NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning

Gagandeep Singh[a,c] Juan Gómez-Luna[b] Giovanni Mariani[c] Geraldo F. Oliveira[b]
Stefano Corda[a,c] Sander Stuijk[a] Onur Mutlu[b] Henk Corporaal[a]
[a]Eindhoven University of Technology [b]ETH Zürich [c]IBM Research - Zurich

ABSTRACT

The cost of moving data between the memory/storage units and the compute units is a major contributor to the execution time and energy consumption of modern workloads in computing systems. A promising paradigm to alleviate this *data movement bottleneck* is near-memory computing (NMC), which consists of placing compute units close to the memory/storage units. There is substantial research effort that proposes NMC architectures and identifies workloads that can benefit from NMC. System architects typically use simulation techniques to evaluate the performance and energy consumption of their designs. However, simulation is extremely slow, imposing long times for design space exploration. In order to enable fast early-stage design space exploration of NMC architectures, we need high-level performance and energy models.

We present NAPEL, a high-level performance and energy estimation framework for NMC architectures. NAPEL leverages ensemble learning to develop a model that is based on microarchitectural parameters and application characteristics. NAPEL training uses a statistical technique, called design of experiments, to collect representative training data efficiently. NAPEL provides early design space exploration 220× faster than a state-of-the-art NMC simulator, on average, with error rates of to 8.5% and 11.6% for performance and energy estimations, respectively, compared to the NMC simulator. NAPEL is also capable of making accurate predictions for previously-unseen applications.

1 INTRODUCTION

Current computing systems are compute-centric: all computation takes place in the compute units while the memory/storage units are passive components. As a result, modern workloads (e.g., machine learning, graph processing, bioinformatics), spend a large fraction of execution time and energy on moving data between the memory/storage units and the compute units. A way to alleviate this *data movement bottleneck* [12, 27] is *near-memory computing* (NMC), which consists of placing processing elements closer to memory. NMC is enabled by new memory integration technologies such as 3-D stacked memory [22, 23, 29], where multiple DRAM layers are stacked in the same chip with a logic layer that can embed processing elements. Past works [1, 2, 4–7, 14, 15, 19] show that NMC architectures can be employed effectively for a wide range of applications, including graph processing, databases, neural networks, bioinformatics. However, a common challenge all such past works face is how to evaluate the performance and energy consumption of the NMC architectures for different workloads systematically and accurately in a reasonable amount of time [27, 34].

In the early design stage, system architects use simulation techniques (e.g., [2, 4, 7, 15, 20]) for architectural performance and energy evaluation. However, this approach is extremely slow, because a single simulation for a real-world application with a representative dataset typically takes hours or even days. Specifically, the speed of a cycle-accurate simulator is in the range of a few thousand instructions per second [33], which is orders of magnitude slower than native execution.

Our goal is to enable fast early-stage design space exploration of NMC architectures without having to rely on time-consuming simulations. To this end, we propose the *NMC Application performance and energy Prediction framework using Ensemble machine Learning* (NAPEL). The key idea is to use ensemble learning to build a model that, once trained for a fraction of programs on a number of architecture configurations, can predict the performance and energy consumption of *different* applications on the same NMC architecture. The ensemble learning mechanism we use is random forest (RF) [8]. NAPEL can make performance and energy predictions for an average application on a specific architecture 220× faster than using simulation. Previous ML-based approaches [9, 35, 37] perform extrapolations to predict, for example, the performance of a known application for a bigger dataset. In contrast, NAPEL can make predictions for *previously-unseen* applications, after being trained with data from applications that are different from the applications that we want to predict.

NAPEL still needs to run simulations to gather training data that is required to construct its predictive model. As discussed above, running simulations is very time-consuming if we apply a brute-force approach to run all the application-input configurations needed for training data. To alleviate this problem, we use a technique called *design of experiments* (DoE) [28] to extract representative data with a small number of experimental runs (between 11 and 31 for the evaluated applications). Specifically, we employ a DoE variant called *central composite design* (CCD), which allows us to explore the interactions and nonlinear effects between the application input parameters and the output response (i.e., performance and energy consumption).

