A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform

M. Reza Hoseiny Farahabady, Albert Y. Zomaya, Fellow, IEEE, and Zahir Tari, Member, IEEE

Abstract—Lambda paradigm, also known as Function as a Service (FaaS), is a novel event-driven concept that allows companies to build scalable and reliable enterprise applications in an off-premise computing data-center as a serverless solution. In practice, however, an important goal for the service provider of a Lambda platform is to devise an efficient way to consolidate multiple Lambda functions in a single host. While the majority of existing resource management solutions use only operating-system level metrics (e.g., average utilization of computing and I/O resources) to allocate the available resources among the submitted workloads in a balanced way, a resource allocation schema that is oblivious to the issue of shared-resource contention can result in a significant performance variability and degradation within the entire platform.

This paper proposes a predictive controller scheme that dynamically allocates resources in a Lambda platform. This scheme uses a prediction tool to estimate the future rate of every event stream and takes into account the quality of service enforcements requested by the owner of each Lambda function. This is formulated as an optimization problem where a set of cost functions are introduced (i) to reduce the total QoS violation incidents; (ii) to keep the CPU utilization level within an accepted range; and (iii) to avoid the fierce contention among collocated applications for obtaining shared resources. Performance evaluation is carried out by comparing the proposed solution with an enhanced interference-aware version of three well-known heuristics, namely spread, binpack (the two native clustering solutions employed by Docker Swarm) and best-effort resource allocation schema. Experimental results show that the proposed controller improves the overall performance (in terms of reducing the end-to-end response time) by 14.9% on average compared to the best result of the other heuristics. The proposed solution also increases the overall CPU utilization by 18% on average (for lightweight workloads), while achieves an average 87% (maximum 146%) improvement in preventing QoS violation incidents.

Index Terms—Serverless Lambda Platform, Function as a Service (FaaS), Model Predictive Control, Dynamic Resource Allocation/Scheduling, Performance Degradation

1 INTRODUCTION

Lambda platform has recently emerged as a virtualized platform that can be used by clients to encapsulate a complex business logic into independent micro-modules that communicate with each other via provided application programming interfaces (API). So, the Lambda platform is responsible to respond to external events by calling the corresponding APIs in a loosely-coupled manner. The emergence of several commercial platforms such as Amazon Lambda [1], Google Cloud Functions [2], IBM and Apache OpenWhisk [3], [4], and Microsoft Azure [5] indicates a strong future growth of this new concept. Therefore, the Lambda paradigm can be effectively exploited as a layer to implement serverless event-driven computational service where the platform is responsible for running the users’ code following the occurrence of certain predefined events.

To enable better scaling at lower cost, a Lambda service provider may decide to host thousands of Lambda functions on the available resources. In many situations, achieving both user and operator goals may be problematic as they could be not compatible with each other [6]. The canonical example is the fast execution time demanded by the end-users versus the high resource utilization as the main objective of the service provider. To this end, most service providers employ workload consolidation as an effective technique to host multiple workloads into a single physical machine (PM) with the main aims of saving energy and enhancing total resource utilization.

However, designing an effective workload consolidation scheme is quite challenging. The problem is exacerbated if one considers the diversity in the kinds of applications that need to be hosted. While each application has its unique resource request characteristics, one should also take its sensitiveness to available resources into account. Such characteristics can be generally categorized as (1) the CPU and memory requirements, and (2) the incoming request rate of triggering events associated to each Lambda function. To a Lambda service provider, neither of such characteristics is known in advance [7]. In addition, each application has associated with a unique quality of service (QoS) level that is enforced by its owner and must be satisfied by the platform while taking resource allocation decision.

We designed a simple experiment to show the importance of employing a resource allocation schema with some kind of mechanisms to explicitly consider the runtime behavior of each host by analyzing the interference among collocated applications. We run 60 randomly selected Lambda functions taken from either SPEC-CPU2006 or PAR-
The rest of this paper is organized as follows. Section 2 provides the main challenges of shared resource interference among applications in a shared environment. Section 3 provides a formal statement of the research problem targeted in this paper. It also discusses the need for defining a metric for measuring QoS violations. Section 4 details the proposed controller and Section 5 presents the performance evaluation of the proposed solution on our in-house cluster. Section 6 briefly discusses the existing work, which is followed by our conclusions in Section 7. Appendix A provides an architectural overview and the main components in a Lambda platform. Appendix B briefly explains the limitation of consolidation of workloads in practice.

2 Background

This section briefly provides some of the main limitations of the current solutions that solely rely on operating-system level metrics to make the resource allocation decisions. More information about Lambda architecture can be found in Appendix A.

Shared resource interference: The issue of low resource utilization in a cloud platform has been extensively blamed as the root cause of energy consumption surge in modern data centers that adopt server virtualization methods (e.g., see [10]). Such a problem continues to exist in a Lambda platform, too. Consolidation of workloads is a promising technique to be employed for increasing the average resource utilization. However, the process of adapting this technique requires a careful planning to fulfill the users’ needs as expressed in the service level agreement while improving the efficiency of resource usage.

A consolidation decision is a multi-dimensional problem by nature that has to be carefully addressed by taking a diverse range of factors into account when making resource allocation decisions. By continuously monitoring the shared resource capacity, the associated interference amongst collocated functions and the resource utilization at every host, a mechanism is proposed to prevent a host to become a bottleneck. Our solution uses a prediction module to predict the state of nodes by making an estimate of the future request rates for each Lambda function. We benchmarked the proposed solution against three resource allocation heuristics (i.e., enhanced spread, enhanced binpack, and best-effort) with respect to three performance metrics (e.g., overall resource utilization, end-to-end response time for processing events, and QoS violation incidents). The results show that the proposed controller improves the overall performance by 14.9% on average. The proposed solution also increases the overall CPU utilization by 18% on average (for lightweight workloads), while achieves an average 87%-146% improvement in preventing QoS violation incidents.

TABLE 1

<table>
<thead>
<tr>
<th>Resource Utilization</th>
<th>Policy</th>
<th>H-1</th>
<th>H-2</th>
<th>H-3</th>
<th>H-4</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(%)</td>
<td>Spread</td>
<td>85</td>
<td>87</td>
<td>82</td>
<td>88</td>
<td>85.5</td>
</tr>
<tr>
<td></td>
<td>IA-Spr</td>
<td>86</td>
<td>84</td>
<td>89</td>
<td>86</td>
<td>86.3</td>
</tr>
<tr>
<td>Mem(%)</td>
<td>Spread</td>
<td>42</td>
<td>30</td>
<td>28</td>
<td>45</td>
<td>36.3</td>
</tr>
<tr>
<td></td>
<td>IA-Spr</td>
<td>39</td>
<td>40</td>
<td>44</td>
<td>40</td>
<td>40.8</td>
</tr>
<tr>
<td>MemBW(%)</td>
<td>Spread</td>
<td>51</td>
<td>19</td>
<td>18</td>
<td>54</td>
<td>35.5</td>
</tr>
<tr>
<td></td>
<td>IA-Spr</td>
<td>40</td>
<td>38</td>
<td>47</td>
<td>42</td>
<td>41.8</td>
</tr>
</tbody>
</table>

SECv2 benchmarks [8], [9]. While most of the applications in each benchmark show a similar CPU or memory access request in average, they have different memory bandwidth usage. Table 1 summarizes the impact of unawareness of shared resource interferences on the performance of hosts in a medium-sized platform with four dedicated servers, where a traditional spread policy or its enhanced (memory bandwidth interference aware) version is employed.

