Sequence-to-sequence model for workload interference prediction on batch processing datacenters

David Buchaca Prats, Joan Marcual, Josep Lluís Berral and David Carrera

Barcelona Supercomputing Center (BSC), Universitat Politècnica de Catalunya (UPC)

{david.buchaca, josep.berral, david.carrera}@bsc.es

Abstract—Co-scheduling of jobs in data-centers is a challenging scenario, where jobs can compete for resources yielding to severe slowdowns or failed executions. Efficient job placement on environments where resources are shared requires awareness on how jobs interfere during execution, to go far beyond ineffective resource overbooking techniques. Current techniques, most of them already involving machine learning and job modeling, are based on workload behavior summarization across time, instead of focusing on effective job requirements at each instant of the execution. In this work we propose a methodology for modeling co-scheduling of jobs on data-centers, based on their behavior towards resources and execution time, using sequence-to-sequence models based on recurrent neural networks. The goal is to forecast co-executed jobs footprint on resources along their execution time, from the profile shown by the individual jobs, to enhance resource managers and schedulers placement decisions. The methods here presented are validated using High Performance Computing benchmarks based on different frameworks (like Hadoop and Spark) and applications (CPU bound, IO bound, machine learning, SQL queries...). Experiments show that the model can correctly identify the resource usage trends from previously seen and even unseen co-scheduled jobs.

Index Terms—Job interference prediction, Job placement, Machine Learning, Deep learning, recurrent neural networks.

I. INTRODUCTION

Resource under-utilization is one of the major problems in data center management, since the average utilization is estimated to be below 50% [1][2] due to over-conservative scheduling policies, to ensure good Quality of Service (QoS) levels. However, most applications do not use resources continuously, even when there is full quota on CPU, memory and I/O to avoid degrading the QoS or violating Service level Agreements (SLAs). Flexible policies allow resource sharing among applications, while at the same time risking concurrent applications to top their requirements. Sharing resources between applications in data centers is crucial to achieve efficient utilization of those resources, reducing power consumption, allowing proper scaling out for currently running applications or accepting new applications into the system.

The principal problem when co-locating resource-sharing applications is to ensure that competition will not ruin their QoS, even when a certain tolerable degradation is expected. Co-located applications do not need to behave as "the sum of their behaviors", since interference creates a new characteristic footprint for each set of concurrent applications. Applications can be profiled in isolation in order to characterize the requirements claimed during their execution, often regarded as "black boxes" when the customer (owner of the application) does not grant access inside the application container or Virtual Machine. However, profiling pairs of applications in order to discover their resulting footprint becomes unfeasible, since the number of pairs grows with the number of applications in the system to be tested, which makes matters even worse as the group of co-located applications grow. Thus, there is the need to predict application resource demands and interference without the burden of running all possible combinations of applications.

The statistical analysis of monitored metrics is fundamental when automating this workload placement. Since resources are finite, smart resource sharing is encouraged by administrators in order to increase resource availability while maintaining energy efficiency[3]. Different approaches exist for predicting resource demand and interference (slow-down produced by resource competition) on co-located applications. Nevertheless, most methods are based on predicting overall interference. When applications show differentiated phases of behavior over time, aggregation cannot capture the true nature of resource demand, thereby giving rise to suboptimal scheduling decisions. Most classic machine learning approaches for this problem, such as [4] or [5], do not consider the temporal dimension of executions, and therefore do not predict resource usage over time. This means that schedulers are obliged to optimize fixed blocks of expected resource usage. In this scenario, schedulers can take suboptimal decisions to block possible colocations of applications that only complete for a fraction of their execution.

In this work we present ResourceNet, a workload-to-workload forecasting methodology to predict the effects of application co-location interference, using sequence-to-sequence models based on Recurrent Neural Networks (RNNs). RNNs are powerful models with the capability of dealing with time series. Of equal importance, RNNs are able to deal with inputs and outputs of arbitrary lengths. The method presented herein predicts the footprint for resource demands of co-located applications, given that the traces of these applications run in isolation. Furthermore, we show how this model can be used to predict accurately the run-time of applications sharing resources.

Our model employs two Gated Recurrent Units (*GRU*) as

building blocks; one *GRU* processes the trace signal of the incoming applications and passes the processed information to the other *GRU*, which outputs the expected resources of the co-located applications over time. The model predicts the whole resource demand trace throughout execution, thereby providing schedulers with a sufficiently accurate estimation for placing applications together and thus minimizing interference. Training the model using a diverse set of benchmarking applications enables it to attempt predictions to unseen co-located applications. The recurrent nature of the model allows it to process input sequences or different (and arbitrary) lengths. Moreover, it is able to generate output sequences of arbitrary length, thereby making our solution adaptable enough to address the problem in question.

We validate the proposed methodology by computing the error of the predicted resources (of co-joined application traces) with respect to the real resources. We use benchmarks from Big Data applications (from both Apache Hadoop and Apache Spark) as workload data. We have selected Hadoop and Spark benchmarking workloads because of their popularity in High Performance Computing (HPC) applications, where scheduling and environment configuration have a high impact on the application performance. Experiments measure the prediction of the low-level resource usage (i.e. CPU, memory and I/O) of pairs of co-located workloads over time. The different benchmarks we use belong to the Intel HiBench [6], the IBM SparkBench [7] and the Databricks SparkPerf benchmarks [8]. The method is trained and evaluated on low-level monitoring metrics which are reasonably available on any HPC setting. Our method is compared against different simpler machine learning alternatives in order to assess the possible drawbacks of using standard models.

For our experiments, we created a dataset with execution traces from the previously mentioned benchmark suites. The dataset consists of triplets $(a, b, a \wedge b)$ where a and b contain the traces of the isolated executions from two applications, and $a \wedge b$ represents an execution of the co-located pair. To build a reasonable dataset, we generated scenarios in which both appli- cations did not start at the same time. In particular, we created co-scheduling situations in which the start time for one of the applications was shifted by a factor of 25%, 50% and 75% of the length of the longest application being co-scheduled.

Contributions

The main contributions of this paper can be summarized as follows:

- A novel application of Recurrent Neural Networks that translates the characterization of two applications a and b in isolation, into the predicted trace of the co-execution of such applications $a \wedge b$.
- A novel feature, *percentage completion time*, for estimating the completion time of co-scheduled applications. This feature improves predictions made by using the standard stopping criteria based on the *end of sequence* feature.
- A comprehensive evaluation of the method against other relevant machine learning approaches. We show the ad-

vantages of our method, which are especially noticeable in two cases: when co-located executions have different lengths, and when co-scheduling heavily impacts the execution time of the applications due to high interference.

The proposed method can be used to characterize applications according to their compatibility with other workloads. Furthermore, it enables resource managers to plan resource allocation and load balancing better in advance.

