ESP: A Machine Learning Approach to Predicting Application Interference

Nikita Mishra
University of Chicago
Dept. of Computer Science
nmishra@cs.uchicago.edu

John D. Lafferty
University of Chicago
Dept. of Statistics
lafferty@galton.uchicago.edu

Henry Hoffmann
University of Chicago
Dept. of Computer Science
hankhoffmann@cs.uchicago.edu

***Abstract*—Independent applications co-scheduled on the same hardware will interfere with one another, affecting performance in complicated ways. Predicting this interference is key to efficiently scheduling applications on shared hardware, but forming accurate predictions is difficult because there are many shared hardware features that could lead to the interference. In this paper we investigate machine learning approaches (specifically, regularization) to understand the relation between those hardware features and application interference. We propose ESP, a highly accurate and fast regularization technique for application interference prediction. To demonstrate this practicality, we implement ESP and integrate it into a scheduler for both single and multi-node Linux/x86 systems and compare the scheduling performance to state-of-the-art heuristics. We find that ESP-based schedulers increase throughput by 1.25-1.8× depending on the scheduling scenario. Additionally, we find that ESP's accurate predictions allow schedulers to avoid catastrophic decisions, which heuristic approaches fundamentally cannot detect.**

I. Introduction

Applications co-scheduled on the same physical hardware *interfere* with one another by contending for shared resources [11, 18, 24, 36]. We quantify interference as *slowdown*—the performance loss one application experiences in the presence of co-scheduled applications. By accurately predicting slowdown, a scheduler can optimally assign applications to physical machines—leading to higher throughput in batch systems and better quality-of-service for latency-sensitive applications.

Data center and super computer operators often have a great deal of accumulated data about past jobs and their interference, yet turning this data into effective interference predictors is difficult [18]. To apply machine learning to build an accurate predictor from this data, two fundamental decisions must be made: (1) what *features* should be measured and (2) what *model* maps these features into an accurate prediction. Smaller feature spaces provide more computationally efficient models, but may miss key data and reduce prediction accuracy. The art to modeling is managing the tradeoffs between feature set size and the model accuracy. One family of machine learning techniques—*regularization*— addresses the particular problem where the number of features is much larger than the number of samples; *i.e.*, the problem is *ill-posed* and unsolvable with standard regression analysis. Regularization methods solve such ill-posed problems by simultaneously selecting both the features and the model [14, 32, 38].

This paper explores such state-of-the-art regularization models for predicting application interference. We find that regularized linear regression methods require a relatively small number of features, but produce inaccurate models. In contrast, non-linear models that include *interaction terms*—*i.e.*, permit features to be multiplied together—are more accurate, but are extremely inefficient and not practical for online scheduling.

We therefore combine linear and non-linear approaches to produce accurate and practical predictions. We call our approach ESP for **E**stimating co-**S**cheduled **P**erformance. ESP's key insight is to split regularization modeling into two parts: *feature selection* and *model building*. ESP uses linear techniques to perform feature selection, but uses quadratic techniques for model building. The result is a highly accurate predictor that is still practical and can be integrated into real application schedulers.

ESP assumes there is a known (possibly very large) set of applications that may be run on the system and some offline measurements have been taken of these individual applications. Specifically, ESP measures low-level hardware features like cache misses and instructions retired during a training phase. At runtime, applications from this set may be launched in any arbitrary combination. The goal is to efficiently predict the interference (*i.e.*, slowdown) of co-scheduled applications.

We evaluate ESP by integrating it into both single and multi-node schedulers running on Linux/x86 servers. In the single-node case, we construct a batch scheduler that orders application execution to minimize the total completion time. In the multi-node case, we build a first-come-first-serve scheduler that assigns applications to nodes as they arrive to minimize application slowdown. We compare ESP-based schedulers to prior scheduling techniques that use contention-aware heuristics to avoid interference [10, 24, 31]. We also compare ESP's accuracy to predictors built with a number of cutting-edge regularization methods. We find:

- The single-node ESP schedules are, on average, 27% faster than techniques based on heuristics. Even with its runtime overhead, the ESP results are only 5% worse (on average) than an oracle that has perfect knowledge of interference and no overhead. See Section IV-B.
- The multi-node ESP schedules are 60% faster than activity vector based schedules and only 5% to 13% worse than an oracle. See Section IV-C.
- Critically, ESP produces better results as more applications are scheduled. ESP produces quantifiable performance predictions while heuristic techniques simply produce a binary decision: co-schedule or not. ESP's quantifiable predictions allow schedulers to make optimal decisions even when interference cannot be avoided. In

contrast, heuristic techniques do not quantify interference and thus cannot rank decisions. As the number of applications increases, the chance of heuristics making a very poor choice in the face of unavoidable contention also increases. See Section IV-D.

- ESP is more accurate than existing linear regression techniques. When considering two applications, ESP is similar to existing regularized linear regression techniques. Considering more than two applications, ESP is uniformly more accurate than standard linear regression techniques. See Section IV-E. For reasons explained in Section II-B, existing regularized regression methods with interaction terms cannot be evaluated for accuracy because their models are too complex to be implemented in practice.

This paper makes the following contributions: 1) ESP, a regularization method for predicting application interference, 2) Demonstration of ESP in both single and multi-node schedulers, 3) Comparison to existing heuristic techniques on real systems, 4) Comparison of ESP's predictive accuracy to other regularization methods, 5) Open-source code release[1].

ESP is designed for relatively long running applications or applications which are run multiple times. ESP requires access to low-level hardware metrics, which its learning algorithm uses as features. These features are not always accessible in virtualized environments; however, we believe the improved performance of ESP's scheduling algorithms presents a strong argument for making these low-level metrics accessible.

ESP helps scheduling at multiple levels: it determines which applications should run together on a single node, it determines which node a new application should be scheduled on, and it avoids disastrous decisions that heuristic schedulers cannot. Support for data analytics has become an important research topic in computing systems, and this paper explores how data analytics can be used to improve computing systems.

II. Regularization

This section discusses current state-of-the-art regularization methods and how they could be used to predict application interference. While complete coverage is beyond the paper's scope, our goal is to present enough background for readers to understand ESP's unique approach.

We use a running example to build intuition:

Example *Suppose we have 4 applications given as,* {`bfs`, `cfd`, `jacobi`, `kmeans`}. *For each application, we measure 4 features of its execution when run by itself: instructions per clock* (`IPC`), *L2 Access Rate* (`L2`), *L3 Access Rate* (`L3`) *and its memory bandwidth* (`MEM`). *The goal is to predict the slowdown that each of these applications will experience when run together in any combination knowing only these 4 features for each application run individually.*

A. Regularization Overview

We are interested in predicting applications' slowdown when co-scheduled with other applications as a function of individual applications' features. The following is a general formulation of this problem:

$$\mathbf{z} \sim f(\mathbf{X}), \tag{1}$$

where $\mathbf{z}$ is a vector each element is one application's slowdown when run with the other applications. $\mathbf{X}$ is a matrix containing the measured features of applications when run individually. $f(.)$ is the*prediction function* that maps the measured features into predicted slowdown. The $\sim$ sign indicates that $\mathbf{z}$ is a random variable drawn from a distribution given by $f(.)$.

To go from Equation (1) to a useful model two issues must be addressed: (1) which features we choose to include in $\mathbf{X}$ and (2) what function $f(.)$ best represents this problem. In fact, these issues are intertwined; depending on the features we include we can choose different functions. It is important to choose a function that captures the data without overfitting so that the model's generalization error is small. The model also must be computationally fast so that it is practical.

