Performance and Power Modeling and Prediction Using MuMMI and Ten Machine Learning Methods

Xingfu Wu, Valerie Taylor
Argonne National Laboratory
University of Chicago
Email: {xingfu.wu, vtaylor}@anl.gov

Zhiling Lan
Department of Computer Science
Illinois Institute of Technology
Email: lan@iit.edu

***Abstract*—In this paper, we use the modeling and prediction tool Multiple Metrics Modeling Infrastructure (MuMMI) and 10 machine learning methods to model and predict performance and power and compare their prediction error rates. We use a fault-tolerant linear algebra code and a fault-tolerant heat distribution code to conduct our modeling and prediction study on the Cray XC40 Theta and IBM Blue Gene/Q Mira at Argonne National Laboratory and the Intel Haswell cluster Shepard at Sandia National Laboratories. 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 models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters for potential optimization efforts associated with the application characteristics and the target architectures, and we predict theoretical outcomes of the potential optimizations. When we compare the prediction accuracy using MuMMI with that using 10 machine learning methods, we observe that MuMMI not only results in more accurate prediction of both performance and power but also presents how performance counters contribute in the performance and power models. This information provides some insights into how to fine-tune the applications and/or systems for energy efficiency.**

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 [14] [6]. Such real-world scientific applications take a long time to execute on current supercomputers, often facing software or hardware failures that delay the time to solution; and future HPC systems are expected to have additional power and resilience requirements representing a multidimensional tuning challenge. To embrace these key challenges, we must understand the complicated tradeoffs among runtime, power, and resilience.

In this paper we explore performance and power modeling and prediction of an algorithm-based fault-tolerant linear algebra code (FTLA) [26] and a fault-tolerant heat distribution code (HDC) [25] using the Multiple Metrics Modeling Infrastructure (MuMMI) [58] [59] and 10 machine learning methods from the R caret package [32] [9].

MuMMI facilitates systematic measurement, modeling, and prediction of performance and power consumption and performance-power tradeoffs and optimization for parallel applications on given HPC systems. The 10 machine learning methods are random forests [35], Gaussian process with radial basis function [31], eXtreme Gradient Boosting [12], stochastic gradient boosting [18], Cubist [33], ridge regression [62], k-nearest neighbors [32], support vector machine with a linear kernel [31], conditional inference tree [28], and multivariate adaptive regression spline [39].

We conduct our experiments on the Cray XC40 Theta [51] and IBM Blue Gene/Q Mira [40] at Argonne National Laboratory and on the Intel Haswell cluster Shepard [50] at Sandia National Laboratories. 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 models for runtime, node power, CPU power, and memory power, we identify the most significant performance counters for potential optimization efforts associated with the application characteristics and the target architectures, and we predict theoretical outcomes of the potential optimizations. When we compare the prediction accuracy using MuMMI with that using the 10 machine learning methods, we observe that MuMMI results in more accurate prediction in both performance and power.

The remainder of this paper is organized as follows. Section 2 discusses the applications FTLA and HDC. Section 3 briefly describes the three architectures used in our experiments and their power-profiling tools. Section 4 presents performance and power characteristics, modeling, and prediction of FTLA using MuMMI. Section 5 discusses the modeling and prediction of FTLA and HDC using 10 machine learning methods and compares them with MuMMI. Section 6 summarizes this work. Notice that we use the formula $(prediction - baseline)/baseline * 100\%$ to calculate the prediction error rate in this paper.

2. Fault-Tolerant Applications: FTLA and HDC

A number of resilience methods have been developed for preventing or mitigating failure impact. Existing resilience strategies can be broadly classified in four approaches: checkpoint based, redundancy based, proactive, 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 the last saved checkpoint in the event of a failure. 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) [7] [25], Scalable Checkpoint/Restart (SCR) [48] [41], VeloC [54], and diskless checkpointing [44]. Redundancy approaches improve resilience by replicating data or computation [21] [22] [23]. Proactive methods take preventive actions before failures, such as software rejuvenation and process or object migration [42]. 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 [30] [1] [10] [5] [6]. ABFT has been applied to High-Performance Linpack (HPL) [13], to Cholesky factorization [27], and to LU and QR factorizations [14] [15] [6]. FTLA [26] [6], developed in particular as an extension to ScaLAPACK [49], tolerates and recovers from fail-stop failure, a process that completely stops the system component from responding, triggering the loss of a critical part of the global application state and halting the application execution.

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 code ftla-rSC13 [26] 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 block QR and LU is identical. We focus on QR in this paper. 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 again is associated with the matrix sizes. For each matrix size, it performs one QR followed by one RQR, which 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 paper, we use the three codes ftla-rSC13, $ftla$, and la to conduct our experiments. They are strong-scaling codes.

The other application used in this paper is an FTI version of the MPI heat distribution code (HDC) [25], which computes the heat distribution over time based on a set of initial heat sources. FTI [7] leverages local storage, along with data replication and erasure codes, to provide several levels of reliability and performance. It provides four levels of 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), single-node failure, multiple-node failure, and all other failures that the lower levels cannot take care of, respectively. The checkpointing file size is 2 MB per MPI process. HDC is a compute-intensive, weak-scaling code.

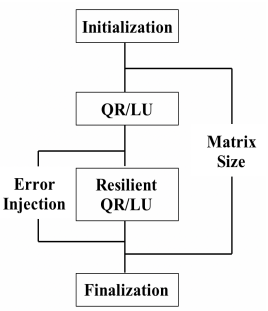

Figure 1. Control flow of the ftla-rSC13 code

While fault tolerance methods and power-capping techniques continue to evolve, tradeoffs among execution time, power efficiency, and resilience strategies are still not well understood. 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 [60] presented an empirical study evaluating the runtime and power requirements of multilevel checkpointing MPI applications using FTI on four different parallel architectures. Recent research has focused on a theoretical analysis of energy and runtime for fault tolerance protocols [2] [38] [3] [19] [20] [52]. In this paper, we use FTLA and HDC for our modeling and prediction study.

3. System Architectures and Environments

We conduct our experiments on three parallel systems with different architectures: the Cray XC40 Theta [51] and IBM BG/Q Mira [40] at Argonne National Laboratory and the Intel Haswell cluster Shepard [50] at Sandia National Laboratories. 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), 192 GB of DDR4 RAM, and a 128 GB SSD. Each BG/Q node has 16 compute cores—one BG/Q PowerPC A2 1.6 GHz chip with the TDP of 55 W [7]) and shared L2 cache of 32 MB and 16 GB of memory. Each Haswell node 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.