The rest of the paper is structured as follows: Section III provides background on the commonly used recurrent neural network methods. Section IV presents the methodology, scenario and used techniques. Section V provides a description of the data used for experimentation. Section VI describes the experiments conducted to validate this work. Section II describes Related Work, and finally Section VII discusses the conclusions and future work.

II. RELATED WORK

This section presents related work that is relevant for resource, runtime and interference prediction. The three topics are closely related and still an active field of research with many different applications. Works that target those topics usually take into account more than one topic since they are closely related. Two relevant applications are dynamic provisioning and job placement. In dynamic provisioning scenarios the main goal is to predict whether there will be a change in the workload trace that needs some hardware decision to take place (like pro- visioning more nodes for a workload). In the case of job place- ment, resource predictions also play an important role, since similar resource needs for different jobs resources may affect. Thus, schedulers may aim for colocations of applications that do not degrade when trying to maximize the use of available hardware.

Resource prediction related work

The main goal of oracles in workload provisioning scenarios is to guess accurately when workloads will need more resources in the future. Works like [9] and [10] use neural networks that take as input resource usage in a given time window the aim of predicting the future resource requirements for the workloads. In [10] authors use differential evolution as a means to train the models, whereas in [9] they use standard gradient based algorithms.

Works like Sanz-Marco et al. [11] apply machine learning to forecast memory requirements towards interference avoidance. They proposed a mixture of expert approach to predict the amount of memory needed by a SPARK application, given the size of the input data. From a dataset consisting of applications and SPARK profiling features, captured from hardware counters (CPI, number of interruptions, number of cache misses, etc). Then, after using Principal Component Analysis to reduce dimensionality, the mixture of experts each learn regression function to predict the required memory using the size of input data as input. The application is first profiled with a small input 10%, and it then passes through a k-Nearest Neighbor in order to choose the most representative expert. Finally, the application is run with different data sizes in order

to tune the function parameters to determine the best fit for this application.

Interference focus-related work

In [12] the authors present Paragon, an approach to predicting the interference between two applications by modeling them using SVD and PQ-reconstruction [13]. In Paragon, the profiling across pairs is limited to the first minute due to time constraints, without covering applications with different execution phases and behaviors over time, although they acknowledge the problem by trying to mitigate it by labeling nodes with unexpected changes in resource demands, then leave them running until jobs are finished.

Heracles et al. [14] present an approach for interference mitigation in high priority applications through job isolation techniques. Among other tools, their solution uses *cpugroups*, *qDisc* or the *CAT* technology [15] to properly ensure resource availability for specific jobs. Such a solution tends to over-resource applications for solving the interference problem, thus becoming subject to machine under-utilization. Furthermore, the optimization method poses an NP-hard solution, which makes it feasible for only a small number of jobs.

In a similar line, Mishra et al. [5] presented ESP, a predictive model for forecasting the interference between two applications. ESP uses low level metrics as input for the model, but instead of making predictions at sequence level they are held at a scalar level, both at the input and the output, by means of aggregation metrics such as the average of the sequence. Thus, given the average of two metrics, it predicts the average of the concurrent execution in a two-step model as well; one for feature selection and the other for using a regression model to retrieve the expected aggregated metric as output when both tasks are co-located.

Another interesting work dealing with protection against interference is Stay-Away by [16]. The idea they propose starts by projecting the resource consumption of the applications in a 2-dimensional space. They then predict how applications will move in that 2D space. If applications are predicted to have collisions between applications in that space, they then adopt a counteraction.

Run-time prediction-related work

In [17] the authors present a convolutional neural network named PRIONN. The model predicts run-time and IO (bytes read and total bytes written) of applications based on the source code in the input script. This work maps job scripts code into image-like representations so there is no need for a feature generation process to process the raw text of the scripts.

In [18] Aloja-ML is presented as a framework for characterization and knowledge discovery in Hadoop deployments. The authors present different methods for execution time prediction based on hardware resources, workload type and input data size.

In [19] a mixture of runtime prediction and interference is presented in two methods: The first, aimed at calculating the cost of a Spark job run in isolation, is approximated by modeling a function from the execution parameters (the number of data partitions, the number of stages, the number of jobs per stage, etc.) to predict the total cost of the job. The second, designed to predict the cost when two jobs run in concurrency by modeling interference, is tackled by adapting the previous formulas. They run a small part of the input data for each Spark stage combination to compute a ration of interference β, to be added to the modeling function as a factor of the addition of required resources as $resource_i(AB) = \beta \cdot (resource_i(A) + resource_i(B))$. This work recognizes the different behavior across the execution caused by the different phases of executions, but is limited to predefined Spark phases that have to be given by the user.

III. BACKGROUND ON RNNS

In this section we present an overview of the fundamentals on Recurrent Neural Networks (RNNs) and some important enhancements on the standard models. We also provide the building blocks of our model. For the rest of this work, we denote x_t a vector of d components (containing monitored metrics) at time step t. A sequence of n vectors are denoted as $x_{1:n}$. The k'th coordinate from a vector x_t is denoted as $x_t[k]$.

A. Recurrent Neural Networks

A Recurrent Neural Network is a function that maps a sequence of input vectors into an output vector. Specifically, the RNN takes a sequence $x_{1:t}$, where each $x_j \in \mathbb{R}^{d_{in}}$, and returns a single output vector $y_t \in \mathbb{R}^{d_{out}}$, depending on a set of parameters θ that need to be learned. Equation (1a) denotes the output of the RNN at time t, while equation (1b) denotes the whole sequence of outputs $y_{1:n}$ produced by applying (1a) at every time step and concatenating the output results.

$$y_t = \text{RNN}(x_{1:t}; \theta) \tag{1a}$$
$$y_{1:n} = \text{RNN}^*(x_{1:n}; \theta) \tag{1b}$$

Output y_t is used to predict relevant information about the problem at hand at time t. Since we use the RNN to predict the expected resource demand of two concurrent applications at time t, y_t will be a vector containing the resources used by the two co-located applications.

In order to keep information from previously seen elements in sequences, RNNs have a hidden state s_t that is updated at every time step. The hidden state influences the predictions of the model over time. Although the hidden state is implicit in the equations (1a) and (1b), we can make the state explicit by using the notation $(y_t, s_t) = \text{RNN}(x_t, s_{t-1}; \theta)$. This indicates that the output y_t has been computed through input x_t and previous state s_{t-1}. Figure 1 shows a diagram of an RNN.

