# A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs

Daniel Nemirovsky*, Tugberk Arkose*, Nikola Markovic†, Mario Nemirovsky*, Osman Unsal*, Adrian Cristal*
*Barcelona Supercomputing Center
{daniel.nemirovsky, tugberk.arkose, mario.nemirovsky, osman.unsal, adrian.cristal}@bsc.es
†Microsoft
nimarkov@microsoft.com
‡ICREA

***Abstract*—As heterogeneous systems become more ubiquitous, computer architects will need to develop novel CPU scheduling techniques capable of exploiting the diversity of computational resources. Accurately estimating the performance of applications on different heterogeneous resources can provide a significant advantage to heterogeneous schedulers seeking to improve system performance. Recent advances in machine learning techniques including artificial neural network models have led to the development of powerful and practical prediction models for a variety of fields. As of yet, however, no significant leaps have been taken towards employing machine learning for heterogeneous scheduling in order to maximize system throughput.**

**In this paper we propose a unique throughput maximizing heterogeneous CPU scheduling model that uses machine learning to predict the performance of multiple threads on diverse system resources at the scheduling quantum granularity. We demonstrate how lightweight artificial neural networks (ANNs) can provide highly accurate performance predictions for a diverse set of applications thereby helping to improve heterogeneous scheduling efficiency. We show that online training is capable of increasing prediction accuracy but deepening the complexity of the ANNs can result in diminishing returns. Notably, our approach yields 25% to 31% throughput improvements over conventional heterogeneous schedulers for CPU and memory intensive applications.**

***Index Terms*—machine learning, artificial neural networks, scheduling, heterogeneous systems**

## I. Introduction

Heterogeneous computational resources have allowed for effective utilization of increasing transistor densities by combining very fast and powerful cores with more energy efficient cores as well as integrated GPUs and other accelerators. Interest in heterogeneous processors within the industry has recently translated into several practical implementations including ARM's big.LITTLE [5]. However, in order to fully utilize and exploit the opportunities that heterogeneous architectures offer, multi-program and parallel applications must be properly managed by a CPU scheduler.

Effective schedulers should be aware of a system's diverse computational resources, the variances in thread behaviors, and be able to identify patterns related to a thread's performance on different cores. Furthermore, since applications may perform differently on distinct core types, an efficient scheduler should be able to estimate performances in order to identify an optimal mapping scheme. Mapping determines which thread to send to which core and is a problem that shares similarities with recommendation systems and navigation systems both of which have benefitted using machine learning.

Machine learning (ML) and deep learning techniques have been applied with considerable success in the fields of computer vision, natural language processing, and for recommender systems. Artificial neural networks (ANNs) in particular, are beginning to be utilized in a wide variety of fields due to their great promise in learning relationships between input data and both numerical or categorical outputs. The relationships learned by the ANNs are often hard to identify and program for manually but can provide excellent prediction accuracies. Though ML techniques have been gaining traction over the last few years, its application toward improving hardware performance remains in its earliest stages. As of yet, there has been no seminal work applying ML for predicting thread performance on heterogeneous cores and maximizing system throughput.

The objective of this work is the proof of concept of the opportunities that arise by applying ML to computer architecture designs. To achieve this we have chosen to employ ML to improve scheduling for heterogeneous systems since scheduling is an area which is shares similarities with others who have benefitted using ML. In this paper we use lightweight ANN performance predictors to improve a scheduler's ability to identify the mapping scheme which results in the highest system throughput that other conventional schedulers cannot.

The preliminary results show significant performance benefits between 25%-31% compared to an efficient state-of-the-art heterogeneous scheduler. We also show how increasing the complexity of the ANN predictors may result in diminishing returns compared with the more lightweight ANNs which are easier to implement and have less overheads. These initial results help to validate the proof of concept that to our knowledge is the first to leverage machine learning to predict thread performance at the scheduling quantum granularity and improve heterogeneous scheduling.

Our contributions include:

- A heterogeneous scheduling model making use of a next quantum thread behavior predictor, machine learning based performance predictors, and a system throughput maximization scheduling policy.

978-1-5090-1233-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SBAC-PAD.2017.23

IEEE computer society

- The design, implementations, and evaluation of two lightweight and one deep ANN based performance predictors for two different core types which produce an estimated instruction per cycle (IPC) value per scheduling quantum for different threads.

The rest of this paper is structured as follows. Section II details the motivation for this work and provides a brief technical background. Section III presents our proposed ANN based heterogeneous scheduling model. The experimental validation of our work is discussed in Section IV. Lastly, we present related work in Section V and future work and conclusion in Section VI.