Several vendor-specific power management tools exist, such as Cray's CapMC and out-of-band and in-band power monitoring capabilities [37], IBM EMON API on BG/Q [7], Intel RAPL [47], and NVIDIA's power management library [43]. In this work, we use simplified PoLiMEr [36] to measure power consumption for the node, CPU, and memory at the node level on Theta; we use MonEQ [56] to collect power profiling data on Mira; and we use PowerInsight [34] to measure the power consumption for the node, CPU, memory, and hard disk at the node level on Shepard.

PoLiMEr uses Cray's CapMC to obtain power and energy measurements of the node, CPU, and memory on Theta. The power-sampling rate used is approximately 2 samples per second (default). On Mira, EMON API [7] provides 7 power domains to measure the power consumption for the node, CPU, memory, and network at the node-card level. The power-sampling rate used is approximately 2 samples per second (default). 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. Hence, we conduct our experiments on multiple node-cards to obtain the power-profiling data. 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 discuss our use of the three codes ftla-rSC13, ftla, and la with matrix sizes from 6,000 to 20,000 with a stride of 2,000 and a block size of 100 to conduct our FTLA experiments with a maximum of 5 error injections on each of the three computer platforms. We analyzed their performance and power characteristics and used the performance counter-based modeling tool MuMMI [58] [59] to model performance and power and to predict theoretical outcomes for the potential optimizations.

We used MuMMI with support of PoLiMEr, MonEQ, and PowerInsight to instrument these codes in order to collect performance data, power data, and performance counters on the Cray XC40 Theta, IBM BG/Q Mira, and Intel Haswell Shepard. We used the same default compiler options from the FTLA code ftla-rSC13 to compile the codes. We executed each application 14 times on each system to ensure the consistency of the results while collecting different sets of performance counters with a total of 40 performance counters for performance and power modeling. Since the variation of the application runtime is very small (less than 1%), we used the performance metrics corresponding to the smallest runtime for our work.

4.1. Cray XC40 Theta

Each XC40 node [51] has one Intel KNL, which brings in an on-package memory called Multi-Channel DRAM (MCDRAM) in addition to the traditional DDR4 RAM.

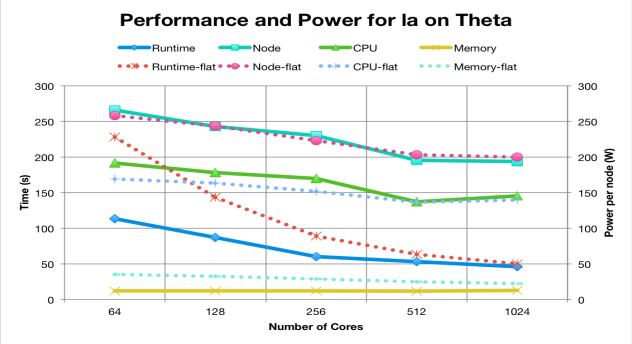

Figure 2. Performance and power comparison for la on Theta

MCDRAM has a high bandwidth (4 times more than DDR4 RAM) and low-capacity (16 GB) memory. MCDRAM can be configured as a shared L3 cache (cache mode) or as a distinct NUMA node memory (flat mode). The different memory modes make it challenging from a software perspective to understand the best mode for an application. We used the codes la and $ftla$ to investigate the performance and power impacts under the cache and flat mode use of MCDRAM.

Figure 2 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. The advantage of using MCDRAM as cache is that an application may run entirely in MCDRAM, improving the application performance significantly. We find that the application runtime using the cache mode is almost half of the runtime using the flat mode on 64 cores. Both runtimes are close as the number of cores increases to 1,024, however, because the application is strong scaling and the amount of workload per core decreases by 16 times. These results indicate that to take advantage of MCDRAM requires a large amount of workload per core. From the figure, we also observe that both node power consumptions are close. 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 3. Therefore, in the remainder of this section, we use the cache mode for our experiments on Theta.

Figure 4 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 1,024 cores.

Figure 5 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 FTLA with one error injection and five error injections on 1,024 cores. As we can see in Figure 6, the CPU power mainly affects the node power changes for both cases. Because of the dynamic power

