A Deep Learning Mapper (DLM) for Scheduling on Heterogeneous Systems

Daniel Nemirovsky[1(✉)], Tugberk Arkose[1], Nikola Markovic[2], Mario Nemirovsky[1,3], Osman Unsal[1], Adrian Cristal[1], and Mateo Valero[1]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{daniel.nemirovsky,tugberk.arkose,mario.nemirovsky,osman.unsal, adrian.cristal,mateo.valero}@bsc.es
[2] Microsoft, Belgrade, Serbia
nimarkov@microsoft.com
[3] ICREA, Barcelona, Spain

Abstract. As heterogeneous systems become more ubiquitous, computer architects will need to develop new CPU scheduling approaches capable of exploiting the diversity of computational resources. Advances in deep learning have unlocked an exceptional opportunity of using these techniques for estimating system performance. However, as of yet no significant leaps have been taken in applying deep learning for scheduling on heterogeneous systems.

In this paper we describe a scheduling model that decouples thread selection and mapping routines. We use a conventional scheduler to select threads for execution and propose a deep learning mapper to map the threads onto a heterogeneous hardware. The validation of our preliminary study shows how a simple deep learning based mapper can effectively improve system performance for state-of-the-art schedulers by 8%–30% for CPU and memory intensive applications.

1 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 [8]. 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. As a result, heterogeneous scheduling has become a popular area of research and will be essential for supporting new diverse architectures down the line.

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

E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 3–20, 2018.
https://doi.org/10.1007/978-3-319-73353-1_1

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 and deep learning.

Deep learning (DL) techniques and deep neural networks (DNNs) 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 numerical or categorical outputs. The relationships are often hard to identify and program manually but can result in excellent prediction accuracies using DNNs. Though DL 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 DL for predicting thread performance on heterogeneous systems and maximizing system throughput.

The objective of this work is the proof of concept of the opportunities that arise by applying DL to computer architecture designs. The novelty of this work centers on decoupling the selection and mapping mechanisms of a heterogeneous scheduler and fundamentally, the implementation of a deep learning mapper (DLM) which uses a DNN to predict system performance. The selector remains responsible for ensuring fairness and selecting the threads to execute next scheduling quantum while the mapper is charged with identifying an optimal mapping of selected threads onto available cores. Initial results of our proposal are promising, the DLM is capable of improving the performance of existing conventional schedulers (round-robin, fairness-aware, Linux CFS) by 8%, 20%, and 30% respectively for computational and memory intensive applications.

Our contributions include:

- A heterogeneous scheduling model which abstracts and decouples thread selection and mapping.
- An implementation of a deep learning mapper (DLM) that uses a deep neural network for predicting the system performance of different mapping schemes. To our knowledge this work is the first to apply deep learning to CPU scheduling for heterogeneous architectures.

The rest of this paper is structured as follows. Section 2 discusses our motivation and a brief technical overview of mapping, machine/deep learning techniques, and heterogeneous scheduling issues. Section 3 presents our proposed scheduling model with a description of a practical implementation. Validation of our implementation with experimental results is found in Sect. 4. Lastly, we discuss related work in Sect. 5 and future work and conclusion in Sect. 6.

2 Motivation

This section highlights the efficiency opportunities attainable by optimizing mapping on heterogeneous systems and also discusses the rationale for applying DL towards predicting system performance and how decoupling the thread selection and mapping mechanisms can provide model scalability while still ensuring fairness for all threads.

2.1 Mapping

Finding the optimal thread to core mapping on a heterogeneous system is no trivial feat. This is especially the case when executing diverse workloads since the performance of each application is likely to vary from quantum to quantum and core to core. These differences can vary from application to application as well as from phase to phase within an application.

Figure 1 illustrates the performance differences that result from executing SPEC2006 on a large core compared to a small core (for core details see Sect. 4.1). On average, the applications achieve about 2x better system instructions per cycle (IPC) when executing on the large core vs. the small core. Variations in IPC differences can also be observed between applications. For some applications, these IPC differences can be either very minor (mcf 29%, bzip2 33%, and hmmer 36%) or very sizable (gemsFDTD 171%, omnetpp 161%, and perlbench 153%). These 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.

