Predicting Cross-Architecture Performance of Parallel Programs

Daniel Nichols†, Alexander Movsesyan†, Jae-Seung Yeom*, Abhik Sarkar*, Daniel Milroy*, Tapasya Patki*, Abhinav Bhatele†

†*Department of Computer Science, University of Maryland*
**Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*

***Abstract*—A variety of hardware architectures, both CPUs and GPUs, are used today to build supercomputers and parallel clusters. Often times, users can choose which hardware platform they want to run on. Modern scientific workflows have multiple computational tasks, and each task may be better suited for a different architecture in terms of performance. Deciding where to run an application or workflow task is not straightforward because of the complexity of applications, and hardware architectures, which makes performance predictions challenging. Hence, modeling the performance of scientific applications across a variety of architectures is important for achieving the best performance. In this paper, we present a machine learning based methodology to model the relative performance of applications across multiple architectures using hardware performance counters. Our machine learning model can predict the relative performance of an application with a mean absolute error of 0.11, and can be used effectively to make performance-aware and multi-architecture scheduling decisions, reducing makespan by up to 20%.**

***Index Terms*—performance modeling, architectures, machine learning, multi-cluster scheduling**

I. Motivation

An increasing number of scientific workloads are being expressed as workflows with sets of computational tasks and dependencies between them [1], [2]. These workflows typically involve ensembles of tasks (jobs) in a pipeline that run different codes such as simulations, uncertainty quantification analysis, and machine learning training. As applications become more portable due to the emergence of portable programming models [3], package managers [4], and containerization techniques, different tasks or jobs might be better suited for different hardware architectures. Given these portable workflows and the increasingly heterogeneous set of computing resources available to end users today, it is important to develop capabilities to efficiently place these tasks on the most efficient resources available.

Different tasks or applications in a workflow can be assigned to different architectures if users have access to a variety of compute nodes via a multi-resource job scheduler, which is becoming increasingly common, both in data centers and HPC facilities. As a result, the demand for such multi-resource schedulers [5] is emerging. In an ideal setting, scheduler can automatically decide the most suitable architecture for different jobs in terms of performance. This can remove the user from the decision making process and let a system scheduler decide what hardware to run an application on. However, in practice, this requires being able to predict the performance of incoming jobs across diverse architectures. This is a complex problem that would involve developing models for understanding the performance of scientific applications across diverse architectures.

Cross-architecture performance modeling is a challenging problem because application execution times are dependent on several factors with non-trivial relationships to performance. The performance depends on how well the application's behavior aligns with the properties of the hardware it is running on. These hardware properties, such as peak flop/s, memory bandwidth, and cache sizes are easy to obtain, however, the behavior of the application is non-trivial to model. Application performance can depend on a number of characteristics such as arithmetic intensity, memory loads/stores, branching behavior, I/O, and many more. Characterizing these and using them to model performance on a diverse set of architectures is challenging due to the number of contributing factors and complexity of the relationship.

In this paper, we propose a solution to the cross-architecture performance modeling task by training a machine learning model to predict the relative performance of an application across a set of architectures given performance counters of the application from one architecture. In order to accomplish this, we collect a data set of application runs from four different HPC systems with different architectures and measure a hand selected set of performance counters. These counters, along with the recorded execution times, are used to train a regression model to predict relative performance vectors. Additionally, we demonstrate the generalizability of our model by evaluating it on a set of applications it has not seen before. To our knowledge, this is the first model to be able to predict performance across multiple architectures at the same time that works on entire applications. Finally, we demonstrate the makespan improvement from using this model in a multi-resource scheduling simulation.

In this paper we make the following contributions:

- A dataset of hardware performance counters for a wide variety of scientific applications recorded on four different HPC systems.
- A regression model that can predict the relative performance of an application across multiple systems with a

mean absolute error of 0.11.
- A qualitative comparison of the importance of different counters in cross-architecture performance modeling.
- A demonstration of the potential makespan improvement if our model is used to assist scheduling decisions in a multi-resource scheduler.

II. Background

In this section, we provide background on performance profiling, relative performance vectors, and regression modeling.

A. Performance Profiling

When studying performance related aspects of an application, performance profiling tools are often used to collect data. These tools record profiling metrics such as wall time during an application execution, and often attribute those metric values to different regions of the application's code.

A popular performance profiling tool for parallel programs is HPCToolkit [6]. It is a sampling-based tool that can collect numerous operating system and hardware counters and attribute them to nodes on a calling context tree. It can record counters such as cache misses, floating point operations, branch instructions, etc. While many tools can record this type of data, HPCToolkit has been demonstrated to be more accurate than the others with relatively low overheads [7].

Typically, this analysis is done through HPCToolkit's graphical interface, hpcviewer, making studying trends in large numbers of profiles very difficult [8]. The Python library, Hatchet [9], solves this problem by providing a programmatic interface to the profiles produced by HPCToolkit and other popular profilers. Additionally, it provides extensive functionality for calling context tree pruning and analysis through pandas Dataframe operations.

B. Regression Modeling

Traditional machine learning (ML) is often tasked as learning to predict some output given an input to the machine learning model. When the output is discrete it is called *classification*. On the other hand, when the output is continuous it is called *regression*. The latter of these training objectives is used in the modeling in this paper.

When training a regression model, it is necessary to have a dataset, $\mathcal{D}$, of existing data where outputs are known. The amount of data, $|\mathcal{D}|$, needed is dependent on the model, features, and problem complexity, however, typical regression tasks can require thousands to tens of thousands of training samples. This data, along with its corresponding outputs, is used to optimize the model's predictions respective to some learning objective. Common learning objectives in regression are to minimize *mean absolute error*, *mean squared error*, coefficient of determination (R^2), etc. on a testing data set. The testing data set is separate from the data that the model was trained on, so that reported values do not include overfitting, i.e. the model does not memorize the data set. Often times the function being minimized is additively combined with a regularization term to reduce model complexity.

$$\theta^\star = \min_\theta \mathcal{L}(\boldsymbol{x};\theta) + \Omega(\theta)$$

Here $\mathcal{L}(\boldsymbol{x};\theta)$ is the loss function at data sample $\boldsymbol{x} \in \mathcal{D}$ parameterized by θ. It is intended to model predictive capacity of the model such as with mean absolute error. The second term, $\Omega(\theta)$, is the regularization term, which models the complexity of the machine learning model. Penalizing model complexity helps prevent overfitting of the data set.

III. Related Work

Below, we present related work in the areas of machine learning based performance modeling and cross-platform performance modeling. We further discuss how our work differs from and builds on top of existing work.

