Received: 18 December 2020 | Revised: 18 November 2021 | Accepted: 9 July 2022
DOI: 10.1002/cpe.7254

WILEY

SPECIAL ISSUE PAPER

Performance and power modeling and prediction using MuMMI and 10 machine learning methods

Xingfu Wu[1] | Valerie Taylor[1] | Zhiling Lan[2]

[1]Mathematics & Computer Science Division, Argonne National Laboratory, University of Chicago, Lemont, Illinois, USA

[2]Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA

Correspondence
Xingfu Wu, Mathematics & Computer Science Division, Argonne National Laboratory, University of Chicago, Lemont, IL 60439, USA.
Email: xingfu.wu@anl.gov

Funding information
U.S. National Science Foundation, Grant/Award Numbers: CCF-1801856, CCF-2119203; U.S. Department of Energy, Grant/Award Numbers: DE-AC02-06CH11357, RAPIDS2

Summary

Energy-efficient scientific applications require insight into how high performance computing system features impact the applications' power and performance. This insight can result from the development of performance and power models. In this article, we use the modeling and prediction tool MuMMI (Multiple Metrics Modeling Infrastructure) and 10 machine learning methods to model and predict performance and power consumption and compare their prediction error rates. We use an algorithm-based fault-tolerant linear algebra code and a multilevel checkpointing fault-tolerant heat distribution code to conduct our modeling and prediction study on the Cray XC40 Theta and IBM BG/Q Mira at Argonne National Laboratory and the Intel Haswell cluster Shepard at Sandia National Laboratories. Our experimental results show that the prediction error rates in performance and power using MuMMI are less than 10% for most cases. By utilizing the models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters for potential application optimizations, and we predict theoretical outcomes of the optimizations. Based on two collected datasets, we analyze and compare the prediction accuracy in performance and power consumption using MuMMI and 10 machine learning methods.

KEYWORDS
fault tolerant applications, machine learning, modeling, MuMMI, performance, power, prediction

1 | INTRODUCTION

Energy-efficient scientific applications require insight into how high performance computing (HPC) system features impact the applications' power and performance. This insight can result from the development of performance and power models. Dense matrix factorizations, such as LU, Cholesky, and QR, are widely used for scientific applications that require solving systems of linear equations, eigenvalues, and linear least squares problems.[1,2] Such real-world scientific applications often take days, weeks, and even months to be executed on tens and even hundreds of thousands of compute nodes, thereby relying on resilience software techniques and hardware fault tolerance to successfully finish the long executions because of software or hardware failures. While reducing execution time is still a major objective for HPC, future HPC systems and applications will have additional power and resilience requirements that represent a multidimensional tuning challenge. To embrace these key challenges, we must understand the complicated tradeoffs among runtime, power, and resilience. In this article, we explore performance and power modeling and prediction of an algorithm-based fault-tolerant linear algebra code (FTLA)[3] and a multilevel checkpointing fault-tolerant heat distribution code (HDC)[4] using MuMMI (Multiple Metrics Modeling Infrastructure)[5,6] and 10 machine learning (ML) methods.[7,8]

In this work, we use FTLA and HDC to conduct our experiments on the Cray XC40 Theta[9] and IBM BG/Q Mira[10] at Argonne National Laboratory, and on the Intel Haswell cluster Shepard[11] at Sandia National Laboratories. We analyze FTLA's performance and power characteristics and use MuMMI and 10 machine learning methods to model, predict and compare performance and power of FTLA and HDC. MuMMI[5] is a modeling

 © 2022 John Wiley & Sons, Ltd.

infrastructure that facilitates systematic measurement, modeling, and prediction of performance and power consumption, and performance-power tradeoffs and optimization for parallel systems. The 10 popular ML methods that cover tree/rule-based, nonlinear and linear ML methods from the R caret package[7,8] are random forests (RF),[12] Gaussian process (GP) with radial basis function,[13] extreme gradient boosting (xGB),[14] stochastic gradient boosting (SGB),[15] Cubist (Cub),[16] ridge regression (RR),[17] k-nearest neighbors (kNN),[7] support vector machines (SVM) with linear kernel,[13] conditional inference tree (CIT),[18] and multivariate adaptive regression spline (MAR).[19]

Our experimental results show that the prediction error rates in performance and power using MuMMI are less than 10% for most cases. By utilizing the models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters for potential application optimization efforts associated with the application characteristics and the target architectures, and we predict theoretical outcomes of the optimizations. Then, based on two datasets collected for the applications FTLA and HDC, we analyze and compare the prediction accuracy using MuMMI and 10 ML methods.

The remainder of this article is organized as follows. Section 2 discusses the applications FTLA and HDC. Section 3 briefly describes three architectures and their power profiling tools. Section 4 presents performance and power characteristics, modeling and prediction of FTLA using MuMMI. Section 5discusses the modeling and prediction of FTLA and HDC using 10 ML methods and compares them with MuMMI. Section 6 summarizes this work.

2 | FAULT-TOLERANT APPLICATIONS: FTLA AND HDC

Many resilience methods have been developed for preventing or mitigating failure impact. Existing resilience strategies can be broadly classified into four approaches: checkpoint based, redundancy based, proactive methods, and algorithm based. Checkpoint/restart is a long-standing fault tolerance technique to alleviate the impact of system failures, in which the applications save their state periodically, then restart from a previous checkpoint. Multilevel checkpointing is the state-of-the-art design of checkpointing, focusing on reducing checkpoint overhead to improve checkpoint efficiency. Such checkpointing libraries include fault tolerance interface (FTI),[4,20] scalable checkpoint/restart (SCR),[21,22] VeloC,[23] and diskless checkpointing.[24] Redundancy approaches improve resilience by replicating data or computation.[25-27] Proactive methods take preventive actions before failures, such as software rejuvenation and process or object migration.[28] Algorithm-based fault tolerance (ABFT) methods maintain consistency of the recovery data by applying appropriate mathematical operations on both the original and recovery data, and they adapt the algorithm so that the application dataset can be recovered at any moment.[2,29-32] ABFT was applied to High Performance Linpack (HPL),[33] to Cholesky factorization,[34] and to LU and QR factorizations.[1,2,35] The FTLA[2,3] in particular was developed as an extension to ScaLAPACK[36] that tolerates and recovers from fail-stop failures, which is defined as a process that completely stops responding, triggering the loss of a critical part of the global application state, and halts the application execution. While fault tolerance methods and power management techniques continue to evolve, tradeoffs among execution time, power efficiency, and resilience strategies are still not well understood. To understand the tradeoffs among runtime, power, and resilience, in this article we explore performance and power modeling and prediction of two fault-tolerant applications under different resilience strategies.

Fault tolerance studies focus mainly on the tradeoffs between execution time, fault tolerance overhead, and resiliency, whereas most power management studies focus on the tradeoffs between execution time and power. Understanding the tradeoffs among all these factors is crucial because future HPC systems will be built under both reliability and power constraints. Our previous work[37] presented an empirical study evaluating the runtime and power requirements of multilevel checkpointing MPI applications using FTI on four different parallel architectures and collected a large amount of performance data. Recent research has focused on a theoretical analysis of energy and runtime for fault tolerance protocols[38-43] and benchmarking machine learning methods for performance modeling of scientific applications.[44] In this work, we use the algorithm-based fault-tolerant FTLA and the multilevel checkpointing fault-tolerant HDC to conduct our modeling and prediction study in power and performance.

Matrix QR factorization decomposes a matrix A into a product A = QR, where Q is an orthogonal matrix and R is an upper triangular matrix. The FTLA code ftla-rSC13[3] consists of two main components: one QR operation followed by a resilient QR (RQR) operation, where the RQR performs one QR, checkpointing, and repairing a failure until completing without a failure as shown in Figure 1. The structure of the block QR and LU is identical. We focus on the QR in this article. The main loop is associated with the matrix sizes. For each matrix size, it performs one QR followed by one small loop. The small loop size is the number of error injections. For each error injection, it performs one RQR.

We remove all segments for error injections from ftla-rSC13 to create another code called *ftla*. The main loop is associated with the matrix sizes. For each matrix size, it performs one QR followed by one RQR. The RQR performs one QR and checkpointing. Then we remove the checkpointing segments from *ftla* to get a code called *la*, which is similar to ScaLAPACK QR. In this article, we use the three codes ftla-rSC13, *ftla*, and *la* to conduct our experiments. They are configured as strong scaling problems.

The other application used in this article is an FTI version of MPI heat distribution benchmark code (HDC),[4] which computes the heat distribution over time based on a set of initial heat sources in the FTI package. FTI[20] leverages local storage, along with data replication and erasure codes, to provide several levels of reliability and performance. It provides four-level checkpointing: local write (L1), Partner copy (L2), Reed-Solomon coding (L3), and PFS write (L4). The four checkpointing levels correspond to coping with the four types of failures: no hardware failure (software failure),

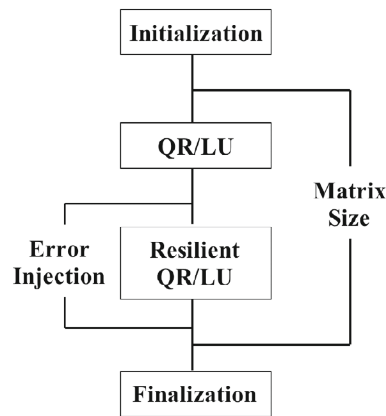

FIGURE 1 Control flow of the ftla-rSC13 code

single-node failure, multiple-node failure, and all other failures the lower levels cannot take care of, respectively. The checkpointing file size is 2 MB per MPI process. HDC is compute-intensive and weak scaling.

3 | SYSTEM ARCHITECTURES AND ENVIRONMENTS

We conduct our experiments on three HPC systems with different architectures: the Cray XC40 Theta[9] and IBM BlueGene/Q (BG/Q) Mira[10] at Argonne National Laboratory and the Intel Haswell cluster Shepard[11] at Sandia National Laboratories. Details about each system are given in Table 1. Each Cray XC40 node has 64 compute cores: one Intel Phi Knights Landing (KNL) 7230 with the thermal design power (TDP) of 215 W, 32 MB of L2 cache, 16 GB of high-bandwidth in-package memory (MCDRAM) with the bandwidth of 480 GB/s, 192 GB of DDR4 RAM with the bandwidth of 90 GB/s, and a 128 GB SSD. MCDRAM can be configured as a shared last level cache (cache mode) shown in Figure 2 or as a distinct NUMA node memory (flat mode) in Figure 3 or somewhere in between. For the flat mode, the default memory allocation preference is DDR4 first, then MCDRAM. Theta uses the Cray Aries dragonfly network with user access to a Lustre parallel file system with 10 PB of capacity and 210 GB/s bandwidth.

Each BG/Q node of Mira has 16 compute cores—one BG/Q PowerPC A2 1.6 GHz chip with the TDP of 55 W[45] and shared L2 cache of 32 MB and 16 GB of memory. The IBM BG/Q architecture features a quad floating-point unit that can be used to execute four-wide SIMD instructions or two-wide complex arithmetic SIMD instructions. Mira uses a 5D torus network with user access to a GPFS file system.[10] Each Haswell node of

TABLE 1 Specifications of three different architectures

System name	ANL Cray XC40 Theta	ANL IBM BG/Q Mira	SNL Linux Cluster Shepard
Architecture	Intel KNL	IBM BG/Q	Intel Haswell
Number of nodes	4392	49,152	36
CPU cores per node	64	16	32
Sockets per node	1	1	2
CPU type and speed	Xeon Phi KNL 7230 1.30 GHz	PowerPC A2 1.6 GHz	Xeon(R) E5-2698 V3 2.3 GHz
L1 cache per core	D: 32 kB/I: 32 kB	D: 16 kB/I: 16 kB	D: 32 kB/I: 32 kB
L2 cache per socket	32 MB (shared)	32 MB (shared)	256 kB (per core)
L3 cache per socket	None	None	40 MB (shared)
Memory per node	16 GB/192 GB	16 GB	128 GB
Network	Cray Aries Dragonfly	5D Torus	Mellanox FDR InfiniBand
Power tools	CapMC/PoLiMEr	EMON/MonEQ	PowerInsight
TDP per socket	215 W	55 W	135 W
File system	Lustre PFS	GPFS	Regular NFS

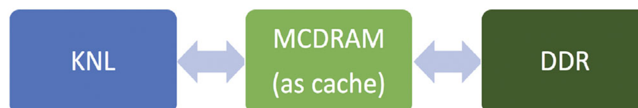

FIGURE 2 Cache memory mode on Cray XC40 Theta

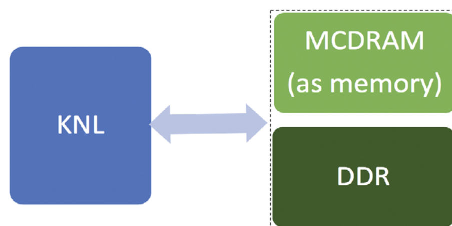

FIGURE 3 Flat memory mode on Cray XC40 Theta

Shepard has 32 compute cores—two Xeon E5-2698 V3 2.3 GHz chips with the TDP of 135 W per chip and shared L3 cache of 40 MB and 128 GB of memory. Shepard uses a Mellanox fourteen data rate InfiniBand network with a regular NFS file system.[11]

Several vendor-specific power management tools exist, such as Cray's CapMC and out-of-band and in-band power monitoring capabilities,[46] IBM EMON API on BG/Q,[45] Intel RAPL,[47] and NVIDIA's power management library.[48] In this work, we use simplified PoLiMEr[49] to measure power consumption for the node, CPU, and memory at the node level on Theta. PoLiMEr uses Cray's CapMC to obtain power and energy measurements of the node, CPU, and memory on Theta. The power sampling rate is approximately 2 samples per second (default). We also use EMON API-based MonEQ[50] to collect power profiling data on Mira. EMON API[45] provides 7 power domains to measure the power consumption for the node, CPU, memory, and network at the node-card level. Each node-card consists of 32 nodes. To obtain the power consumption at the node level, we calculate the average power by dividing by 32. So we conduct our experiments on multiple node-cards to obtain the power-profiling data. The power sampling rate is approximately 2 samples per second (default). We use PowerInsight[51] to measure the power consumption for the node, CPU, memory, and hard disk at the node level on Shepard. PowerInsight provides the measurement for 10 power rails for CPU, memory, disk, and motherboard on the Intel Haswell system Shepard. The power sampling rate used is 1 sample per second (default).

