and discuss the proof techniques.

We now move to the proof that the reachability problem has linear time complexity in the APP-calculus. Intuitively, the proof proceed as follows. In the light of Lemma 2 we can restrict to the simplest simple($\mathit{\Delta}$,w) scheduling policy which considers only the target worker w. If simple($\mathit{\Delta}$,w) contains at least one scheduling block capable of scheduling the function f on w when w is empty, then we know that Reach($\mathit{\Delta}$,N,f,w). On the opposite direction, if Reach($\mathit{\Delta}$,N,f,w) we know that there is a sequence of scheduling events ending with the scheduling of f on w. By the anti-monotonic property formalized in Lemma 1, we have that the scheduling of f on w can be anticipated at the beginning (when w is empty). Verifying whether there is a scheduling blocks for function f on an empty worker w in simple($\mathit{\Delta}$,w) corresponds to a linear-time check of the lists of invalidation policies associated with blocks of f and check if the invalidation conditions are not met.

Theorem 1. *Reach($\mathit{\Delta}$,N,f,w) has linear time complexity in the APP-calculus.*

Similarly, we prove that also co-occurrence has linear time complexity by reducing the problem to solving the reachability problem twice.

Theorem 2. *CoOccur($\mathit{\Delta}$,N,f,g,w) has linear time complexity in the APP-calculus.*

5 Reachability and Co-occurrence in the aAPP-calculus

We now prove that both reachability and co-occurrence become PSPACE-complete in the full aAPP-calculus, i.e., when consider (anti-)affinity constraints. While this result formalises the expressiveness gap – i.e., aAPP can specify more sophisticated scheduling policies than APP– it severely complicates reachability verification, which motivates the introduction of our planner-based tool in Sec. 7.

To prove that reachability is PSPACE-complete in the aAPP-calculus we first need to show that the problem is in PSPACE. This first result follows from the following observations: (i) a network $N \in \mathcal{N}$ can be represented in memory by a matrix (with a row for each worker, a column for each function, and the number of instances of a function on a worker stored in the corresponding cell) which occupies a polynomial amount of space; (ii) reachability can be detected with a nondeterministic algorithm using that amount of space; (iii) NPSPACE coincides with PSPACE [46]. Then we need to prove that the problem is PSPACE-hard. This can be proved by reduction from the single post-condition variant of PLANSAT which was proved to be PSPACE-hard by Bylander [11]. In a few words, this problem consists of checking whether a target state can be reached starting from an initial state represented by a finite set of ground atomic conditions that are considered initially true, by applying operators that change the current state by checking some positive and negative pre-conditions (the positive ones should hold, the negative ones should not hold) and then modifying the state by making a ground atomic condition either true or false. This problem can be modeled in aAPP by means of an appropriate encoding which represents each ground atomic condition with two possible functions, one representing the condition true and one representing the condition false. The operators are modeled by means of a scheduling policy capable of: (i) scheduling a specific function f_o representing one operator o only if the functions representing the corresponding pre-conditions are already scheduled, and (ii) scheduling the function representing the post-condition of o only if the function f_o is already scheduled. We use affinity to check the presence of the functions representing the pre-conditions, while anti-affinity is used to guarantee that if the post-condition requires to change the value of a ground atomic condition, the function representing the new value true (resp. false) is not scheduled contemporaneously with the opposite previous value false (resp. true).

Theorem 3. `Reach`*(Δ,N,f,w) is PSPACE-complete in the aAPP-calculus.*

We have that the co-occurrence problem can be reduced to the reachability problem. This allows us to conclude the following.

Corollary 1. `CoOccur`*(Δ,N,f,g,w) is PSPACE-complete in the aAPP-calculus.*

6 Reachability and Co-Occurence with Polarised Affinities

We move to investigate *polarised* aAPP sublanguages, where one can express either affinity or anti-affinity but not mix them. Practically, we want to understand if we can use algorithmically tractable analyses with fragments of aAPP, e.g., when a script uses only the syntax of a polarised fragment. We call *positively-polarised* aAPP the variant with only affinity and *negatively-polarised* aAPP the one with only anti-affinity.