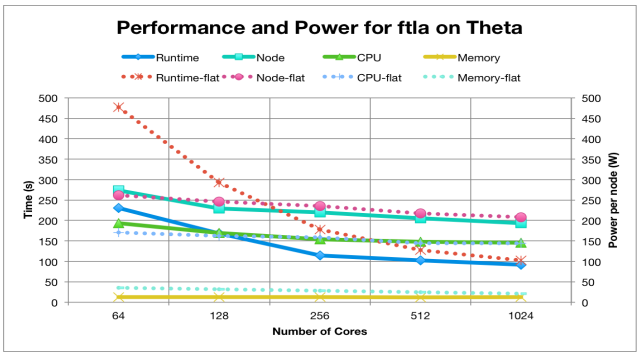

Figure 3. Performance and power comparison for ftla on Theta

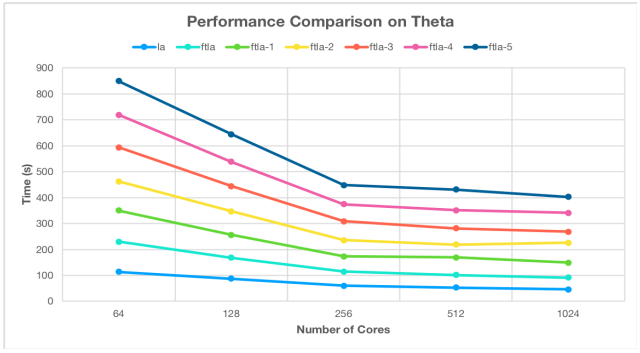

Figure 4. Performance comparison on Theta

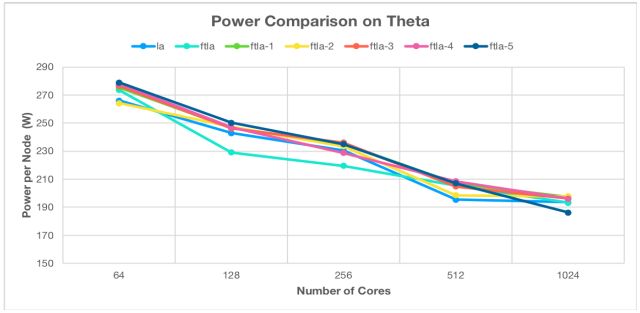

Figure 5. Power comparison on Theta

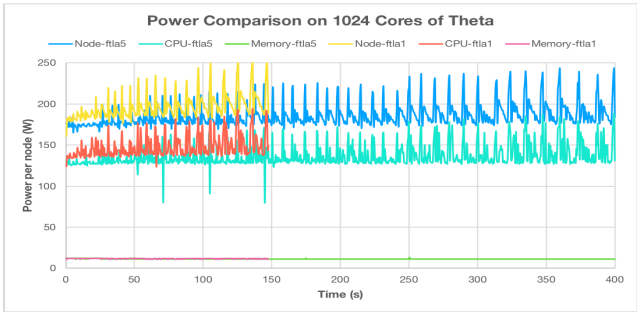

Figure 6. Power over time on 1,024 cores of Theta

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 7. Four models of runtime, node power, CPU power, and memory power on Theta

management on Theta, during each matrix loop the power adjusts dynamically, increasing with the increase in matrix size from 6,000 to 20,000. The runtime mainly results in a large energy increase.

To develop accurate models of runtime and power consumptions for the code ftla-rSC13, we used the power and performance modeling tool MuMMI from our previous work [59] [58]. We collected 40 available performance counters on Theta with different system configurations (numbers of cores: 64, 128, 256, 512, and 1,024) and different numbers of error injections (1, 2, and 3) as a training set. We then used a Spearman correlation and principal component analysis (PCA) to identify the major performance counters $(r_1, r_2, ..., r_n (n << 40))$, which are highly correlated with the metric: runtime, system power, CPU power, or memory power. Then we used a nonnegative multivariate regression analysis to generate our four models based on the small set of major counters and CPU frequency (f), as shown in Figure 7, where a numeric value is the coefficient for the counter in the corresponding model.

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

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

Here, T is the component predictor used to represent the value for runtime; the intercept is β_0; each β_n represents the regression coefficient for performance counter r_n; and β represents the coefficient for the CPU frequency. Equation 1 can be used to predict the runtime for larger numbers of error injections (4 or 5 error injections).

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

$$P = \alpha_0 + \alpha_1 * r_1 + \alpha_2 * r_2 ..... \alpha_n * r_n + \alpha * f^3. \quad (2)$$

Here, P is the component predictor used to represent the value for the CPU power; the intercept is α_0; each α_n represents the regression coefficient for performance counter r_n; 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

TABLE 1. 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%

regression model was constructed for each metric (node power, memory power) of the same application.

Table 1 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. These rates indicate 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 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 identified the most significant performance counters for the application. Figure 8 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 models; L2_DCM and L1_TCM (Level 1 cache misses) contribute most in the node power models; TLB_DM and L1_TCM contribute 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, optimization efforts for the code should focus on the units associated with L2_DCM, TLB_DM, and L2_STM on Theta. For instance, as shown in Figure 9, we used our what-if prediction system based on the four model equations to predict the theoretical outcomes of the possible optimization by reducing L2_DCM by 30%; the other counters may be changed based on the correlation with this counter. The theoretical improvement is 2.99% in runtime, 10.08% in node power, 7.44% in CPU power, and 7.10% in memory power. The default page size on Theta is 4 KB; but Theta supports several huge page sizes ranging from 2 MB to 2 GB. 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 application performance improvement.

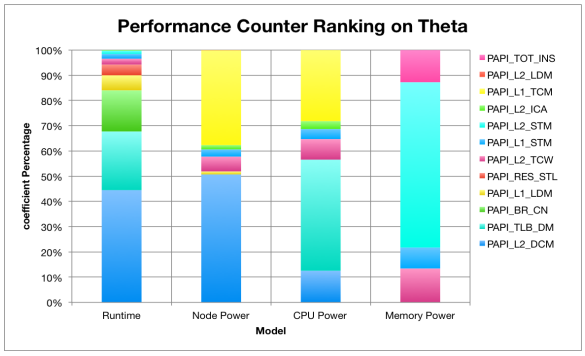

Figure 8. Counter ranking on Theta

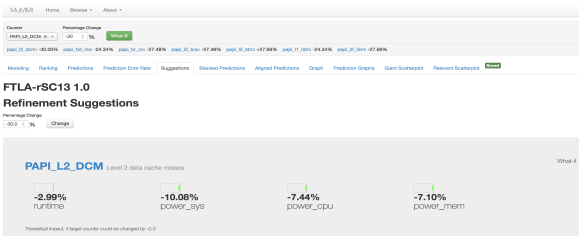

Figure 9. Theoretical prediction on Theta

4.2. IBM Blue Gene/Q Mira

To develop accurate models for runtime and power consumption on Mira, we collected 40 available performance counters with different system configurations (numbers of cores: 512, 1,024, 2,048, 4,096, 8,192, and 16,384) and different numbers of error injections (1, 2, and 3) as a training set. We then used MuMMI to generate our four models based on the small set of major counters and CPU frequency (f), as shown in Figure 10, where a numeric value is the coefficient for the counter in the corresponding model.

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 in runtime are less than 0.1%. These indicate that our counter-based performance models are accurate. The prediction error rates are less than 5% in node power and less than 8.7%

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 10. Four models for runtime, node power, CPU power, and memory power on Mira

TABLE 2. PREDICTION ERROR RATES ON MIRA

#Cores	ftla-4				ftla-5			
	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%
16384	0.076%	2.45%	5.52%	-9.62%	-0.086%	0.97%	3.03%	-6.36%