In this paper, we make the following contributions:

(1) We propose NAPEL, a new, fast high-level performance and energy estimation framework for NMC architectures. NAPEL is the first such model to leverage ensemble learning techniques, specifically random forest, to quickly estimate the performance and energy consumption of previously-unseen applications in early stages of design space exploration for NMC architectures.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317867

(2) We reduce the simulation time needed to gather training data for NAPEL by employing a DoE technique [25], which selects a small number of application-input configurations that well represent the entire space of input configurations.
(3) We show that NAPEL can provide performance and energy estimates 220× faster than a state-of-the-art microarchitecture simulator with an average error rate of 8.5% (performance) and 11.6% (energy) compared to the simulator.
(4) We show that we can use NAPEL to accurately determine if, and by how much, executing a certain workload on a specific NMC architecture can improve performance and reduce energy consumption versus execution on a CPU.

2 NAPEL

NAPEL is a performance and energy estimation framework that targets the early stages of NMC system design. In this section, we describe the main components of the framework. First, we give an overview of NAPEL training and prediction (Section 2.1). Second, we describe the target NMC architecture we consider in this work (Section 2.2). Third, we explain the code-instrumentation process for the applications used to generate training datasets and for the applications under performance and energy prediction (Section 2.3). Fourth, we describe the two most important components of NAPEL training: the design of experiments methodology (Section 2.4) and the ensemble machine learning (ML) technique (Section 2.5).

2.1 Overview

NAPEL is based on ensemble learning. Thus, it needs to be trained before it can predict performance and energy consumption. Figure 1 depicts the key components of NAPEL training and prediction.

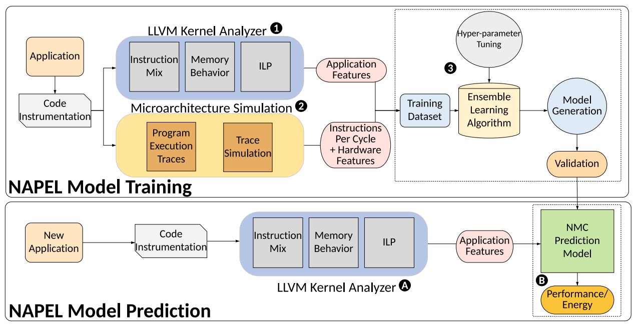

Figure 1: Overview of NAPEL training and prediction

Model Training. NAPEL training consists of three phases. The first phase (❶ in Figure 1) is an LLVM-based [21] kernel analysis phase (Section 2.3), which extracts architecture-independent workload characteristics. First, we instrument applications or parts of them that we use to gather data for model training. We consider the instrumented codes for execution on NMC compute units with a specific architecture configuration. Second, we characterize the instrumented codes in a microarchitecture-independent manner by using a specialized plugin of the LLVM compiler framework [3]. This type of characterization excludes any hardware dependence and captures the inherent characteristics of workloads.

In the second phase ❷, microarchitectural simulations are performed to gather architectural responses for training. For the simulations, we use central composite design (CCD) [25], a technique for the design of experiments (DoE) method [26]. With CCD, we can minimize the number of simulation experiments to gather training data for NAPEL while ensuring good quality of the training data (Section 2.4). The generated simulator responses along with application properties from the first phase and the microarchitectural parameters form the input to our ML algorithm.

In the third phase ❸, we train our ML algorithm (Section 2.5). During this phase, we perform additional tuning of our ML algorithm's hyper-parameters. Hyper-parameters are sets of ML algorithm variables that can be tuned to optimize the accuracy of the prediction model. We validate the prediction model against performance and energy simulation results from the second phase. Once trained, the framework can predict the performance and energy of a previously-unseen application on a specific NMC architecture.

Model Prediction. NAPEL prediction has only two phases. The first phase (Ⓐ in Figure 1) is the same LLVM-based kernel analysis phase as in NAPEL training. This phase extracts architecture-independent features of the workload for which NAPEL will predict the performance and energy consumption. The second phase Ⓑ performs the prediction by using the trained model. We feed the model with the architecture-independent workload features and the model provides the performance and energy estimations.

2.2 NMC Architecture

Figure 2 depicts the reference computing platform that we consider in this work. It contains a host processor and an external memory equipped with NMC compute units.

