Modeling Power and Energy Usage of HPC Kernels

Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, Allan Snavely
Performance Modeling and Characterization Laboratory
San Diego Supercomputer Center
San Diego, California, USA
{tiwari, michaell, lcarring, allans}@sdsc.edu

***Abstract*—Compute intensive kernels make up the majority of execution time in HPC applications. Therefore, many of the power draw and energy consumption traits of HPC applications can be characterized in terms of the power draw and energy consumption of these constituent kernels. Given that power and energy-related constraints have emerged as major design impediments for exascale systems, it is crucial to develop a greater understanding of how kernels behave in terms of power/energy when subjected to different compiler-based optimizations and different hardware settings. In this work, we develop CPU and DIMM power and energy models for three extensively utilized HPC kernels by training artificial neural networks. These networks are trained using empirical data gathered on the target architecture. The models utilize kernel-specific compiler-based optimization parameters and hardware tunables as inputs and make predictions for the power draw rate and energy consumption of system components. The resulting power draw and energy usage predictions have an absolute error rate that averages less than 5.5% for three important kernels – matrix multiplication (MM), stencil computation and LU factorization.**

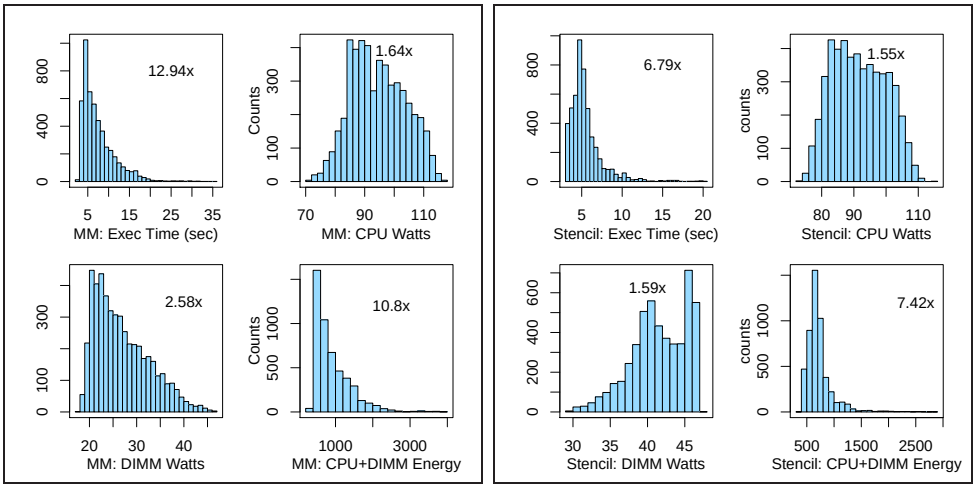
I. Introduction

Compute intensive kernels make up the majority of execution time in large scale HPC applications [8]. These kernels are often executed a large number of times during a single execution of an application. A lot of recent research efforts have therefore been directed towards automatically identifying these kernels based on attributes such as computational demand and memory access patterns [17], and towards developing compiler-based optimization strategies to help improve their performance. Gaining a better understanding of the power and energy usage behavior of these kernels when subjected to various compiler-based optimizations and different hardware-related tunables is also very important given that the *power wall* has emerged as the key bottleneck in the design of exascale systems [11]. Models can be used to aid in developing this understanding.

We focus on three extensively studied kernels — MM, stencil computation and LU factorization. Over the years, various optimization strategies that target specific resource(s) in computing platforms have been developed to reduce execution time of these kernels. Some examples of such strategies include loop tiling and unrolling to target reuses in caches and registers respectively. Most of these tuning strategies are *parametrized*. That is, they expose a set of optimization-related parameters that can be selected to change the ways optimizations are applied to a piece of code. An exhaustive approach to evaluating all possible optimizations in this parameter space is often not practical because these spaces are multi-dimensional — i.e., a given piece of code can benefit from the simultaneous application of many strategies which can result in billions (or more!) possible optimization parameter combinations.

Recently, various empirical auto-tuners [24] [22] have successfully used search or heuristics based approaches to generate and evaluate only a manageable subset of the optimization parameter space for computational kernels and select the code variant that obtains the best performance in terms of execution time. The work presented in this paper draws inspiration from the success of these empirical approaches to optimization. However, rather than using a search-based approach to navigate the parameter space, we develop kernel-specific power draw and energy usage models for certain system components using artificial neural networks (ANNs).

