A Machine Learning Framework for Performance Coverage Analysis of Proxy Applications

Tanzima Z. Islam, Jayaraman J. Thiagarajan, Abhinav Bhatele, Martin Schulz, Todd Gamblin
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551
E-mail: {tanzima, jayaramanthi1, bhatele, schulzm, tgamblin}@llnl.gov

***Abstract*—Proxy applications are written to represent subsets of performance behaviors of larger, and more complex applications that often have distribution restrictions. They enable easy evaluation of these behaviors across systems, e.g., for procurement or co-design purposes. However, the intended correlation between the performance behaviors of proxy applications and their parent codes is often based solely on the developer's intuition. In this paper, we present novel machine learning techniques to methodically quantify the coverage of performance behaviors of parent codes by their proxy applications. We have developed a framework, VERITAS, to answer these questions in the context of on-node performance: a) which hardware resources are covered by a proxy application and how well, and b) which resources are important, but not covered. We present our techniques in the context of two benchmarks, STREAM and DGEMM, and two production applications, OpenMC and CMTnek, and their respective proxy applications.**

***Index Terms*—Machine learning, Unsupervised learning, Performance analysis, Scalability**

I. INTRODUCTION

As we move towards exascale, it has become important to take application characteristics into account when designing the hardware architecture and vice versa. This approach requires an agile co-design loop with hardware architects, system software developers and domain scientists working together to make informed decisions about features and tradeoffs in the design of applications, algorithms, the underlying system software, and hardware.

Ideally, all co-design efforts would be driven by performance measurements directly based on the targeted production applications. However, those applications are often too large or too complex to setup to be used in early design studies, or have distribution restrictions. On the other hand, traditional benchmark suites such as NAS [1] cover a number of popular parallel algorithms but do not include modern adaptive methods such as Monte Carlo or discrete ordinates. This has led to the widespread development of proxy applications to better characterize the performance of complex production applications [2], [3], [4], [5].

Proxy applications are typically developed to capture specific performance-critical modules in a production application. By retaining the parent application's behavior, they offer convenience and flexibility to analyze performance without requiring the time, effort and expertise to port or modify production codes. Typically, the proxy emulates certain computational aspects of the parent such as specific implementations of an algorithm or performance aspects such as memory access patterns. Proxy applications are comparatively small in size, easy to understand, and are typically publicly available for the community to use. For example, XSBENCH is a proxy application for OPENMC that implements a number of random indirect array lookup operations to compute neutron cross-section on a one-dimensional array [3]. However, the corresponding kernel in OPENMC implements a number of additional operations to compute these indices [6]. This poses the following questions – (a) do the two applications have the same performance behavior on a target architecture? and (b) which performance behaviors are different between them?

Given the important role proxy applications play in the co-design process, it is critical to understand which salient performance characteristics of a parent application are covered by a proxy and how well. Unfortunately, the notion of "coverage" is currently highly subjective, and the strategies such as linear correlation and Principal Component Analysis (PCA) adopted for such comparative analysis are ineffective. To the best of our knowledge, this paper is the first to present a principled, machine learning approach that methodically quantifies the quality of a match.

We introduce novel machine learning techniques to identify important performance characteristics of applications and quantify the coverage provided by a proxy application compared to its parent. Our approach adopts ideas from sparse learning theory to identify performance metrics that can describe the performance characteristics (e.g., efficiency loss) of an application. Further, we define two new metrics – 1. a new quality metric, *Resource Significance Measure*, to measure the significance of hardware resources in predicting application performance, computed by accumulating the beliefs from each of the constituent metrics in the learned sparse model; and 2. a *Coverage* metric to indicate the quality of a match between the resource utilization behavior of a proxy and its parent application. Note that instead of aggregating pairwise correlations between the individual metrics, our approach constructs subspace models for both proxy and parent using all metrics corresponding to a hardware resource and estimates how well the models agree. In addition to being robust, this provides a principled way to compare multiple metrics simultaneously. We implement these methodologies in VERITAS, a machine learning framework for comparative performance analysis.

We focus on on-node performance behaviors of applications to tackle the increasing node complexity on current and upcoming systems. However, similar analysis could be applied to

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

off-node communication or other performance characteristics. We apply VERITAS to five different kernels implemented in three proxy applications. These proxy applications are developed to capture on-node performance characteristics of two production codes in the area of nuclear reactor design – OPENMC, a Monte Carlo particle transport simulation code focused on neutron criticality calculations [6], and CMTNEK, a spectral element code covering thermal hydraulics [7]. We analyze these kernels on two different architectures – IBM Blue Gene/Q (BG/Q) and Intel Xeon.

In summary, the main contributions of our work are:

- We define a novel high-level metric, "Resource Significance Measure (RSM)" and present a machine learning technique for computing RSM in order to identify which high-level resource utilizations cause performance loss in applications as we scale.
- We define a novel metric, "Coverage" and present a machine learning technique for computing Coverage to quantify how well performance behaviors match between proxy and their parent applications.
- We implement a framework, VERITAS, that presents RSM and Coverage in an easy to understand format, and reports individual performance metrics that contribute significantly to performance loss of applications.

The remainder of this paper is organized as follows. Section II provides the background on machine learning techniques applied in this paper. Section III explains the machine learning techniques we have developed for performance coverage analysis. Section IV presents the implementation of the validation approach in VERITAS. Section V explains the experimental setup. Section VI presents validation results for three applications on two different architectures. Section VII reviews related work and Section VIII concludes the paper.

II. BACKGROUND

In this section, we briefly review the mathematical preliminaries for the development of our coverage analysis approach.

A. Feature Selection using Sparse Coding

We pose the task of identifying metrics that predict the performance attribute (e.g., runtime, parallel efficiency) of applications as a feature selection problem. Due to the large number of performance metrics that can be collected, the task of identifying the salient ones that are actually relevant to the performance of applications can be daunting. The feature selection process filters the number of attributes that application developers need to investigate in order to diagnose the cause of performance loss. Though there is a broad spectrum of feature selection techniques, sparse learning has recently emerged as a powerful tool to obtain models with high degree of interpretability from high-dimensional data [8]. At a high level, sparse learning is a regression technique with sparsity imposed, which improves regularization and robustness to noisy data. There is extensive literature on sparse machine learning, with terms such as compressed sensing [9], Lasso regression and convex optimization [10] associated with the field. Several successful applications of sparse learning have been reported in image/signal processing and data analysis [11].

B. Evidence Computation

Our analysis is based on understanding the impact of different hardware counters on performance. The number of counters in a typical system can be quite large, which makes it challenging for the analyst to interpret the results from even a sparse model. In order to alleviate these challenges, we categorize these hardware counters into a small number of high-level groups (details can be found in Section III) and quantify the amount of information each group provides in predicting the performance attribute. In particular, we utilize a well-known statistical inferencing framework, Dempster-Shafer theory (DST) [12], to estimate the degree of belief for each hardware counter and then accumulate their beliefs to obtain a measure of significance for each counter group. Let Θ be the universal set of all hypotheses, i.e., the set of all hardware counters in our case, and 2^Θ be its power set. A probability mass can be assigned to every hypothesis $H \in 2^\Theta$ such that,

$$\mu(\varnothing) = 0, \sum_{H \in 2^\Theta} \mu(H) = 1 \tag{1}$$

where $\varnothing$ denotes the empty set. This measure provides the confidence that hypothesis H is true. Using DST, we can compute the uncertainty of the hardware counters in predicting a performance attribute using the belief function, bel$(.)$, which is the confidence on that hypothesis being supported by strong evidence. The belief function is defined as

$$\text{bel}(H) = \sum_{B \subseteq H} \mu(B) \tag{2}$$

We develop a new evidence measure by accumulating the beliefs within a group, using Dempster combination rules.

