SmartTuning

1 Introduction

The deployment of Cloud-Native applications (CNA), relies on placing a collection of small, independent, loosely coupled and, ideally, self-managed services into a cluster. This architectural style brings reliability to the application, due to decoupled failures be isolated to avoid the crash of the whole application, and facilitates a decoupled evolution of theirs components. However, the high decoupling in this architectural style makes difficult the application management, considering the large number of distributed pieces of software that engineers should handle during the execution of the application.

Consequently, these services are mostly bundled up with several runtime abstraction layers to facilitate the their execution or runtime management. Some examples of these layers are application servers (e.g., OpenLiberty), runtime environment (e.g., Java Virtual Machine), hardware isolation (e.g., containers and virtual machines), and cluster orchestrator (e.g., Kubernetes).

Runtime abstractions layers are general purpose with many parameters (knobs), that the application engineer adjusts to better satisfy the use cases of the application. Many of these knobs are related to data structures or infrastructure mechanisms, so their adjustment affects the application performance. An important issue regarding these knobs is the colossal number of possibilities that an engineer has to consider to improve the application execution.

Every layer added to an application increases exponentially the number of possible configurations that the engineer has available to experiment. Making matters worse, an optimal configuration for an application is mutable, it should changes according to the environment, such as fluctuations of number of users requests, saturation of the cluster, version of libraries being used, and so on. For tuning an application the engineer has to evaluate many different configurations to find one that extracts the most of the application performance for a given situation.

To tackle this challenge, we are proposing SmartTuning, a mechanism to automatically identify and apply optimal configurations onto CNA regarding the environment where they are deployed. The rational of this mechanism is, based on changes of the application workload along the time (incoming requests and resources consumption), find out and continually updates, a (quasi-) optimal configuration which extracts the maximum performance of the application and, might, reduce its resource consumption.

2 Assumptions

We have made some assumptions to design SmartTuning. Following a non-definitive list of them.

- An application starts with a default configuration initially computed on its test phase. We assuming that the optimal configuration found during the tests are good enough to give the application a better initial configuration than a set by engineers manually.
- An application is an harmonic-ish function. We assuming that if an application is observed long enough, some patterns on its behaviour and resources usage appears, so that SmartTuning can learns and use it to improve how optimal configurations are computed and applied on applications.
- The application is a black-box function. Despite our assumption of the application be a sort of harmonic function (and then differentiable), we cannot found its max and min analytically. It is impossible to know *a priori* the frequency of the application's patterns neither the rules that define the slopes of their performance and resource usage along time. Therefore, we assume that best way to find an optimal configuration is using a sort of sequential optimization method.

- SmartTuning observes the patterns that come up from different aspects of the application, such resources consumption and inner behavior. We are assuming that all applications which SmartTuning observes are properly instrumented, and all metrics necessary are exposed by the application at tuning phase.
- We are assuming that the inner behavior of the application can be modeled as a histogram of which urls of an application are reached at runtime. Besides, we assuming that the inner behavior is responsible to make changes on resources usages and performance of the application, i.e., different shapes of the histograms result on a different resource consumption and application performance.
- Two equals workflow, i.e., both have same shape and intensity of all buckets, may results in different resources consumption and performance of the application. So, we are assuming that histograms cannot suppress the need of resource consumption metrics.
- Along the time, my patterns arise from the application and they can be grouped given a threshold. We are assuming that frequently patterns come up from the application, but they are not equals and may exists slight differences among them. Besides, we assuming that theses differences can be parametrized so that SmartTuning can groups these different patterns.
- We are assuming that when a different type of pattern comes up from the application means that it has new non-functional requirements. So, for each new workload type a new configuration should be set to the application to get its best on performance and resources consumption.
- We are assuming that the application remains that same, i.e., without new deployments, long enough so an optimal configuration be properly computed and the application still take advantage of the new configuration for a significant amount of time. These times are related to each application.
- The application can be modeled by observing a reduced scope. We are assuming that the incoming traffic of the application is the only input necessary to provides changes on the application performance. We also assuming that we can model the application behavior by measuring the only part incoming traffic, so that we can generalize the patterns of the application by observing the patters that arise from a single replica, for instance.

3 Design Overview

The SmartTuning operation unfolds into two main steps: to identify and group workloads of the application, and find out an optimal configuration to a given workload. Figure 1 depicts these steps.

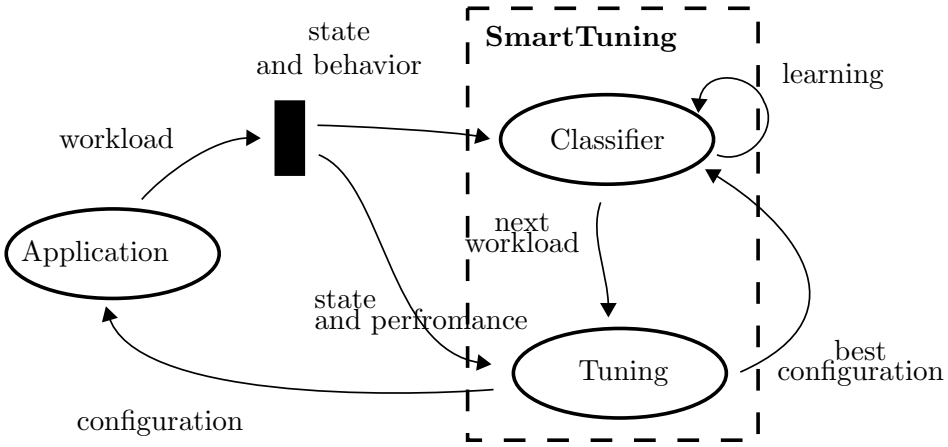

Figure 1: SmartTuning Overview.