## II. Motivation

Finding a mapping scheme which maximizes throughput requires knowledge of the different system resources and estimating how different threads will behave on the cores. Figure 1 illustrates the performance differences that result from fully executing SPLASH-2 benchmarks on a large core compared to a small core (for core details see Section IV-A). The blue bar corresponds to IPC achieved when running the application on the large core and the grey bar is the IPC when run on the small core. The red line corresponds to the percentage difference between the two bars (i.e., $IPC_{large}/IPC_{small} - 1$). On average, the applications achieve nearly 2x more IPC when executing on the large core vs. the small core. Variations in IPC differences can also be observed between applications (inter-application). For some applications, these IPC differences can be either very minor (radix 1% and barnes 6%) or very sizable (water.nsq 200% and fmm 232%). These inter-application variations can be partially explained by the code's structure and algorithms including loops, data dependencies, I/O and system calls, and memory access patterns among others. Similar differences are observed with the SPEC2006 benchmarks which are not shown due to space limitations.

Similarly, intra-application variations of core to core IPC differences are also present within the different basic blocks and phases of every application. The more intra- and inter-application variations there are, the harder it is for a scheduler to identify the optimal mapping scheme, but the more opportunities for improvement there are.

To showcase how identifying these core to core IPC differences can translate into mapping benefits, consider the case where four applications (e.g., A, B, C, and D) are selected to run on a chip multiprocessor (CMP) with one large core and three small cores. Four mapping schemes which assign one application to the large core and the other three to the small cores can be A-BCD, B-CDA, C-DAB, D-ABC. Each mapping scheme will produce a different resulting system IPC. The overall benefits of an effective mapper will be based upon the difference between the best and worst mapping schemes. For instance if A-BCD is the best mapping scheme resulting in a system IPC of 4 and C-DAB is the worst with a system IPC of 2, then the difference in percentage terms would be 100% (i.e., $(4-2)/2$).

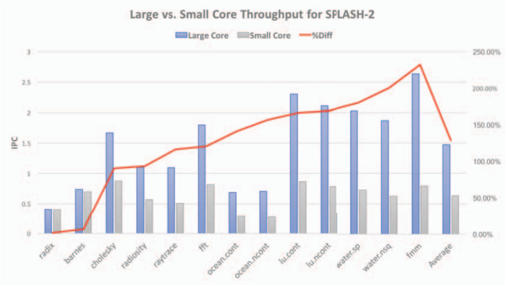


Fig. 1: Core to core IPC differences for SPLASH-2 benchmarks.

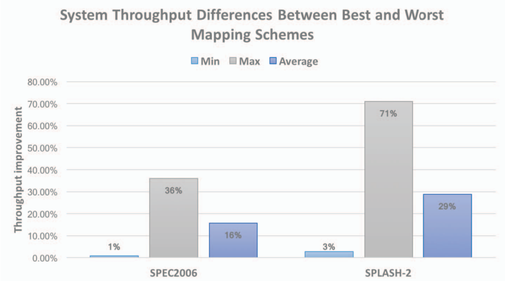


Fig. 2: The average, maximum, and minimum percentage differences between the best and worst mapping schemes for all possible combinations of SPEC2006 and SPLASH-2 benchmarks on a 1-large and 3-small core system.

To demonstrate this in practical terms, we found the differences between the best and worst mapping schemes for all possible combinations of four applications from both the SPEC2006 and SPLASH-2 suites. Figure 2 presents the average, minimum, and maximum differences (in percentage terms) between best vs. worst mapping schemes for all possible combinations. The minimum differences range from 1%-3%, average differences from 16%-29%, and maximum differences from 36%-71% for all possible mapping schemes for SPEC2006 and SPLASH-2 respectively. These results expose the theoretical benefits that may be gained from an effective scheduler at the application level granularity. Practical schedulers, however, work at the quantum level granularity and may additionally identify and take advantage of intra-application core to core performance differences which could expose greater opportunities for mapping optimization.

In order to identify an optimal mapping scheme, a heterogeneous scheduler should be able to estimate the system performance that each individual mapping scheme would produce. Conventional schedulers, however, typically do not make use of the mechanisms needed to exploit this potential. As we shall see, machine learning can be a powerful tool for schedulers to utilize in order to help estimate system performance.

### A. Background

*1) Machine learning (ML):* We believe the predictive power of ML techniques such as artificial neural networks (ANNs) can be of significant use to computer architects for estimating system performance. In this work we focus on increasing system throughput by implementing different ANN based performance predictors. ANNs were explored in contrast to other methods such as simple linear regression, support vector machines and decision trees due to their relative lightweight implementation (for lower overheads), high accuracies for similar prediction problems, and growing hardware support.

ANNs consist of a set of input parameters connected to a hidden layer of artificial neurons which may be connected to other hidden layers before connecting to one or more output neurons. These connections are each assigned a numerical weight that is multiplied with its corresponding input parameter and then added together with the result of the neuron's other incoming connections. The sum is then passed into the activation function of the neuron, typically either a rectified linear. The output of these neurons is then fed as input to the next layer of neurons or to the the output neuron(s). An ANN can learn to produce accurate predictions by training its weights based on input data using supervised learning. Learning is performed via a learning algorithm such as backpropagation that adjusts the weights in order to find an optimal minima which minimizes the prediction error based on a target output and error function. Advances in learning algorithms have enabled faster training times and allowed for the practical use of very deep and intricate ANN implementations. Another useful feature of ANNs is that they can also keep learning dynamically (often called online learning) by periodically training as new data samples are provided. Additionally, the calculation overheads required may be done in parallel for all neurons in the same layer. These calculations can also be sped up through the use of hardware support including GPUs, FPGAs, or specialized accelerators.