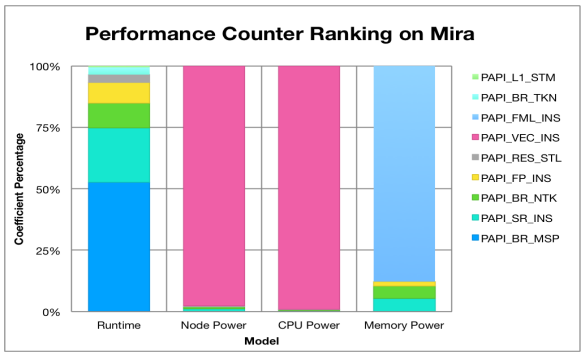

Figure 11. Counter ranking on Mira

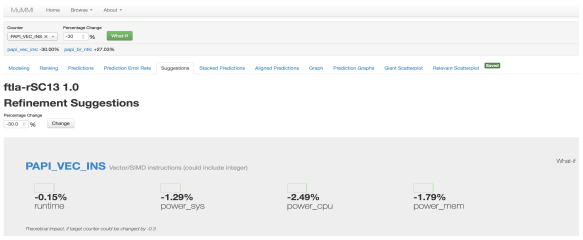

Figure 12. Theoretical prediction on Mira

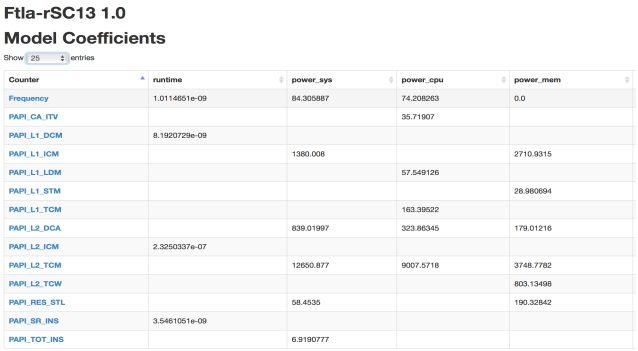

Figure 13. Four models for runtime, node power, CPU power, and memory power on Shepard

in CPU power; and the error rates are less than 10% in memory power for most cases except 15.30% for ftla-4 on 2,048 cores.

Based on the models for runtime, node power, CPU power, and memory power, we identified the most significant performance counters for the application. Figure 11 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, optimization efforts for the code should focus on the units 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 12, 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 is 0.15% in runtime, 1.29% in node power, 2.49% in CPU power, and 1.79% in memory power.

4.3. Intel Haswell Shepard

To develop accurate models for runtime and power consumption on Shepard, we used MuMMI to generate the four models based on the small set of major counters and CPU frequency (f), as shown in Figure 13 with the training dataset for different system configurations (numbers of cores: 32, 64, 128, 256, 512, and 1,024) and different numbers of error injections (1, 2, and 3).

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 (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% for node power and less than 6.6% for CPU power; and the prediction error rates for 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

TABLE 3. PREDICTION ERROR RATES ON SHEPARD

#Cores	ftla-4				ftla-5			
	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%

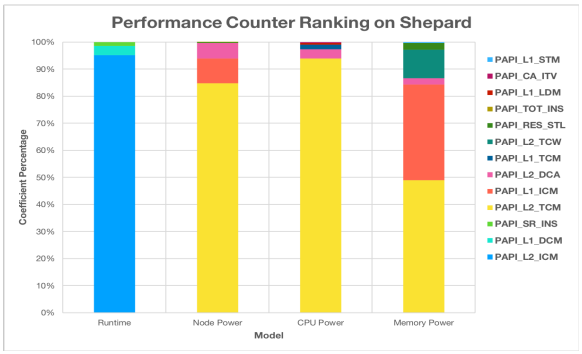

Figure 14. Counter ranking on Shepard

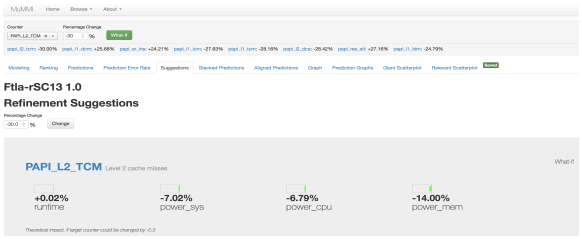

Figure 15. Theoretical prediction on Shepard

power, and memory power, we identify the most significant performance counters for the application. Figure 14 shows the performance counter rankings of the four models using 13 different counters. The results show 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, optimization efforts for the code should focus on the units associated with L2 and L1 caches on Shepard. For instance, as shown in Figure 15, we use our what-if prediction system based on the four model equations to predict the theoretical outcomes of the possible optimization by reducing L2_TCM by 30%; the other counters may be changed based on the correlation with this counter. The theoretical improvement 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.

5. Modeling and Prediction Using 10 Machine Learning Methods

In this section we discuss our use of 10 machine learning (ML) methods from the R caret package [9] [32] 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. Fourth, we compare the prediction error rates for these methods using a violin plot from the R violin package [55], which is a combination of a box plot and a kernel density plot to visualize the distribution of the prediction error rates.

The 10 ML methods are described as follows.

Random forest (RF) [35] is a classification algorithm based on a forest of trees using random inputs, which was constructed in [4] as a tree-based model. A random forest 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.

Gaussian process (GP) [57] with radial basis function [31] is based on the 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.

eXtreme Gradient Boosting (xGB) [12] is an efficient implementation of the gradient boosting framework in [11]. It provides a sparsity-aware algorithm for handling sparse data and a theoretically justified weighted quantile sketch for approximate learning.

Stochastic gradient boosting (Sgb) [18] is an implementation of extensions to the AdaBoost algorithm [16] and gradient boosting machine [17]. 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.

Cubist (Cub) [33] is a regression model using rules with added instance-based corrections that combines the ideas in [45] and [46]. A cubist regression model is fit for each rule based on the data subset defined by the rules. The set of rules is 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.

Ridge regression (RR) [62] [29] 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.

k-nearest neighbors (kNN) [32] 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 k-nearest neighbors in the predictor space. The predicted

response for the new sample is then the mean of the k neighbors' responses.

Support vector machine (SVM) with a linear kernel [31] is kernlab's implementation of support vector machines [53]. It chooses a linear function in the feature space by optimizing some criterion over the sample.

Conditional Inference Tree (CIT) [28] embeds tree-structured regression models into a well-defined theory of conditional inference procedures. This non-parametric class of regression trees is applicable to all kinds of regression problems, including nominal, ordinal, numeric, and censored as well as multivariate response variables and arbitrary measurement scales of the covariates.

mltivariate adaptive regression (MAR) [39] builds a regression model using the techniques in [24]. It is an extension to linear regression that captures nonlinearities and interactions between variables.

5.1. FTLA

For FTLA with fixed problem size (matrix sizes from 6,000 to 20,000 with a stride of 2,000 and a block size of 100, strong scaling), we ran 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 1,024) with five CPU frequency settings (1.2, 1.5, 1.8, 2.1, and 2.3 GHz) to collect a total of 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, and memory power. 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, and L2_STM. We then used TOT_CYC to normalize all the performance counters. We split the data as training and test datasets with the 80/20% rule so that the training dataset consisted of 116 samples and the test dataset of 28 samples. For fair comparison, we applied the same training and test datasets to all modeling methods.