Example *Suppose we want to predict the interference of* `bfs` *and* `cfd` *when co-scheduled. The vector* z *represents slowdown, with the first element* `bfs`*'s slowdown and the second is* `cfd`*'s. X in Equation* (1) *is constructed using* `IPC`, `L2`, `L3` *and* `MEM` *of the* `bfs` *and* `cfd` *when they run in isolation. The first four columns of X are the features for* `bfs` *and the last four columns are* `cfd`*'s features. The goal is to find the $f(.)$ that produces the best prediction of slowdown.*

B. Linear Regression

Regression models are the most commonly used statistical models for predictions. A linear model predicts outcomes as a linear combination of input features:

$$\mathbf{z} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{2}$$

$\mathbf{z} \in \mathbb{R}^n$ is the dependent variable to be predicted. $\mathbf{X} \in \mathbb{R}^{n \times p}$ is the measured independent features. $\boldsymbol{\beta} \in \mathbb{R}^p$ is the coefficient vector that combines the features to produce the prediction. $\boldsymbol{\epsilon} \in \mathbb{R}^n$ is a vector representing inherent noise.

Building a linear model is the process of determining $\boldsymbol{\beta}$ by running experiments and measuring both the features and the corresponding outcomes. These measurements are samples of $\mathbf{z}$ and $\mathbf{X}$, written as $(s(\mathbf{z}), s(\mathbf{X}))$. After sampling, $s(\mathbf{z})$ and $s(\mathbf{X})$ are known and we can solve for $\boldsymbol{\beta}$. Then we use $\boldsymbol{\beta}$ and new features ($\mathbf{X}$) to predict $\mathbf{z}$ for unobserved instances.

C. Regularized Linear Regression

When determining $\boldsymbol{\beta}$, if the number of samples in $\mathbf{z}$ (or length of $s(\mathbf{z})$) is less than the number of features p the problem is ill-posed; *i.e.*, $\mathbf{X}$ is not invertible and there are infinitely many solutions. This particular case where $p > n$ is called the *high-dimensional* setting. Machine learning researchers have developed several *regularization* methods which add structure to high-dimensional problems to make them solvable.

Example *To apply linear regression, $f(.)$ is chosen to be $f(.) = \mathbf{X}\boldsymbol{\beta}$ and $\boldsymbol{\beta}$ is the parameter that to be learned from the data. For example, we could observe 3 pairs from the set of 6 pairs (for the 4 example applications) and predict the performance for the other 3 pairs. $\mathbf{z}$ denotes our applications' performance vector, thus for 3 pairs, $\mathbf{z} \in \mathbb{R}^6$. The matrix $\mathbf{X}$ in Equation* (1) *is $\mathbf{X} \in \mathbb{R}^{6 \times 8}$ and $\boldsymbol{\beta} \in \mathbb{R}^8$. Now, the system of equations $\mathbf{z} = \mathbf{X}\boldsymbol{\beta}$ has infinitely many solutions so we must* regularize*—i.e., add more structure to—the problem.*

There are several methods to add structure to a regression problem. In general, none are uniformly better than the others

[1]The code is available at https://github.com/ESPAlg/ESP

and their performance is data-dependent. Hence, standard practice is to use the model with the highest out-of-sample predictive power; *i.e.*, to separate samples into training and test data, build multiple models with the training data, and use the model that most accurately predicts the test data.

1) Ridge regularization: One way to add structure to the problem is to include additional constraints. *Ridge* regression penalizes large coefficients in $\boldsymbol{\beta}$ [14] by requiring that $\|\boldsymbol{\beta}\|_2^2 \leq t$, where t is a threshold:

$$\min_{\boldsymbol{\beta}} \|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \quad \text{s.t. } \|\boldsymbol{\beta}\|_2^2 \leq t \tag{3}$$

Example *Adding the ridge regularizer to our example, the modified problem is* $\|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2$ *and* $\sum_{i=1}^{8} \boldsymbol{\beta}^2[i] < t$*. This problem has a unique solution for a given* t*. This method shrinks large* $\boldsymbol{\beta}$ *values but does not make them 0. Hence, even if feature* i *is not important it has a positive coefficient* β_i*. Thus this approach may incorporate irrelevant features into the model. In our example, it is unlikely that L2 Access Rate is a valuable feature for predicting application interference, as L2 caches are private in most server-class processors and therefore not a source of contention.*

2) Lasso regularization: feature selection is another regularization method equivalent to setting some elements of $\boldsymbol{\beta}$ to 0, so those features cannot influence $\mathbf{z}$. Formally, feature selection can be achieved by adding $\mathcal{L}_0$ constraint—$\|\boldsymbol{\beta}\|_0 \leq t$—to Equation (2), but doing so makes the problem non-convex and NP-hard. $\mathcal{L}_1$ regularization—known as *lasso*—is the best convex relaxation for $\mathcal{L}_0$ regularization:

$$\min_{\boldsymbol{\beta}} \|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \quad \text{s.t. } \|\boldsymbol{\beta}\|_1 \leq t \tag{4}$$

Using lasso, the number of selected features will be smaller than the number of samples. A potential drawback, however, is that it cannot capture models where there are more relevant features than samples.

Example *The modified example looks like,* $\|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2$ *and* $\sum_{i=1}^{8} |\boldsymbol{\beta}[i]| < t$*. Lasso regularization will set* $n - p$ *features to 0. In our example, that means three features will be zeroed out. This could be a problem because it is likely that IPC, L3 accesses, and Memory Bandwidth for both applications (i.e., a total of 6 features for the two applications) will be necessary to predict interference as all three of these features correspond to shared hardware structures. Specifically, lasso regularization will probably zero-out the L2 Accesses for both applications, but by construction, it must also zero-out at least one of the relevant features, likely producing lower accuracy estimates.*

3) Elastic-net regularization: Elastic-net addresses the drawbacks of the two previous techniques [38]:

$$\min_{\boldsymbol{\beta}} \|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \quad \text{s.t. } \alpha\|\boldsymbol{\beta}\|_2^2 + (1-\alpha)\|\boldsymbol{\beta}\|_1 \leq t \tag{5}$$

where $\alpha = \lambda_1/(\lambda_1 + \lambda_2)$ and λ_1 and λ_2 are the regularization parameters. In practice, setting $\alpha = 0.5$ is common and λ_1 is set during model training using cross-validation. Thus under these settings elastic-net is,

$$\text{EN}(\mathbf{z}, \mathbf{X}) = \underset{\|\boldsymbol{\beta}\|_2^2 + \|\boldsymbol{\beta}\|_1 \leq t}{\arg\min} \|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \tag{6}$$

Elastic-net *groups* variables so that strongly correlated variables tend to be selected or rejected together.

Example *Adding elastic-net to our example, the modified problem looks like,* $\|\mathbf{z} - \mathbf{X}\boldsymbol{\beta}\|_2^2$ *and* $\sum_{i=1}^{8} \alpha|\boldsymbol{\beta}[i]| + (1-\alpha)\|\boldsymbol{\beta}[i]\|^2 < t$*. The regularization in this case would shrink large values for* $\boldsymbol{\beta}$ *and some coefficients would shrink to 0. It would zero-out unimportant feature like* `L2` *for both applications, yet capture all the important features (IPC, L3, MEM) for both applications even when the number of samples is smaller than the number of features.*

D. Higher-order Models

In some cases, linear models do not accurately predict the problem. One higher-order approach is to add *interaction* terms to the model, meaning that the dependent variable is possibly a multiplicative combination of some features.

A model with interaction terms can be found by adding additional terms to $\mathbf{X}$ and β. Thus, a high dimensional problem becomes even higher dimensional, increasing model complexity and overhead. For a matrix $\mathbf{X}$ with dimensions $n \times p$, $n \ll p$ *elastic-net* is $\mathcal{O}(p^3)$ [38], hence for a quadratic model the computational complexity blows up to $\mathcal{O}(p^6)$. Even though this model is richer and captures complex interactions among features, it is prohibitively expensive. ESP's motivation is to achieve the prediction accuracy of interaction terms while maintaining the practicality of linear models.