Notice that while y_t (the output at time t) is not affecting y_{t+1}, the updated state s_t will affect next state s_{t+1}. The initial state s_0 is usually defined as a vector of zeros, unless some other known contextual information can be provided for the model. For example, in *sequence-to-sequence* models (described in Section III-C) the initial hidden state of the decoder plays the crucial role of providing a summary of
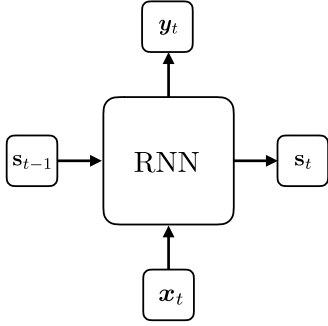
Fig. 1. The computation of y_t and s_t depends on s_{t-1} and x_t.

the input sequence to the decoder. The way in which s_t is computed is what mostly differentiates RNN models, such as Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM). Independently of how the hidden state is computed, RNNs are usually trained by using optimization algorithms based on Stochastic Gradient Descent (SGD) in order to minimize the difference between the predicted outputs and the expected outputs.

B. Gated Recurrent Unit

Gated Recurrent Units (GRU) are a type of recurrent neural network designed to avoid gradient vanishing problems [20]. The GRU is one of the simplest RNNs employing a gating mechanism that enables the network to learn which information needs to be kept over time and which can be forgotten. Equations (2a) to (2d) define, at time t, the output state s_t, given the input data x_t and the previous state s_{t-1}. In the presented equations, σ is the sigmoid function.

$$z_t = \sigma\left(W^{xz}x_t + W^{sz}s_{t-1} + b^z\right) \tag{2a}$$

$$r_t = \sigma\left(W^{xr}x_t + W^{sr}s_{t-1} + b^r\right) \tag{2b}$$

$$\tilde{s}_t = \tanh\left(W^{xs}x_t + W^{ss}(r_t \odot s_{t-1}) + b^s\right) \tag{2c}$$

$$s_t = \tilde{s}_t \odot z_t + (1 - z_t) \odot s_{t-1} \tag{2d}$$

The output state s_t is controlled by two vectors: the update gate z_t and the reset gate r_t defined in equations (2a) and (2b) respectively. Since both vectors come from applying a sigmoid function, most of their components will be close to 0 or 1, controlling the information passed by any element-wise multiplied vector. In extreme cases, when the gate components are 0 or 1, the flow of information can be completely blocked or remain intact. For example, if the reset gate r_t is zero and the update z_t is one, the previous hidden state s_{t-1} does not affect the next state s_t.

Once the reset and update gates are computed, a proposed hidden state $\tilde{s}_t$ is created. Equation (2c) defines the proposed hidden state $\tilde{s}_t$ at time t. Notice that $\tilde{s}_t$ depends on the input vector and the information from the past hidden state s_{t-1} that is allowed to pass by the reset gate r_t. Afterwards the true state s_t is computed as a combination of components from the proposed state $\tilde{s}_t$ and the previous state s_{t-1}. Notice that in equation (2d) the element-wise multiplications $\tilde{s}_t \odot z_t$ and $(1 - z_t) \odot s_{t-1}$ can control which components of the proposed

state and the previous state are kept. The matrices W^{**} and vectors b^* are the weights adjusted during learning.

C. Sequence-to-sequence models

Most neural network architectures prior to [21] were designed for problems where the goal is to map input vectors x to fixed size output vectors y. Sequence-to-sequence models proposed an end-to-end trainable neural network architecture designed to tackle sequential data. The architecture is based on two Recurrent Neural Networks named *encoder* and *decoder*. The encoder is designed to read the input sequence and produce a vector that is passed to the decoder. Since the hidden state of the encoder is updated by reading all the information from the input, it can be interpreted as a summary of the input sequence. The decoder reads the encoded vector and produces a new sequence, but instead of initializing the hidden state of the decoder with zeros, it is initialized to the last hidden state of the encoder RNN. These types of models have been successfully applied to different tasks ranging from machine translation, e.g [21][22], to image captioning [23].

D. Attention mechanism

The main drawback of the *sequence-to-sequence* approach explained in Section III-C is that the whole input sequence is compressed into a single vector. Since the amount of parameters and the computational cost of running RNNs grows with the number of neurons in the hidden state, it is not feasible to use hidden states sizes of the order of $n \cdot d$, where n is the number of elements in the sequence and d the number of features. Storing all the information of a sequence into a single vector of fixed size is therefore a difficult task.

The work done by Bahdanau et al. [22] improves the architecture of the previously mentioned sequence-to-sequence model by furnishing the decoder with an *attention mechanism*. At every step of the decoding process, this mechanism provides a vector containing information about what *might* be relevant from the input sequence. Therefore, the decoding process is not performed from a single vector as in [21], but from a vector generated at every time step. This vector generated by the *attention mechanism* is called the context vector. In [22] authors showed that the performance of a sequentence-to-sequence model with attention did not degrade for sequences of various lengths, whereas the same model without attention gave far worse results for long sequences.

The decoding process for an RNN with attention works as follows: First, the encoder reads the input sequence and generates the matrix $E = \mathrm{RNN}^*_{\text{enc}}(x_{1:n};\theta)$. Then, at time t, the decoder $\mathrm{RNN}_{\text{dec}}$ receives as input (in addition to any recurrent inputs) the concatenation of two vectors: $[\tilde{y}_{t-1}; c_t]$. This concatenation provides extra information at the decoder at every time step. The *context* vector presented in [22] is defined by equations (3a) to (3c).

$$\mu_t[k] = v^T \tanh(W^\mu E[\cdot, k] + V^\mu s_{t-1} + b^\mu) \tag{3a}$$

$$\alpha_t = \text{softmax}(\mu_t) \tag{3b}$$

$$c_t = E\alpha_t \tag{3c}$$

In (3c) the *context* vector is computed as a matrix vector product, which can be interpreted as a weighted sum of the columns of E. The weights given to the columns of E are the values of the *attention* vector α_t. The vector α_t from (3b) is called the *attention* vector because it contains the weights used to "focus" (or attend) to different parts of E. This vector is computed by normalizing (with a softmax function) the *attention* energy μ_t. The *attention* energy is computed in (3a) by using a single Multi Layer Perceptron (MLP), with a single tanh layer. The MLP predicts a single scalar which defines the k'th coordinate from μ_t. The MLP is applied at every position k from 1 to n, where n is the number of columns of E, making $\mu_t, \alpha_t \in \mathbb{R}^{n \times 1}$. Notice that the product of $W^\mu E[\cdot, k]$ in (3a) does not depend on the time step t and therefore it can be precomputed in advance.

One of the key properties of the presented *attention* mechanism is the ability to construct a fixed-size vector c_t of dimension d which does not depend on the length of the input sequence. The vector μ_t has variable size because, for a given sequence of arbitrary length n, vector μ_t is computed by applying (3a) for $k \in \{1, \ldots, n\}$. Nevertheless, the product $E\alpha_t$ is a vector of size d, where d is the number of rows in E because $E \in \mathbb{R}^{d \times n}$ and therefore $c_t = E\alpha_t \in \mathbb{R}^{d \times 1}$.

