NeurOPar, A Neural Network-driven EDP Optimization Strategy for Parallel Workloads

Cristiano A. Kunas[§], Fábio D. Rossi[†], Marcelo C. Luizelli[*], Rodrigo N. Calheiros[‡], Philippe O. A. Navaux[§], and Arthur F. Lorenzon[§]
[*]Campus Alegrete, Federal University of Pampa, Brazil
[†]Campus Alegrete, Federal Institute Farroupilha, Brazil
[§]Western Sydney University, Australia
[§]Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
fabio.rossi@iffarroupilha.edu.br, marceloluizelli@unipampa.edu.br, R.Calheiros@westernsydney.edu.au
{*cakunas, navaux,aflorenzon*}*@inf.ufrgs.br*

***Abstract*—The pursuit of energy efficiency has been driving the development of techniques to optimize hardware resource usage in high-performance computing (HPC) servers. On multicore architectures, thread-level parallelism (TLP) exploitation, dynamic voltage and frequency scaling (DVFS), and uncore frequency scaling (UFS) are three popular methods applied to improve the trade-off between performance and energy consumption, represented by the energy-delay product (EDP). However, the complexity of selecting the optimal configuration (TLP degree, DVFS, and UFS) for each application poses a challenge to software developers and end-users due to the massive number of possible configurations. To tackle this challenge, we propose *NeurOPar*, an optimization strategy for parallel workloads driven by an artificial neural network (ANN). It uses representative hardware and software metrics to build and train an ANN model that predicts combinations of thread count and core/uncore frequency levels that provide optimal EDP results. Through experiments on four multicore processors using twenty-five applications, we demonstrate that *NeurOPar* predicts combinations that yield EDP values close to the best ones achieved by an exhaustive search and improve the overall EDP by 42% compared to the default execution of HPC applications. We also show that *NeurOPar* can enhance the execution of parallel applications without incurring the performance and energy penalties associated with online methods by comparing it with two state-of-the-art strategies.**

***Index Terms*—Parallel Computing, Artificial Neural-Network, Performance-Energy Optimization**

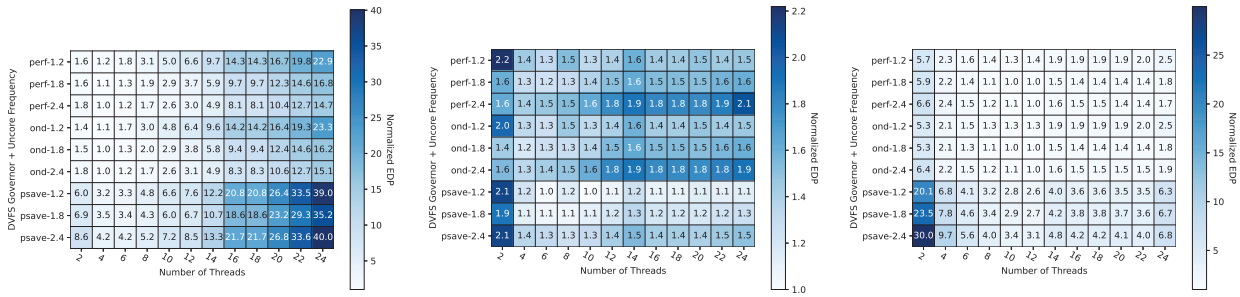
I. Introduction

Modern high-performance computing (HPC) systems have become essential for executing data-intensive parallel workloads, such as machine learning, biomedical, and video/audio recognition. Simultaneously, power consumption has emerged as a critical concern and a constraint for building exascale computing infrastructures since it is directly proportional to the increasing availability of data to compute. Therefore, instead of only focusing on maximizing performance, hardware designers and software developers have started offering tools or techniques through which users can improve performance while saving energy [1].

The inherent limits in parallel applications and systems are critical factors that surface the need for such techniques. When thread-level parallelism (TLP) is exploited, hardware and software-related aspects may prevent linear performance and energy improvements as active threads increase [2]–[4]. They include data synchronization, shared memory contention, and off-chip bus saturation, as we discuss in Section II. It means that exposing as much parallelism as possible and using the maximum number of hardware resources available in the system will not always deliver the best outcome in performance and energy. In these scenarios, techniques that select the ideal TLP degree of parallel applications can be employed to reduce energy without jeopardizing their performance.

When exploiting TLP, the operating frequency of the core and uncore parts of a chip/package plays an essential role in the performance and power consumption. While the core subsystem consists of the processing units and private caches (e.g., L1 and L2), the uncore part comprises shared components (e.g., last-level cache – LLC – and the quick path interconnect controllers). It occupies about 30% of the die area, accounting for up to 20% of the overall chip power consumption [5]. Hence, the same reasoning applied to TLP exploitation can be applied to core and uncore subsystems: using the maximum frequency level will not always result in the best energy efficiency. In this scenario, techniques may be applied to reduce the power consumption of the chip/package, such as dynamic voltage and frequency scaling (DVFS) and uncore frequency scaling (UFS). These mechanisms allow the component to have a min/max frequency and a governor that governs it to manage the operating frequency levels with a corresponding variation in the supply voltage.

Unfortunately, efficiently employing TLP exploitation, DVFS, and UFS to optimize performance and energy consumption is complex due to the number of variables involved. Besides the hardware and software issues affecting thread scalability, applications may present different CPU and memory usage behaviors, influencing the frequency levels. It means that the configuration (TLP degree, DVFS governor, and uncore frequency) that delivers the best outcome changes according to the application. We illustrate this scenario in Fig. 1 for the execution of three applications on an Intel Xeon Silver 24-core processor with a different number of threads, DVFS governors, and uncore frequencies. Each plot depicts the energy-delay product – EDP (i.e., the trade-off between performance and



(a) N-Body (b) STREAM (c) LULESH2.0

Fig. 1: EDP results of three applications running with different configurations of the number of threads $\times$ DVFS governor $\times$ Uncore frequency on an Intel 24-core system. Values are normalized to the best one, so the lower the value, the better.

energy) of all configurations normalized to the best one. Hence, the closer to 1.0, the better the EDP. As observed, the ideal configuration changes according to the application characteristics: *N-Body*, which has thread scalability limited by data synchronization, achieves the best EDP with a low TLP degree and higher core/uncore frequency; *STREAM*, where the off-chip bus saturation limits the performance, is better executed with a low core/uncore frequency and a medium TLP degree; and *LULESH*, a proxy application with performance limited by shared memory accesses, has the best outcome with a medium TLP degree and higher CPU/uncore frequency.

Therefore, reliance on end-users to make decisions regarding TLP, DVFS, and UFS can lead to a non-ideal use of hardware resources due to the huge design space exploration. As an example, let us consider the optimization of a parallel application on a system that supports *88* HW threads, with *3* DVFS governors available and *8* uncore frequency levels. An exhaustive search would need to test $2,112$ configurations ($88 \times 3 \times 8$) every time the input changes, being impractical. To deal with this challenging scenario, we explored the fact that numerous applications manifest comparable characteristics regarding their TLP degree, instructions per cycle (IPC), and inter-thread communication patterns (e.g., DRAM bandwidth utilization, LLC misses, and NUMA node accesses). Our hypothesis is that this information could be used to enhance the execution of parallel workloads without incurring the performance and energy penalties associated with online methods and the resource-intensive nature of offline strategies.

Given the scenario above, we propose *NeurOPar*, an optimization strategy for single-process multi-threaded workloads driven by an artificial neural network (ANN). It considers the hardware and software metrics associated with a specific application to predict configurations encompassing the number of threads, DVFS governor, and uncore frequency that yield the best performance and energy consumption outcome. To achieve this, *NeurOPar* is divided into two distinct phases. First, a *Training Phase* involves constructing and training an ANN model, which utilizes a comprehensive set of representative metrics and applications exhibiting diverse behavior patterns. Subsequently, in the *Execution Phase*, *NeurOPar* employs the trained ANN model, alongside the hardware metrics gathered from a subset of the target parallel workload, to predict the most suitable configuration for executing it.