The calculus corresponding to positively-polarised aAPP is the fragment of the aAPP-calculus that does not contain invalidation conditions of the kind `anti_affine` F (generated by the forbidden anti-affinity requirements); while the calculus corresponding to negatively-polarised aAPP is the fragment of the aAPP-calculus that does not contain invalidation conditions of the kind `affine` F (generated by the forbidden affinity requirements). We study the complexity of the reachability and co-occurrence problems for the two fragments. We show that in the positively-polarised fragment the problems are NP-hard (hence not tractable unless P=NP), while in the negatively-polarised fragment both problems have linear time complexity.

First, we prove that in the positively-polarised fragment of the aAPP-calculus the reachability and co-occurrence problems are NP-hard. The proof is by reduction from 3SAT, a well known NP-hard problem [14] consisting of checking the satisfiability of a boolean formula in conjunctive normal form, where each clause has at most three literals (where a literal is a boolean variable or its negation). The reduction is based on a scheduling policy which allows for the initial scheduling of functions which represent a possible assignment of boolean values to the variables in the formula. Then, affinity is used to allow the scheduling of functions representing whether a given clause is satisfied by the assignment. These functions are forced to be scheduled in sequence, from the first clause to the last one. In this way, the functions corresponding to the last clause can be scheduled if-and-only-if all the clauses can be satisfied, i.e., the formula is satisfiable.

Theorem 4. `Reach`*(Δ, N, f, w) and* `CoOccur`*(Δ, N, f, g, w) are NP-hard in the positively-polarised fragment of the aAPP-calculus.*

```
(:goal
  (= (number_of_f_in_W f w) 1)
)
```

```
(:goal
  (and (= (number_of_f_in_W f1 w) 1)
       (= (number_of_f_in_W f2 w) 1)
))
```

Fig. 5. Encoding of `Reach` and `CoOccur` in PDDL, on the left and right side respectively.

Note that the proof of linear time complexity of `Reach`(Δ, N, f, w) and `CoOccur`(Δ, N, f, g, w) cannot be applied to the positively-polarised fragment of the aAPP-calculus because Lemma 1 considers scheduling policies that do not contain invalidations of the kind `affine` F. On the other hand, the proof can be applied to the negatively-polarised fragment of the aAPP-calculus, thus we can conclude the following.

Theorem 5. `Reach`*(Δ,N,f,w) and* `CoOccur`*(Δ,N,f,g,w) have linear time complexity in the negatively-polarised fragment of the aAPP-calculus.*

7 Automated verification of aAPP scripts

Given the results of the `Reach` and `CoOccur` problems for APP and aAPP, we have a firmer grasp of how to automate their analysis, provided an initial configuration. Since we proved that the complexity of this analysis for aAPP is PSPACE-hard, we present a proof-of-concept solution developed on existing solvers for that class of problems.

Specifically, we encode the `Reach` and `CoOccur` problems as automated planning tasks. We choose to use PDDL [23], i.e., the de-facto standard for encoding planning problems. Mainly, PDDL specifications include two parts: a domain description, regarding all available actions and their preconditions and effects, and a problem description, modelling the initial state of the specific problem instance and the desired goals.

The choice of PDDL has many advantages, the main ones include: *i*) it lets us use different dedicated and efficient planning solvers, *ii*) it naturally provides a witness (i.e., a sequence of actions) leading to the violation of some desired property (e.g., if two functions shall be anti-affine and we find an instance where they co-occur, we can see a trace that produces the offending result), and *iii*) it lets users express more elaborate goal configurations than reachability and co-occurrence (e.g., we can ask if there exists a worker that can host a given set of functions and still preserve a certain capacity).

7.1 Encoding of the PDDL domain

To define the domain, we encoded the semantic rules reported in Fig. 4. The two main encoded actions are the *(start, _, _)* and *(done, _, _)* transitions. Each action alters the state appropriately, e.g., removing a function from a worker increases its available capacity accordingly. The encoding of the rules uses a combination of boolean predicates (e.g., for the affinity of a block) and numerical functions (e.g., for capacity constraints). We did not encode the *(fail, _, _)* transition since it does not affect the state– it is equivalent to a null operation in the context of a plan.

