ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

AutoMOMML: Automatic Multi-Objective Modeling with Machine Learning[1]

Prasanna Balaprakash, Ananta Tiwari, Stefan M. Wild, Laura Carrington, and Paul D. Hovland

Mathematics and Computer Science Division

Preprint ANL/MCS-P5421-1015

October 2015

[1]This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program under contract number DE-AC02-06CH11357.

AutoMOMML: Automatic Multi-Objective Modeling with Machine Learning

Prasanna Balaprakash*†, Ananta Tiwari‡, Stefan M. Wild*, Laura Carrington‡, and Paul D. Hovland*
* *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA*
{*pbalapra, wild, hovland*}*@mcs.anl.gov*
† *Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA*
‡ *Performance Modeling and Characterization Lab, San Diego Supercomputer Center, La Jolla, CA, USA*
{*tiwari, lcarring*}*@sdsc.edu*

***Abstract*—In recent years, automatic data-driven modeling with machine learning has received considerable attention as an alternative to analytical modeling for many modeling tasks. While ad hoc adoption of machine learning approaches has obtained success, the real potential for automation in data-driven modeling has yet to be achieved. We propose AutoMOMML, an end-to-end, machine-learning-based framework to build predictive models for objectives such as performance, power, and energy. The framework adopts statistical approaches to reduce the modeling complexity and automatically identifies and configures the most suitable learning algorithm to model the required objectives based on hardware and application signatures. The experimental results using PAPI hardware counters as these signatures show that the median prediction error of performance, processor power, and DRAM power models are 13%, 2.3%, and 8%, respectively.**

I. Introduction

Modeling objectives such as performance, failures of critical subcomponents, power, and energy as functions of application and platform characteristics play a central role in managing extreme-scale computing goals. These models can be used to quantify meaningful differences across the decision space and to provide error bounds/distributions associated with their predictions; to offer a convenient mechanism for exposing near-optimal spots in the decision space; and to prune the decision space and search-related tasks in autotuning. In a nutshell, multi-objective models can help compilers, operating systems, and runtime systems to make decisions proactively and/or reactively in order to best map applications to target platforms.

Analytical performance models based on first-principle, closed-form mathematical expressions may not be sufficiently accurate for all objectives of interest. In such settings, data-driven (or "empirical") modeling can bridge the gap. In this approach, application and platform characteristics and their corresponding objectives are collected directly on the target platform, and a predictive model is built for each objective using statistical/machine-learning (ML) approaches.

The empirical models presented in the high performance computing (HPC) literature have been guided primarily by the expertise of the human modeler. It has become increasingly evident in the ML literature that success with ML algorithms depends not merely on the adoption of the suitable learning approach for a given data set, but also on the mastery of a more complex feature and algorithm engineering process [5]. Challenges in predictive modeling can be attributed to two major factors: the modeling complexity and the degrees of freedom modelers encounter when developing predictive models. Crucial aspects in predictive model development comprise variable selection, model selection, parameter tuning, cross-validation techniques, and background knowledge in disciplines such as machine learning, statistics, and mathematical optimization.

We propose as a solution to the aforementioned problem an automated, end-to-end modeling framework called AutoMOMML (for Automatic Multi-Objective Modeling with Machine Learning). AutoMOMML employs a pipeline of statistical approaches in a systematic way to automate the predictive modeling process. The framework identifies the important variables, and selects and tunes the learning algorithms to model the required objectives based on hardware and application characteristics. Applications are characterized by using a set of performance hardware counters, which are simple counts of microarchitectural events (e.g., L1 cache misses). To generate training data, AutoMOMML uses a series of prepackaged microkernels to "probe" a target system in order to develop a comprehensive understanding of the degree to which application characteristics and hardware configurations affect component-level power draw and performance. That understanding is then encapsulated in models by using ML algorithms. The end-to-end framework greatly reduces the barrier to entry in model development for software developers, run-time designers, and hardware engineers and has the potential to bring modeling into the mainstream workflow of software and hardware stakeholders.

The models presented in this paper can be used either within higher-level autotuning frameworks or within the introspective and adaptive runtime systems envisioned for future extreme-scale systems [7]. The models can be used to gather architectural insights in terms of which components (e.g., front-end, branch unit and memory hierarchy) bottleneck the performance in a given system. The models also provide feedback in terms of how the power resource is

divided across different architectural components and how this division changes with a change in application characteristics and hardware settings. These insights can then be used within a multi-objective tuning framework [3, 15, 25] to steer systems towards stipulated goals for energy and performance. Although integrating the models within a tuning framework is out of the scope of this paper, we will highlight attractive features of the models for this purpose.

The key contribution of the paper is the general-purpose multi-objective modeling framework that comprises a pipeline of effective ML algorithms. We demonstrate the use of the framework for offline-modeling of performance and power. In addition to being automatic and end-to-end, the framework is designed to produce analysis results after each stage of the pipeline that help understand what architectural factors affect the objectives, and how application signatures and objectives relate to each other.

II. The Problem and Setup

Given a target platform, the task of multi-objective modeling is to find a function

$$F(x) = [F_1(x), \ldots, F_m(x)] : \qquad x \in \mathcal{D} \subset \mathbb{R}^n, \qquad (1)$$

where x is a vector of size n that parameterizes a hardware and application signature and $\mathcal{D}$ is a domain of possible values for x. The unknown function F takes the signature vector x as input and returns a vector $[F_1(x), \ldots, F_m(x)]$ quantifying m objectives, where each component corresponds to an objective of interest.

Approaches available for modeling F can be grouped into analytical and empirical modeling. The former deals with developing analytical approximations for each component function F_j using expert knowledge. The latter adopts statistical or ML methods to derive a surrogate model $S_j \approx F_j$ using a set of training points $\mathcal{T} = \{(x_1, y_1), \ldots, (x_l, y_l)\} = \{X, Y\}$ obtained from microkernels. Each point in the training set comprises the signature vector $x_{(.)}$ and its corresponding multi-objective vector $y_{(.)} \in \mathbb{R}^m$.

Modeling with ML typically requires a pipeline of methods such as data preprocessing, variable importance analysis, variable selection, and model selection. The complexity of employing an effective ML pipeline is beyond most HPC users because of the algorithmic choices available for each method; therefore, users tend to resort to rules of thumb, which often result in nonrobust models. We develop a methodology that automatically selects an ML pipeline for the multi-objective modeling problem.