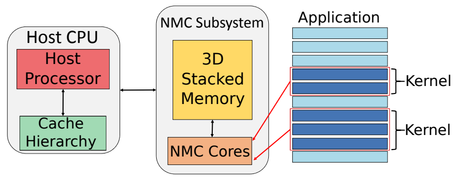

Figure 2: Overview of a system with NMC capability. On the right, an abstract view of application code with kernels that are offloaded to NMC

The NMC subsystem consists of a 3D-stacked memory [2, 22, 23, 29] with processing elements (PEs) embedded in its logic layer. The memory is divided into several vertical DRAM partitions, called vaults, each with its own DRAM controller in the logic layer. In this work, we model NMC PEs as in-order, single-issue cores with a private cache as proposed in previous work [2, 11], taking into account the limited thermal and area budget in the logic layer. NAPEL can be extended to support other types of general-purpose cores and accelerators by selecting the appropriate architectural features (see Table 1) for training the NAPEL model.

2.3 Code Instrumentation and Analysis

In the first phase of NAPEL training and prediction, the programmer annotates the region of the source code, called *kernel* (*k*), which is a candidate for offloading to NMC (i.e., execution on NMC processing elements). Then, that specific region is converted into an LLVM intermediate-representation (IR), which provides the basis for performing hardware-independent kernel analysis. Hardware-independent profiling enables us to generate an application profile *p* independently of the NMC design.

The application profile *p(k, d)* is obtained by executing the instrumented application kernel *k* while processing a dataset *d*. *p(k,* *d)* is a vector where each parameter is a statistic about an application feature *f*. Table 1 lists the main application features we extract by using the LLVM-based PISA analysis tool [3]. We select these features to analyze the memory access behavior of an application (data reuse distance, memory traffic, memory footprint, etc.), which is key for assessing the suitability of NMC for the application. Ultimately, the application profile *p* has 395 features, which includes all the sub-features of each metric we consider. Such a large number of features enables complex relationships to be identified between the analyzed application and its performance and energy consumption on the underlying NMC architecture [25].

Table 1: Main application and architectural features

Application Feature	Description
Instruction Mix	Fraction of instruction types (integer, floating point, memory read, memory write, etc.)
ILP	Instruction-level parallelism on an ideal machine.
Data/Instruction reuse distance	For a given distance δ, probability of reusing one data element/instruction (in a certain memory location) before accessing δ other unique data elements/instructions (in different memory locations).
Memory traffic	Percentage of memory reads/writes that need to access the main memory, assuming a cache of size equal to the maximum reuse distance.
Register traffic	Average number of registers per instruction.
Memory footprint	Total memory size used by the application.
NMC Arch. Features	**Description**
Core type	In-order
#PEs	Total number of near-memory processing units
Core frequency	Operating frequency of the core
Cache line size	Total size of a cache line (bytes)
#cache-lines	Number of cache lines
DRAM layers	Number of stacked DRAM layers
Size of DRAM	Total size of memory (bytes)
Cache access fraction	Cache hit ratio
DRAM access fraction	Cache miss ratio

2.4 Central Composite Design

In the second phase of NAPEL training, we use the design of experiments (DoE) method [26] as a way to minimize the number of experiments to train NAPEL without sacrificing the amount and quality of the information gathered by the experiments. DoE is a set of statistical techniques meant to locate a small set of points in a parameter space with the goal of representing the whole parameter space. The traditional brute-force approach to collecting training data is time-consuming: the sheer number of experiments renders detailed simulations intractable. Thus, the DoE strategy to gather a training dataset is a critical component of our model.

We apply the Box–Wilson central composite design (CCD) [25], the goal of which is to minimize the uncertainty of a nonlinear polynomial model that accounts for parameter interactions. While applying CCD, we treat the application input dataset *d* as a parameter vector (e.g., dataset size, number of threads, etc.) and each input configuration as a point in a multidimensional parameter space. For example, application *atax* from the PolyBench benchmark suite [31] has two significant parameters (*dimension*, *threads*) (see Table 2). In CCD, each input parameter in the vector *d* can have one of five levels: *minimum*, *low*, *central*, *high*, *maximum*. First, we select these levels for each parameter. For example, for *atax*, the levels of *dimension* are (500, 1250, 1500, 2000, 2300). Second, we place in the parameter space a point for each parameter combination (i.e., input configuration) with *low* and *high* levels (the corners of the solid-line square in Figure 3). In the case of *atax*, the points (*dimension*, *threads*) are (1250, 8), (1250, 32), (2000, 8), (2000, 32). Third, we draw a multidimensional sphere (represented as a circle in Figure 3) that circumscribes the initial square. This sphere generalizes the DoE to capture the nonlinearity in the system. Fourth, we obtain additional points on the sphere by combining the *central* level of each parameter with the *maximum* and *minimum* levels of the other parameters. For *atax*, these points (*dimension*, *threads*) are (1500, 4), (1500, 64), (500, 16), (2300, 16). Fifth, we include the *central* configuration, which is (1500, 16) for *atax*.