In many applications, E is generated using a bidirectional RNN which simply concatenates the hidden states of two RNNs. One RNN_f receives the input sequence forwards (from x_1 to x_n) while the other RNN_b receives the sequence backwards (from x_n to x_1). In [22] E is created by stacking $\text{RNN}_f^*(x_{1:n})$ on top of $\text{RNN}_b^*(x_{n:1})$.

IV. RESOURCENET

ResourceNet is a fully learnable system based on a sequence-to-sequence architecture with an attention mechanism. The model takes as input pairs of sequences containing measurements of applications run on isolation. The input sequences are aggregated and fed to a decoder which produces the expected measurements of running both co-located applications sharing resources. Since the model is based on a sequence-to-sequence architecture it can naturally work with sequences of different lengths, which makes it suitable for the scenario presented here, in which measurements belong to the workload traces.

Figure 2 shows the diagram for the model, indicating the inputs and outputs, and the four building blogs composing the data pipeline. The entire shaded box represents the whole model at a high level. The first component, $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$, performs the joining operation where input sequences a and b are stacked to form a single matrix of measurements; this process is explained in detail in Subsection IV-A. The second component, RNN_{enc}, encodes the input sequences with a GRU and produces as output a matrix E. The third component, E, represents the stored encoded input sequences. The fourth component, A is the attention mechanism, which receives the hidden state from the decoder and generates a context vector that is fed to RNN_{dec}. The last component of the diagram is the decoder. The decoder is a GRU that generates as output the expected resource demands of the co-scheduled applications. We denote those resource predictions by $a \wedge b$.
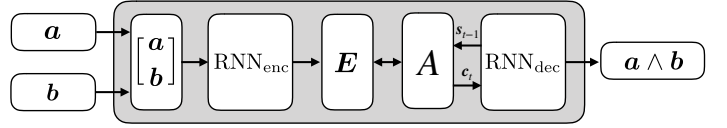

Fig. 2. Input-output diagram of the proposed model

The learning process is performed by minimizing the Mean Squared Error loss function, which computes the error between the predicted outputs and the true trace values at every time step. We change the parameters of the model using a gradient descent procedure.

A. Stacking measurements of the input sequences

The input sequences of the model are stacked in order to generate a single matrix of measurements. Given two vectors $v = (v_1, \ldots, v_p)^\top$ and $w = (w_1, \ldots, w_q)^\top$ we will denote by $\left[\begin{smallmatrix} v \\ w \end{smallmatrix}\right]$ or $[v; w]$ the vector $(v_1, \ldots, v_p, w_1, \ldots, w_q)^\top$.

Given two arbitrary sequences a and b of length $\text{l}(a)$ and $\text{l}(b)$ respectively we denote by $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ or $[a; b]$ a sequence of length $N = \max(\text{l}(a), \text{l}(b))$ containing the measurements of both sequences at every time step. By convention we write the longest sequence first. Since sequences may differ in length we append the shortest sequence with zeros to match the length of the longest sequence. If we denote by $\mathbf{0}_d$ a vector containing d zeros where d is the number of elements in b_k we have:

$$\begin{bmatrix} a \\ b \end{bmatrix} := \left(\begin{bmatrix} a_1 \\ b_1 \end{bmatrix}, \ldots, \begin{bmatrix} a_{\text{l}(b)} \\ b_{\text{l}(b)} \end{bmatrix}, \begin{bmatrix} a_{\text{l}(b)+1} \\ \mathbf{0}_d \end{bmatrix}, \ldots, \begin{bmatrix} a_{\text{l}(a)} \\ \mathbf{0}_d \end{bmatrix} \right)$$

Example: Let us consider $a = (a_1, a_2, a_3, a_4)$ and $b = (b_1, b_2)$ containing measurements on $\mathbb{R}^2$. That is $a_i, b_j \in \mathbb{R}^2$ for any valid i, j. Then the generated matrix for two applications that start at the same time is $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] := \left(\begin{bmatrix} a_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} a_2 \\ b_2 \end{bmatrix}, \begin{bmatrix} a_2 \\ \mathbf{0}_2 \end{bmatrix}, \begin{bmatrix} a_4 \\ \mathbf{0}_2 \end{bmatrix} \right)$.

The notation $[a, b]$ is reasonable for describing the situation in which the model is fed with traces of programs that start execution at the same time. Nevertheless, we may want to predict scenarios where applications do not start at the same time. In order to provide the network with such information, we pad a sequence with zeros before it starts. If we wish to tell ResourceNet that application b starts k time-steps after a, we add k vectors containing zeros at the beginning of b. In order to make ResourceNet capable of making predictions in that environment, where applications are not required to start at the same time, our training data will contain co-scheduled applications starting with different time delays.

B. Encoding input traces, Decoding co-scheduled trace

The encoding process reads the traces from the isolated runs and generates the matrix $E = \text{RNN}_{\text{enc}}^*(x_{1:n}; \theta)$, using the equations of the GRU from Subsection III-B. To this end, we use a GRU as our encoder.

The decoding process takes the produced E as input and generates the output sequence one vector at a time. This output

vector has as many components as it does features we wish to predict (in our case, 9), plus some features used to decide when applications finish. A detailed description of the features can be found in Section V. The decoder RNN_{dec} receives as input at time t the vector $x_t := [\tilde{y}_{t-1}; c_t]$ which is the concatenation of $\tilde{y}_{t-1}$ and c_t (in addition to any recurrent inputs). Vector $\tilde{y}_{t-1}$ is the output predicted by the decoder at the previous time step. Vector c_t is the "context vector" generated by the attention mechanism of the decoder. Given a hidden state s_t and a input x_t the predicted resources of the colocated applications at time t is $\tilde{y}_t = RNN_{dec}(x_t, s_t; \theta)$.

C. Predicting completion time

We have experimented with two mechanisms for predicting the completion time of the co-scheduled jobs. The standard End of Sequence (EOS) feature approach and our own Percentage Completion (PC) feature approach.

End of Sequence feature

The first strategy, based on an End of Sequence feature, is the standard approach used to decide when an RNN should stop producing more vectors. This feature vector is takes value 0 at every time step except the last one, where it takes value 1. This vector simply tells the decoder when both applications finish. If our decoder can predict EOS with reasonable precision then we can build stopping criteria based on those values to predict the completion time of the co-scheduled applications.

The presented End of Sequence approach for our problem is not equivalent to the analogous problem in other domains such as Neural Machine Translation (NMT). In NMT, the decoder emits conditional probability over the vocabulary at every time step which can be used to select the word with the highest probability. In that scenario, EOS is just a special word that stops the decoding process, which means that if the decoder emits the "EOS" symbol the decoding process is stopped. In our setting the EOS is a new feature that takes a real value at every time step.