In this paper we focus on a signature vector consisting primarily of hardware performance counters exposed by the underlying hardware and collected by using the Performance Application Programming Interface (PAPI) [26]. Hardware counters provide a convenient mechanism to measure the extent to which applications utilize/stress different architectural elements. For example, counters that measure the number of branch instructions can be used to assess the level of stress that different applications put on the branch prediction units. As such, a vector of hardware counters can be used to describe an application. In addition, a given application's power and performance behavior are affected by power- and performance-related hardware settings (e.g., CPU clock frequency, duty cycles, and power capping). Consequently, we add CPU clock frequency to the signature vector.

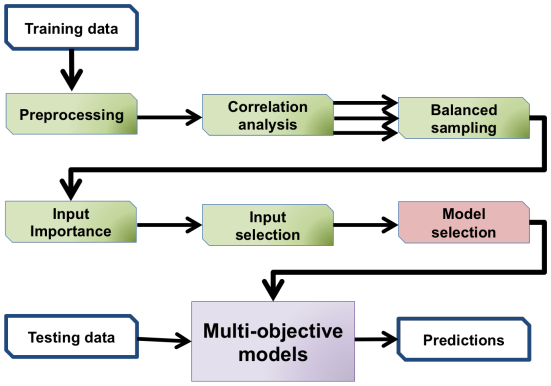

Figure 1. Overview of AutoMOMML framework. Multiple arrows after correlation analysis indicate that subsequent models are run for each objective.

III. Proposed Approach

AutoMOMML consists of a pipeline of algorithmic modules (as depicted in Figure 1) that can be grouped into two main stages. The first stage is the dimension reduction stage, which reduces the number of inputs and outputs required for modeling via correlation analysis, input importance, and input selection algorithmic modules. This stage plays a critical role in reducing the modeling complexity. The second stage in the pipeline is the model selection stage, where several supervised-learning methods are evaluated and fine-tuned on the training set; high-performing methods are then composed to obtain the multi-objective models.

A. Dimension reduction

Data preprocessing: Different entries in the signature vector x can take different ranges of values. For example, instruction-cache-related counts (e.g., L1 instruction cache misses) are usually orders of magnitude smaller than data-cache-related counts (e.g., L1 data cache misses). This difference in the range of values that entries in x can take affects several algorithmic modules in the pipeline. AutoMOMML adopts *range transformation* [10] to scale the values of each column in X to [0, 1].

Correlation analysis: This module computes the pairwise correlation to check for correlation among inputs. Given two input columns j and j' of X, the Pearson product-moment correlation coefficient $\rho \in [-1.0, 1.0]$ is given by the ratio of the covariance between j and j' to the product of the standard deviations of j and j'. When the value of $|\rho|$ is greater than a user-defined cutoff parameter `ccoff`, the input $x^{j'}$ that corresponds to column j' is removed from

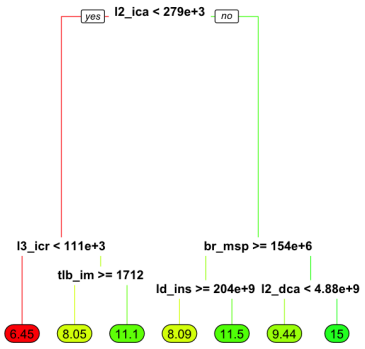

Figure 2. Example decision tree for predicting DRAM power.

further analysis. The same analysis is applied on the output matrix Y to reduce the dimension of the output space.

For each uncorrelated output u, AutoMOMML creates an output-specific training set $\mathcal{T}_u$. It comprises all uncorrelated inputs and the output u. After this analysis, the subsequent algorithmic modules are run for each $\mathcal{T}_u$; consequently, each $\mathcal{T}_u$ can be tackled independently.

Balanced sampling: Heavily skewed distribution of response u in $\mathcal{T}_u$ can lead to unbalanced training points. When these points are used for predictive modeling, the model will have high prediction accuracy in high (probability) density regions but not in other areas. Two strategies can be adopted to address this issue: (1) *under sampling*, in which training points from high-frequency ranges are discarded, and (2) *over sampling*, in which training points are sampled repeatedly from low-frequency ranges. Although the latter artificially increases the number of training points (and eventually the training time and memory footprint), it is desirable in offline settings because it makes full use of the available data.

The over sampling strategy that we adopt proceeds as follows. Given an output-specific set $\mathcal{T}_u$, we consider E equal-sized intervals for the output u. Let $E_{\max}$ be the number of points that belong to the high-frequency interval. For each of the remaining intervals, the number of points is increased to $E_{\max}$ by repeatedly sampling (with replacement) from that interval. Consequently, for each output, the total number of points in the training set will be $E \times E_{\max}$. We denote the resulting balanced training set by $\mathcal{T}'_u$.

Input importance: This module analyzes the impact of the uncorrelated inputs on the output and tries to remove some inputs from further consideration. The results can be used to understand (and rank) the application characteristics that affect the power and performance responses the most.

For this purpose, the random forest (`rf`) method [11], a state-of-the-art supervised-learning method for nonlinear regression, is adopted. The `rf` method uses a decision-tree-based approach that recursively partitions the multidimensional input space $\mathcal{D}$ into hyperrectangles such that inputs with similar output values fall within the same hyperrectangle. Partitions give rise to a decision tree of if-else rules as shown in Figure 2. The tree shows that DRAM power is highly dependent on how the codes utilize on-chip and off-chip instruction and data caches, along with the behaviors of the TLB and branch predictor. High DRAM power is associated with a higher number of accesses to slower caches (L2 and beyond) and more TLB misses (more on this in Sec. V). The depth of the tree is determined by a parameter `dpt`, which controls the learning ability of the tree.

Typically, a constant value is assigned to the leaf of the tree and is given by the mean of output values that fall within the same hyperrectangle. Prediction for a new input x^* is obtained by finding the hyperrectangle to which this point belongs using the decision tree and returning the constant value at the associated leaf. The strength of the `rf` method lies in using a collection of decision trees because the ensemble corrects the instability of the individual trees. The predicted value for x^* is given by the average of leaf values obtained from each generated tree.