Example *We believe the interaction between our features captures the interference more accurately than a linear model. However, we do not know which of these feature interactions are important in advance—e.g., is it IPC and L3, L3 and MEM, the L3 features for both applications, or something else? Clearly, even in our simple example, the design space has become much more complex. Specifically, we capture the new interaction terms by extending the feature matrix* $\mathbf{X}$ *to* $\tilde{\mathbf{X}}$*, which has 8 linear terms plus* $\binom{8}{2}$ *higher order terms. The new design matrix* $\tilde{\mathbf{X}}$ *in Equation* (1) *is* $\tilde{\mathbf{X}} \in \mathbb{R}^{6\times 36}$ *and* $\boldsymbol{\beta} \in \mathbb{R}^{36}$*. We see—even for our simple example—the model complexity has greatly increased.*

E. A New Regularization Method

To summarize, regression models map features into predictions. When the feature space is large, regularization adds structure to make the problem well-formed. Higher-order models may provide more accurate predictions, but increase the cost of both training and applying the model.

Inspired by these observations, we present ESP, which splits regression modeling into two parts: (1) feature selection and (2) model building with interactive terms. First, ESP builds a linear model with elastic-net, which greatly reduces feature size without capturing interaction terms. Second, ESP builds a higher-order model with interaction terms using just those features selecting in the first step. By reducing the feature size in the first step, the complexity of the higher-order model remains tractable and we get the benefits of both approaches: *highly accurate predictions with manageable complexity.*

Given a set of m applications k applications to be co-scheduled, there are $n = \binom{m}{k}$ possible sets of k applications. We use p to denote the number of features. Let $f^{(k)}(.)$ be the prediction function; *i.e.*, $f^{(k)}(.)$ predicts the slowdown of each of the k co-scheduled applications using features measured

Algorithm 1 ESP

Input: Training samples: $\big(s(\mathbf{z}^{(k)}),\ \mathbf{x}^{(k)} = s(\mathbf{X}^{(k)})\big)$; Features: $\mathbf{X}^{(k)}$,
1: Variable selection using elastic-net on linear model: $\mathcal{S}$: $\{i \in \mathcal{S}$ if $\boldsymbol{\beta}^{(k)}[i] \neq 0\}$, where $\boldsymbol{\beta}^{(k)} = \text{EN}(s(\mathbf{z}^{(k)}), \mathbf{x}^{(k)})$.
2: New feature matrix with higher order terms: $\tilde{\mathbf{X}} = [X_{\mathcal{S}}, \text{Int}(X_{\mathcal{S}}))]$.
3: Estimating performance: $\hat{\mathbf{z}}^{(k)} = \tilde{\mathbf{X}}^{(k)}\boldsymbol{\beta}^{(k)}$, where $\boldsymbol{\beta}^{(k)} = \text{EN}(s(\mathbf{z}^{(k)}), s(\tilde{\mathbf{X}}^{(k)})$
4: **return** Slowdown: $\hat{\mathbf{z}}^{(k)}$.

when each application runs individually:

$$\mathbf{z}^{(k)} = f^{(k)}(\mathbf{X}^{(k)}) + \boldsymbol{\epsilon}, \tag{7}$$

where $\mathbf{z}^{(k)} \in \mathbb{R}^{nk}$ is the slowdown vector, $\mathbf{X}^{(k)} \in \mathbb{R}^{nk \times 2p}$ is feature matrix and $\boldsymbol{\epsilon} \in \mathbb{R}^{nk}$ is the Gaussian error vector. For any index j = index(i,S) of vector $\mathbf{z}$, z_j is the slowdown of application-i when co-scheduled with applications from set S. $X_{(j,:)}$ = [features of application-i, $\sum_{s \in S}$(features of application-s)]. In practice, the set of possible features includes everything that can be measured with the Intel performance counter monitor tool [34]. Taking these measurements for each core on our test system we get $p = 409$ unique features per individual application.

Algorithm 1 summarizes ESP, which predicts slowdown when k applications are co-scheduled. The algorithm takes a few samples of random combinations of applications running together, denoted by $\big(s(\mathbf{z}^{(k)}),\ \mathbf{x}^{(k)} = s(\mathbf{X}^{(k)})\big)$, a design matrix containing the low-level-features for all $\binom{n}{k}$ combinations (denoted by $\mathbf{X}^{(k)}$) and outputs a slowdown vector for all combinations of applications. The first step does a feature selection on the linear model using Elastic-net as $\boldsymbol{\beta}^{(k)} = \text{EN}(y^{(k)}, \mathbf{x}^{(k)})$. The non-zero coefficients of $\boldsymbol{\beta}^{(k)}$ indicate the selected features, denoted by $\mathcal{S}$. Then, we construct a higher-order feature matrix $\tilde{\mathbf{X}}$ for those selected features only. We run elastic net again for the new feature space to obtain the final model which makes the performance prediction, denoted by $\hat{\mathbf{z}}^{(k)}$.

Algorithm 1 is run entirely offline. It produces the slowdown estimates for all applications (even those not sampled) in up to k co-scheduling groups. These slowdown estimates can then be used to predict performance for new combinations of applications in online schedulers. We show examples in the next section. We also show how the estimates can be trivially updated as new measurements become available online.

III. Scheduling with ESP

We examine two use cases for ESP: (1) batch scheduling applications on a single processor, (2) and scheduling dynamically arriving applications on multiple processors.

A. Single-node Scheduling

We assume a set S of applications with $|S| = m$. Each of the applications have work $w_1, w_2, ...w_m$. They can be scheduled to run alone or with other applications. Our goal is to compute the schedule that completes all applications' work in the minimal total time. The optimal schedule can be described as a solution to a linear program if the performance of every application was known ahead of time. Since we do not know the exact performance, our algorithm uses ESP to predict the performance and then generate near-optimal schedules.

We assume that at most k-tuples of applications can run together at a time, meaning we must predict the performance for $2\binom{m}{2} + 3\binom{m}{3} + \ldots + k\binom{m}{k}$ application combinations. The intuition behind the restriction on k is that the system will become saturated at some point and it is no longer beneficial to continue to add applications to a saturated node.

We first develop a linear program for optimal scheduling assuming known performance. We will relax that assumption momentarily. We also assume that preemption is allowed. Then, an optimal schedule is given by:

$$\begin{aligned} \mathbf{y} = \underset{\mathbf{y} \geq 0}{\arg\min}\ \ & \|\mathbf{y}\|_1 \\ \text{subject to}\quad & \mathbf{A}\mathbf{y} = \mathbf{w} \end{aligned} \tag{8}$$

This equation is quite simple, but the complexity is in the structure of the $\mathbf{y}$ vector and $\mathbf{A}$ matrix. Each element of $\mathbf{y}$ is the time for a specific set of applications to be co-scheduled, while the columns of $\mathbf{A}$ represent the each application's performance when co-scheduled in that set. For example, if $m = 3$, the superset of all possible sets of applications is $\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{1,3\},\{1,2,3\}\}$. If we order these sets, then $\mathbf{y}_j$ in Equation (8) is the time spent when all the applications in set j run together and $\mathbf{A}_{ij}$ is the speed of application i when co-scheduled with applications from set j. This linear program has a sparse solution with at most $2m$ non-zero solutions, hence the context switching cost is not very high and is bounded by the total number of applications.