Percentage Completion Features

The second strategy consists of a novel approach based on two additional features (one per job) which we call Percentage Completion (PC) features. PC features keep track of the percentage completion of the workloads, providing ResourceNet with extra information relevant to the job estimation runtime.

We denote by PCF_a and PCF_b the percentage completion features for input sequences a and b, respectively. Both features contain at time t how much of the workload has been completed until t, expressed as a percentage. For a given sequence s of length N, we define $PCF_s = \left(\frac{1}{N} \cdot 100, \frac{2}{N} \cdot 100, \ldots, \frac{N}{N} \cdot 100\right)$. Notice that, by construction, PCF features contain monotonically increasing values that must finish with value 100. Nevertheless the rate of increment at every time step will depend on the overall number of time steps of the sequence.

Example: Let us consider $a = (a_1, a_2, \ldots, a_{100})$ and $b = (b_1, b_2, \ldots, b_{300})$. Then $PCF_a = (1, 2, \ldots, 100)$ and $PCF_b = (0.333, 0.666, 1.0, \ldots, 100)$ are two vectors of length 100 and 300 respectively.

V. WORKLOAD DATA

A. Monitoring running applications

In order to capture the trace of each execution we have profiled them by using the basic Linux performance analysis tools: *vmstat*, *iostat*, *ifstat* and *perf*. These tools gather system performance metrics of running processes in a non-invasive way. Additionally, these tools have a lower performance impact in the system, causing negligible overhead to the executions. From these tools we have gathered a total of 141 features with time granularity of one second.

For our study we have selected 9 key features, shown in Table I, that are especially relevant for interference prediction and resource estimation [24][4][11].

Feature	Description
CPU	Percentage of total CPU used
RAM	Amount of Bytes of main memory used
IOR	Blocks received from a block device (blocks/s)
IOW	Blocks send to a block device (blocks/s)
CPI	Cycles per instruction
LLCM	Last level cache misses
FLCM	First level cache misses
PF	Page faults
TLBM	Translation lookaside buffer misses

TABLE I
WORKLOAD METRICS RECORDED AT EACH TIME STEP.

The dataset used in the experiments contains traces generated by a variety of micro-benchmarks (workloads). The workloads used have been extracted from different suites: HiBench [6], Spark-perf [8] and SparkBench [7]. These suites contain different Big Data workloads, and have also been used in similar studies [25][26][27][28]. The benchmarks include machine learning, data mining and big data benchmarks. Some examples of workloads are executions of *PageRank*, *K-means*, *Naive Bayes*, *Logistic Regression*, etc. The traces are executed using a server with two Intel Xeon E5-2630 processors and 128 GB of RAM, up to 400 isolated and co-located executions.

B. Dataset description

The dataset is composed of workloads *triplets*. Each triplet contains a combination of three execution traces. The first two traces correspond to isolated executions of the two applications. The third trace contains the execution of the co-located application from the first two traces. Therefore, the data has the form $D = \left\{(w^{[i]}, w^{[j]}, w^{[i]} \wedge w^{[j]}) \mid i, j \in I\right\}$ where I is a set of indices of the workloads.

In real world scenarios, co-located applications do not necesarilly start at the same time. In order to increase the co-location cases in our collected dataset, we prepared different scenarios where co-located applications a and b start with different delays. For the benchmarking executions, one of the concurrent applications is delayed to start after its co-located

peer application. The phase differences used in the dataset generation are 0 (synchronized), 0.25, 0.5 and 0.75. A phase difference of 1.0 is not taken into consideration since it would mean applications running completely in isolation, one after the other. Notice that bigger differences usually give rise to less interference, since applications have fewer runtime sharing resources.

VI. Experiments

ResourceNet is validated through a test set of executions which are used to evaluate the prediction capabilities in different job co-location scenarios. We use the Mean Absolute Percentage Error (MAPE) to assess the quality of the predictions at every time step of the execution trace. We have designed two experiments:

- In Experiment 1 we evaluate the quality of the predictions when both applications are running with the different models presented. We present a table with the error metrics computed in a test set.
- In Experiment 2 we evaluate the accuracy of the model presented when predicting the runtime of co-scheduled applications. We present experiments with different methods to assess the job runtime of the co-scheduled applications.

Evaluation methodology

To evaluate the advantages of the proposed method we compare it with other sensible alternatives. Firstly, we use a naïve baseline model to estimate resources. The baseline model predicts the resource usage at time t as the sum of the resources of the isolated applications at time t. This means that the output at time t for a given input $[a; b]$ is $\tilde{y}_t = a_t + b_t$. Notice that $[a; b]$ is a matrix of features that already contains padded zeros to encode any temporal phase difference of sequences (should they exist). Therefore, if b starts k time steps after a then this baseline should predict correctly the features values of the co-located applications for the first k time-steps (since there is a single application running and there is no competition for resources).

We also compare our approach with a linear regression (LR) and a Multi-Layer Perceptron (MLP). The two regression models present a natural improvement over the baseline approach. Both methods predict the resources usage at time t as a function of the input features a_t and b_t. The baseline, the LR and the MLP take an input vector containing measurements from the isolated sequences at time t and estimate the features of the the co-located sequence at time t. This approach involves two critical problems:

- The temporal nature of the data suggests that values at time t might be correlated with nearby values . This is not taken into consideration.
- If the input has length n and the output has length $n + k$ there will be k time-steps where the models cannot make predictions, because the model has no input data from which to make predictions.

The first problem can be somewhat mitigated by using a sliding window mechanism over the input data. Nevertheless, including a sliding window mechanism increases the complexity of the solution and adds a new hyperparameter to be tuned (the length of the sliding window).

The second problem is even harder to assess. A sensible approach could be to pad the execution with zeros in the k time steps where there is no data. However, this solution invents timesteps with no resource usage and does not address the issue that, in reality, co-scheduled executions resource consumption "stretches in time" when co-scheduled jobs share resources. Models therefore require some sort of memory from past values in order to understand the effect left produced by the co-location over time. Sequence-to-sequence models with an attention mechanism naturally deal with these two issues.

A. Experiment 1: Resource usage predictions

In this experiment we evaluate the accuracy of the predictions made by the different models on co-scheduled jobs. Table II contains the MAPE errors of the predictions on the test set. The best results are obtained by ResourceNet, followed by the Multilayer perceptron. Notice that for some metrics, such as CPU and IOR, the error produced by the proposed model is reduced more than half when compared to the other models. Moreover, the model presents a lower standard deviation in most cases.

To assess the behavior of the presented models in the test set visually, we plotted the resource usage of three triplets. Figures 3, 4 and 5 show three different pairs of co-located applications with different properties. Each figure is composed of three columns. The first two columns show the resources usage trace in the isolated runs. The third column shows the trace of the co-located applications with the predictions made by the MLP and the sequence to sequence model. The shaded region shown in the third column displays the period at which both applications run at the same time. This is the period used to compute the error metrics (when both applications are running at the same time). Moreover, vertical discontinuous lines mark the time step at which the input $[a; b]$ finishes. In the interest of clarity, the figures do not contain the predictions of the other models in order to avoid excessively cluttered images.