With the generated ANN model in the training phase, *NeurOPar* automatically predicts a configuration of TLP degree, DVFS governor, and uncore frequency level for each parallel application that optimizes the EDP. We validate *NeurOPar* by executing a list of twenty-five well-known applications from different domains on four multicore processors. When considering the entire benchmark set and processors, *NeurOPar* improves the EDP by 43% compared to the default execution of parallel applications (max number of threads and max levels of core/uncore frequency). When comparing to the results found by an exhaustive search, we show that more than 84% of the predicted solutions of *NeurOPar* are at most in the top-10 configurations. Moreover, we show that *NeurOPar* can enhance the execution of parallel applications without incurring the overhead associated with online methods by comparing it with two state-of-the-art online strategies.

II. Background

A. Dynamically Management of Core and Uncore Frequencies

In modern processors, optimizing core and uncore frequency is vital in maximizing performance and energy efficiency. Dynamic voltage and frequency scaling (DVFS) is a processor feature that enables software to adjust a core's clock frequency at runtime without a reset [6]. The core subsystem consists of computing units and private caches per core (e.g., L1 and L2). In this scenario, DVFS aims to dynamically scale the CPU's supply voltage for a given frequency, ensuring it operates at the minimum speed required for the specific task [6]. This approach significantly reduces chip/package power consumption due to the quadratic relationship between voltage and dynamic power. To simplify DVFS management for software developers, Operating Systems provide frameworks that allow each core to have a min/max frequency and a governor for control. Governors are kernel modules that control core frequency/voltage operating points. Common governors include *powersave*, which sets the CPU frequency at the minimum allowable; *performance*, which fixes the CPU frequency at the maximum; and *ondemand*, where the frequency changes based on workload behavior.

Uncore subsystem power optimization is also vital for HPC workloads, accounting for ≈20% of overall package power consumption and ≈30% of die area [5]. It comprises shared components between cores, such as the last-level cache, quick-path interconnect controller, and integrated memory controller. By independently managing uncore elements (introduced in Intel Haswell-EP), one can configure different frequency levels for the core and uncore subsystems according to the workload demands. In this scenario, uncore frequency scaling techniques are used to select the frequency automatically.

B. Scalability of Parallel Applications

Many works have shown that running some kinds of parallel applications with all the available resources (e.g., number of cores and cache memories) will not necessarily lead to the best outcome in performance and energy consumption due to hardware and software issues that prevent linear improvements with the number of active threads [2], [3]. Applications that work with large volumes of private data stored in main memory and require frequent data retrieval face scalability challenges due to the saturation of the off-chip bus [2]. In this scenario, as thread count rises, demand for the bus increases linearly, but its bandwidth is limited by I/O pins [7] and does not scale with active threads. Thus, additional thread count, rather than yielding performance gains, only increases energy consumption when the bus saturates.

Similarly, when threads must read shared data, the amount of access to shared memory addresses influences performance and energy consumption as the number of threads increases. Since threads communicate by accessing shared data in areas that are usually more distant from the core (e.g., last-level cache and main memory) and have higher latency and power consumption per access than private caches, this communication potentially leads to bottlenecks, affecting the parallel application performance and energy consumption [3]. There are also scenarios where different threads must communicate by accessing the same variable addresses. In this case, synchronization directives are employed to avoid race conditions between threads and ensure the correctness of the result. However, when multiple threads reach a synchronization point, only one thread executes it at a time, which means that part of the application is serialized. Hence, with more threads, more serialization occurs within critical sections, which impacts the overall execution time and energy consumption [2].

Moreover, as threads experience non-uniform memory access times in NUMA architectures, uneven workload distribution across threads and the thread/data placement policy can lead to load imbalance. In this scenario, some threads may complete their tasks quickly while others remain busy, resulting in idle resources and reduced overall performance. This imbalance becomes more pronounced as the number of threads increases, limiting the potential for linear performance and power consumption improvements. In summary, these hardware and software factors highlight the importance of careful thread optimization and mitigation strategies to address performance and energy limitations in parallel applications.

TABLE I: Metrics retired from Intel VTune Profiler to analyze the behavior of parallel applications

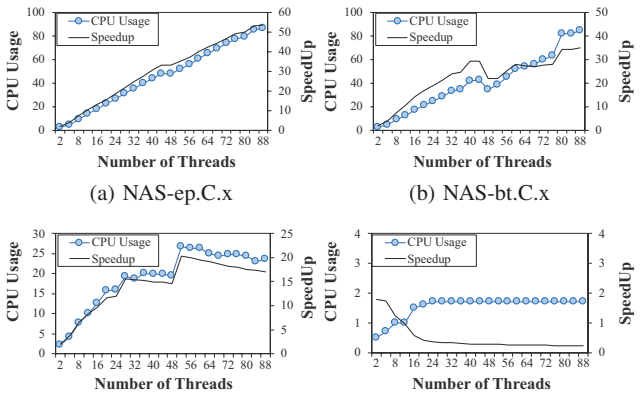
Metric	Description
Physical core utilization	Measures the parallel efficiency of the application by determining the proportion of physical CPU cores that the application utilizes.
Logical core utilization	The same as the previous, but considering logical CPU cores.
CPI Rate	Indicates how much time each executed instruction took.
Cache Bound	Shows how often the machine was stalled on L1, L2, and L3 caches.
Memory Bound	Indicates how the memory subsystem issues affect the performance.
DRAM Bound	Shows how often the CPU was stalled on the main memory
Memory Bandwidth	Represents a fraction of cycles during which the application could be stalled due to approaching bandwidth limits of the main memory.
Memory Latency	Represents a fraction of cycles during which an application could be stalled due to the latency of the main memory.
NUMA: % of Remote Accesses	In NUMA machines, it shows the percentage of remote accesses.

C. Influence of Hardware Metrics on the Thread Scalability and Core/Uncore Frequency Levels

Modern multicore processors provide profiling tools (e.g., *Intel VTune* and *AMD uProf*) so users can analyze the application and get insights on how to optimize it. In this scenario, to identify the metrics that play an essential role when selecting ideal levels of thread scalability and core/uncore frequency, we have executed applications from our benchmark set (discussed in Section IV) with different combinations of the *number of threads* $\times$ *core frequency* $\times$ *uncore frequency* in two Intel multicore machines. For each execution, we collected the metrics depicted in Table I through the *Intel VTune profiler*.

We start by discussing the metrics that provide insights w.r.t. the TLP degree of the application: *Physical and Logical core utilization*. Fig. 2 illustrates the relationship between performance improvements over the sequential execution (secondary *y-axis*) and the percentage of CPU utilization (*primary y-axis*) for four parallel applications with different TLP behavior. These applications range from *ep.C.x*, which exhibits the highest available TLP degree, to *MRI*, which has the lowest opportunity for TLP exploitation. As observed, the performance improvements are linearly influenced by the percentage of CPU utilization. To analyze how the metric correlates with the TLP degree, we used the Pearson Correlation [8]. It considers a range of $[-1, \cdots, +1]$, where the stronger the correlation, the closer the value will be to $+1$ or -1. We found values of $r = 0.93$, $r = 0.99$, $r = 0.99$, and $r = -0.99$ for *ep.C.x*, *bt.C.x*, *Heartwall*, and *MRI*, respectively. Hence, optimization frameworks can leverage such information to guide the decisions when defining the best number of threads. Furthermore, we found that the other metrics depicted in Table I presented r values lower than $|0.35|$, representing a small strength of linear association with the TLP degree.

When considering the core and uncore frequency levels, the metrics related to memory and CPU usage are helpful when finding optimal combinations of frequency levels. To illustrate



(a) NAS-ep.C.x (b) NAS-bt.C.x

(c) Rodinia-Heartwall (d) Parboil-MRI

Fig. 2: Performance improvements over sequential version (*SpeedUp*) and the respective CPU usage for different number of threads on a 2x44-core Intel processor

this scenario, Table II depicts the EDP behavior of three applications from our benchmark set with different CPU and memory usage. For *STREAM* and *FFT*, which are memory-intensive applications, the higher the uncore frequency, the better the CPI values and the lower the time spent in shared memories (e.g., NUMA % of remote accesses, LLC, and DRAM), resulting a better EDP relative to the execution with a minimal level of uncore frequency. Moreover, in these memory-intensive applications, the lower the core frequency, the better the EDP. On the other hand, for a CPU-intensive application (e.g., *N-Body*), the only outcome of increasing the uncore frequency is a negative impact on the EDP.