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

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 system with 1-large core and 3-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. $(4-2)/2$).

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 the SPEC2006 benchmark suite. The differences in system performance between the best and worst possible mapping scheme for each combination of four SPEC2006 applications range from 1%–36%. On average, identifying the most adventageous mapping scheme for a given set of four SPEC2006 applications on a 1-large 3-small core system can lead to 16% improvements in system performance. 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.

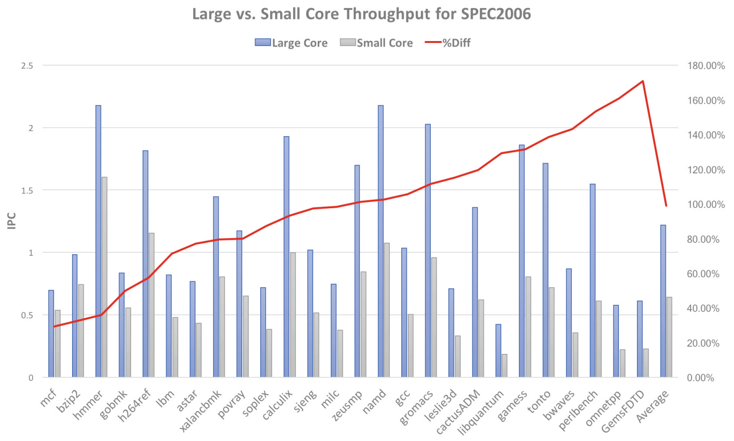


Fig. 1. The performance differences that result from executing each SPEC2006 benchmark on a large vs. small core.

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 such as the Linux Completely Fair Scheduler (CFS), however, typically do not make use of the mechanisms needed to exploit this potential. As we shall see, deep learning can be an effective tool for schedulers to utilize in order to help estimate system performance.

2.2 Machine/Deep Learning

Part of the attraction of machine/deep learning is the flexibility that its algorithms provide to be useful in a variety of distinct scenarios and contexts. For instance, advances in computer vision and natural language processing using convolutional neural network techniques [10,12] have led to high levels of prediction accuracy enabling the creation of remarkably capable autonomous vehicles and virtual assistants. In particular, it is our belief that the predictive power of artificial neural networks (ANNs) will be of great use for computer architects seeking to improve system performance and optimize the utilization of diverse hardware resources. Deep learning (DL) methods expand on more simplistic machine learning techniques by adding depth and complexities to existing models. Using deep ANNs (DNNs) or ensembles of different machine learning models is a typical example of DL.

DNNs consist of a set of input parameters connected to a hidden layer of artificial neurons which are then connected to other hidden layers before connecting to one or more output neurons. The inputs to the hidden and to the output neurons 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 fed into an activation function (usually a rectified linear, sigmoid, or similar). The output of these neurons is then fed as input to the next layer of neurons or to the output neuron(s).

A DNN can learn to produce accurate predictions by adjusting its weights using a supervised learning method and training data. This is performed via a learning algorithm such as backpropagation that adjusts the weights in order to find an optimal minima which reduces the prediction error based on an estimated output, the target output, and an error function. Advances in learning algorithms have enabled faster training times and allowed for the practical use of intricate DNN architectures. DNNs can also keep learning dynamically (often called online learning) by periodically training as new data samples are generated. Moreover, several of the training calculations may be executed in parallel for the neurons within the same layer. The latency of these calculations can be further mitigated through the use of hardware support including GPUs, FPGAs, or specialized neural network accelerators.

2.3 Program Behaviors and CPU Scheduling

Recognizing and exploiting the behavioral variations of programs is instrumental for achieving optimal scheduling schemes to maximize fairness and system performance. Behaviors represent the different characteristics of the program or thread while executing on the physical cores. These can include cache accesses and miss rates, branch prediction accuracies, and instructions per cycle (IPC). While not all programs exhibit the same behavior, studies [7,24] have revealed 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. Behavioral variations may be additionally influenced by interference effects between threads. These effects are generally due to shared data and physical resources between threads and should be taken into consideration by an optimal scheduler.