C. Subspace Analysis

The problem of comparing the characteristics of two applications can be viewed as the comparison of the feature spaces, i.e. counters corresponding to each resource group. In statistical modeling, it is common to assume that data can be effectively embedded in low-dimensional linear subspaces, i.e. the underlying data distribution can be described using fewer degrees of freedom. For example, PCA determines the directions of maximal variance and projects the data onto that subspace. The resulting subspaces often emphasize the salient structure in data while rejecting noise and outlier data. Consequently, instead of comparing the high-dimensional data, we compare their corresponding low-dimensional representations for robust analysis. A linear subspace can be conveniently represented using its basis $\mathbf{B} \in \mathbb{R}^{C \times d}$, where C is the dimensionality of the data, and d is the dimensionality of the subspace. The collection of all d-dimensional linear subspaces forms the Grassmannian $G(d, C)$, a smooth Riemannian manifold, which allows effective geometric and statistical analysis of subspaces. Based on this idea, we introduce a novel and

computationally efficient similarity measure for comparing two subspaces.

III. Methodology

In this section, we describe the proposed methodology for comparing the performance characteristics of an application and its proxy. More specifically, we view the problem of comparative analysis as measuring how closely the data subspaces of two applications match.

A. Metrics and Resources

The first step is to select the metrics for our comparative analysis. Since we focus on on-node performance, which is typically constrained by hardware bottlenecks on the processors or in the memory system, we rely on hardware performance counters in this work. In order to identify salient performance characteristics of applications at a granularity that is intuitive, we first organize all input metrics into high-level semantic groups using a suitable heuristic. In particular, for this work we group the hardware counters based on which hardware components (e.g., memory, L1, L2) they monitor [13], [14]. This enables users to directly identify the high-level hardware resources where scalability bottlenecks could appear. In this paper, we refer to these high-level groups as "resource groups" or simply "resources".

Table I presents the names of the resource groups for the two architectures we use in our evaluation. We categorize counters into resource groups in two steps. First, we use a static metrics-to-resource group mapping to categorize counters. The static map file is useful when users want to assign counters to resources based on their understanding of the hardware or when counter names themselves do not offer clear information about which resources they belong to. Second, we classify the rest of the counters based on matching resource group names as substrings of counter names. The substring matching approach is based on the rationale that PAPI [15] annotates hardware counter names with their associated resource group names.

TABLE I: Resource groups on Blue Gene/Q and Xeon

Resource Group	Blue Gene/Q	Xeon
Instruction unit	IU	-
Floating point unit	AXU	FP
Execution unit	XU	-
Branch unit	BR	BR
Load/store unit	LSU	-
Prefetch events	-	PREFETCH
L1 prefetcher	L1P	-
L1 cache	L1	L1
L2 cache	L2	L2
Last level L3 cache	-	L3
Translation look aside buffer	TLB	TLB
Memory	MEM	MEM

The set of performance metrics corresponding to the application and its proxy are denoted by the matrices $\mathbf{Y} \in \mathbb{R}^{C \times N}$ and $\mathbf{Z} \in \mathbb{R}^{C \times M}$ respectively. Here, C indicates the set of performance metrics considered for analysis, while the variables N and M denote the total number of runs for the two applications using different configurations (e.g., by using different numbers of threads and workloads). The performance-metric-to-resource-mapping, specific to the system architecture, is denoted by $\mathbf{M} \in \mathbb{R}^{C \times R}$ where R is the total number of resources.

B. Approach and Target Attribute

With the resources defined, our approach has two main steps: (a) determining the significance of each resource in predicting a target performance attribute of an application, and (b) measuring the resource level compatibility between the application and its proxy. While the former step enables the designer to identify salient performance characteristics (e.g. where bottlenecks appear) in the application, the latter step reveals aspects in the proxy that are consistent with the target application.

In this paper, we use parallel efficiency loss (Equation (3)) for on-node strong scaling as the target performance attribute. The rationale for this choice is that efficiency loss directly impacts the final time to solution, the metric users care most about, yet allows for easy relative comparisons and normalization across applications and systems. Efficiency loss typically increases as an application uses more cores on a multi-core machine, and since every performance metric has a non-zero penalty, we want to identify those that follow the same growth pattern as efficiency loss.

$$\text{efficiency\_loss}(p) = 1 - \frac{T(1)}{(T(p) \times p)} \tag{3}$$

where p = number of tasks (threads or processes) and $T(p)$ = execution time when using p tasks.

C. Computing Resource Significance

Finding important metrics that are predictive of a target attribute is essentially a feature selection problem. Existing work such as [16] applies regression models to different feature subsets and adopts heuristic measures such as the R^2 statistic to identify the minimal subset that leads to the best prediction. However, this can be ineffective in the presence of noise in the data, when the dataset is limited, or in cases of model overfitting.

A natural approach to understanding the effect of a feature (performance metric in our case) on a target attribute (i.e., efficiency loss) is to measure their correlation properties. However, this approach has some fundamental shortcomings. *First,* a strong correlation does not imply causation. *Second,* this analysis is carried out with one predictor variable at a time, and hence inferring a group of metrics that are jointly correlated to the target attribute is not straightforward. This issue can be alleviated to an extent by applying linear transformations to the data using techniques such as Principal Component Analysis (PCA) or Canonical Correlation Analysis (CCA). However, these methods create new, complex features by linearly combining the original features, and hence make interpretation of the results difficult.

In contrast, we measure the significance of resources in predicting a target performance attribute (e.g., efficiency loss) by building a linear predictive model based on the collected metrics. In particular, we adopt the idea of *sparse representations* to identify performance metrics that can effectively describe the scaling behavior of an application. Using the inferred sparse model, we first derive per-metric beliefs in causing the observed performance behavior, and then aggregate those beliefs to estimate per-resource significance. The latter step allows us to understand which hardware resources impact the scalability of applications and which of these characteristics are covered by a proxy or not.

Identifying Salient Performance Metrics: The underlying assumption in sparse representations is that the data is drawn from a union of low-dimensional subspaces, which need not be completely disjoint or orthogonal. In other words, a complex pattern can be effectively decomposed into a small set of diverse, elementary patterns. Mathematically, this assumption is expressed as follows: given a target (efficiency loss), $\mathbf{t} \in \mathbb{R}^N$, where N is the number of runs, and a dictionary of representative patterns, $\mathbf{D} \in \mathbb{R}^{N \times C}$, where C is the set of performance metrics, the sparse representation, $\mathbf{a} \in \mathbb{R}^{C \times 1}$ can be obtained as

$$\min_{\mathbf{a}} \|\mathbf{t} - \mathbf{D}\mathbf{a}\|_2^2 \text{ s.t. } \|\mathbf{a}\|_0 \leq \kappa \tag{4}$$

where $\|.\|_0$ denotes the ℓ_0 norm that counts the total number of non-zero entries in a vector, $\|.\|_2$ is the ℓ_2 norm, and κ is the desired sparsity. In our case, the dictionary $\mathbf{D} = \mathbf{Y}^T$ is the collection of metrics (C). This optimization problem solves for the sparsest set of performance metrics that can predict $\mathbf{t}$. Exact determination of the sparsest representation using Equation (4) is an NP-hard problem [17], and hence it is common to adopt greedy pursuit techniques to solve it [18]. Greedy procedures for computing sparse representations operate by choosing the dictionary element that is most strongly correlated to the $\mathbf{t}$, remove its contribution and iterate. In effect, they make a sequence of locally optimal choices in an effort to approximate the globally optimal solution. In particular, we use the *Orthogonal Matching Pursuit* (OMP) algorithm [19], which ensures that the residual vector computed in each iteration is orthogonal to the dictionary elements picked so far, and picks the smallest, diverse subset of metrics to describe the performance loss.