*2) CPU scheduling:* CPU scheduling is responsible for determining which software threads are to be run on the hardware resources (i.e., mapping schemes). Schedulers are generally triggered periodically via a scheduling quantum to preempt the current execution of the running thread(s) and produce a new mapping scheme. They may also be called asynchronously as in the case of a thread stall. Schedulers produce overheads which may mitigate efficiency gains due to the cost of calculating optimal thread to core mapping as well as from context swap penalties. Therefore, it is imperative for effective schedulers to balance finding an optimal mapping without triggering unnecessary context swaps and being as lightweight as possible.

Recognizing and exploiting the variations in program behavior is instrumental for effective schedulers to achieve an optimal mapping scheme to maximize system performance. While not all programs exhibit the same behavior, studies [4], [18] have shown that the behavioral periodicity in different applications is typically consistent. In fact, the behavioral periodicity has been shown to be roughly on the order of several millions of instructions and is present in various different and even non correlated metrics stemming from looping structures inside of applications. Heterogeneous architectures provide excellent environments for exploiting the behavioral diversity of concurrently executing programs. However, non-heterogeneous aware schedulers like the current Linux Completely Fair Scheduler (CFS) [15] cannot take advantage of diverse system resources.

## III. ML BASED HETEROGENEOUS SCHEDULING

In this section we present our heterogeneous CPU scheduling model using ANN based performance predictors. The structure of the scheduler is shown in Figure 3 and consists of four parts. First is the statistical information about each thread's state and which core type it has been executing on. The second is a next quantum behavior predictor (NQP) that predicts what will be a thread's behavior during the next scheduling quantum. Thirdly is a set of ANN based performance predictors which use a thread's behavior statistics to estimate its performance for a given core type. One ANN is used to estimate performance on the large core and a separate ANN is used to predict for the small core. The fourth and final part is a scheduling policy that uses the estimated performance of all threads on all core types in combination with knowledge of the available system resources to determine and initiate a mapping scheme for the next scheduling quantum which maximizes system throughput.

The scheduler contains a list of threads where each entry details the thread ID, state (e.g., running, ready, or stalled), and which core it previously executed on. There are also two additional fields storing the performance estimates of the thread if it were to be run on the large or small core during the next quantum. One of these fields will be determined by the NQP while the other will be provided by an ANN performance predictor. For example, after a thread executes on the large core, the scheduler stores the observed IPC in the large core performance estimate field and then uses the value generated by the small core ANN predictor to fill in the small core performance estimate field. As a result, threads only need to use one of the two ANN performance predictors that correspond to the core type they did not run on last quantum.

Waiting or stalled threads do not need to go through the NQP or performance predictors since their estimated IPC values will remain the same since last executing. At the end of each scheduling quantum, new IPC estimates are generated for all running threads and the scheduler will apply its scheduling policy to determine the optimal thread to core mapping which maximizes total system throughput.

### *A. Parameter engineering*

Being able to predict the performance of a thread on a particular core depends on characterizing the type of instructions that will be executed next quantum as well as estimating their effects on the given hardware resources. It is therefore important to choose an appropriate set of thread statistics that will be used as input parameters to our predictors. However, in order to use statistical information as input to performance predictors for different core types, the statistics should be as generic as possible. Normalizing the statistics into ratios (e.g.,

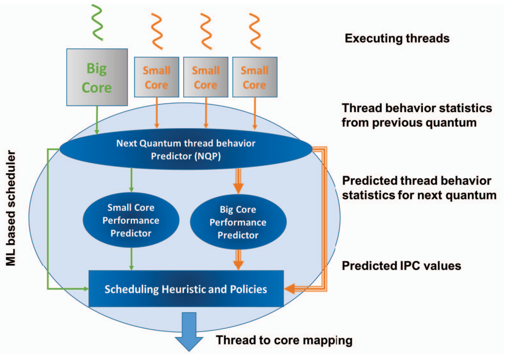

Executing threads
Big Core
Small Core
Small Core
Small Core
Thread behavior statistics from previous quantum
Next Quantum thread behavior Predictor (NQP)
ML based scheduler
Small Core Performance Predictor
Big Core Performance Predictor
Predicted thread behavior statistics for next quantum
Predicted IPC values
Scheduling Heuristic and Policies
Thread to core mapping

Fig. 3: The proposed ML based heterogeneous scheduling model. After each scheduling quantum, statistics collected from the threads pass through the NQP and the ANN based performance predictors. A scheduling policy then uses the estimated IPC performance of all threads for both the large and small cores to identify a mapping scheme to maximize system throughput for the next quantum.

instruction mix ratios) ensures ANN input consistency in cases when training is done with data collected from the small core but predicting is done using input statistics gathered from the large core.