Equation (8) produces a minimal time schedule but is not practical. It requires all application interference is known and it assumes deterministic application performance. Real performance is subject to inherent noise. We therefore extend Equation (8) by assuming that the observed performance $\hat{\mathbf{A}}$ is drawn from a Gaussian distribution:

$$\hat{A}_{ij} \sim \mathcal{N}(A_{ij}, \Sigma_{ij}). \tag{9}$$

Thus, we sample $\hat{A}_{ij}$ to predict A_{ij} and Σ_{ij}. If $\tilde{\mathbf{A}}$ is our prediction for $\mathbf{A}$, then instead of solving Equation (8) we solve:

$$\begin{aligned} \mathbf{y} = \underset{\mathbf{y} \geq 0}{\arg\min}\ \ & \|\mathbf{y}\|_1 \\ \text{subject to}\quad & \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w} \end{aligned} \tag{10}$$

This equation is a proxy for Equation (9), but it cannot guarantee that the required work is finished because it uses predictions rather than true performance. To deal with this uncertainty we design an iterative algorithm which repeatedly solves the approximate linear program in Equation (10) until all work is finished. This approach accounts for inherent noise due to error in both performance measurement and prediction.

Algorithm 2 takes four parameters: an error tolerance (defaulted to 10^{-6}), ESP's predicted performance matrix, a vector representing the total work for each application, and a scalar η controlling the trade-off between exploration-exploitation. If we believe that ESP's performance prediction is very good, we set $\eta = 1$ to run fewer iterations, saving computation. If we have less confidence, then we send a smaller amount of work to the linear program, reducing each iteration's error and learning more about the application mix scheduled that iteration.

This algorithm loops until all work is complete. For each loop iteration, the algorithm takes a *step*, with the step size inversely proportional to η. It then solves Equation (10) using the predicted performance and schedules the applications according to this solution. After running the schedule for

Algorithm 2 Iterative Scheduling Algorithm

Input: Tolerance: TOL $= 10^{-6}$
Performance prediction matrix from ESP: $\tilde{\mathbf{A}}$
Total work: $\mathbf{w}$
Exploration-exploitation parameter: η ($\eta \geq 1$)

1: Initialize scheduling matrix $\mathbf{y}^{(0)} = 0$, work remaining: $\mathbf{w}_{rem} = \mathbf{w}$, total scheduling time: $s = 0$.
2: **while** $\|\mathbf{w}_{rem}\|_2 \geq$ TOL **do**
3: Work to be processed, $\mathbf{w}_{sent} = \mathbf{w}_{rem}/\eta$.
4: Solve linear program based on the prediction, $\mathbf{y} = \arg\min_{\mathbf{y} \in \mathcal{C}} \|\mathbf{y}\|_1$, where $\mathcal{C} = \{\mathbf{y} : \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w}_{sent}, \mathbf{y} \geq 0\}$.
5: Schedule/run applications according to time slices given by $\mathbf{y}$. Update scheduling time as, $s = s + \|A\mathbf{y}\|_1$.
6: Update predicted performance $\tilde{\mathbf{A}}$ based on true performance observed $\hat{\mathbf{A}}$ and update remaining work, $\mathbf{w}_{rem} = (\mathbf{w}_{sent} - \hat{\mathbf{A}}\mathbf{y})_+$.
7: **end while**
8: **return** Total scheduling time: s.

the specified time, the algorithm updates the performance prediction using its latest observations and updates the work vector with the work accomplished in this step. Given perfect knowledge of the application interference and no system noise, the algorithm requires only a single step and would be optimal.

B. Multi-node Scheduling

We now consider scheduling dynamically arriving applications on multiple processors. Applications arrive in a stream and a centralized scheduler assigns an application to a processor (which may already have applications). We assign jobs in FIFO order; a new job is immediately assigned a processor with the goal of minimizing job completion time.

The multi-node scheduler (see Algorithm 3) takes as input: (1) an error tolerance (again defaulted to 10^{-6}), (2) ESP's performance prediction A, (3) a job sequence (we do not look ahead, so in practice the sequence does not need to be known in advance), (4) the number of nodes q, and the exploration-exploitation trade-off η. Each job is denoted as $J_i = (a_i, v_i)$, where a_i is the application index and v_i is that application's work. The algorithm loops until all jobs are completed. Each iteration takes the next job and determines the expected work and time if assigned to each processor (lines 4–7). It chooses the processor that has the fastest predicted completion time (line 8) and schedules that job on the processor using Algorithm 2 (line 9).

IV. Evaluation

A. Experimental Setup

1) Experimental System and Benchmarks: Our test platform is composed of four dual-socket Ubuntu 14.04 system with SuperMICRO X9DRL-iF motherboards and two Intel Xeon E5-2690 processors. These processors have eight cores, hyper-threading (eight additional virtual cores), and Turbo-Boost. Each node has 64 GB of RAM.

We use 15 benchmarks from different suites including PARSEC (`blackscholes`, `fluidanimate`, `swaptions`, `x264`) [2], Minebench (`Kmeans`, `HOP`, `svmrfe`) [26], Rodinia (`cfd`, `particlefilter`, `vips`, `btree`, `backprop`, `bfs`) [3] and SEEC (`Dijkstra`, `jacobi`) [16]. These benchmarks test a range of important applications with both compute-intensive and I/O-intensive workloads. All the applications run with up to 32 threads (the maximum supported in hardware on our test machine). We construct multiprocessor workloads by randomly selecting benchmarks from this list. When we need more than 15 benchmarks, we allow duplicates.

Algorithm 3 Multi-node Iterative Scheduling Algorithm

Input: Tolerance: TOL $= 10^{-6}$
Performance prediction matrix from ESP: $\tilde{\mathbf{A}}$
Jobs arriving in stream: $J_1, J_2, \ldots, J_T$
Number of processors: q.
Exploration-exploitation parameter: η ($\eta \geq 1$)

1: Initialize work matrix: $\mathbf{W} = \mathbf{0}_{m \times q}$
2: **for all** i = 1:T **do**
3: Obtain job $J_i = (a_i, v_i)$. where a_i is the application index and v_i denotes the work for that application.
4: **for all** j = 1:q **do**
5: Expected work, $\mathbf{w}_{tmp} = \mathbf{W}(:, j)$; updated with new job's work $w_{tmp}[i] = w[a_i, j] + v_i$
6: Expected schedule time, $s[j] = \min_{\mathbf{y} \in \mathcal{C}} \|\mathbf{y}\|_1$, where $\mathcal{C} = \{\mathbf{y} : \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w}_{tmp}, \mathbf{y} \geq 0\}$.
7: **end for**
8: Greedily choose processor $P : P = \arg\min_{i \in [q]} s_i$ with least expected scheduling time and update corresponding entry in $\mathbf{W}$.
9: Run Algorithm 2 for processor P with $\mathbf{w}_{tmp}$ amount of Total work, $\tilde{\mathbf{A}}$ as the performance prediction matrix and $\eta = 5$.
10: **end for**
11: **return** Total scheduling time for all processors: $\mathbf{s}$.

We measure performance of all applications as wall-clock execution time. Interference is the slowdown one application experiences when co-scheduled with one or more other applications. We evaluate the schedulers in terms of time to complete scheduled jobs. We evaluate the accuracy of interference predictors in terms of the difference between the predicted and actual slowdown.

2) Points of Comparison: We compare ESP with:

- *Activity Vectors* – Activity vectors maximize resource usage variance [24]; *i.e.*, they co-schedule applications with widely differing resource needs. Extensions to activity vectors have made similar ideas suitable for dynamic consolidation in virtualized data centers [31] and for scheduling dynamically arriving applications [10]. We compare against the original activity vector approach for the single-node case and against the extension to dynamically arriving applications in the multi-node case. This approach results in compute-intensive and memory intensive applications being scheduled together. Unlike ESP none of these approaches produce quantified slowdown predictions, but instead make decisions to co-schedule or not. We find that biasing this approach to be more sensitive to different resources produces different results. We consider variants biased toward:
  - *Memory (MEM)*: favors co-scheduling applications with different bandwidth needs.
  - *Instructions per cycle (IPC)*: favors co-scheduling applications with differing compute needs.
  - *L3 Request (L3R)*: favors co-scheduling applications with differing L3 cache needs.
- *Random (RND)*: co-schedules applications randomly.
- *Oracle*: represents the best schedule given perfect knowledge of application interference; *i.e.*, it is equivalent to knowing the true performance $\mathbf{A}$ in Equation (9). Not implementable in practice, we construct the oracle through a brute force search for all application mixes.