The spread policy uniformly allocates the available resources to the Lambda functions until a host becomes fully saturated or reaches to a predefined CPU utilization threshold. While the memory bandwidth on hosts 1 and 4 are heavily utilized (more than 50%), the spread policy does not migrate any Lambda function from those hosts to either host 2 or host 3 since the CPU/memory utilization level on all four hosts are quite similar. However, an interference-aware version of this policy avoids collocating workloads that heavily consume the memory bus, and tries to find another host that has a lower memory bandwidth usage to migrate some Lambda functions into. Equipping the traditional policy with such an improvement enables a proper balancing the memory utilization of buses across all hosts.

Yet another important challenge is to fulfill the quality of service (QoS) requirements specified in the SLA contract, e.g., the upper bound for the acceptable waiting time for processing the requests coming to each Lambda function. While a resource allocation strategy only provisions a new server when the total existing resource capacity within the entire cluster cannot cope with the incoming requests load, a QoS-aware resource manager must not only maximize the overall system utilization, but it also has to comply with the QoS enforcements. This can be achieved by reducing the number of QoS violation incidents that could occur as the result of unexpected spikes in the incoming traffic.

This paper presents a closed-loop (with feedback) controller based on the model predictive control (MPC) framework that considers shared resource interference among collocated Lambda functions when making resource allocation decision. The main goals include mitigating such undesirable interferences while trying to increase the overall resource utilization. Another objective of the proposed solution is to guarantee the requested QoS levels enforced by the end-users. The key feature of the proposed resource allocation scheme is to take both QoS enforcements and shared-resource contention among collocating workloads into account when making resource allocation decisions. By continuously monitoring the shared resource capacity, the associated interference amongst collocated functions and the resource utilization at every host, a mechanism is proposed to prevent a host to become a bottleneck. Our solution uses a prediction module to predict the state of nodes by making an estimate of the future request rates for each Lambda function. We benchmarked the proposed solution against three resource allocation heuristics (i.e., enhanced spread, enhanced binpack, and best-effort) with respect to three performance metrics (e.g., overall resource utilization, end-to-end response time for processing events, and QoS violation incidents). The results show that the proposed controller improves the overall performance by 14.9% on average. The proposed solution also increases the overall CPU utilization by 18% on average (for lightweight workloads), while achieves an average 87%-146% improvement in preventing QoS violation incidents.

The rest of this paper is organized as follows. Section 2 provides the main challenges of shared resource interference among applications in a shared environment. Section 3 provides a formal statement of the research problem targeted in this paper. It also discusses the need for defining a metric for measuring QoS violations. Section 4 details the proposed controller and Section 5 presents the performance evaluation of the proposed solution on our in-house cluster. Section 6 briefly discusses the existing work, which is followed by our conclusions in Section 7. Appendix A provides an architectural overview and the main components in a Lambda platform. Appendix B briefly explains the limitation of consolidation of workloads in practice.

2 Background

This section briefly provides some of the main limitations of the current solutions that solely rely on operating-system level metrics to make the resource allocation decisions. More information about Lambda architecture can be found in Appendix A.

Shared resource interference: The issue of low resource utilization in a cloud platform has been extensively blamed as the root cause of energy consumption surge in modern data centers that adopt server virtualization methods (e.g., see [10]). Such a problem continues to exist in a Lambda platform, too. Consolidation of workloads is a promising technique to be employed for increasing the average resource utilization. However, the process of adapting this technique requires a careful planning to fulfill the users’ needs as expressed in the service level agreement while improving the efficiency of resource usage.

A consolidation decision is a multi-dimensional problem by nature that has to be carefully addressed by taking a diverse range of factors into account [11], [12], [13]. One of the major impediments to embrace an effective consolidation technique is to deal with the fierce contention among the consolidated workloads to obtain the capacity of shared resources (e.g., CPU cache, memory bandwidth, and hard drive buffer) [14].

Some applications intrinsically use more shared resources than others [11]; therefore, they can severely slow down the performance of those neighboring applications...
that are highly sensitive to the available shared resources [15]. As an example, one can imagine a scenario in which each application evicts the data of other neighbors from the shared LLC when a context switch occurs; and hence causing other applications to experience an unacceptable delay to retrieve their data from the main memory in the subsequent cycles [17], [18].

Limitations of existing solutions: While relatively few studies have investigated the challenge of resource allocation or virtual machine mapping to hosts in distributed systems or hypervisor-based cloud environments [19], [20], the limitations of earlier studies exist and there is a great deal of work left to be done. A vast majority of such solutions rely solely on monitoring the operating-system-level performance metrics (e.g., CPU utilization and memory demand) to make resources allocation decisions. The main objective of such a strategy is to effectively balance the CPU and memory utilization demands across different physical nodes. Consequently, such resource allocation schemes in general are unable to avoid interference in the shared resource at micro-architecture level.

There exists some prior work, e.g., [21], [22], [23], which suggested to detect the interference of shared resources between collocated applications by performing offline profiling techniques over the target applications. While such a method could well work in some scenarios, we realize that applying this tactic in a practical Lambda platform has at least two major drawbacks. Firstly, performing an offline profiling might not always be possible in real scenarios, e.g., for a major service provider such as Amazon in which Lambda functions are constantly submitted by different users. Secondly, the state of the Lambda function can continuously change depending on the input rate, the live performance level of the underlying platform, the service rate, or the ever-changing condition among the application phases itself. So, as time passes, the state of the monitored application might be substantially different from what is obtained by performing the offline profiling.

3 Problem Statement

This section provides a formalisation of the resource management problem in a Lambda platform to incorporate awareness of resource interference and QoS enforcements.

3.1 Problem formulation

Let \( \mathcal{E} = \{e_1, e_2, \cdots \} \) denote a finite set of (software) event sources that generate some events, and \( \Lambda = \{f_1, f_2, \cdots \} \) denote a finite set of computational Lambda functions, each associated with exactly one event from \( \mathcal{E} \). We use the notation “\( e_j \rightarrow f_i \)” if such a relation exists. The main platform duty is to invoke \( f_i \) once its associated event \( e_j \) occurs. The Lambda service provider can opt to host multiple Lambda functions with different quality of service (QoS) enforcements in a shared platform to raise revenue. QoS enforcement is a contract between the service provider and the application owners to specify the minimum service level to be satisfied for each function during run time.