A. Performance Modeling with Machine Learning

Performance modeling is a well studied research area with lots of literature surrounding analytical and statistical models. Recently, with the increase in machine learning innovations, there has been a large focus on the latter. Machine learning can help model complex relationships between applications and their final performance. It has been used to model job runtimes [10], [11], variability [12], power consumption [13], and many other things [14].

These models are often used to study and understand complex relationships between applications and their performance. Malakar et al. [14] compare the capability of various different machine learning methods on modeling performance. Furthermore, Zhou et al. [15] demonstrate how to extrapolate models from small scale runs to larger scale runs. Many works also use these models in downstream tasks to improve performance. In [16] standard machine learning techniques such as k-Nearest-Neighbors and XBoost are used to model MPI collective performance and inform auto-tuning decisions. This fits into the broader study of using machine learning models to more efficiently explore the combinatorial search space in auto-tuning [17]–[19]. Similar to this paper, there are other works that use machine learning models to make informed scheduling decisions on HPC systems such as to reduce variability [12] or avoid IO bottlenecks [20].

B. Cross-Architecture Performance Modeling

Two works by Ardalani et al. [21], [22] focus specifically on cross-architecture performance modeling. More specifically, they consider predicting performance across architectures. Like our work these use expert derived counters to model the computational behaviour of an application. However, they only focus on mapping sequential C code performance to GPU performance. They do not look at a multiple architectures or a wide variety of applications and they only consider single functions rather than entire application binaries. Another similar paper by Yang et al. [23] introduces a model for predicting performance of parallel applications between two architectures using cumulative averages and a filter model.

This work differs from ours in that it requires running the application on both architectures to make predictions and it only considers CPU architectures. These lines of work, [21], [22] and [23], do not explore any potential uses of their models in applications such as multi-resource scheduling. Some works have used heuristics and machine learning models to do resource placements for tasks in workflows [24], [25]. These have used test runs and search space pruning to find optimal resource sets. However, none of them use cross-architecture predictive models to inform their resource selection.

IV. Overview of Our Methodology

We first provide an overview of our methodology to predict the relative performance of an application across a set of architectures given performance counters of the application from one architecture. This includes two things – the data collection phase and the model training phase (Figure 1). In the first phase, we collect performance profiles for a variety of applications running on N different HPC systems with different architectures and record a hand-selected set of performance counters. These counters, along with the recorded execution times, are used to train a regression model to predict relative performance in the second phase.

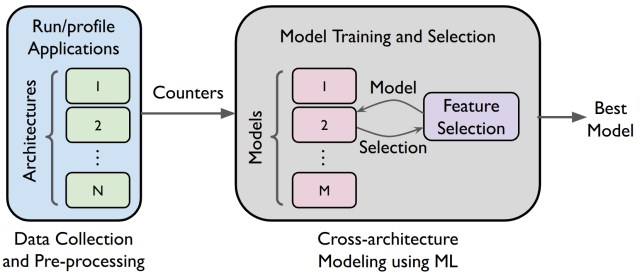
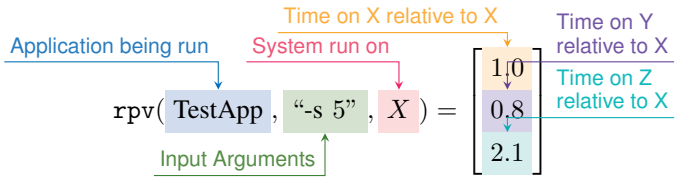


Fig. 1. Overview of data and machine learning pipeline. Applications are profiled on several architectures and performance counters are collected for training the model. Model and feature selection are done iteratively until the best set is selected.

Since our goal is to predict performance on other architectures relative to a baseline on one architecture, we introduce the term *Relative Performance Vector* (RPV) that encodes the relative performance of an application across several architectures. To define RPV, let us consider a set of applications A, corresponding input problems I_A, and systems S. For a particular application and input problem pair $(a, i) \in A \times I_A$ executed on N systems in S we can define the *Relative Performance Vector* as $\texttt{rpv} : (A, I_A) \times S \mapsto \mathbb{R}^N$ such that $\texttt{rpv}(a, i, s)$ is the vector of the performance of (a, i) across all platforms relative to that on system s. Here we assume that (a, i) can run on all the systems in S. For example, consider running an application-input pair (TestApp, "-s 5") on systems X, Y, and Z. If the application runs in ten minutes on system X, eight minutes on system Y, and 21 minutes on system Z, then the performance vector relative to X would be:



We also define $\texttt{rpv}(\cdot, \cdot, \min)$ and $\texttt{rpv}(\cdot, \cdot, \max)$ as the performance vectors relative to the systems where lowest and highest performance is obtained, respectively. The rpv provides a concise, mathematical representation for relative performance across systems that can be used in our further downstream modeling tasks.

In order to model the mapping $\texttt{rpv} : (A, I_A) \times S \mapsto \mathbb{R}^N$, we need a large number of input and output data to train on. This requires a large number of samples in the $(A, I_A) \times S$ space. To collect these, we profile a variety of aplications at several of their inputs on several architectures. These runs provide hardware counters that may provide insight into an application's behavior for many application, input, and architecture tuples.

We use the counters collected during profiling as the data set for the machine learning (ML) component (second phase). The ML component uses the profiled counters from a particular architecture to predict the relative performance vector across a set of systems. We try different ML models and feature sets to identify the best performing model. This model is exported and used in downstream relative performance prediction tasks such as cross-architecture scheduling.

V. Data Collection and Pre-processing

In this section, we provide details of how we generated the dataset used for our modeling problem. We describe the process of running and profiling the applications, and collecting the performance metrics.

A. Scientific Applications

In order to model the relative performance of applications run on an HPC machine, we need to collect performance data from applications that are typically run on these machines. We accomplish this by running a set of applications, benchmarks, and proxy applications from the ECP Proxy Applications Suite [26] and E4S Test Suite [27]. These are chosen because they are designed to be representative of actual workloads on HPC systems, but are simpler to build and run than full scientific applications.

Table II lists the applications used in our data set. There are 20 applications in total, and eleven of them have GPU support. The GPU support comes from a variety of libraries such as OpenMP, Kokkos [28], RAJA [3], and native CUDA or HIP. Each application is paired with different input configurations when run, in order to test different problems and problem sizes. We build and install all of the applications with their default build settings in their respective Spack [4] packages.