Without using normalization to get the instruction mix ratio, we would be left with inconsistent statistical input to the ANNs since the number of actual executed instructions of each type depend heavily on the microarchitecture of the cores (e.g., an out-of-order core may execute more instructions than an in-order core even though the instruction mix ratios may be the same). Different forms of normalization can also be used in cases when the core types have different ISAs or cache configurations. Normalizing statistics enables our approach to be useful in systems with a variety of different architectures.

In determining the final set of statistics to use as input parameters, we sought to balance suitable ANN predictor accuracy while minimizing the collection and arithmetic overheads. After exploring how different statistics affect ANN prediction accuracies we settled upon the following nine values: **1)** DL1, **2)** L2, and **3)** L3 cache miss rates, instruction mix ratios (percent of total instructions) including **4)** loads, **5)** stores, **6)** floating point multiplications and divisions, **7)** floating point additions and subtractions, **8)** branches, and **9)** generic arithmetic operations.

The simulation framework we have used is capable of dynamically profiling these parameters as the applications are being executed. Moreover, many conventional CPUs come with hardware support such as counters for collecting similar statistics and in future work we will seek to further improve the set of statistics used as input parameters.

### B. Next quantum thread-behavior predictor (NQP)

Several novel approaches [4], [20] have been proposed which predict program behavior based upon various statically or dynamically collected program statistics. However, to lower overheads and for simplicity, in this work, we use a next quantum thread behavior predictor (NQP) that will always predict the next behavior to be equal to the previous quantum behavior. The NQP does not rely on states, instead it sends the nine behavioral statistics gathered during a thread's previous scheduling quantum as the input parameters to the ANNs.

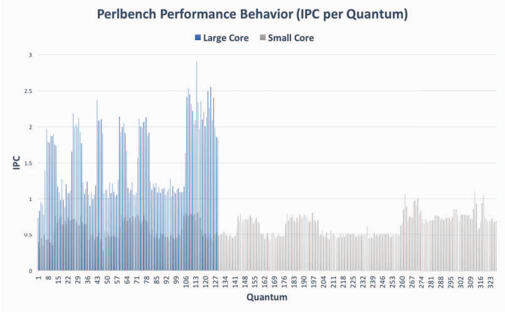


Fig. 4: The IPC per quantum behavior of perlbench.

Figure 4 helps to visualize this behavioral periodicity. It shows the IPC variability of the SPEC2006 benchmark perlbench throughout its simulated execution on an Intel Nehalem x86 using a 1ms scheduling quantum. Though there are clearly periodic behavioral phases that span tens and sometimes hundreds of quanta, it is also possible to observe that for finer granularities, the IPC variation from quantum to quantum is quite minimal, and more so on the small core. The figure also highlights the intra and inter application core to core IPC differences which can fluctuate between phases.

The NQP results in average errors (including both benchmark suites) of 8% and 7% for the large and small cores respectively. The errors were calculated after measuring the IPC differences of threads after running on the same core for sequential quanta using the equation: $error_i = \frac{|y_i - t_i|}{t_i}$; $\mu_{error} = \frac{1}{n} \times \sum_{i=1}^{n} error_i$. Where $y$ is the predicted IPC and $t$ is the target (i.e., observed) IPC value for quantum $i$ and $n$ is the total number of quanta (i.e., samples).

The errors are may vary amongst benchmarks (not plotted due to space constraints), and can impact on the accuracy of the ANN predictors and the efficiency of the resulting mapping scheme.

### C. ANN performance predictor models

We implement and evaluate three different ANN performance predictor models which are described below. Each model uses two separate but architecturally similar artificial neural networks; one to predict for the large core and one to predict for the small core. For simplicity, we utilize fully connected and feedforward ANN implementations for all models. The number of hidden layers and units for each ANN model was the result of calibrating the models to predict accurately for both training and testing data while remaining feasible to implement and utilize every 1ms. The input parameters to the ANNs are the set of 9 statistics that characterize each thread provided by the NQP. The ANNs have a single output

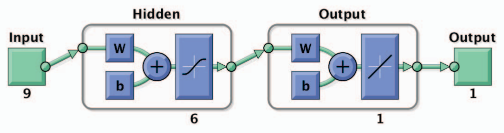


Fig. 5: ANN used by the Base and Online performance predictor models.

value corresponding to the estimated IPC that the thread is will achieve on a particular core during the next scheduling quantum. The ANNs are implemented using the Matlab ML toolkit [11]. The three ANN performance predictor models are:

**Base**: Uses two predictors (one for the large core and one for the small core), each of which are based on the same lightweight ANN architecture (shown in Figure 5). It is composed of nine input parameters, one hidden layer with six hidden units, and one output unit. All input units are interconnected to every hidden unit. The hidden unit's activation function is the hyperbolic tangent sigmoid transfer function while the output unit used is a rectified linear unit in order to predict a numerical value for the IPC. The Base model is used to evaluate the accuracy of the predictor as if it would only have seen and been trained on a subset of all the applications that are to be run on a system. To accomplish this, we train both the large and small core predictors with data collected from individual executions of all SPLASH-2 benchmarks on both the large and the small cores. We then evaluate the accuracy of the predictor on both the SPLASH-2 and SPEC2006 benchmark suites.