AutoMOMML deploys the permutation accuracy importance of `rf`. For a given training set $\mathcal{T}'_u$, by randomly permuting the values of column j in X, the original relationship between input x^j and the response u will be broken. When X with permuted x^j is used to predict u, the prediction accuracy decreases substantially as compared with that of the original dataset with nonpermuted x^j. The impact of an input parameter x^j on the output o is computed as follows. For each tree, the random subsample $X' \subset X$ is split into in-bag and out-of-bag. The in-bag is used for building the tree, and the mean squared error (MSE) on the out-of-bag data is computed before and after permuting x^j in in-bag. The differences between the two are then averaged over all trees and normalized by the standard deviation of the differences. A significant increase in MSE after permuting values of x^j indicates that x^j has significant impact on u. When the increase in MSE is small (e.g., <5%), x^j is removed from the set of inputs required for predicting u. Compared to other sensitivity analysis methods, this approach covers the impact of each input both individually and in combination with other inputs. Moreover, each tree is constructed only from a fraction of random inputs, thereby reducing the number of training points required. The resulting training set is denoted as $\mathcal{T}''_u$, which comprises only the significant inputs for output u.

Input selection: Given the training set $\mathcal{T}''_u$ and a set S of input sizes, this module tries to find an input size $s \in S$ for predicting the output u. As shown in Algorithm 1, input selection is done in two phases.

The first phase is K-fold cross-validation (lines 1–11). Training points are partitioned into K equal-sized folds by using random sampling without replacement. Out of the K folds, a single fold is retained as an out-of-bag set for validation; the remaining $K-1$ folds are used as in-bag points for training. Importance of each input in the in-bag

Algorithm 1 Model-based input selection.

Input: Number of folds, K, for cross-validation, training points $\mathcal{T}_u''$, a set S of subset sizes, error tolerance percentage $\delta\%$, set $\mathcal{I}_u$ of inputs

/ cross-validation phase */*

1 create K folds $\{\mathcal{T}_{u1}'', \cdots, \mathcal{T}_{uK}''\}$ from $\mathcal{T}_u''$
2 **for** $k = 1 : K$ **do**
3 $\quad \mathcal{T}_{\text{out}} \leftarrow \mathcal{T}_{uk}''$; $\mathcal{T}_{\text{in}} \leftarrow \mathcal{T}_u'' \backslash \mathcal{T}_{\text{out}}$
4 $\quad u_{obsr} \leftarrow$ observed output in $\mathcal{T}_{\text{out}}$
5 $\quad \mathcal{M}_{rf} \leftarrow \texttt{fit}(\mathcal{T}_{\text{in}}, \mathcal{I}_u'')$
6 $\quad$ compute *permut. acc. importance* of $\mathcal{I}_u''$
7 $\quad$ **for** each $s \in S$ **do**
8 $\quad\quad \mathcal{I}_s \leftarrow s$ most important inputs from $\mathcal{I}_u''$
9 $\quad\quad \mathcal{M}_{rf} \leftarrow \texttt{fit}(\mathcal{T}_{\text{in}}, \mathcal{I}_s)$
10 $\quad\quad u_{pred} \leftarrow \texttt{predict}(\mathcal{M}_{rf}, \mathcal{T}_{\text{out}})$
11 $\quad\quad err_{ks} \leftarrow \texttt{RMSE}(u_{obsr}, u_{pred})$
12 $\quad$ **end for**
13 **end for**

/ subset selection phase */*

14 $\overline{err}_s \leftarrow \frac{\sum_{k=1}^{K} err_{ks}}{K}$
15 $\overline{err}^* \leftarrow \min_{s \in S} \overline{err}_s$

Output: $s_{\text{best}} = \arg\min\{s \in S : \overline{err}_s \leq \overline{err}^* \times (1 + \delta\%)\}$

Algorithm 2 Model selection.

Input: number K, training points $\mathcal{T}$, set Z of methods, set Q of parameter configurations for each method in Z

/ cross-validation */*

1 create K folds $\{\mathcal{T}_{u1}, \cdots, \mathcal{T}_{uK}\}$ from $\mathcal{T}_u$
2 **for** $k = 1 : K$ **do**
3 $\quad \mathcal{T}_{\text{out}} \leftarrow \mathcal{T}_{uk}$; $\mathcal{T}_{\text{in}} \leftarrow \mathcal{T}_u \backslash \mathcal{T}_{\text{out}}$
4 $\quad u_{obsr} \leftarrow$ observed output in $\mathcal{T}_{\text{out}}$
5 $\quad$ **for** each $z \in Z$ **do**
6 $\quad\quad Q_z \leftarrow$ subset of param configs in Q for z
7 $\quad\quad$ **for** each $q \in Q_z$ **do**
8 $\quad\quad\quad \mathcal{M}_z \leftarrow \texttt{fit}(\mathcal{T}_{\text{in}}, q)$
9 $\quad\quad\quad u_{pred} \leftarrow \texttt{predict}(\mathcal{M}_z, \mathcal{T}_{\text{out}})$
10 $\quad\quad\quad err_{kzq} \leftarrow \texttt{RMSE}(u_{obsr}, u_{pred})$
11 $\quad\quad$ **end for**
12 $\quad$ **end for**
13 **end for**

/ select best parameter setting for each method */*

14 **for** each $z \in Z$ **do**
15 $\quad$ **for** each $q \in Q_z$ **do**
16 $\quad\quad \overline{err}_{zq} \leftarrow \frac{\sum_{k=1}^{K} err_{kzq}}{K}$
17 $\quad$ **end for**
18 $\quad \overline{q}_z \leftarrow \arg\min_{q \in Q_z} \overline{err}_{zc}$
19 **end for**

/ select best method(s) using statistical test */*

20 $z^* \leftarrow \arg\min_{z \in Z} \overline{err}_{z\overline{q}_z}$
21 **for** each $z \in Z$ **do**
22 $\quad$ **if** $\texttt{t-test}(err_{(.)z\overline{q}_z}, err_{(.)z^*\overline{q}_{z^*}})$ cannot reject **then**
23 $\quad\quad \mathcal{M}_z \leftarrow \texttt{fit}(\mathcal{T}, \overline{q}_z)$
24 $\quad$ **end if**
25 **end for**

Output: $\mathcal{M} = \cup_z \mathcal{M}_z$

points is obtained with the permutation accuracy importance of `rf`. For each candidate value $s \in S$, an `rf` model is retrained with the s most important inputs, and the root-mean-squared error (RMSE) for the prediction is obtained from the out-of-bag points. This process is repeated so that each of the K folds is used exactly once as out-of-bag.

The second phase of input selection consists of analyzing the results from the K folds to compute a single best subset size. The mean prediction error $\overline{err}_s$ for each $s \in S$ is obtained by averaging the prediction error over K folds. The algorithm chooses the smallest $s \in S$ whose prediction error $\overline{err}_s$ is not more than $\delta\%$ of the minimal mean prediction error $\overline{err}^*$. Smaller s values are preferred because they reduce the input space and can improve the predictive power of the model. Note that the s_{best} inputs for each fold can be different because the training points are different. This module handles such cases by computing the average rank of each input over all K folds and selects the top s_{best} inputs using the aggregated rank value.