III. NeurOPar

NeurOPar aims to optimize the trade-off between performance and energy consumption of shared-memory parallel applications regardless of the API used to implement them (e.g., OpenMP, PThreads). For that, it exploits the intrinsic characteristics of these applications w.r.t. their scalability and CPU/Memory usage to predict the ideal combination of TLP degree and core and uncore frequency levels. *NeurOPar* is split into two phases: *training* and *execution*.

A. Training Phase

NeurOPar learns an ideal combination of TLP degree, core, and uncore frequency levels by evaluating parallel applications with distinct characteristics regarding their thread scalability and CPU/Memory usage. This particular phase is performed solely once during the entire optimization procedure and applies the steps illustrated in Fig. 3a.

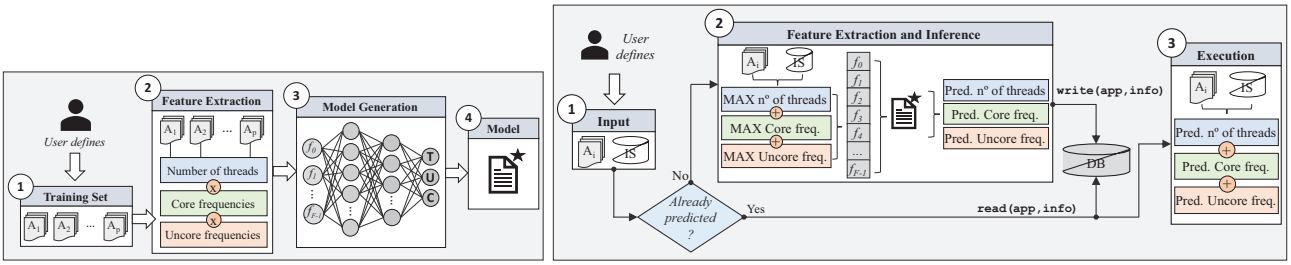
1) Defining the Training Set.: In this step, the user is responsible for providing a list of p application binaries (A_1, A_2, $\cdots$, A_p) with their dataset. They are used by *NeurOPar* to extract hardware metrics regarding their behavior (e.g., thread scalability, CPU, and memory usage). Furthermore, the metrics extracted from these applications are used to build and train the ANN model.

TABLE II: EDP behavior of three distinct applications when varying the uncore frequency level

Uncore Frequency	STREAM			FFT			N-Body		
	CPI	*MEM-Bound (%)*	*EDP Relative*	*CPI*	*NUMA (%) Accesses*	*EDP Relative*	*CPI*	*MEM-Bound (%)*	*EDP Relative*
1.2GHz	2.20	89.4	1.00	1.06	77.8	1.00	0.59	23.4	1.00
1.4GHz	2.00	86.1	0.89	0.99	70.0	0.90	0.58	23.5	1.01
1.6GHz	1.69	84.4	0.86	0.94	62.5	0.88	0.59	23.8	1.06
1.8GHz	1.54	80.2	0.80	0.91	62.5	0.90	0.59	23.4	1.19
2.0GHz	1.50	78.6	0.76	0.89	62.5	0.88	0.58	23.8	1.24
2.2GHz	1.45	76.1	0.75	0.88	61.8	0.87	0.59	23.4	1.31
2.4GHz	1.37	74.2	0.72	0.84	61.5	0.87	0.59	23.4	1.33
2.6GHz	1.34	71.1	0.72	0.83	61.2	0.87	0.58	23.7	1.42
2.8GHz	1.32	70.3	0.72	0.80	60.9	0.87	0.59	23.6	1.47
3.0GHz	1.27	69.6	0.70	0.79	60.1	0.86	0.59	23.6	1.57

2) Feature Extraction.: During this step, *NeurOPar* performs a design space exploration (DSE) on the target architecture considering the applications defined in the training set with the possible combinations of the number of threads, core (through DVFS governors), and uncore frequency levels. For each execution, *NeurOPar* collects and stores in a dataset (*i*) the execution time, energy consumption, and EDP to define the combination that delivers the best outcome; and (*ii*) the hardware and software metrics discussed in Section II-C. The metrics are extracted via Intel VTune Profiler on Intel processors and AMDuProf on AMD systems. To mitigate potential data-related issues that could introduce bias into the machine learning model, such as under- and over-fitting), *NeurOPar* applies the following operations over the dataset: *Discretization*, to ensure that all data are represented as numerical values. For example, each DVFS governor is represented as a unique identifier, such as powersave (*0*), ondemand (*1*), and performance (*2*); and *Normalization*, where the metric values of the dataset are normalized to a common scale in the range of $[0, 1]$ through the Min-Max normalization strategy.

3) Model Generation.: When building an ANN model, different parameter values may be used to control the learning process and significantly affect the performance of the ANN model. These parameters correspond to the (*i*) the number of hidden layers; (*ii*) the number of neurons in each layer; (*iii*) the activation function used; (*iv*) the learning rate; (*v*) the momentum; and (*vi*) the number of epochs. Hence, to optimize the selection of such parameter values, *NeurOPar* applies the *KerasTuner*, a scalable hyperparameter optimization framework that solves the points of hyperparameter search [9]. Once the parameter values are defined, the ANN model is trained using the dataset built in the last step as the input. To train the model, *NeurOPar* employs the Stratified *k*-Fold cross-validation, a variation of the standard *k*-Fold cross-validation technique designed to be effective in cases of a significant imbalance of the target value in the dataset. It splits the dataset on *k* folds such that each fold contains approximately the same percentage of samples of each target class as the complete set. *NeurOPar* performs 20 executions of this validation strategy and selects the model that presents the best accuracy to be used as the *Predictor*. Finally, the output of this step is a predictor model used in the *Execution phase* to infer the best combination of the number of threads and core and uncore frequency levels for any given parallel application.



(a) Training Phase

(b) Execution Phase

Fig. 3: NeurOPar Workflow

B. Execution Phase

Once the ANN model is trained for the target architecture (*Training Phase*), *NeurOPar* can use it to predict configurations of the number of threads, core, and uncore frequency levels that optimize the performance and energy consumption of parallel applications. For that, the workflow of this phase is illustrated in Fig. 3b and described below.

1) Input: The input of this phase, provided by the user, is the parallel application along with its dataset that the user wants to execute on the target architecture.

2) Feature Extraction and Inference: This stage is responsible for predicting a TLP degree and core/uncore frequency based on the characteristics of the application. For that, *NeurOPar* first reads the database to check if the application has been previously predicted by the ANN model. From this point, *NeurOPar* decides the execution flow based on the available information. When it is the first time the application is executed on the architecture, the feature extraction and inference stage starts: (*a*) the application executes with the default configuration (e.g., maximum number of hardware threads, and maximum levels of core/uncore frequency). (*b*) during the execution, *NeurOPar* extracts the same hardware metrics as the training phase (previously discussed). (*c*) *NeurOPar* pre-processes the data (discretization and normalization) to be input for the predictor model. (*d*) the Predictor is applied. The output of this stage is the predicted number of threads, core, and uncore frequency. This information is stored in the database along with the application's information so it can be retrieved the next time the application is deployed for execution. That is, if the application has already been predicted, *NeurOPar* recognizes it by checking the database (comparing the hash information of the application binary). From this point on, it gets the predicted configuration to execute such an application and configures the frequency levels and TLP degree for its execution.

3) Execution: In this step, the application is executed with the predicted combination of TLP degree, core, and uncore frequency levels.

C. Implementation

NeurOPar is implemented with the *Python3* language. To perform the training phase, the user only needs to provide the applications along with their dataset by calling the command `python3 neuropar.py train listApps`, where *train* means that *NeurOPar* will train a new model and *listApps* file contains the path to the application binaries. This phase is performed only once on the target architecture. From this moment on, the user can use the predicted model to optimize the execution of parallel applications implemented with distinct APIs (e.g., OpenMP and PThreads): `python3 meuropar.py execution App`, where *execution* indicates that *NeurOPar* will apply the predicted model on the application binary *App*. The EDP of each application execution during the training phase is obtained by multiplying the execution time (in seconds) by the energy consumption (in Joules). The execution time is obtained with the `time.time()` function from the **time** module from Python3. On the other hand, energy consumption is obtained directly from the hardware counters present in modern processors. In the case of Intel processors, the Running Average Power Limit (RAPL) library is used [10], while the Application Power Management library is used for AMD processors [11].