Choosing Metrics with Similar Behavior: Because of the sparsity constraint, expressed in Equation (4), two different metrics that have similar patterns might not be picked together. However, identifying related metrics, in particular if they correspond to different hardware resources, is crucial to our analysis. To overcome this challenge, we introduce a new version of the *OMP* algorithm, referred to as the *Ensemble OMP*, which creates a plurality of sparse representations by randomizing the greedy selection procedure in each step of *OMP*. More specifically, instead of choosing the most correlated dictionary element in each step, we select τ most correlated metrics and create a discrete distribution based on their correlations, i.e., the higher the correlation, the higher probability to be chosen in that step. We randomly pick a metric using the discrete distribution, compute the residual, and repeat this process until the desired sparsity κ is met or the model achieves sufficient fidelity. We repeat this randomized algorithm T times independently (set to $5,000$ in our experiments), and similar to existing ensemble approaches [20], the final representation is obtained by averaging the solutions in the ensemble. The iterative algorithm can automatically stop augmenting the model when the metrics cannot improve the model fidelity any further or the desired κ is reached.

Resource Significance Measure: The final step in our algorithm is to use the average representation from the ensemble to estimate metric-wise belief, and subsequently evaluate the significance of their corresponding hardware resource groups. The belief for each of the metrics, if they have a non-zero coefficient in the sparse representation, can be obtained as the maximum likelihood estimate, where the likelihood probability is inversely proportional to the reconstruction error. These estimates can be propagated to their corresponding resources by obtaining the compound evidence [12]. The steps for computing Resource Significance Measure are listed in Algorithm 1.

Algorithm 1 Compute the significance measure for a given resource group r using the desired target attribute $\mathbf{t}$.

1) **Input**: desired sparsity κ, ensemble size T and ensemble parameter τ.
2) **Initialize**: $i = 1$.
3) Construct the dictionary matrix $\mathbf{D} = \mathbf{Y}^T$ and normalize the columns to unit ℓ_2 norm.
4) While $i \leq T$:
   a) Initialize $\Omega = \emptyset$, $\mathbf{r} = \mathbf{t}$, Loop index $l = 1$.
   b) While **Stopping Criterion** is not met:
      i) Construct the discrete probability distribution $\mathcal{P}$ using the τ largest elements in $\mathbf{r}^T\mathbf{D}$.
      ii) Randomly choose an index k_l using $\mathcal{P}$ and update the index set $\Omega \leftarrow \Omega \cup k_l$.
      iii) Compute the coefficient vector $\mathbf{a}_i$ for the index set Ω using least squares.
      iv) Compute $\mathbf{r} \leftarrow \mathbf{t} - \sum_j^{|\Omega|} \mathbf{a}_i[k_j]\mathbf{d}_{k_j}$.
      v) $l \leftarrow l + 1$.
   c) $i \leftarrow i + 1$.
5) Using the ensemble representations $\{\mathbf{a}_i\}_{i=1}^T$, compute the average representation $\mathbf{a} = \frac{1}{T}\sum_i \mathbf{a}_i$.
6) Estimate the significance measure RSM_r using Equation (7).

For a metric, indexed by j, which has a non-zero coefficient value, i.e., $a_j \neq 0$, its belief is inversely proportional to the reconstruction error. In particular, we compute it as

$$\boldsymbol{\alpha}_j = \exp\left(-\gamma \|\mathbf{t} - \mathbf{d}_j a_j\|_2^2\right) \tag{5}$$

Here, the parameter $\gamma > 0$ is the (inverse) width of the Gaussian function. The larger the reconstruction error, the less information the metric j provides about the target attribute and hence its belief is lower. To pool the evidences of multiple metrics within a hardware resource, we use a belief propagation strategy similar to the one by Hegarat-Mascle [21]. Given two metrics i and j in a resource group r, we estimate the accumulated evidence $\boldsymbol{\alpha}_i \oplus \boldsymbol{\alpha}_j$ as

$$1 - (1 - \eta_0.\boldsymbol{\alpha}_i) \times (1 - \eta_0.\boldsymbol{\alpha}_j) \tag{6}$$

where $0 < \eta_0 \leq 1$ is the amount of belief we place on a metric j when the reconstruction error is zero (set to 1 in our experiments). Generalizing this, the significance for a resource r of a workload w can be computed as

$$RSM_r^w = 1 - \prod_{j \in L_r} (1 - \eta_0.\boldsymbol{\alpha}_j) \tag{7}$$

where L_r is the set of all metrics belonging to resource r with non-zero coefficient values. In order to easily interpret the relative significances of the different resources, we normalize the *RSM* corresponding to all resources between 0 and 1. In cases where we are interested in understanding the resource significance in describing performance characteristics of different workloads, we evaluate

$$RSM_r = \frac{1}{W} \sum_{w=1}^{W} RSM_r^w \tag{8}$$

Here, W denotes the total number of workloads considered, and RSM_r^w corresponds to the resource significance for the resource r and workload w.

D. Comparing Application and Proxy

In this section, we describe our approach for resource-wise comparative analysis between the parent and its proxy by measuring the compatibility between the two feature spaces defined by the set of performance metrics (C). More specifically, we build low-dimensional PCA subspaces for the two feature spaces, and construct the geodesic curve between them using it to estimate the dissimilarity (Figure 1). Note that, for a given resource group, the lower the resource subspace dissimilarity measure ($RSDM$), the higher is the compatibility between the two applications.

Subspace Construction: We begin by creating d-dimensional PCA subspaces $\mathbf{B}_{PA}$ and $\mathbf{B}_{PR}$ for the parent and proxy applications respectively. In order to estimate the dimension d of the PCA subspaces, we combine the two datasets and compute the joint PCA basis, $\mathbf{B}_{PA+PR}$. Intuitively, if the two applications are similar, then all three subspaces should not be too far away from each other on the Grassmannian. Formally, we measure

$$\mathcal{D}(d) = 0.5[sin\alpha_d + sin\beta_d] \tag{9}$$

where α_d and β_d denote the dth principal angle between the subspace pairs $\mathbf{B}_{PA}$ and $\mathbf{B}_{PA+PR}$, and $\mathbf{B}_{PR}$ and $\mathbf{B}_{PA+PR}$ respectively. When $\mathcal{D}(d)$ is small, it implies that the two subspaces are well-aligned. At its maximal value of 1, the two subspaces will be completely orthogonal, thereby indicating that the two applications are completely unrelated. Optimally, the dimensionality d should be high enough, while ensuring that the directions are not completely orthogonal. To identify the optimal d, we use a greedy strategy,

$$d^* = min\{d | \mathcal{D}(d) \geq \delta\} \tag{10}$$

In our implementation, we fixed $\delta = 0.4$, since in all data collected in our experiments, this led to an estimated d^* that captured more than 90% of the energy in both the individual feature spaces.

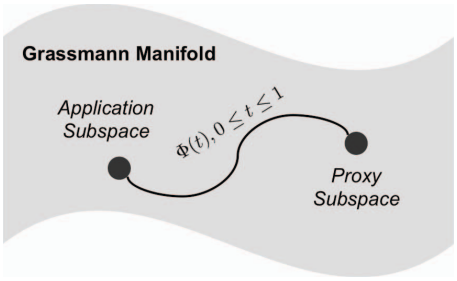

Fig. 1: Illustration of the geodesic flow between the application and proxy subspaces, on the Grassmannian, for a given resource. We use tools from Grassmann analysis to measure the compatibility between the two applications.