4 | MODELING AND PREDICTION USING MUMMI

In this section, we use the three codes ftla-rSC13, ftla, and la with matrix sizes from 6000 to 20,000 with a stride of 2000 and a block size of 100 to conduct our experiments with a maximum of error injections of 5 on Cray XC40 Theta, IBM BG/Q Mira, and Intel Haswell Shepard. We analyze their performance and power characteristics, and use performance counter-based modeling tool MuMMI[5,6] to model performance and power and to predict theoretical outcomes for the potential optimizations.

We use MuMMI with support of PoLiMEr, MonEQ and PowerInsight to instrument these codes to collect performance data, power data, and performance counters on Cray XC40 Theta with the Cray compiler (Intel compiler 18.0), IBM BG/Q Mira with IBM XL compiler 12.1, and Intel Haswell Shepard with Intel compiler 16.1. We use the same default compiler options from the FTLA code ftla-rSC13 to compile the codes. For a given application run, we execute the application 14 times on each system to ensure the consistency of the results while collecting different sets of available performance counters for performance and power modeling. We found that the variation of the application runtime is very small (less than 1%), so we use the performance metrics corresponding to the smallest runtime for our work. In the rest of this article, we use the formula $(\text{prediction} - \text{baseline})/\text{baseline} * 100\%$ to calculate the prediction error rate.

4.1 | Cray XC40 Theta

Each Theta node[9] has one Intel KNL with MCDRAM in addition to the traditional DDR4 RAM. MCDRAM can be configured as a shared last level cache L3 (cache mode) or as a distinct NUMA node memory (flat mode). With the different memory modes by which the system can be booted, it becomes a challenge from a software perspective to understand the best mode for an application. We use the codes *la* and *ftla* to investigate the performance and power impacts under the cache or flat mode use of MCDRAM.

Figure 4 presents the performance and power comparison of *la* using the two memory modes on Theta, where the terms with **-flat** stand for using the flat mode and the terms without **-flat** stand for using the cache mode (default). The advantage of using MCDRAM as cache is that an application may run entirely in MCDRAM so that the application performance may be improved significantly. We find that the application runtime using the cache mode is almost half of the runtime using the flat mode on 64 cores because the application with the problem sizes fits into the MCDRAM and MCDRAM has the bandwidth of 480 GB/s which is much higher than the DDR4 bandwidth of 90 GB/s. From this figure, we also observe that the difference for both node power consumptions is small. The CPU power for using the cache mode is higher, but the memory power for using the cache mode is much lower. Overall, using the cache mode results in lower energy consumption for these cases. We find the same trend for the code ftla shown in Figure 5. Therefore, in the remainder of this section, we use the cache mode to conduct our experiments on Theta.

Figure 6 presents a performance comparison of the three codes on Theta, where **ftla-1** stands for the code ftla-rSC13 with one error injection. We observe a proportional increase in application runtime with increasing numbers of error injections on up to 1024 cores because of the proportional increase in the number of error injections.

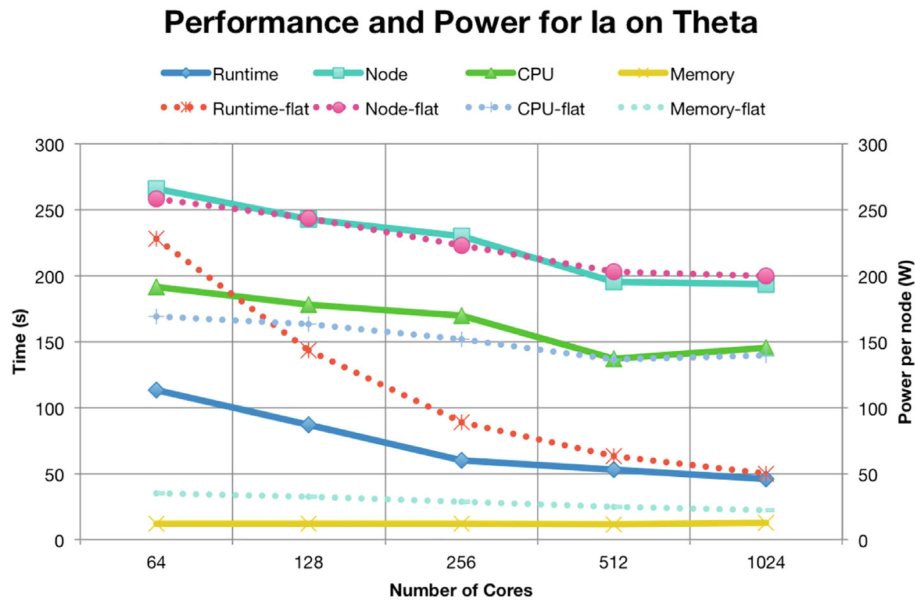

FIGURE 4 Comparison for la on Theta

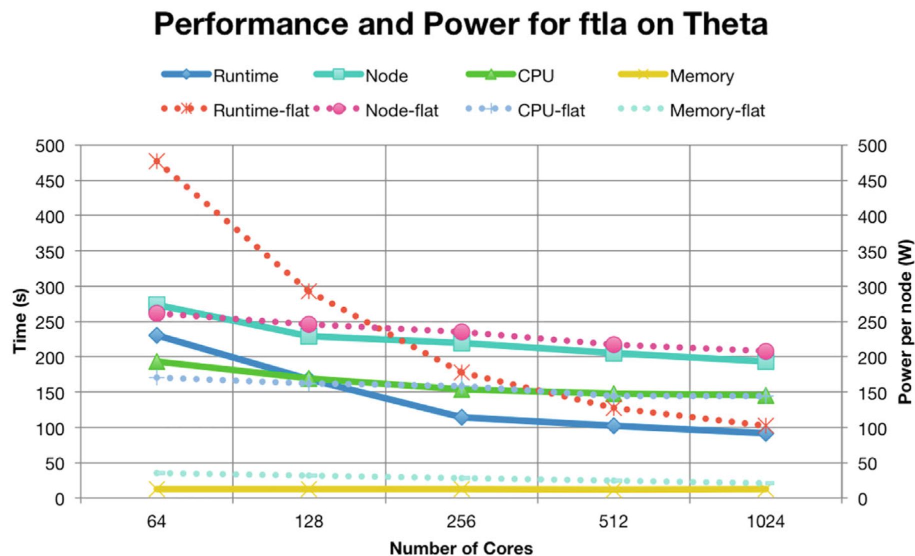

FIGURE 5 Comparison for ftla on Theta

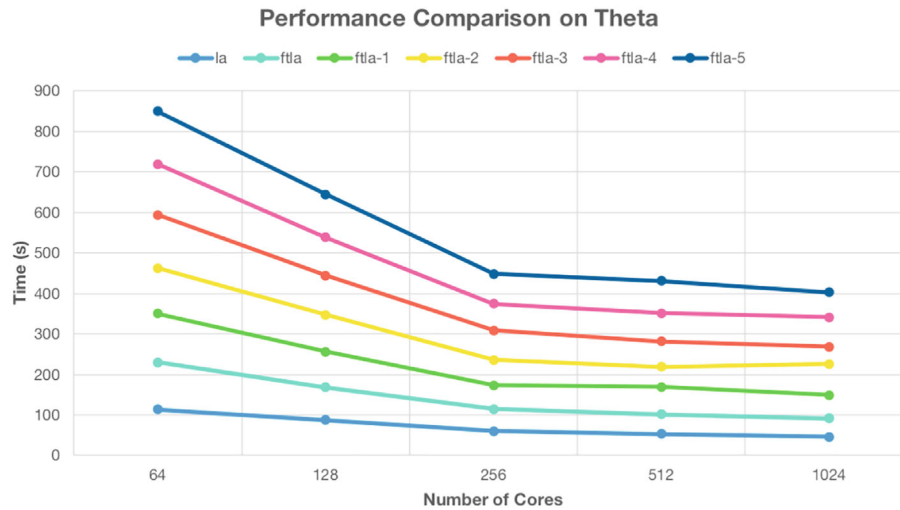

FIGURE 6 Performance comparison on Theta

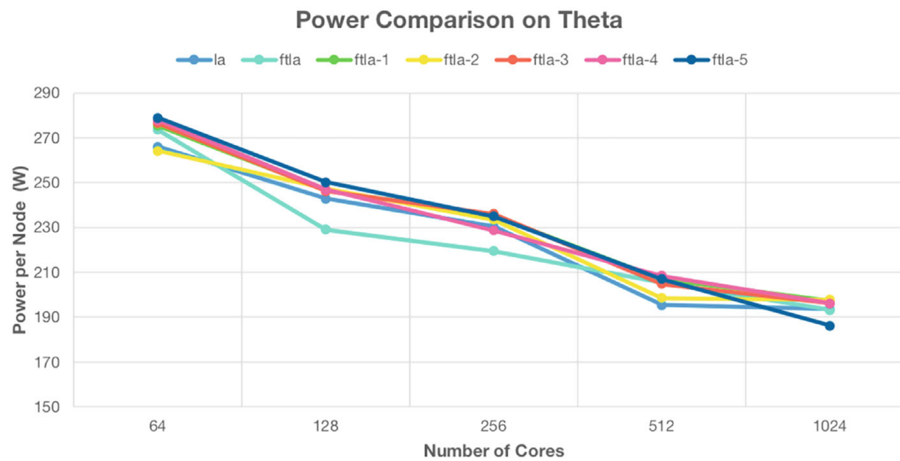

FIGURE 7 Power comparison on Theta

Figure 7 shows the average node power consumption on Theta. The node power consumption decreases with increasing numbers of cores because of the strong scaling and dynamic power management support. Further, we compare power over time for the FTLA with one error injection and five error injections on 1024 cores in Figure 8. We observe that the CPU power mainly affects the node power changes for both cases. Because of the dynamic power management on Theta, during each matrix loop the power adjusts dynamically, the power increases with the increase in the matrix size from 6000 to 20,000. The runtime mainly results in the large energy increase.

To develop accurate models of runtime and power consumptions for the code ftla-rSC13, we use the power and performance modeling tool MuMMI from our previous work[5,6] for the data collection. We collect 26 available performance counters on Theta with different system configurations (numbers of cores: 64, 128, 256, 512, and 1024) and the number of error injections (1, 2, and 3) as a training set. We then use a Spearman correlation and principal component analysis (PCA) to identify the major performance counters $(r_1, r_2, \dots, r_n (n \ll 26))$, which are highly correlated with the metric: runtime, node power, CPU power, or memory power. Then we use a nonnegative multivariate regression analysis to generate four models based on the small set of major counters and CPU frequency (f), as shown in Figure 9, where a numeric value is the coefficient for the counter in the corresponding model; power_sys stands for node power model; power_cpu stands for CPU power model; and power_mem stands for memory power model.

For the model of runtime T, we develop the following equation:

$$T = \beta_0 + \beta_1 * r_1 + \beta_2 * r_2 + \cdots + \beta_n * r_n + \beta * \frac{1}{f}. \tag{1}$$

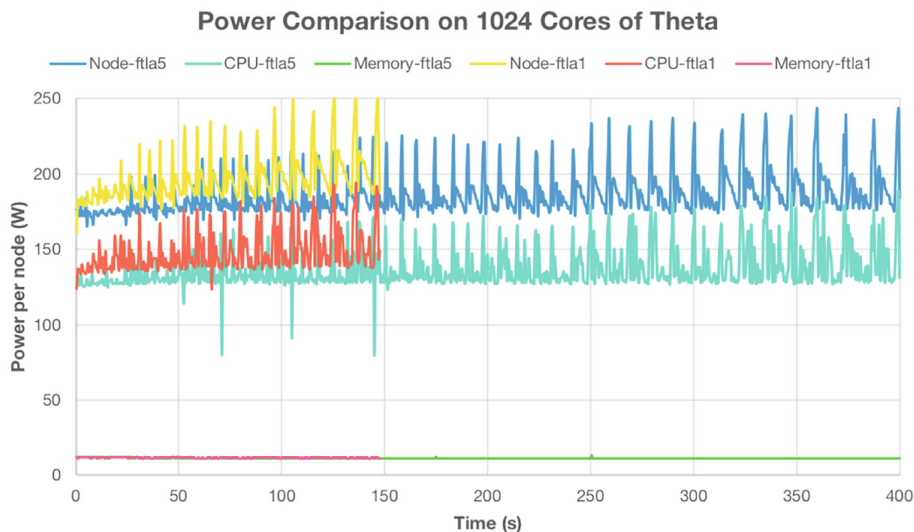

FIGURE 8 Power over time on 1024 cores of Theta

FTLA-rSC13 1.0

Model Coefficients

Show 25 entries

Counter	runtime	power_sys	power_cpu	power_mem
Frequency	0.0	0.0	0.0	0.0
PAPI_BR_CN	1.1654554e-08			
PAPI_L1_LDM	4.2520972e-09	232.78364		
PAPI_L1_STM	1.3389905e-09	559.99345	409.0503	4.2548225
PAPI_L1_TCM		7355.5034	2894.4924	
PAPI_L2_DCM	3.1814643e-08	9913.4035	1292.4448	
PAPI_L2_ICA	1.7074352e-10	331.02443	321.91886	
PAPI_L2_LDM		4.1376356		
PAPI_L2_STM	9.2365091e-10			33.586047
PAPI_L2_TCW	1.6738179e-09	1151.7689	838.30442	6.8989722
PAPI_RES_STL	3.0464497e-09			
PAPI_TLB_DM	1.6620536e-08		4524.0593	
PAPI_TOT_INS				6.5347421

FIGURE 9 Four models on Theta

In this equation, T is the component predictor used to represent the value for runtime. The intercept is β_0; each $\beta_i(i = 1, 2, \dots, n)$ represents the regression coefficient for performance counter r_i, and β represents the coefficient for the CPU frequency. Equation (1) can be used to predict the runtime for the larger numbers of error injections (4 or 5 error injections).