IV. Methodology

A. Benchmarks

We have considered forty parallel applications already written in C/C++ from assorted benchmark suites, split into two classes as discussed below: *training* and *validation*.

1) Training: Fifteen applications were used as the input for the *Training phase* to extract hardware and software metrics and train the ANN model of *NeurOPar*. *Eight* from the NAS-PB: *bt.c*, *bt.b*, *cg.b*, *ft.b*, *lu.b*, *mg.b*, *sp.b*, and *ua.b*. *Three* from the Rodinia Benchmark: *needleman-wunsch_large*, *streamcluster_small*, and *streamcluster_large*. *Two* from the Parboil: *Histo* and *stencil*. *Two* from different domains: *N-Body* and *FFT*. As we illustrate in Figures 4 and 5, the chosen applications to extract the hardware/software metrics cover different behaviors that impact thread scalability and operating frequency levels: (*i*) the TLP degree (as defined by the authors in [12] – the close this value is to 1.0 normalized to the total number of available cores, the more TLP is available), which varies from *histo* (lowest TLP available), where less than 5% of the execution is performed in parallel to the *bt.C* benchmark (highest TLP available); (*ii*) the percentage of time spent in memory accesses during the entire execution (MEM-Bound metric from Table II) and (*iii*) the average amount

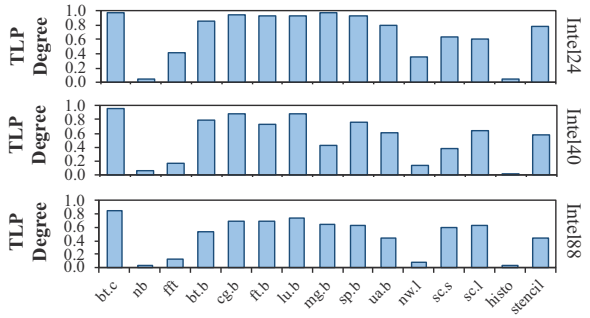


Fig. 4: TLP degree of each benchmark used for training

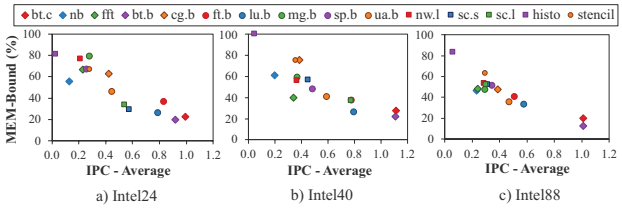


Fig. 5: Memory/IPC behavior of each bench. used for training

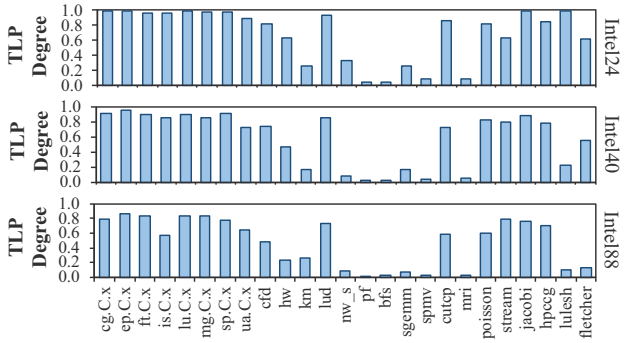


Fig. 6: TLP degree of each benchmark used for Validation

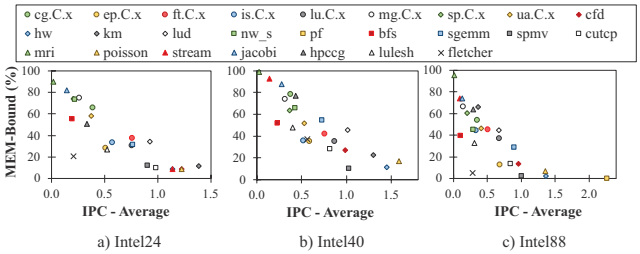


Fig. 7: Memory/IPC of each bench. used for validation

of IPC exhibited during the execution, showing whether the application is CPU- or Memory-bound.

2) Validation: Twenty-five applications with different characteristics were used to validate *NeurOPar*, as illustrated in Figures 6 and 7. *Eight* from the NAS-PB: *cg.c*, *ep.c*, *ft.c*, *is.c*, *lu.c*, *mg.c*, *sp.c*, and *ua.c*. *Six* from the Rodinia suite: *cfd*, *heartwall*, *kmeans*, *lud*, *Needleman-Wunsch_s*, and *pathfinder*. *Five* from the Parboil suite: *bfs*, *sgemm*, *spmv*, *cutcp*, and *mri*. *Four* from different domains: *Poisson*, *Stream*, *Jacobi*, and *HPCCG*. *Two* real-world applications: *LULESH2.0*, used by a variety of computer simulations of science and engineering problems. It is one of the five challenge problems implemented by the Lawrence Livermore National Laboratory (LLNL) in the DARPA-UHPC program and has been studied as a proxy application in the DoE co-design efforts to reach the exascale era [13]; and Fletcher Modeling, an application widely used in Oil and Gas companies that simulates the propagation of waves throughout time [14].

B. Execution Environment

To validate *NeurOPar*, we performed the experiments on four multicore platforms: *Intel24*, *Intel40*, *Intel88*, and *AMD64*, as shown in Table III. All the architectures used Linux Kernel v.4.19. We compiled the applications with GCC/G++ 10.2, using the *-O3* flag and OpenMP v.5. Although it is not possible to change the uncore frequency of the *AMD64* system, we want to evaluate how *NeurOPar* performs on such an environment. We compare the results achieved by *NeurOPar* with the following scenarios: *PAR_STD*: it executes each application with the maximum number of threads available and the core/uncore frequencies set to the maximum allowed, which is the standard behavior when running parallel applications on HPC servers. *Oracle*: each parallel application executes with the configuration (i.e., TLP degree, DVFS governor, and uncore frequency) that delivers the best outcome in EDP. The ideal configuration was obtained via an exhaustive execution of each application with different combinations of TLP degree (from 1 to the number of cores available in the architecture), DVFS governor, and uncore frequency.

Moreover, to observe if *NeurOPar* can enhance the execution of applications without incurring performance and energy penalties associated with online methods, we compare it to two online strategies: *EM-OMP*, an energy minimization model that adjusts the core voltage/frequency scaling and thread count based on workload predictions [15]. In these executions, the uncore frequency was set to be managed by the OS; and *Aurora*, an online heuristic that adapts the number of threads for each parallel region of OpenMP applications [16]. During the experiments, the core/uncore frequency was managed by the OS. It is worth mentioning that we selected these strategies because, as far as we know, there is no online method that tunes all three hardware knobs at the same time. The results shown next consider the average of ten executions.

TABLE III: Main characteristics of each processor

Model	Intel Xeon Silver 4214R	Intel Xeon E5-2650 v3	Intel Xeon E5-2699 v4	AMD Ryzen Threadr. 3990x
Microarch.	Cascade Lake	Haswell	Broadwell	Zen 2
#Cores (#Threads)	12 (24)	2x10 (40)	2x22 (88)	64 (128)
Core Freq	1.2 - 3.5 GHz	1.2 - 3.0 GHz	1.2 - 3.6 GHz	2.9 - 4.3 GHz
Uncore Freq	1.2 - 2.4 GHz	1.3 - 3.0 GHz	1.2 - 2.8 GHz	–
TDP	100W	105W/processor	145W/processor	280W
L3 Cache	16.5 MB	50 MB	110 MB	256 MB
RAM	64 GB	128 GB	256 GB	128 GB
Name	*Intel24*	*Intel40*	*Intel88*	*AMD64*

TABLE IV: Configurations found by the *Oracle* and *NeurOPar* on each multicore system