Algorithm 2 Compute the resource subspace dissimilarity measure for a given resource group r.

1) Compute the PCA subspaces for the parent data $\mathbf{Y}$, the proxy data $\mathbf{Z}$ and for the combined dataset $[\mathbf{Y}^T, \mathbf{Z}^T]^T$.
2) Estimate the optimal subspace dimension d^* using Equation (10) and retain only the top d^* components of the PCA subspaces $\mathbf{B}_{PA}$ and $\mathbf{B}_{PR}$.
3) Project data onto the corresponding PCA subspaces.
4) For each dimension $i \leq d^*$: Fit two 1D Gaussians $\mathcal{A}_i$ and $\mathcal{P}_i$ to the ith dimension of the parent and proxy subspaces, and compute the symmetricized KL-divergence between them as

$$\lambda_i = KL(\mathcal{A}_i || \mathcal{P}_i) + KL(\mathcal{P}_i || \mathcal{A}_i)$$

where

$$KL(\mathcal{A}_i || \mathcal{P}_i) = \sum_j \mathcal{A}_i(j) \log \frac{\mathcal{A}_i(j)}{\mathcal{P}_i(j)}$$

5) Compute the dissimilarity measure by averaging λ_i over all d^* dimensions, weighted by the corresponding principal angles:

$$RSDM_r = \frac{1}{d^*} \sum_i \theta_i \lambda_i$$

Measuring Dissimilarity: We consider two applications to be compatible with respect to a given resource group when (a)

the two subspaces are geometrically well-aligned, and (b) the data in the projected subspaces are similarly distributed. The steps involved in this algorithm are listed in Algorithm 2. Note that, the diagonal elements of the matrix $\mathbf{\Sigma}$ from the SVD of $\mathbf{B}_{PA}^T \mathbf{B}_{PR} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ correspond to $cos\theta_i$, where θ_i are the principal angles between $\mathbf{B}_{PA}$ and $\mathbf{B}_{PR}$. They measure the amount of overlap between the two subspaces. Similar to the previous case, we normalize the $RSDM$ measure between 0 and 1. Finally, in cases where there are multiple workloads, the dissimilarity measure is obtained as the average of the individual workloads. For easy interpretation of the results, we define the $Coverage$ measure for each resource r as

$$Coverage_r = 1 - RSDM_r, \forall r. \tag{11}$$

IV. The Veritas Framework

In this paper, we focus on on-node scalability of applications. Hence, we collect, analyze, and present metrics that are relevant to multi-core resources via hardware performance counters. Since our analysis methodology depends on the concept of explaining a target variable based on a dictionary of features, it does not depend on the types of metrics collected. We use an open-source per-thread hardware performance counter collection tool called perf-dump [22]. Perf-dump is based on PAPI [15], a library that provides a portable interface to access hardware performance counters. We collect both preset counters that are general across architectures and native counters that are hardware-specific. We categorize them into several high-level hardware resource groups (Table I) for two different architectures, Xeon and BG/Q based on the manufacturer's stated semantics. Although resource categorization is architecture-specific, we assert that this is a one-time operation and hence practical.

We implement the coverage analysis methodology presented in the previous section in a framework called Veritas. Figure 2 shows its overall workflow. In order to compare the performance behaviors of a proxy with its parent, we annotate corresponding code regions using perf-dump instead of the entire application. Once corresponding code regions are annotated and applications are compiled, we collect all hardware performance counters by executing them on the same machine. The framework implements three modules to process input, analyze data, and visualize the output of the analysis.

The `preprocessor` module in Veritas, written in Python, reads performance data collected by perf-dump, computes averages of the performance metrics, and generates a metrics-to-resource-map using the technique described at the end of this section. This map is later used by the `analysis` module for computing resource-level significance.

The `analysis` module of Veritas, implemented in Matlab, takes the event-to-resource-map along with the performance data collected for both applications and applies the methodology described in Section III for computing resource significance (RSM) and coverage for each resource group.

The `visualization` module generates three different pieces of information: (1) an overall coverage analysis output

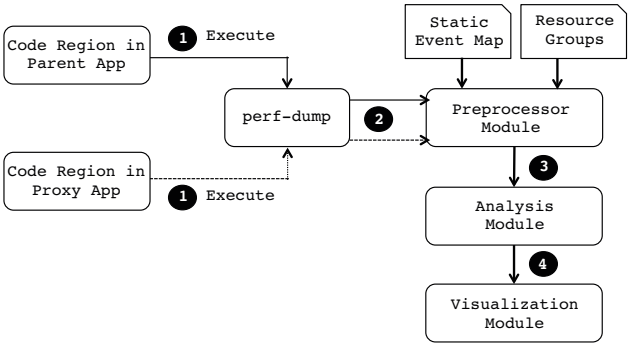

Fig. 2: Workflow showing the coverage analysis methodology as implemented in Veritas.

for each resource (e.g., Figure 6), (2) per-workload breakdown of coverage output (omitted due to space constraints), and (3) for each resource, hardware counters that have non-zero beliefs (α). As an example, Figure 6 presents the importance of resource groups for a parent-proxy pair along the X axis (between 0 and 1, where 0 means the lowest and 1 means the highest importance), and coverage along the Y axis (between 0 and 1, where 0 means "does not cover" and 1 means "covers exactly"). Coverage greater than 0.8 means that on average the proxy covers more than 80% of the utilization behavior of the parent for a certain resource.

RSM is a non-linear quantity, i.e., the summation of RSM scores across resources does not equal to 1. RSM computed by Veritas provides a relative ranking of resource importance in the order of their prediction power. A resource with RSM of 0.8 means that information about that resource alone can be used to predict the efficiency loss with 80% probability. A resource with RSM of 0.5 on the other hand indicates a 50% chance and cannot be used with confidence to predict the efficiency loss of an application. In our experience, coverage ≥ 0.8 and RSM ≥ 0.8 are sufficient to provide a realistic picture. However, others can adjust this user-defined threshold.

Static Event Map: We use the following rules to generate the static event map for the experiments presented in this paper: (1) events measuring any resource **access** or **hit** belong to the resource their names reflect (e.g., PAPI_L1_DCA or L1 data cache access is categorized in L1 cache); (2) events measuring cache **misses** are associated with the next level of cache since their latency is bound at least by the access cost of the next level cache (e.g., PAPI_L1_DCM or L1 data cache miss is categorized in L2); (3) TLB misses are categorized in TLB since different architectures resolve TLB misses differently depending on the memory needs of the application [23] (e.g., a TLB miss on BG/Q may or may not result in an L2 request, whereas on Xeon may result in a page walk that could hit any level of cache); (4) Stalls are attributed to the next level of hardware resource since their latency is bound by the latency of the resource they are waiting upon (e.g., events named PEVT_L1P_BAS_*_STALL_* indicate the L1 prefetcher is stalled for data to arrive at the L2

cache on BG/Q and hence are categorized in L2). Stall events such as RESOURCE_STALLS2:ALL_FL_EMPTY and PARTIAL_RAT_STALLS:FLAGS_MERGE_UOP represent contention during out-of-order execution on Xeon and could have resulted due to contention at either the core or the memory system. Since, the reason for these stall events could be more than one, we omit these two stalls from our analysis.

V. Experimental Setup

In this section, we describe the target platforms and applications used to analyze performance coverage.