The models utilize kernel-specific compiler-based optimization parameters and hardware tunables as inputs and can make predictions for power draw and energy usage of the CPUs and DIMMs for all code variants in the parameter space. The ANNs that form the basis of the models are trained using empirical power and performance data gathered for the kernel on the target architecture. These training data points are drawn randomly from the parameter combination space and constitute a very small portion (less than 3%) of the



(a) (b)

Fig. 1. Distributions of execution time (seconds), CPU and DIMM Power Draw (Watts) and CPU+DIMM Energy usage (Joules) for (a) MM and (b) stencil code variants along with the $\left(\frac{max}{min}\right)$ ratio for the respective entities shown in the histograms. The code variants utilize different tiling factors, unrolling factors and were run using 7 available CPU clock frequencies on a Xeon. Input data size is kept constant.

total points in the space.

The optimization strategies that we apply to each of the computational kernels have been shown to improve those kernels' performance in various research articles. The focus here is not to establish or verify the applicability of these strategies. Rather the focus is on whether we can use the optimization parameters to accurately predict the power draw rate of individual system components.

The power draw and energy usage landscape can be quite large and varied. Figure 1 shows histograms of CPU power draw, DIMM power draw and CPU+DIMM energy consumption collected for a small, random subset of the possible code variants for MM and stencil kernels. The best and the worst performing code variants for MM on a Xeon system (described in Section III-A), for example, are over an order of magnitude different in terms of CPU+DIMM energy usage. Therefore characterizing these differences will help understand the impact of compiler optimizations on energy usage.

II. Methodology

We model power and energy for a parameter combination space of a particular kernel by measuring execution time and power (and therefore energy) for a small subset of the points in that kernel's optimization parameter space. These measured points are then used as training input for a set of ANNs whose outputs are models of power draw, execution time, and energy usage of that kernel for all optimization parameter combinations.

A. Problem Formulation

For each model, as shown in Equation 1, we develop a function f, whose inputs are a set of optimization parameters $x_1 \ \cdots \ x_n$ and whose output y is some measure of the efficacy of the optimization (in terms of power, time or energy) on the computation.

$$y = f(x_1 \ \cdots \ x_n) \tag{1}$$

Equation 2 shows a specific formulation of Equation 1 as the model of DIMM power draw (P_d) for kernel MM in terms of the compiler-level optimization parameters (ti, tj, tk, ui and uj – loop tiling and unrolling factors), CPU clock frequency ($freq$) and matrix size ($msize$).

$$P_d = f(ti, tj, tk, ui, uj, msize, freq) \tag{2}$$

For every such model, f is determined by exposing the problem to an artificial neural network.

B. Artificial Neural Networks

An artificial neural network (ANN) is a computing system that consists of densely interconnected and adaptive processing elements that are highly capable of knowledge discovery in the input dataset [2] [5]. ANNs, which have been used successfully in the area of HPC in the past to derive performance models for scientific applications [9] [15], have several desirable properties that make them attractive as the approach for performance, power and energy prediction problems.

1. ANNs can capture complex linear and non-linear interactions between the input parameters and between

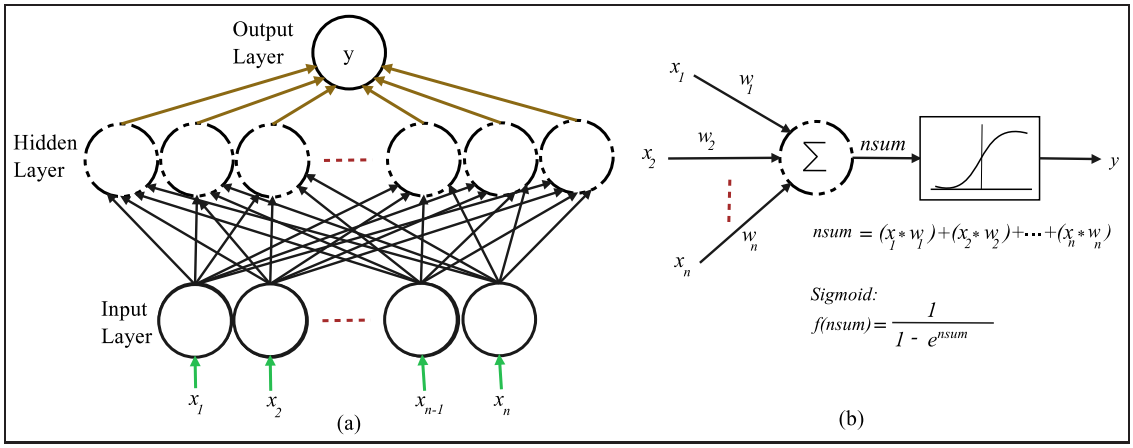


Fig. 2. (a) Network diagram for ANN. (b) Illustration of the learning process within a single network unit (adapted from [9] and [10])

the input and output parameters. Often these types of interactions are difficult to discover and even when they are available, it is challenging to completely and accurately describe them.