Yet, even after accounting for program behaviors, finding the optimal scheduling scheme is far from simple. CPU schedulers rely chiefly upon two mechanisms to fulfill their policy objectives: (1) thread selection and (2) thread to core mapping. The thread selection mechanism is responsible for selecting a subset of threads to run from a larger pool of available threads. It does so by using heuristics which order the threads using priorities or scores related to how critical the threads are (e.g. time constrained or system level tasks may be given a higher priority than background tasks which search for application updates) or how much execution time or progress the threads have made so far. The selection mechanism also generally ensures that no threads are continually starved of system resources thereby guaranteeing a certain level of fairness. On homogeneous systems where all cores are identical, the task of mapping individual threads to particular cores depends mainly upon keeping threads close to their data in the cache hierarchy. On heterogeneous systems, in contrast, the mapping mechanism must take into regard the different microarchitectural characteristics of the cores in order to find an optimal mapping of the threads to the cores which is the most effective for its scheduling objective. As a result, schedulers targeted towards homogeneous systems are unable to optimally exploit the resource diversity in heterogeneous systems.

The current Linux Completely Fair Scheduler (CFS) [19] is one such example of a homogeneous scheduler. The state-of-the-art CFS selection scheme combines priorities with execution time metrics in order to select the threads to run next, however, the mapping scheme is relatively simplistic. When mapping, the CFS evenly distributes the threads onto the cores such that all cores have approximately the same number of threads to run. These threads are effectively pinned to the core because they are only swapped with threads on their assigned core and not with those of another core (i.e. threads don't move from the core they were initially assigned to).

Heterogeneous architectures, however, provide excellent environments for exploiting the behavioral diversity of concurrently executing programs and several schedulers targeting these systems have been recently proposed. The fairness-aware scheduler by Van Craeynest et al. [27] is one such scheduler which works similarly to the CFS but instead of mapping all threads evenly on all cores and pinning them there, it maps the highest priority thread (i.e. the one that has made the fewest progress) to the most powerful core. For example, in a 4 core system with 1 powerful core and 3 smaller energy efficient cores, this scheduler will send the thread with the highest priority to the large core and the next 3 highest priority threads to the other 3 small cores.

Another scheduler targeted at heterogeneous systems is the hardware round-robin scheduler by Markovic et al. [15]. Instead of using priorities for thread selection, this approach chooses which threads to run next in a round-robin manner (thereby guaranteeing fairness) and then maps the selected threads to the cores. Using the same 4 core system as described above, this scheduler will rotate the threads in a manner similar to a first in first out queue, from small core to small core to small core to large core and then back into the thread waiting pool until all threads have had a chance to execute.

Scheduling also produces overheads which may reduce the total efficiency gains due to the cost of calculations as well as context swap penalties. It is therefore imperative for effective lightweight schedulers to balance finding an optimal scheduling scheme without triggering costly context swaps.

3 Scheduling Model

In this section we present our scheduling model (shown in Fig. 2) with decoupled thread selection and mapping mechanisms. This scheduling model uses a conventional scheduler (CS) to select a subset of available threads to execute next quantum (using its prioritization scheme) and the deep learning mapper (DLM) to map the selected threads onto the diverse system resources (using a throughput maximization scheme). The scheduling quantum (the periodicity to run the scheduler) chosen is 4 ms for the CS and 1ms for the DLM which reflect typical quantum granularities of CS approaches. This difference allows the DLM to take advantage of the finer grained variations in program behaviors and optimize the mapping on the heterogeneous system while still maintaining CS objectives. Furthermore, the context swap penalties are generally lower for the DLM since it only swaps threads which are already running and have data loaded in the caches while the CS may select to run any thread that may not have any of its data in the caches.

In addition to selecting the threads to run next, the CS is responsible for thread management, including modifying their statuses, dealing with thread stalls, and mapping for the first quantum of new threads or when the number of available threads is less than the number of available cores. When active, the DLM essentially provides a homogeneous abstraction of the underlying heterogeneous hardware to the CS since it only needs to select threads to run and not whether to execute on a large or small core.