Let \( Q_i \) denote the QoS enforcement requested by each function \( f_i \). To force application owners to report truthfully the private value of \( Q_i \), the service provider has to employ a truthful mechanism (e.g., VCG mechanism [24]). To perform the requested service, the host machine’s kernel has to be configured correctly to allocate a certain amount of CPU or memory to each function \( f_i \).

Let \((\text{CPU}_{x}^{\mathcal{E}_i}, \text{Mem}_{x}^{\mathcal{E}_i})\) denote the minimum amount of two resources—CPU and memory, to be allocated to function \( f_i \) to serve the arriving requests during interval \( \tau = [t, t + \Delta t) \). In an ideal scenario, each Lambda function needs to be designed as a stateless (or idempotent) software component; hence, all the information that a Lambda function needs to do its requested functionality must be provided as its input parameters. By taking advantage of this fact, the platform resource manager can increase the number of concurrent threads associated with a particular Lambda function once a severe QoS violation is detected. In addition, it is not necessary for the resource manager to allocate the whole requested resources to a Lambda function a priori.

Let us assume that the server farm consists of \( m \) physical servers, denoted by \( \mathcal{M} = \{p_1 \cdots p_m\} \), where each of them has a certain computational or memory capacity, denoted by \((\text{CPU}_{p}, \text{Mem}_{p})\), \( \forall p_x \in \mathcal{M} \). To run a copy of each Lambda function \( f_i \), the resource manager can choose a subset of available machines, denoted by \( M_i \subseteq \mathcal{M} \), and allocate a portion of available resources in each machine to the function \( f_i \). Let \((\text{CPU}_{x}^{f_i}, \text{Mem}_{x}^{f_i})\) denote the amount of CPU and memory capacity in machine \( p_x \) to be allocated to function \( f_i \) during \( \tau \).

A desirable resource allocation should fulfill the total amount of resource capacity requested by each function (i.e., \( \sum_{p_x \in M_i} R_x^{f_i} \geq R_i \) for both resource types, \( R \in \{\text{CPU}, \text{Mem}\} \), but occasionally difficulties arise that need to be resolved. Particularly, in case of a sudden increase in incoming traffic load, the amount of requested resources by all functions is significantly higher than the available resource capacity, the resource manager has to waive some resource requests with regard to the level of QoS violation as well as other performance metrics (e.g., resource utilization level, migration cost, etc).

3.2 Micro-architecture Interference Identifier

The shared resource interference in micro-architecture level among the consolidated applications can severely degrade their running performance. The main reason is that collocated applications can fiercely compete with each other to acquire the available capacity of the last level cache (LLC) on a host; hence, they may evict the data of each other that already resides in LLC, which in turn can increase the access time of the other applications to re-fetch their data from the main memory in future [25].

To quantify the slowdown rate caused by a consolidation action, we pursue an effective method based on the solution initially introduced in [7], [25]. Subramanian and Wang et al. suggested that the impact of workloads’ contention on both LLC and memory bandwidth can be seen as a sudden rise in the memory bandwidth utilization, denoted by \( MBW_{util} \). Indeed, by analyzing the two standard hardware events of \( \text{UNC}_{\text{QMC NORMAL READS}} \), as an indicator of memory reads, and \( \text{UNC}_{\text{QMC WRITES}} \), as an indicator of memory
writes, one can calculate the utilization level of memory bandwidth \[7], \[25].

To quantify the amount of interference in shared resources for a host \[p_x\] at any given interval \(\tau\), one can define a set of finite threshold levels for the utilization of memory bandwidth, denoted by \(T_{MBW} = \{\text{Thr}_1, \text{Thr}_2, \cdots\}\), and then compare the measured value of \(MBW_{util}(p_x, \tau)\) with the threshold levels. The resource manager needs to associate a cost of executing a new application, such as \(f_i\), on a host \(p_x\) by measuring \(MBW_{util}(p_x, \tau)\) for each host.

We use a cost function of the following form for screening any decision that negatively imposes a high performance degradation due to interference in the shared resource level (i.e., cost-benefit analysis).

\[
\sigma_{p_x, \tau} = \eta^i \text{ if } \text{Thr}_i \leq MBW_{util}(p_x, \tau) \leq \text{Thr}_{i+1}, \quad (1)
\]

where \(\eta \geq 1\) is a fixed constant.

As a concrete example, let us assume there are three hosts, namely \(p_1, p_2\) and \(p_3\), and the four threshold levels are defined for each host. Let us assume that the measured values of \(MBW_{util}(p_i=1,2,3, \tau)\) lie within the first, third, and the fourth threshold levels, respectively. As a result, the cost of running a copy of a particular Lambda function on \(p_1\) (or \(p_3\)) would be 9 (or 16) by using the proposed cost function when \(\eta = 2\). In most practical scenarios, having three threshold levels for the utilization of memory bandwidth as \(\text{Thr}_1 = 0\%\), \(\text{Thr}_2 = 60\%\) and \(\text{Thr}_3 = 90\%\) is enough, as pointed by [25]. We use the same setting in our experiments, too.

### 3.3 The Semantic of QoS Violation

Depending on the SLA (Service Level Agreement), each Lambda function is associated with a QoS level that specifies the minimum service level to be fulfilled. In practice, a QoS enforcement level is expressed as a set if performance metrics, such as instruction per cycle (IPC) or the end-to-end response time (RT), that reflects the run-time behavior of Lambda functions. We present a mechanism for quantifying the QoS violation incidents to correctly respond to the service level violations occasionally occur in practice.

Applications can differently tolerate a performance degradation in the run-time. Some Lambda functions, such as those which are used in high-frequency trading applications or health monitoring domains, are highly sensitive to any delay in the response time, while others, such as those used in environmental monitoring or network intrusion detection, can be less sensitive to such an issue. Even in the context of a single application, clients can request a variety of service levels; hence, the service provider has to figure out a way to allocate resources based on prioritization of requests coming from different customers.

The work in [26] showed that applying a fair policy as suggested in [27], [28] does not always result in a true satisfaction for customers as expected. By applying a fair policy, a resource allocation policy would be considered as “good” as long as the following condition remains true. If one application experiences some amount of QoS violation, all other applications would also experience a similar issue (obviously it is not permissible from customers perspective). A strategy like [29] which tries to minimize the number of QoS violations across the cluster can yield adverse outcomes, too. Particularly, if the incident rates for multiple event sources suddenly rise simultaneously, a resource allocation needs to revoke some resources that are previously assigned to other applications. So, if the objective function only aims for minimization of the number of QoS violations, the resource manager may revoke resources from important applications (presuming that there is a way to quantify the importance of an application). To this end, we propose a metric called QoS detriment (1) to precisely measure such incidents explicitly, and (2) to clearly identify the QoS violation incidents for important applications.