2. Generally ANNs work well even in the presence of some noise in the training data [10], [2]. Noise in our empirically collected data can be caused by OS jitter, other processes running on the system, or limitations of the hardware measurement infrastructure [3].

3. Most classical optimization algorithms require the definition of the objective function. However, for performance and energy modeling problems we may not know these functions a priori. ANNs do not need a formulation of the objective function and are therefore a good match for performance, power and energy modeling. The fact that no objective function definition is required to train ANNs offers another benefit — the models are generated using no domain-specific knowledge, meaning that the modeling techniques are more likely to be useful in other application/kernel domains.

C. Theory

While ANNs have been used in tackling various complex real-world problems — pattern classification, clustering, optimization, etc. [2], this work uses ANNs to approximate the function f in Equation 1. In the context of function approximation, ANNs take a series of predictors (input parameters $x_1 \cdots x_n$ in Equation 1) and produce a map of those input parameters to one or more outcomes (y in Equation 1). The network consists of a set of layers – an input layer, one or more hidden layers and an output layer. Figure 2(a) provides a schematic diagram for the network that is utilized in this work. The network is a *fully-connected single hidden layer feed-forward network* with one output unit. The number of nodes in the input layer is equal to the number of predictors. In fully-connected networks, all nodes in a given layer are connected to all the nodes immediately following that layer. Each edge that connects two nodes has an associated weight. Feed-forward networks have no cycles in their connection topography.

Developing a model requires two phases – training and validation. During the training phase, a set of empirically gathered data points are presented to the input layer sequentially. The input layer simply passes the incoming values to the hidden units. The hidden units take a weighted sum of the incoming values ($nsum$ in Figure 2(b)) and pass the weighted sum through an activation function. The activation function that we use is the sigmoid function (see Figure 2(b)). Sigmoid, which is a strictly increasing function, is the most commonly used activation function in ANNs because it exhibits smoothness and has desirable asymptotic properties [10].

At the output layer, the contributions from each of the hidden units are summed to derive a predicted value. The calculated error in prediction is back-propagated to update the edge weights using the gradient descent method [2]. Edges are iteratively updated until the model's predicted value is within some small error range when compared to the observed value.

We now discuss specific issues that affect the predictive accuracy of ANNs and schemes to address them.

1. Over-fitting the training data (describing noise in the data rather than the underlying relationships) is a danger when using machine learning techniques such as ANN because it results in degraded prediction accuracy on non-training inputs. To avoid this we use the k-fold cross validation method during the validation phase, which has been shown to help mitigate the problem of

over-fitting [20]. In k-fold cross validation, the training dataset is randomly partitioned into k equal-sized subsets. k different models are then constructed, each using $k-1$ of the k partitions as training input so that 1 of the k sets can be set aside for model validation. Each of the k models are then validated against the validation set and the model that yields the minimum error is selected.

2. Data scaling and centering is needed to make sure that large values of some input and output parameters do not unduly affect the learning process. Several techniques have been proposed to achieve this [2]. In this work, taking the logarithm of the predictors and the outcomes was enough to construct networks with stable accuracy.

3. The number of hidden units also play an important role. While there are many techniques that can be used to determine the "right" number of hidden units, our experience showed that setting the number of hidden units equal to twice the number of predictor variables works well for our purposes.

III. Experimental Setup

This section describes our experimental setup, which consists of an Intel Xeon workstation equipped with a DC power measurement harness. We utilize a source-to-source code transformer to generate code variants.

A. Experimental Platform

The experiments were conducted on an Intel Xeon E5530 workstation. The E5530 has 2 quad-core processors. Each core has its own 32KB L1 cache and 256KB L2 cache. Each of the quad-core processors has a shared 8MB L3 cache (for a total of 16MB of L3 for the 8 cores). The processors can be clocked at 1.60, 1.73, 1.86, 2.00, 2.13, 2.26, or 2.39 GHz. Processor clock frequency is changed using the `cpufreq-utils` package [1] that is available with many popular Linux distributions.

B. Power Measurement

To derive system component-level power measurements, we utilize a PowerMon2 apparatus [3]. PowerMon2 is a hardware and software framework designed to obtain fine-grained current and voltage measurements for different components of a target system such as CPUs, memory subsystem, disks, GPUs, etc. The device sits between an ATX power supply and the power inputs of various system components and reports measurements on up to eight channels via a PC's USB port. We identify the DC-rails that supply power to the CPUs and DIMMs and collect the measurements for those rails for each test.