Figure 16 shows the prediction error rates for the models in runtime, node power, CPU power, and memory power using MuMMI. The prediction error rates are between -0.41% and 0.37% in runtime and between -6.31% and 6.06% in node power. The mean error rate is 0.03% in runtime, -0.26% in node power, 0.93% in CPU power, and 1.00% in memory power. Overall, these predictions are accurate in runtime and power using MuMMI.

For simplicity, in this paper we use ML methods to model only the performance and node power. The violin plots in Figure 17 show the distribution of the performance prediction error rates for the 10 ML methods and MuMMI (MuM). We observe that Cub and xGB result in the lowest

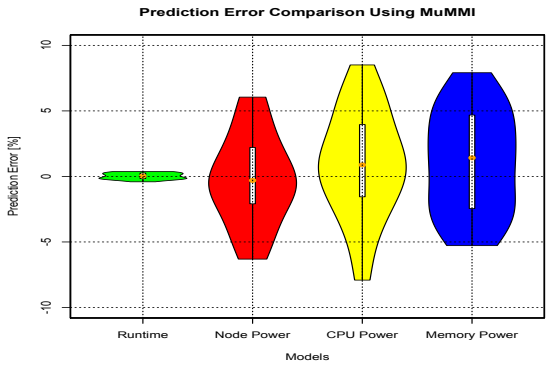

Figure 16. Prediction error rates for FTLA using MuMMI

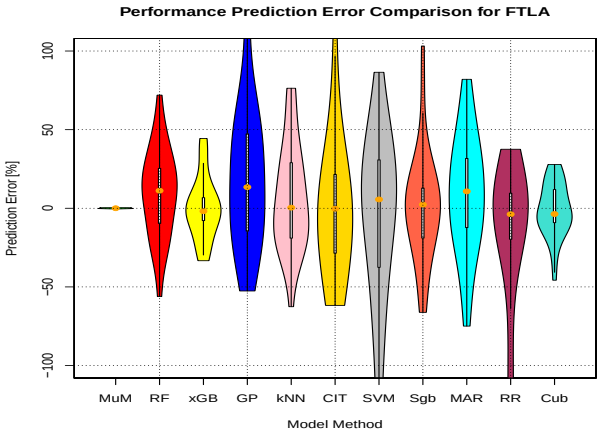

Figure 17. Prediction error rates (runtime) for FTLA

error rates in performance among the 10 ML methods. For the other ML methods, the maximum error rates are more than 50%. Overall, MuMMI outperforms all of them in performance prediction.

Figure 18 shows the node power prediction error rates using MuMMI and the 10 ML methods. We observe that the rates are between -15% and 30%. MAR, Cub, and xGB result in the lowest error rates in node power among the 10 ML methods and outperform MuMMI, although the node power prediction error rates using MuMMI are between -6.31% and 6.06%.

For simplicity, we chose the two ML methods—Cub and xGB—for an in-depth analysis of performance and power modeling and prediction.

Figure 19 shows the prediction error rates for runtime and node power models using 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 mean error rate is -1.16% in runtime and -0.54% in node power. The performance model underpredicted for the worst case. Let's look at how the variables contribute in the performance and node power models. To measure predictor importance for Cub models [32], we can enumerate how many times a