Similarly, we can model CPU power consumption P using the following equation:

$$P = \alpha_0 + \alpha_1 * r_1 + \alpha_2 * r_2 + \cdots + \alpha_n * r_n + \alpha * f^3. \tag{2}$$

In this equation, P is the component predictor used to represent the value for the CPU power. The intercept is α_0; and each $\alpha_i(i = 1, 2, \dots, n)$ represents the regression coefficient for performance counter r_i, and α represents the coefficient for the CPU frequency. Equation (2) can be used to predict the CPU power on larger numbers of error injections. Similarly, a multivariate linear regression model is constructed for the other metrics (node power, memory power).

Table 2 shows the prediction error rates for the runtime and power of the application with 4 and 5 error injections using Equations (1) and (2). Overall, the prediction error rates (absolute values) are less than 3.3% in runtime. This indicates that our counter-based performance models are very accurate. The prediction error rates are less than 8.9% in node power, less than 6.3% in CPU power, and less than 7.9% in memory power. These

TABLE 2 Prediction error rates on Theta

	ftla-4				ftla-5			
# cores	Runtime	Node power	CPU power	Memory power	Runtime	Node power	CPU power	Memory power
64	0.19%	−3.66%	−2.51%	−5.97%	−0.54%	−3.88%	−3.86%	−0.82%
128	−2.1%	−1.96%	0.16%	−6.85%	−3.21%	−3.96%	−2.71%	−0.85%
256	0.75%	2.02%	3.13%	−7.89%	−1.32%	−2.62%	−3.07%	−1.30%
512	−0.49%	−1.86%	−1.50%	−6.24%	0.74%	−3.63%	−5.80%	−7.03%
1024	−1.01%	2.85%	2.52%	−3.86%	−0.65%	8.89%	6.29%	7.74%

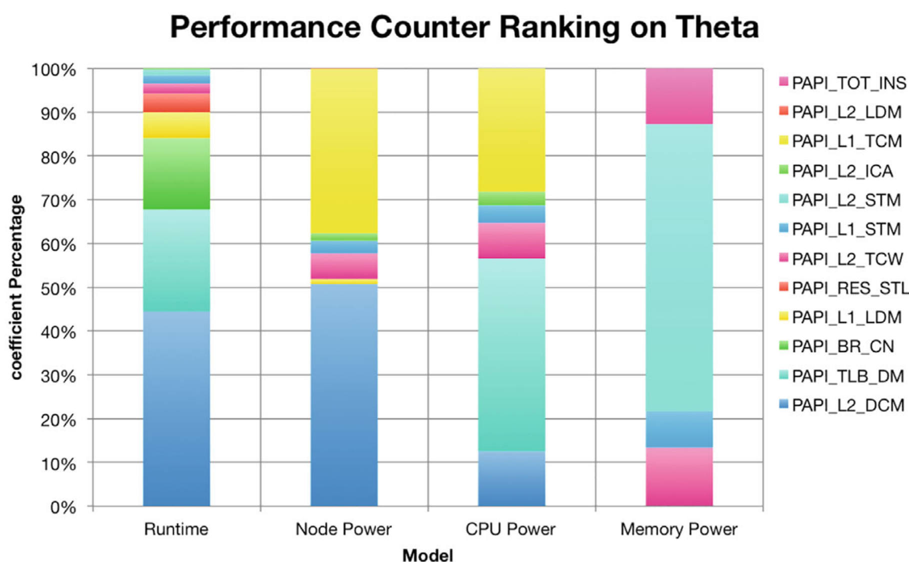

FIGURE 10 Counter ranking on Theta

performance and power models are generated from different system configurations and problem sizes, thus providing a broader understanding of the application's usage of the underlying architectures. This in turn results in more knowledge about the application's energy consumption on the given architecture.

Based on the models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters for the application. Figure 10 shows the performance counter rankings of the four models using 12 different counters. We found that the L2_DCM (Level 2 data cache misses) and TLB_DM (Data translation lookaside buffer misses) contribute most in the runtime model; L2_DCM and L1_TCM (Level 1 cache misses) contribute most in the node power; TLB_DM and L1_TCM contributes most in CPU power models; and L2_STM (Level 2 store misses) contributes most in the memory power model. TLB_DM is correlated with L1_TCM. Therefore, the optimization efforts for the code should focus on the components associated with L2_DCM, TLB_DM, and L2_STM on Theta. For instance, as shown in Figure 11, we use our what-if prediction system based on the four model equations to predict the theoretical outcomes of the possible application optimization by reducing L2_DCM by 30%, the other counters may be changed based on the pairwise correlation with this counter. The theoretical improvement percentage is 2.99% in runtime, 10.08% in node power, 7.44% in CPU power, and 7.10% in memory power. Theta supports several huge page sizes ranging from 2 MB to 2 GB with the default page size of 4 kB. In order to reduce the TLB miss (TLB_DM), the main kernel address space is mapped with huge pages—a single 2 MB huge page requires only a single TLB entry, while the same memory, in 4 kB pages, would need 512 TLB entries. Using the huge pages will result in the application performance improvement.

4.2 | IBM Blue Gene/Q Mira

On Mira, the EMON API provides the power measurement at a node-card level, so we conduct our experiments on 512 cores (one node card), 1024, 2048, 4096, 8192, and 16,384 cores. To develop accurate models for runtime and power consumptions on Mira, we collect 40 available performance

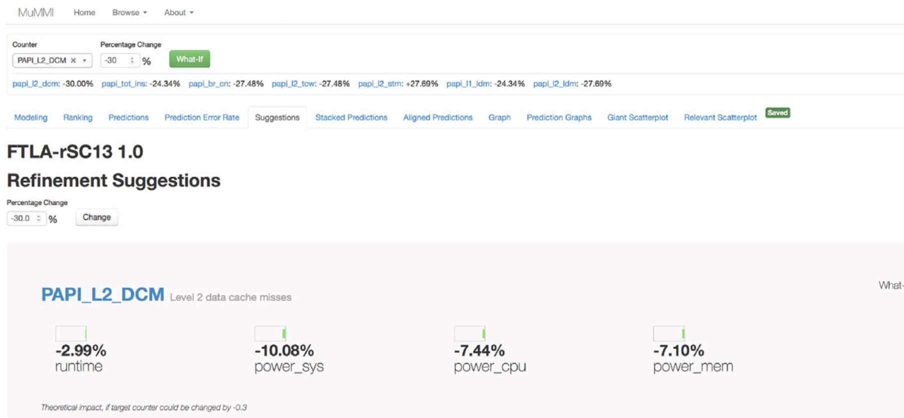

FIGURE 11 Theoretical prediction on Theta

ftla-rSC13 1.0

Model Coefficients

Show 10 entries

Counter	runtime	power_sys	power_cpu	power_mem
Frequency	5.736374e-10	0.0	0.0	0.0
PAPI_BR_MSP	4.6019294e-09			
PAPI_BR_NTK	8.8844693e-10	207.47193	146.12911	123.24915
PAPI_BR_TKN	2.6036362e-10			
PAPI_FML_INS				2217.3777
PAPI_FP_INS	7.3597041e-10			48.050968
PAPI_L1_STM	5.0845408e-11			
PAPI_RES_STL	2.7328663e-10	84.64592	53.433128	
PAPI_SR_INS	1.9297883e-09	272.87974	142.9152	137.33241
PAPI_VEC_INS		23326.1	33401.773	

FIGURE 12 Four models on Mira

counters with different system configurations (numbers of cores: 512, 1024, 2048, 4096, 8192, and 16,384) and the number of error injections (1, 2, and 3) as a training set. We then use MuMMI to generate four models based on the small set of major counters and CPU frequency (f), as shown in Figure 12, where a numeric value is the coefficient for the counter in the corresponding model.

Table 3 shows the prediction error rates for the runtime and power of the application with 4 and 5 error injections using Equations (1) and (2). Overall, the prediction error rates in runtime are less than 0.1%. This indicates that our counter-based performance models are accurate. The prediction error rates are less than 5% in node power and less than 8.7% in CPU power; and the error rates are less than 10% in memory power for most cases except 15.30% for ftla-4 on 2048 cores.

Based on the models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters. Figure 13 shows the performance counter rankings of the four models using 9 different counters. We find that the BR_MSP (conditional branch instructions mispredicted) contributes most in the runtime model and is correlated with the counters SR_INS (Store instructions), BR_TKN (Conditional branch instructions taken), FP_INS (floating-point instructions), and RES_STL (Cycles stalled on any resource); VEC_INS (Vector/SIMD instructions (could include integer)) contributes most in the node power and CPU power models; and FML_INS (floating-point multiply instructions) contributes most in the memory power model. VEC_INS and FML_INS are not correlated with any other counters. Therefore, the optimization efforts for the code should focus on the components associated with BR_MSP, VEC_INS, and FML_INS on Mira. For instance, Mira features a quad floating-point unit that can be used to execute four-wide SIMD instructions or two-wide complex arithmetic SIMD instructions. In order to take advantage of vector instructions supported by BG/Q processors, the compiler options -qarch-qp and -qsimd=auto may be applied to compile the code to improve the energy efficiency. For instance, as shown in Figure 14, we use our what-if prediction system based on the four model equations to predict the theoretical outcomes of the possible optimization. By accelerating VEC_INS by 30%, the theoretical improvement percentage is 0.15% in runtime, 1.29% in node power, 2.49% in CPU power, and 1.79% in memory power.

TABLE 3 Prediction error rates on Mira

	ftla-4				ftla-5			
# cores	Runtime	Node power	CPU power	Memory power	Runtime	Node power	CPU power	Memory power
512	0.009%	4.11%	3.19%	9.82%	0.009%	−3.40%	−3.65%	−9.00%
1024	0.018%	3.65%	8.64%	−6.18%	0.020%	−3.43%	−3.37%	−5.34%
2048	0.046%	3.23%	1.92%	15.30%	0.049%	0.05%	3.77%	6.95%
4096	0.035%	1.40%	3.98%	−7.12%	0.041%	−4.01%	−4.77%	−3.49%
8192	−0.021%	−0.50%	−0.12%	−5.71%	−0.043%	−1.89%	−2.44%	−4.13%
16,384	0.076%	2.45%	5.52%	−9.62%	−0.086%	0.97%	3.03%	−6.36%

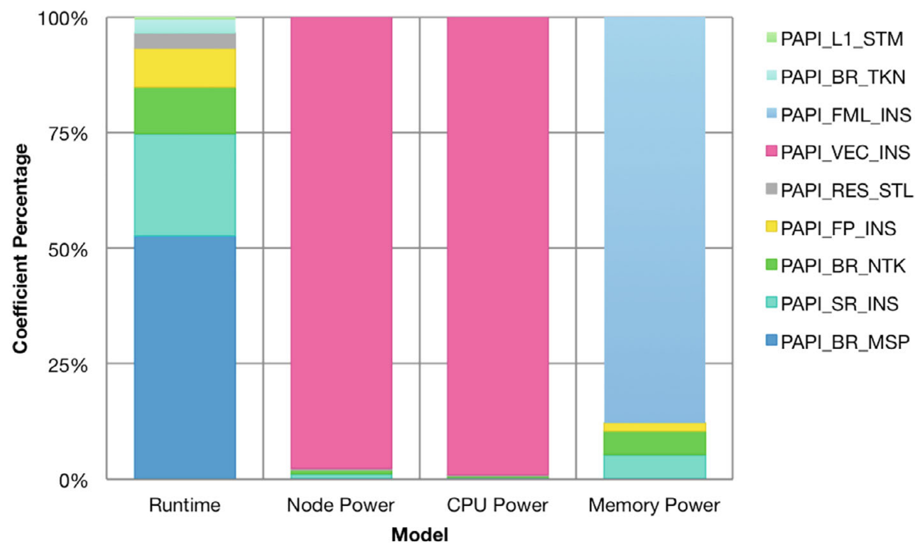

FIGURE 13 Counter ranking on Mira

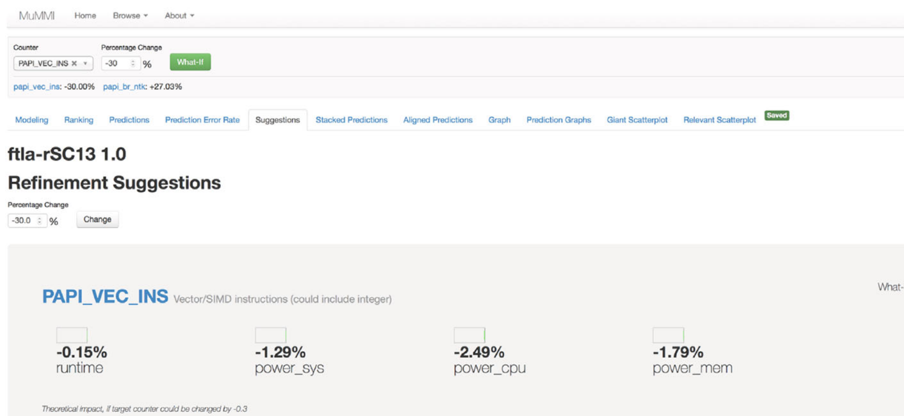

FIGURE 14 Theoretical prediction on Mira

4.3 | Intel Haswell Shepard

To develop accurate models for runtime and power consumptions on Shepard, we collect 32 available performance counters for different system configurations (numbers of cores: 32, 64, 128, 256, 512, and 1024) and the number of error injections (1, 2, and 3) as the training dataset, and we use MuMMI to generate the four models based on the small set of major counters and CPU frequency (*f*), as shown in Figure 15.