C. Code Variants Generation

Code variants representing different loop optimization strategies are generated using CHiLL [7], which is a polyhedral loop transformation and code generation framework. CHiLL provides a high-level script interface that we leverage to describe a set of loop transformation strategies and instantiate their associated parameters for a given piece of code. In CHiLL nomenclature, the scripts that describe loop transformations are called "recipes".

IV. Experimental Results

These experiments are designed to study the effect of training dataset size of the artificial neural networks on the overall predictive accuracy of the trained networks. Recall that the training data for the models are gathered empirically — i.e. for each of the points in the training dataset, a corresponding code variant is generated using CHiLL; the generated variant is compiled and executed on the target architecture to measure execution time and component-level power draws. Therefore, the time cost of the modeling step is directly related to the size of the training set. For each of the three kernels studied in this paper, we use a four-step process to construct and validate models. First, from the entire parameter space, we randomly sample a set of points (N) and generate, compile and evaluate the corresponding code variants. Second, we select a set of random samples, V, from N and use that subset for model validation. Third, we construct training datasets from $(N \setminus V)$ with varying numbers of points and train separate neural networks. We repeat steps two and three for each of the properties we are attempting to model — CPU power draw, DIMM power draw and execution time. Finally, the trained models are used to predict the outcomes for all points in V. The CPU power draw, DIMM power draw and execution time models are then used to predict CPU+DIMM energy usage.

We use R, an open-source statistical computing environment, to automate the process of model training and validation. R packages `caret` [12] and `nnet` [23] are leveraged to generate the ANN models.

A. Matrix Multiplication (MM)

We provide a simple to understand MM implementation in Table I. We refer the readers to Tiwari et. al. [22] for details on the optimization strategy for MM. The strategy that we use for MM exploits the reuse of array c in registers, and the reuse of arrays a and b in caches. Data copying is applied to avoid conflict misses. The set of seven free parameters ti, tj, tk (tiling factors for

TABLE I
KERNELS USED FOR EXPERIMENTS

Kernel	Naive Code	Parameters	Parameter Domains

MM [22]‡	`do i = 1, n`
	`  do k = 1, n`
	`  do j = 1, n`
	`  c(i,j) = c(i,j)+a(i,k)*b(k,j)`
		ti, tj, tk (i,j,k tiles)
		ui, uj (i,j unrolls)
		CPU freqs ($freq$)
		matrix sizes ($msize$)
			$ti, tj, tk \in [2,4,8,\ldots,1024]$
			$ui \in [factors(ti)]$, $ui \leq 8$
			$uj \in [factors(tj)]$, $uj \leq 8$
			$ui \neq ti$, $uj \neq tj$
			$msize \in [1300, 1400, \cdots, 2000]$
			Total points: 647360

Stencil [22]‡	`do k=2, n-1`
	`do j=2, n-1`
	`do i=2, n-1`
	`a(i,j,k)=c*(b(i-1,j,k)+b(i+1,j,k)`
	`+b(i,j-1,k)+b(i,j+1,k)`
	`+b(i,j,k-1)+b(i,j,k+1))`
		ti, tj, tk (i,j,k tiles)
		ui, uj (i,j unrolls)
		CPU freqs ($freq$)
			$ti, tj, tk \in [2,4,8,\cdots,512]$
			$ui \in [factors(ti)]$, $ui \leq 8$
			$uj \in [factors(tj)]$, $uj \leq 8$
			$ui \neq ti$, $uj \neq tj$
			Total points: 56700

LU [7]‡	`do k=1, n-1`
	`do i = k+1, n`
	`a(i,k)=a(i,k)/a(k,k)`
	`do i = k+1, n`
	`do j = k+1, n`
	`a(i,j)=a(i,j)-a(i,k)*a(k,j)`
		tj (loop j tile)
		ti_1, tj_1, tk_1 (trsm tiles)
		ui_1, uj_1 (trsm unrolls)
		ti_2, tj_2, tk_2 (MM tiles)
		ui_2, uj_2 (MM unrolls)
		matrix size ($msize$), $freq$
			$ti_*, tj_*, tk_* \in [2,4,8 \cdots 1024]$
			$ui_* \in [factors(ti_*)]$, $ui_* \leq 8$
			$uj_* \in [factors(tj_*)]$, $uj_* \leq 8$
			$ui_* \neq ti_*$, $uj_* \neq tj_*$
			$msize \in [1500, 1600, \cdots, 2200]$
			Total points: 8.34e+10

‡ Due to space limitations, we are not able to provide full transformation recipes. Please refer to the cited articles for full recipes.

loops i, j and k), ui, uj (unrolling factors for loops i and j), $msize$ and $freq$ constitute the set of predictors. Table I provides further details on the parameter space and the constraints on the predictor variables. The total number of points in the parameter space is 6.5×10^5.