Although Alg. 1 relies heavily on the `rf` method, it has been shown that for a number of modeling tasks input selection from the `rf` method is robust, and can improve the accuracy of other supervised-learning methods [20].

B. Model selection

The model selection module consists of finding an appropriate method (and associated parameters) from a set of supervised-learning methods. A supervised-learning method that performs well on some predictive modeling tasks could be a bad choice for other tasks [10, 5]. Moreover, choosing appropriate parameter settings for a given supervised-learning method is critical because it balances the bias-variance tradeoff [10]: High bias produces simpler models but leads to poor prediction accuracy, while high variance results in complex models with high prediction accuracy on the training set but can have poor prediction accuracy on the testing set.

Model selection is a difficult optimization problem. In ML research, this task has been traditionally tackled by using a trial-and-error process. New algorithmic methods have begun to emerge and have proven to be more effective than default settings and manual model selection [5]. The model selection module of AutoMOMML considers a set of supervised-learning methods of varying complexities, tunes the parameters of each method, and combines the high-performing models to form a single model.

Algorithm 2 shows the model selection module. In addition to K and $\mathcal{T}$, it requires a set Z of supervised-learning methods and a subset $Q_z \in Q$ of parameter configurations for each method $z \in Z$. The module comprises three phases. First, for each method $z \in Z$ and for each parameter configuration $q \in Q_z$, the cross-validation phase consists of configuring z with q, training on the in-bag points and computing the prediction error on the out-of-bag points (lines 1–13). The second phase identifies the best configuration $\overline{q}_z$ for each method z by comparing the mean prediction error. In the third phase, the module selects the method z^* (configured with $\overline{q}_{z^*}$) with minimal mean prediction error as a baseline and adopts the statistical t-test to check the prediction errors of a method z ($err_{(.)z\overline{q}_z}$) is different from the baseline z^* ($err_{(.)z^*\overline{q}_{z^*}}$). The method z gets eliminated when the t-test rejects the null hypothesis that the difference

is equal to zero. The methods that survive the elimination are configured with their corresponding best parameter setting and trained on all training points. The resulting models are returned as candidate models. For a given output, when there is more than one candidate model, the predictive value of a new testing point is given by the average of predicted values from each candidate model.

In addition to `rf`, we consider five supervised-learning methods. A brief summary of each method is given below.

Linear regression (`lm`) is perhaps the simplest and the most well known. It takes the form $h(x) = c + \sum_{i=1}^{M} \alpha^i \times x^i$, where c is a constant and α^i is the coefficient of the input x^i. Training the model consists in finding the appropriate value of (c, α). This is obtained by minimizing the sum of differences between observed values in the training set and the corresponding values provided by the model.

$\mathcal{K}$-nearest-neighbor (`knn`) regression [10] computes the mean of the outputs of the $\mathcal{K}$ nearest (we adopt the Euclidean distance metric) points in the training set.

Support vector machines (`svm`) [22] for nonlinear regression project the input space of the training points into a higher-dimensional feature space and performing linear regression in that space. Training the `svm` consists of solving a quadratic programming problem. We adopt the widely used Gaussian radial basis function (RBF) as our kernel function. The cost v of constraint violation in the quadratic programming problem and the window parameter σ of the RBF provide tradeoffs between bias and variance.

Cubist (`cbt`) [1] is similar to `rf` but with the following differences. The `nt` trees are built sequentially such that the model of the bth tree is adjusted to correct the prediction error of the $(b-1)$th tree on the whole training set. This correction is obtained by adding the residuals of the $(b-1)$th tree to the response vector and fitting a new tree. Given a new testing point x^*, each tree can predict a value and nt predictions are averaged to give a final prediction. During prediction, it performs additional corrections based on `nn` nearest neighbors in the training set.

Stochastic gradient boosting (`sgb`) [17] is similar to `cbt`, in which nt trees are built sequentially but on a random subset of the training points. Each tree is generated with depth `dpt` and its leaves have at least $\min_o$ observations. At the bth iteration, a tree model is built to minimize the prediction error of the $(b-1)$th model. The key idea is that the residuals of the $(b-1)$th model are used as the negative gradient of the squared error loss function being minimized. Similar to gradient-descent algorithms, `sgb` generates a new model at the bth iteration by adding the bth tree that fits the negative gradient to the $(b-1)$th model. The bth model is multiplied by a parameter $0 < \lambda \leq 1.0$ to control the bias-variance tradeoff.

Except `lm`, all other supervised-learning methods require user-specified values for their respective parameters. Since promising values for each parameter are available for `knn`, `svm`, `rf`, `cbt`, and `sgb`, we define the set Q_z of parameter configurations for the method z using a grid. For example, if the method z has two parameters with 3 and 5 values, respectively, then we consider all possible combinations ($|Q_z| = 3 \times 5$). The set Q_z over all $z \in Z$ forms Q, which is given as input to Algorithm 2.

IV. Experimental Setup

We now describe the hardware testbed, benchmarks and applications, data collection techniques, and other methodological decisions made for data collection.

Hardware Testbed Specifications: The testbed is a dual-processor node with two 8-core Intel Xeon E5-2650v2 (Ivy Bridge) processors. Each core has a 64 KB L1 cache (32 KB instruction cache and 32 KB data cache), a 256 KB combined L2 cache, and a 20 MB shared, last-level cache. The system has 64 GB of DDR3 DRAM. Hyperthreading and turbo boost are disabled for all the experiments. Each of the processors can be independently clocked at frequencies of 2.60 GHz to 1.20 GHz (at 100 MHz intervals). Processor clock frequency is changed by using the `cpufreq-utils` package available with many popular Linux distributions.

We use the RAPL (Running Average Power Limit) interface [16] to measure component-level power draw. For the processor, we collect "package power," which consists of power drawn by cores, the last-level cache and memory controller. We also collect power drawn by DRAM.

Model Training Benchmarks: AutoMOMML comes prepackaged with microkernels that exercise a target system's architectural components (e.g., CPU, and memory subsystem) in different ways[1]. Together, these computational kernels can be used to create power and performance profiles of the system and to learn what hardware events correlate with those profiles. These microkernels have different patterns of computation and memory access, and are highly prevalent in large-scale applications. Our hypothesis is that a sufficiently diverse set of computational kernels can be used as the basis for a general understanding of the impact that different computational properties have on performance and power draw.