### 3.4 QoS Detriment Metric

In the proposed architecture, we assume that there are exactly \(Q\) different classes in the SLA contract. We also assume that the desirable performance metric from the user’s perspective is the average end-to-end delay of processing all events during a given interval. So, a value of \(\omega_q^*\) is assigned to each class, where \(1 \leq q \leq Q\), that represents an upper-bound of the absolute delay that is accepted and must be guaranteed by the Lambda engine for all functions \(f_i\) that belong to a particular class \(q\). Let \(\omega_{f_i, \tau}\) be the measured amount of the intended performance metric for a function \(f_i\) during \(\tau\). We define the ratio of \(\omega_{f_i, \tau}^* = \frac{\omega_{f_i, \tau}}{\omega_q^*}\) as an indicator of attained QoS level in a given interval. While a “fair” policy tries to minimize the variance of \(\omega_q^*\) among all applications, we propose a different approach as follows.

To determine if a particular function experiences a QoS violation during a given interval, one can compare the value of measured target performance with the value of \(\omega_q^*\). However, avoiding QoS violations for all applications in the run-time is almost impossible. To relax such a limit, the resource allocation algorithm is allowed to violate QoS enforcements for some applications on demand. We define another function for each class of QoS contract, denoted as \(q(\tau)\), that reflects an upper bound for the percentage of QoS violations that is acceptable to run functions that belong to class \(q\). One good candidate for \(q(\tau)\) is a simple rule of \(q(\tau) = 1 - \frac{\omega_{f_i, \tau}}{\omega_q^*}\) for any function belong to class \(q\), where \(C\) is a fixed constant and \(Q\) denotes the total number of QoS classes exist in the SLA contract. Therefore, the delay of event processing can be higher than \(\omega_q^*\) only during \((1 - \frac{q}{C+1})\%\) of any arbitrary interval for functions belong to such a class.

The QoS detriment metric is introduced to quantify the amount of QoS violation occurring at each host \(p_x\) during an interval \(\tau\) as follows:

\[
D_{p_x, \tau} = \sum_{f_i \in \Lambda_{p_x, \tau}} \mathcal{V}(f_i), \quad (2)
\]

where \(\Lambda_{p_x, \tau} \subseteq \Lambda\) denotes the set of Lambda functions that experience a QoS violation during \(\tau\). The function \(\mathcal{V}(f_i)\) represents the importance weight associated with each \(f_i\). A good candidate for the importance weight is a linear rule that depends on the QoS enforcement level, too, such as \(\mathcal{V}(f_i) = q_{f_i}\), where \(q_{f_i}\) shows the QoS class which the function \(f_i\) belongs to. In this way, if a function demands a higher QoS enforcement value, it contributes more in the
QoS detriment metric by experiencing any QoS violation. The aim of this work is to progressively reduce the value of \( \sum_{p_i \in D} D_{p_i} \) over all available servers.

4 Feedback-Based Adaptive Resource Controller

An effective implementation of the Lambda platform needs to exploit the fact that the execution of different Lambda functions is independent from each other; hence, they can be performed concurrently. In other words, the resource manager can execute a same Lambda function as a “working thread” over different incoming event flows in a parallel fashion. Working threads are simply the replicas of the same Lambda function that are defined by the developer.

To achieve the best performance of the available resource capacity, the resource manager not only has to determine the right number of working threads per each Lambda function, but also should assign the right amount of the computing resource capacities to each working thread in a way to balance the computational load among workers and avoid any physical node to become a bottleneck. For example, a simple round-robin strategy that assigns the incoming events of the \( i \)-th Lambda function to the \( j \)-th working thread where \( j \equiv n i \) (\( n \) is the number of working threads) can neither effectively cope with the temporal changes in the incoming traffic (such as an abnormal rise) nor with the QoS requirement be fulfilled for each Lambda function.

In situations where the Lambda platform operates at a significant load of incoming events, we can use queuing theory to model the total waiting time of outstanding events in each Lambda function that is an essential step for predicting the consequences of a particular resource allocation decision on the performance of the system. Therefore, the resource manager can allocate/dismiss resources to achieve the requested end-to-end response time enforced by QoS rules. On the other hand, “control theory” arranges a set of precise mathematical principles to design a controller for a dynamic system. Particularly, a resource manager based on feedback control theory can regulate the set of performance parameters in response to a continuous feedback from the usage of resources as well as online states of the underlying platform [30].

4.1 Solution Overview

The proposed dynamic resource allocation schema is essentially a closed-loop (feedback) model predictive controller (MPC) that uses a model to predict the dynamic behavior of the underlying Lambda platform in the near future, and then makes the (near-) optimal decision based on the value of input vectors as the feedback loop. Our solution also incorporates the awareness of shared-resource interference when consolidate multiple applications into a single node. Figure 1 illustrates the core components and interfaces of the proposed controller.

While the usage of MPC framework is quite recent in computing platforms, it has been used in other research disciplines extensively, e.g., [31], [32], in which authors employ MPC framework as a solution to devise a capacity provisioning solution for cloud computing, and [29], [33] in which authors propose elastic scaling mechanisms based on MPC model in stream data processing. A comprehensive review in the theory and design of MPC can be found in [34].

Essentially, an MPC controller encompasses three major components; a model, a forecaster, and an optimizer. The model abstracts the complex behavior of the target system. The forecaster predicts the changes in the system state that is caused by any changes in the independent input parameters, and the optimizer module iteratively try to find the best solution for a set of controllable input variables within the near future intervals.

The key idea in an MPC framework is that at any given interval \( \tau \), the system output, denoted as \( y_\tau \), must follow an ideal set-point trajectory, denoted by \( r_\tau \), as its target value. The optimizer module aims to control the set of output variables in such a way that the system output converges towards the desired trajectory after a certain times of working at an exponential rate. Precisely, let \( T \) be the response speed factor and \( \epsilon_T = |r_\tau - y_\tau| \) represents the error between the current output and the desirable set-point trajectory. Such an error in the next \( s \) steps must satisfy \( \epsilon_{\tau+s} = e^{-s\frac{1}{T}}\epsilon_{\tau} \), where \( \frac{1}{T} \) is the length of the sampling interval. For example, setting \( \frac{1}{T} = \frac{1}{3} \) forces the output to smoothly converge to the desirable set-point trajectory by the next three sampling intervals. This choice not only imposes a low computational overhead, but also can mitigate the negative side effects of inaccurate modeling of the system or inaccuracy in the forecaster tool on the solution quality.

The proposed controller divides the entire operational course into equal profiling intervals and makes its decisions only at the beginning of each interval. Let \( \tau = 1, 2, \ldots \) denote such time-frames. The sequence of actions happening at the beginning of each interval \( \tau \) is as follows.

1) The profiling component collects some metrics to determine the resource usage and demands of the computing nodes in the entire platform. It also profiles the CPU and memory utilization and memory bandwidth usage of each Lambda function per host. The degree of shared resource interference can be determined by capturing two hardware events of \texttt{UNC\_QMC\_NORMAL\_READS} (memory reads) and \texttt{UNC\_QMC\_WRITE} (memory writes). This helps determining if any node is over-utilized.
2) The values for non-controllable exogenous parameters such as the future request rate for each event or the average processing time required for serving a request are estimated (see Section 4.3).

3) The system model is used to disclose the relationship between the target performance metrics such as end-to-end delays) and the configurable input variables.