A. Target Systems

We used VERITAS to analyze the coverage of five different kernels of three proxy applications on two different architectures: IBM Blue Gene/Q and Intel Xeon. We ran our experiments on Vulcan [24], a 24-rack, 5 Petaflop/s BG/Q system with $24,576$ nodes. Each compute node contains 16 GB of main memory and hosts a 64-bit PowerPC A2 with 16 cores at 1.6 GHz, each running up to 4 hardware threads. Each core has an L1 instruction and data cache of size 16 KB each, and an L1 prefetcher cache with 32×128 bytes. The L2 cache of 32 MB is shared among all cores and split into sixteen 2 MB slices. We ran the rest of the experiments on Cab [25], a 431 Tflop/s Intel Xeon cluster with $1,296$ nodes. Each node on Cab is a dual-socket Sandy Bridge processor with 32 GB of memory, 20 MB shared L3 cache across all eight cores on each socket. Each core has a private 256 KB L2 cache and 32 KB L1 data and instruction caches.

B. Parent and Proxy Applications

OPENMC is a Monte Carlo (MC) particle transport simulation code focused on neutron criticality calculations [6] for computer-based simulations of nuclear reactors. The application is written in FORTRAN with parallelism supported by a hybrid OpenMP/MPI model and is available online at [26]. It computes the path of a neutron as it travels through a nuclear reactor. MC methods are popular since they require few assumptions, resulting in highly accurate results given adequate statistical convergence.

XSBENCH [27] is a proxy for OPENMC that models the most computationally intensive part of a typical MC reactor core transport algorithm, the calculation of macroscopic neutron cross-sections, which accounts for around 85% of the total runtime of OPENMC [28]. XSBench retains the essential performance-related computational conditions and tasks of fully featured reactor core MC neutron transport codes, yet at a small fraction of the programming complexity of the full application. The application is written in C, with multicore parallelism support provided by OpenMP and is available online at [3].

RSBENCH is a proxy application similar in purpose to XSBENCH, but models an alternative method for calculating neutron cross-sections – the multipole method. This algorithm presented by Tramm et al. [29] compresses data into an abstract mathematical format. The basic idea of this method is to reduce the memory footprint of cross-section data and improve data locality at the cost of an increase in the number of computations required to reconstruct it. Since data movement is more expensive than computation, this method may provide significant performance improvements compared to the classical approach implemented in OPENMC and XSBENCH.

CMTNEK solves compressible Navier-Stokes equations for multiphase flows and is implemented in FORTRAN. The objective of this application is to perform high-fidelity, predictive simulations of particle laden explosively dispersed turbulent flows under conditions of extreme pressure and temperature. It builds on the highly scalable Nek5000, a Gordon Bell prize-winning, spectral element, computational fluid dynamics code developed at Argonne National Laboratory for simulating unsteady incompressible fluid flow with thermal and passive scalar transport [7].

CMTBONE [4] encapsulates the key data structures and computation operations of CMTNEK. CMTBONE is still under development and will eventually allow a variety of architecture evaluations and scalability studies. Additionally, the particle simulation capabilities present in CMTBONE will allow for the study and modeling of a variety of dynamic load balancing algorithms. The developers of CMTBONE are interested in monitoring and comparing three specific regions in the code. These are as follows:

`point_compute_kernel:` This is where the primitive variables are computed.

`compute_kernel:` This is where fluxes and spatial partial derivatives of the fluxes are computed. This is the key computation step in CMTNEK.

`comm_kernel:` Contrary to the name, this module currently only implements computation steps necessary for calculating face fluxes.

VI. Results

First, we present an empirical validation of the generated models using the STREAM [30] and DGEMM [31] benchmarks. Both STREAM and DGEMM were compiled using Intel compiler version 13.1.163. Following this, we present the coverage of five different kernels of three different proxy applications.

A. Model Validation

STREAM: The four computation kernels in the memory benchmark STREAM are described in Table II. We compiled the application with the flag "-nolib-inline" in order to ensure that the compiler did not replace the `COPY` kernel with the optimized "memcp" function. We ran the benchmark on Xeon with array size set to $50M$ elements, and ran each experiment 100 times. We collected all hardware performance counters on Xeon for each of these kernels and Figure 3 shows that VERITAS identifies memory subsystem (specifically, L3 for `ADD` and `TRIAD`, and memory for `COPY` and `SCALE`) to be the

TABLE II: Basic operations in the STREAM benchmark. The variables `a`, `b`, and `c` are vectors of floating-point numbers. The variable `s` is a scalar floating-point value.

Name	Code
ADD	`a[i] = b[i]+c[i];`
SCALE	`a[i] = s*b[i];`
COPY	`a[i] = b[i];`
TRIAD	`a[i] = b[i]+s*c[i];`

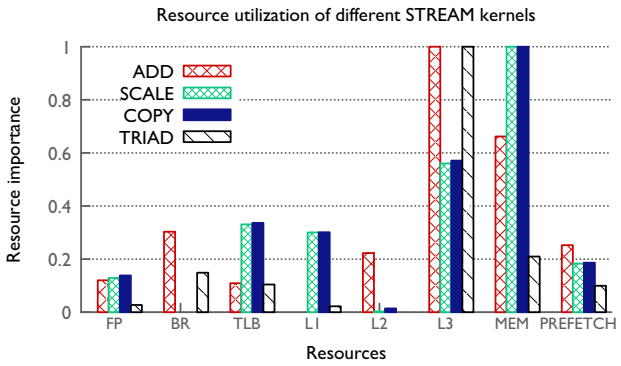

Fig. 3: Model validation by analyzing the resource utilization behaviors of different STREAM kernels.

resources that impact the on-node scalability of these kernels. VERITAS reports that the individual metrics that constitute traffic to the memory are data cache, prefetch, load, and store misses to the last-level cache. Since total memory used was $1GB$, which is greater than the size of the last-level cache, these misses generated traffic between the last-level cache and the node memory. The observations made based on our resource significance measure and the individual counters reported by VERITAS conform to the well known findings that the performance of STREAM kernels with scale are memory bandwidth bound.

Discussion: Although, the `COPY` kernel has no floating-point operations in the computation loop, the PAPI counter PAPI_FP_OPS on Sandy Bridge has over-counted the number of operations and instead of 0 has reported a very small, albeit non-zero number. This is why the model computes a non-zero resource importance for the FP resource. However, our model is robust against this: it shows very small RSM for the core-private resources. Since a value such as 0.13 indicates that a particular resource cannot be used with confidence in predicting the efficiency loss of an application, the ultimate result that L3 and memory (RSM $>$ 80%) are the most important resources for STREAM kernels still holds.

We also compared pairs of STREAM kernels with similar code paths, i.e., `ADD` (proxy) to cover `TRIAD` (parent), and `COPY` (proxy) to cover `SCALE` (parent) for $50M$ elements. Figure 4 shows that the resource utilization behaviors of `ADD` and `COPY` are similar to that of `TRIAD` and `SCALE`

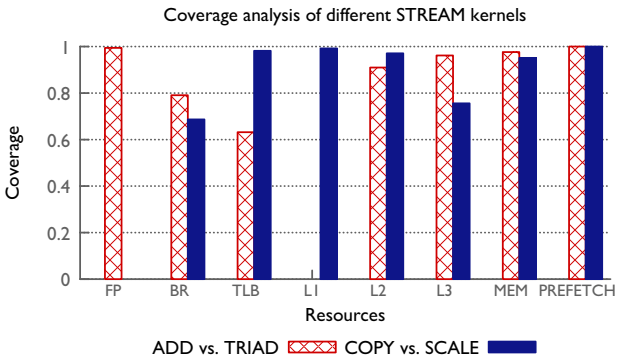

Fig. 4: Comparison of resource utilization behaviors of STREAM kernels. In this figure, we compare `ADD` (proxy) to cover `TRIAD` (parent), and `COPY` (proxy) to cover `SCALE` (parent).