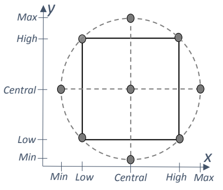

Figure 3: Central composite DoE for two parameters (*x*, *y*). For example, for *atax* (*x*, *y*) are (*dimension*, *threads*)

Table 2: Evaluated applications and their DoE parameters ("DoE param."). For each DoE parameter, we show its five levels (*minimum*, *low*, *central*, *high*, *maximum*) and *test* input

Application			DoE Parameter Levels					
Name	Description	DoE Param.	Min	Low	Central	High	Max	Test
atax	Matrix Transpose	Dimensions	500	1250	1500	2000	2300	8000
	and Vector Mult.	Threads	4	8	16	32	64	32
bfs	Breadth-first	Nodes	400k	800k	900k	1.2m	1.4m	1.0m
	Search	Weights	1	2	4	25	49	4
		Threads	1	9	16	32	64	32
		Iterations	30	40	65	70	80	95
bp	Back-propagation	Layer Size	800k	1m	2m	3.5m	4m	1.1m
		Seed	2	4	5	10	12	5
		Threads	4	8	16	32	64	32
		Iterations	1	3	9	16	25	9
chol	Cholesky	Dimensions	64	384	128	320	512	2000
	Decomposition	Threads	4	8	16	32	64	32
		Iterations	10	20	30	50	80	60
gemv	Vector Multiply	Dimensions	500	750	1250	2000	2250	8000
	and Matrix	Threads	4	8	16	32	64	32
	Addition	Iterations	50	60	80	100	150	60
gesu	Scalar, Vector, and	Dimensions	500	750	1250	2000	2250	8000
	Matrix Mult.	Threads	4	8	16	32	64	32
		Iterations	10	20	40	50	60	50
gram	Gram-Schmidt	Dimension$_i$	64	384	128	320	512	2000
	Process	Dimension$_j$	64	384	128	320	512	2000
		Threads	4	8	16	32	64	32
kme	K-Means	Data Size	100k	300k	700k	900k	1.2m	819k
	Clustering	Clusters	3	5	6	7	8	5
		Threads	1	9	1	32	64	32
		Iterations	10	20	30	40	50	30
lu	LU Decomposition	Dimensions	196	256	320	420	512	2000
		Threads	4	8	16	32	64	32
		Iterations	98	128	256	420	512	2000
mvt	Matrix Vector	Dimensions	500	750	1250	2000	2250	2000
	Product	Threads	4	8	16	32	64	32
		Iterations	10	20	30	50	60	40
syrk	Symmetric Rank-k	Dimension$_i$	64	128	320	512	640	2000
	Operations	Dimension$_j$	64	128	320	512	640	2000
		Threads	4	8	16	32	64	32
trmm	Triangular Matrix	Dimension$_i$	196	256	320	420	512	2000
	Multiply	Dimension$_j$	196	256	320	420	512	2000
		Threads	4	8	16	32	64	32

We run these DoE-selected application-input configurations on different architectural configurations to collect the training dataset. Table 2 lists the parameter levels for the evaluated applications. We include a *test* configuration, which we use in Section 3.4.

2.5 Ensemble Machine Learning

The third phase of NAPEL training is the training of the ML algorithm. As we retrieve hundreds of application features from the application analysis, we make use of the random forest (RF) [8] algorithm, which embeds automatic procedures to screen many input features. RF is an ensemble ML algorithm, which, starting from a root node, constructs a tree and iteratively grows the tree by associating it with a splitting value for an input variable to generate two child nodes. Each node is associated with a prediction of the target metric equal to the mean observed value in the training dataset for the input subspace the node represents. This input subspace is randomly sampled from the entire training dataset.

We employ RF to capture the intricacies of new NMC architectures by predicting instructions per cycle (IPC) when executing an application near memory. Formally, we predict IPC(p, a) ∼ IPC(k, d, a), where p is the hardware-independent application profile representation of kernel k when processing input dataset d on an architecture configuration a. The input data gathered to train our RF model has three parts: (1) a hardware-independent application profile *p(k, d)*, (2) an architectural design configuration a, and (3) responses corresponding to each pair (p, a). To gather the architectural responses, kernel k belonging to training set T with input dataset d is executed on an architectural simulator, simulating an architecture configuration a. This produces IPC*(k, d, a)* for that configuration and is used as a *label* while training our RF algorithm.