We generate a total of 8285 different code variants (N) that utilize randomly selected optimization parameters, CPU clock frequency and input data size. We set aside the majority (65%) of the points in the model validation set V. We construct separate ANN models for CPU power draw, DIMM power draw and execution time using various training dataset sizes — these training subsets are drawn from roughly 2900 points not set aside for validation. Each of the models is then used to predict the CPU and DIMM power draw and execution time for the points in V. Figure 3 shows the absolute mean error percentage and standard deviation of the error percentage on the y-axis and the training dataset size on the x-axis for MM execution time models. As the size of the training dataset increases, the error generally improves as well; the same holds true for standard deviation of the error. What is interesting, however, is that the improvements flatlines after the training dataset size is larger than around 1200 elements. This suggests that a fairly small time investment in collecting empirical training data can give us a execution time model, which has an average error rate of approximately 5.47%.

Table II summarizes the training set size sensitivity results for CPU and DIMM power draw and execution time. Energy predictions made for the points in V using the CPU, DIMM power draw and execution time models constructed with 1400 points, have an average absolute

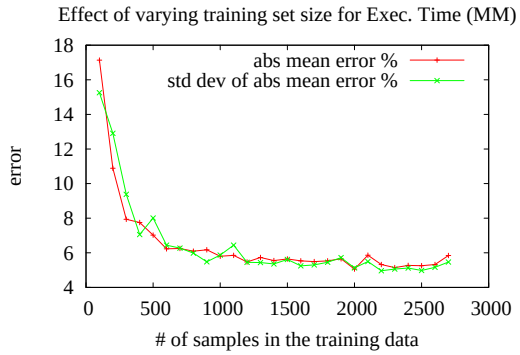


Fig. 3. Training set size sensitivity — MM Execution Time

error of 4.9%. The training dataset consisting of 1400 points corresponds to a very small portion (0.22%) of the total points in the parameter space. A simple extrapolation of the measured execution times suggests that it would require more than 86 days[1] of execution time to evaluate the entire parameter space. Evaluating 1400 points to construct the training set takes roughly 4.6 hours. This contrast highlights one of the major benefits of our modeling technique — we are able to give accurate power and energy predictions using hours, not months, of system execution time.

Finally, Figure 4 shows the modeled versus observed values for CPU and DIMM energy consumption for MM. The red line in the graph is the trend line and for a well-behaved model, this line should be a roughly 45 degree line, which is the case in Figure 4.

[1]Using the average of measured times (11.6s) for all 647360 points.

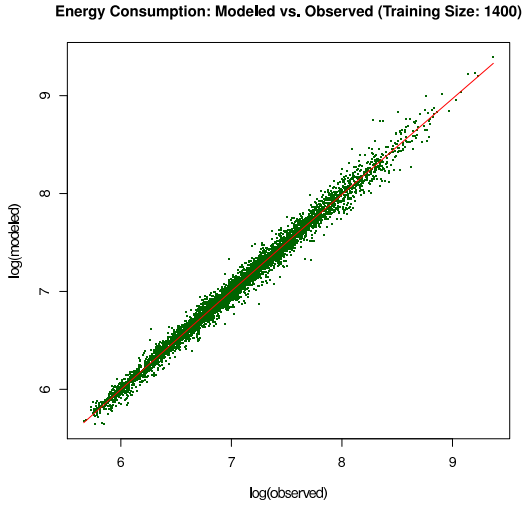


Fig. 4. Modeled vs. Obs. CPU+DIMM energy for MM (log scale)

TABLE II
TRAINING SET SIZE SENSITIVITY

Outcome	Training set size	abs mean error[1] %	std dev
MM			
CPU power draw	900	1.61	1.58
DIMM power draw	1400	3.34	3.13
Execution time	1200	5.47	5.44
Energy prediction	1400	4.89	4.69
Stencil			
CPU power draw	1200	1.08	1.00
DIMM power draw	1100	1.84	1.84
Execution time	1600	4.64	4.25
Energy prediction	1500	3.95	3.66
LU			
CPU power draw	1400	0.98	0.78
DIMM power draw	1500	2.70	2.45
Execution time	1100	4.50	3.81
Energy prediction	1600	3.94	3.34

[1] $error = abs(\frac{observed - modeled}{observed}) \times 100$

B. Stencil

We provide a naïve stencil (in this case a Jacobi) implementation in Table I. We refer readers to Tiwari et. al. [22] for the detailed optimization strategy for stencil. Since array b has reuse on three dimensions, the loops are tiled on three dimensions for reuse in caches. Loops i and j are unrolled for register reuse. The free parameters ti, tj, tk, ui, uj and $freq$ form a five-dimensional parameter space with 5.67×10^4 total points.