**Online**: A replica of the Base ANN (see Figure 5) architecture but whose purpose is to demonstrate the increased accuracy attainable once online learning has been utilized to keep training the predictors dynamically after all applications from both SPLASH-2 and SPEC2006 are executed at least once. Online training helps to generalize the predictions for diverse workloads and is also useful for improving the prediction accuracy for applications that are executed more than once.

**Deep**: A deep learning model that also makes use of two ANNs to predict for the large and small cores. The ANNs are composed of four hidden layers of twenty hidden units and a single rectified linear output unit (not shown due to space limits). This model is used to showcase the outstanding accuracy that ANN based performance predictors are capable of. Like the Online model, it also emulates using online learning for all the benchmarks in both SPLASH-2 and SPEC2006. Though this model is not the most practical in terms of overheads needed for weight storage and online learning, it does help to highlight whether increasing the accuracy of the predictors would add significant benefits to the scheduler's performance, or whether the bottleneck is somewhere else such as the accuracy of the NQP.

*1) Training and prediction:* The ANNs are useful for predicting the performance of a thread on the core type it has not executed on during the previous quantum. In terms of implementation, however, this translates into needing to train the ANNs using data collected from one core type but predicting using data collected from the other core type. To predict how a thread currently running on a large core will perform on the small core during the next quantum, the nine statistics collected from its execution on the large core is used as the input for the small core ANN predictor. Similarly, the large core ANN predictor uses the statistics collected from a thread's previous execution on a small core. The need to use data from one core type to predict for another core type means that the errors from the NQP will have an impact on the ANN prediction accuracy.

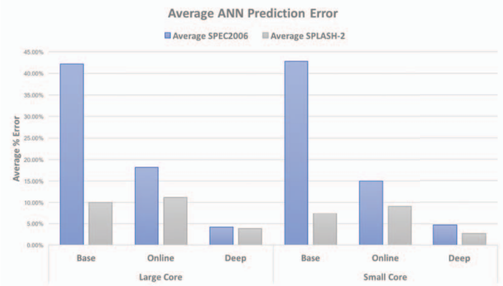


Fig. 6: The average IPC prediction errors of the large and small core ANNs of the three different predictor models when running both benchmark suites after training. Lower numbers are better.

The error minimization function used for training the ANN is the mean square error (mse) which is readily utilized in ML models. The Base ANNs were trained using only SPLASH-2 applications while the Online and Deep ANNs were trained using benchmarks from both SPLASH-2 and SPEC2006 to emulate dynamic learning.

*2) Accuracies:* The accuracy results in terms of average percentage error for the performance predictors are given in Figure 6. As shown, the worst performing predictor model for the SPEC2006 benchmarks is the Base as expected since we have only trained it with data from the SPLASH-2 benchmarks. Its total average error for SPEC2006 is around 42%-43% for the large core and small core ANNs respectively. Its accuracy is much better for SPLASH-2 attaining 10%-7% average error for the large core and small core ANNs respectively.

The Online predictor model greatly reduces the prediction error especially for the SPEC2006 benchmarks but slightly increases the prediction error on the SPLASH-2 benchmarks. This is due to the ANNs generalizing their weights for the behaviors of all benchmarks and is therefore less likely to overfit just for those from the SPLASH-2 suite compared to the Base. The average errors for the Online predictors are 18%-15% (SPEC2006) and 11%-9% (SPLASH-2) for the large and small core ANNs respectively.

The Deep predictor model performs the best due to its complex architecture and use of online learning. It should be noted that this network is the most susceptible to overfitting but can also learn to generalize for newer applications dynamically. Its error are 4%-5% (SPEC2006) and 4%-3 % (SPLASH-2) for the large and small core ANNs respectively.

These results show that the Online predictor model is capable of improving the accuracy of the Base predictor for SPEC2006 by over 4x while maintaining similar accuracies for SPLASH-2. The Deep predictor model improves upon the Base by nearly 10x for SPEC2006 and over 2x for SPLASH-2. Evidently, greater performance prediction accuracy can be achieved by including more diverse training data and using more complex ANNs.

*3) Overheads:* The overheads of the ANNs from the Online predictor model, which we deem to be the most practical approach, consists of approximately 150 floating point and 80 memory operations for each prediction. Conservatively assuming it takes 20 cycles to complete each of the 230 operations, this results in 4,600 cycles of overhead each quantum. The scheduler is called every 1M cycles (CPU clock is set at 1GHz and the scheduling quantum occurs every 1ms) which means that the computational overheads of the Online predictor are approximately 0.5% per prediction. Assuming four predictions (one for each running thread) are to be sequentially calculated every quantum, the total overheads are 2%. Online training may produce more computational overheads but occurs far less often and can be triggered when the system is less busy or there are idle cores.