	Intel24					Intel40					Intel88				
	Oracle	***NeurOPar***	***Ranking***	***Exec. Over. NeurOPar***	***Time of Oracle***	***Oracle***	***NeurOPar***	***Ranking***	***Exec. Over. NeurOPar***	***Time of Oracle***	***Oracle***	***NeurOPar***	***Ranking***	***Exec. Over. NeurOPar***	***Time of Oracle***
cg.C.x	24,ond,1.8	24,ond,1.8	Top-1	27.6s	5889.4s	40,perf,3.0	40,perf,3.0	Top-1	19.2s	7632.4s	88,ond,2.4	88,perf,2.4	Top-5	11.7s	13156.4s
ep.C.x	24,perf,1.2	24,ond,1.8	Top-3	23.3s	10601.2s	40,perf,1.3	40,perf,2.2	Top-3	16.4s	10336.5s	88,perf,1.6	88,perf,2.0	Top-3	5.8s	11879.6s
ft.C.x	12,perf,1.2	24,perf,1.8	Top-15	23.6s	4654.6s	40,perf,2.6	40,perf,2.6	Top-1	15.7s	5535.2s	44,perf,2.8	40,perf,2.8	Top-3	9.3s	9600.8s
is.C.x	24,perf,1.2	24,ond,1.8	Top-5	2.1s	679.6s	40,perf,1.8	40,perf,2.2	Top-3	1.5s	763.7s	44,perf,2.4	40,perf,2.4	Top-3	1.4s	1293.7s
lu.C.x	24,ond,1.8	24,perf,1.2	Top-15	70.2s	17143.5s	40,perf,2.2	40,perf,2.2	Top-1	40.1s	16088.5s	84,ond,2.4	88,perf,2.0	Top-2	20.9s	24026.8s
mg.C.x	12,psave,1.2	8,ond,1.8	Top-15	79.2s	9753.9s	20,perf,2.6	20,perf,2.6	Top-1	46.5s	6537.5s	44,psave,2.0	36,psave,1.6	Top-20	43.3s	15362.8s
sp.C.x	4,ond,1.8	8,perf,1.8	Top-15	246.9s	25685.1s	12,perf,2.6	20,perf,2.6	Top-5	91.3s	17652.9s	20,perf,2.4	16,perf,2.4	Top-7	76.2s	37140.1s
ua.C.x	12,perf,1.2	12,ond,1.8	Top-8	105.8s	19210.5s	40,perf,2.2	40,perf,2.2	Top-1	50.2s	15624.2s	80,perf,2.8	40,perf,2.4	Top-10	28.8s	24685.8s
cfd	24,perf,1.8	24,perf,1.8	Top-1	26.8s	8440.1s	40,perf,2.2	40,perf,2.2	Top-1	20.5s	7720.4s	44,perf,2.4	88,perf,2.4	Top-2	12.8s	13076.3s
hw	20,perf,1.2	22,ond,1.8	Top-10	3.9s	443.6s	40,perf,1.8	40,perf,2.2	Top-7	3.7s	973.9s	52,perf,2.0	56,perf,2.4	Top-5	5.0s	2429.7s
km	24,ond,1.8	24,perf,2.4	Top-6	10.8s	1152.1s	40,perf,2.2	40,perf,2.2	Top-1	9.2s	1861.4s	88,ond,2.0	80,ond,2.0	Top-20	7.3s	3007.7s
lud	12,perf,1.2	12,perf,1.2	Top-1	37.6s	3395.6s	20,perf,2.2	20,perf,2.2	Top-1	11.8s	2468.4s	44,ond,1.6	40,perf,2.4	Top-14	19.4s	10740.8s
nw_s	8,ond,1.2	8,ond,1.2	Top-1	6.9s	1299.4s	24,perf,2.6	24,perf,2.6	Top-1	11.8s	1832.7s	12,perf,2.4	12,perf,2.4	Top-1	22.3s	6943.5s
pf	24,ond,1.2	6,ond,1.2	Top-5	17.7s	5367.9s	32,perf,1.3	32,perf,1.3	Top-1	13.0s	5503.8s	16,ond,1.2	16,ond,1.2	Top-1	7.3s	7728.3s
bfs	2,ond,1.2	2,ond,1.8	Top-2	1.2s	693.9s	2,perf,1.8	4,perf,3.0	Top-6	1.2s	481.3s	2,perf,2.4	2,perf,2.4	Top-1	0.9s	880.0s
sgemm	24,ond,1.2	24,ond,1.8	Top-3	1.6s	344.3s	20,perf,2.2	16,perf,3.0	Top-10	1.3s	320.0s	40,perf,2.0	20,perf,2.4	Top-15	1.6s	1320.2s
spmv	24,ond,1.2	24,ond,1.2	Top-1	4.8s	1048.7s	20,perf,1.3	32,perf,1.8	Top-15	24.7s	7656.1s	24,ond,1.6	44,perf,2.4	Top-5	16.2s	11379.5s
cutcp	24,perf,1.2	24,perf,1.2	Top-1	1.7s	240.2s	40,perf,1.8	40,perf,2.2	Top-2	1.7s	286.9s	88,perf,2.4	88,perf,2.4	Top-1	1.5s	681.5s
mri	2,ond,1.8	2,ond,1.8	Top-1	10.0s	1608.6s	2,perf,3.0	2,perf,3.0	Top-1	10.6s	1728.7s	2,perf,2.4	2,ond,1.6	Top-5	7.2s	4014.5s
poisson	24,perf,1.8	24,perf,1.8	Top-1	8.4s	1133.3s	20,perf,2.6	20,perf,2.6	Top-1	21.6s	2090.3s	44,perf,2.4	44,perf,2.4	Top-1	30.8s	8505.5s
stream	10,psave,1.2	4,ond,1.8	Top-10	4.3s	798.6s	20,perf,3	20,perf,3	Top-1	4.0s	716.9s	24,psave,2.0	16,perf,2.4	Top-10	2.0s	1124.3s
jacobi	10,psave,1.2	10,psave,1.8	Top-11	2.6s	456.3s	20,perf,2.2	20,perf,3.0	Top-3	2.7s	455.9s	20,psave,2.0	12,perf,2.0	Top-15	1.8s	988.5s
hpccg	6,ond,1.8	18,perf,1.8	Top-2	6.6s	2257.6s	36,perf,2.6	36,perf,3.0	Top-4	5.3s	1987.2s	24,perf,2.4	44,perf,2.8	Top-6	2.3s	2799.9s
lulesh	12,perf,1.8	12,perf,1.8	Top-1	143.1s	15642.8s	12,perf,3.0	12,perf,3.0	Top-1	186.2s	21313.4s	12,ond,2.4	12,perf,2.4	Top-2	320.5s	104607.0s
fletcher	24,perf,1.2	24,perf,1.2	Top-1	13.0s	3511.8s	36,perf,1.3	36,perf,2.2	Top-3	19.7s	6095.0s	44,perf,2.0	44,perf,2.4	Top-3	21.0s	13137.9s

V. Evaluation

A. Accuracy of NeurOPar

We start by presenting the ANN model built during the training phase of *NeurOPar* on each machine. During the *feature extraction* step, we considered the following configurations: (*i*) the number of threads ranges from $1, 2, 4, 6, ..., n$, where *n* is the number of hardware threads available in the architecture; (*ii*) the selected DVFS governors were *powersave*, *ondemand*, and *performance*; and (*iii*) the uncore frequency levels in the range from min to max in steps of $0.1GHz$, according to Table III. Therefore, to generate the data used to feed the ANN model (which is performed only once during the entire optimization process), *504* executions per application were performed on the **Intel24** (*7,560* in total); *1,197* on the **Intel40** (*17,955* in total), *2,430* on the **Intel88** (*36,450* in total), and *108* on the **AMD64**[1] (*1620* in total). Given that HPC servers have several identical machines, the feature extraction step may be distributed among computational nodes, reducing the time needed to collect metrics.

Given the data generated in the *feature extraction* step, the following ANN models were built in the *model generation* step on each machine. **Intel24**: ANN with four layers (input layer with *38* neurons, 2 hidden with *16/8* neurons, and *29* neurons in the output layer); the number of epochs = *300*; batch size = *16*; learning rate = *0.05*; and dropout rate = *0.4*. **Intel40**: ANN with five layers (input layer with *52* neurons, 3 hidden with *32/16/8* neurons each, and *43* neurons in the output layer); the number of epochs = *1000*; batch size = *32*; learning rate = *0.01*; and dropout rate = *0.4*. **Intel88**: ANN with five layers (input layer with *75* neurons, 3 hidden with *32/16/8* neurons each, and *66* neurons in the output layer); the number of epochs = *400*; batch size = *32*; learning rate = *0.05*; and dropout rate = *0.2*. **AMD64**: ANN with five layers (input layer with *56* neurons, 2 hidden with *32/16/8* neurons each, and *54* neurons in the output layer); the number of epochs = *300*; batch size = *16*; learning rate = *0.04*; and dropout rate = *0.2*.In the end, the best-trained model on each architecture after the *Model Generation* step achieved an accuracy of 91%, 93%, 83%, and 91% on the **Intel24**, **Intel40**, **Intel88**, and **AMD64**, respectively.