TABLE I
OVERVIEW OF THE FOUR ARCHITECTURES WE COLLECT PERFORMANCE DATA ON. THERE ARE TWO CPU ONLY SYSTEMS AND TWO CPU+GPU SYSTEMS. THE CPUS SPAN THREE VENDORS: INTEL, IBM, AND AMD, WHILE THE GPUS ORIGINATE FROM TWO: NVIDIA AND AMD.

System	CPU Type	CPU *cores/node*	CPU *Clock Rate (GHz)*	GPU Type	GPUs/node
Quartz	Intel Xeon E5-2695 v4	36	2.1	—	—
Ruby	Intel Xeon CLX-8276	56	2.2	—	—
Lassen	IBM Power9	44	3.5	NVIDIA V100	4
Corona	AMD Rome	48	2.8	AMD MI50	8

TABLE II
THE APPLICATIONS USED IN OUR STUDY LISTED ALONGSIDE A BRIEF DESCRIPTION OF WHAT EACH APPLICATION DOES AND WHETHER IT SUPPORTS RUNNING ON A GPU.

Application	Description	GPU
AMG	Algebraic multigrid solver	✓
CANDLE	Deep learning models for cancer studies	✓
CoMD	Molecular dynamics and materials science algorithms	
CosmoFlow	3D convolutional neural network for astrological studies	✓
CRADL	Multiphysics and ALE hydrodynamics	✓
Ember	Communication patterns	
ExaMiniMD	Molecular dynamics simulations	✓
Laghos	FEM for compressible gas dynamics	✓
miniFE	Unstructured implicit FEM codes	✓
miniGAN	Generative Adversarial Neural Network training	✓
miniQMC	Real space quantum Monte Carlo algorithms	✓
miniTri	Triangle based data analytics algorithms	
miniVite	Graph community detection	
DeepCam	Climate segmentation benchmark	✓
Nekbone	High-order, incompressible Navier-Stokes solver	
PICSARlite	Particle-in-Cell simulation	
SW4lite	Seismic wave simulation	✓
SWFFT	Distributed-memory parallel 3D FFT	
Thornado-mini	Radiative transfer solver in multi-group, two-moment estimations	
XSBench	Monte Carlo neutronics simulations	

B. Architecture Descriptions

We run each application-input pair on four different machines with different architectures. These are listed in Table I. There are two Intel Xeon based, CPU-only machines and two GPU-based machines. The first GPU machine uses IBM Power9 CPUs and NVIDIA V100 GPUs, while the second uses AMD Rome CPUs and AMD MI50 GPUs.

On each of these systems the applications are run in three configurations – on one core, on one node using all the cores, and on two nodes. The one-core runs use one GPU if applicable. MPI is used for the one and two node runs to make use of all the cores and GPUs on the node. Some applications only support run configurations with square or power of two MPI processes and are, thus, run on the nearest number of ranks possible to one or two nodes. If an application does not support running on a GPU, we run it on the CPU only and use comparable CPU counters. If an application does support running on a GPU, then only GPU counters are collected. During these runs, HPCToolkit [6] (with CUPTI [29] on NVIDIA GPUs or rocProfiler [30] on AMD) is used to record the application counters, and after the application run is complete, Hatchet [9] is used to parse these counters from the HPCToolkit output. For multi-process and multi-GPU runs, we record the mean value of the counters across all processes. The final results from all runs are then collected into a Pandas dataframe for use in the later tasks.

C. Details of Recorded Hardware Counters

To understand the varied computational characteristics of different applications in Table II, we record several hardware counters during the application runs. Table III lists the counters recorded on each architecture in our data set. Counter names are not consistent across different architectures and they may also represent slightly different data. However, we have tried to identify similar counters that model the same underlying performance characteristics that affect final performance. Most of these counters fit into one of three categories: control flow, data intensity, or I/O. These categories capture the main performance characteristics of applications across different architectures. Broadly speaking, applications with more complex control flow will fair better on CPUs, which are geared towards latency. On the other hand, applications with more data intensity generally benefit on throughput-geared GPUs.

D. Preparing the Final Dataset

Using the counters listed on the right of Table III we compute a set of derived values as the final features in the data set. These features are detailed on the left of Table III. The instruction related counters branch, store, load, single FP, double FP, and integer arithmetic are all computed to be ratios of the total number of instructions. This normalizes the values across runs, which may have drastically different numbers of total instructions. The remaining eight features are normalized by subtracting that feature's mean to center its values and dividing them by its standard deviation. We additionally include whether the run was from a GPU or not, how many nodes, and how many cores the run used. The architecture feature is a one-hot-encoded vector encoding what architecture the counters were collected on. In the context of

TABLE III
FINAL FEATURES IN THE COLLECTED DATA SET AND THE COUNTERS/VALUES THEY ARE DERIVED FROM. WE COMBINE DERIVED VALUES FROM THE RECORDED COUNTERS AND META-DATA ABOUT THE RUN CONFIGURATION.

Feature	**Description**	**Source Counters & Values**			
		Quartz	Ruby	Lassen	Corona
Branch Intensity	Ratio of branch instructions to total instructions	PAPI_BR_INS	PAPI_BR_INS	cf_executed	–
Store Intensity	Ratio of store instructions to total instructions	PAPI_SR_INS	PAPI_SR_INS	inst_executed_local_stores, inst_executed_global_stores	LDSInsts, GDSInsts
Load Intensity	Ratio of load instructions to total instructions	PAPI_LD_INS	PAPI_LD_INS	inst_executed_local_loads, inst_executed_global_loads	LDSInsts, GDSInsts
Single FP Intensity	Ratio of single precision FP instructions to total instructions	PAPI_SP_OPS	PAPI_SP_OPS	flop_count_dp	VALUInsts, SALUInsts
Double FP Intensity	Ratio of double precision FP instructions to total instructions	PAPI_DP_OPS	PAPI_DP_OPS	flop_count_sp	VALUInsts, SALUInsts
Arithmetic Intensity	Ratio of integer arithmetic instructions to total instructions	bdw_ep::ARITH	clx::ARITH	inst_integer	–
L1 Load Misses	L1 cache load misses	PAPI_L1_LDM	PAPI_L1_LDM	local_load_requests, local_hit_rate	–
L1 Store Misses	L1 cache store misses	PAPI_L1_STM	PAPI_L1_STM	local_store_requests, local_hit_rate	–
L2 Load Misses	L2 cache load misses	PAPI_L2_LDM	PAPI_L2_LDM	gld_efficiency	TCC_MISS_sum, TCC_EA_RDREQ
L2 Store Misses	L2 cache store misses	PAPI_L2_STM	PAPI_L2_STM	gst_efficiency	TCC_MISS_sum, TCC_EA_WRREQ
IO Bytes Written	Bytes written to IO	IO	IO	IO	IO
IO Bytes Read	Bytes read from IO	IO	IO	IO	IO
Extended Page Table	Extended page table size	EPT	EPT	EPT	EPT
Memory Stalls	Memory stalls	PAPI_MEM_SCY	PAPI_MEM_SCY	GINST:STL_ANY	MemUnitStalled
Nodes	Nodes	Run Configuration	Run Configuration	Run Configuration	Run Configuration
Cores	Cores	Run Configuration	Run Configuration	Run Configuration	Run Configuration
Uses GPU	1 if counters from GPU; 0 otherwise	0	0	1 if app uses GPU	1 if app uses GPU
Architecture	one-hot-encoded vector for what architecture these counters were recorded on	(1 0 0 0)	(0 1 0 0)	(0 0 1 0)	(0 0 0 1)