Table 4 shows the prediction error rates for the runtime and power of the application with 4 and 5 error injections using Equations 1 and 2. Overall, the prediction error rates (absolute values) are less than 0.25% for runtime. These results indicate that our counter-based performance models are accurate. The prediction error rates are less than 7.3% in node power, and less than 6.6% in CPU power; and the prediction error rates in memory power are less than 7.46% for most cases except 16.57% for ftla-4 and 12.88% for ftla-5 on 256 cores.

Based on the models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters. Figure 16 shows the performance counter rankings of the four models using 13 different counters. We found that the L2_ICM (Level 2 instruction cache misses) and L1_DCM (Level 1 data cache misses) contribute most in the runtime model; L2_TCM (Level 2 cache misses) and L1_ICM (Level 1 instruction cache misses) contribute most in the node power; L2_TCM and L1_TCM contributes most in CPU power models; and L2_TCM and L1_ICM contribute most in memory power model. L2_ICM is correlated with L1_TCM, and L2_TCM is correlated with L1_ICM. Therefore, the optimization efforts for the code should focus on the components associated with L2 and L1 caches on Shepard. For instance, as shown in Figure 17, we use our what-if prediction system based on the four model equations to predict the theoretical outcomes of the possible application optimization by reducing L2_TCM by 30%, the other counters may be changed based on the correlation with this counter. The theoretical improvement percentage is −0.02% in runtime, 7.02% in node power, 6.79% in CPU power, and 14% in memory power. For instance, loop optimization methods such as loop blocking and unrolling may help improve the cache locality.

Ftla-rSC13 1.0

Model Coefficients

Show 25 entries

Counter	runtime	power_sys	power_cpu	power_mem
Frequency	1.0114651e-09	84.305887	74.208263	0.0
PAPI_CA_ITV			35.71907	
PAPI_L1_DCM	8.1920729e-09			
PAPI_L1_ICM		1380.008		2710.9315
PAPI_L1_LDM			57.549126	
PAPI_L1_STM				28.980694
PAPI_L1_TCM			163.39522	
PAPI_L2_DCA		839.01997	323.86345	179.01216
PAPI_L2_ICM	2.3250337e-07			
PAPI_L2_TCM		12650.877	9007.5718	3748.7782
PAPI_L2_TCW				803.13498
PAPI_RES_STL		58.4535		190.32842
PAPI_SR_INS	3.5461051e-09			
PAPI_TOT_INS		6.9190777		

FIGURE 15 Four models on Shepard

TABLE 4 Prediction error rates on Shepard

	ftla-4				ftla-5			
# cores	Runtime	Node power	CPU power	Memory power	Runtime	Node power	CPU power	Memory power
32	−0.05%	7.25%	6.57%	7.45%	−0.04%	1.64%	2.24%	4.08%
64	0.06%	−3.07%	−2.91%	−6.05%	0.09%	−4.88%	−4.99%	−5.72%
128	0.05%	−0.48%	−0.82%	0.10%	−0.10%	−1.71%	−0.44%	0.92%
256	0.02%	2.21%	0.58%	16.57%	0.12%	3.12%	2.21%	12.88%
512	−0.04%	−0.27%	−0.74%	6.42%	0.02%	−0.71%	−1.27%	−1.87%
1024	0.19%	−0.99%	−2.18%	5.30%	0.24%	−1.21%	−1.58%	−1.58%

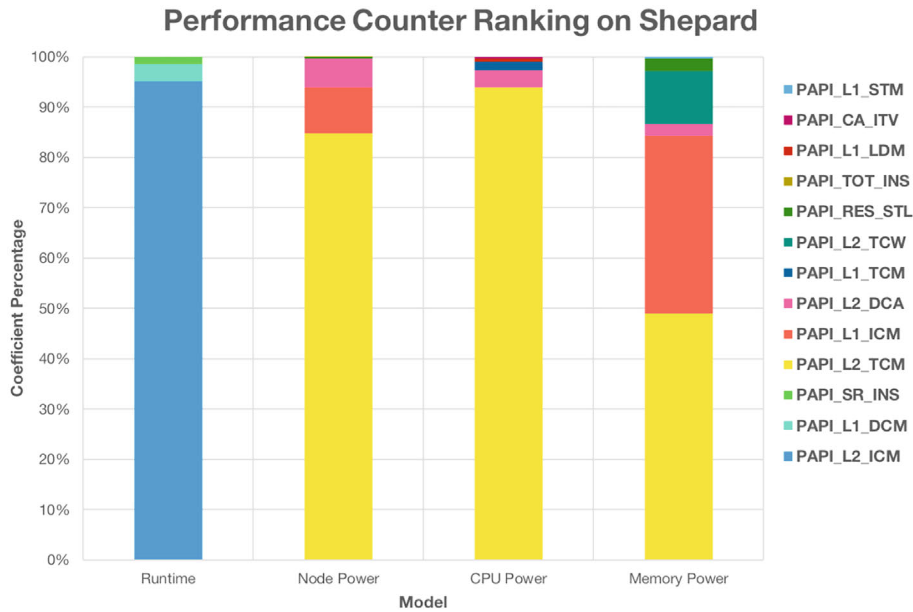

FIGURE 16 Counter ranking on Shepard

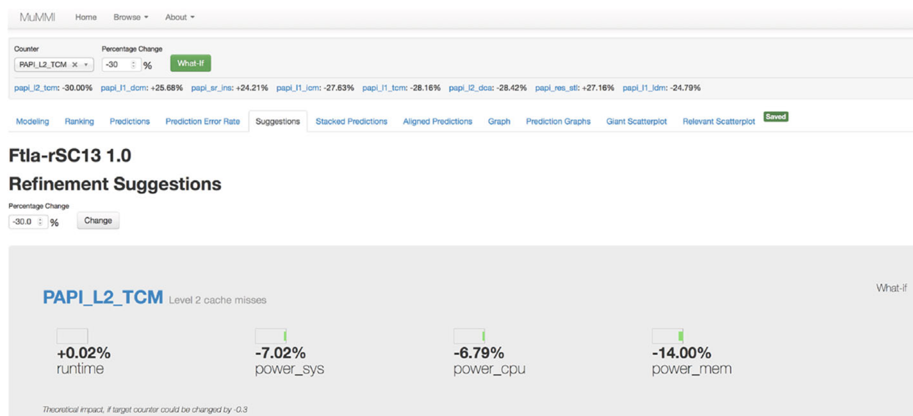

FIGURE 17 Theoretical prediction on Shepard

4.4 | Summary

Table 5 summarizes the top performance counters for each model using MuMMI on the three systems. For the same FTLA application, we observe that different architectures really impacts the application performance and power consumption distinctly. Both Theta and Shepard have Intel architectures such as Intel KNL for Theta and Intel Haswell for Shepard. The top performance counters in performance and power models mainly are related to L2 cache activities except the top performance counter TLB_DM in CPU power model for Theta. Mira has the IBM BlueGene/Q architecture which features a quad floating-point unit that can be used to execute four-wide SIMD instructions or two-wide complex arithmetic SIMD

TABLE 5 Top performance counters for each model using MuMMI on three systems

System	Runtime model	Node power model	CPU power model	Memory power model
Theta	L2_DCM	L2_DCM	TLB_DM	L2_STM
Mira	BR_MSP	VEC_INS	VEC_INS	FML_INS
Shepard	L2_ICM	L2_TCM	L2_TCM	L2_TCM

instructions. Its top performance counters are BR_MSP, VEC_INS, and FML_INS that are mainly related to the quad floating-point unit. Therefore, for the potential optimizations of the same application on different architectures, it needs to seriously consider how well the application characteristics fit into the underlying architectures for efficient execution based on the insights from the most important performance counters.

5 | MODELING AND PREDICTION USING 10 ML METHODS

In this section, we use 10 popular ML methods from the R caret package[7,8] to model and predict performance and power of FTLA and HDC. Our methodology is as follows. First, we use the datasets for FTLA or HDC as input to split the data into the training and test datasets based on the 80/20% rule, and find out what the training and test datasets are by setting the seed 3456 of R's random number generator set.seed() so that creating the random objects can be reproduced. Second, we apply the same training and test datasets to the 10 ML methods. Third, we use the same training and test datasets to build the performance and power models using MuMMI online. Finally, we compare the prediction error rates for these methods using violin plot from R violin package[52] which is a combination of a box plot and a kernel density plot to visualize the distribution of the prediction error rates.

The 10 popular ML methods from R caret package cover tree/rule-based, nonlinear and linear ML methods, and they are RF,[12] GP with radial basis function,[13] xGB,[14] SGB,[15] Cub,[16] RR,[17] kNN,[7] SVM with linear kernel,[13] CIT,[18] and MAR.[19]

RF[12] for regression or classification based on a forest of trees using random inputs, which was constructed in Reference 53 as a tree-based model. An RF model achieves the variance reduction by selecting strong, complex learners that exhibit low bias. This ensemble of many independent, strong learners yields an improvement in error rates.

GP[54] with radial basis function,[13] which is based on the prior assumption that adjacent observations should convey information about each other. It is assumed that the observed variables are normal, and that the coupling between them takes place by means of the covariance matrix of a normal distribution. Using the kernel matrix as the covariance matrix is a convenient way of extending Bayesian modeling of linear estimators to nonlinear situations.

xGB,[14] which is an efficient implementation of the gradient boosting framework in Reference 55. It provides a sparsity aware algorithm for handling sparse data and a theoretically justified weighted quantile sketch for approximate learning.

SGB,[15] which is an implementation of extensions to AdaBoost algorithm[56] and gradient boosting machine.[57] It includes regression methods for least squares, absolute loss, t-distribution loss, quantile regression, logistic, multinomial logistic, Poisson, Cox proportional hazards partial likelihood, AdaBoost exponential loss, Huberized hinge loss, and Learning to Rank measures.

Cub,[16] which is a regression modeling using rules with added instance-based corrections that combines the ideas in References 58 and 59. A Cub regression model is to fit for each rule based on the data subset defined by the rules. The set of rules are pruned or possibly combined, and the candidate variables for the linear regression models are the predictors that were used in the parts of the rule that were pruned away.

RR,[17,60] which adds a penalty on the sum of the squared regression parameters to create biased regression models. It reduces the impact of collinearity on model parameters. Combatting collinearity by using biased models may result in regression models where the overall mean squared error is competitive.

kNN,[7] which imply predicts a new sample using the k-closest samples from the training set. To predict a new sample for regression, It identifies that sample's kNN in the predictor space. The predicted response for the new sample is then the mean of the k neighbors' responses.

SVM with linear kernel,[13] which is the kernlab's implementation of SVMs.[61] It chooses a linear function in the feature space by optimizing some criterion over the sample.

CIT,[18] which embeds tree-structured regression models into a well-defined theory of conditional inference procedures. This nonparametric class of regression trees is applicable to all kinds of regression problems, including nominal, ordinal, numeric, censored as well as multivariate response variables and arbitrary measurement scales of the covariates.

MAR,[19] which builds a regression model using the techniques in Reference 62. It is a form of regression analysis that is an extension to linear regression that captures nonlinearities and interactions between variables.

5.1 | FTLA

For FTLA with the fixed problem size (matrix sizes from 6000 to 20,000 with a stride of 2000 and a block size of 100, strong scaling), we ran the FTLA with five numbers of error injections (1, 2, 3, 4, and 5) on six different numbers of cores (32, 64, 128, 256, 512, and 1024) with five CPU frequency settings (1.2, 1.5, 1.8, 2.1, and 2.3 GHz) to collect the total 144 data samples on Shepard. Each data sample includes 53 variables such as application name, system name, number of cores, matrix sizes, stride size, block size, number of error injections, CPU frequency, 32 available performance counters, runtime, system power, CPU power, memory power, and so on. The 32 performance counters are TOT_CYC, TOT_INS, L1_TCM, L2_TCM, L3_TCM, CA_SHR, BR_CN, BR_TKN, BR_NTK, BR_MSP, CA_CLN, CA_ITV, RES_STL, L2_TCA, L1_STM, L2_TCW, L1_LDM, L2_DCA, L2_DCR, L2_DCW,

L1_ICM, BR_INS, L1_DCM, L2_ICA, TLB_DM, TLB_IM, L2_DCM, L2_ICM, LD_INS, SR_INS, L2_LDM, L2_STM, then TOT_CYC is used to normalize all the performance counters. The metrics for performance and power are runtime, node power, CPU power and memory power. We split the data as training and test datasets with the 80/20% rule so that the training dataset consists of 116 samples, and the test dataset consists of 28 samples. For the fair comparison, we apply the same training and test datasets to all modeling methods.

For the sake of simplicity, we use ML methods to model the performance (runtime) and node power only in this section. Figure 18 shows the performance prediction error rates using 10 ML methods and MuMMI. These violin plots visualize the distribution of the prediction error rates for each method. For MuMMI (MuM), the prediction error rates are between −0.41% and 0.37% in runtime. We observe that Cub and xGB resulted in the lowest error rates in runtime among 10 ML methods, and for other ML methods, the maximum error rates are more than 50%. Overall, MuMMI outperformed all ML methods in performance prediction.