We depict in Table IV the combinations found by the exhaustive search performed by the *Oracle* solution and the ones predicted by *NeurOPar* for the twenty-five evaluated applications on each target architecture. Each combination is represented by <*number of threads*, *DVFS governor*, *uncore frequency level*>. The next column shows the index of the configuration predicted by *NeurOPar* in ranking the best solutions. Then, the column "*Exec. Over. NeurOPar*" indicates the execution overhead, in seconds, needed by *NeurOPar* to extract the features and predict the combination for each application when it is executed for the first time in the architecture. Finally, the last column for each processor depicts the total time taken by the exhaustive search performed by the *Oracle* to find the ideal combination for each application and architecture.

Let us first compare the accuracy of the predictions made by *NeurOPar* with the *Oracle* solutions. As depicted in the first two columns of Table IV for each processor, it predicted correctly the ideal combination of TLP degree, DVFS governor, and uncore frequency level in 40%, 56%, and 16% of the applications in the **Intel24**, **Intel40**, and **Intel88**, respectively. Even though the number of **exact** predictions is relatively low, it is essential to highlight the nature of the optimization problem and the space of possible combinations per application: In the exhaustive search, a total of *504* possible configurations were

[1]it is worth mentioning that for the AMD64 system, we were not able to dynamically change the uncore frequency dynamically

evaluated on the **Intel24**, *1,197* on the **Intel40**, and *36,450* on the **Intel88**. This scenario highlights the main challenge of finding the best configuration for each application. Therefore, predicting a configuration capable of delivering results close to the *Oracle* is essential when it cannot reach the best solution. With that, 70% of the predictions made by *NeurOPar* reached a solution within the *Top-5* (among the five best found by the exhaustive search), and 84% of the times within the Top-10.

When it comes to the EDP results, Fig. 9a compares the results of *NeurOPar* to the *Oracle* for each application and multicore processor. EDP is normalized to the *Oracle*, so the closer the values are to *1.0*, the closer the configuration predicted by *NeurOPar* to the best possible one. Because *NeurOPar* can predict configurations that are most of the time in the Top-10 best results, the overall EDP difference between it and *Oracle* for the entire benchmark set is only 13% on the **Intel24**, 4% on the **Intel40**, and 9% on the **Intel88**. Overall, the top-10% (90-percentile) configuration presented 31% of EDP gains over *PAR_STD*. In the top-85 and top-80 percentile, the EDP gains were 22% and 11% over *PAR_STD*, respectively. In specific cases (e.g., *ft.C.x* and *sp.C.x* on the **Intel24**), *NeurOPar* failed to deliver an EDP close to the one delivered by *Oracle*. The worst scenario happened for the *sp.C.x* execution on the **Intel24**, where the number of threads predicted was twice the number that delivered the best EDP (*8* instead of *4*, see Table IV). Since this application has its thread scalability limited by the overhead of shared memory communication and data-synchronization between threads [16], assigning more resources to it other than the optimal number of threads will only increase the execution time and energy consumption, and hence, the EDP, as highlighted in Figure 8. Moreover, there are cases in which *NeurOPar* predicted a core and/or uncore frequency different from the ideal (e.g., *ep.C.x* in all the processors). However, the impact on EDP is smaller than predicting a non-ideal number of threads since the TLP degree has been shown to have more impact on the behavior of the evaluated parallel applications.

B. NeurOPar vs. State-of-the-art Strategies

Figure 9 depicts the EDP results for the entire benchmark set with the geometric mean (*gmean*) considering the Intel three processors, while Fig. 10 shoes the results for the **AMD64** system. We compared *NeurOPar* to the standard execution of parallel applications in Fig. 9b (*PAR_STD*), while Figures 9c and 9d compare *NeurOPar* to the *EM-OMP* and *Aurora* strategies. Each bar represents the EDP achieved by *NeurOPar* normalized to the method to be compared, represented by the black line. Hence, values below the line (1.0) mean that *NeurOPar* achieved better EDP than the strategy.

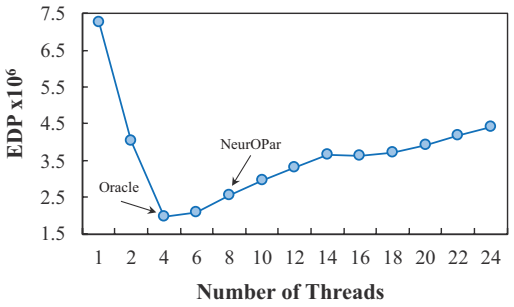
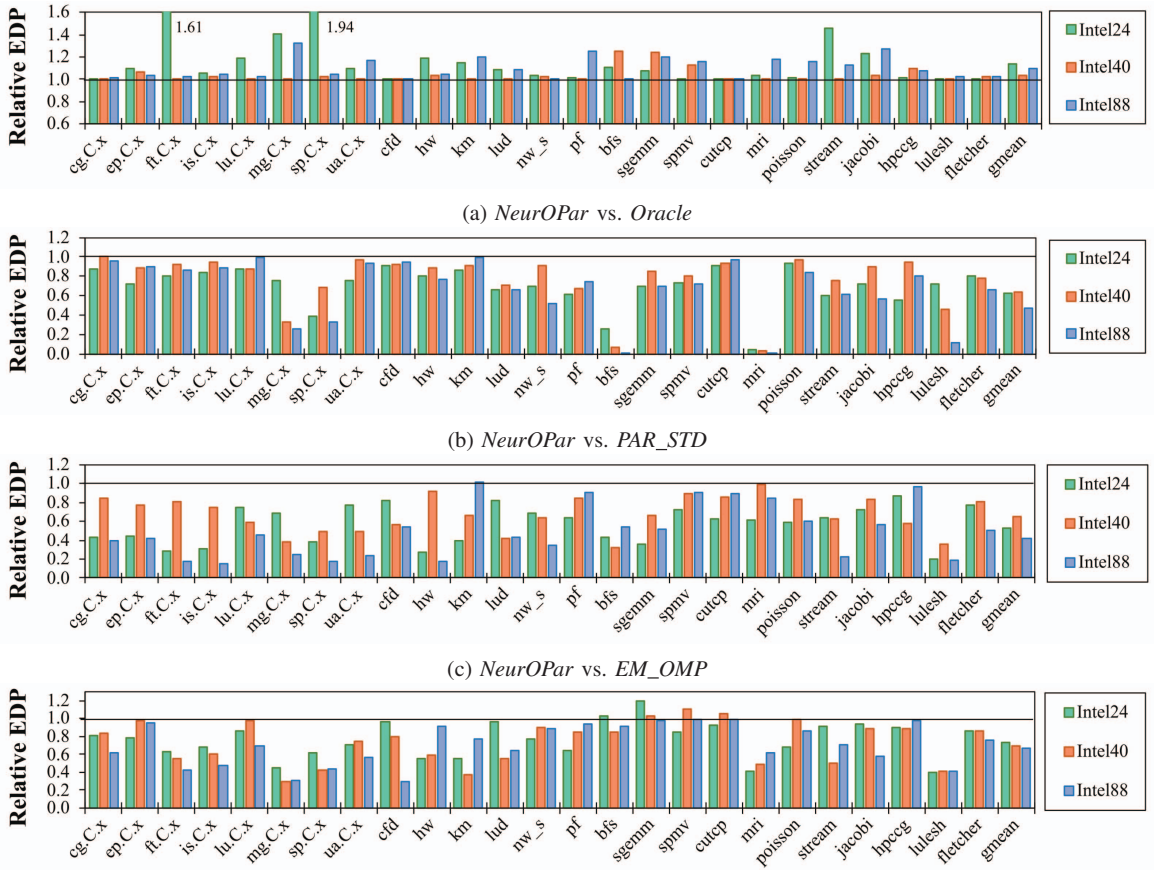


Fig. 8: EDP behavior of *sp.C* when varying the number of threads. DVFS governor and uncore frequency are set to the values that deliver the best result for each number of threads.