this paper, that is four separate features that are used to denote whether the run is from Quartz, Ruby, Corona, or Lassen.

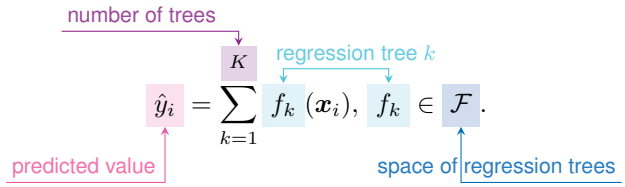
The final data set has 21 columns and 11,312 rows. Each row represents a run of an application-input pair for a specific number of MPI processes on a single architecture. The columns are derived from the counters collected during the run and meta-data about the run (see Table III).

VI. MODELING WITH MACHINE LEARNING

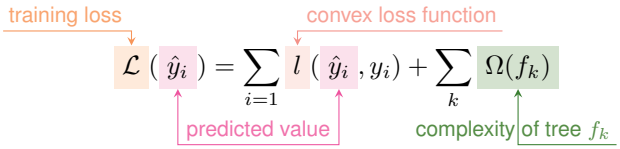
In this section we present our methodology for training the machine learning models, finding the best features/models, and evaluating their performance.

A. Training

Now that we have a data set of counters from applications and the corresponding relative performance vectors across a set of architectures, we want to use machine learning to predict the relative performance vectors given counters from one of the architectures. In order to learn how to predict relative performance vectors we use the XGBoost (eXtreme Gradient Boosting) regression model [31]. This model is an ensemble of decision trees that are additively combined to make final predictions. If $\hat{y}_i \in \mathbb{R}$ is the predicted regression value of the model, then it can be computed as

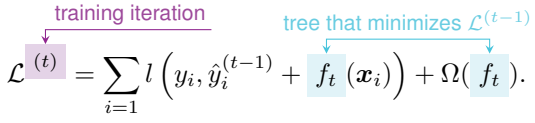


As described in Section II-B we can add a regularized objective function to avoid over-fitting.



(1)

Since this is parameterized by functions ($f_k \in \mathcal{F}$) it cannot be optimized using typical optimization methods. Thus, *gradient tree boosting* greedily adds in the best functions throughout training iterations by selecting the f_t that minimizes Equation 1 the most. These f_t can be additively combined into a new loss function as



This can be optimized using 2nd-order approximizations and standard convex minimization methods. XGBoost implements this gradient tree boosting method alongside a number of state-of-the-art techniques for tree splitting and pruning. Additionally, it provides efficient implementations that can scale to large numbers of data samples and run on GPUs. It is a state-of-the-art machine learning algorithm for learning on tabular data.

In order to train an XGBoost regressor we use its publicly available Python library at version 1.7.1. We train the model on a CPU on the Ruby system. Training the XGBoost model takes on the order of tens of seconds on average. The model takes the features from Section V-D and predicts the relative performance across all four architectures as a vector. Mean absolute error (MAE) is used as the minimization objective during training. During this training 10% of the data is set aside as a testing data set, while the other 90% is shown to the model as a training data set. While training on the training data set, the data is further split into 5 folds as part of k-fold cross-validation. The model is trained on 4 out of the 5 folds at a time, while the other is used as validation. This is done for all 5 combinations and the average MAE is reported.

We additionally train several other common machine learning regressors to compare the quality of the XGBoost model to other state-of-the-art methods. For this we include linear regression and decision forests. These are implemented from the scikit-learn Python library. As with XGBoost these are trained with a 90-10 train-test split and 5-fold cross-validation. We also test against *mean* prediction as a baseline for the ML models. This regressor guesses the mean rpv in the training set for all samples in the test set.

B. Model and Feature Selection

To select the best model and feature set we first train all the models on all the features. After training we select the best set of features using those reported by XGBoost and the decision forest, since these models expose feature importances. These features are then used to re-train all the models again.

In order to measure the feature importances of the trained model we use XGBoost to easily recover importance values. XGBoost, in its Python framework, computes feature importances during training and exposes them in its model interface. It calculates them based on the average gain across all decision splits in the trees. During training a tree will add splits on a feature to improve its predictive performance. The improvement in performance from this split is called the gain. When there are multiple regression targets the gain is averaged over each output.

For any given feature if we average the gain from all the splits on that particular feature in a tree, then we can compute the importance of the feature for that tree. This includes all the splits in XGBoost's sets of trees. Finally, we can compute this value for all of the features in the data set to retrieve a feature importance vector.

With this method of calculating feature importances we can expose the relative contribution of each feature to the model's predictions. A higher importance indicates that that feature contributes more to the models performance than other lower scored features. Decision tree feature importances can also be calculated based on the frequency and coverage of splits for a feature, however, these can be biased towards features with a large number of unique values and numeric features. Both of these are present in our data set, so we elect to use the average gain.

Since the data set has a relatively small number of features, the feature selection will likely have negligible impact on model training time. However, discovering the most impactful features gives insight into what is most necessary in predicting cross-architecture performance. Additionally, it allows us to collect less features in future implementations of this methodology. This is a considerable optimization as data collection is the most time and resource intensive portion of our machine learning pipeline.