We improve NAPEL training by tuning the algorithm's hyper-parameters [30]. Hyper-parameter tuning can provide better performance estimates for some applications. First, we perform as many iterations of the cross-validation process as hyper-parameter combinations. Second, we compare all the generated models by evaluating them on the testing set, and select the best one.

After training our RF algorithm, we can predict the IPC of a kernel that is *not* in the training set. The predicted IPC can be used for performance evaluation of a kernel on an NMC system. The execution time Π_{NMC} of the kernel offloaded to NMC can be calculated as $\Pi_{\text{NMC}} = \frac{I_{\text{offload}}}{IPC \cdot f_{\text{core}}}$, where f_{core} is the frequency of the NMC processing cores and I_{offload} is the total number of offloaded instructions. Similarly, we build another model for energy prediction where we use energy consumption as a *label* when we train our RF algorithm.

3 EXPERIMENTAL RESULTS

3.1 Experimental Setup

We consider different workloads from the PolyBench [31] and Rodinia [10] benchmark suites that cover a wide range of domains, such as image processing, machine learning, graph processing, radio astronomy. First, we instrument the region of code that is considered for offloading to NMC processing elements. Second, we apply CCD to these workloads to select a small set of application input configurations that represent the space of possible input configurations. Third, we carry out the LLVM-based [21] microarchitecture-independent characterization to extract application metrics (Table 1) by using the PISA analysis tool [3].

We evaluate host performance on a real IBM POWER9 system [16] and NMC performance on a state-of-the-art simulator, Ramulator [20]. We extend Ramulator with a 3D-stacked memory model to simulate the NMC processing elements [32]. Table 3 summarizes the system details used for the host system and the NMC system. We collect dynamic execution traces of the instrumented code with a Pin tool. We feed the acquired traces to Ramulator. We use the simulation results as training data for our RF algorithm. Once trained, we use NAPEL to predict the performance and energy consumption of previously-unseen applications.

Table 3: System parameters and configuration

Host CPU System	
Configuration	IBM® POWER9 AC922 @2.3 GHz, 16 cores (4-way SMT), 32 KiB L1 cache, 256 KiB L2 cache, 10 MiB L3 cache, 32GiB RDIMM DDR4 2666 MHz
NMC System	
Cores	32× single issue, in-order execution @ 1.25 GHz
L1-I/D	2-way, cache size = 2 cache lines, 64B per cache line
DRAM Module	32 vaults, 8 stacked-layers, 256B row buffer; 4GB total size; closed-row policy
Off-chip Link	16-bit full duplex high-speed serializer/deserializer (SerDes) I/O link @ 15 Gbps [29]

3.2 Model Training and Prediction Time

Table 4 shows the time for performing training simulations (see "DoE run (mins)") with the selected DoE configurations ("#DoE conf.") to gather training data. The table also includes the time for training and tuning ("Train+Tune (mins)") and the prediction time ("Pred. (mins)") for each application. Once the model is trained, the DoE simulation time is amortized every time we predict performance and energy consumption for a previously unseen application. Thus, quick exploration and large prediction time savings compared to simulation are possible for a previously unseen application.

Table 4: Number of DoE configurations ("#DoE conf") for gathering training data ("DoE run (mins)"), NAPEL training time ("Train+Tune (mins)"), including tuning, and NAPEL prediction time ("Pred. (mins)")

Application	**Training/Prediction Time**			
Name	#DoE conf.	DoE run (mins)	Train+Tune (mins)	Pred. (mins)
atax	11	522	34.9	0.49
bfs	31	1084	34.2	0.48
bp	31	1073	43.8	0.47
chol	19	741	34.9	0.49
gemv	19	741	24.4	0.51
gesu	19	731	36.1	0.51
gram	19	773	36.5	0.52
kme	31	742	36.9	0.55
lu	19	633	37.9	0.51
mvt	19	955	38.0	0.54
syrk	19	928	35.7	0.51
trmm	19	898	37.6	0.48

For all the evaluated applications, we compare the prediction time using trained NAPEL models with the prediction time using Ramulator simulations. Figure 4 shows NAPEL's prediction speedup over Ramulator for 256 DoE configurations for all the evaluated workloads. We observe that NAPEL is, on average, 220× (min. 33×, max. 1039×) faster than simulation.