Let us first discuss the results on the Intel processors. Compared to *PAR_STD*, *NeurOPar* shows EDP improvements in most cases. The best scenario for *NeurOPar* is for the benchmarks that present the lowest TLP degree due to the existence of critical sections inside their parallel regions: *bfs* and *mri*. Hence, executing them with the maximum number of threads (as *PAR_STD* does) dramatically increases the EDP. Only in specific scenarios where the best configuration is equal to the one used by the *PAR_STD*, both strategies delivered similar EDP: *cg.C.x* on the **Intel24** and *lu.C.x* on the **Intel88**. When considering the overall geometric mean on each multicore system, *NeurOPar* provided 37.2%, 36.1%, and 53.2% of EDP improvements on the **Intel24**, **Intel40**, and **Intel88** machines, respectively.

When it comes to the online strategies (*EM-OMP* and *Aurora*), even though they can deliver better results than the *STD_PAR* for many applications due to the thread scalability and CPU/Memory behavior, the performance and energy penalties incurred due to the learning time at runtime limit the EDP improvements, therefore, *NeurOPar* achieves better EDP results than the evaluated online strategies in most cases, as illustrated in Figures 9c and 9d. Compared to the *EM-OMP*, on the overall of all benchmarks and processors, *NeurOPar* achieved an EDP 47.6% better than it. We have experimentally found that *EM-OMP* spends too much time converging to a solution because the learning algorithm starts with the TLP degree equal to the minimum value (e.g., 1) and increases it in steps of 1 at each iteration until its convergence. This, in turn, penalizes the performance and energy consumption of the applications, mainly the ones that present a high degree of TLP. Concerning the results achieved by *Aurora*, *NeurOPar* showed 30.1% of EDP improvements over it. The primary sources of these improvements come from applications that present a high TLP degree (e.g., *ft.C.x* and *is.C.x*). Even though *Aurora* employs a smarter learning algorithm than *EM-OMP* (a heuristic that evaluates the thread scalability and then applies a hill-climbing-based algorithm with lateral movements) was still not able to reach the EDP levels of *NeurOPar* in such applications. Furthermore, in specific applications, where the workload changes as the application executes, the nature of *Aurora* learning algorithm can adapt the execution environment accordingly and provide better EDP than *NeurOPar* (e.g., *sgemm*, *spmv*, and *cutcp*). We also show in Fig. 10 the results of all strategies on the *AMD64* system. As observed, *NeurOPar* also delivers better EDP results than all the evaluated strategies by only optimizing the thread count and core frequency.

A significant challenge in proposing energy efficiency optimization techniques for HPC systems is to avoid a significant degradation in the application's performance. Hence, to demonstrate the performance improvements achieved by using *NeurOPar* while optimizing EDP, we conducted a performance



(a) *NeurOPar* vs. *Oracle*

(b) *NeurOPar* vs. *PAR_STD*

(c) *NeurOPar* vs. *EM_OMP*

(d) *NeurOPar* vs. *Aurora*

Fig. 9: EDP Results of *NeurOPar* normalized to each strategy (represented by the black line) for the entire validation set with the *geomean* on each multicore system. Values below 1.0 mean that *NeurOPar* delivers better EDP

and energy comparison. We compared the results of *NeurOPar* with all previously discussed strategies and illustrate in Fig. 11 the execution time and energy consumption, considering the geometric mean of the entire benchmark set, for each multicore system and configuration evaluated (*Configs* – all configurations of the exhaustive search, *Oracle*, *PAR_STD*, *EM-OMP*, and *Aurora*).

Because *NeurOPar* predicts configurations that are most of the time in the *Top-10* best solutions (Table IV), it shows performance and energy improvements compared to the other strategies even targeting EDP only. On the geometric mean of benchmarks and architectures, *NeurOPar* improves the performance and energy by 17% and 33% over *PAR_STD*; by 18%; and 20% over *EM-OMP*, and by 10% and 15% over *Aurora*. Furthermore, the difference in performance and energy to the *Oracle* solution is only 2.3% and 7.4%, respectively.

C. Overhead of NeurOPar

The outcome of being able to predict a configuration that is most of the time at the Top-10 is an EDP result very close to the one achieved by the exhaustive search (*Oracle*) in most cases. However, as *NeurOPar* profiles the application with the default configuration and only extracts information from hardware and software during the first time it executes, the execution overhead of *NeurOPar* is significantly smaller than the exhaustive search for the entire benchmark set (Table IV): only 0.62%, 0.43%, and 0.20% of the time spent by the exhaustive search for the **Intel24**, **Intel40**, and **Intel88** machines, respectively. Furthermore, there is an implicit overhead of *NeurOPar* regarding the database and the time to access it. *NeurOPar* occupies *9.2 kB* of memory space; each hash containing the applications information adds *174 bytes*; the time to update the database with a new entry is *0.004s*; and the time for searching a combination and reading it is *0.008s*.

The highest computational cost of *NeurOPar* relates to the DSE to feed the ANN, which depends on the target processor: it took *18.2*, *18.9*, and *45.9* hours on the **Intel24**, **Intel40**, and **Intel88**, respectively. When considering the energy costs, it spent $3.06x10^6$ joules on **Intel24**, $6.62x10^6$ joules on **Intel40**, and $19.4x10^6$ joules on **Intel88**. The inference time is split into execution overhead (the time to get the features from the application, shown in Table IV) and inference, the time to run the predictor model. The inference time per application was: *0.0071s* on **Intel24**, *0.0072s* on **Intel40**, and 0.0075s

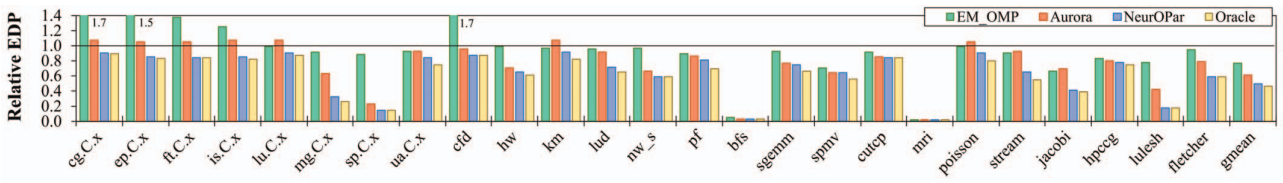
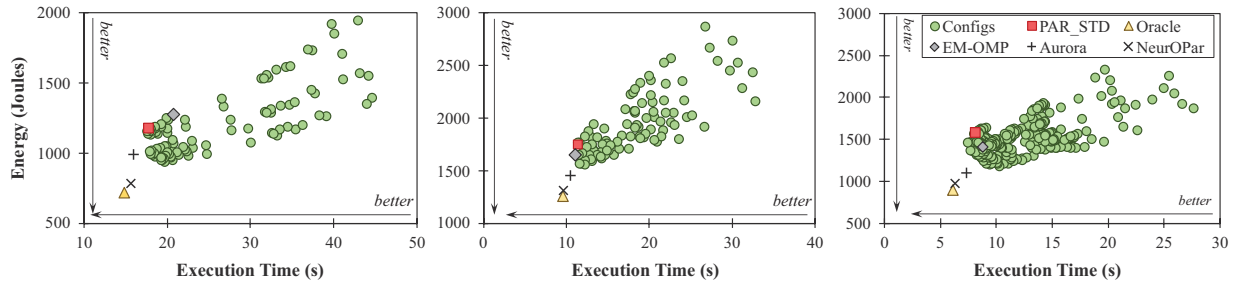


Fig. 10: EDP Results of each strategy normalized to *PAR_STD* (represented by the black line) on the AMD 64-Core system. Values below 1.0 mean better results.



a) Intel24 **b) Intel40** **c) Intel88**

Fig. 11: Energy consumption and execution time results for the evaluated strategies considering the geometric mean of all benchmarks. To provide a better view, we cropped the plots on the *x* and *y* axes, which excluded the worst results

on **Intel88**. However, it is worth highlighting that the DSE is only performed once to extract the metrics for the ANN model. Moreover, the DSE cost can be reduced (which is not the focus of this paper) by employing distinct strategies, such as sampling, reduction in the input set of the application, or distributing the DSE across identical machines.