C. Evaluation Metrics

We evaluate the model's performance by two different metrics: *Mean Absolute Error* (MAE) and *Same Order Score* (SOS). The MAE encodes the average magnitude of error in the relative performance predictions. This measure provides a value that is easy to reason about regarding predictive performance. An MAE of 0.1 means that the model predicts the relative performance of applications within ± 0.1 on average across each vector.

$$\text{MAE} = \frac{1}{|\mathcal{D}_{\texttt{rpv}}|} \sum_{i=1}^{|\mathcal{D}_{\texttt{rpv}}|} \frac{\| \texttt{rpv}_i - \widehat{\texttt{rpv}}_i \|_1}{d}$$

data set of relative performance vectors; predicted rpv for run i; rpv for run i; number of architectures

The SOS score denotes the number of samples where the model predicts the relative performance vector in the correct order. We define two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as being in the same order if the i-th elements $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ are both the n-th largest in their respective vector, for all i. The SOS is then defined as the fraction of predicted relative performance vectors that are in the same order as their respective true relative performance vector. This metric shows how well the model understands the ordering of performance on different architectures, but ignores the magnitude of its predictions. Thus, the SOS combined with MAE gives reasonable insight into how well the model is predicting relative performance vectors. Both of these metrics are computed over the testing set for data samples that the model has not seen before.

VII. Scheduling Experiment

Once a model is trained to predict relative performance vectors it can be used to make informed cross platform scheduling decisions. We test this capability in our trained model by simulating a multi-resource scheduling environment. We create a workload of 50,000 jobs randomly sampled from our existing data set with replacement. These are scheduled using the First-Come-First-Serve with EASY backfilling scheduling algorithm

(FCFS+EASY) [32] presented in Algorithm 1. This algorithm uses the `Machine` function to map a job to a machine: Quartz, Ruby, Lassen, or Corona. If the machine cannot satisfy the resource requirement of a job (the number of nodes it needs), then the job is reserved at the earliest possible time or backfilled. Otherwise, it is run immediately, and the function $\texttt{Start}(j, m)$ represents running job j on machine m. We use the observed run times on each machine from the data set to determine how long the job would run for simulation purposes.

Algorithm 1 Multi-Resource Scheduling Algorithm using FCFS+EASY. This standard algorithm queues jobs using policy $\mathcal{R}_1$ and uses EASY to backfill smaller jobs. The function `Machine` is used to map jobs to resources. The symbol $\setminus$ represents the set minus operation.

Input: $Q \leftarrow$ queue of jobs
$\mathcal{R}_1 \leftarrow$ Queue ordering policy
$\mathcal{R}_2 \leftarrow$ Backfill ordering policy
$M \leftarrow$ Set of machines used for multi-resource scheduling
$\texttt{Machine}(j, i, M) \leftarrow$ Function that maps jobs to machines

$i \leftarrow 0$
sort Q according to $\mathcal{R}_1$
for job $j \in Q$ **do**
  if j can start now **then**
    pop j from Q
    $Start(j, \texttt{Machine}(j, i, M))$
    $i \leftarrow i + 1$
  else
    Reserve j at earliest possible time
    $L \leftarrow Q \setminus \{j\}$
    sort L according to $\mathcal{R}_2$
    for job $j' \in L$ **do**
      if j' can start now without delaying j **then**
        pop j' from L and Q
        $Start(j', \texttt{Machine}(j', i, M))$
        $i \leftarrow i + 1$

We run this scheduling simulation with four different machine placement functions: Round-Robin, Random, User+RR, and Model-based. These functions expose the common interface for scheduling, $\texttt{Machine}(j, i, M)$, where j is the job to schedule, i is the index of j in the queue, and M is the set of machines considered for multi-resource scheduling. Depending on the algorithm some of these arguments are not used. The Round-Robin placement places jobs on machines in a round robin fashion rotating between machines for each consecutive job. The Random placement uniformly selects a random machine of the four to run on. The User+RR placement mimics traditional user behavior by running on GPU systems for GPU enabled apps and CPU only systems otherwise. Round robin is used to decide which GPU system to use for GPU enabled apps and likewise for CPU-only apps. Finally, the Model-based placement, Algorithm 2, uses an ML-based model to pick the fastest machine for each job and run it there. If the machine cannot satisfy the resource requirement of the job, then it picks the next fastest and so on.

Algorithm 2 Performance-aware machine placement for scheduled jobs using the machine learning model to predict relative performance.

Input: $j \leftarrow$ Job to schedule
$i \leftarrow$ Index of j in queue
$M \leftarrow$ Set of machines used for multi-resource scheduling

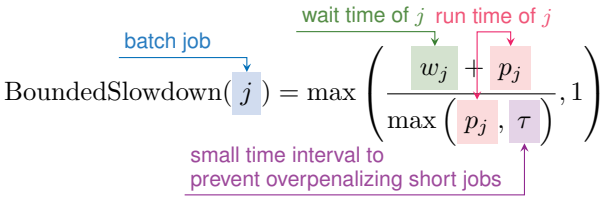
function $\texttt{Machine}_{\texttt{Greedy}}(j,\ i,\ M)$
  $rpv \leftarrow \texttt{Model}(j)$
  $m \leftarrow \text{argmax}_{i \in M}\ rpv$
  if all $i \in M$ are full **then**
    return m
  else
    $M' \leftarrow M$
    while m is full **do**
      $M' \leftarrow M' \setminus \{m\}$
      $m \leftarrow \text{argmax}_{i \in M'}\ rpv$
  return m

We implement this scheduling simulation in Python using our data set to get run time information for jobs. The nodes available on each machine reflect the number available on the actual machines. This is not meant to substitute rigorous scheduling simulation studies but only to demonstrate a potential use case.

A. Evaluation Metrics

When evaluating the efficiency of our scheduling algorithm we are concerned with performance from the perspective of individual jobs as well as the scheduler as a whole. Users will hope to see a faster turnaround time from job submission to completion for their jobs, while system administrators may look at the job throughput of a given scheduler to measure its performance. To quantify both of these we use *average bounded slowdown* and *makespan*.

The average bounded slowdown represents the average slowdown of a set of jobs with a fixed bound to prevent overpenalizing very short jobs. Slowdown is the ratio of submission-to-completion time with a wait time versus without a wait time. This provides a per-job evaluation metric to see how much each job is affected by the scheduling algorithm. The bounded slowdown can be calculated as shown below.



$\tau = 10$ in our evaluation. Using the function below we can compute the average bounded slowdown over a set of jobs J.

$$\overline{\text{BoundedSlowdown}}(J) = \frac{1}{|J|}\sum_{j \in J} \text{BoundedSlowdown}(j)$$

(set of jobs)

Using the same set of jobs J we can also define the makespan as the time from the first job submission to the time when the last job finishes. This measures the amount of time it takes for a scheduler to complete a set of work and is commonly used to compared different scheduling algorithms over fixed workloads.

$$\texttt{makespan}(J) = \left(\max_{j \in J}\left(w_j + r_j + p_j\right)\right) - \left(\min_{j \in J} r_j\right)$$

(wait time of j; start time of j; set of jobs; duration of j)

Both of these metrics are computed for each scheduling algorithm across our workload. We compare them between all the scheduling algorithms to observe the benefit from cross architecture performance modeling. For each metric a lower value indicates better performance.