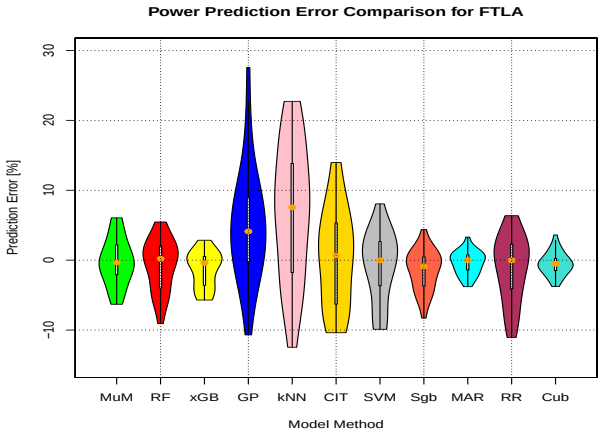

Figure 18. Prediction error rates (node power) for FTLA

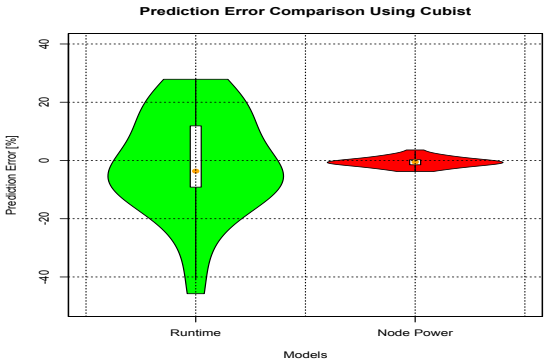

Figure 19. Prediction error rates using Cubist

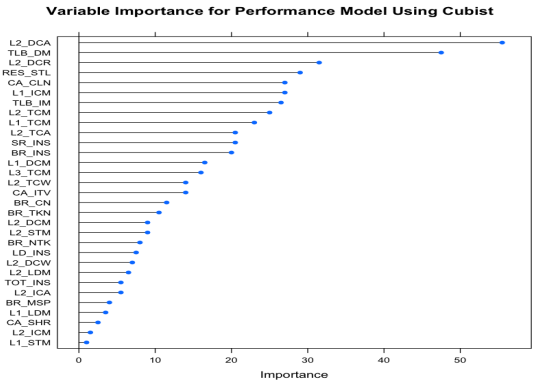

Figure 20. Variable importance for performance model of FTLA

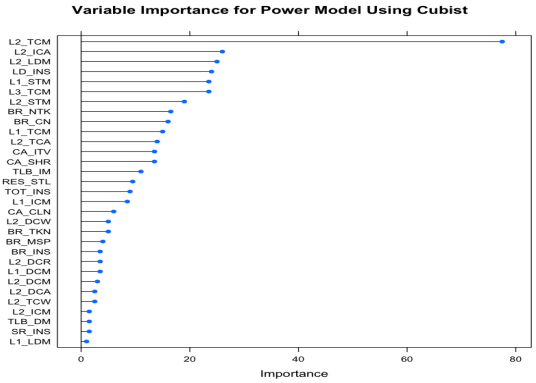

Figure 21. Variable importance for node power model of FTLA

predictor variable was used in either a linear model or a split and use these tabulations to get a rough idea of the impact each predictor has on the model. Figure 20 shows the top 31 most important predictors for the 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 to contain predictive information for the response and should be considered for removal from the model. Figure 21 shows the top 31 most important predictors for the node power model of FTLA. We observe that the top 3 counters in the performance model are L2_DCA, TLB_DM, and L2_DCR; the top 3 counters in the power model are L2_TCM, L2_ICA, and L2_LDM. Overall, L2 cache and TLB mainly impact the performance and power when using Cub. We observe, however, that the top 3 counters in the performance model are in the bottom of the counter list in the power model and that the top 3 counters in the power model are in the bottom of the counter list in the performance model.

Figure 22 shows the prediction error rates for runtime and node power models using xGB. The prediction error rates are between -33.35% and 44.38% in runtime and between -5.71% and 2.85% in node power. The mean error rate is 0.27% in runtime and -1.32% in node power. Variable importance in boosting is a function of the reduction in the squared error. Figure 23 shows the top 31 most important predictors for the performance model of FTLA. Figure 24 shows the variable importance for the node power model of FTLA. We observe that the top 3 counters in the performance model are TLB_DM, L2_TCW, and L2_DCA; the top 3 counters in the power model are L2_TCM, L1_STM, and LD_INS. Overall, TLB, L2 cache, and L1 cache mainly impact the performance and power when using xGB. Similarly, we observe that the top 3 counters in the performance model are in the bottom of the counter list in the power model, and the top 3 counters in the power model are in the bottom of the counter list in the performance model. Conparing the importance results to Cub in Figure 20 and 21, we see that 2 of the top 5 counters are the same (L2_DCA and TLB_DM in the performance model; L2_TCM and L1_STM in the power model); however, the importance orderings are much different.

In summary, we find that TLB_DM is one of the dominant factors in the performance models and that L2_TCM is the dominant factor in the power models. These results also validate that L2_TCM is the dominant factor in the power

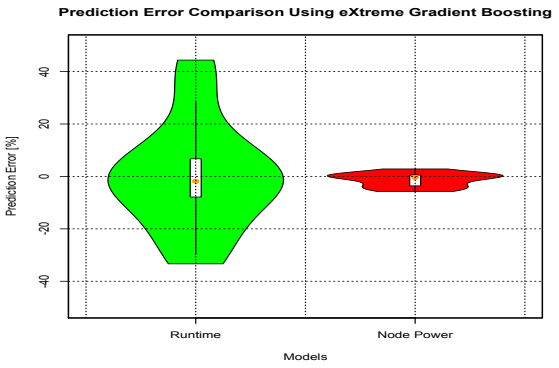

Figure 22. Prediction error rates using xGB

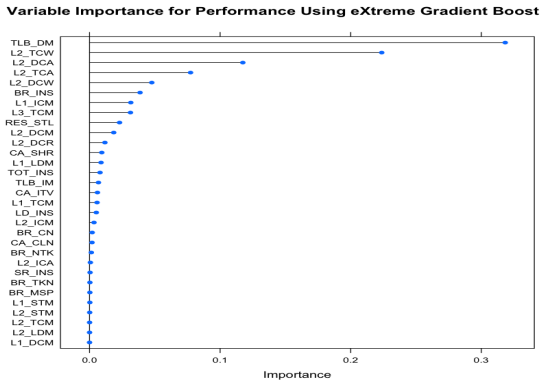

Figure 23. Variable importance for the performance model of FTLA

models using MuMMI,as shown in Figure 14. Since each ML method has a different way of learning the relationship between the predictors and the target object and provide different variable importance results, it is hard to identify which ML provides the most robust variable importance.

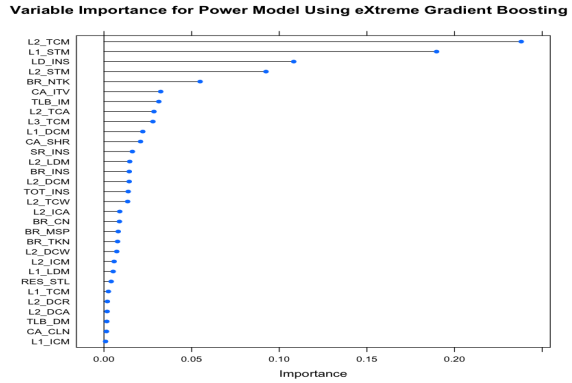

Figure 24. Variable importance for the node power model of FTLA

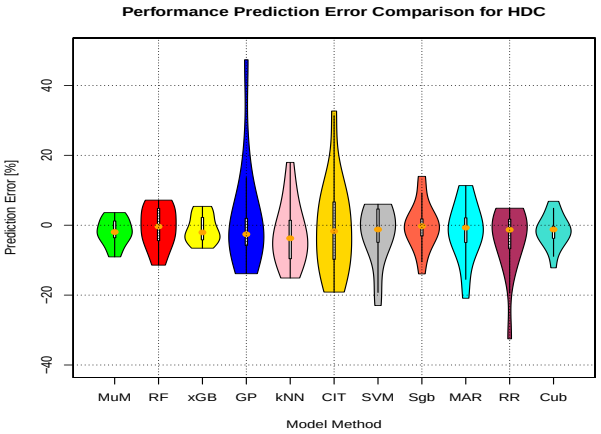

Figure 25. Prediction error rates (runtime) for HDC

5.2. HDC

We ran HDC with a checkpointing file size of 2 MB per MPI process (weak scaling), with 10 different four-level checkpointing configurations on eight distinct numbers of cores (32, 64, 128, 256, 512, 640, 960, and 1,024) with a CPU frequency of 2.3 GHz. We collected a total 80 data samples. Each data sample had 54 variables, including application name, system name, number of cores, number of iterations, checkpointing file size, Levels 1–4 checkpoint, CPU frequency, 32 available performance counters, runtime, system power, CPU power, and memory power. We split the data into training and test datasets with the 80/20% rule so that the training dataset consisted of 64 samples and the test dataset of 16 samples. For fair comparison, we applied the same training and test datasets to all modeling methods.