7.2 Encoding of the PDDL problems

Since `aAPP`'s semantics works on `aAPP` scripts and configurations (cf. Fig. 4), we define the problem by encoding an input `aAPP` script with a companion configuration definition, which describes the existing workers and functions with the respective capacities/occupancies – the latter, defined using PDDL *numeric fluents* to model non-binary, discrete resources. Given a configuration, in its initial state workers have no allocated functions and maximal capacity. We define the `Reach` and `CoOccur` problems using of the proposition *number_of_f_in_W*, which indicates the number of copies of a function on a given worker in the final state. The code on the left of Fig. 5 exemplifies the `Reach` problem, where we check if we can satisfy the `goal` of allocating at least one copy of function `f` on worker `w`; on the right of Fig. 5, we realise `CoOccur` as the conjunction of two *number_of_f_in_W*, checking if worker `w` can host both functions `f1 and f2`.

7.3 Tool Components and Support for Advanced Queries, an Example

To provide a smoother experience to users, we provide users with a simple, YAML-like notation for configurations (exemplified in Fig. 6) and translators that generate the PDDL encoding of part of the problem, given a configuration file and an `aAPP` script. The last piece of the problem, required from the user, is the definition of the `goal`, which uses the PDDL notation. The encoding and its companion tools are available at [16].

We illustrate the experience of using our analysis tool by defining a configuration, an `aAPP` script, and a property. Let us consider the configuration in Fig. 6, with two workers, `local` and `public`, and three functions `f`, `u` and `g`, resp. tagged `private`, `unsafe`, and `generic`. We want to define an `aAPP` script such that *it never happens that we allocate* `f`*s with other non-*`f` *functions*. We encode this property in Fig. 7, where we use existential quantifiers to range over both workers and functions and check whether we can find a worker that can host both `f` and a function different from `f`. If the planner is unable to find a solution, the property holds for the provided `aAPP` script.

In Fig. 8, we show a possible candidate script. Running the planner with (the encodings of) the excerpts in Fig. 6 and Fig. 8 and the goal from Fig. 7, we obtain the output in the red box at the centre of Fig. 10. The output is a trace of actions that satisfies the goal; namely, the allocation of `f` with another function on the same worker. Specifically, we found that `f` can co-occur with a function `g`, tagged `generic`, on the `local` worker. Hence, we need to fix our candidate `aAPP` script to prevent the allocation of `generic` functions with `f`s. Looking closely at Fig. 8, we indeed notice that we forgot to indicate that also the `generic` (tagged) functions are anti-affine with the `private` ones. Remember that `affinity` in `aAPP` is not symmetric, so, to ensure that `private` functions are anti-affine with the other functions, we need to mark also all the latter anti-affine with the `private` ones.

Fig. 9 shows a fixed version of the `aAPP` script from Fig. 8 where we add mutual anti-affinity – the new `affinity` clause under the `generic` block. The new constraint makes the goal from Fig. 7 unsatisfiable, as shown in the green box next to Fig. 9.

Performance-wise, we tested our encoding using the MetricFF [28] planner on examples similar to Fig. 10, obtaining results (either providing a solution or proving