VIII. Results

In this section we present the results from our training and scheduling experiments.

A. Evaluation of ML Models

Figures 2 and 3 show the mean absolute error and same-order-score of each model on the testing data set. We see that XGBoost performs the best for both of these metrics.

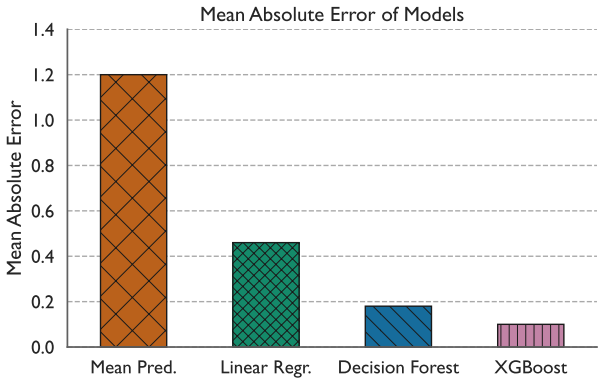


Fig. 2. The MAE of each machine learning model over the testing data set after training. XGBoost outperforms the other models with an MAE of 0.11. Lower MAE is better.

The XGBoost model scores a MAE of 0.11 (see Figure 2). This means that the model can take counters recorded on one architecture and predict its relative performance to the others within 0.11 on average. This is a 81.6% improvement over guessing the mean relative performance vector from the data. From this we can infer that the model is not simply guessing according to the distribution of the runtime data, but is rather correlating counter data with its performance prediction.

The linear and decision forest models perform better than guessing the mean, but do not exceed the MAE of XGBoost. The decision forest scores the closest to XGBoost likely since they are both ensembles of decision trees. However, XGBoost implements boosting alongside a number of other pruning techniques that strengthen its prediction.

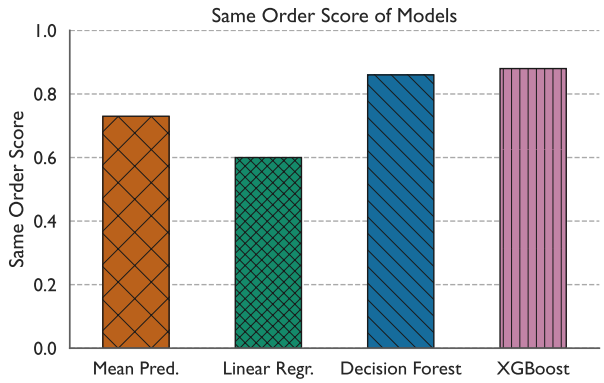


Fig. 3. The SOS of each machine learning model over the testing data set after training. Higher SOS is better.

We see similar performance from XGBoost on the SOS metric where it is the best model (see Figure 3). It is able to predict the relative performance vector in the correct architecture order in 76% of samples in the testing set. This means it is frequently able to predict the fastest and slowest architectures for a particular application and input, which is a valuable result to a user who is likely trying to avoid the slowest architecture and run on the fastest. Additionally, if the system with the fastest architecture is busy, then the user can select the next fastest and so on.

As with the MAE metric the decision forest has similar, but lower performance to XGBoost. The linear model is next as with MAE. It scores slightly higher than the SOS from guessing only the mean relative performance vector.

B. Ablation Study

Here we study the effects on modeling performance when removing certain features and/or data from the training set. Figures 4 and 5 further detail how well the models perform when given counters from each individual architecture. In both of these figures the ”mean” prediction row is constant, since the mean relative performance vector is independent of the input features. The first, Figure 4, shows the MAE scores for each ML model. We observe the same trends as Figure 2 where XGBoost has the best MAE. However, we notice that profiles from Ruby lead to a lower MAE and, thus, better predicted relative performance vectors. In fact, profiles from the two CPU systems, Ruby and Quartz, generally leader to better MAE. This same trend continues for the SOS metric.

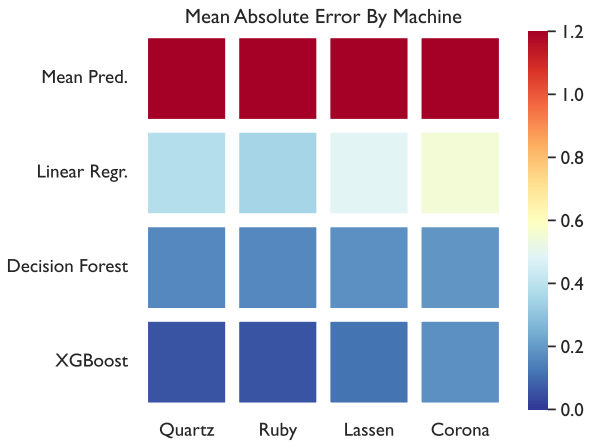


Fig. 4. The MAE of each model when predicting using profiles from one particular machine. For instance, the bottom right of the plot represents the MAE when predicting relative performance vectors with XGBoost and profiles from Ruby.

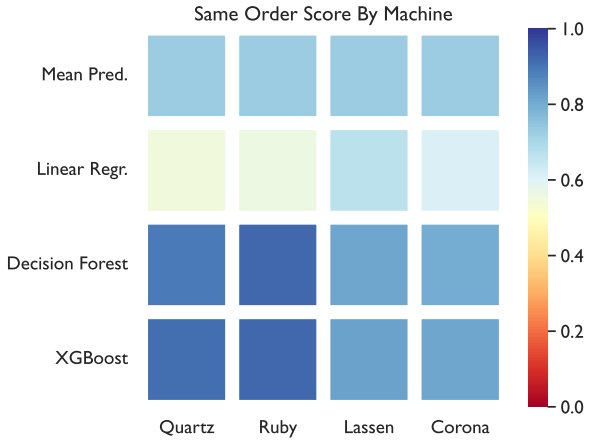


Fig. 5. The SOS of each model when predicting using profiles from one particular machine.

The fact that counters recorded on CPU machines lead to better predictions on average is an important observation for using this model in practice. CPU machines are generally cheaper and more readily available. Users can run their code on them and get predictions from the model for less available resources, such as GPUs. Additionally, users can obtain an estimate of the speedup from running on a given architecture without actually being capable of running on that architecture. For instance, if a particular application does not support AMD GPUs a user could estimate the performance increase/decrease if they were to implement AMD GPU support.