VI. Related Work

Several works have been proposed to optimize parallel applications' performance levels and energy consumption by adjusting just one variable (e.g., either the TLP degree, DVFS governor, or uncore frequency). Works that employ some heuristic for searching for an ideal number of running threads include Feedback-Driven Threading (FDT - [2]), MAESTRO [17], Varuna [18], and Aurora [16]. Approaches that aim at selecting optimal CPU frequency levels by considering workload characteristics include SERAS [19], VDP [20], and PL-DVFS [21]. With the availability of drivers on the latest versions of the Linux Kernel, there have been efforts to optimize the energy consumption of parallel workloads by tuning the uncore frequency [22]–[25]. Works that adapt two variables to optimize the energy efficiency of parallel applications include Nornir [26], *Conductor* [27], *Hoder* [28], PARMA [29], CoScale [30], and Powerspector [31].

Only a few approaches simultaneously optimize the number of threads, the core, and the uncore frequency levels. A. Navarro et al., [32] exploit the energy savings when dynamic concurrency throttling, DVFS, and uncore frequency scaling are applied for parallel task-based applications. Chadha and Gerndt [33] propose a framework for managing search spaces and region-level optimizations for OpenMP applications. Differently from both works, *NeurOPar* does not rely on the need for a particular compiler or specific parallel programming model. Because of that, it can be applied to optimize any application regardless of how it was implemented. Furthermore, the prediction model generated by *NeurOPar* can be used together with online strategies by providing insights to improve performance and energy consumption further.

VII. Conclusions and Future Work

We have presented *NeurOPar*, an optimization strategy for parallel workloads driven by an artificial neural network. It considers hardware and software metrics associated with an application. It predicts combinations of the number of threads, core, and uncore frequency levels that yield the best trade-off between performance and energy consumption. Through experiments on four multicore processors using twenty-five applications, we demonstrate that *NeurOPar* predicts combinations that yield EDP values close to the best ones achieved by an exhaustive search and improve the overall EDP by 42% compared to the default execution of HPC applications. Moreover, we have shown that *NeurOPar* can enhance the execution of parallel applications without incurring the overhead associated with online methods. In future work, we intend to leverage information from the predictor to adapt the application execution at runtime and consider information from similar applications to improve the prediction of new applications.

Acknowledgment

This study was financed in part by the CAPES - Finance Code 001, FAPERGS, CNPq, CNPq/MCTI/FNDCT - Universal Ed18/2021 number 406182/2021-3, and by Petrobras 2020/00182-5.

REFERENCES

[1] P. O. A. Navaux, A. F. Lorenzon, and M. da Silva Serpa, "Challenges in high-performance computing," *Journal of the Brazilian Computer Society*, vol. 29, no. 1, pp. 51–62, 2023.

[2] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs," *SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 277–286, 2008.

[3] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *IEEE HPCA*, 2013, pp. 639–650.

[4] A. F. Lorenzon and A. C. S. Beck Filho, *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature, 2019.

[5] Johannes Hofmann and Georg Hager and Dietmar Fey, "On the accuracy and usefulness of analytic energy models for contemporary multicore processors," in *Lecture Notes in Computer Science*. Springer Int. Publishing, 2018, pp. 22–43.

[6] E. Le Sueur and G. Heiser, "Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns," in *Proc. of the 2010 Int. Conf. on Power Aware Computing and Systems*, ser. HotPower'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[7] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *IEEE HPCA*, 2013, pp. 424–435.

[8] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient*. Berlin, Heidelberg: Springer, 2009, pp. 1–4.

[9] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi *et al.*, "KerasTuner," https://github.com/keras-team/keras-tuner, 2019.

[10] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Performance Evaluation Rev.*, vol. 40, no. 3, pp. 13–17, 2012.

[11] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2013, pp. 194–204.

[12] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of Thread-level Parallelism in Desktop Applications," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 302–313, 2010.

[13] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[14] R. P. Fletcher, X. Du, and P. J. Fowler, "Reverse time migration in tilted transversely isotropic (TTI) media," *Geophysics*, vol. 74, no. 6, pp. WCA179–WCA187, 2009.

[15] R. A. Shafik, A. Das, S. Yang, G. Merrett, and B. M. Al-Hashimi, "Adaptive Energy Minimization of OpenMP Parallel Applications on Many-Core Systems," in *Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures*, ser. PARMA-DITAM '15. New York, NY, USA: ACM, 2015, p. 19–24.

[16] A. F. Lorenzon, C. C. de Oliveira, J. D. Souza, and A. C. S. Beck, "Aurora: Seamless Optimization of OpenMP Applications," *IEEE TPDS*, vol. 30, no. 5, pp. 1007–1021, 2019.

[17] A. K. Porterfield, S. L. Olivier, S. Bhalachandra, and J. F. Prins, "Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs," in *IEEE IPDPSW*, 2013, pp. 884–891.

[18] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 169–180, 2014.

[19] H. A. Hassan, S. A. Salem, and E. M. Saad, "A smart energy and reliability aware scheduling algorithm for workflow execution in DVFS-enabled cloud environment," *Future Generation Computer Systems*, vol. 112, pp. 431–448, 2020.

[20] S. Hajiamini, B. Shirazi, A. Crandall, and H. Ghasemzadeh, "A dynamic programming framework for dvfs-based energy-efficiency in multicore systems," *IEEE Trans. on Sustainable Computing*, vol. 5, no. 1, pp. 1–12, 2020.

[21] M. Safari and R. Khorsand, "PL-DVFS: Combining Power-Aware List-Based Scheduling Algorithm with DVFS Technique for Real-Time Tasks in Cloud Computing," *J. Supercomput.*, vol. 74, no. 10, p. 5578–5600, oct 2018.

[22] J.-Y. Won, X. Chen, P. Gratz, J. Hu, and V. Soteriou, "Up by their bootstraps: Online learning in Artificial Neural Networks for CMP uncore power management," in *IEEE HPCA*, 2014, pp. 308–319.

[23] N. Gholkar, F. Mueller, and B. Rountree, "Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. NY, USA: ACM, 2019.

[24] E. André, R. Dulong, A. Guermouche, and F. Trahay, "DUF : Dynamic Uncore Frequency scaling to reduce power consumption," Feb. 2020, working paper or preprint.

[25] J. Corbalan, O. Vidal, L. Alonso, and J. Aneas, "Explicit uncore frequency scaling for energy optimisation policies with EAR in Intel architectures," in *IEEE CLUSTER*, 2021, pp. 572–581.

[26] D. D. Sensi, M. Torquati, and M. Danelutto, "A Reconfiguration Algorithm for Power-Aware Parallel Applications," *TACO*, vol. 13, no. 4, pp. 43:1–43:25, 2016.

[27] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A Run-Time System for Power-Constrained HPC Applications," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer, 2015, pp. 394–408.

[28] J. Schwarzrock, C. C. de Oliveira, M. Ritt, A. F. Lorenzon, and A. C. S. Beck, "A Runtime and Non-Intrusive Approach to Optimize EDP by Tuning Threads and CPU Frequency for OpenMP Applications," *IEEE TPDS*, vol. 32, no. 7, pp. 1713–1724, 2021.

[29] M. A. N. Al-hayanni, A. Rafiev, F. Xia, R. Shafik, A. Romanovsky, and A. Yakovlev, "PARMA: Parallelization-Aware Run-Time Management for Energy-Efficient Many-Core Systems," *IEEE Transactions on Computers*, vol. 69, no. 10, pp. 1507–1518, 2020.

[30] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *IEEE/ACM Micro*, 2012, pp. 143–154.

[31] X. You, H. Yang, Z. Xuan, Z. Luan, and D. Qian, "PowerSpector: Towards Energy Efficiency with Calling-Context-Aware Profiling," in *IEEE IPDPS*, 2022, pp. 1272–1282.

[32] A. Navarro Muñoz, A. F. Lorenzon, E. Ayguadé Parra, and V. Beltran Querol, "Combining Dynamic Concurrency Throttling with Voltage and Frequency Scaling on Task-Based Programming Models," in *ICPP*. NY, USA: ACM, 2021.

[33] M. Chadha and M. Gerndt, "Modelling DVFS and UFS for Region-Based Energy Aware Tuning of HPC Applications," in *IEEE IPDPS*, 2019, pp. 805–814.