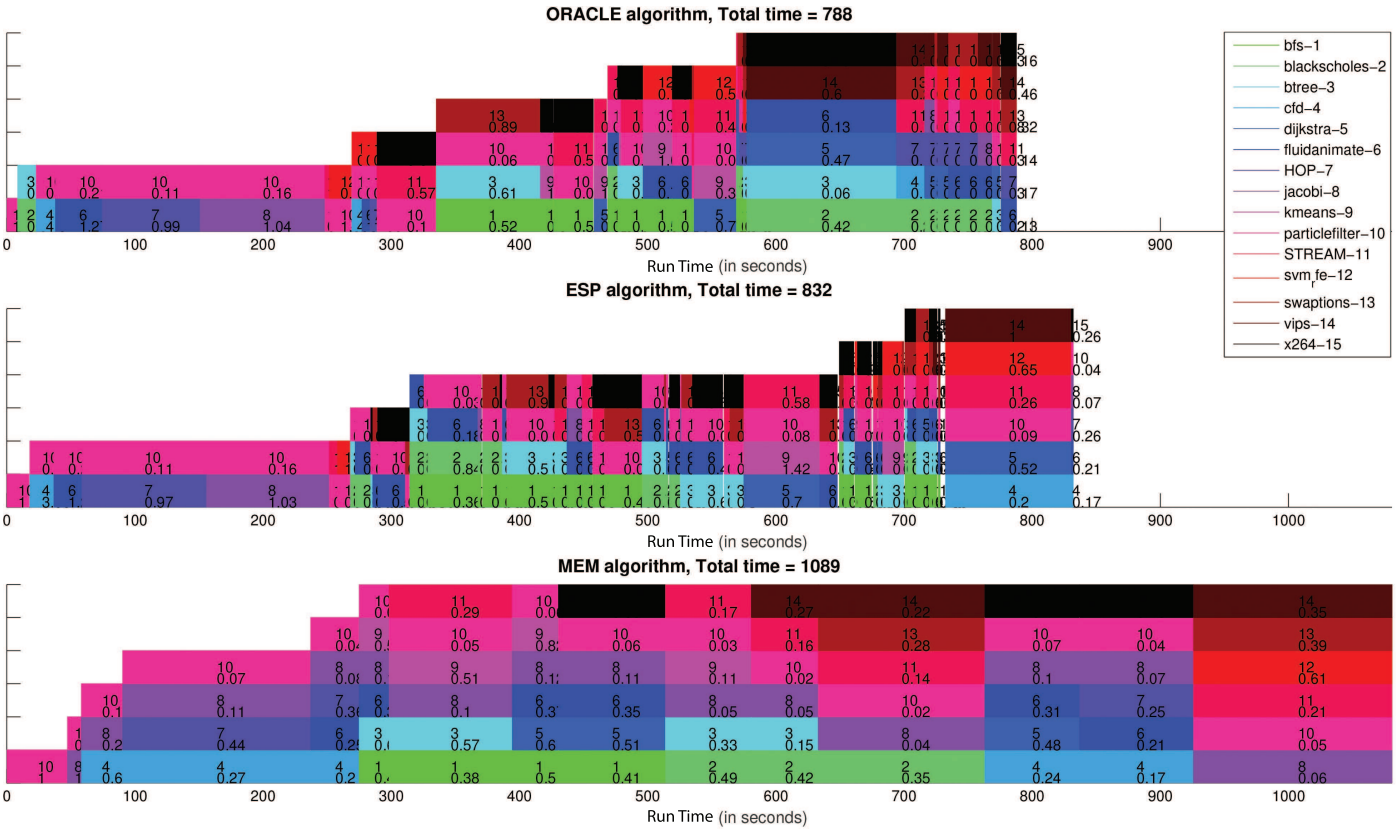

Fig. 2. Comparison of schedules from different algorithms with up to 6 co-scheduled applications (k = 6) (more compact is better). Each block represents an application running with other applications. The number in the top of the block is the application index. The number at the bottom is the application's slowdown. ESP's schedule is significantly more compact than the MEM baseline.

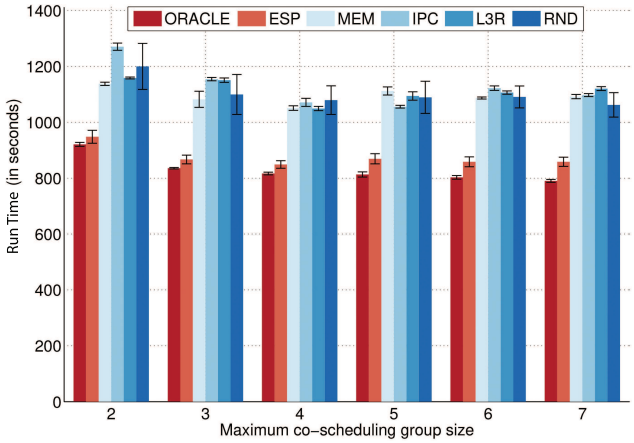

Fig. 1. Single processor scheduling performance. On average ESP is 25% better than MEM, 29% better than IPC, 27% better than L3R, 26% better than RND, and only 5% worse than ORACLE.

B. Single Node Scheduling Results

To test single-node batch scheduling, we launch all 15 benchmarks listed in Section IV-A. The schedulers order application execution to minimize total job completion time. We vary k, the maximum number of applications that can be co-scheduled from two to six. We find that it is never beneficial to schedule more than six applications simultaneously and no scheduler (other than random) ever attempted to do so even when more were allowed. We run 15 separate experiments for each k, where each experiment differs by the split between training and testing data for ESP.

Fig. 1 shows the results. The x-axis is k, the y-axis is the run time, and there is a bar showing the mean scheduling time for each of our points of comparison with an error bar showing one standard deviation. Overall, ESP performs much better than the baseline algorithms and only slightly worse than the ORACLE. On average over different k, ESP is 25% faster than MEM, 29% faster than IPC, 27% faster than L3R and 26% faster than RND. These results include prediction and scheduling overhead, yet ESP is only 5% worse than the ORACLE, which has no overhead and knows the future. We conclude that ESP produces highly accurate predictions and yet is efficient enough for practical use.

To provide intuition, Fig. 2 illustrates the schedules produced by: ORACLE (top), ESP (middle), and MEM (bottom) when they are allowed to co-schedule up to six applications. For each chart, the horizontal axis represents time in seconds. Each colored block represents an application running with other applications. The top number in the block represents the application index (see the legend for mapping from index to name), and the bottom number represents the actual slowdown the application experienced at that time (this actual slowdown is determined after the fact). A better schedule is more compact and completes further to the left. Clearly, ESP's schedule is more compact and closer to the ORACLE than MEM. ORACLE runs the maximum number of applications together less than half of the time. MEM, in contrast, runs the maximum allowed number of applications most of the time. Overall in