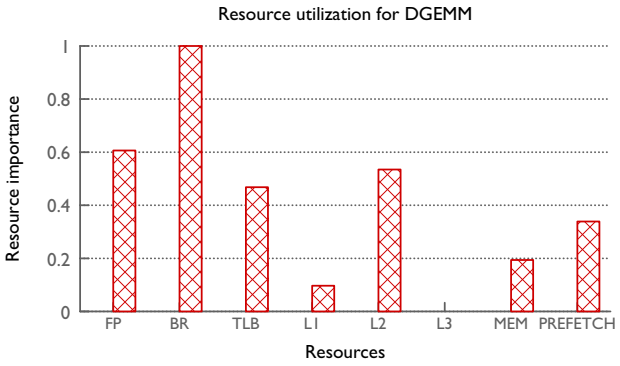

Fig. 5: Model validation with DGEMM.

respectively for all of the important resources.

DGEMM: In this experiment, we use a compute-bound benchmark, DGEMM, to validate our model. We collect the performance metrics for the parallel matrix-matrix multiplication kernel in DGEMM. To operate DGEMM in compute-bound mode, we use the compilation flag "-DUSE_MKL", and collect the performance metrics of the master thread alone since the benchmark calls the optimized DGEMM kernel provided by Intel compiler 13.1.163. We use $N = 4096$ (memory used $384MB$), which on average leads to 262 Gflop/s or 78% of the peak for the 16 thread case. Even though the memory requirement for this benchmark is larger than the cache sizes, the optimized DGEMM routine uses cache blocking to avoid stressing the TLB and the memory subsystem. We run one thread per core on an Intel Xeon machine, and scale DGEMM from 1 to 16 threads. Figure 5 shows that the efficiency loss of DGEMM with scale is due to the utilization of the computational units, since the RSM for the memory subsystem is low. This is expected from a compute-bound kernel, and the output from the model correctly reflects that. Figure 5 also shows that for FP, RSM is below the threshold of 80%,

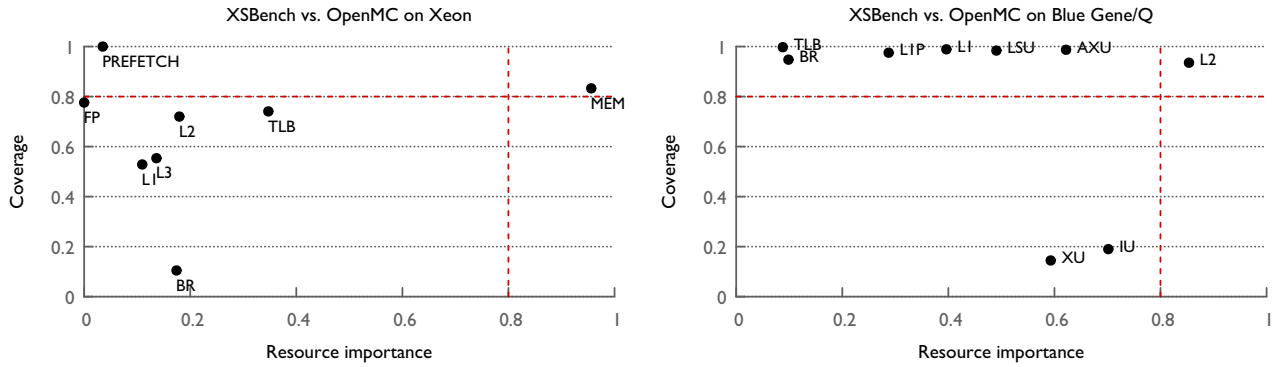

Fig. 6: Coverage analysis versus resource importance of XSBENCH against OPENMC on two different architectures.

since DGEMM shows good strong scaling and consequently the number of floating point operations decreases rather than being a bottleneck. Details provided by VERITAS show that as DGEMM scales, the contention for the branch prediction unit contributes strongly to the little remaining efficiency loss of DGEMM. This is likely caused by increasingly smaller tiles as we strong scale.

We also validated our model for computing "Coverage" (described in Section III-D) by computing compatibility between the same application with itself. The resulting Coverage was found to be 1, that is an application completely covers itself (figure not included in this paper).

B. XSBench

The objective of this experiment is to identify which resources on two different architectures cause efficiency loss when OPENMC scales for different workloads, and how well XSBENCH covers them. We ran XSBENCH and OPENMC on Xeon and BG/Q using six different workloads. These workloads essentially vary the size of a lookup table that is accessed randomly in both of these applications. Table III shows the workload sizes and the amount of memory consumed by each workload. Workload 1 is the smallest (445MB), and workload 5 is the largest (5710MB). These input workloads are publicly available in the OPENMC git repository. Each workload specifies the number of isotopes of a fuel material in a nuclear reactor. For a similar workload, XSBENCH takes the number of isotopes of the fuel material as input.

TABLE III: Workloads for OPENMC and XSBENCH.

Workload ID	Number of isotopes	Memory consumed (MB)
1	55	447
2	192	1721
3	242	2692
4	292	3877
5	341	5248
6	356	5700

From Figure 6 (left) we can observe that on Xeon the memory behavior of OPENMC impacts scalability across all workloads and XSBENCH captures 80% of that behavior ($Coverage > 0.8$). The beliefs of performance counters reported by VERITAS further reveals that prefetch, load, and store misses to the main memory, that is shared across cores on the same socket, are mainly responsible for efficiency loss on Xeon. The large contribution of node prefetch miss can be explained by the fact that random access through memory makes prefetch ineffective. Further investigation reveals that the large number of load and store misses to the on-node memory is generated because contiguous numbered cores physically reside on different sockets. Since we bound threads contiguously (thread 1 on core 1, thread 2 on core 2 and so on), more data lines allocated on different sockets were accessed by threads as these applications scaled.

From Figure 6 (right) we can observe that on BG/Q, the L2 utilization behavior of OPENMC causes efficiency loss. The performance counters reported by VERITAS reveals that the execution units stalled for data to be available on the L2 cache. This observation conforms to the general understanding of the fact that the on-node scalability of OPENMC is latency bound. XSBENCH covers 90% of the L2 utilization behavior on BG/Q.

C. RSBench

RSBENCH implements an alternative method to computing cross-section lookup. This algorithm reduces the memory consumption from 5.6 GB to 41 MB, and instead of using a lookup table, uses a large number of computation cycles to recompute data needed.

We can observe from Figure 7 (left) that RSBENCH exhibits different resource utilization characteristics than OPENMC on Xeon. The memory utilization behavior of OPENMC can be used almost exclusively to predict efficiency loss with scale, however RSBENCH only captures approximately 75% of that behavior. Since RSBENCH implements a more computationally expensive multipole method that reduces data movement to memory, its memory utilization behavior does not impact its scalability as much.