Our suite draws compute kernels from a variety of scientific domains. Some of the kernels are modified versions of microkernels from the Polybench [32] and SPAPT suites [4], both of which are used to evaluate compiler-driven autotuning strategies [29]. Such kernels include matrix-matrix and matrix-vector operations, stencil kernels, and correlation and covariance calculation kernels. We also use the source code transformation framework CHiLL [14] to generate alternative implementations of a subset of Polybench kernels (dsyrk, dsyr2k, mm, mvt, and trmm). We apply cache tiling and loop unrolling code-transformation techniques to generate these variants.

[1]We will make the packages and the framework available on our website (http://www.sdsc.edu/~tiwari/AutoMOMML) at the time of publication.

The kernels are configured to run in an embarrassingly parallel mode using MPI and using all cores available on the testbed. Each MPI process initializes its own set of data structures and waits on a barrier for all the processes to finish initialization; all processes then do the exact same calculation. This configuration was motivated by two factors. First, RAPL power measurements can be made only at the per-socket level; per-core-level measurements are not available. Second, since our goal is to use the models to make performance and power draw predictions for real parallel workloads, using the node-level view is more appropriate since real workloads usually fully subscribe the available cores. Furthermore, running parallelized workloads also enables the models to take contention on shared resources (e.g., last-level cache and DRAM) into consideration.

Each kernel in our training suite is configured to run either in single or double precision. We also configure each of the kernels to work off of different levels in the memory hierarchy; that is, for each kernel, we have multiple working set sizes that fit in L1, L2, and last-level caches and main memory. This results in a total of eight configurations for each computational kernel.

Model Validation Applications: To validate the models, we use five application benchmarks (CG, FT, LU, MG, and SP) from the NAS parallel Benchmarks (NPBs) [2] and two co-design mini-applications from the Mantevo suite (miniGhost and CoMD) [23]. For all NPBs, we consider class C problems. We include all four stencil operations (5-, 7-, 9-, and 27-point) available in miniGhost in our evaluation. We consider both the Lennard-Jones (LJ) and embedded atom method (EAM) within CoMD. For all versions of miniGhost and CoMD, we consider a 128^3-sized grid for our evaluations. We compiled all our tests with *gcc-4.6.3* and the *-O2* flag.

For all the application benchmarks, we first profile the codes to determine hot-spot loops. We then manually instrument the source code to collect hardware counters around such loops. Intel's documentation states that RAPL counters are updated once every millisecond. However, others have noted that such updates do not occur at such intervals [21]. To ensure we have a sufficient number of power readings for each of the hot loops, we only consider loops that have per-visit lengths of more than 5 ms.

Tools: To measure the hardware counters, we wrote a simple library-based tool that allows us to "register" compute loops in kernels and applications for hardware counter data collection. Internally, the tool uses PAPI to collect the hardware counters. Each MPI process produces its own output file, and the outputs are merged to generate the node-level characterization for computations.

Data Collection: The models developed in this work are based on performance hardware counters. These counters are available on all modern processors and record low-level microarchitectural events (e.g., number of L1 cache accesses, number of mis-predicted branches) and are accessible via special-purpose model-specific registers. Hardware-level parameters (e.g., CPU clock speed and power caps on CPU and DRAM) also affect power draw and performance of computations. To also encapsulate the effects of those parameters, we measure power and performance of computations at different CPU frequencies.

Training kernels and application loops are instrumented to measure all PAPI-supported hardware counters. Hardware counter collection can be a noisy process, and care must be taken to reduce the noise in the measurements. We take two steps to limit this noise. First, we limit the number of hardware counters that we measure at a time to the number of counters that can be measured without multiplexing. This means measuring at most 11 compatible counters at a time on the Ivy Bridge testbed. Second, we measure each hardware counter five times; from among these five measurements, we take the average of the three values that are closest to each other and discard the remaining two.

Component-level power draw is measured by using PAPI. We reset the RAPL energy counters at the start of each computational loop. At the completion of the loop, the counters are read again to capture the per-component energy required to run the loop. To derive power, we divide the energy measurements with time. We express performance as cycles per instruction (`cpi`).

Model Evaluation Metrics: To evaluate the predictive accuracy of the models, we rely on a set of metrics. The first of these is the arithmetic mean absolute prediction error (AMAPE), a simple and widely reported metric in the HPC literature. The main drawback of AMAPE is that it is sensitive to outliers and can even lead to misleading conclusions [36]. Given that the kernels and applications in our prediction set are diverse and have wide a range of run times and power draws, the distribution of prediction errors can be skewed. Therefore, we also rely on geometric mean absolute prediction error (GMAPE) and median absolute prediction error (MedAPE). We also report the usual ML metrics such as RMSE and R^2.

V. Experimental Results

We adopt a robust out-of-sample model validation strategy; first, from among all the microkernels, we select 75% for training (≈ 150 microkernels) and use them as the *kernel training set*—this is given as training data set $\mathcal{I}$ to the AutoMOMML. The remaining points in the kernel data set are tagged as the *kernel testing set*, which is then used to validate the models. This setting is based on an exploratory study in which we tested 25%, 50%, 75%, and 90% for *kernel training set* and selected reasonable data points in the kernel testing set. Note that sampling is done by using the names of the microkernels, which will ensure that all configurations of any given microkernel (single- and double-precision versions with working set sizes that fit in the L1,

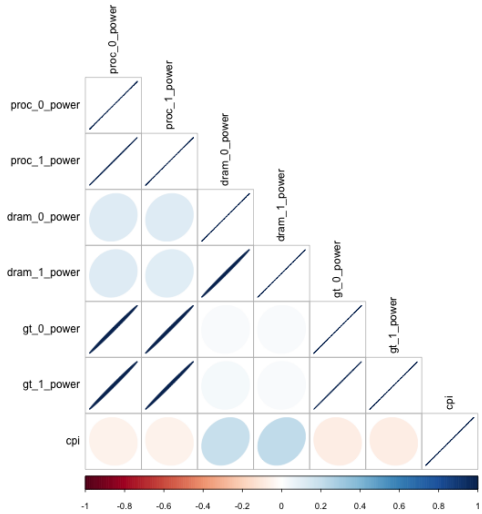

Figure 3. Correlation analysis for output metrics.

L2 and last-level caches and memory) belong to either the training or test set. Furthermore, we validate the models by using all points from the mini-applications, which are referred to as the *application testing set*.