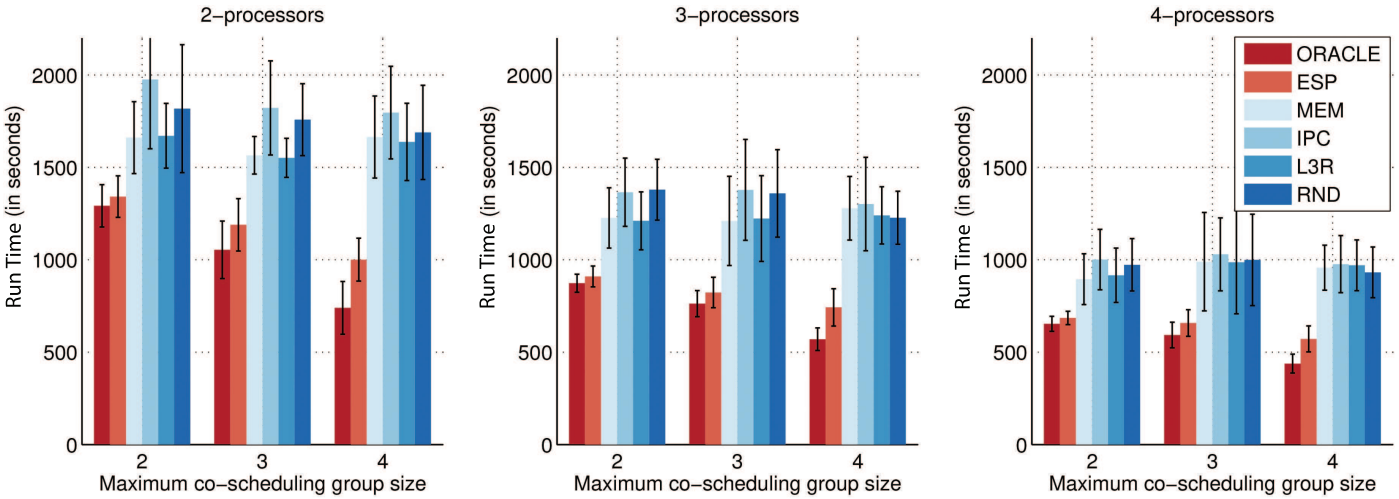

Fig. 3. Comparison of multi-node scheduling times for stream of 40 applications (lower is better). On an average over different processes and tuples, ESP is 47% better than MEM, 61% better than IPC, 47% better than L3R and 54% better than RND.

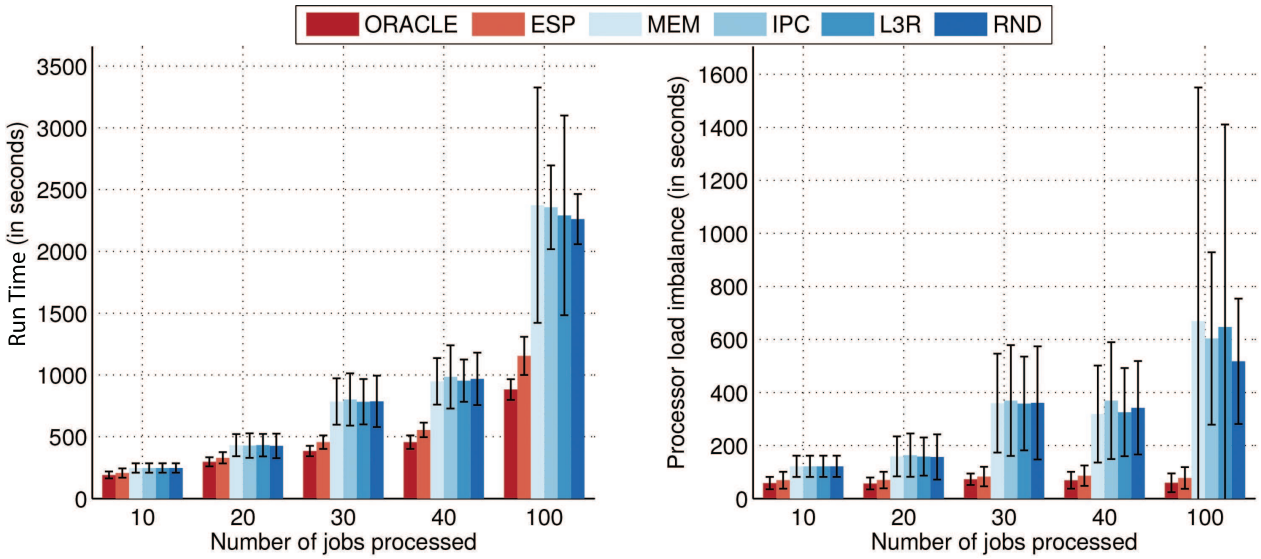

Fig. 4. Multi-node scheduling performance with varying number of incoming jobs, allowing up to 4 co-scheduled applications per node (lower is better). The left figure shows scheduling time (in seconds) – on average ESP is 60% better than MEM, 61% better than IPC, 58% better than L3R and 58% better than RND. The right figure shows the load imbalance (the difference between the highest and lowest scheduling time across nodes). As we increase the number of jobs the load imbalance increases faster for the baseline algorithms and seems relatively constant for ESP and the ORACLE.

this particular instance, the scheduling time for the ORACLE is 788 seconds, 832 for ESP, and 1089 for MEM.

C. Multi-node Scheduling Results

In this section we discuss multi-node scheduling performance. We use two, three, and four copies of our base test platform (described above). We assume that applications arrive randomly from our set of 15 benchmarks (so some benchmarks may have multiple instances live in the system). The challenge is to assign an application to a processor as it arrives such that the total impact on the system is minimized; *i.e.*, put the application on the node that minimizes interference.

The oracle computes the best possible schedule, the heuristics use activity vectors to select the best node and the ESP-based approach uses Algorithm 3. Fig. 3 summarizes the results for 2, 3 and 4 processors and for $k = 2, 3$ or 4. As we increase the number of processors, the scheduling time improves for all approaches. ESP performs almost as well as the ORACLE, performing 5% to 13% worse on an average. In fact, ESP is always within 1 standard deviation of ORACLE. ESP performs significantly better than the activity vector algorithms. On average over different processes and tuples, ESP is 47% better than MEM, 61% better than IPC, 47% better than L3R, and 54% better than RND. Additionally, the standard deviation in performance for ESP is much lower compared to the baseline algorithm, leading to much better performance predictability. This predictability is further evidence of our claim that ESP allows the schedulers to avoid bad predictions.

D. Multi-node Sensitivity to Job Size

We also study the scheduling performance as a function of the total number of applications scheduled. To be clear, jobs

are still scheduled as they arrive (the multi-node scheduler does not reorder applications), we simply have more of them. Specifically, we send up to 100 jobs to a system with 4 nodes and we are allowed to co-schedule up to 4 applications together. For each experiment, we perform 15 trials with different random job arrivals. All results report the mean with error bars indicating one standard deviation.

Fig. 4 shows how scheduling performance changes as a function of the total number of applications scheduled. The figure consists of two charts: the one on the left showing scheduling time as a function of applications scheduled and the one on the right showing load imbalance in terms of the largest difference in execution time between one processor and another. The number of applications scheduled is shown on the x-axis and the time on the y-axis. As the number of applications increases, ESP's relative performance improves compared to the activity vector approaches. The key to this result is ESP's accurate quantification of interference. The ability to directly reason about slowdown allows the ESP-based approach to rank scheduling decisions and always choose the one with the least impact on the performance of both the running applications and the application that just arrived.

ESP's foresight becomes more crucial as the number of jobs increase because bad decisions can create severe load imbalance on the parallel machine, as shown on the right side of Fig. 4. As we increase the number of jobs the processor load imbalance increases vastly for the activity vector approaches and seems relatively constant for ESP as well as ORACLE. On an average ESP is 60% better than MEM, 61% better than IPC, 58% better than L3R and 58% better than RND. Again, the standard deviation (shown by the error bars) for ESP is much lower than the baselines—leading to much more predictable performance for latency sensitive applications.