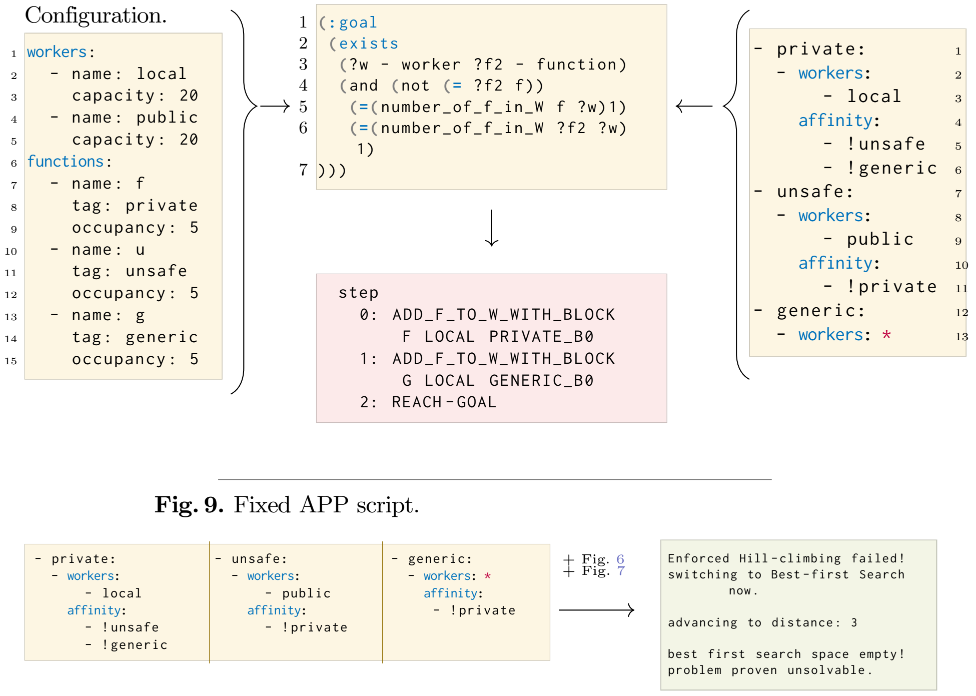

Fig. 6. Configuration.

```
workers:
  - name: local
    capacity: 20
  - name: public
    capacity: 20
functions:
  - name: f
    tag: private
    occupancy: 5
  - name: u
    tag: unsafe
    occupancy: 5
  - name: g
    tag: generic
    occupancy: 5
```

Fig. 7. Checker predicate.

```
(:goal
 (exists
  (?w - worker ?f2 - function)
  (and (not (= ?f2 f))
   (=(number_of_f_in_W f ?w)1)
   (=(number_of_f_in_W ?f2 ?w)
    1)
)))
```

```
 step
   0: ADD_F_TO_W_WITH_BLOCK
       F LOCAL PRIVATE_B0
   1: ADD_F_TO_W_WITH_BLOCK
       G LOCAL GENERIC_B0
   2: REACH-GOAL
```

Fig. 8. APP script.

```
- private:
  - workers:
      - local
    affinity:
      - !unsafe
      - !generic
- unsafe:
  - workers:
      - public
    affinity:
      - !private
- generic:
  - workers: *
```

Fig. 9. Fixed APP script.

```
- private:
  - workers:
      - local
    affinity:
      - !unsafe
      - !generic
```

```
- unsafe:
  - workers:
      - public
    affinity:
      - !private
```

```
- generic:
  - workers: *
    affinity:
      - !private
```

```
Enforced Hill-climbing failed!
switching to Best-first Search
        now.

advancing to distance: 3

best first search space empty!
problem proven unsolvable.
```

Fig. 10. Example of incorrect aAPP script for the *singleton* example (top) and fix (bottom).

its non-existence) within a few milliseconds – we obtain a solution in 2.4ms for the script in Fig. 8 and the exhaustion for the one in Fig. 9 in 4ms, averaged over 10^4 runs.

Here, we focus on the feasibility of our approach based on using off-the-shelf planning solvers, hence we leave as future work the study of which planners work best and how well they scale under different cases. Like SAT solvers, which can solve large SAT problems despite their NP-hardness, we conjecture that planners can efficiently solve the majority of reachability and co-occurrence problems. Intuitively, except for those combinations of configuration, script, and predicates that require the saturation of workers to reach the goal, the plan to prove the violation of the property is rather short and therefore solvers can find it quickly.[9]

8 Related Work and Discussion

Common scheduling problems are renownedly NP-complete [56]. However, we are not aware of results on the complexity of scheduling problems where affinity is the key point for turning the complexity of the problem from linear-time to PSPACE-hard (as formalised by Theorems 3 and 4). We consider these result an interesting point to highlight in relation to other computational models. We initially thought that the

[9] The complexity of the planning problem becomes pseudo-polynomial with bounded plan length – $O(n^k)$ with n size of the planning encoding and k maximal length of the plan.

source of the increase of complexity was mainly due to anti-affinity (i.e., check the absence of a set of functions F before scheduling another function g on a worker). In fact, in many computational models, "negative" tests increase the complexity of verifying properties. For example, it is well-known that in register-based computational models (RAMs) the ability to test whether a register is empty is necessary for RAMs to be Turing complete, while in Petri nets reachability and coverability problems are decidable, but they become undecidable in Petri nets extended with inhibitor arcs (i.e., the ability to test whether a place is empty).