When applications have low resource demands, and especially when they compete for resources only during small periods of time, the expected demands are easy to predict. Figure 3 shows a pair of low-demand applications competing for resources during only a small period of time. In this case, all models are able to predict all metrics well. In contrast, in cases where more demanding applications are co-located, the expected resource usage of the co-located applications over time is not straightforward to predict. One of the main difficulties in this type of scenarios is the slowdown of both applications while competing for resources, which usually implies a big difference in execution time with respect to the applications being run in isolation. This can be seen in Figs. 4 and 5. In Fig. 4 the input jobs take around 40 time steps to execute in isolation, but require more than 80 under the presented co-schedule. The same effect, even more pronounced, can be seen in Figure 5.

	baseline		linear regression		multilayer perceptron		ResourceNet	
	MAPE	std	MAPE	std	MAPE	std	MAPE	std
CPU	14.9	19.6	16.5	12.7	14.2	12.7	6.4	10.4
CPI	24.4	14.9	7.6	6.0	7.5	5.9	4.1	5.9
IOR	13.7	15.6	11.7	8.5	10.7	8.2	3.8	4.5
IOW	3.5	8.0	1.7	1.8	1.8	1.9	1.4	1.6
RAM	57.7	37.6	13.8	12.3	12.5	12.0	7.0	9.1
TLBM	6.5	05.3	3.9	3.3	3.9	3.3	1.7	3.1
PF	3.7	6.9	4.5	4.8	4.6	4.9	2.7	4.4
LLCM	8.8	12.2	6.4	7.9	6.4	7.6	4.4	7.9
FLCM	8.1	08.2	5.3	5.3	4.9	4.8	3.1	4.7

TABLE II

MEAN ABSOLUTE PERCENTAGE ERROR FOR EACH METRIC AND MODEL, EVALUATED IN THE TEST SET
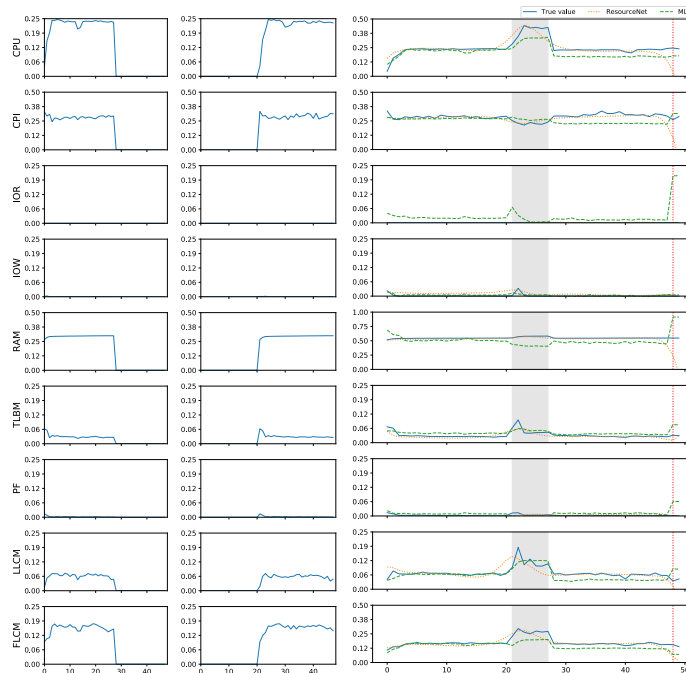

Fig. 3. This figure shows two co-located applications that share resources only during a fraction of their execution. In this example the second application starts roughly at time step 20 while the first one finishes around time step 25. In this scenario all models can capture the increase in the metrics (specially CPU usage) during the brief period in which both applications run at the same time. Notice that the vertical discontinuous line is around time step 48. This means that the applications took around the same time when they were run in isolation than when they are run co-scheduled.

In the cases presented, ResourceNet is capable of capturing the trend of the resources correctly; not only in the shadowed region but throughout the entire sequence. For example, the CPU usage in Fig. 4 starts at around 75% of usage, then decreases at around 50 and ends again at around 75% of CPU. In this figure, one may clearly observe the undesired drawback of the element-wise models previously explained. Around time step 38, we can see the predictions of the MLP getting stuck at a particular value. This is easy to observe for IOR, RAM and CPU usage, where predictions are far from the true values. Such behavior is caused due to the lack of input features in the isolated traces, so the model has to rely on making predictions out of zero-padded feature values because either a or b or both have finished. The sequence-to-sequence model is capable of generating and output sequence longer than the input sequence in a "natural" way, without any need to pad the original features.

Figure 5 shows two jobs which require a high amount of RAM. The first requires low CPU and high IOR, while the

second is just the opposite. We can see that once the first job has finished the IOR demand in the co-located trace goes to zero and our model can successfully detect this trend. Even though models capture the trends of all features reasonably well, none of them are able to predict the burstiness of some features, especially LLCM and FLCM.

B. Experiment 2: Estimating co-scheduled execution time

Stopping criteria of the co-scheduled trace

We have experimented with different criteria to predict the runtime of co-scheduled applications. Our methods are based on PC and EOS features which are detailed in Section IV-C. Using these features, we can test different criteria to predict the completion time of a co-scheduled pair of jobs.