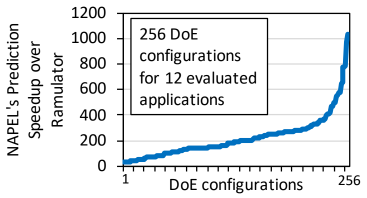

Figure 4: NAPEL's prediction speedup (in increasing order) over Ramulator for 256 DoE configurations

3.3 Accuracy Analysis

We analyze the accuracy of NAPEL for previously unseen applications by performing cross-validation [30]. To evaluate the prediction accuracy for a particular application, our training data comprises all the collected data (using an LLVM kernel analyzer and a microarchitecture simulator) for all applications *except* the application for which the prediction will be made. We repeat the same process to gather test prediction results for all applications, yet every time we test for a particular application, we do *not* include it in the training set. Therefore, when predicting performance and energy consumption of an application on NMC, we do *not* use any data related to that application. This makes the prediction more difficult because the ML algorithm has no knowledge of the application to be predicted. Thus, the test set differs from the training set as much as applications differ from each other. We evaluate the accuracy of the proposed model in terms of relative error ϵ_i to indicate how close the predicted value y'_i is to the actual value y_i. We calculate the mean relative error (MRE) for each application with Equation 1.

$$MRE = \frac{1}{N}\sum_{i=1}^{N}\epsilon_i = \frac{1}{N}\sum_{i=1}^{N}\frac{|y'_i - y_i|}{y_i} \quad (1)$$

Figure 5 shows NAPEL's MRE for the workloads in Table 2. NAPEL's average MRE is 8.5% for performance predictions and 11.6% for energy-consumption predictions. The highest error is for *bfs*, *bp*, and *kmeans* because these applications exhibit quite different characteristics compared to the other evaluated applications. In Figure 5, we also compare NAPEL with two other ML algorithms that can be used to predict performance and energy consumption: an artificial neural network (ANN) based on Ipek *et al.* [17] and a linear decision tree used by Guo *et al.* [13]. We make the following three observations. First, NAPEL is 1.7× (1.4×) and 3.2× (3.5×) more accurate in terms of performance (energy) prediction than the ANN and the linear decision tree, respectively. Second, the linear decision tree is very inaccurate, as shown by its high MRE. Decision trees are suitable mainly for linear regression, so they cannot capture the nonlinearity present in NMC performance and energy. Third, ANN is more accurate than the decision tree, but it is less accurate than NAPEL for almost all workloads. ANN requires a much larger training dataset to reach NAPEL's accuracy. When running these experiments, we also observe that the ANN takes more training time than NAPEL with hyper-parameter tuning (up to 5×).

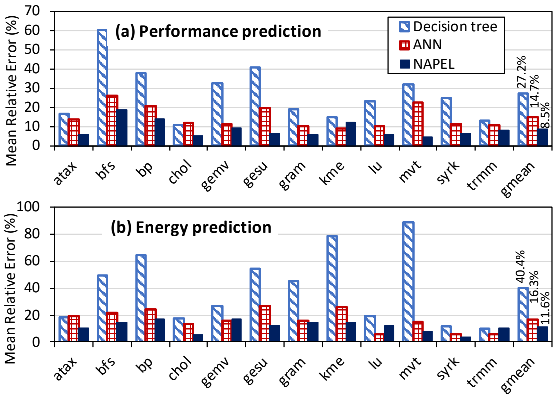

Figure 5: Mean relative error for performance (a) and energy (b) predictions using NAPEL vs. other methods

3.4 Use Case: NMC-Suitability Analysis

In this section, we use NAPEL to perform an NMC-suitability analysis, i.e., to assess the potential benefit of offloading a workload to NMC. This analysis compares the energy-delay product (EDP) of executing a workload on the NMC units, which we obtain from NAPEL's predicted NMC performance and energy consumption, to the measured EDP of executing the workload on a host processor. We use EDP as our major metric of reference in this analysis because both energy and performance are critical criteria for evaluating NMC suitability.

In order to obtain EDP results for the host system, we use a POWER9 system with 16 cores each supporting four-thread simultaneous multi-threading. We measure power consumption by monitoring built-in power sensors on our host system via the AMESTER[1] tool. Figure 6 shows the execution time and energy consumption of each workload on the POWER9. For the EDP results on the NMC system, we use NAPEL with tuned hyper-parameters.