We hypothesize that the CPU performance metrics give better predictions due to the maturity of CPU performance counters and the profiling tools used to record them. CPU performance counters have been used extensively and the difficulties in recording them accurately have been well studied. On the other hand, GPU profiling, particularly for AMD, is a relatively new feature in HPCToolkit and the counters may not be as reliable as those recorded on a CPU.

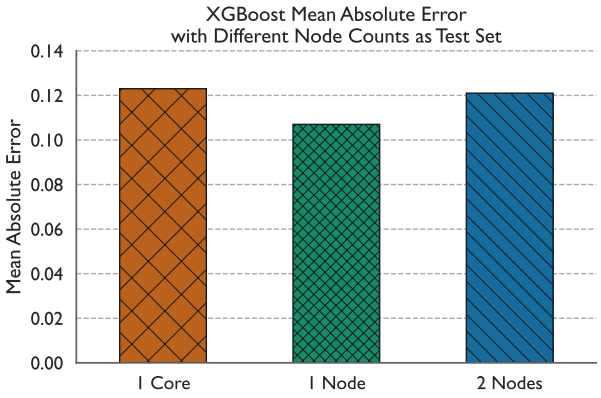


Fig. 6. Evaluation MAE of XGBoost when each resource count is removed from the training set and used for evaluation. The model performs best at predicting 1 node performance when trained on 1 core and 2 node date. Note that all scores are lower and still very strong.

Figure 6 shows the performance of XGBoost when trained on data from two of the three resources amounts and evaluated on the third. We observe that predicting the one node relative performance vectors gives the best MAE. It is unclear whether this is because modeling the one node performance is easier or that the one core and two node data is more representative. Regardless, all three node counts score very close to 0.11 MAE, which is still a strong result.

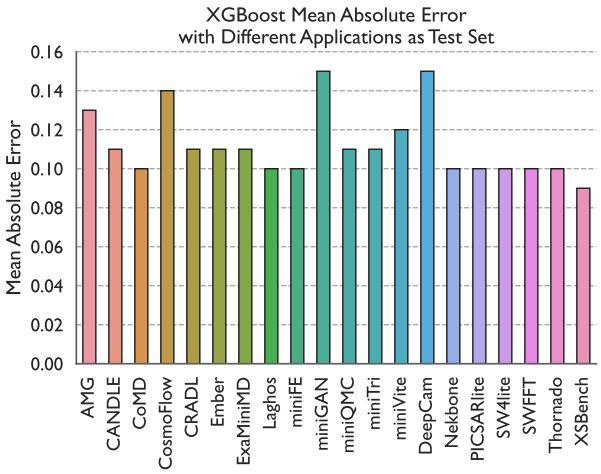


Fig. 7. Evaluation MAE of XGBoost when each application is removed from the training set and used for evaluation. Results are generally strong across all applications.

Additionally, we can see the performance of XGBoost when trained on all but one application and evaluated on the remaining applications in Figure 7. Again, we see that the model performs well across all applications. However, it does notably

perform worse for the ML and Python-based applications. This is possibly due to more noise and/or complicated software stacks involved in running each of these applications. These applications also tend to depend on more libraries and have more dependencies than the other applications.

C. Feature Importances

Figure 8 shows the feature importances for the XGBoost model. The most important feature is the ratio of branch to total instructions. This feature captures the control flow complexity of a program as those with more branch instructions have a more complex control flow. Since programs with more control flow generally perform worse on GPUs, the model likely uses this feature to make CPU-GPU predictions.

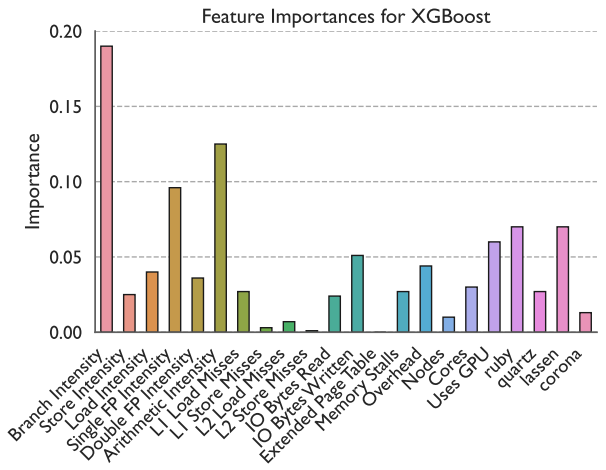


Fig. 8. Importances of each feature in the XGBoost model. A higher feature importance value means it is more influential in the decision making of the model. The branch instructions intensity is the most important feature followed by the integer and floating point arithmetic intensity.

Next we see that the ratio of integer and single precision FP arithmetic to total instructions are the next most important features in prediction. These provide insight into the data throughput of the model. In this case, applications with higher data intensity are more likely to perform better on the GPU as they are designed for high throughput data-parallel computation. These two features combined with the branching intensity make sense as the three most important features as they help the model predict relative performance between CPUs and GPUs, which is where we see the largest performance differences in the data.

The next three most important features are Ruby, Lassen, and Uses GPU, which detail where the counters were collected. This is necessary for the model to predict the relative performance vector and is likely why these are the next three most important features. We also see that the L2 store misses and extended page table features are not used in the prediction, so we can remove these during feature selection.

D. Evaluation of Scheduling Simulations

Figures 9 and 10 show the results from the scheduling simulation. The first, Figure 9, lists the makespan for the scheduler with each machine placement algorithm. The Model-based machine placement method gives the lowest makespan at 0.87 hours on the set of jobs meaning it is able to finish the job set in a shorter amount of time than the others. Placing jobs on the most efficient resource helps improve the makespan by allowing jobs to finish sooner. The next best method is the User+RR placement algorithm. This method represents how users submit jobs to the scheduler with only the limited knowledge of the performance of their applications across machines. This is followed by the Round-Robin and Random placement methods that perform the worst.

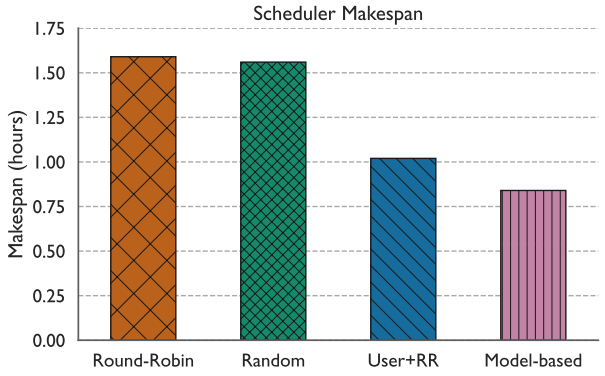


Fig. 9. The makespan of each machine selection algorithm in the scheduling simulation. Lower is better.