For stencil modeling, we randomly sampled the parameter space for a total of 4900 points. Of these 4900 points, we set aside 40% of the points[2] (1960) for model validation (V). From the remaining 2940 points, we trained separate networks using varying-sized training subsets. The networks constructed using those training datasets are validated against the points in V.

[2]Based on our experience with MM and LU modeling (see next section), we collected a smaller set of empirical data for stencil.

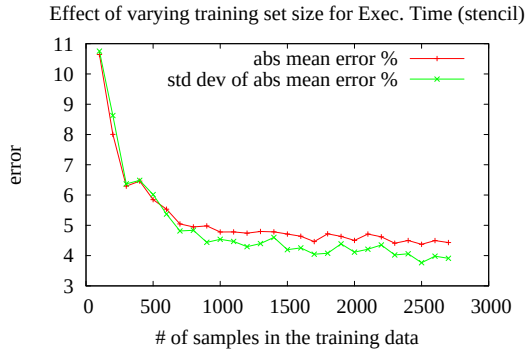


Fig. 5. Training set size sensitivity — Stencil Execution Time

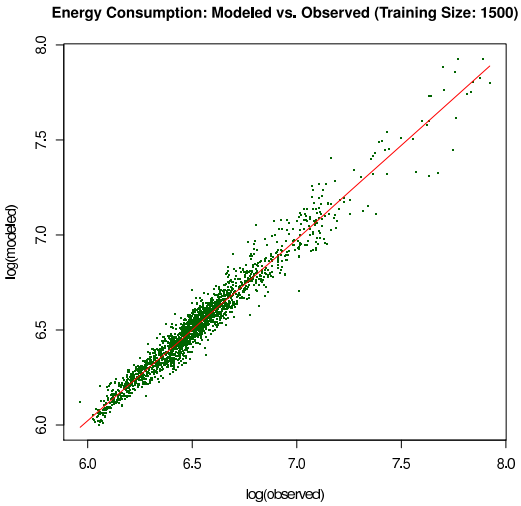


Fig. 6. Modeled vs. Obs. CPU+DIMM energy for stencil (log scale)

Figure 5 shows the ANN learning curve attributes (absolute error % and standard deviation) for stencil. The error percentage flatlines after the training dataset size of 1600 elements. At a training set size of 1600, the average error rate for execution time prediction is 4.6%.

Table II summarizes the training set size sensitivity results. Figure 6 shows the modeled vs. observed values for CPU+DIMM energy usage for this kernel. At a training set size of 1500, the average absolute error rate of energy prediction is 3.9%. The training dataset consisting of 1500 points corresponds to 2.6% of the total points in the parameter space. If one were to evaluate all the points in the parameter space, it would require more than 3.6 days[3] of execution time. Evaluating 1500 points to construct the training set requires roughly 2.2 hours.

[3]Using the average of measured times (5.4s) for all 56700 points.

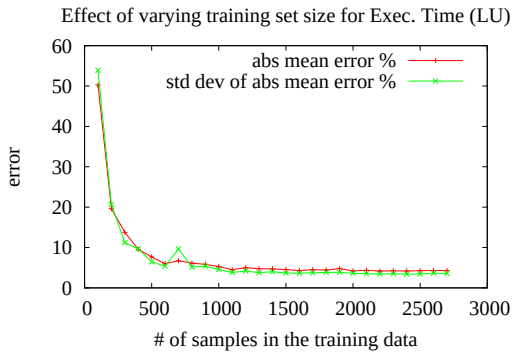


Fig. 7. Training set size sensitivity — LU Execution Time

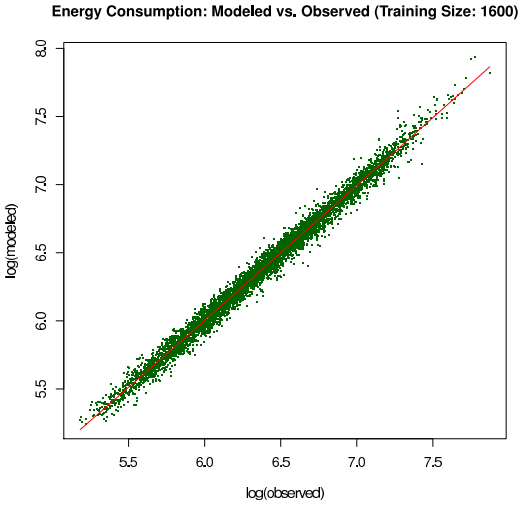


Fig. 8. Modeled vs. Obs. CPU+DIMM energy for LU (log scale)

C. LU Factorization

We provide a naïve LU implementation in Table I. The detailed description of how LU kernel is transformed is provided by Hall et. al. [7]. The idea is to first apply tiling to loop j and split the iteration space inside the j-tile control loop into an imperfect loopnest and a perfect loopnest. The perfect loop nest is then split into two sub-loopnests — one that has non-overlapping array accesses (a MM kernel) and the other that has overlapping array accesses (a triangular solve kernel). Each of these sub-loopnests are then independently optimized to target reuses in caches and registers. The recipe shown for LU transformation in [7] has a total of 11 free parameters (listed in Table I along with domain definitions and constraints). Together with the CPU clock frequency and input matrix size, we have a 13-dimensional parameter space with a total of approximately 8.32×10^{10} points.