Figure 25 shows the performance prediction error rates using the 10 ML methods and MuMMI (MuM). We observe that xGB results in the lowest error rates in the performance models among the 10 ML methods; for the other ML methods, the maximum error rates are more than 11%. The performance prediction error rates using MuMMI are between -9.07% and 3.60% in runtime. Overall, MuMMI outperforms all of them in performance prediction.

Figure 26 shows the node power prediction error rates using MuMMI and the 10 ML methods. We observe that the error rates are between -9% and 9%. The prediction error rates for node power are between -4.17% and 5.72% using xGB, and between -4.47% and 4.43% using kNN. kNN and xGB have the lowest error rates in node power among the 10 ML methods and outperform MuMMI, although the node power prediction error rates using MuMMI are between -5.07% and 6.56%.

6. Conclusions

In this paper, we used MuMMI and 10 ML methods to model, predict, and compare the performance and power of

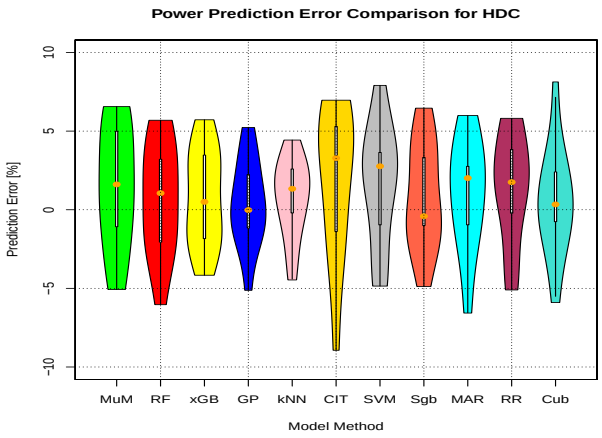

Figure 26. Prediction error rates (node power) for HDC

FTLA and 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 models for runtime, node power, CPU power, and memory power, we identified the most significant performance counters for potential optimization efforts associated with the application characteristics and the target computer platforms, 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 results in more knowledge about the application's energy consumption on a given architecture.

When we compare the prediction accuracy using MuMMI with that using the 10 ML methods, we observe that MuMMI results in more accurate prediction in both performance and power. Since the 10 ML methods each have their own way of learning the relationship between the predictors and the target object and provide different variable importance, it is hard to identify which ML provides the most robust variable importance for potential improvements. To address this issue, we plan to utilize ensemble learning to combine these ML methods, with the aim of providing more accurate models and obtaining better variable importance needed for latent variable modeling. Performance and power modeling tools such as MuMMI can help in application optimizations for energy efficiency, power, or energy-aware job schedulers and system performance and power tuning. Moreover, the general methodology presented in this paper can be applied to large-scale scientific applications [59] and deep learning applications [61] on other parallel 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, and in part by NSF grants CCF-1801856 and 1618776. We acknowledge Argonne Leadership Computing Facility for use of Cray XC40 Theta and Blue Gene/Q Mira under the DOE INCITE project PEACES and ALCF project EE-ECP, and Sandia National Laboratories for use of Intel Haswell Shepard testbed.

References

[1] C. Anfinson and F. Luk, A linear algebraic model of algorithm-based fault tolerance, IEEE Trans. on Computers, 37(12): 1599–1604,1988.

[2] G. Aupy, A. Benoit, T. Herault, Y. Robert, and J. Dongarra, Optimal checkpointing period: Time vs energy. In the 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2013.

[3] P. Balaprakash, L. Bautista-Gomez, M. Bouguerra, S. M. Wild, F. Cappello and P. D. Hovland, Analysis of the tradeoffs between energy and run time for multilevel checkpointing. In the 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2014.

[4] L. Breiman, Random forests, Machine Learning, 45: 5–32, 2001.

[5] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, Algorithm-based fault tolerance applied to high performance computing, J. Parallel Distributed Computing, 69: 410–416, 2009.

[6] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra, Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy, ACM Trans. Parallel Computing, 1(2): 10:1–10:28, January 2015.

[7] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, FTI: High performance fault tolerance interface for hybrid systems. In SC2011, Seattle, WA, 2011.

[8] R. Bertran, Y. Sugawara, H. Jacobson, A. Buyuktosunoglu, and P. Bose, Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems, IBM Journal of Research and Development, 57(1): 4:1–4:17, 2013.

[9] caret package: https://topepo.github.io/caret/index.html, https://cran.r-project.org/web/packages/caret/.

[10] Z. Chen and J. Dongarra, Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In IPDPS'06.

[11] T. Chen and C. Guestrin, XGBoost: A Scalable Tree Boosting System. In KDD'16, August 13–17, 2016.

[12] T. Chen, T. He, M. Benesty, et al., Extreme Gradient Boosting, Package "xgboost", August 1, 2019.

[13] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, High Performance Linpack benchmark: A fault tolerant implementation without checkpointing. In the International Conference on Supercomputing (ICS'11), 2011.

[14] P. Du, A. Bouteiller, G.Bosilca, T. Herault, J. Dongarra, Algorithm-Based Fault Tolerance for Dense Matrix Factorization. In the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012.

[15] P. Du, P. Luszczek, S. Tomov, J. Dongarra, Soft error resilient QR factorization for hybrid system with GPGPU. In SC2011 Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, 2011.

[16] Y. Freund and R. Schapire, Experiments with a new boosting algorithm. In the thirteenth International Conference on Machine Learning, 1996.

[17] J. Friedman, Greedy function approximation: A gradient boosting machine, Annals of Statistics, 29(5): 1189–1232, 2001.

[18] B. Greenwell, B. Boehmke, and J. Cunningham, Generalized boosted regression models, Package "gbm", January 14, 2019.

[19] M. el Mehdi Diouri, O. Gluck, L. Lefevre, and F. Cappello, Energy considerations in checkpointing and fault tolerance protocols. In 2nd International Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), 2012.

[20] M. el Mehdi Diouri, O. Gluck, L. Lefevre, and F. Cappello, ECOFIT: A framework to estimate energy consumption of fault tolerance protocols for HPC applications. In 13th IEEE/ACM International Symp. Cluster, Cloud and Grid Computing, 2013.

[21] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, Combining partial redundancy and checkpointing for HPC. In ICDCS'12, 2012.