### *D. Mapping*

Each mapping scheme assigns specific threads to particular cores. In the case of a 1-large 3-small core system executing four threads, at least four different mapping schemes exist which result in a different thread being assigned to the large core. For each possible mapping scheme, the scheduler estimates the total system IPC by calculating the sum of the predicted IPC of each thread for each core. The scheduler then ranks the different mapping schemes based on the system IPC estimates and selects the highest for the next quantum.

## IV. Evaluation

This section presents the experimental setup and results of our simulations validating our proposal. The significance of the results should be viewed in terms of demonstrating the potential that applying lightweight ML techniques such as ANNs can have for improving system throughput.

### *A. Methodology*

This work uses the Sniper [1] simulation platform. Sniper is a popular hardware-validated parallel x86-64 multicore simulator capable of executing multithreaded applications as well as running multiple programs concurrently. The simulator can be configured to run both homogeneous and heterogeneous multicore architectures and uses the interval core model to obtain performance results.

The processor that is used for all experimental runs in this work is a quad-core heterogeneous asymmetric multi-core processor consisting of 1 large core and 3 identical small cores. Both types of cores are based on the Intel Nehalem x86 architecture running at 2.66GHz. Each core type has a 4 wide dispatch width, but whereas the large core has 128 instruction window size, 8 cycle branch misprediction penalty (based on the Pentium M predictor), and 48 entry load/store queue, the small core has a 16 instruction window size, 14 cycle branch misprediction penalty (based on a one-bit history predictor), and a 6 entry load/store queue. The cache hierarchies of both core types are the same (private 256KB L1, 512KB L2, and shared 8MB L3).

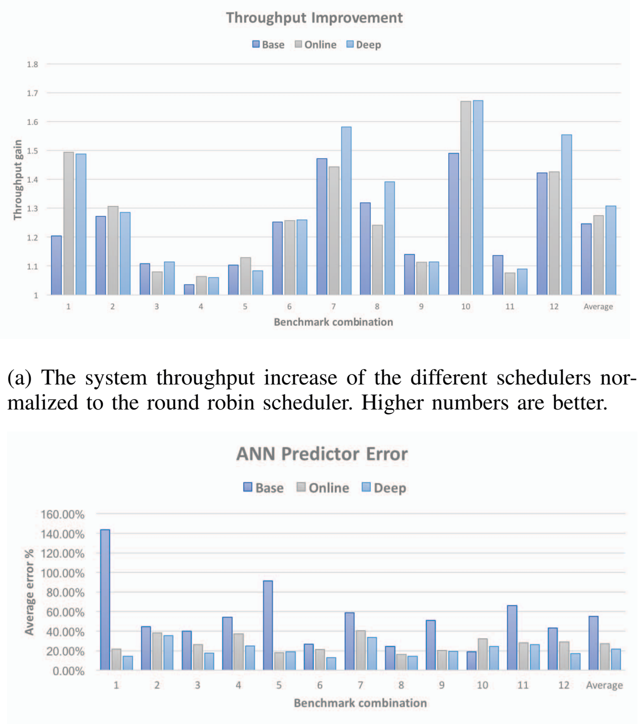
| Combo | Benchmarks | ANN error |
|---|---|---|
| 1 | astar / lbm / tonto / h264ref | High |
| 2 | gcc / cactusADM / gromacs / bwaves | Med |
| 3 | hmmer / mcf / gcc / bwaves | Med |
| 4 | leslie3d / gcc / bwaves / mcf | Med |
| 5 | omnet / astar / bwaves / gemsFDTD | High |
| 6 | perlbench / calculix / gamess / bzip2 | Low |
| 7 | calculix / mcf / soplex / gcc | Med |
| 8 | gobmk / xalancbmk / namd / zeusmp | Low |
| 9 | astar / zeusmp / cactusADM / gemsFDTD | High |
| 10 | namd / gromacs / calculix / cactusADM | Med |
| 11 | povray / gromacs / libquantum / bzip2 | High |
| 12 | sjeng / leslie3d / gobmk / milc | Med |

TABLE I: Simulated Benchmark Combinations

The 1-large 3-small multi-core system configuration is based on the experimental framework used in previous work [10], [21]. These works also make use of Sniper, which unfortunately does not provide for a wide selection of different architectures such as ARM but does support hardware validated x86 core types. We believe that using the (admittedly limited) experimental setup as employed in previous work allows for the fairest comparison.