4) The optimizer module is used to find the best possible setting for the configurable input variable over a period of $s \geq 1$ steps in the future.

5) The controller applies one step configurable input variable to the system (by considering the response speed parameter, $T$).

6) At time $\tau + 1$, the controller re-monitors the system’s behavior as a feedback loop and repeats the whole cycle of prediction, modeling, and optimization process.

4.2 System Model

A mathematical model uses a set of algebraic relationships between different components/metrics of a system to describe its behavior that (often) varies over time. A feedback controller needs a way to quantify such relationships that exist among the input variables of the system, e.g., the resource capacity allocated to each Lambda function, with the output variables, e.g., the average end-to-end response time experienced by each Lambda function. Developing a system model that accurately reflects the run-time behavior of the system can significantly improve the efficiency of the resource controller.

We employ a simple model to estimate an upper-bound of the average end-to-end response time that is experienced by each Lambda function $f_i$ during the next interval. The model uses a history of several parameters up to the past $h$ intervals to build such a model (let us call $h$ the retrospective factor). The model considers the effect of allocated computing resources (in terms of CPU share and memory capacity) on the response time of each Lambda function over the past observation periods. We place a hat (or bar) sign over a parameter to denote that an estimator (or the average) of the parameter being used in the context. Table 2 lists set of all performance variables that are acting as either input or output variables in the proposed model.

Let $\lambda_{j,\tau}$ be the arrival rate of events generated by source $e_j \mapsto f_i$ per unit time during an arbitrary interval $\tau' \leq \tau$, i.e., $T_A$ where $T_A$ is the mean inter-arrival time of events. Let $CPU_{\tau'}^{f_i}(p_x)$ (RAM$_{\tau'}^{f_i}(p_x)$) denote the capacity of CPU (RAM) in the machine $p_x$ that is dedicated for running function $f_i$ during any interval $\tau'$. We use a simple formula to model the average service rate of each Lambda function per physical server, denoted by $\mu_{f_i|p_x}$, by considering the amount of allocated computing resources (i.e., CPU and RAM) in such a machine over the prior periods retrospectively as follows.

$$\mu_{f_i|p_x} = \alpha_1 f_i \text{ Avg. } \{CPU_{\tau'}^{f_i}(p_x)\} + \alpha_2 f_i \text{ Avg. } \{RAM_{\tau'}^{f_i}(p_x)\}$$

where $\alpha_{1,2}$ indicates to what extent a Lambda function is sensitive to the amount of available CPU and memory, respectively. For example, a higher value of $\alpha_1 f_i$ associated with a Lambda function $f_i$ indicates that it highly depends on the available CPU. The coefficients of this model can be consistently calculated (and updated) by performing maximum likelihood estimation (MLE) method over past observations.

We employ Allen-Cunneen approximation of $G/G/M$ queue [35], a queue that consists of $M$ servers with a general distribution in arrival and service time, to estimate an upper-bound of the average end-to-end delay time (i.e., the sum of waiting time in the queue and the processing time), denoted by $W_{E,i}^f$, experienced by each event to be processed by its associated Lambda function, as follows.

$$W_{E,i}^f = \frac{\bar{P}_{cb,M}}{M \times \bar{\mu}_i \times (1 - \rho)} \left( \frac{CV_A^2 + CV_S^2}{2} \right), \quad (4)$$

where $CV_A = \sigma_{T_A}/E(T_A)$ is the coefficient of variation for inter-arrival time, $CV_S = \sigma_{T_S}/E(T_S)$ represents the coefficient of variation for service time, and $\bar{\mu}_i$ is the mean service rate (i.e., average number of processed requests per unit time) calculated by finding the statistical average of service rates per machine using Eq. (3). The term $\frac{CV_A^2 + CV_S^2}{2}$ is sometimes referred as the queue stochastic variability [36].

The term $\bar{P}_{cb,M}$ represents the probability that all servers in a queueing system are busy (i.e., the waiting time of a new customer is above zero). If there is only one worker, then this value would be equal to $\bar{P}_{cb,M} = \rho$, where $\rho < 1$ is the service traffic intensity (i.e., the utilization of a server). For the general case of $M \geq 2$, the following formula can be used to estimate the probability of waiting of a newly arrived customer.

$$\bar{P}_{cb,M} = \sum_{j=M+1}^{\infty} p_j = \frac{(M\rho)^M}{M!(1 - \rho)} \rho^0, \quad (5)$$

where $p_j$ is the probability that there are already $j$ customers staying in each queue. A simpler formula can be employed to approximate the value for the waiting probability distribution (as suggested in [37]), as follows.

$$\bar{P}_{cb,M} \approx \begin{cases} (\rho^M + \rho)/2 & \text{if } \rho \geq 0.7 \\ \rho^{M+1} & \text{otherwise.} \end{cases} \quad (6)$$

### Table 2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{e,j}^f$</td>
<td>Effective arrival rate of events emitted from $e_j$</td>
</tr>
<tr>
<td>CPU$_{\tau'}^{f_i}(p_x)$</td>
<td>Allocated CPU portion at $p_x$ to $f_i$</td>
</tr>
<tr>
<td>RAM$_{\tau'}^{f_i}(p_x)$</td>
<td>Allocated memory portion at $p_x$ to $f_i$</td>
</tr>
<tr>
<td>$W_{E,i}^f$</td>
<td>End-to-end delay of events when using $M$ servers</td>
</tr>
<tr>
<td>$\rho = \lambda/\mu$</td>
<td>Server utilization (in case of one server)</td>
</tr>
<tr>
<td>$\bar{P}_{cb,M}$</td>
<td>Probability for a non-empty system</td>
</tr>
<tr>
<td>$CV_A$</td>
<td>Coefficient of variation of inter arrival times</td>
</tr>
<tr>
<td>$CV_S$</td>
<td>Coefficient of variation of service time</td>
</tr>
</tbody>
</table>
It is worth mentioning that, although the A-C formula was developed using some computational-based estimation techniques without formal proof, it often gives a very good approximation to the average waiting time of customers in a $G/G/M$ queue. As reported by Tanner in [38], the values obtained by the A-C formula were within 10% of their actual values in most scenarios.

### 4.3 Prediction Model

The prediction model provides an estimation of value for non-controllable input vectors for the purpose of optimization process. At any given time $\tau$, the controller forecasts the value of $\lambda^{\tau+1}$ for the future intervals $\tau' > \tau$. Let $X_\lambda$ be the random variable associated with parameter $\lambda$. If the probability distribution of $X_\lambda$ is known in advance, then $\lambda^{\tau+1}$ can be estimated by applying stochastic analysis over the recent observations. Otherwise, an estimation tool such as time-series analysis [39], Kalman filter [40], or auto regressive integrated moving average (ARIMA) model [41] can be employed. We use ARIMA model in this project.