Regarding Petri nets, it is interesting to observe that our results exhibit notable differences compared to the known complexity/undecidability results in that context. Our reachability problems are expressed as the possibility to reach a configuration with at least some given functions on some given worker. This formulation has a strong correspondence with the coverability problem in Petri nets, consisting of checking the reachability of a marking with at least a given amount of tokens in some given places. Coverability is EXPSPACE-hard in Petri nets [36] while our reachability problems are PSPACE-complete as in 1-safe Petri nets [13], the fragment of Petri nets where places can contain at most one token. Our calculus is significantly different from 1-safe Petri nets because we need to count how many instances of a function are present in a worker. Moreover, our semantics also incorporates priorities because the scheduling blocks must be considered in order, i.e., a block is considered only if the previous ones fail. Differently from our context, the addition of priorities to Petri nets makes them Turing complete [26], hence coverability becomes undecidable.

Another interesting highlight concerns the linear time complexity of co-occurrence in the APP-calculus (Theorem 2) and in negatively-polarised aAPP (Theorem 4). On the positive side, we have that we can efficiently check possible violations of security properties of scripts, like the co-location of "trusted" and "untrusted" functions. However, this efficiency comes at a cost, i.e., the impossibility to express any reasonable form of affinity. As a corollary of Lemma 1, we have that if $\langle N, \Delta \rangle \xrightarrow{f\downarrow w} \langle N_1, \Delta \rangle$ in a network N where the worker w already hosts instances of g, we have that also $\langle N', \Delta \rangle \xrightarrow{f\downarrow w} \langle N_1', \Delta \rangle$ where $\langle N', \Delta \rangle$ is the same as $\langle N, \Delta \rangle$, but with all occurrences of g removed from w. Such property implies that scheduling policies expressed in APP, or in the negatively-polarised fragment of aAPP, cannot admit the scheduling of a function f in case we want f to be affine with g, i.e., we want the guarantee that f can be scheduled only on workers which already host instances of another function g.

Looking at the industry, we see platforms using both affinity and anti-affinity. Azure Service Fabric [39] provides a notion of *service affinity* that ensures the placement of replicas of a service on the same nodes as those of another, affine service. Another example is Kubernetes, which has a notion of *node affinity* and *inter-pod (anti-)affinity* to express advanced scheduling logic for the optimal distribution of pods [34]. Affinity and anti-affinity have been also studied in the context of microservices. Proposals in this direction are by Baarzi and Kesidis [7], who present a framework for the deployment of microservices that infers and assigns affinity and anti-affinity traits to microservices to orient the distribution of resources and microservices replicas on the available machines; Sampaio et al. [44], who introduce an adaptation mechanism for microservice deployment based on microservice affinities (e.g., the more messages microservices exchange the more affine they are) and resource usage; Sheoran et al. [49], who propose an approach that computes procedural affinity of communication among microservices to make placement decisions. Overall, the mentioned work proves the usefulness of affinity-aware deployments at lower layers than FaaS (e.g., VMs, containers, microservices) but we are not aware of any formal study on the semantics and expressiveness of these constructs.

In the literature on serverless, proposals might adopt affinity and anti-affinity relations to encode and complement desirable properties of function scheduling, otherwise implemented via ad-hoc, platform-wide policies. For instance, some works present serverless architectures that enable the efficient composition of functions co-located on the same host [3,50,43]. Here, aAPP can help to parametrise the co-location of functions, expressed in terms of affinity and anti-affinity constraints. Another example regards security, e.g., Pubali et al. [15] present a serverless platform where developers can constrain the information flow among functions to avoid attacks due to container reuse and data exfiltration. In this case, aAPP can complement flow policies with affinity and anti-affinity constraints that restrict the co-tenancy of functions and their flows of communication. Another interesting proposal, Palette [2], uses optional opaque parameters in function invocations to inform the load balancer of Azure Functions on the affinity with previous invocations and the data they produced. While Palette does not support (anti-)affinity constraints, it allows users to express which invocations benefit from running on the same node. We deem an interesting future work extending aAPP to support a notion of (anti-)affinity that considers the history of scheduled functions.