E. ESP Prediction Accuracy

We compare ESP with the *elastic-net* regularization method on the linear model (*Enet-lin*) described in Section II-C3. To evaluate quantitatively, we measure the *accuracy* of the predicted performance $\hat{\mathbf{z}}$ with respect to the true data $\mathbf{z}$, by computing the *coefficient of determination* (R^2 value):

$$\text{accuracy}(\hat{\mathbf{z}}, \mathbf{z}) = \left(1 - \frac{\|\hat{\mathbf{w}} - \mathbf{w}\|_2^2}{\|\mathbf{w} - \bar{\mathbf{w}}\|_2^2}\right)_+, \qquad (11)$$

where $\mathbf{w} = \min(\mathbf{z}, \mathbf{1})$ and $\hat{\mathbf{w}} = \min(\hat{\mathbf{z}}, \mathbf{1})$. This metric captures how well the predicted results correlate with the measured results. Unity represents perfect correlation.

Fig. 5 shows a box-plot for out-of-sample predictive accuracy of ESP and Enet-lin when we train both the models with 70% data and test the prediction on the remaining data. For $k = 2$, ESP performs only slightly better than the Enet-lin with on average 76% accuracy whereas the baseline is 73% accurate. For $k > 2$, the prediction accuracy for the ESP varies from 93% to 96% with very small variance. On the other hand, Enet-lin is only between 85% to 88% accurate.

F. ESP Accuracy versus the Netflix Algorithm

We compare ESP's accuracy to that of collaborative filtering; *e.g.*, the Netflix algorithm, which is used by other machine-learning based interference predictors [8, 9]. Specifically, we use collaborative filtering to predict which pairs of applications will work well together. The intuition is that if two applications we have never run together work well with some third application, they may work well together themselves.

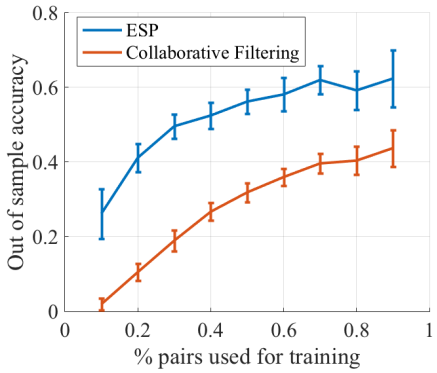

Fig. 6. Sample complexity plot for Netflix algorithm used by Paragon

Fig. 6 compares the accuracy of the Netflix algorithm and ESP for predicting the interference of pairs of applications. The y-axis shows accuracy and the x-axis shows the percentage of samples. This experiment shows that ESP always outperforms collaborative filtering for our benchmarks. These results demonstrate the importance of incorporating low-level features into the prediction.

G. Overhead

ESP has an offline sampling phase followed by an estimation phase. Once we have collected the samples from the batch of co-scheduled applications, we run ESP to obtain the predictions for the rest of the combinations of the applications. We have summarized the overhead results in the Table I. We require less than 0.7 seconds to build the performance model for a batch of k-tuples running together. Once the model is built, the prediction time for ESP is very small and would range from 0.5 milliseconds to 60 milliseconds. For such small training data, the prediction accuracy for $k = 2$ is around 80% and for $k > 2$ it is around 86%.

The scheduling overhead, again has two components: first obtaining the application's batch run profile which is done using ESP, and then solving the linear program to obtain the schedule. The first part (in the top of Table I) is done offline. The vast bulk of online work is done by Equation (10), which solves a very sparse optimization problem. The total overhead per scheduled job for Algorithm 2 – the online portion of the approach – ranges from 0.008 seconds for $k = 2$ to 0.22 seconds for $k = 7$. Almost all of this work could be parallelized on a multicore (or accelerated with SIMD instructions) but that is beyond the scope of this work. We note that it is quite practical to consider co-scheduling up to 4 jobs. The overhead of scheduling 7 jobs may become prohibitive, but it is never useful on our test system. This is not an insignificant amount of time but this approach is suitable for long running applications and the benchmarks that we have used in this paper are all long running benchmarks with at least 10 seconds of individual runtime.

V. Related Work

Accurate performance estimates are essential for scheduling and resource allocation [7], but these estimates are difficult to obtain due to the complexity and diversity of large-scale

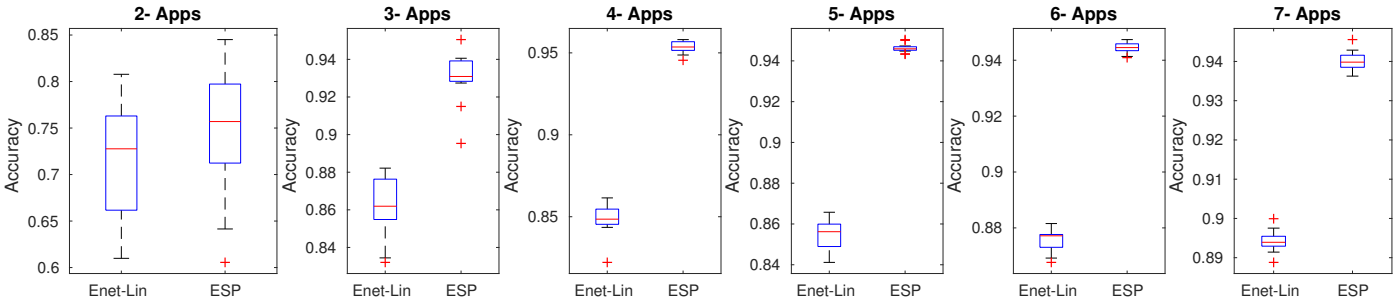

Fig. 5. Prediction accuracy comparison of elastic-net linear regression and ESP.

TABLE I. OVERHEAD

	k	2	3	4	5	6	7
ESP	Model training (s)	0.42	0.59	0.68	0.69	0.48	0.51
	Prediction time (s)	0.00056	0.003	0.0076	0.0198	0.04	0.06
	Training sample size (%)	70%	40 %	10 %	4%	1 %	1%
Scheduling Alg. 2	Lin. prog. (in s/job)	0.008	0.014	0.023	0.06	0.117	0.22

systems [19]. A particular challenge is modeling performance loss due to contention [18]. Better contention models could improve system utilization while ensuring quality-of-service in latency sensitive applications [36].

Several statistical machine learning approaches estimate power, performance, and energy of a single application running on a single system. Examples predict the power and performance of various resource allocations to single applications [29] or optimize energy efficiency under latency constraints [25]. A recent approach combines machine learning with control theory to guarantee energy consumption, but only for a single application [15]. Mantis is very similar to ESP as it also uses higher-order regularized regression models (based on Lasso) to predict smartphone app performance [20]. Mantis, however, does not predict contention among multiple applications, which is ESP's focus.

Other approaches predict and mitigate contention in single-node systems. Many decide to co-schedule or not, but they do not produce quantitative slowdown estimates. For example, Dwyer et al. propose a classifier that predicts whether contention will be high or low, but this approach does not produce a numerical estimate [11]. Similarly, ReSense detects highly contended resources, but it never quantifies contention [10]. Another approach estimates throughput (total system performance), but does not produce estimates of individual application performance [4, 35]. Subramanian et al. produce accurate performance estimates (within 9.9%) based on only last-level cache access rate and memory bandwidth [30]. These results are achieved on a simulator rather than a real system, however, and on our real system these two features are not sufficient to predict contention with any level of accuracy. Another single-node system, D-Factor, uses non-linear models to predict the slowdown of a new application given current resource usage [21]. Unlike ESP, D-Factor cannot predict how two applications will interfere if neither is currently running.