Figure 19 shows the node power prediction error rates using MuMMI and 10 ML methods. For MuMMI (MuM), the prediction error rates are between −6.31% and 6.06% in node power. MAR, Cub and xGB resulted in the lowest error rates in node power among 10 ML methods and outperformed MuMMI. Because the runtime of the strong scaling FTLA had decreased significantly with the increased number of cores, but the average node power had decreased a little bit, when we used the datasets to build performance counter-based model for the runtime or the node power using 10 ML methods, the only difference is from the object metric (runtime or node power). This is mainly why 10 ML methods performed much better for power prediction than for performance prediction.

For the sake of simplicity, we choose four ML methods: Cub, xGB, RF, and MAR to do an in-depth analysis in performance and power modeling and prediction. For Cub, the prediction error rates are between −45.75% and 27.88% in runtime, and between −3.78% and 3.61% in node power. The performance model under-predicted for most cases. Let's look at how the variables contribute in performance and node power models. To measure predictor importance for Cub models,[7] we can enumerate how many times a predictor variable was used in either a linear model or a split and use these tabulations to get a rough idea the impact each predictor has on the model. Figure 20 shows the important predictors for performance model of FTLA, where the x-axis is the total usage of the predictor (i.e., the number of times it was used in a split or a linear model). The larger the importance value, the more important the predictor is in relating the latent predictor structure to the response. Very small importance values are likely not considered to contain predictive information for the response and should be considered as candidates for removal from the model. Figure 21 shows the important predictors for node power model of FTLA. We observe that the top 3 counters in performance model are L2_DCA, TLB_DM, and L2_DCR; the top 3 counters in power model are L2_TCM, L2_ICA, and L2_LDM. Overall, L2 cache and TLB mainly impact the performance and power

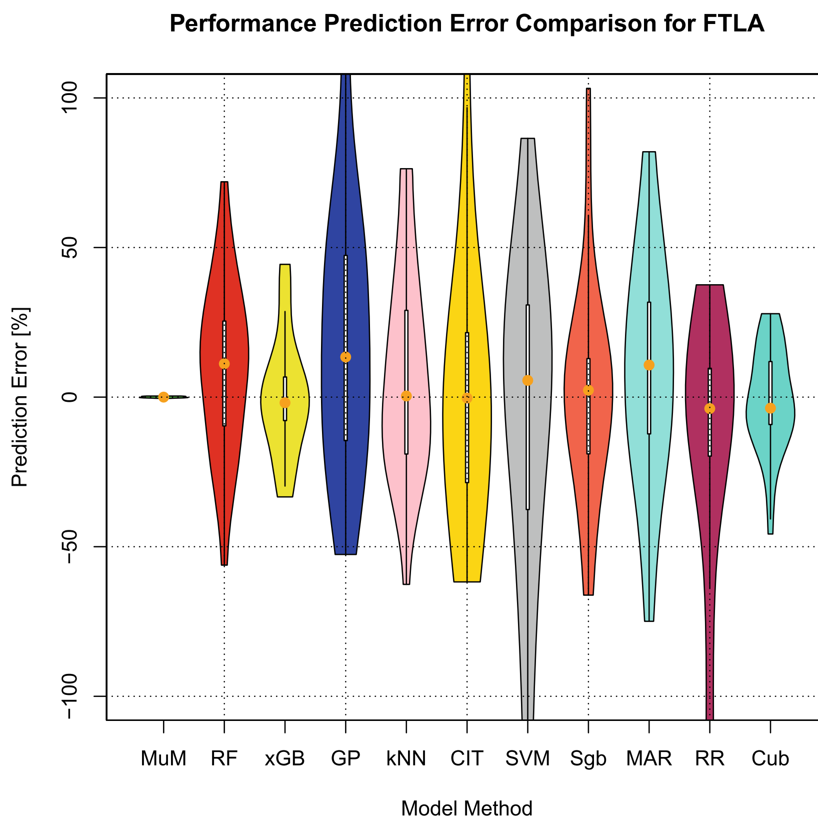

FIGURE 18 Prediction error rates (runtime) for FTLA

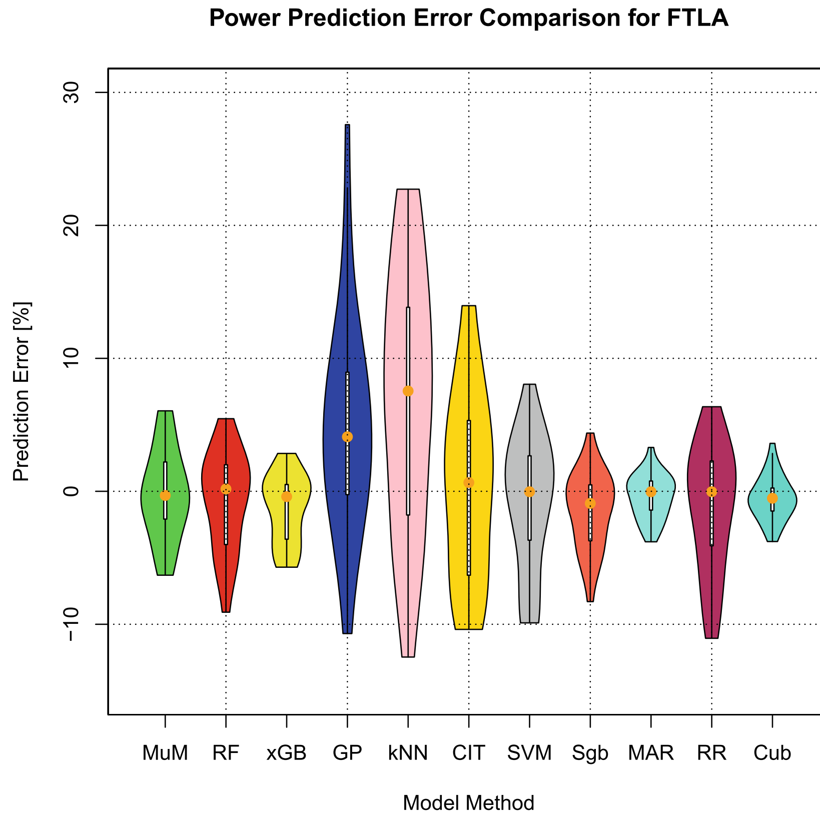

FIGURE 19 Prediction error rates (node power) for FTLA

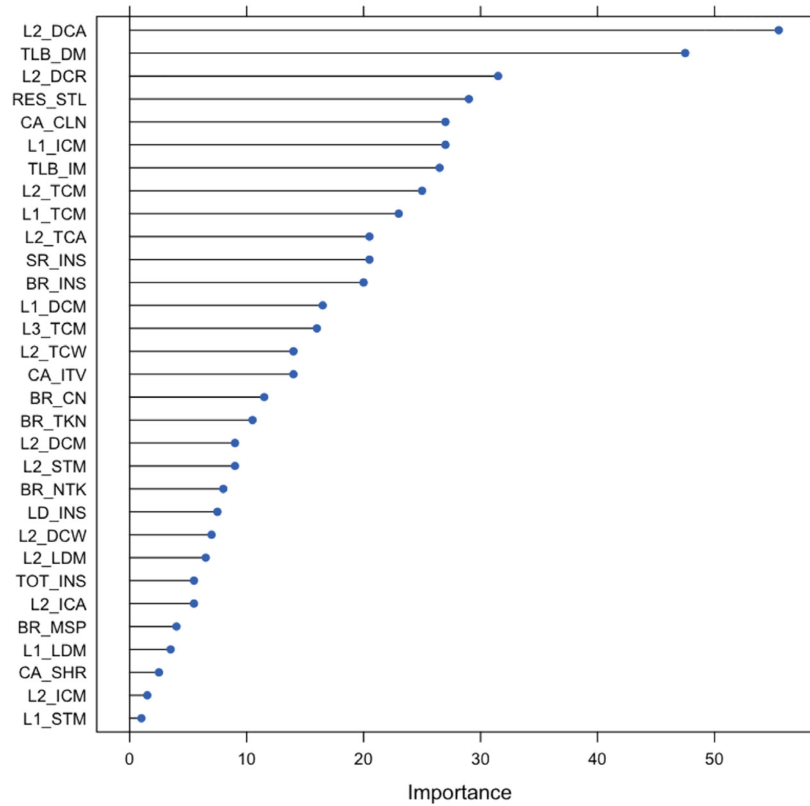

FIGURE 20 Variable importance for performance model

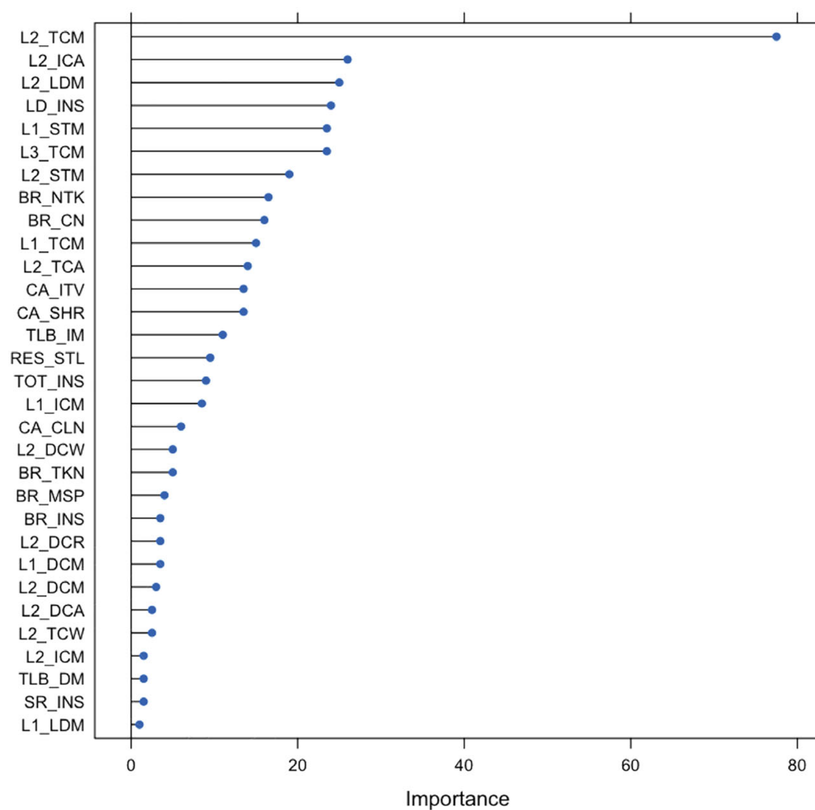

FIGURE 21 Variable importance for node power model

models using Cub, however, it is interesting to observe that the top 3 counters in performance model are in the bottom of the counter list in power model, and the top 3 counters in power model are also in the bottom of the counter list in performance model.

For xGB, the prediction error rates are between −33.35% and 44.38% in runtime, and between −5.71% and 2.85% in node power. Variable importance for the boosting method is a function of the reduction in squared error. Figure 22 shows the important predictors for performance model of FTLA. Figure 23 shows the variable importance for node power model of FTLA. We observe that the top 3 counters in performance model are TLB_DM, L2_TCW, and L2_DCA; the top 3 counters in power model are L2_TCM, L1_STM, and LD_INS. Overall, TLB, L2 cache and L1 cache mainly impact the performance and power models using xGB. Similarly, we observe that the top 3 counters in performance model are in the bottom of the counter list in power model, and the top 3 counters in power model are also in the bottom of the counter list in performance model. Contrasting the importance results to Cub in Figures 20 and 21, we see that 2 of the top 5 counters are the same (L2_DCA and TLB_DM in performance model; L2_TCM and L1_STM in power model), however, the importance orderings are much different.

For RF, the prediction error rates are between −56.15% and 71.97% in runtime, and between −9.09% and 5.46% in node power. Figure 24 shows the variable importance for performance model of FTLA. Figure 25 shows the variable importance for node power model of FTLA. We observe that the top 3 counters in performance model are L2_DCA, TLB_DM, and L1_ICM; the top 3 counters in power model are L2_TCM, L1_STM, and L2_STM. Overall, L2 cache, TLB, and L1 cache mainly impact the performance and power models using RF, however, it is interesting to observe that the top 3 counters in performance model are in the bottom of the counter list in power model, and the top 3 counters in power model are also in the bottom of the counter list in performance model. The variable importance in RF models is similar to that in Cub models, albeit in different order, because both are tree/rule-based models.

The prediction error rates using MAR are between −56.15% and 71.97% in runtime, and between −9.09% and 5.46% in node power. Figure 26 shows the variable importance with only 10 counters used in performance model of FTLA. Figure 27 shows the variable importance with only 4 counters used in node power model of FTLA. We observe that the top 3 counters in performance model are TLB_DM, L1_ICM, and RES_STL; the top 3 counters in power model are L2_TCM, BR_CN, and BR_NTK. Overall, TLB and L2 cache mainly impact the performance and power models using MAR.