Using ARIMA model, the future values of a random variable can be estimated ahead based on a series of its past values as follows [41].

$$\hat{X}_{\tau+1} = \kappa + \varepsilon_{\tau} + \sum_{\ell=0, \ldots, h} \beta_{\ell} \varepsilon_{\tau-\ell} + \theta_{\ell} \varepsilon_{\tau-\ell}, \quad (7)$$

where $\kappa$ is a constant and $\varepsilon$’s are independent and identically distributed errors taken from a normal distribution with mean zero and a finite variance, e.g., a white noise process. $\beta$’s and $\theta$’s are coefficients to be calculated (and updated) using least-squares regression method right after a new observation. To estimate the $s$ steps ahead for a random variable using ARIMA model, one can simply repeat the one-step process in a conditional form.

### 4.4 Optimization Process

The global optimizer determines the allocation of processing budget to each Lambda function per host (in analogy with the vertical scaling), and the number of Lambda function working threads per host (horizontal scaling). There are three main objectives to be satisfied by the optimizer: (1) keep the CPU utilization level within an accepted range, (2) reduce the sum of QoS detriment cost over all machines, and (3) avoid fierce interference among collocated working threads to obtain shared resource capacity. At each step $\tau$, the controller try to optimize an objective function as the weighted sum of three mentioned cost functions over the future $s$ steps.

- **Resource utilization cost ($C(U)$).** This function penalizes derivations from the desirable resource utilization at each machine. The study in [42] showed that an ideal CPU utilization in the steady state working condition should be between $U_{CPU}^{upper} \in [70\% - 90\%]$ to reach the best balance between performance and energy consumption. We employ a cost function that penalizes more any derivation from the upper bound comparing to the derivation from the lower threshold, as shown below. Such a choice avoids the exploitation of the full CPU capacity that may turn an over-utilized CPU becoming a bottleneck of the system (“meltdown point” phenomenon).

$$C(U) = \begin{cases} \frac{(U - U_{CPU}^{upper})^2}{U_{CPU}^{upper}} & \text{if } U \geq U_{CPU}^{upper} \\ 0 & \text{if } U_{CPU}^{lower} \leq U \leq U_{CPU}^{upper} \\ \left(1 - \frac{U}{U_{CPU}^{lower}}\right)^2 & \text{if } U \leq U_{CPU}^{lower} \end{cases}$$

where $U$ is the measured value of average CPU utilization at any given interval.

- **QoS detriment cost ($\sum p_x D_{p_x}$).** The optimizer prefers a resource allocation that causes less QoS detriments over all physical hosts (Section 3.3). The proposed controller uses the system model introduced in Section 4.2 along with the prediction module (Section 4.3) to predict the future incoming event generation rate to find out the optimum value for $M^h$, $CPU^{(i)}_{p_x}$, and $RAM^{(i)}_{p_x}$ for every Lambda function such that the average waiting time of processing of events remains within the accepted range (i.e., incurs less QoS detriment penalty).

- **Interference cost ($\sum_{p_x} \sigma_{p_x}$).** To quantify the slowdown rate caused by consolidation of several Lambda functions in a single machine, we propose a simple yet effective solution based on the techniques introduced in [7], [25]. We simply monitor the utilization of memory bandwidth in each host, denoted by $MBW_{util}(p_x, t)$, to detect the negative impact of workloads’ contention on both LLC and the memory bus. MBW’s performance metric can be computed using perf, a Linux performance monitoring tool, by measuring the two standard hardware events of $UNC_{QMC\_NORMAL\_READS}$ as an indicator of memory reads and $UNC_{QMC\_WRITES}$ as an indicator of memory writes [25]. Therefore, the memory bandwidth consumed at each host can be used as a cost function to indicate the degree of its shared resource interference (as discussed in Section 3.2).

The objective function to be optimized at any sampling interval, $(\mathcal{J}_s)$, can be expressed as the weighted sum of the cost over a prediction horizon of future $s$ steps.

$$\min_{\mathcal{J}_s} = \sum_{p_x \in \mathcal{P}} (\gamma_1 C(U(p_x)) + \gamma_2 D_{p_x} + \gamma_3 \sigma_{p_x}) \quad (8)$$

We compute the norm of a normalized vector of all terms in Equation 8 whose components are the original values of the measured/estimated values of corresponding metrics, each divided by its maximum expected value. Each of coefficients $\gamma_{i=1,3}$ acts as the weight of the corresponding cost function. The coefficients need to be configured separately by the service provider. She can use various tools, such as experiments, simulation tools, efficient Frontier or Pareto Frontier analysis to determine such weights. For simplicity, we give equal weight for each cost function in this paper.

After solving the optimization problem for a prediction window of length $s$ and by considering the value of $T/T_{ref} > 2$ as the response speed, the controller only applies the first step of the optimal value to the system (see Section 4.1). Having omitted the negative effect of wrong
predictions or non-optimal decisions, this allows the output of the system smoothly converge on the desirable target (see [43]).

4.5 Search Space Reduction

We use a technique based on particle swarm optimization (PSO) heuristic to solve the optimization problem at each stage. PSO is a population based stochastic optimization technique as a fast evolutionary computational technique for solving optimization problems with multiple local extrema [44]. PSO can converge to the (near-) optimal results in a much faster and cheaper way comparing with other optimization methods [45].

We also adopt two other techniques to reduce the computational overheads associated with the exploring the exponential-size of feasible search space. Firstly, at each epoch $\tau$, we allow the optimizer to continue its execution to find a solution only for a fixed fraction (e.g., 1%) of the control step interval. For example, if $|\tau|$ is selected to be 1 minute, then the maximum time that the solver is allowed to find a solution is bounded to 600 milliseconds. Within that time limit, the best solution obtained by the PSO solver will be the input vector of the controller in the next step. Secondly, we allow the PSO solver to keep searching for a better solution until the set of the next step’s outputs is revealed. While this solution cannot be used for the system input at the current step, it is beneficial for being used as the next round starting point.

5 EXPERIMENTAL EVALUATION

The effectiveness of the proposed approach is carried out with respect to three aspects: resource utilization, QoS detrimen, and robustness. We assessed the proposed approach against an enhanced version of three greedy algorithms, namely enhanced spread, enhanced binpack and the best-effort approaches. The spread policy tries to assign Lambda functions uniformly on all accessible physical machines. On the other hand, the binpack policy assigns the maximum number of Lambda functions on a single physical machine till it becomes fully utilized, then looks for the next available machine. Binpack tries to use as few machines as possible for any given setting.

The best-effort essentially uses the first fit decreasing (FFD) algorithm [46] to find a set of machines that can support both anticipated resources usages and QoS enforcement of submitted Lambda functions. It allocates an additional machine for the remaining Lambda functions with high QoS level if the QoS violation value exceeds a certain threshold (10% in our study). The enhanced interference-aware version of each policy (including best-effort) averts collocating functions in any host that its memory bandwidth utilization is above the highest threshold level for $T_{MBW}$ (i.e., 90% in this study) whenever it is possible.