We have used the popular SPEC2006 [7] and SPLASH-2 [22] benchmark suites to evaluate and train our scheduler. The SPEC2006 benchmark suite is an industry-standardized, CPU-intensive benchmark suite, stressing a system's processor and memory subsystem. The SPLASH-2 benchmark suite is composed of a mix of different multithreaded kernels and applications focusing on high performance computing, graphics, and signal processing. The entirety of the benchmark suites are used with the exception of those which did not compile in our platform (volrend, dealII, wrf, and sphinx3).

In running the simulations presented in this evaluation section, we created 12 combinations of 4 different SPEC2006 applications given in Table I. These different combinations were chosen to fairly include benchmarks which exhibit low, medium, and high amounts of IPC prediction errors on our performance predictors (shown categorized in the last column of the table). All four benchmarks of each combination are run from start to finish and the benchmarks that finish first are restarted such that the number of benchmarks running at any one time on the system remains constant. Once all four benchmarks finish at least once, the simulation for that combination is ended. This is done to be consistent with simulating in the context of a real system which generally has enough ready threads to stay continuously busy and to



(a) The system throughput increase of the different schedulers normalized to the round robin scheduler. Higher numbers are better.

(b) The average ANN prediction error for all benchmark combinations for the different schedulers. Lower numbers are better.

Fig. 7: Simulation results of our ML based heterogeneous schedulers.

demonstrate the ability of our scheduler to maximize system throughput.

*1) Schedulers:* We have chosen to use the efficient state-of-the-art Hardware Round Robin Scheduler [10] as the baseline to validate our proposals since it has been shown to already provide performance improvements for single and multi-threaded workloads over both the Linux scheduler and Fairness-aware scheduler [15], [21]. The round robin scheduler functions by swapping the thread executing on the large core with one of the threads executing on a small core every quantum.

The three ML schedulers evaluated only differ in the ANN performance predictors they use and are named as such (i.e., the Base scheduler utilizes Base ANN models for predicting for the large and small cores, the Online scheduler uses the Online ANN model, and the Deep uses the Deep ANN model).

To account for context swap overheads, we apply a 1000 cycle penalty for consistent with the metric value utilized in the round robin study.

### *B. Results*

Figure 7a shows the system throughput improvement of our schedulers over the round robin approach. Since our scheduling policy optimizes for maximum system throughput, these results are central for evaluating and validating our ML based heterogeneous scheduling approach. The three proposed schedulers achieve impressive average throughput improvements over the round robin scheduler of 25%, 28%, and 31% for the Base, Online, and Deep schedulers respectively. These improvements reflect the importance and significant differences that can arise from using distinct mapping schemes. The results show that the ANN schedulers are able to identify optimal mapping schemes that the round robin scheduler cannot.

The variations between the three schedulers for each individual benchmark combination are noticeably related to the differences in their ANN and NQP errors. Some of the throughput differences between the schedulers for some of the combinations can also be attributed the fact that the different ANNs learn different relationships between the input parameters and output performance. Some of these learned relationships may be very beneficial for some applications but less so for others which behave irregularly. In addition, secondary effects such as context swap penalties and thread interference can vary depending on the mapping and thread behaviors. We seek to include these effects as part of a future model.

Figure 7b shows the average system performance (IPC) prediction error using the three different ANNs models. As expected, the Base scheduler, which averages 55% error, performs worse than both the Online and Deep predictors and achieves very poor accuracy results for a couple of the combinations. These large errors illustrate that the Base ANN did not learn to generalize adequately to account for the behaviors present in these combinations. Continuing to learn dynamically as new applications are run can greatly improve the accuracy as illustrated by the Online scheduler results averaging 27% error. Further increasing the complexity of the ANNs can also result in accuracy improvements as shown by the Deep results which average 21% error. The errors are higher than those shown earlier (Section III-C2) for the individual benchmark runs because these errors also inherently include those of the NQP. This is the case because the input to the ANNs comes directly from the NQP.

However, though the measured average error rate of the NQP can reach up to 14%, our proposed schedulers nonetheless result in high accuracy and performance gains. This demonstrates that even accounting for these phase and periodic variations in application behavior, it is still a reasonable approach for our proposal to predict the behavior for the next quantum based upon the previous quantum. In sum, these promising results validate the instrumental value that using ANNs performance predictors can provide for effective heterogeneous scheduling.

## V. Related work

To our knowledge this paper is the first to apply ML to predict individual thread performance each scheduling quantum for CPU schedulers to maximize system throughput on heterogeneous systems. The work presented in [8] studies the accuracy of SVMs and linear regression in predicting the performance of threads on two different core types. However, they do so at the granularity of 1 second, use only a handful of benchmarks, and do not implement the predictor inside of a scheduler. In the study [6], CPU burst times of whole jobs for computational grids are estimated using a ML approach using decision trees and k-nearest neighbors. A related work targeting grids by predicting execution time, memory and disk consumption for bioinformatics applications is done in [12]. An approach that utilized ML for selecting whether to execute a task on a CPU or GPU based on the size of the input data is done by Shulga et. al. [19]. Nearly all of these approaches deal with either program or process level predictions and target homogeneous systems.