Figure 10 shows the average bounded-slowdown for each machine placement method. The slowdown measures the ratio of wait time and run time to just run time. As with makespan the Model-based placement performances the best compared to the other algorithms.

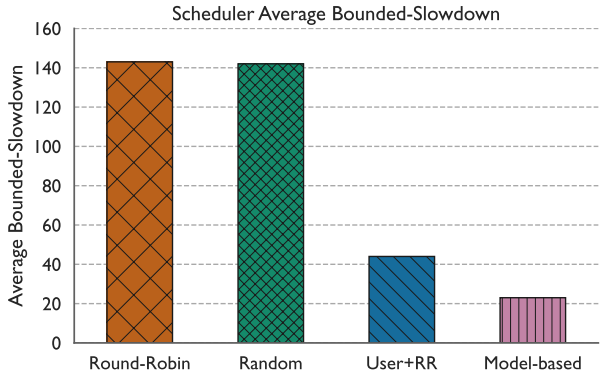


Fig. 10. The average bounded-slowdown of each machine selection algorithm in the scheduling simulation. Lower is better.

IX. Conclusion

The convergence of traditional HPC and new simulation, analysis, and data-science approaches provides unprecedented

opportunities for scientific discovery, but also creates workflows that are more complex than ever before. These workflows often combine many applications with vastly different performance requirements that are best handled by certain types of computing hardware. Meanwhile, HPC centers and cloud platforms offer various types of computing resources to satisfy diverse needs. In this work, we study one of the many capabilities workflow users need to effectively utilize such resources: cross-platform performance modeling. We collect a dataset of hardware counters across several different architectures for numerous scientific applications.We create expert derived features from these counters and train a machine learning model to predict relative performance vectors across a set of architectures with a MAE of 0.11. We further showcase how this can be used to efficiently schedule jobs across a heterogenous set of resources.

ACKNOWLEDGMENT

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under Contract DE-AC52-07NA27344 (LLNL-CONF-855652). This work was supported in part by LLNL LDRD projects 23-ERD-045 and 24-SI-005.

REFERENCES

[1] H. I. e. a. Ingólfsson, "Machine learning-driven multiscale modeling reveals lipid-dependent dynamics of ras signaling proteins." in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 119,1, 2022.

[2] D. H. Ahn, X. Zhang, J. Mast, S. Herbein, F. Di Natale, D. Kirshner, S. A. Jacobs, I. Karlin, D. J. Milroy, B. De Supinski, B. Van Essen, J. Allen, and F. C. Lightstone, "Scalable composition and analysis techniques for massive scientific workflows," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 32–43.

[3] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[4] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623

[5] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, J. Koning, T. Patki, T. R. W. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," in *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2018, pp. 10–19.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[7] O. Cankur and A. Bhatele, "Comparative evaluation of call graph generation by profiling tools," in *High Performance Computing*, A.-L. Varbanescu, A. Bhatele, P. Luszczek, and B. Marc, Eds. Cham: Springer International Publishing, 2022, pp. 213–232.

[8] A. Bergel, A. Bhatele, D. Boehme, P. Gralka, K. Griffin, M.-A. Hermanns, D. Okanovic, O. Pearce, and T. Vierjahn, "Visual analytics challenges in analyzing calling context trees," in *Programming and Performance Visualization Tools*, ser. Lecture Notes in Computer Science, vol. 11027, Apr. 2019. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-17872-7_14

[9] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Nov. 2019, lLNL-CONF-772402. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356219

[10] L. Zhou, X. Zhang, W. Yang, Y. Han, F. Wang, Y. Wu, and J. Yu, "Prep: Predicting job runtime with job running path on supercomputers," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3473521

[11] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer, "Prionn: Predicting runtime and io using neural networks," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225091

[12] D. Nichols, A. Marathe, K. Shoga, T. Gamblin, and A. Bhatele, "Resource utilization aware job scheduling to mitigate performance variability," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '22. IEEE Computer Society, May 2022.

[13] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Predictive modeling for job power consumption in hpc systems," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 181–199.

[14] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 33–44.

[15] W. Zhou, J. Zhang, J. Sun, and G. Sun, "Using small-scale history data to predict large-scale performance of hpc application," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 787–795.

[16] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting mpi collective communication performance using machine learning," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 259–269.

[17] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning parameter choices using bayesian optimization," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '20. IEEE Computer Society, May 2020.

[18] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018.

[19] Y. Cho, J. W. Demmel, J. King, X. S. Li, Y. Liu, and H. Luo, "Harnessing the crowd for autotuning high-performance computing applications," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 635–645.

[20] M. R. Wyatt, S. Herbein, K. Shoga, T. Gamblin, and M. Taufer, "Canario: Sounding the alarm on io-related performance degradation," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 73–83.

[21] N. Ardalani, U. Thakker, A. Albarghouthi, and K. Sankaralingam, "A static analysis-based cross-architecture performance prediction using machine learning," 2019.

[22] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 725–737. [Online]. Available: https://doi.org/10.1145/2830772.2830780

[23] L. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 40–40.

[24] L. L. Nesi, L. M. Schnorr, and A. Legrand, "Multi-phase task-based hpc applications: Quickly learning how to run fast," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 357–367.

[25] B. Tovar, B. Lyons, K. Mohrman, B. Sly-Delgado, K. Lannon, and D. Thain, "Dynamic task shaping for high throughput data

analysis applications in high energy physics," *IPDPS International Parallel and Distributed Processing Symposium*. [Online]. Available: https://par.nsf.gov/biblio/10356916

[26] "Ecp proxy applications," https://proxyapps.exascaleproject.org/, accessed: 2023-09-30.

[27] "The extreme-scale scientific software stack," https://e4s-project.github.io/index.html, accessed: 2023-09-30.

[28] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[29] "Cupti," accessed: 2023-09-30. [Online]. Available: https://docs.nvidia.com/cuda/cupti/index.html

[30] "rocprofiler," accessed: 2023-09-30. [Online]. Available: https://rocm.docs.amd.com/projects/rocprofiler/en/latest/rocprof.html

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[32] J. Lelong, V. Reis, and D. Trystram, "Tuning easy-backfilling queues," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and N. Desai, Eds. Cham: Springer International Publishing, 2018, pp. 43–61.