Figure 7 (right) shows that on BG/Q, surprisingly for strong thread-scaling experiments, the resource utilization behaviors of RSBENCH are similar to that of OPENMC even though

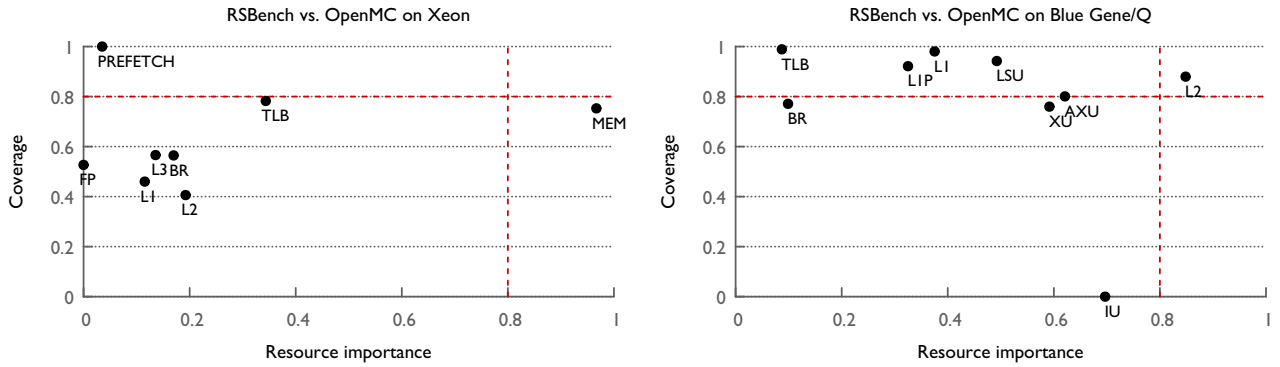

Fig. 7: Coverage analysis versus resource importance of RSBENCH against OPENMC on two different architectures.

these two applications implement two different algorithms. Details provided by VERITAS indicate that the major reason for efficiency loss of both OpenMC and RSBench on BG/Q was because the core was waiting for the shared L1 prefetcher to read data from L2 (PEVT_L1P_BAS_LU_STALL_* counters). This indicates that both of these algorithms, although implemented differently, are latency bound.

D. CMTBONE

In this section, we separately present the performance coverage behaviors of the three CMTBONE kernels described earlier. We varied the polynomial order and the number of elements parameters in CMTNEK and CMTBONE. We did strong scaling of both CMTNEK and CMTBONE from 1 through 16 processes on one Xeon node. We varied the workload by changing N_x to $5, 7, 9, 11, 13, 15$[1] and the number of elements to 512 and 1000. Altogether, there were 12 different workloads. Below, we present results on Xeon to demonstrate how VERITAS can be used to analyze the performance coverage of CMTNEK.

`point_compute_kernel`: This kernel computes primitive variables that are necessary for evaluating surface and volume integrals for CMTNEK and CMTBONE. Primitive variable computation is essentially a number of vector-vector multiplications. Figure 8 shows that on Xeon, the corresponding kernel in CMTBONE covers 100% of all but one of the resource utilization behaviors. The individual counters reported by VERITAS reveals that the only floating point performance counter that causes non-zero contribution to the efficiency loss in CMTBONE is the number of floating point operations that retire without generating an interrupt (measured by "FP_COMP_OPS_EXE:X87"), while the performance of CMTNEK does not depend on any of the floating point operations. This is not a significant difference and hence can be considered a perfect coverage.

`compute_kernel`: This kernel is the key module in CMTNEK that essentially performs matrix-matrix multiplications

[1]Odd N_x ensures even polynomial order and integral whole numbers for the number of points for overintegration.

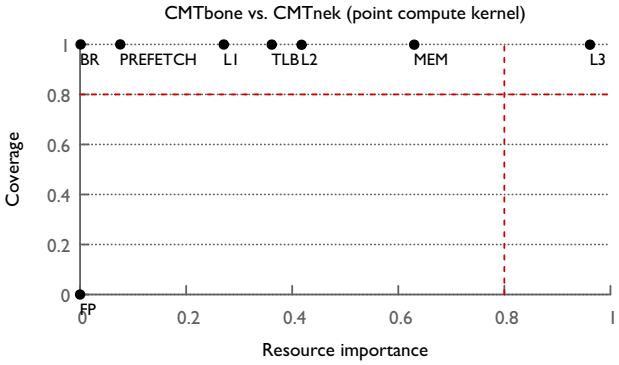

Fig. 8: The `point_compute_kernel` of CMTBONE covers the resource utilization of that for CMTNEK.

for computing fluxes and spatial partial derivatives of fluxes. Figure 9 shows that the `compute_kernel` of CMTBONE captures more than 80% of the memory utilization behavior of CMTNEK. Additionally, the proxy application covers the rest of the resource utilization behaviors almost exactly. This figure also shows that the major cause of efficiency loss with scale in the `compute_kernel` for both applications is the memory utilization behavior. Further investigation reveals that the gradient and the volume integral computations in this module perform a number of matrix-matrix multiplications where matrices are accessed in column major order. This results in a large number of L3 store misses that generates a large number of node memory store operations.

`comm_kernel`: In this kernel both CMTNEK and CMTBONE implement steps necessary to compute face fluxes. Figure 10 shows that even though memory utilization is the most important behavior that impacts scalability (Resource Importance ≥ 0.9), the `comm_kernel` in CMTBONE does not capture that characteristics ($Coverage < 0.5$). Further investigation reveals that the face flux computation function in CMTNEK implements a number of memory copy operations from a $3D$ matrix to an array that the corresponding

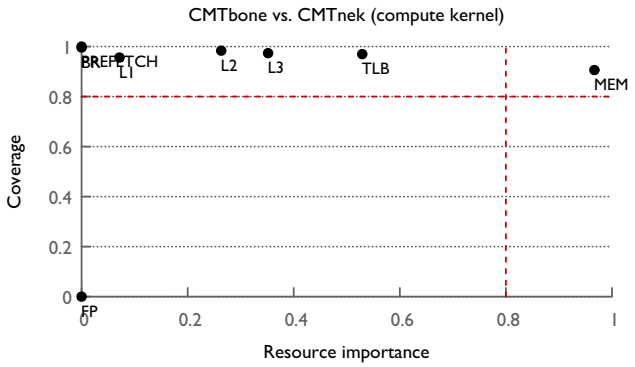

Fig. 9: The `compute_kernel` of CMTBONE covers the resource utilization of that for CMTNEK.

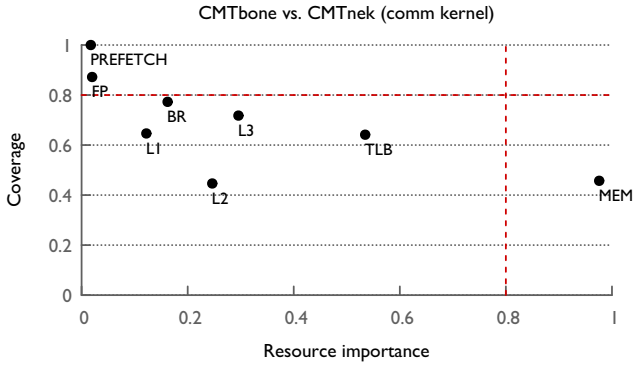

Fig. 10: The `comm_kernel` of CMTBONE does not cover the memory utilization behavior of CMTNEK.

kernel in CMTBONE does not implement. Since memory copy is expensive, it affects the scalability of the `comm_kernel` in CMTNEK, but is not covered similarly by that in CMTBONE. The significant impact of these missing operations was not expected by the application developers. This discrepancy already led to new developments for CMTBONE in order to improve the match.

VII. Related Work

Tramm et al. [27], [29] analyze XSBENCH and RSBENCH using a correlation-based approach to understand how well XSBENCH and RSBENCH cover the resource utilization behaviors of OPENMC. However, since correlation does not mean causation, performance counters strongly correlated with runtime do not necessarily contribute to the performance loss. Also, none of these techniques can address the fact that a number of these performance counters can be inter-correlated in nature with the performance attribute. In contrast, our methodology identifies groups of performance metrics that cause the performance loss in applications as we scale.