[22] K. Ferreira, R. Riesen, P. Bridges, D. Arnold, J. Stearley, J. Laros, R. Oldfield, K. Pedretti, and R. Brightwell, Evaluating the viability of process replication reliability for exascale systems. In SC2011, 2011.

[23] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, Detection and correction of silent data corruption for large-scale high performance computing. In SC2012, 2012.

[24] J. H. Friedman, Multivariate Adaptive Regression Splines, The Annals of Statistics, 19(1): 1–67, 1991.

[25] FTI: Fault Tolerance Interface, leobago.github.io/fti/.

[26] FTLA: Fault Tolerant Linear Algebra, http://icl.cs.utk.edu/ft-la/software/index.html (ftla-rSC13.tgz).

[27] D. Hakkarinen and Z. Chen, Algorithmic Cholesky factorization fault recovery. In the IEEE International Symposium on Parallel Distributed Processing (IPDPS'10), 2010.

[28] T. Hothorn, K. Hornik, C. Strobl, and A. Zeileis, A Laboratory for Recursive Partytioning, Package "party", March 5, 2020.

[29] A. Hoerl, Ridge regression: Biased estimation for nonorthogonal problems, Technometrics, 12(1):55–67, 1970.

[30] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations, IEEE Transactions on Computers, 33(6): 518–528, 1984.

[31] A. Karatzoglou, A. Smola, and K. Hornik, kernlab – An S4 Package for Kernel Methods in R, Journal of Statistical Software, 11(9): 1–20, 2004.

[32] M. Kuhn and K. Johnson, Applied Predictive Modeling, Springer, 2013.

[33] M. Kuhn, S. Weston, C. Keefer, N. Coulter, and R. Quinlan, Rule- and instance-based regression modeling, Package "Cubist", January 10, 2020. https://topepo.github.io/Cubist.

[34] J. H. Laros III, P. Pokorny and D. DeBonis, PowerInsight – A commodity power measurement capability. In International Green Computing Conference, 2013.

[35] A. Liaw and M. Wiener, Breiman and Cutler's random forests for classification and regression, Package "randomForest", March 25, 2018.

[36] I. Marincic, V. Vishwanath, and H. Hoffmann, PoLiMEr: An energy monitoring and power limiting interface for HPC applications. In SC2017 Workshop on Energy Efficient Supercomputing, 2017.

[37] S Martin, D Rush, M Kappel, M Sandstedt, and J Williams, Cray XC40 power monitoring andcControl for Knights Landing. In 2016 Cray User Group Conference, 2016.

[38] E. Meneses, O. Sarood, and L.V. Kale, Assessing energy efficiency of fault tolerance protocols for HPC systems. In 24th IEEE Intern. Symp. on Computer Architecture and High Performance Computing, 2012.

[39] S. Milborrow, Multivariate adaptive regression splines, package "earth", November 9, 2019.

[40] Mira, IBM Blue Gene/Q system, https://www.alcf.anl.gov/mira.

[41] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system. In SC2010, 2010.

[42] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott. Proactive fault tolerance for HPC with Xen virtualization. In ICS2006, June 2006.

[43] NVIDIA, NVML API Reference Manual, 2012.

[44] J. Plank, K. Li, and M. Puening, Diskless Checkpointing, IEEE Trans. on Parallel and Distributed Systems, 9(10):972–986, 1998.

[45] R. Quinlan, Learning with continuous classes. In the 5th Australian Joint Conference on Artificial Intelligence, 1992, pp. 343–348.

[46] R. Quinlan, Combining instance-based and model-based learning. In the Tenth International Conference on Machine Learning, 1993, pp. 236-243.

[47] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, Power-management architecture of the Intel microarchitecture code-named Sandy Bridge, IEEE Micro, 32(2): 20–27, 2012.

[48] Scalable checkpoint/restart project, https://computation.llnl.gov/project/scr/.

[49] ScaLAPACK: Scalable Linear Algebra PACKage, http://www.netlib.org/scalapack/.

[50] Shepard, Advanced Systems Technology Test Beds, Sandia National Laboratories, http://www.sandia.gov/asc/computational_systems/HAAPS.html.

[51] Theta, Cray XC40 system, https://www.alcf.anl.gov/theta.

[52] L. Tan, S. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen, A survey of power and energy efficient techniques for high performance numerical linear algebra operations, Parallel Computing 40: 559–573, 2014..

[53] V. Vapnik, Statistical Learning Theory, Wiley, New York, 1998.

[54] VeloC: Very Low Overhead transparent multilevel Checkpoint/restart, http://www.mcs.anl.gov/project/veloc-very-low-overhead-transparent-multilevel-checkpointrestart.

[55] violin package: https://cran.r-project.org/web/packages/vioplot/index.html, https://www.data-to-viz.com/graph/violin.html.

[56] S. Wallace, V. Vishwanath, S. Coghlan, J. Tramm, Z. Lan, and M. E. Papka, Application power profiling on Blue Gene/Q. In IEEE Conference on Cluster Computing, 2013.

[57] C. Williams and C. Rasmussen, Gaussian Processes for Regression. In D. S. Touretzky, M. V. Mozer, and Me E, Hasselmo (Eds.),,Advances in Neural Information Processing, Vol. 8, MIT Press, 1995. URL http://books.nips.cc/papers/files/nips08/0514.pdf.

[58] X. Wu, V. Taylor, C. Lively, H. Chang, B. Li, K. Cameron, D. Terpstra and S. Moore, MuMMI: Multiple Metrics Modeling Infrastructure. In Tools for High Performance Computing, Springer, 2014.

[59] X. Wu, V. Taylor, J. Cook, and P. Mucci, Using performance-power modeling to improve energy efficiency of HPC applications, IEEE Computer, 49(10): 20–29, Oct. 2016.

[60] X. Wu, V. Taylor, and Z. Lan, Evaluating runtime and power requirements of multilevel checkpointing MPI applications on four different parallel architectures: An empirical study. In 2018 Cray User Group Conference, Stockholm, Sweden, May 20-24, 2018.

[61] X. Wu, V. Taylor, J. M. Wozniak, R. Stevens, T. Brettin, and F. Xia, Performance, energy, and scalability analysis and improvement of parallel cancer deep learning CANDLE benchmarks. In the 48th International Conference on Parallel Processing, Kyoto, Japan, August 5–8, 2019.

[62] H. Zou and T. Hastie, Elastic-Net for Sparse Estimation and Sparse PCA, Package "elasticnet", August 31, 2018.