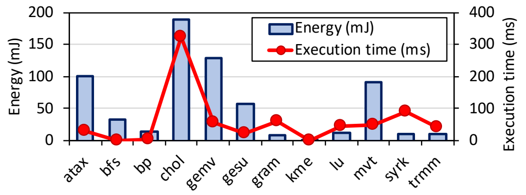

Figure 6: Execution time and energy on an IBM POWER9

Figure 7 shows estimated EDP reduction when executing each application on the NMC system compared to executing the same application on the host system using the *test* dataset (see Table 2). For each application, we show two bars: (1) NAPEL's estimated EDP reduction, and (2) the estimated EDP reduction obtained by simulating the application using the cycle-accurate Ramulator [20] ("Actual"). We make five observations. First, NAPEL estimates the same workloads to be NMC suitable as Ramulator does (i.e., workloads with EDP reduction greater than 1). Second, the MRE of NAPEL's EDP prediction is between 1.3% and 26.3% (14.1% on average). Third, *gemver*, *gesummv*, *lu*, *mvt*, *syrk*, and *trmm* are not suitable for NMC, since their EDP reduction is less than 1. These applications have enough data locality to leverage the host cache hierarchy. Fourth, *bfs*, *bp*, *cholesky*, *gramschmidt*, and *kmeans* are good fits for NMC. These applications are memory intensive and have irregular memory access patterns, so the host execution suffers from expensive offchip data movement. Fifth, *atax* benefits from the host cache hierarchy when performing vector multiplication, which has high data locality. However, it also performs matrix transposition, which is memory intensive. For *atax*-like workloads, the introduction of a small cache or scratchpad memory in the NMC compute units (larger than the 128B L1 cache in Table 3) can be beneficial, such that the data locality of of the application can still be exploited.

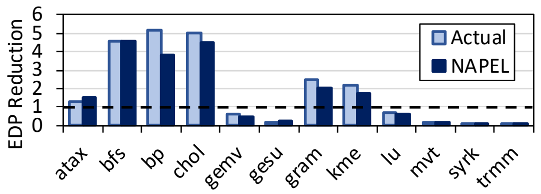

Figure 7: Estimated EDP reduction of offloading to NMC units versus execution on the baseline host CPU. "Actual" denotes the estimation with Ramulator. "NAPEL" denotes NAPEL's prediction results

[1]https://github.com/open-power/amester

4 RELATED WORK

The lack of evaluation tools is a critical challenge to the adoption of near-memory computing (NMC) [27, 34]. The importance of architecture simulators is widely acknowledged. However, simulators are generally very slow as they may take hours to simulate even a single configuration [13, 33].

Recent works propose ML-based performance prediction methods for faster early-stage design space exploration of different architectures. Table 5 lists recent works (including NAPEL) that use different prediction techniques for several architectures. Joseph *et al.* [18] and Guo *et al.* [13] use linear regression models to predict CPU performance. Linear models cannot accurately capture nonlinearity between application and processor responses, as shown in Figure 5. Wu *et al.* [36] use an ANN for GPU performance prediction without applying the DoE technique. Unlike NAPEL, this work uses traditional, time-consuming brute-force techniques to collect the training dataset. In the HPC domain, Mariani *et al.* [25] predict the performance of applications on cloud architectures using random forest and genetic algorithms, which are trained using DoE techniques. Ipek *et al.* [17] use an ANN with variance-based sampling for CPU performance prediction. Likewise, Li *et al.* [24] use an ANN with Latin hypercube sampling for design-space exploration of multicore CPUs. To our knowledge, NAPEL is the first performance and energy-prediction framework for NMC architectures that uses machine-learning models. NAPEL can make accurate predictions for previously unseen applications on NMC architectures.

Table 5: Related works in different domains

Name	Approach	Architecture	DoE
Joseph *et al.* [18]	Linear Regression	CPU	D-optimal Design
Ipek *et al.* [17]	ANN	CPU	Variance Based Sampling
Wu *et al.* [36]	ANN	GPU	None
Guo *et al.* [13]	Model Tree	CPU	None
Mariani *et al.* [25]	Random Forest, Genetic Algorithm	HPC	D-optimal Design, CCD
SemiBoost [24]	ANN	CPU	Latin Hypercube Sampling
NAPEL	**Random Forest**	**NMC**	**CCD**

5 CONCLUSION

We introduce NAPEL, the first high-level machine learning-based prediction framework for fast and accurate early-stage performance and energy-consumption estimation on NMC architectures. NAPEL avoids time-consuming simulations to predict the performance and energy consumption of previously unseen applications on various NMC architecture configurations. To achieve this, NAPEL relies on random forest, an ensemble learning technique, to build its prediction models.