Supervised learning techniques have been used in the literature successfully for selecting features that can effectively predict runtimes of applications. Jain et al. [32] and Bhatele et al. [16] respectively apply regression models to predict the runtime of applications based on network congestion metrics and use rank-wise correlation and R^2 measures to evaluate the quality of prediction. In addition, Bhatele et al. perform an extensive search on all possible subsets of performance metrics to select features that result in the best prediction performance. Since the number of hardware counters can be large (on the order of hundreds), such an exhaustive feature selection process is not ideal. In contrast, VERITAS extends an unsupervised learning technique by building linear regression models with inherent feature selection. Commonly referred to as sparse coding or Lasso regression, this approach encodes the target feature (to be predicted) using a sparse set of input features. The requirement for sparsity both regularizes the regression problem to avoid overfitting, and makes the inference robust to noise/outlying data.

Principal Component Analysis (PCA) and clustering-based analysis approaches have been used for analyzing massive number of performance counters to understand program behavior. Phansalkar et al. [33] apply these techniques to the SPEC CPU2006 benchmark suite to study whether applications included are well balanced or not. Ahn et al. [34] apply these techniques to understand performance bottlenecks of applications. While PCA is an important tool for reducing the number of features to investigate, it combines multiple features into groups that cannot be used to tie efficiency loss back to individual resources that are culprits. In the context of performance coverage analysis, identifying individual metrics is the key to answering which resource utilizations are covered.

Yoo et al. [35] provide a decision tree based methodology for associating performance metrics to program execution behavior and apply it to micro-kernels with single bottleneck each. While this idea is useful for identifying performance bottlenecks for individual applications, the decision tree based technique cannot be applied to study which performance behaviors of two applications match.

VIII. Conclusion

Proxy applications are designed to cover a subset of performance characteristics of larger production applications. This paper presents a methodology that identifies which performance characteristics of an application lead to efficiency loss during on-node scaling; which of these are covered by a proxy, and how well. In this paper, we present two novel quality metrics along with machine learning techniques to compute them: Resource Significance Measure and Coverage. We implement our methodology in a framework called VERITAS and apply it to analyze five different kernels. Our framework reports both overall resource-level significance and the individual performance metrics that contribute to efficiency loss. These results can be used by application developers to analyze which resource bottlenecks are covered by a proxy with high confidence, and which important performance characteristics of the parent applications are not. This shows whether the existing proxies are sufficient or whether new, additional proxies need to be developed in order to complete the intended coverage.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-696018).

REFERENCES

[1] "NAS parallel benchmarks (NPB)." [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[2] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '13. IEEE Computer Society, May 2013, lLNL-CONF-586774.

[3] "XSBench: The monte carlo macroscopic cross section lookup benchmark," 2014. [Online]. Available: https://github.com/jtramm/XSBench

[4] N. Kumar, M. Sringarpure, T. Banerjee, J. Hackl, S. Balachandar, H. Lam, A. George, and S. Ranka, "CMT-bone: A mini-app for compressible multiphase turbulence simulation software," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 785–792.

[5] "Mantevo benchmark suite." [Online]. Available: https://software.sandia.gov/mantevo/index.html

[6] P. K. Romano and B. Forget, "The OpenMC monte carlo particle transport code," *Annals of Nuclear Energy*, vol. 51, pp. 274–281, 2013.

[7] P. F. Fischer, J. Lottes, S. Kerkemeier, K. Heisey, A. Obabko, O. Marin, and E. Merzari, "http://nek5000.mcs.anl.gov," 2014.

[8] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. Zhang, "A survey of sparse representation: Algorithms and applications," *IEEE Access*, vol. 3, pp. 490–530, 2015.

[9] D. L. Donoho, "Compressed sensing," *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, April 2006.

[10] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *The Annals of Statistics*, vol. 32, no. 2, pp. 407–499, 04 2004.

[11] J. J. Thiagarajan, K. N. Ramamurthy, P. Turaga, and A. Spanias, "Image understanding using sparse representations," *Synthesis Lectures on Image, Video, and Multimedia Processing*, vol. 7, no. 1, pp. 1–118, 2014.

[12] I. Bloch, "Some aspects of dempster-shafer evidence theory for classification of multi-modality medical images taking partial volume effect into account," *Pattern Recognition Letters*, vol. 17, no. 8, pp. 905 – 919, 1996.

[13] "BGPM hardware performance counters." [Online]. Available: https://bgq1.epfl.ch/navigator/resources/doc/bgpm/bgpm_events.html

[14] D. Levinthal, "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors," *Intel Performance Analysis Guide*, 2009.

[15] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP User Group Conference*, Jun. 1999.

[16] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '15. IEEE Computer Society, May 2015, LLNL-CONF-663150.

[17] G. Davis, S. Mallat, and M. Avellaneda, "Greedy adaptive approximation," *Journal of Constructive Approximation*, vol. 13, pp. 57–98, 1997.

[18] J. A. Tropp, "Greed is good: Algorithmic results for sparse approximation," *IEEE Trans. Inf. Theory*, vol. 50, no. 10, pp. 2231–2242, October 2004.

[19] A. Bruckstein, M. Elad, and M. Zibulevsky, "On the uniqueness of nonnegative sparse solutions to underdetermined systems of equations," *IEEE Transactions on Information Theory*, vol. 54, no. 11, pp. 4813–4820, 2008.

[20] K. Ramamurthy, J. Thiagarajan, A. Spanias, and P. Sattigeri, "Boosted dictionaries for image restoration based on sparse representations," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 1583–1587.

[21] S. Le Hegarat-Mascle, I. Bloch, and D. Vidal-Madjar, "Application of dempster-shafer evidence theory to unsupervised classification in multisource remote sensing," *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 35, no. 4, pp. 1018–1031, Jul 1997.

[22] LLNL, "perf-dump: A tool for collecting PAPI counter values per-process, per-thread, and per-timestep." https://github.com/scalability-llnl/perf-dump.

[23] "A deeper look at TLBs and costs," http://lwn.net/Articles/379748/.

[24] "Vulcan: Blue Gene/Q system in Livermore Computing," http://computation.llnl.gov/computers/vulcan.

[25] "Cab: Intel Xeon system in Livermore Computing," http://computation.llnl.gov/computers/cab.

[26] "Openmc monte carlo code," 2014. [Online]. Available: https://github.com/mit-crpg/openmc

[27] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench–the development and verification of a performance abstraction for monte carlo reactor analysis," 2014.

[28] A. R. Siegel, K. Smith, P. K. Romano, B. Forget, and K. G. Felker, "Multi-core performance studies of a monte carlo neutron transport code," *International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 87–96, 2014.

[29] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations," pp. 39–56, 2014.

[30] J. D. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, pp. 19–25, 1995.

[31] "DGEMM." [Online]. Available: http://www.nersc.gov/research-and-development/apex/apex-benchmarks/dgemm/

[32] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635857.

[33] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 412–423.

[34] D. H. Ahn and J. S. Vetter, "Scalable analysis techniques for microprocessor performance counter metrics," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 3–3.

[35] W. Yoo, K. Larson, S. Kim, W. Ahn, R. Campbell, and L. Baugh, "Automated fingerprinting of performance pathologies using performance monitoring units (pmus)," in *Proc. of USENIX Workshop on Hot topics in parallelism. USENIX Association*, 2011.