AutoMOMML comprises few high-level component level parameters that can potentially affect the tradeoff between accuracy and the model building time. Based on component-level exploratory studies on the kernel training set, we set and recommend the following settings as default. The cutoff value in the correlation module is set to 0.90, and the number E of intervals in the balanced sampling module is set to 10. For the input selection, the number of folds K is set to 10, and the tolerance level, δ, is set to 1%. For each output, we generate 10 subset sizes by generating a sequence of 10 equally spaced values from from 3 to $|\mathcal{I}_s|$. When the equally spaced value is not an integer, it is rounded off to the nearest integer.

A. Modeling Complexity Reduction

Data Preprocessing, Correlation Analysis, and Balanced Sampling: We observe that several inputs are highly correlated. Among the 40 PAPI counters, only 20 counters are uncorrelated. Highly correlated counters include data cache misses on L1 and L2 data cache accesses. These sets of counters effectively measure the same underlying phenomenon. The results from the correlation analysis module for the outputs are shown in Figure 3. Of 7 outputs, only 3 outputs are uncorrelated—`dram_0_power` and `proc_0_power` values are highly correlated to `dram_1_power` and `proc_1_power`, respectively. This is to be expected because we run the exact same workload (recall that we do so in an embarrassingly parallel mode) on both sockets of the testbed. Therefore, given dram and processor power for one socket, it will be straightforward to predict the values for the other socket. From this phase, AutoMOMML applies the rest of the modules in the

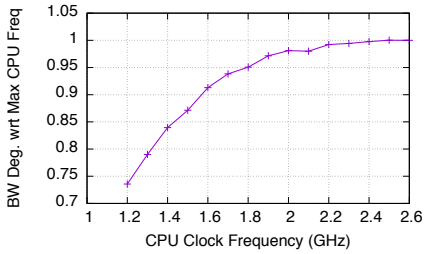

Figure 5. Memory performance at different CPU clock frequencies.

pipeline for each of the three outputs (`dram_0_power`, `proc_0_power`, and `cpi`) independently.

Input Importance and Input Selection: Figure 4 shows the results from the input importance module. The plots show the impact of the input on the three outputs using permutation accuracy importance of the `rf` method. The x-axis shows percentage increase in MSE (%IncMSE) after permuting the input column x^i in the training set. For `dram_0_power`, we observe that on-chip (L1 and L2) and off-chip (L3) cache-related activities (reads/writes/accesses/misses for both data and instructions) emerge as the most significant inputs. CPU clock frequency (freq) appears as the second most important input. To explain this counterintuitive observation, we took note of previous work [33] and used the lmbench benchmark [28] to measure memory read bandwidth across different CPU clock frequencies on our testbed. The results are plotted in Figure 5. The curve shows that memory performance (measured in terms of the read bandwidth) degrades by roughly 26% when CPU clock frequency is reduced to 1.2 GHz from 2.6 GHz. The power drawn by DRAM at 1.2 GHz is roughly 7% lower than the power drawn at 2.6 GHz CPU clock frequency. A particularly interesting entry in the rankings is the branch mispredicted event. We attribute this to the potential of branch mispredictions to increase instruction cache misses by fetching the wrong instruction streams [1]. If the instruction footprint is larger than the exclusive L1 instruction cache, the trips to memory for instructions can significantly increase if instructions cached in inclusive L2 (and L3) caches are frequently evicted because of contention with data. TLB data misses also contribute to the DRAM power draw. Each TLB miss (on data or instruction) triggers a load from main memory in addition to a page walk. TLB misses will, therefore, also have implications for the `cpi`.

For the `proc_0_power`, the most significant parameter is freq, which is followed by memory, floating point, TLB, and cache-related events. For cpi, events related to memory (loads and stores) and branch units are the most significant. Memory and branch units contribute heavily to CPU stalls (or wasted cycles). Memory-related stalls are mainly due to poor data locality, which leads to poor cache usage. The performance of branch units is important because more than 10% of the total instructions in the microkernels, on average, are branch instructions. Whether or not a given branch is

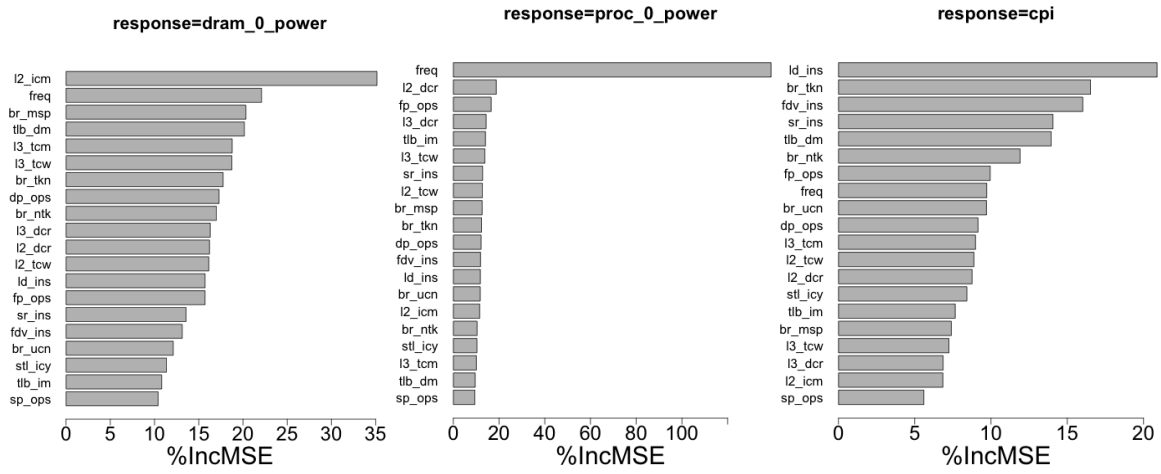

Figure 4. Input importance with permutation accuracy importance.

accurately predicted therefore has a significant impact on performance. Even though branch predictor units in modern Intel processors are highly accurate, branch mispredictions incur a high penalty by requiring a complete flush of the deep instruction pipeline.

Using the results from this module, AutoMOMML removes inputs that do not have significant impact on the output. For each output, when %IncMSE for an input x^i is less than 5%, it is removed from the predictor list for the corresponding output. Nevertheless, the results show that no input is insignificant for the three outputs. Note that the `rf` method is effective in identifying impactful parameters but it has limitations in detecting insignificant inputs [20].

Figure 6 shows the results from the input selection module. Algorithm 1 is run for each output with 10 subset sizes. The general trend is that increasing the number of inputs decreases RMSE, but the reduction becomes insignificant after a certain number of inputs. The results also show the number of inputs under various tolerance levels. For the adopted default tolerance of 1%, AutoMOMML selects 8, 6, and 13 inputs for `dram_0_power`, `proc_0_power`, and `cpi`, respectively. The selected j inputs for an output correspond to the top j inputs for the same output in Figure 4.