Moncrieff et al. [14] and Menasce et al. [13] analytically examined the tradeoffs between utilizing fast and slow processors in heterogeneous processors. Their study showed that a system composed of few fast cores and many slow cores are effective in terms of cost and performance. Optimal scheduling of independent applications running on a preemptive heterogeneous CMP has been studied by Liu et al. [9].

Chronaki et al. [2], [3] propose a heterogeneous scheduler for a dataflow programming model which improves performance using a prioritization scheme and dynamic task dependency graph to assign newly created and critical tasks to fast cores. Another fair based scheduling approach for asymmetric CMPs was explored by Saez et. al. in [17]. A statistical method using extreme value theory is used in [16] to determine the probabilities for optimal task assignment in massively multithreaded processors.

## VI. Future work and conclusion

In this paper we have pioneered applying machine learning to a throughput maximizing heterogeneous CPU scheduler capable of predicting the performance of multiple threads on diverse system resources at the scheduling quantum granularity. We have shown how a lightweight ANN can provide highly accurate performance predictions for a diverse set of applications even while only being trained on a small subset. In addition, we described how the predictor accuracy can be greatly improved upon through the use of online learning and deep learning. Our approach yields significant results with average throughput improvements between 25% to 31% over conventional heterogeneous schedulers for CPU and memory intensive applications.

We seek to expand the scope of our work in the future by a further analysis of ANN hyperparameters and alternative ML models as well as testing our approach on a physical heterogenous CMP. We hope that the novelty of this work has exposed some of the exciting opportunities available by applying machine learning techniques in the field of computer architecture.

## References

[1] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–12.

[2] K. Chronaki, A. Rico, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 329–338.

[3] K. Chronaki, A. Rico, M. Casas, M. Moreto, E. Ayguade, M. Valero *et al.*, "Task scheduling techniques for asymmetric multi-core systems," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[4] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. EEE, 2003, pp. 220–231.

[5] P. Greenhalgh, "big.little processing with arm cortex-a15 & cortex-a7," 2011, [Online] Available: http://www.arm.com /files/downloads/bigLITTLE_Final_Final.pdf.

[6] T. Helmy, S. Al-Azani, and O. Bin-Obaidellah, "A machine learning-based approach to estimate the cpu-burst time for processes in the computational grids," in *Artificial Intelligence, Modelling and Simulation (AIMS), 2015 3rd International Conference on*. IEEE, 2015, pp. 3–8.

[7] S. M. J. Henning, "Spec cpu2006 benchmark descriptions," in *Proc. of of the ACM SIGARCH Computer Arch. News*, 2006, pp. 1–17.

[8] C. V. Li, V. Petrucci, and D. Mossé, "Predicting thread profiles across core types via machine learning on heterogeneous multiprocessors," in *Computing Systems Engineering (SBESC), 2016 VI Brazilian Symposium on*. IEEE, 2016, pp. 56–62.

[9] J. W. Liu and A.-T. Yang, "Optimal scheduling of independent tasks on heterogeneous computing systems," in *Proceedings of the 1974 annual conference-Volume 1*. ACM, 1974, pp. 38–45.

[10] N. Markovic, D. Nemirovsky, V. Milutinovic, O. Unsal, M. Valero, and A. Cristal, "Hardware round-robin scheduler for single-isa asymmetric multi-core," in *European Conference on Parallel Processing*. Springer, 2015, pp. 122–134.

[11] MATLAB, *version R2016b*. Natick, Massachusetts: The MathWorks Inc., 2017.

[12] A. Matsunaga and J. A. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 495–504.

[13] D. Menasce and V. Almeida, "Cost-performance analysis of heterogeneity in supercomputer architectures," in *Supercomputing'90., Proceedings of*. IEEE, 1990, pp. 169–177.

[14] D. Moncrieff, R. E. Overill, and S. Wilson, "Heterogeneous computing machines and amdahl's law," *Parallel Computing*, vol. 22, no. 3, pp. 407–413, 1996.

[15] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.

[16] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: a statistical approach," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 235–248, 2012.

[17] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "Towards completely fair scheduling on asymmetric single-isa multicore processors," *Journal of Parallel and Distributed Computing*, vol. 102, pp. 115–131, 2017.

[18] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE micro*, vol. 23, no. 6, pp. 84–93, 2003.

[19] D. Shulga, A. Kapustin, A. Kozlov, A. Kozyrev, and M. Rovnyagin, "The scheduling based on machine learning for heterogeneous cpu/gpu systems," in *NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), 2016 IEEE*. IEEE, 2016, pp. 345–348.

[20] O. S. Unsal, I. Koren, C. Khrishna, and C. A. Moritz, "Cool-fetch: A compiler-enabled ipc estimation based framework for energy reduction," in *Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004. Eighth Workshop on*. IEEE, 2004, pp. 43–52.

[21] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-isa heterogeneous multi-cores," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 177–188.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 1995, pp. 24–36.