5.1 Methodology

We implemented a proof-of-concept prototype of the proposed controller on a distributed Apache OpenWhisk. We adopted the guidelines in [3], [4], [47] to setup the Lambda cluster. All experiments reported in the following section have been performed in a local cluster consisting of three nodes with total 24 logical cores and 48 GB of main memory. Each machine is installed with 16 GB of main memory and equipped with eight 2.40 GHz Intel Xeon E7-8870 cores.

The controller, which is developed in Python 2.7, uses a dedicated node equipped with Intel i7 2.3 GHz with 16GB of RAM and a SSD drive. Each experiment was running three times and the average of measurements is reported, as the results show a small standard deviation (less than 7% of the average value), we skip showing the error bars in the corresponding figures. The parallelism factor for each Lambda function is a variable to be set dynamically by the controller. We assume that each event source is only associated with one Lambda function; hence, if an event can cause a chain of functions to activate, it has to be modeled by adding a new event source as the output of the rest in the chain.

5.2 Workload Attributes

We created $|\Lambda| = \{100, 200, 400\}$ Lambda functions, each taken from PHP implementation of RUBiS benchmark, a well-known cloud web application that emulates the core functionality of an auction site: selling, browsing and bidding [48]. To generate the incoming requests of each Lambda function, we use httpmon package that generates HTTP requests to the associated REST API of an action. Furthermore, the httpmon package can periodically collect the application performance, such as 95th percentile of the end-to-end response time.

We consider the request arrival rate prescribed by the workload induced on the system as the only related attributes of the operating environment. The httpmon reads the time stamp for each request from a set of trace files, and replays the traces with the requests happening with their recorded interarrival time. The event generation rate is derived from either a Poisson or Weibull distribution with parameters of $\theta \in \{0.2, 0.5\}$ for Poisson, or $\alpha \in \{0.2\}$ and $\beta \in \{10\}$ for Weibull. In the Poisson process, $\theta^{-1}$ represents the average number of events per second. In the Weibull process, which occurs often in applications with heavy-tailed workload patterns, $\alpha$ and $\beta$ are the scale and shape parameters of a standard Weibull distribution, respectively.

The average number of events per second in such a process is derived by $1/\alpha\Gamma(\beta^{-1} + 1)$ ($\Gamma$: Gamma function). In all scenarios, we use 500 threads emulating 500 concurrent clients to generate workload for each Lambda function. The average execution time of each function to process an event is 100ms, and we allow each scenario to run for 1 hour.

In each scenario, there are three different QoS enforcement classes, i.e., $|Q| = 3$. In this way, the upper bound associated with each QoS class is fixed as $\omega_{q=1,3} \in \{200ms, 300ms, 600ms\}$ and $\gamma_{q=1,3} \in \{0.05, 0.85, 0.70\}$. We also assign a Lambda function $f_i$ to a QoS class $q_i$ if $i \equiv q$. We choose the sampling interval epoch (control interval), the history window ($h$), and the prediction horizon ($s$) used in each scenario to be $\tau = 1$ minute, $h = 2$, and $s = 2$, respectively. In total, we ran 9 different scenarios based on the different parameters for $|\Lambda|$ and generation rate. For
The plot in Figure 2 depicts the improvement in the average total processing time (i.e., system response time) that is experienced by each client when the proposed controller is used for prepossessing her requests. The x axis represents the number of Lambda functions in each scenario that is increased from 100 to 400. Compared to other three static allocation modes, the proposed controller is able to co-ordinate CPU resource allocations to all Lambda functions driving by their perceived response time. In addition, changing the arrival distribution of requests affects the quality of our solution for improving the average processing time in different scenarios. In fact, such an improvement is more significant in the Weibull distribution (a case for heavy-tailed workloads) when the proposed solution is employed.

For example, it automatically adjusts resources allocated to Lambda functions belong to the QoS with the highest priority such that their response time improves by a factor of almost three (from 582ms, achieved by the binpack approach, to 200ms). In average, the proposed solution can improve this metric by 14.9% (maximum 23.4%) compared to the best result which is achieved by enhanced best-effort strategy. The experimental results indicate that the proposed controller ensures that the performance of running collocated Lambda functions are not slowed down because of shortsighted consolidation decisions in the overloaded servers.

The proposed controller enhances the absolute value of CPU utilization by 18% and 4% in average for lightweight and heavyweight workloads, respectively, compared to the best outcome of other three heuristics (considering the ideal CPU utilization level).

Figure 3 depicts the amount of reduction in the overall QoS violation incidents achieved by applying our solution compared to the best outcome of other heuristics in different scenarios. The proposed controller can reduce the QoS violation incidents on average by 87% (maximum 146%) compared to the enhanced-spread heuristic, which uses all available resources and shows the best result with regard to this metric. Specifically, such an improvement is more significant on heavyweight workloads, e.g., Weibull case. The average improvement in this case is 108%.

**Computational running time.** The running time of the proposed controller to find a solution is a fixed value (0.6 second) as the controlling time-frame is one minute.
5.4 Sensitivity analysis

To find out the effect of errors in estimation procedure of different parameters (e.g., incoming rate), we conducted some experiments to measure the sensitiveness of the proposed controller to the accuracy of the prediction model. A promising solution must tolerate such errors by introducing some techniques to lessen the negative impacts of prediction errors on the quality of decision variables. We incorporate two mechanisms to mitigate the sensitiveness of our controller to such errors: (1) use $\epsilon_r$ as a noise in the ARIMA model to explicitly associate uncertainty with the prediction model, and (2) choose the response speed value strictly greater than one, i.e., $T/Tr_{ef} > 1$ (through our experiments we fixed it to 3), which forces the controller to gradually apply the inputs to the system in more than one step. Indeed, after applying the partial input, the whole process of getting feedback and re-optimizing occurs again in the subsequent step.

The sensitivity analysis of the proposed controller has been conducted as follows. We start with a prediction model with a zero error with regard to the rate of the incoming requests. Then we deliberately inject an error ranging from 10% to 100% to the prediction model. Finally, we collect some experiments to measure the sensitiveness of different parameters ($e$) on the quality of decision variables. We incorporate some techniques to lessen the negative impacts of prediction errors on the quality of decision variables. We define a new metric, called the sensitivity coefficient (denoted as $\psi$), to understand the solution quality with regard to a particular performance metric, denoted by $z$, varies if the estimation of a parameter (like $x$) has an error of $\epsilon$. Precisely, $\psi_{z,x} = |z(x) - z(x + \epsilon)| / |z(x)|$.

Figure 5 depicts the sensitivity coefficients for both the average response time and the average CPU utilization with respect to the error percentage injected to the event generation rate. The results show that even an error of 100% in the prediction model has little influence in the solution quality (up to 31% for response time and 18% for CPU utilization). Because of employing the two above-mentioned methods, we can conclude that the proposed MPC controller is not too sensitive to the accuracy of prediction model.

Figure 6 represents the accuracy of the response time model (proposed in Section 4.2) compared to the actual response time in different scenarios. The proposed model can predict the future response times in average by 70% across all scenarios.