For LU factorization modeling, we randomly sampled the parameter space for a total of 9700 points (approximately 1 in every 8.5 million points). Of these 9700 points, we set aside the majority (65%) of the points for model validation (V). From the remaining 3400 points, we developed models using varying-sized training subsets. The models constructed using those training datasets are validated against the points in V.

Figure 7 shows the absolute mean error percentage and standard deviation of the error percentage on the y-axis and the training dataset size on the x-axis for LU execution time models. We see a similar trend that we saw for MM. The error percentage flatlines after the training dataset size of 1100 elements (i.e. adding more than 1100 data points in the training set does not add new information to the model). At 1100 training set size, the average error rate for execution time prediction is 4.5%.

Table II summarizes the training set size sensitivity results. Figure 8 shows the modeled versus observed values for CPU+DIMM energy consumption for LU. Energy predictions made for points in V using the CPU, DIMM and execution time networks trained with 1600 points have an average absolute error rate of 3.9%. The training dataset consisting of 1600 points corresponds to a very small portion (less than 0.0000019%) of the total points in the parameter space. If one were to evaluate all the points in the parameter space, it would take more than 14500 years[4]. Evaluating 1600 points to construct a training set takes roughly 2.4 hours.

V. Discussion

In order to further validate the capability of the models, we sampled 666312 random points from the parameter space of LU (henceforth referred to as L) and used the energy model developed Section IV-C to predict the energy usage for each variant in L. The total time required to calculate these energy predictions on our Intel E5530 workstation is roughly 20 seconds, compared to an estimated 42 days to run and measure all points in L. For each matrix size in L, we then empirically gathered energy usage data for the kernel variants whose modeled energy consumption fell in either extremity (the 10 highest and 10 lowest for each of 8 matrix sizes). We call this set of 160 points E. The average absolute error percentage in energy prediction for points in E is 7.0% compared to 3.9% (see Table II): this error rate is still sufficiently accurate to enable interesting exploration of the space. Further analysis of the modeled vs. observed energy usage for the points in E revealed some interesting results. For instance, for $msize$=2100,

[4]Using the average of measured times (5.5s) for 8.32×10^{10} points.

the model found (and we empirically verified) a code variant that consumes 7.7% less energy than the lowest-energy point from the training dataset for this $msize$. Also, for $msize$=1900, the model found a code variant that consumes 33% more energy than the highest-energy variant for this $msize$ in the training set. This shows that even though the models are developed using a small number of measured points that have no guarantee of encapsulating the range of nominal results that can be expected from target kernel, that it is reasonable to extend the models over that entire range. Search-based auto-tuners, for example, can utilize our models to learn the shape of the space and to make better decisions in terms of where to start the navigation of the search space and what regions to avoid.

We also see various other use-cases for the models presented in this paper. For example, if power draw is a primary concern, one can build and query a model for component power draw. For example, for LU, we were able to find and empirically verify two parameter configurations that were different in energy usage by only 3%, but one configuration drew 20% less CPU and DIMM power.

The models developed in this work also find applicability in research based on exhaustive parameter sweeps in order to learn some feature(s) of a target system. Laurenzano et. al. [13] use the results of an exhaustive set of benchmark runs to guide dynamic clock frequency selections. Our models could reduce the number of actual benchmark runs required to fill their benchmark results space, greatly reducing the overhead of filling this space because most of it could be filled with modeled results.

VI. Related Work

Ipek et. al. [9] and Lee et. al. [15] use ANNs to predict execution time of HPC applications. Their models use application-level input parameters (e.g. processor topology, input datasize) as predictors. Our work uses kernel-specific compiler optimization parameters as inputs to predict energy usage by system components. Our models can certainly be used in conjunction with the models presented by Ipek et. al. and Lee et. al. to develop an end-to-end modeling infrastructure.