Closing our revision of related work, we mention our previous publication [18], that presents an alternative definition of a semantics for the APP language. The differences between that work and the aAPP-calculus we present are twofold. First, we consider also affinity-aware scheduling policies, i.e., we model APP's semantics and extend it with affinities, giving us the aAPP language. Second, we define a more elegant and simplified semantics. Indeed, the previous semantics definition [18] considers three distinct layers (for $[\mathit{Workers}]$, $[\mathit{Blocks}]$, and $[\mathit{Configurations}]$), while the semantics we propose has just one layer. Moreover, our semantics omits the transition (fail, f) – used in the alternative semantics [18] to model the failure of an attempt to schedule a function f – because it is irrelevant for the study of the reachability and co-occurrence problems.

9 Conclusion and Future Work

We shed some light on formal characterisations of reachability analysis of FaaS scheduling policies and show immediate application of these mathematical results. We first introduce the aAPP-calculus to rigorously reason on the semantics of function scheduling that includes affinity constraints and then proceed to study the expressiveness of aAPP. We start with APP, which is a subset of aAPP that does not express either affinity or anti-affinity constraints. Performing our analysis of APP, we show that it is unfit for expressing interesting (anti-)affinity constraints between functions. Turning to aAPP, we formally prove the increment of expressiveness of adding affinity by showing that affinity-aware reachability problems, having linear time complexity in APP, turn out to be PSPACE-complete in aAPP. The proof of PSPACE-hardness is by reduction from PLANSAT, a well-known planning problem. We complement the above results by studying fragments of aAPP that can only express either affinity or anti-affinity constraints, proving that the complexity increase derives from affinity.

We summarise the complexity bounds of the co-occurrence (and reachability) problem of APP, aAPP, and the latter's affinity-only and anti-affinity-only fragments.

	APP	neg. polarised aAPP	pos. polarised aAPP	aAPP
Lowerbound	Linear	Linear	NP	PSPACE
Upperbound	Linear	Linear	PSPACE	PSPACE

Given the above results, we define a relationship from aAPP to PDDL – a standard language for planning problems – and realise a tool for automatic verification of scheduling properties for aAPP that exploits off-the-shelf PDDL solvers.

Directions for future work include capturing configurations whose workers can change, i.e., when workers (dis)appear while the platform is running. In such dynamic scenarios, the configuration space is infinite, which could have repercussions on the complexity/decidability of the reachability and co-occurrence problems. We also plan to define a timed semantics for aAPP, e.g., to specify the expected execution time of functions on workers. One such model could support quantitative analysis, e.g., to estimate the completion time of a FaaS application or the distribution of workers' load over time, i.e., it would allow us to quantitatively reason on policies, e.g., whether they could lead to bad performance or underutilisation of workers.

References

1. Abad, C.L., Boza, E.F., Eyk, E.V.: Package-Aware Scheduling of FaaS Functions. In: Proc. of ACM/SPEC ICPE. pp. 101–106. ACM (2018). https://doi.org/10.1145/3185768.3186294
2. Abdi, M., Ginzburg, S., Lin, X.C., Faleiro, J., Chaudhry, G.I., Goiri, I., Bianchini, R., Berger, D.S., Fonseca, R.: Palette load balancing: Locality hints for serverless functions. In: Proceedings of the Eighteenth European Conference on Computer Systems. pp. 365–380 (2023)
3. Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: SAND: Towards High-Performance Serverless Computing. In: 2018 Usenix Annual Technical Conference (USENIX ATC 18). pp. 923–935 (2018)
4. Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., Winstein, K.: Secure serverless computing using dynamic information flow control. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–26 (2018)
5. Amazon Web Services: AWS Lambda Getting Started. https://aws.amazon.com/lambda/getting-started/ (2024), accessed: Aug 2025
6. Apache Software Foundation: Apache OpenWhisk. https://openwhisk.apache.org/ (2025), accessed: Aug 2025
7. Baarzi, A.F., Kesidis, G.: Showar: Right-sizing and efficient scheduling of microservices. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 427–441 (2021)
8. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research advances in cloud computing, pp. 1–20. Springer (2017)