Overall, Table 6 summarizes the top performance counters for each model for FTLA using the four ML methods. For the four ML methods, we find that L2_DCA or TLB_DM are one of the dominant factors in performance models, and L2_TCM is the dominant factor in power models. This

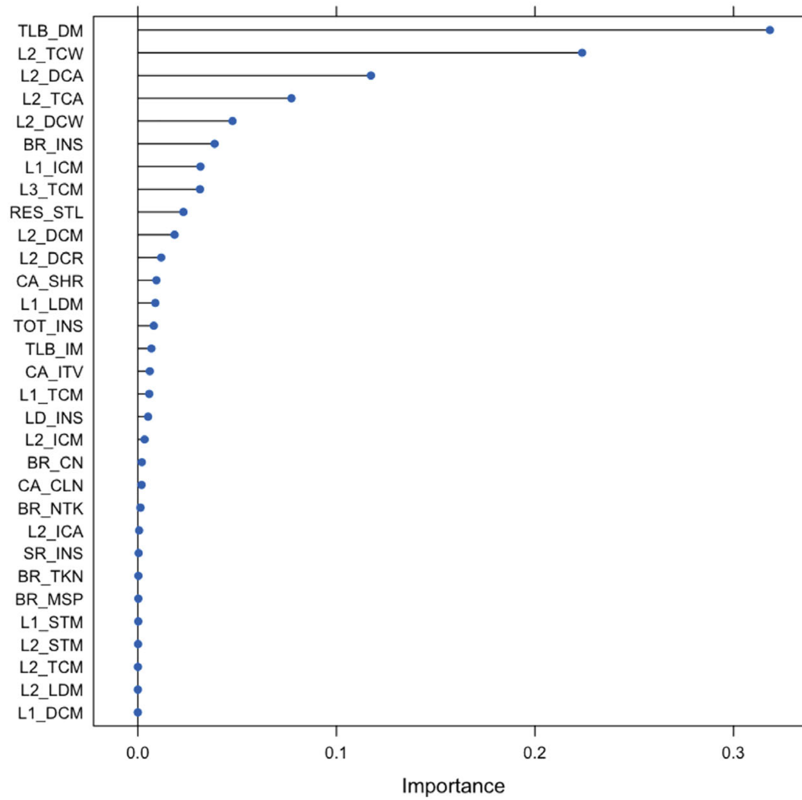

FIGURE 22 Variable importance for performance model

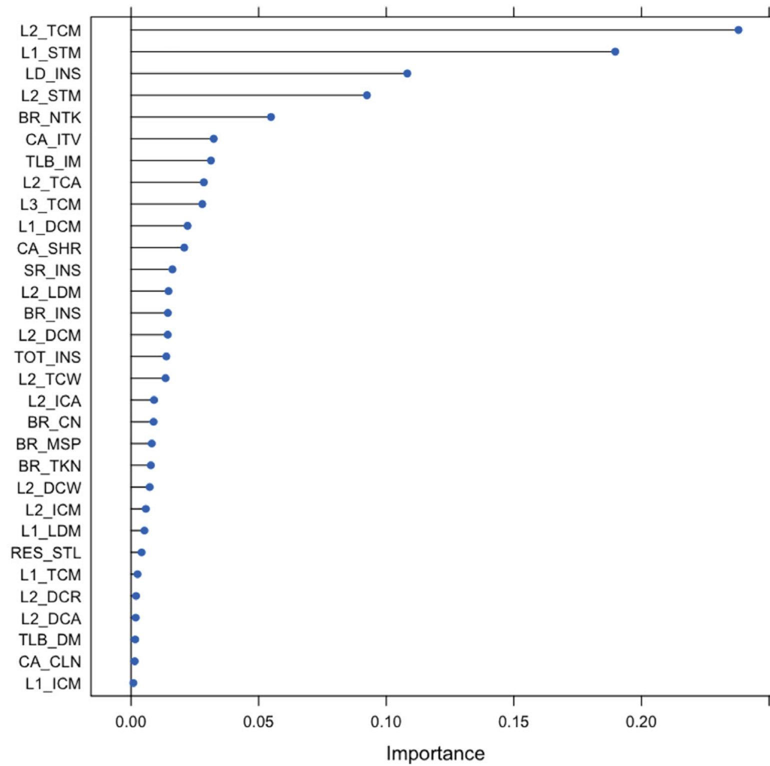

FIGURE 23 Variable importance for node power model

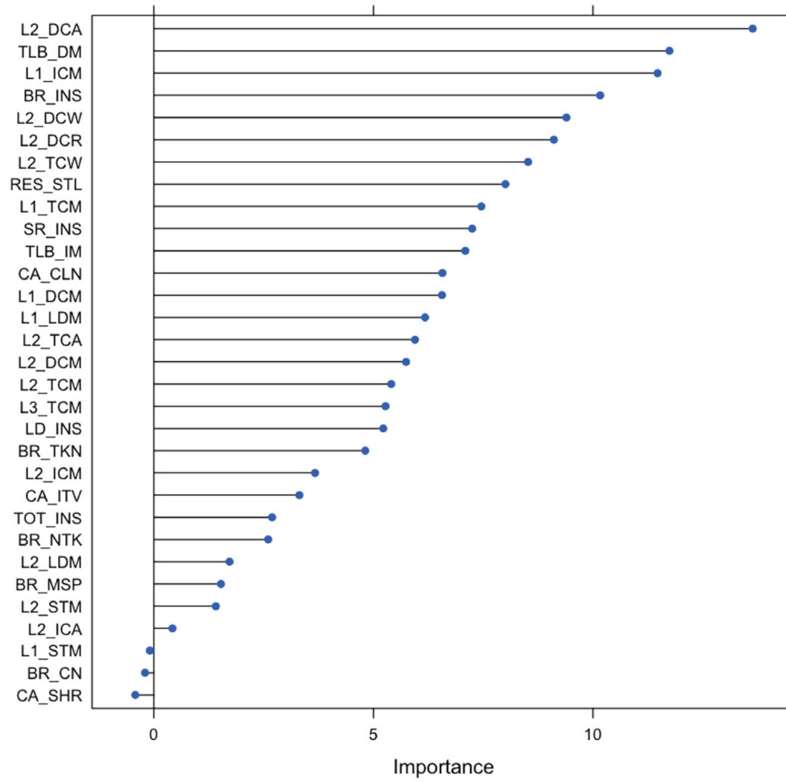

FIGURE 24 Variable importance for performance model

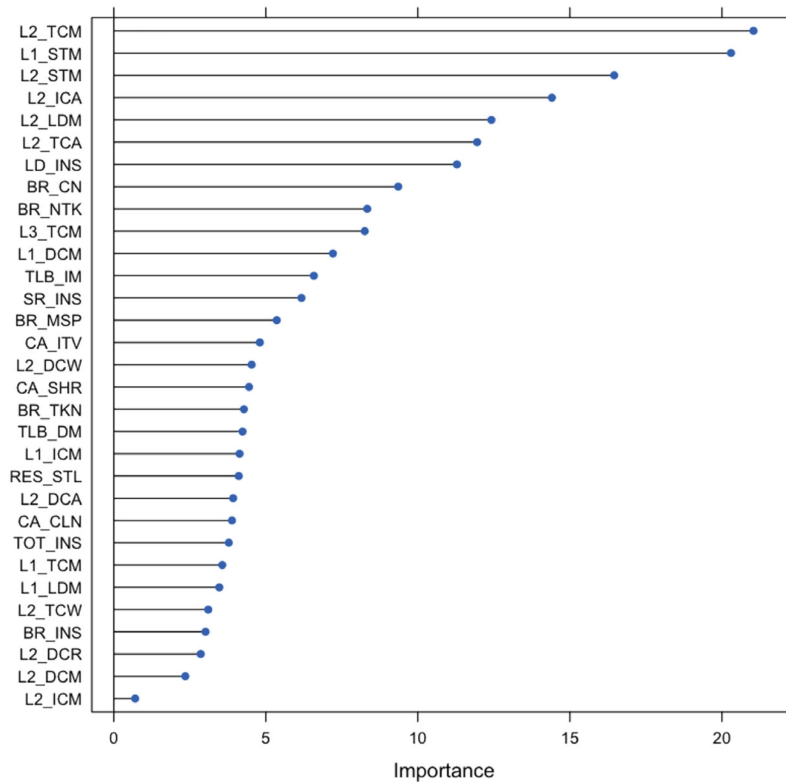

FIGURE 25 Variable importance for node power model

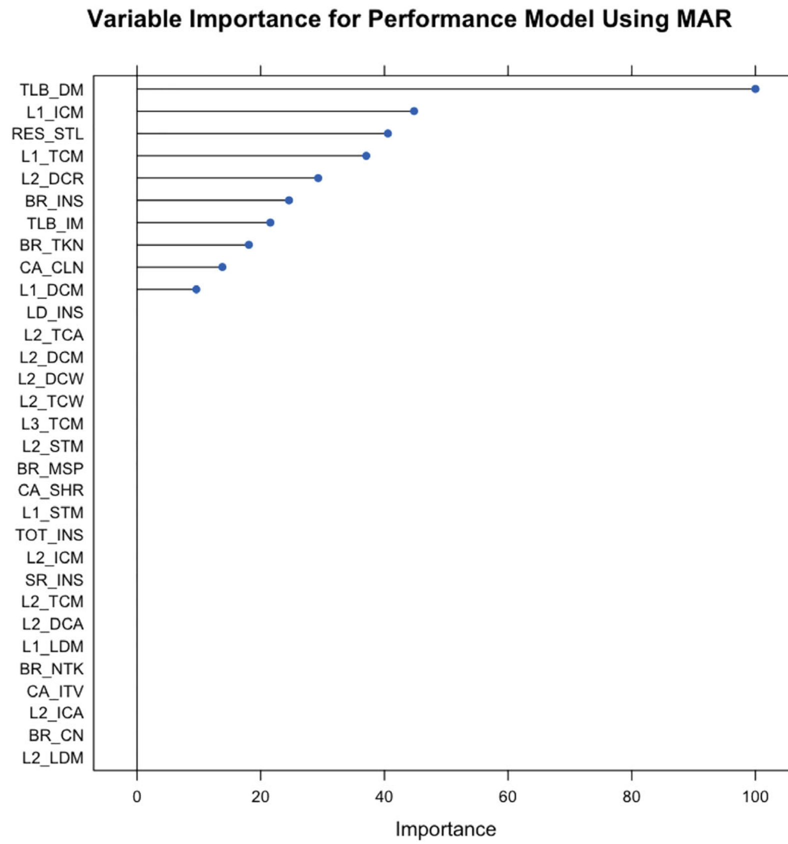

FIGURE 26 Variable importance for performance model

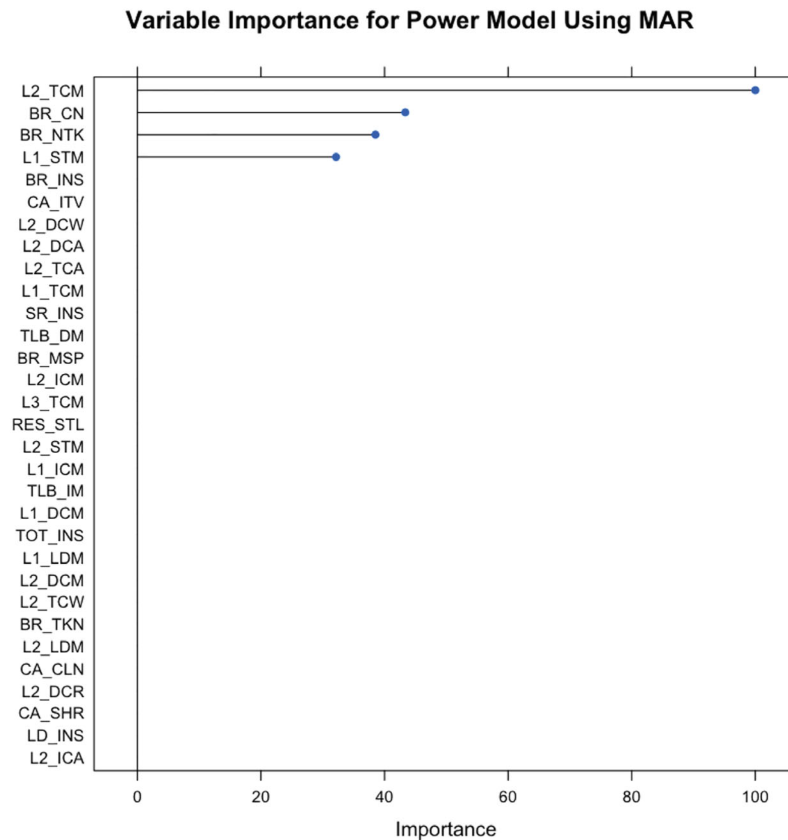

FIGURE 27 Variable importance for node power model

TABLE 6 Top performance counters for each model using ML methods for FTLA

ML method	Runtime model	Node power model
Cubist	L2_DCA	L2_TCM
Extreme gradient boosting	TLB_DM	L2_TCM
Random forests	L2_DCA	L2_TCM
Multivariate adaptive regression spline	TLB_DM	L2_TCM

indicates that MuMMI and the ML methods identify the same important performance counter in the power models. MuMMI requires the small amount of data, but the ML methods requires large amount of data for high prediction accuracy. Given the datasets for FTLA, the 10 ML models have been fit to the same datasets. Since the ML methods have their own way of learning the relationship between the predictors and the target object and provide different variable importance, it is not easy to identify which ML provides the robust variable importance.

5.2 | HDC

For HDC with the checkpointing file size of 2 MB per MPI process (weak scaling), we ran the HDC with 10 different four-levels checkpointing configurations on eight distinct numbers of cores (32, 64, 128, 256, 512, 640, 960, and 1024) with the CPU frequency of 2.3 GHz to collect the total 80 data samples. Each data sample includes 54 variables such as application name, system name, number of cores, number of iterations, checkpointing file size, Level 1 checkpoint, Level 2 checkpoint, Level 3 checkpoint, Level 4 checkpoint, CPU frequency, 32 available performance counters, runtime, system power, CPU power, memory power, and so on. We split the data as training and test datasets with the 80/20% rule so that the training dataset consists of 64 samples, and the test dataset consists of 16 samples. For the fair comparison, we apply the same training and test datasets to all modeling methods.