B. Model Selection

Each learning method is configured to run with 30 parameter configurations. The best parameter setting is obtained from the 10-fold cross-validation, as described in Algorithm 2. Figure 7 shows the box plots obtained from 10 RMSE values of each method with its best parameter setting. As evidenced by the box plot, the `t-test` establishes different model combinations based on the given output: for `dram_0_power`, `rf` and `sgb` are selected for bagging; for `cpi` `rf`, `cbt`, and `sgb` require being combined; for `proc_0_power`, `sgb` outperforms all other models.

To build the final predictive models, for each output, the selected methods are configured with their corresponding best parameter setting and retrained with the kernel training set (150 microkernels). Given an unseen point, the predicted value is given by the arithmetic mean of predicted values from the corresponding models — e.g., `cpi` prediction is given by the mean of predicted values from `rf`, `cbt`, and `sgb` models.

C. Model Validation

Table I summarizes the validation results on kernel and application testing set. On the kernel testing set, we observe a high prediction accuracy for `dram_0_power` and `proc_0_power` with AMAPE, GMAPE, and MedAPE values within 5%. MAPE's sensitivity to a few outliers is evident in the case of `cpi`. While the MAPE for `cpi` is 17.4%, GMAPE and MedAPE values are 8.57% and 13.3%, respectively. We also note that R^2 value for `cpi` is 0.93, which suggests that the model accurately captures the relationship between inputs and outputs well and that the model can be effective in comparing two competing code optimization strategies in an autotuning (e.g., selection of code variants) or run-time environment (e.g., selection of CPU clock frequency).

The results with the application testing set are promising and show a trend similar to that of kernel testing set. In particular, for `cpi` and `proc_0_power`, AMAPE, GMAPE, MedAPE, and R^2 values are similar to the values observed in the kernel testing set. Despite the fact that AMAPE for `dram_0_power` is 15.9%, GMAPE and MedAPE are not more than 8%. The RMSE value shows that, on average, the prediction is off only by 2.36 W. Closer examination of the results for `dram_0_power` prediction reveals that one instance of the NAS Parallel Benchmark, FT, shows a rather high prediction error ($\sim$30%). Compared to other mini-applications and kernel testing set, FT's computational loops tend to have large number of function calls, which can affect instruction cache performance and execute significantly large number of branch instructions.

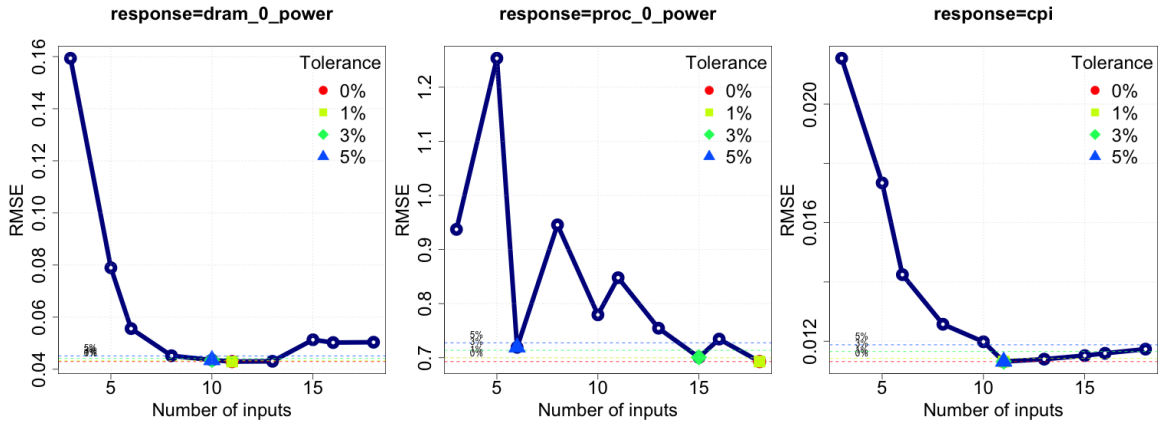

Figure 6. Model-based input selection results.

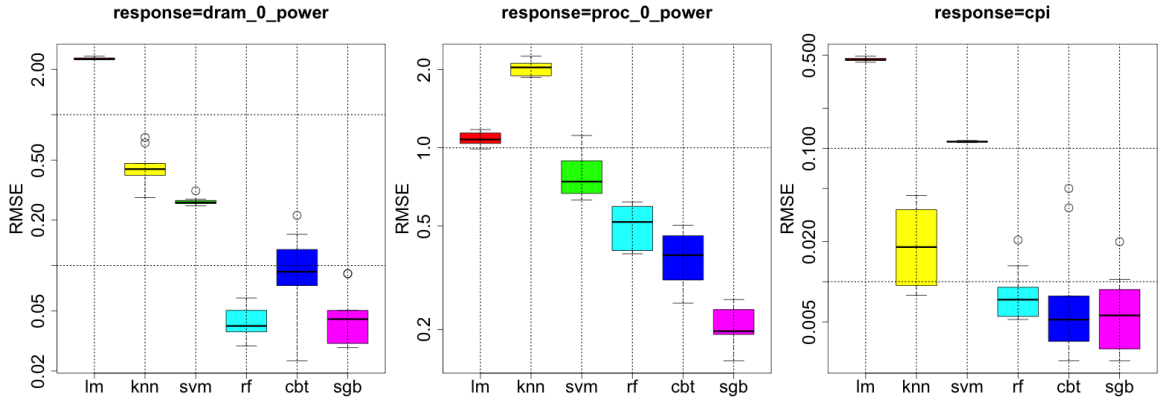

Figure 7. Model selection results.

Table I
MODEL VALIDATION (BAGGING) RESULTS

Response	R^2	RMSE	AMAPE (%)	MedAPE (%)	GMAPE (%)
Kernel Testing Set					
proc_0_power	0.99	0.84	1.41	0.94	0.76
dram_0_power	0.89	1.00	4.60	1.23	1.50
cpi	0.93	0.19	17.4	13.3	8.57
Application Testing Set					
proc_0_power	0.95	1.98	3.11	2.30	1.87
dram_0_power	0.45	2.36	15.9	8.00	7.16
cpi	0.91	0.47	15.5	12.9	7.63

VI. RELATED WORK

The closest related work is the MuMMI [38] end-to-end automatic multi-objective modeling framework. It requires that training and testing points come from the same kernel/application. Hence it uses linear correlation for input selection and a linear model to capture the relationship between PAPI counters and performance, power, and energy.