Since we do not know in advance how long co-scheduled applications can take to finish, we have decided to treat the length of the generated trace as a hyperparameter to be adjusted. The different criteria tested are as follows:
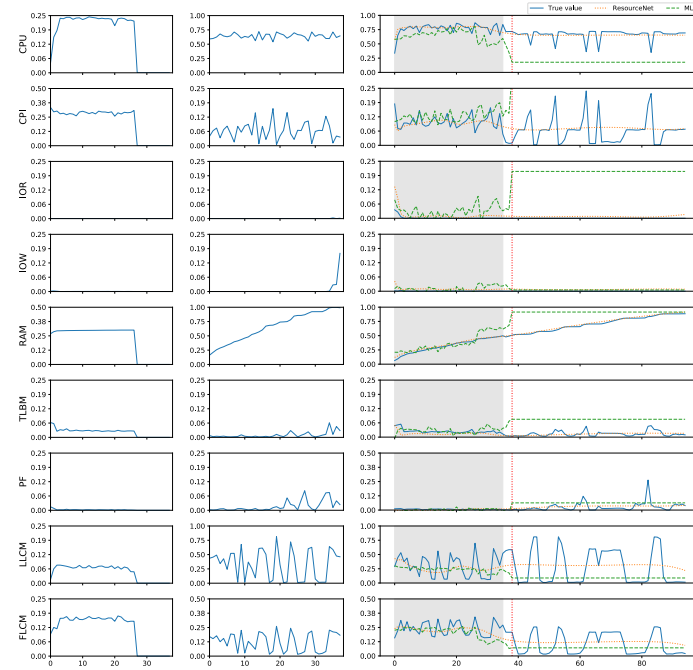
Fig. 4. This figure shows a scenario where one application demands almost all memory available. Notice how the MLP makes high error predictions for CPU and IOR when the input traces finish (marked by the vertical red line). Nevertheless the resource usage trend is predicted semi-accurately while both applications run.
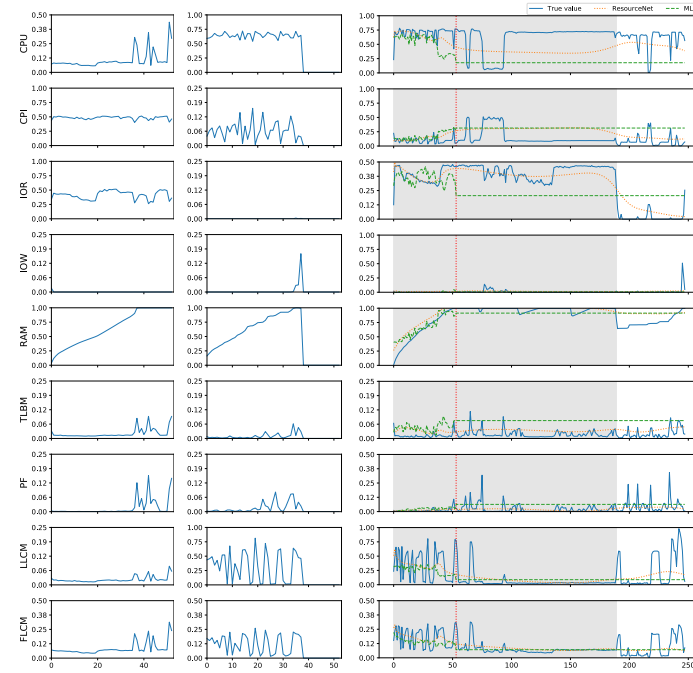

Fig. 5. This figure shows a bad co-scheduling scenario where both applications use all the available memory at some point during the execution. Notice that the execution time of the concurrent applications is roughly five times more the amount they need while running in isolation.

- *first max EOS*: applications are predicted to finish when the first local maximum is found in *EOS*.
- *first max PC sum*: applications are predicted to finishwhen the first local maximum is found in the sum of PC_a and PC_b.
- *argmax EOS*: applications are predicted to finish at the argmax of the sum of *EOS*.

- *argmax PC sum*: applications are predicted to finish at the argmax of the sum of PC_a and PC_b.

For each of the criteria, Table III reports the MAPE error and the standard deviation depending on the length of the generated trace. The first column *length* contains at position kx the expected error when the co-scheduled trace is generated up to k times x timesteps, where x is the longest of the

input traces when executed in an isolated environment. Results show than if the length of the generated trace is too short (for example $1x$), then errors are large because the stopping criteria is fired before the co-scheduled applications might finish. Errors decrease as k increases up to a point where the errors start to increase. The best results are achieved at $k = 4$. Our experiments show much lower errors for the criteria based on PC features than for criteria based on the EOS feature.
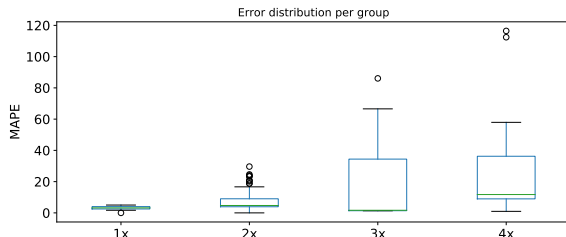

Fig. 6. This figure shows the distribution of the MAPE grouped into 4 categories. The farther away the co-located applications are predicted to finish the larger the expected error. The box plot shows results for the criterion *argmax PC sum*.

The errors made by our criteria behave differently in accordance with the length of time we predict applications will need to finish with respect to the runs made in isolation. In Fig 6 one may observe the distribution of errors made by our criteria, grouped according to where we predict the execution will end. Applications that are expected to run between $1x$ and $2x$ have low errors, while applications that are expected to finish between $3x$ and $4x$ have higher errors and more variance. Notice that this knowledge can be used as a rule for deciding when applications should not be co-scheduled. A simple rule for avoiding co-scheduling applications could be to forbid scenarios in which the model runtime prediction falls within $3x$ or $4x$.

Looking at EOS and PC feature predictions

Figure 7 shows the EOS and PC features behaviour for three different co-scheduled workload pairs. Notice that the longer a pair takes to execute, the less a signal is shown by the EOS feature. Nevertheless, PC features maintain a relatively good quality even for co-scheduled pairs with high resource competition.

Our results suggest that criteria build to predict the length of a co-scheduled trace using EOS should not be based on the actual value of the EOS feature, but rather on a function taking into account the maximum value achieved during the prediction process. Notice that, in the first case, the predicted EOS arrives roughly at 0.75. However, the second case barely takes value 0.25 and the third case the EOS scarcely barely goes above zero. This behavior increases the difficulty of predicting runtime even more with this feature.

A reasonable explanation for the behavior of EOS is that our model is trained by minimizing the mean squared error be- tween predictions and true traces. It is logical to argue that the optimization procedure induces the network to focus more on predicting PC features better than EOS, since the EOS feature takes value 1 at only 1 time step while PC features increase at every time step. The overall contribution to the error of always predicting 0 in EOS is therefore much lower than always predicting 0 in PC_a or PC_b.

VII. Conclusions

This paper introduces the use of recurrent neural networks for interference and resource prediction tasks of co-scheduled applications. We adapt a sequence-to-sequence model to work with two input sequences instead of one, thereby providing a padding containing the start time of one application with respect to the other. Our method predicts resource usage of the monitored metrics over time and is adaptable enough to be trained with workloads of arbitrary input and output lengths. Moreover, since training is done for a regression task instead of a classification task, we introduce the percentage completion features which significantly improve completion time prediction of co-scheduled applications.

Our experiments show that the model is able to predict the resource usage of co-located applications sharing resources over time, even when application performance degrades drastically due to a high demand of similar resources at the same time. Moreover, as a baseline, we compare our model with standard machine learning algorithms. The experiments show that our model makes more accurate predictions and is able to deal with the sequential nature of the data, thus making it suitable for the presented scenario where input/output pairs have different lengths.

References

[1] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale. Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12, pages 1–13, 2012.

[2] The Case and Energy-proportional Computing. The Case for Energy-Proportional Computing. Google, (December):33–37, 2007.

[3] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking, e-Energy '10, pages 215–224, New York, NY, USA, 2010. ACM.