Figure 28 shows the performance prediction error rates using the 10 ML methods and MuMMI. For MuMMI (MuM), the prediction error rates are between −9.07% and 3.60% in runtime. We observe that xGB resulted in the lowest error rates (between −6.57% and 5.40%) in performance

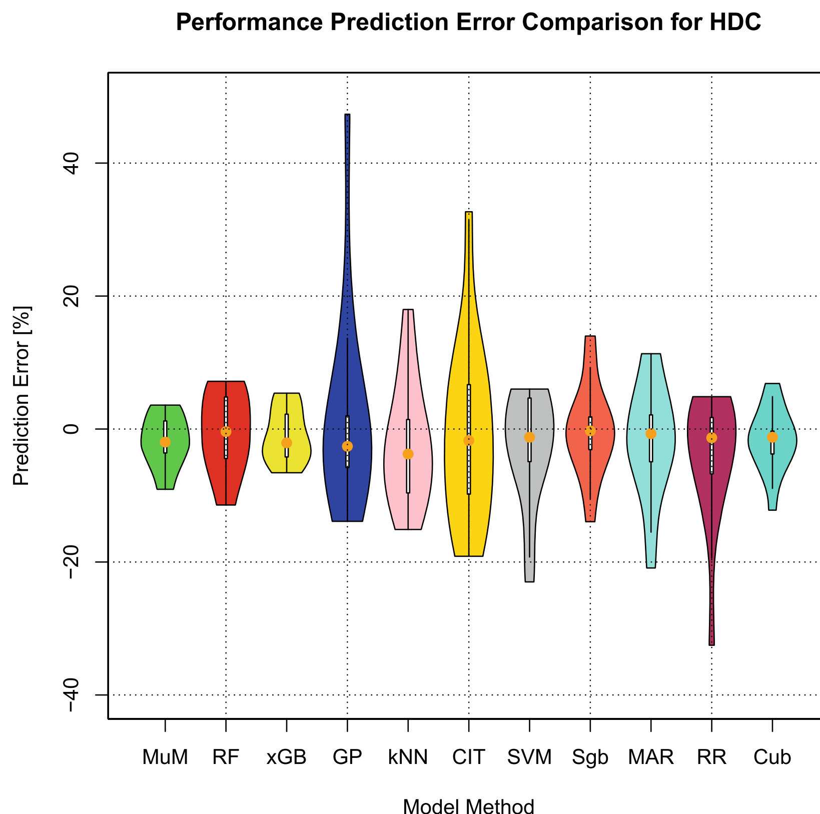

FIGURE 28 Prediction error rates (runtime) for HDC

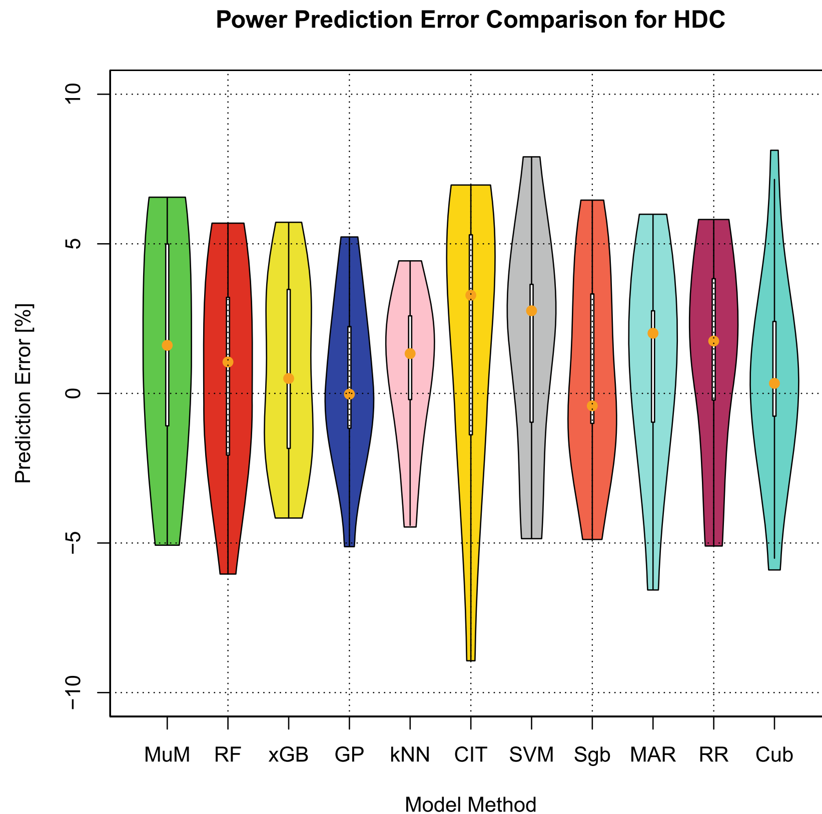

FIGURE 29 Prediction error rates (node power) for HDC

models among the 10 ML methods and outperformed MuMMI, and for other ML methods, the maximum error rates are more than 11%. Therefore, MuMMI outperformed the ML methods in performance prediction except xGB.

Figure 29 shows the node power prediction error rates using MuMMI and the 10 ML methods. For MuMMI (MuM), the prediction error rates are between −5.07% and 6.56% in node power. We observe that all these error rates are between −9% and 9%. kNN, xGB, GP, and RR resulted in the lowest error rates in node power among the 10 ML methods and outperformed MuMMI. Because the runtime of the weak scaling HDC had increased with the increased number of cores because of the increased communication overhead, and the average node power had decreased slightly. When we used the datasets to build performance counter-based model for the runtime or the node power using the 10 ML methods, the only difference is from the object metric (runtime or node power). This is mainly why 10 ML methods performed much better for power prediction than for performance prediction.

The prediction error rates using xGB are between −6.57% and 5.40% in runtime, and between −4.17% and 5.72% in node power. Figure 30 shows the variable importance for performance model of HDC. Figure 31 shows the variable importance for node power model of HDC. We observe that the top 3 counters in performance model are BR_INS, TLB_DM, and L2_TCW; the top 3 counters in power model are CA_ITV, L1_TCM, and L2_TCW. It is interesting to observe that the top counter BR_INS in performance model is in the bottom of the counter list in power model, and the top counter CA_ITV in power model is also in the bottom of the counter list in performance model.

The prediction error rates using kNN are between −15.11% and 17.98% in runtime, and between −4.47% and 4.43% in node power. Figure 32 shows the variable importance for performance model of HDC. Figure 33 shows the variable importance for node power model of HDC. We observe that the top 3 counters in performance model are L2_DCR, L2_DCA, and TLB_DM; the top 3 counters in power model are TLB_IM, BR_CN, and L2_DCR. Contrasting the importance results to xGB in Figures 30and 31, we see that 4 of the top 5 counters are the same in performance model and only 1 of the top 5 counters is the same in power model, and the importance orderings are much different.

Overall, Table 7 summarizes the top performance counters for each model for HDC using the ML methods xGB and kNN. These top performance counters for performance and power models are different. Given the datasets for HDC, the 10 ML models have been fit to the same datasets, however, the ML methods have their own way of learning the relationship between the predictors and the object metric and provide different variable importance, thus it is not easy to identify which ML provides the robust variable importance.

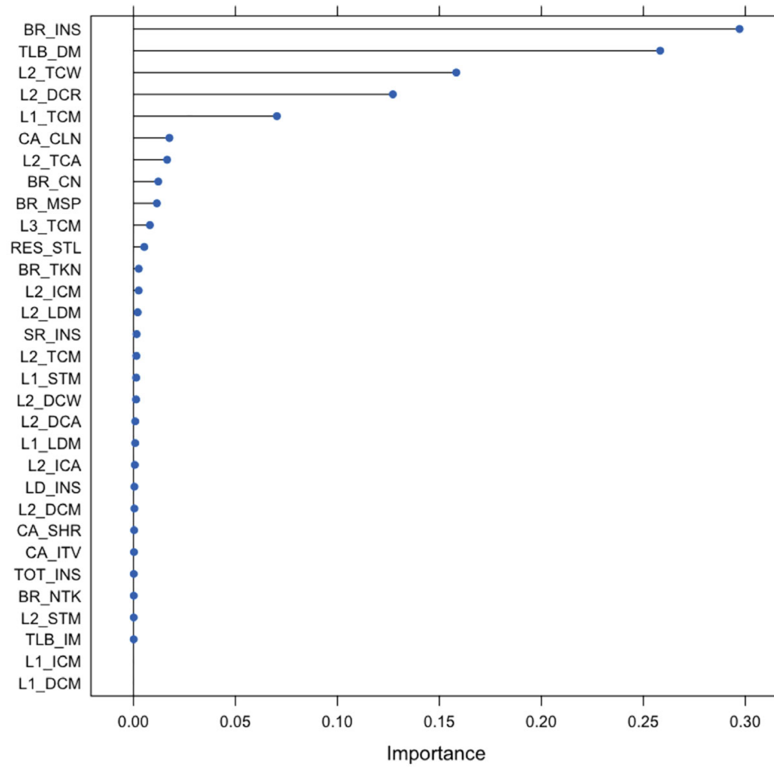

FIGURE 30 Variable importance for performance model

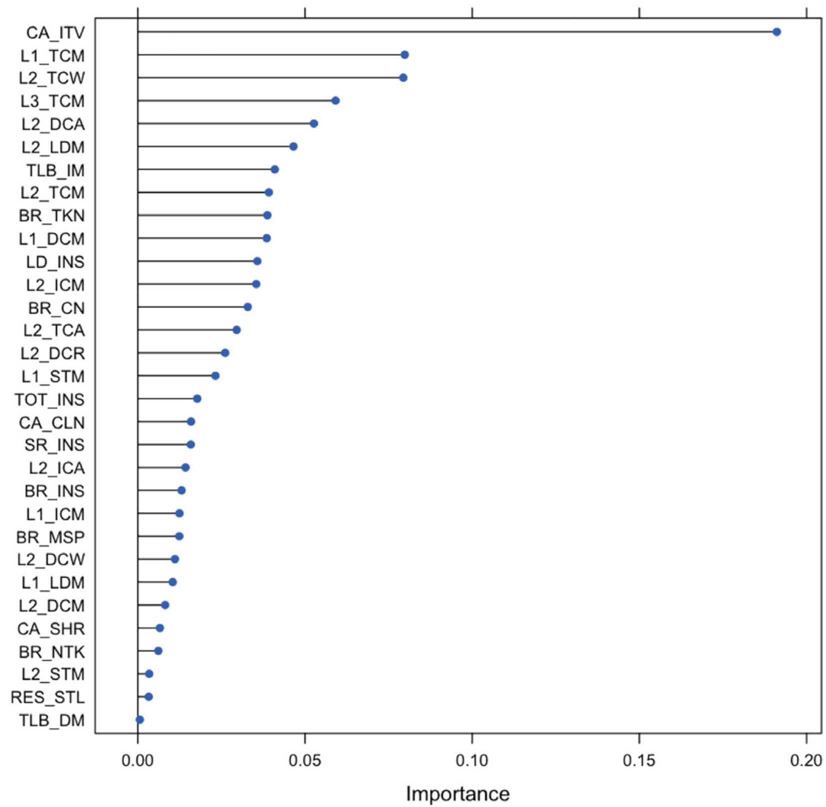

FIGURE 31 Variable importance for node power model

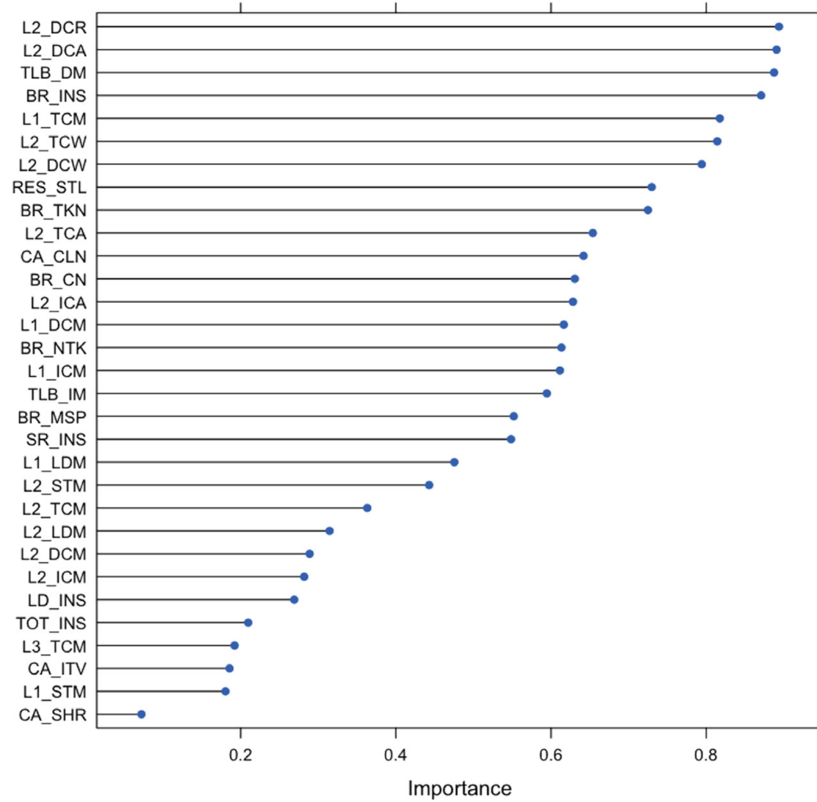

FIGURE 32 Variable importance for performance model

Variable Importance for Power Model Using k-Nearest Neighbors

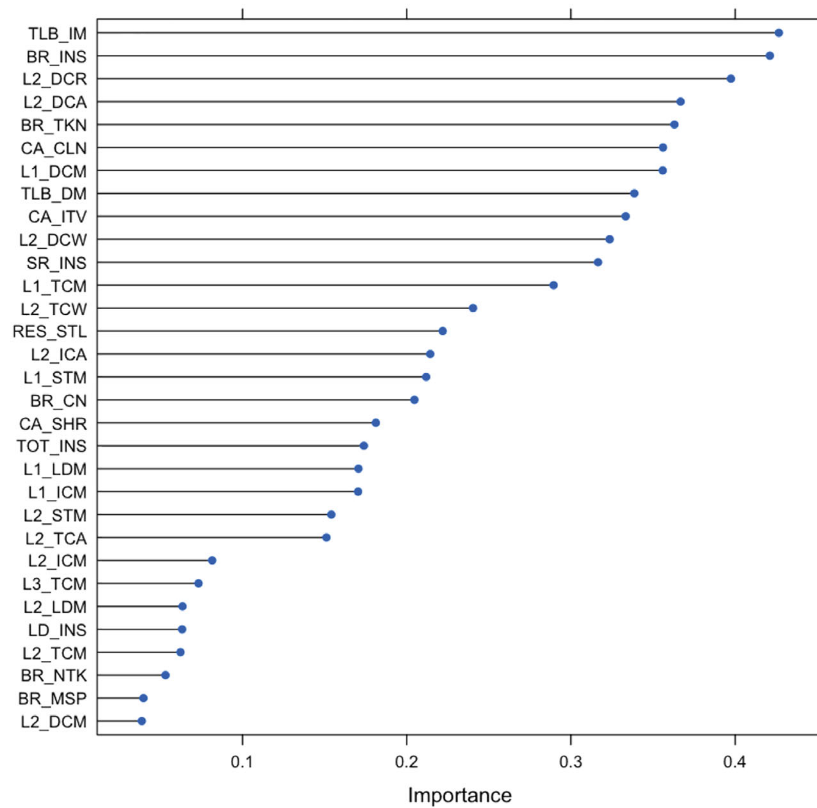

FIGURE 33 Variable importance for node power model

TABLE 7 Top performance counters for each model using ML methods for HDC

ML method	Runtime model	Node power model
Extreme gradient boosting	BR_INS	CA_ITV
k-Nearest neighbors	L2_DCR	TLB_IM