6 RELATED WORK

AWS Lambda [1], IBM OpenWhisk [3], Google Cloud Functions [2], and Microsoft Azure [5] are the most popular platforms offering scalable serverless solutions with the main aim of easing the deployment burden. Employing such a technology, enterprises can focus more on the development of the business logic instead of handling all sorts of problems related to the infrastructure maintenance. The core philosophy in the new architecture is based on using an event-driven approach to build complex softwares. Such an approach not only simplifies the process of development, but also makes it easier to design an application logic, especially when the business logic becomes progressively more complex.

Some recent studies have indicated that the lambda architecture has a great potential to be leveraged in different domains. Authors in [49] presented an implementation of a lambda architecture to create a data-handling platform and batch data processing on Amazon EC2. Their solutions can offer a high throughput system for processing intense data demand delivered as services with the main objective of minimizing the cost of the network maintenance or near-real time processing, respectively. Authors in [50] proposed a model for SLA-based resource allocation and Map Reduce tasks scheduling for Big Data processing in the batch layer of Lambda architecture as a cloud platform. We believe that the given methods are still in their very early stages, and need to be improved in several dimensions.

Devising an elastic resource allocation for a Lambda platform is a recent research field. The main aim is to provide a mechanism to scale resources up or down on demand when the rate of events/requests intensively fluctuates. This problem has been well investigated in traditional distributed platforms, such as [51], [52], [53], [54], [55], or cloud environment, e.g., [56], [57], [58], [59], [60], [61], [62], in which authors use different methods, e.g., threshold-based rules over the CPU utilization, to decide when to add/remove computing resources.

Many existing resource management strategies, e.g., [63], [64], [65], [66], [67], [68], manage resources based on OS level metrics, such as per core utilization, I/O capacities, and energy usage of resources while ignore the negative performance caused by interference at the shared resources,
e.g., LLC or memory bandwidth. However, a careful study by [25] confirmed that any resource management schema that is unaware about the interference of shared resources is entirely a failure. Such a mechanism is necessary to avoid the performance degradation problem caused by consolidation decision among collocated workloads.

The work in [23] attempt to consider the microarchitecture-level interference by using an offline profiling phase to captures the interference attributes of applications. However, obtaining an interference signature through profiling might not be feasible in all cases. The interference attributes of applications could change over the run-time, too. Rao et al. proposed an effective metric to predict the performance of applications running in a NUMA system [69]. Such a metric can be leveraged to design a resource allocation that is aware of contention among shared resources. Authors in [70], [71], [72] proposed a method to reduce the negative impact of architecture-level shared resource contention on a hypervisor-based cloud platform. However, it seems that these projects concern about tuning resources on a single node, while our focus is to devise a resource allocation in a cluster of hosts that parallelization of each application (i.e., Lambda functions) can be greater than one.

Using predictive model controller is not new in distributed and parallel systems, e.g., [73], [74], [75]. Padala et al. [51] present a multi-input, multi-output (MIMO) resource controller that automatically adapts to dynamic changes in a shared virtualized infrastructure to achieve application SLOs. Their model estimates the complex relationship between application performance and resource allocation, and a MIMO controller allocates the right amount of resources. Such a system adjusts its model by measuring the clients’ response time using an embedded performance sensor within each application.

On the other hand, our proposed controller responds to the degraded performance level in the application layer by measuring the number of outstanding unprocessed events. The system model used by AppController is an ARMA estimator that automatically develops a real-time model for capturing the relationship between application’s resource allocation and its performance metrics, while we use a more accurate queuing based formula to estimate the response time of each application. We simply use an ARIMA estimator (as a less accurate tool) for predicting the future rate of incoming events per Lambda function.

7 Conclusion

Understanding the run-time attributes of workloads can be of great practical importance to design a well-utilized resource allocation strategy for a Lambda platform. We have devised a contention aware solution based on model predictive controller for achieving the following goals: (1) well-utilization of computing resources, (2) reducing average response times of processing events, and (3) satisfying the QoS demand levels of each Lambda function.

The effectiveness of the proposed solution has demonstrated its efficacy with an average improvement of 18% in overall CPU utilization (for lightweight workloads) and an average 14.9% (maximum 23.4%) reduction in response time compared to the best result that is achieved by three heuristics of spread, binpack and best-effort in different scenarios. The proposed solution also reduces the QoS violation incidents on average by 87% (maximum 146%) compared to the spread heuristic that uses all available resources in the cluster farm.

As a future direction, we plan to extend our solution as a distributed model predictive control as a strategy in which the control problem is decomposed into several sub-problems solved by a set of autonomous nodes in a distributed manner, as suggested by [75], [76], [77]. In addition, we consider the possibility of change the horizon length as a function of the prediction error, similar to work in [75].

Acknowledgments

This work is carried out in the context of the grant LP# 160100406. We would like to thank the ARC (Australian Research Council) for their support.

References


M.Reza HoseinyFarahabady received the BSc degree in computer engineering and the MSc degree in information technology and network engineering both from Sharif University of Technology, Tehran, Iran, in 2005 and 2007, respectively. He received the PhD degree from the School of Information Technologies at the University of Sydney, in 2015. He is currently a research associate in the Centre for Distributed and High Performance Computing, School of Information Technologies at the University of Sydney. His current research interests include scheduling and resource allocation for parallel and distributed computing systems, and control systems engineering.

Albert Y. Zomaya is currently the Chair Professor of High Performance Computing and Networking and Australian Research Council Professorial Fellow in the School of Information Technologies, The University of Sydney. He is also the Director of the Centre for Distributed and High Performance Computing which was established in late 2009. He is the author/co-author of six books and he has edited over 25 conference proceedings. Zomaya is a recipient of over 6M $ in funding from ARC (Australian Research Council) and lately part of a successful 7th Framework AU2EU (Australia to European) bid on Authorisation and Authentication for Entrusted Unions. Zomaya was an associate editor of the IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS) and IEEE Magazine on Cloud Computing.

Zahir Tari is a full professor in Distributed Systems at RMIT University (Australia). He received a bachelor degree in Mathematics from University of Algiers (USTHB, Algeria) in 1984, MSc in Operational Research from University of Grenoble (France) in 1985 and PhD degree in Computer Science from University of Grenoble (France) in 1989. From 1990-1992, Zahir worked at the Database Laboratory at EPFL (Swiss Federal Institute of Technology) as a senior researcher, where he looked at various aspects of systems. In 1996, he moved to RMIT University as a senior lecturer and currently Professor, where he led the DSN (Distributed Systems and Networking) discipline. Zahir’s expertise is in the areas of system performance (e.g., Web servers, P2P, Cloud) and system security (e.g., SCADA, Smart Grid, Cloud). He is the co-author of six books and he has edited over 25 conference proceedings. Zahir is a recipient of over 6M $ in funding from ARC (Australian Research Council) and lately part of a successful 7th Framework AU2EU (Australia to European) bid on Authorisation and Authentication for Entrusted Unions. Zahir was an associate editor of the IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS) and IEEE Magazine on Cloud Computing.