We model an application as a black-box function $X : (c_k, t) \rightarrow w_t$, where X maps a configuration c_k in a time interval t to an workload w_t. A configuration is a set of knobs which the engineer can set in the application's runtime layers. An workload, $w_t = (b_t \in B, s_t \in S, p_t \in P)$, models the behavior, the state, and the performance of the application along a time interval t.

B are all possible behaviors of application and b_t models the application's behavior through the distribution of urls hits against the application endpoints in the interval t, i.e., urls histogram. In turn, S and P are all possible states and performance of an application, and s_t and p_t models respectively the resource consumption, e.g., CPU and memory, and application performance of the application, e.g., throughput and latency in the same interval t. Therefore, SmartTuning aims to improve the application X, SmartTuning $= \arg\max_c X(c, t)$. The improvement of X is given by a configuration c^* which maximize the performance p_t and might minimize the resource consumption s_t subject to an behavior b_t in a time interval t, Equation 1.

$$X(c^*, t) = \max X(c, t) = \begin{cases} \{s_t \mid s_t \in S \wedge \forall r_t \in S : s < r\} \\ \{p_t \mid p_t \in P \wedge \forall q_t \in P : p > q\} \end{cases}, \text{ subject to } b_t \tag{1}$$

Initially, the module named Classifier observes the workloads of the application under analysis. It groups the workloads based on similarities among behavior and states, normalizing them into types, Equation 2. As next step, the Classifier learns when a workload type comes up from the application, associating an workload type to time time-intervals while the application is running, Equation 3. Finally, it can forecast which workload type will come up from the application for a given time interval t, Equation 4.

$$K : w_t \rightarrow type_i \tag{2}$$

$$T : (t,\, type_i) \cup L, L = \{\forall t\, \exists \text{type} \mid t \rightarrow \text{type}_i\} \tag{3}$$

$$F : (t,\, L) \rightarrow type_i \tag{4}$$

After SmartTuning has learned about the workload's application, it is time of the module named Tuning figures out the best configuration for the application. The Tuning module uses the Classifier to forecast when and which will be the next workload type that will come up so it can compute a new configuration. For each workload type, Tuning computes several configurations trying to find out the one, c^*, which best improves the application. Then, SmartTuning updates the learning model L, appending the optimal configuration of each type, Equation 5.

$$T' : (\text{type}_i, c^*) \cup L', L' = \{\forall \text{ type } \exists\, c^* \mid \arg\max_c X(c, t) = c^*\} \tag{5}$$

When Tuning applies a configuration to the application, SmartTuning observes differences on application workload's performance p or resource consumption s. Because the application is a black-box function, and thus unknown whether it is differentiable, SmartTuning has to sequentially tries different configurations c_k looking for one which leads the application to its best – high performance or low resources consumption, Figure 2. To do so, Tuning has to try different configurations for a same type of workload and maintain internally a history of the configurations already tested against the application. Tuning applies a configuration c_k to the application and wait t' for new application's state and performance. The number of iterations to find an optimal configuration depends on the application and which optimization technique SmartTuning uses.

After n, $n \geq k$ iterations for workloads of a same type, SmartTuning learns which is the best configuration for that type. The optimization is driven by the types of workloads, whenever a new type comes up, Tuning has to internally changes its context to does not mess up with the computations made previously for the other types. To guarantee that each type has an optimal configuration, Tuning maintain an inner state for each type of workload under analysis. Figure 2 depicts its behavior. When Classifier forecasts a new workload type from the application, Tuning computes a new configuration and evaluates if it improves the application X.

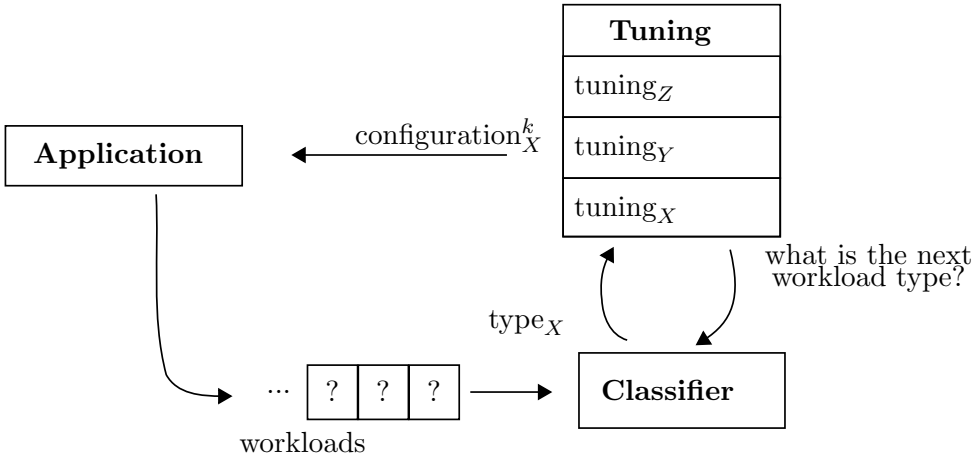

Figure 2: K-th tuning iteration for workload of type X.