From a methodological perspective, in [30] six supervised-learning methods are used to learn the relationship between hardware counters, source-code transformation parameters (tiling, parallelization, vectorization, and data locality improvement) and performance. It is a semi-automatic approach because input importance and selection and model selection are manually driven. In [35], kernel-specific surrogate models built by using artificial neural networks were used to model the relationship between compiler transformation parameters and objectives such as power draw, execution time, and energy usage of HPC kernels.

Research in model-guided autotuning has focused on developing online surrogate models for performance [31, 18, 27, 34]. In [12, 37, 13, 19], the authors developed online surrogate models for several scientific kernels on multicore architectures. In [6], the authors adopted boosted regression trees for obtaining online surrogate models for a GPU implementation of an image-filtering kernel.

Performance counters have been used to develop predictive power models in multiple research projects [8, 24, 9]. Bertran et al. [8] use the notion of "power components" (closely related architectural elements), develop microkernels that stress those components separately, identify a set of performance counters that can be used to quantify such stress, and use those performance counters to develop linear-regression-based power models. Isci and Martonosi [24] use a similar approach to identify a set of performance counters that can used to approximate the activities within key architectural components. The hardware counters are then used to develop component-level power models. Bircher and John [9] use performance hardware counters to develop power models that can predict system level power usage as

a combination of component-level power usage.

VII. Conclusion

Automated modeling techniques have the potential to provide valuable hints for proactive and/or reactive steering of extreme scale systems towards better energy efficiency and reliability. Towards that end, this paper presented the AutoMOMML framework, a general-purpose machine-learning based framework for modeling multiple objectives. We applied the framework to model power and performance on a widely used Intel architecture and showed that the framework is capable of 1) producing highly accurate models for real-world application benchmarks, and 2) providing valuable information on how the modeled objectives relate to the properties of applications and power and performance related hardware parameters. Our work in this area is only beginning. We are currently pursuing multiple exciting research avenues – 1) incorporating AutoMOMML within an autotuning framework, and 2) incorporating the framework within an energy-aware computing framework.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. Int. Conf. Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, 1999.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks–summary and preliminary results. In *Proc. ACM/IEEE Conf. Supercomp.*, SC '91, New York, 1991.

[3] P. Balaprakash, A. Tiwari, and S. M. Wild. Multi objective optimization of HPC kernels for performance, power, and energy. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 239–260. Springer, 2014.

[4] P. Balaprakash, S. Wild, and B. Norris. SPAPT: Search problems in automatic performance tuning. *Proc. Comp. Sci.*, 9:1959–1968, 2012.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, 2012.

[6] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar'12)*, pages 1–9. IEEE, 2012.

[7] M. Berry, T. E. Potok, P. Balaprakash, H. Hoffmann, R. Vatsavai, and Prabhat. Machine learning and understanding for intelligent extreme scale scientific computing and discovery. Technical report, 2015.

[8] R. Bertran, M. González, X. Martorell, N. Navarro, and E. Ayguadé. A systematic methodology to generate decomposable and responsive power models for CMPs. *IEEE Trans. Comp.*, 62(7):1289–1302, 2013.

[9] W. L. Bircher and L. K. John. Complete system power estimation: A trickle-down approach based on performance events. In *Int. Sym. on Perf. Anal. of Sys. & Soft.*, ISPASS '07, pages 158–168. IEEE, 2007.

[10] C. M. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer, New York, 2006.

[11] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

[12] E. A. Brewer. High-level optimization via automated statistical modeling. *ACM SIGPLAN Notices*, 30(8):80–91, 1995.

[13] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *IEEE Int. Symp. Code Gen. Opt. (CGO'07)*, pages 185–197, 2007.

[14] C. Chen, J. Chame, and M. W. Hall. CHiLL: A framework for composing high-level loop transformations. TR 08-897, Univ. of Southern California, Jun 2008.

[15] R. S. Chen and J. K. Hollingsworth. Angel: A hierarchical approach to multi-objective online auto-tuning. In *Int. Workshop on Runtime and Operating Systems for Supercomp.*, ROSS '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[16] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, Aug 2010.

[17] J. H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, 2002.

[18] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, et al. MILEPOST GCC: Machine learning based research compiler. In *GCC Summit*, 2008.

[19] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *IEEE Int. Conf. Data Engineering (ICDE'09)*, pages 592–603, 2009.

[20] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010.

[21] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan. 2012.

[22] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Scholkopf. Support vector machines. *IEEE Intel. Sys. App.*, 13(4):18–28, 1998.

[23] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[24] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proc. ACM/IEEE Conf. Supercomp.*, SC '12, pages 10:1–10:12. IEEE Computer Society Press, 2012.

[26] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The PAPI cross-platform interface to hardware performance counters. In *Dept. Defense Users' Group Conf. Proc.*, pages 18–21, 2001.

[27] A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proc. Int. Conf. High Perf. Comp. Networking Storage Anal.*, SC '13, 2013.

[28] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *USENIX Annual Tech. Conf.*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[29] B. Norris, A. Hartono, and W. Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, pages 443–462. Chapman & Hall, 2007.

[30] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *Int. J. Parallel Programming*, 41(5):704–750, 2013.

[31] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *ACM Int. Conf. Compilers Arch. Synth. Embed. Sys.*, pages 65–74, 2011.

[32] L.-N. Pouchet. *PolyBench: The polyhedral benchmark suite*, 2012. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[33] R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: an analysis of current x86 64 processors. *Proc. USENIX Conf. Power-Aware Comp. Sys.*, 2012.

[34] O. Spillinger, D. Eliahu, A. Fox, and J. Demmel. Matrix multiplication algorithm selection with support vector machines. 2015.

[35] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of HPC kernels. In *IEEE Int. Conf. Par. Distrib. Proc. Symp. Workshops (IPDPSW12)*, pages 990–998, 2012.

[36] C. Tofallis. A better measure of relative prediction accuracy for model selection and model estimation. *J. Oper. Res. Soc.*, 2014.

[37] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perf. Comput. Appl.*, 18(1):65–94, 2004.

[38] X. Wu, C. Lively, V. Taylor, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, D. Terpstra, and V. Weaver. Mummi: Multiple metrics modeling infrastructure. In *ACIS Int. Conf. Softw. Engin., Artif. Intel. Network. Parallel/Distributed Comp. (SNPD)*, pages 289–295, 2013.