[4] Christina Delimitrou and Christos Kozyrakis. Paragon : QoS-Aware Scheduling for Heterogeneous Datacenters. pages 77–88, 2013.

[5] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. ESP: A Machine Learning Approach to Predicting Application Interference. Proceedings - 2017 IEEE International Conference on Autonomic Computing, ICAC 2017, pages 125–134, 2017.

[6] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. Lecture Notes in Business Information Processing, 74 LNBIP(May):209–228, 2011.

[7] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In Proceedings of the 12th ACM International Conference on Computing Frontiers, CF '15, pages 53:1–53:8, New York, NY, USA, 2015. ACM.

[8] SparkPerf, Spark benchmarking and simuting system.

[9] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems, 28(1):155–162, 2012.

[10] Jitendra Kumar and Ashutosh Kumar Singh. Workload prediction in cloud using artificial neural network and adaptive differential evolution. Future Generation Computer Systems, 81:41–52, 2018.

[11] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. Improving Spark Application Throughput Via Memory Aware Task Co-location: A Mixture of Experts Approach. 2017.

trace length	first max EOS MAPE	std	first max PC sum MAPE	std	argmax EOS MAPE	std	argmax PC sum MAPE	std
1x	96.6	5.1	98.3	8.1	77.5	21.3	59.0	17.7
2x	96.6	5.1	77.7	39.1	49.1	35.9	25.1	20.4
3x	96.6	5.1	46.2	45.2	18.8	24.7	14.3	15.6
4x	96.6	5.1	29.4	37.8	30.8	35.9	12.7	18.2
5x	96.6	5.1	29.4	37.8	31.5	45.6	28.2	39.0
6x	96.6	5.1	29.4	37.8	43.8	64.5	30.0	44.1
7x	96.6	5.1	28.1	37.0	48.8	75.1	106.9	96.9
8x	96.6	5.1	28.1	37.0	63.6	97.2	107.6	98.6

TABLE III

MAPE ERRORS MADE BY PREDICTING THE EXECUTION TIME OF CO-SCHEDULED APPLICATIONS FOR THE DIFFERENT CRITERIA BASED EOS AND PC FEATURES.
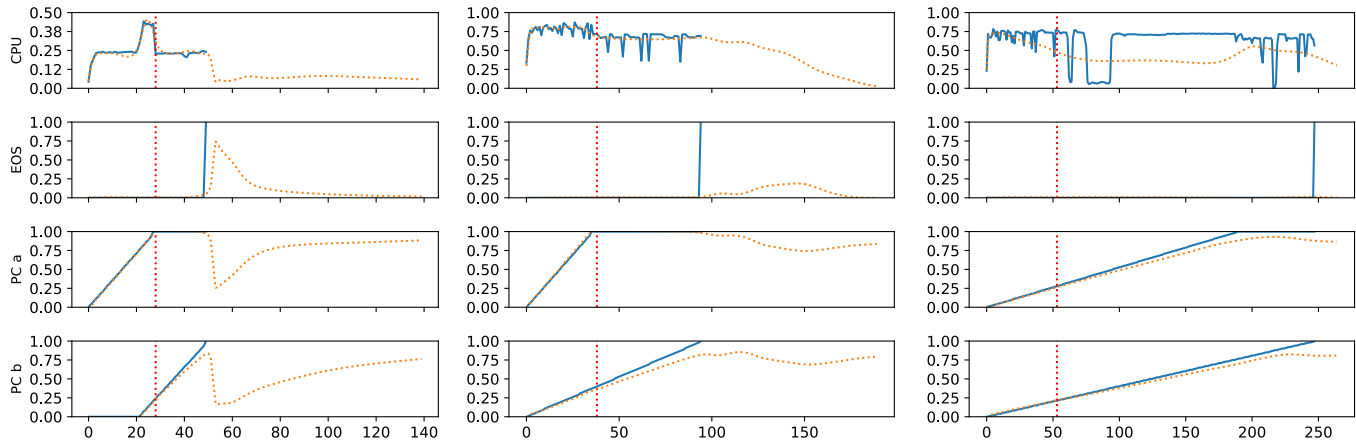

Fig. 7. This figure shows the EOS and PC features for the workloads seen in Figures [3] [4] and [5]. Every column corresponds to a different colocation of two workloads. The continuous lines show the true values of the different features and the dotted traces show the predictions made by the sequence to sequence model. Vertical lines show the length of the longest workload executed in isolation. Plots in this figure show predictions generated up to five times the length of the longest input sequence. The CPU resource usage of the co-scheduled pair has been added into the plot to contextualize the other features.

[12] Christina Delimitrou and Christos Kozyrakis. IBench: Quantifying interference for datacenter applications. Proceedings - 2013 IEEE International Symposium on Workload Characterization, IISWC 2013, pages 23–33, 2013.

[13] A. rajaraman and j. ullman. "textbook on mining of massive datasets", 2011.

[14] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15, pages 450–462, 2015.

[15] Intel 64 r and ia-32 architectures software developer's manual vol. 3b: System programming guide, part 2, sep 2014.

[16] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-Away , protecting sensitive applications from performance interference. Proceedings of the 15th International Middleware Conference on - Middleware '14, (December):301–312, 2014.

[17] Michael R. Wyatt, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H. Ahn, and Michela Taufer. PRIONN: Predicting Runtime and IO using Neural Networks. ICPP '18 Proceedings of the 47th International Conference on Parallel Processing, (ls):1–12, 2018.

[18] Josep Lluis Berral, Nicolás Poggi, David Carrera, Aaron Call, Rob Reinauer, and Daron Green. {ALOJA-ML:} {A} Framework for Automating Characterization and Knowledge Discovery in Hadoop Deployments. In Proceedings of the 21th {ACM} {SIGKDD} International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015, pages 1701–1710, 2015.

[19] Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. Modeling interference for apache Spark jobs. IEEE International Conference on Cloud Computing, CLOUD, pages 423–431, 2017.

[20] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 2014.

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. pages 1–9, 2014.

[22] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. pages 1–15, 2014.

[23] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 07-12-June:3156–3164, 2015.

[24] D. B. Prats, J. L. Berral, and D. Carrera. Automatic generation of workload profiles using unsupervised learning pipelines. IEEE Transactions on Network and Service Management, 15(1):142–155, March 2018.

[25] Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguade, Jesus Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the MareNostrum supercomputer. Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015, pages 299–306, 2015.

[26] T. Ivanov, R. Niemann, S. Izberovic, M. Rosselli, K. Tolle, and R. V. Zicari. Performance evaluation of enterprise big data platforms with hibench. 2:120–127, Aug 2015.

[27] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. 17:1–7, 2015.

[28] T. Chiba and T. Onodera. Workload characterization and optimization of tpc-h queries on apache spark. pages 112–121, April 2016.