Various researchers have utilized hardware counters to develop power draw characterizations for HPC applications and kernels [21] [6] [16] [18]. Others have used processor performance events [4] and architectural parameters and parameters drawn from application's characteristics [14]. Singh et. al. [21] derive an analytic, workload-independent piece-wise linear power model that maps performance counters and temperature to energy usage. Lively et. al. [16] also use hardware counters to develop application-centric models for the runtime and power draw of the system. Rahman et. al. [18] use models based on hardware counters to estimate power draw of chip multiprocessors and use that information to guide the usage of various compiler optimizations. Our models do not use hardware counters and are solely based on compiler-level optimization parameters.

Seng et. al. [19] examine the effect of compiler optimization levels and a few specific compiler optimization flags on the energy usage of the Intel Pentium 4 processor. Rather than relying on optimization flags, we exercise a greater control over how different code transformation strategies are applied. Moreover, our technique can model component-level power and energy usage.

VII. Conclusion

In this work, we used artificial neural networks to predict component-level power draw and energy usage of certain HPC computational kernels. For each of the kernels, we showed that the number of points in the training dataset is a very small fraction of the total number of points in the parameter space. This means that we have to generate and evaluate a fairly small number of code variants to characterize the energy usage behavior of kernels when subjected to various well-studied parametrized compiler-optimization strategies and can generate these models using hours of execution time rather than days, months or years (depending on the size of the optimization parameter space). Once the networks are trained, they can be used to predict power draw rate and energy usage of the CPUs and DIMMs for all code variants in the parameter space. Using these small training sets, the maximum absolute error rate of the models averages 5.5% for power draw, execution time, and energy usage over three important HPC kernels.

VIII. Acknowledgements

This work was supported partly by the DOE Office of Science through the SciDAC award titled SUPER (Institute for Sustained Performance, Energy and Resilience).

References

[1] CPU Freq. Scaling. https://wiki.archlinux.org/index.php/Cpufrequtils.

[2] I. Basheer and M. Hajmeer. Artificial Neural Networks: Fundamentals, Computing, Design, and Application. *Journal of Microbiological Methods*, 43(1):3 – 31, 2000.

[3] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield. PowerMon: Fine-grained and integrated power monitoring for commodity computer systems. In *IEEE SoutheastCon 2010*, pages 479 –484, 2010.

[4] W. Bircher and L. John. Complete system power estimation: A trickle-down approach based on performance events. In *ISPASS 2007*, pages 158 –168, april 2007.

[5] M. Caudill. Neural Networks Primer, Part I. *AI Expert*, 2:46–52, December 1987.

[6] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Green Computing Conference, 2010 International*, pages 135 –146, aug. 2010.

[7] M. W. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Transformation recipes for code generation and auto-tuning. In *LCPC'09*, Oct 2009.

[8] J. He, A. Snavely, R. Van der Wijngaart, and M. Frumkin. Automatic recognition of performance idioms in scientific applications. In *IPDPS'11*, pages 118 –127, may 2011.

[9] E. Ipek, B. R. D. Supinski, M. Schulz, and S. A. Mckee. An approach to performance prediction for parallel applications. In *Euro-Par, Springer LNCS*, 2005.

[10] A. Jain, J. Mao, and K. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31 –44, mar 1996.

[11] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[12] M. Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software*, 28(5):1–26, 11 2008.

[13] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. EuroPar'11, Bordeaux, France, 2011.

[14] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. ASPLOS-XII, pages 185–194, New York, 2006. ACM.

[15] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP '07*, pages 249–258. ACM, 2007.

[16] C. Lively, X. Wu, V. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. Cameron. Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems. *Computer Science - Research and Development*, pages 1–9. 10.1007/s00450-011-0190-0.

[17] C. Olschanowsky, L. Carrington, M. Tikir, M. Laurenzano, T. S. Rosing, and A. Snavely. Fine-grained energy consumption characterization and modeling. In *DOD High Performance Computing Modernization Program UGC*, 2010.

[18] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *6th International Conf. on High Performance and Embedded Arch. and Compilers*, HiPEAC '11, NY, USA, 2011.

[19] J. S. Seng and D. M. Tullsen. The Effect of Compiler Optimizations on Pentium 4 Power Consumption. In *7th Workshop on Interaction between Compilers and Computer Arch.*, INTERACT '03, Washington, DC, USA, 2003.

[20] R. Setiono. Feedforward neural network construction using cross validation. *Neural Comput.*, 13:2865–2877, Dec 2001.

[21] K. Singh, M. Bhadauria, and S. A. McKee. Prediction-based power estimation and scheduling for cmps. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 501–502, New York, NY, USA, 2009. ACM.

[22] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. A Scalable Auto-Tuning Framework for Compiler Optimization. In *IPDPS'09*, Rome, Italy, May 2009.

[23] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[24] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE conference on Supercomputing (1998)*, Wash., DC, USA, 1998.