9. Banaei, A., Sharifi, M.: ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform. The Journal of Supercomputing (9 2021). https://doi.org/10.1007/s11227-021-04057-z
10. Baresi, L., Quattrocchi, G.: PAPS: A serverless platform for edge computing infrastructures. Frontiers in Sustainable Cities **3**, 690660 (2021)
11. Bylander, T.: The Computational Complexity of Propositional STRIPS Planning. Artif. Intell. **69**(1-2), 165–204 (1994). https://doi.org/10.1016/0004-3702(94)90081-7, https://doi.org/10.1016/0004-3702(94)90081-7
12. Casale, G., Artač, M., Van Den Heuvel, W.J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., et al.: Radon: rational decomposition and orchestration for serverless computing. SICS Software-Intensive Cyber-Physical Systems **35**(1), 77–87 (2020)
13. Cheng, A., Esparza, J., Palsberg, J.: Complexity Results for 1-safe Nets. In: Shyamasundar, R.K. (ed.) Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings. Lecture Notes in Computer Science, vol. 761, pp. 326–337. Springer (1993). https://doi.org/10.1007/3-540-57529-4_66, https://doi.org/10.1007/3-540-57529-4_66
14. Cook, S.A.: The Complexity of Theorem-Proving Procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA. pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047, https://doi.org/10.1145/800157.805047
15. Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., Bates, A.: Valve: Securing function workflows on serverless computing platforms. In: Proceedings of The Web Conference 2020. pp. 939–950 (2020)
16. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Reachability Analysis of Function-as-a-Service Scheduling Policies (Artifact). https://zenodo.org/records/17047090, accessed: Aug 2025
17. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling. In: IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022. pp. 337–342. IEEE (2022). https://doi.org/10.1109/ICWS55610.2022.00056
18. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Function-as-a-Service Allocation Policies Made Formal. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15219, pp. 306–321. Springer (2024). https://doi.org/10.1007/978-3-031-73709-1_19
19. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: On the Complexity of Reachability Properties in Serverless Function Scheduling. CoRR **abs/2407.14159** (2024). https://doi.org/10.48550/ARXIV.2407.14159, https://doi.org/10.48550/arXiv.2407.14159
20. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Affinity-aware Serverless Function Scheduling. In: 22nd IEEE International Conference on Software Architecture, ICSA 2025, Odense, Denmark, March 31 - April 4, 2025. pp. 49–59. IEEE (2025). https://doi.org/10.1109/ICSA65012.2025.00015, https://doi.org/10.1109/ICSA65012.2025.00015
21. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation. In: Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12571, pp. 416–430. Springer (2020). https://doi.org/10.1007/978-3-030-65310-1_29

22. Fission Project: Fission. https://fission.io/ (2024), accessed: Aug 2025
23. Fox, M., Long, D.: PDDL2. 1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research **20**, 61–124 (2003)
24. Ghallab, M., Nau, D.S., Traverso, P.: Automated planning - theory and practice. Elsevier (2004)
25. Google: Google Cloud Functions. https://cloud.google.com/functions/ (2024), accessed: Aug 2025
26. Hack, M.: PETRI NET LANGUAGE. Tech. rep., USA (1976)
27. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
28. Hoffmann, J.: Extending FF to Numerical State Variables. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02). pp. 571–575. wil, Lyon, France (Jul 2002)
29. Jia, Z., Witchel, E.: Boki: Stateful Serverless Computing with Shared Logs. In: Proc. of ACM SIGOPS SOSP. pp. 691–707. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483541
30. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud Programming Simplified: A Berkeley View on Serverless Computing. Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley (02 2019)
31. Kehrer, S., Scheffold, J., Blochinger, W.: Serverless skeletons for elastic parallel processing. In: 2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE. pp. 185–192 (2019)
32. Kelly, D., Glavin, F., Barrett, E.: Serverless computing: Behind the scenes of major platforms. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). pp. 304–312. IEEE (2020)
33. Kotni, S., Nayak, A., Ganapathy, V., Basu, A.: Faastlane: Accelerating Function-as-a-Service Workflows. In: Proc. of USENIX ATC. pp. 805–820. USENIX Association (2021)
34. Kubernetes: Assign Pods to Nodes using Node Affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/ (2025), accessed: Aug 2025
35. Kuntsevich, A., Nasirifard, P., Jacobsen, H.A.: A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform. In: Proc. of Middleware (Posters). pp. 3–4 (2018)
36. Lipton, R.: The reachability problem requires exponential space. Research report (Yale University. Department of Computer Science), Department of Computer Science, Yale University (1976), https://books.google.it/books?id=7iSbGwAACAAJ
37. Microsoft: Configuring and using service affinity in Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-advanced-placement-rules-affinity (2025), accessed: Aug 2025
38. Microsoft: Microsoft Azure Functions. https://azure.microsoft.com/ (2025), accessed: Aug 2025
39. Microsoft: Microsoft Azure Service Fabric Overview. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview (2025), accessed: Aug 2025
40. Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., Sukhomlinov, V.: Agile Cold Starts for Scalable Serverless. In: Proc. of HotCloud 19. USENIX Association, Renton, WA (jul 2019)

41. Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: SOCK: Rapid task provisioning with Serverless-Optimized containers. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 57–70 (2018)
42. OpenFaaS Ltd.: OpenFaaS. https://www.openfaas.com/ (2024), accessed: Aug 2025
43. Sabbioni, A., Rosa, L., Bujari, A., Foschini, L., Corradi, A.: A shared memory approach for function chaining in serverless platforms. In: 2021 IEEE Symposium on Computers and Communications (ISCC). pp. 1–6. IEEE (2021)
44. Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S.: Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications **10**(1), 1–30 (2019)
45. Sampé, J., Sánchez-Artigas, M., García-López, P., París, G.: Data-Driven Serverless Functions for Object Storage. In: Middleware. pp. 121–133. Middleware '17, ACM (2017). https://doi.org/10.1145/3135974.3135980, https://doi.org/10.1145/3135974.3135980
46. Savitch, W.J.: Relationships Between Nondeterministic and Deterministic Tape Complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970). https://doi.org/10.1016/S0022-0000(70)80006-X, https://doi.org/10.1016/S0022-0000(70)80006-X
47. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proc. of MICRO. pp. 1063–1075 (2019)
48. Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R.: Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proc. of USENIX ATC. pp. 205–218 (2020)
49. Sheoran, A., Fahmy, S., Sharma, P., Modi, N.: Invenio: Communication Affinity Computation for Low-Latency Microservices. In: Proceedings of the Symposium on Architectures for Networking and Communications Systems. pp. 88–101 (2021)
50. Shillaker, S., Pietzuch, P.: Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In: Proc. of USENIX ATC. pp. 419–433. USENIX Association (2020)
51. Silva, P., Fireman, D., Pereira, T.E.: Prebaking Functions to Warm the Serverless Cold Start. In: Proc. of Middleware. pp. 1–13. Middleware '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3423211.3425682
52. Smith, C.P., Jindal, A., Chadha, M., Gerndt, M., Benedict, S.: Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In: 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC). pp. 17–25. IEEE (2022)
53. Solaiman, K., Adnan, M.A.: WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. In: 2020 IEEE International Conference on Cloud Engineering (IC2E). pp. 144–153 (2020). https://doi.org/10.1109/IC2E48712.2020.00022
54. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Gonzalez, J.E., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful Functions-as-a-Service. Proc. VLDB Endow. **13**(12), 2438–2452 (Jul 2020). https://doi.org/10.14778/3407790.3407836
55. Suresh, A., Gandhi, A.: FnSched: An Efficient Scheduler for Serverless Functions. In: WOSC@Middleware. pp. 19–24. ACM (2019). https://doi.org/10.1145/3366623.3368136
56. Ullman, J.D.: NP-Complete Scheduling Problems. J. Comput. Syst. Sci. **10**(3), 384–393 (1975). https://doi.org/10.1016/S0022-0000(75)80008-0, https://doi.org/10.1016/S0022-0000(75)80008-0
57. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 133–146 (2018)
58. YAML: YAML Specification. https://yaml.org/spec/ (2025), accessed: Aug 2025