Several approaches estimate and mitigate contention for multi-node systems. Activity vectors work on single or multi-node systems by maximizing the variance among resource usage in co-scheduled applications [24]. This heuristic makes intuitive sense—applications with very different resource needs are less likely to interfere—but this approach does not produce quantitative estimates and therefore can make bad decisions when contention is unavoidable. Merlin also estimates contended resources and migrates virtual machines to areas of lower contention, but it also does not produce slowdown estimates [31]. DejaVu classifies application workloads and then schedules according to known good schedules for the class [33]. DejaVu creates an *interference index* which ranks slowdowns for VM migration or resource reallocation, but it does not produce estimates of the actual slowdowns incurred. Similarly, Quasar [9] and Stay-Away [27] predict applications that are likely to interfere, but neither produces performance estimates. POET can maintain a target application's performance in the presence of other applications, but it does not optimally schedule application mixtures [17]. Bubble-flux [36]—an improvement over the earlier Bubble-up [23]—does produce slowdown estimates and—like ESP—it is efficient enough to consider interference among more than two applications. The main difference between Bubble-flux and ESP is that Bubble-flux uses no offline prior information and must dynamically probe the system.

Table II compares ESP to some examples of prior work, including both heuristic (Bubble-flux and Activity Vectors) and machine learning (Paragon/Quasar) approaches. Both Bubble-flux and Quasar require long periods of online measurement. Quasar—based on the earlier Paragon—requires up to a few minutes of online sampling before it can generate an interference estimation [8]. Bubble-flux and ESP represent extreme ends of interference estimation approaches. Bubble-flux needs no prior knowledge, while ESP assumes prior measurements. When a new application enters the system, Bubble-flux must run it in an isolated setting with a special *bubble* benchmark to estimate whether this application should be run alone or consolidated. Bubble-flux then migrates the application to a new server based on observed behavior with the bubble. ESP requires much greater offline profiling, but makes an immediate decision about where to place a new application.

Much prior work considers virtual machine consolidation, which requires interference estimation between applications running with some isolation guarantees [5, 28]. Govindan et al. provide a heuristic which depends on measuring the impact of cache interference [13]. Maji et al. design a control

TABLE II. COMPARISON OF ALGORITHMS WITH N APPLICATIONS AND M FEATURES.

Category	Example Project	Online measurements	Online overhead
Heuristic	Bubble-flux [36]	✓	$O(1)$
	Activity vectors [24]	×	$O(1)$
Learning	Paragon/Quasar[8, 9]	✓	$O(NM)$
	ESP	×	$O(M^2)$

based algorithm to consolidate VMs for QoS applications, where the control parameters are estimated online using a regression approach [22]. Chiang and Huang propose an interference model that requires profiling not just individual applications but also combinations of applications [6]. Unlike these approaches, ESP requires access to low-level features for predicting application interference; but these may not be available in virtualized environments. Given the benefits these feature provide for estimating interference, we propose that VMs consider making this data available to improve co-location and consolidation.

We model interference estimation as a *high-dimensional* regression problem with prohibitively many dimensions. Many statistical methods address high-dimensionality (*e.g.*, SURE [12]). In computer system performance, however, measurable features are highly correlated and existing methods do not provide high accuracy. Other statistical models emit more accurate predictors given correlated features (*e.g.*, [37] and [1]), but they do not scale to our problem size. Even though we make a strong assumption that the interaction terms are present only if their individual linear terms are significant, we achieve high accuracy and good schedules in practice.

VI. CONCLUSION

This paper explores machine learning methods for predicting application interference in computing systems. Specifically, we explore several state-of-the-art regularization techniques for high-dimensional problems—when more features are available than samples—and we conclude that existing linear techniques are not accurate enough, while existing higher-order techniques are accurate but slow. Inspired by these observations we present ESP, a combination of linear feature selection with higher-order model building that achieves the practicality of linear models with the accuracy of higher-order models. We demonstrate ESP's quantitative predictions produce significantly better schedules than existing heuristics for both single and multi-node systems, with up to 1.8× improvement in application completion time and significantly lower variance. ESP achieves much higher prediction accuracies than prior approaches—over 93% when considering three or more applications. We make source code available so that others may improve on or compare with ESP.

Acknowledgments We thank the anonymous reviewers for their feedback and Anshul Gandhi for paper shepherding. The effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

REFERENCES

[1] J. Bien et al. "A lasso for hierarchical interactions". In: *Annals of statistics* 41.3 (2013), p. 1111.

[2] C. Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *PACT*. 2008.

[3] S. Che et al. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *IISWC*. 2009.

[4] X. Chen et al. "Performance and power modeling in a multi-programmed multi-core environment". In: *DAC*. 2010.

[5] X. Chen et al. "Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds". In: *MASCOTS*. 2015.

[6] R. C. Chiang and H. H. Huang. "TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments". In: *SC*. 2011.

[7] S.-H. Chiang et al. "The impact of more accurate requested runtimes on production job scheduling performance". In: *Job Scheduling Strategies for Parallel Processing*. 2002.

[8] C. Delimitrou and C. Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *ASPLOS*. 2013.

[9] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *ASPLOS*. 2014.

[10] T. Dey et al. "ReSense: Mapping Dynamic Workloads of Colocated Multithreaded Applications Using Resource Sensitivity". In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013).

[11] T. Dwyer et al. "A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads". In: *SC*. 2012.

[12] J. Fan and J. Lv. "Sure independence screening for ultrahigh dimensional feature space". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 70.5 (2008), pp. 849–911.

[13] S. Govindan et al. "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 22.

[14] A Hoerl and R Kennard. *Ridge regression, in Encyclopedia of Statistical Sciences, Vol. 8*. 1988.

[15] H. Hoffmann. "JouleGuard: energy guarantees for approximate applications". In: *SOSP*. 2015.

[16] H. Hoffmann et al. "SEEC: a general and extensible framework for self-aware computing". In: (2011).

[17] C. Imes et al. "POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints". In: *RTAS*. 2015.

[18] M. Kambadur et al. "Measuring interference between live datacenter applications". In: *SC*. 2012.

[19] S. Kanev et al. "Profiling a warehouse-scale computer". In: *ISCA*. 2015.

[20] Y. Kwon et al. "Mantis: Automatic performance prediction for smartphone applications". In: *USENIX ATC*. 2013.

[21] S.-H. Lim et al. "D-factor: A Quantitative Model of Application Slow-down in Multi-resource Shared Systems". In: *SIGMETRICS Perform. Eval. Rev.* 40.1 (June 2012).

[22] A. K. Maji et al. "Ice: An integrated configuration engine for interference mitigation in cloud services". In: *ICAC*. 2015.

[23] J. Mars et al. "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations". In: *MICRO*. 2011.

[24] A. Merkel et al. "Resource-conscious scheduling for energy efficiency on multi-core processors". In: *Eurosys*. 2010.

[25] N. Mishra et al. "A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints". In: *ASPLOS*. ACM. 2015.

[26] R. Narayanan et al. "MineBench: A Benchmark Suite for Data Mining Workloads". In: *IISWC*. 2006.

[27] N. Rameshan et al. "Stay-Away, Protecting Sensitive Applications from Performance Interference". In: *Middleware*. 2014.

[28] A. Roytman et al. "PACMan: Performance Aware Virtual Machine Consolidation." In: *ICAC*. 2013, pp. 83–94.

[29] D. C. Snowdon et al. "Koala: A Platform for OS-level Power Management". In: *EuroSys*. 2009.

[30] L. Subramanian et al. "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory". In: *MICRO*. 2015.

[31] P. Tembey et al. "Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems". In: *SOCC*. 2014.

[32] R. Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288.

[33] N. Vasić et al. "DejaVu: Accelerating Resource Allocation in Virtualized Environments". In: *ASPLOS*. 2012.

[34] T. Willhalm. *Intel performance counter monitor-a better way to measure CPU utilization*. 2012.

[35] C. Xu et al. "Cache contention and application performance prediction for multi-core systems". In: *ISPASS*. 2010.

[36] H. Yang et al. "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers". In: *ISCA*. 2013.

[37] M. Yuan and Y. Lin. "Model selection and estimation in regression with grouped variables". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1 (2006), pp. 49–67.

[38] H. Zou and T. Hastie. "Regularization and variable selection via the elastic net". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320.