6 | CONCLUSIONS

We used MuMMI and the 10 ML methods to model, predict and compare the performance and power of the strong scaling FTLA and the weak scaling HDC. Our experiment results show that the prediction error rates in performance and power using MuMMI are less than 10% for most cases. Based on the performance counters of these models, we identified the most significant performance counters for potential optimization efforts associated with the application characteristics on these systems, and we used our what-if prediction system to predict the theoretical performance and power of a possible application optimization. These performance and power models were generated from different system configurations and problem sizes, thus providing a broader understanding of the application's usage of the underlying architectures. This in turn resulted in more knowledge about the application's energy consumption on a given architecture.

When we compared the prediction accuracy using MuMMI with that using the 10 ML methods, we observe that MuMMI outperformed these ML methods in performance prediction except one case which the xGB outperformed MuMMI for the application HDC, however, in power prediction, only three or four ML methods outperformed MuMMI. The 10 ML methods performed much better for the weak scaling HDC than the strong scaling FTLA because of the much smaller difference in the runtime and node power in the datasets of HDC. As we illustrated that the 10 ML methods had their own way of learning the relationship between the predictors and the target object metric and provide different variable importance, it is not easy to identify which ML provides the robust variable importance for potential application improvements. To address the issue in our future work, we plan to utilize ensemble learning to combine several ML methods to result in more accurate models and provide the robust variable importance for the latent improvements as shown in Reference 63. Overall, performance and power modeling tools like MuMMI and some ML methods are able to aid in application optimizations for energy efficiency, power or energy-aware job schedulers, and system performance and power tuning. The methodology presented in this article can be applied to large scale scientific applications[6] and deep learning applications[64] on other HPC systems.

ACKNOWLEDGMENTS

This work was supported in part by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under contract DE-AC02-06CH11357, in part by DoE RAPIDS2, and in part by NSF grants CCF-1801856 and CCF-2119203. We acknowledge Argonne Leadership Computing Facility for use of Cray XC40 Theta and BlueGene/Q Mira under the DOE INCITE project PEACES and ALCF project EE-ECP, and Sandia National Laboratories for use of Intel Haswell Shepard testbed. We also acknowledge the reviewers for their valuable suggestions and comments.

DATA AVAILABILITY STATEMENT

Data are available on request from the authors.

ORCID

Xingfu Wu https://orcid.org/0000-0001-8150-5171

REFERENCES

1. Du P, Bouteiller A, Bosilca G, Herault T, Dongarra J. Algorithm-based fault tolerance for dense matrix factorization. Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP; 2012.
2. Bouteiller A, Herault T, Bosilca G, Du P, Dongarra J. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Trans Parallel Comput*. 2015;1(2):1-22.
3. FTLA Fault tolerant linear algebra. http://icl.cs.utk.edu/ft-la/software/index.html (ftla-rSC13.tgz).
4. FTI Fault tolerance interface, leobago.github.io/fti/.
5. Wu X, Taylor V, Lively C, et al. MuMMI: multiple metrics modeling infrastructure (Book chapter). Tools for High Performance Computing; 2014; Springer.
6. Wu X, Taylor V, Cook J, Mucci P. Using performance-power modeling to improve energy efficiency of HPC applications. *IEEE Comput*. 2016;49(10):20-29.
7. Kuhn M, Johnson K. *Applied Predictive Modeling*. Springer; 2013.
8. caret package. https://topepo.github.io/caret/index.html, https://cran.r-project.org/web/packages/caret/
9. Theta, Cray XC40 system. https://www.alcf.anl.gov/theta
10. MIRA. IBM BlueGene/Q system. https://www.alcf.anl.gov/mira
11. Shepard. *Advanced Systems Technology Test Beds*, Sandia National Laboratories. 2019. http://www.sandia.gov/asc/computational_systems/HAAPS.html
12. Liaw A, Wiener M. Breiman and Cutler's random forests for classification and regression, package εrandomForestε; March 25, 2018.

13. Karatzoglou A, Smola A, Hornik K. kernlab – An S4 package for kernel methods in R; 2004.
14. Chen T, He T, Benesty M, et al. Extreme gradient boosting, package εxgboostε; August 1, 2019,
15. Greenwell B, Boehmke B, Cunningham J. Generalized boosted regression models, Package εgbmε; January 14, 2019.
16. Kuhn M, Weston S, Keefer C, Coulter N, Quinlan R. Rule- and instance-based regression modeling, package εCubistε; January 10, 2020. https://topepo.github.io/Cubist
17. Zou H, Hastie T. Elastic-net for sparse estimation and sparse PCA, package εelasticnetε; August 31, 2018.
18. Hothorn T, Hornik K, Strobl C, Zeileis A. A Laboratory for Recursive Partytioning, Package εpartyε; March 5, 2020.
19. Milborrow S. Multivariate adaptive regression splines, package εearthε; November 9, 2019.
20. Bautista-Gomez L, Komatitsch D, Maruyama N, Tsuboi S, Cappello F, Matsuoka S. FTI: high performance fault tolerance interface for hybrid systems, SC11, Seattle, WA, 2011.
21. Scalable checkpoint/restart project. https://computation.llnl.gov/project/scr/
22. Moody A, Bronevetsky G, Mohror K, de Supinski BR. Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system, SC10, 2010.
23. VeloC: very low overhead transparent multilevel checkpoint/restart. http://www.mcs.anl.gov/project/veloc-very-low-overhead-transparent-multilevel-checkpointrestart.
24. Plank J, Li K, Puening M. Diskless checkpointing. *IEEE Trans Parallel Distrib Syst*. 1998;9(10):972-986.
25. Elliott J, Kharbas K, Fiala D, Mueller F, Ferreira K, Engelmann C. Combining partial redundancy and checkpointing for HPC, ICDCS'12, 2012.
26. Ferreira K, Riesen R, Bridges P, et al. Evaluating the viability of process replication reliability for exascale systems, SC2011; 2011.
27. Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K, Brightwell R. Detection and correction of silent data corruption for large-scale high performance computing SC2012; 2012.
28. Nagarajan A, Mueller F, Engelmann C, Scott S. Proactive fault tolerance for HPC with Xen virtualization. ICS06, 2006.
29. Huang K, Abraham J. Algorithm-based fault tolerance for matrix operations. *IEEE Trans Comput*. 1984;33(6):518-528.
30. Anfinson C, Luk F. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans Comput*. 1988;37(12):1599-1604.
31. Chen Z, Dongarra J. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources, IPDPS'06.
32. Bosilca G, Delmas R, Dongarra J, Langou J. Algorithm-based fault tolerance applied to high performance computing. *J Parallel Distrib Comput*. 2009;69:410-416.
33. Davies T, Karlsson C, Liu H, Ding C, Chen Z. High performance linpack benchmark: a fault tolerant implementation without checkpointing. Proceedings of the International Conference on Supercomputing (ICS'11); 2011.
34. Hakkarinen D, Chen Z. Algorithmic Cholesky factorization fault recovery. Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'10); 2010.
35. Du P, Luszczek P, Tomov S, Dongarra J. Soft error resilient QR Factorization for Hybrid System with GPGPU. SC2011 Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems; 2011.
36. ScaLAPACK: scalable linear algebra PACKage. http://www.netlib.org/scalapack/.
37. Wu X, Taylor V, Lan Z. Evaluating runtime and power requirements of multilevel checkpointing MPI applications on four different parallel architectures: an empirical study. Proceedings of the2018 Cray User Group Conference, Stockholm, Sweden; 2018.
38. Aupy G, Benoit A, Herault T, Robert Y, Dongarra J. Optimal checkpointing period: time vs Energy. In 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems; 2013.
39. Meneses E, Sarood O, Kale LV. Assessing energy efficiency of fault tolerance protocols for HPC systems. Proceedings of the 24th IEEE International Symposium on Computer Architecture and High Performance Computing; 2012.
40. Balaprakash P, Bautista-Gomez L, Bouguerra M, Wild SM, Cappello F, Hovland PD. Analysis of the tradeoffs between energy and run time for multilevel checkpointing. Proceedings of the 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems; 2014.
41. el Mehdi Diouri M, Gluck O, Lefevre L, Cappello F. Energy considerations in checkpointing and fault tolerance protocols. Proceedings of the 2nd International Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS); 2012.
42. el Mehdi Diouri M, Gluck O, Lefevre L, Cappello F. ECOFIT: a framework to estimate energy consumption of fault tolerance protocols for HPC applications. Proceedings of the 13th IEEE/ACM International Symposium Cluster, Cloud and Grid Computing; 2013.
43. Tan L, Kothapalli S, Chen L, Hussaini O, Bissiri R, Chen Z. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Comput*. 2014;40:559-573.
44. Malakar P, Balaprakash P, Vishwanath V, Morozov V, Kumaran K. Benchmarking machine learning methods for performance modeling of scientific applications. Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS18); 2018.
45. Bertran R, Sugawara Y, Jacobson H, Buyuktosunoglu A, Bose P. Application-level power and performance characterization and optimization on IBM blue gene/Q systems. *IBM J Res Develop*. 2013;57(1):1-17.
46. Martin S, Rush D, Kappel M, Sandstedt M, Williams J. Cray XC40 power monitoring and control for knights landing. Proceedings of the 2016 Cray User Group Conference; 2016.
47. Rotem E, Naveh A, Rajwan D, Ananthakrishnan A, Weissmann E. Power-management architecture of the Intel microarchitecture code-named sandy bridge. *IEEE Micro*. 2012;32(2):20-27.
48. NVIDIA. NVML API reference manual; 2012.
49. Marincic I, Vishwanath V, Hoffmann H. PoLiMEr: an energy monitoring and power limiting interface for HPC applications. Proceedings of the SC2017 Workshop on Energy Efficient Supercomputing; 2017.
50. Wallace S, Vishwanath V, Coghlan S, Tramm J, Lan Z, Papka ME. Application power profiling on blue Gene/Q. Proceedings of the IEEE Conference on Cluster Computing; 2013.
51. Laros JH, Pokorny P, DeBonis D. PowerInsight — A commodity power measurement capability, International Green Computing Conference; 2013.
52. violin package. https://cran.r-project.org/web/packages/vioplot/index.html, https://www.data-to-viz.com/graph/violin.html
53. Breiman L. Random forests. *Mach Learn*. 2001;45:5-32.

54. Williams C, Rasmussen C. Gaussian processes for regression. *Adv Neural Inf Process*. 1995;8:514-520. http://books.nips.cc/papers/files/nips08/0514.pdf
55. Chen T, Guestrin C. XGBoost: a scalable tree boosting system, KDD'16; August 13-17, 2016.
56. Freund Y, Schapire R. Experiments with a new boosting algorithm. Proceedings of the 13th International Conference on Machine Learning; 1996.
57. Friedman J. Greedy function approximation: a gradient boosting machine. *Ann Stat*. 2001;29(5):1189-1232.
58. Quinlan R. Learning with continuous classes. Proceedings of the 5th Australian Joint Conference on Artificial Intelligence; 1992:343-348.
59. Quinlan R. Combining instance-based and model-based learning. Proceedings of the 10th International Conference on Machine Learning; 1993:236-243.
60. Hoerl A. Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*. 1970;12(1):55-67.
61. Vapnik V. *Statistical Learning Theory*. Wiley; 1998.
62. Friedman JH. Multivariate adaptive regression splines. *Ann Stat*. 1991;19(1):1-67.
63. Wu X, Taylor V. Utilizing ensemble learning for performance and power modeling and improvement of parallel cancer deep learning CANDLE benchmarks. *Concurr Comput Pract Exp*. 2021;1-18.e6516. doi:10.1002/cpe.6516
64. Wu X, Taylor V, Wozniak JM, Stevens R, Brettin T, Xia F. Performance, energy, and scalability analysis and improvement of parallel cancer deep learning CANDLE benchmarks. Proceedings of the 48th International Conference on Parallel Processing; August 5-8, 2019; Kyoto, Japan.

How to cite this article: Wu X, Taylor V, Lan Z. Performance and power modeling and prediction using MuMMI and 10 machine learning methods. *Concurrency Computat Pract Exper*. 2022;e7254. doi: 10.1002/cpe.7254