NAPEL is 220× faster than a state-of-the-art NMC simulator, with an accuracy loss in performance (energy) prediction of only 8.5% (11.6%) compared to the simulator. Compared to an artificial neural network, NAPEL is 1.7× (1.4×) more accurate in performance (energy) prediction. NAPEL can accurately perform fast design-space exploration for different applications and NMC architectures. We hope the NAPEL approach enables faster development of NMC systems and inspires the development of other alternatives to simulation for NMC performance and energy estimation.

ACKNOWLEDGMENTS

This work was performed in the framework of the Horizon 2020 program for the project “Near-Memory Computing (NeMeCo)”. It is funded by the European Commission under Marie Sklodowska-Curie Innovative Training Networks European Industrial Doctorate (Project ID: 676240). We also thank SRC, Intel, VMware, Huawei, AliBaba, and Google for their funding support.

REFERENCES

[1] J. Ahn et al. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA 2015*.
[2] J. Ahn et al. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA 2015*.
[3] A. Anghel et al. An instrumentation approach for hardware-agnostic software characterization. *IJPP* (2016).
[4] E. Azarkhish et al. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *ARCS 2016*.
[5] A. Boroumand et al. CoNDA: Enabling efficient near-data accelerator communication by optimizing data movement. In *ISCA 2019*.
[6] A. Boroumand et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *ASPLOS 2018*.
[7] A. Boroumand et al. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *CAL* (2017).
[8] L. Breiman. Random forests. *Machine learning* (2001).
[9] A. Calotoiu et al. Using automated performance modeling to find scalability bugs in complex codes. In *SC 2013*.
[10] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC 2009*.
[11] M. Gao et al. Practical near-data processing for in-memory analytics frameworks. In *PACT 2015*.
[12] S. Ghose et al. The processing-in-memory paradigm: Mechanisms to enable adoption. In *Beyond-CMOS Technologies for Next Generation Computer Design (2019)*.
[13] Q. Guo et al. Microarchitectural design space exploration made fast. *MicPro* (2013).
[14] K. Hsieh et al. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *ICCD 2016*.
[15] K. Hsieh et al. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *ISCA 2016*.
[16] IBM. IBM POWER9 CPU. *URL: https://www.ibm.com/it-infrastructure/power/power9* (2018).
[17] E. Ipek et al. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS 2006*.
[18] P. J. Joseph et al. Construction and use of linear regression models for processor performance analysis. In *HPCA 2006*.
[19] J. S. Kim et al. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics* (2018).
[20] Y. Kim et al. Ramulator: A fast and extensible DRAM simulator. *CAL* (2016).
[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*.
[22] D. Lee et al. Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost. *ACM TACO* (2016).
[23] D. U. Lee et al. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *ISSCC 2014*.
[24] D. Li et al. Processor design space exploration via statistical sampling and semi-supervised ensemble learning. *IEEE Access* (2018).
[25] G. Mariani et al. Predicting cloud performance for HPC applications: a user-oriented approach. In *CCGrid 2017*.
[26] D. C. Montgomery. Design and analysis of experiments. (2017).
[27] O. Mutlu et al. Processing data where it makes sense: Enabling in-memory computation. *MicPro* (2019).
[28] M. Natrella. NIST/SEMATECH e-handbook of statistical methods. (2010).
[29] J. T. Pawlowski. Hybrid memory cube (HMC). In *HCS 2011*.
[30] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *JMLR* (2011).
[31] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* (2012).
[32] SAFARI Research Group. Ramulator for processing-in-memory. https://github.com/CMU-SAFARI/ramulator-pim/.
[33] D. Sanchez and C. Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA 2013*.
[34] G. Singh et al. A Review of near-memory computing architectures: Opportunities and challenges. In *DSD 2018*.
[35] A. Wong et al. Parallel application signature for performance analysis and prediction. *IEEE TPDS* (2015).
[36] G. Wu et al. GPGPU performance and power estimation using machine learning. In *HPCA 2015*.
[37] X. Wu and F. Mueller. Scalaextrap: Trace-based communication extrapolation for SPMD programs. In *PPoPP 2011*.