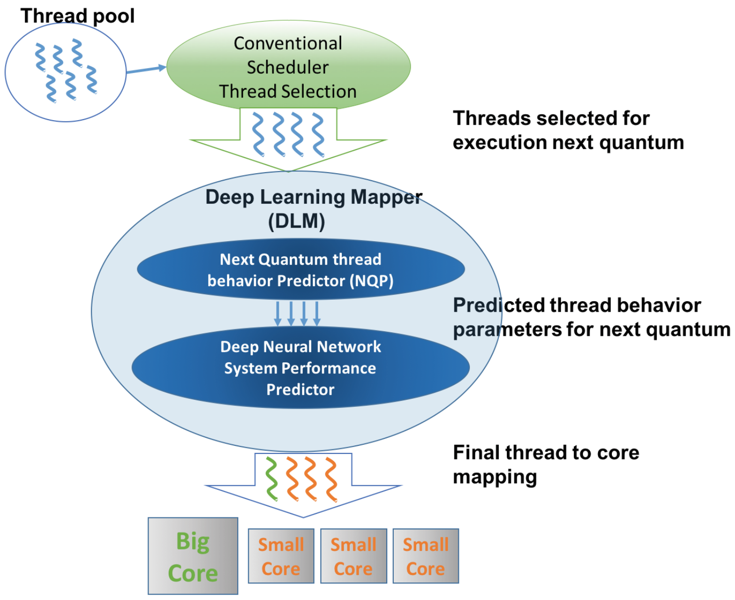

Thread pool
Conventional Scheduler Thread Selection
Threads selected for execution next quantum
Deep Learning Mapper (DLM)
Next Quantum thread behavior Predictor (NQP)
Deep Neural Network System Performance Predictor
Predicted thread behavior parameters for next quantum
Final thread to core mapping
Big Core
Small Core
Small Core
Small Core

Fig. 2. The scheduling model. A conventional scheduler is used to select the threads to run next quantum and the DLM then uses the NQP and DNN predictor to find the optimal mapping to maximize system performance.

3.1 Deep Learning Mapper (DLM)

The DLM is responsible for finding a mapping of the selected threads onto the hardware cores which optimizes system throughput. This objective helps to demonstrate the significant potential that using DNN based performance predictors can have for a continuously busy system. The DLM works by firstly collecting statistical information about each selected thread pertaining to its behavior (described in Sect. 3.1). These are gathered during the thread's previous execution quantum. These statistics are then passed along to the next quantum behavior predictor (NQP) that predicts that the thread's behavior during the next execution quantum will be the same as during its previous quantum. The NQP in essence forwards the behavioral statistics for all threads that have been selected to execute next quantum to our DNN based performance predictor. The DNN is able to estimate the system performance for a given mapping scheme of the threads selected to run next quantum. To identify the most advantageous mapping scheme to initiate for the next quantum, the DLM will utilize the DNN to make separate predictions for all possible mapping schemes given the selected threads and then choses the scheme that results in the highest estimated system performance.

Thread statistics and parameter engineering. It is important to carefully determine the appropriate set of thread statistics that characterize thread behaviors and will be used as input parameters to our system performance predictor. This process, otherwise known as parameter engineering, is critical since the accuracy of the system predictor depends upon the ability of the neural network to find causal relationships between these inputs and the expected output.

Normalizing the statistics into ratios helps to achieve parameter generalization. Using ratios instead of real values such as generating an instruction mix where each instruction type is given as a ratio of the total instructions executed during the last quantum helps to achieve this generalization. Without using this type of normalization, we would be left with inconsistent statistical input to the DNN performance predictor. For example, 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 generalization can also be used in cases when the core types have different ISAs or cache configurations. Generalizing statistics enables our approach to be useful in systems with a variety of different architectures.

In determining the final set of statistics, we sought to balance DNN predictor accuracy while minimizing the overheads due to gathering the statistics and the arithmetic operations needed to be performed. Based upon the heterogeneous system used in our work (detailed in Sect. 4.1), we identified 12 different thread statistics that are useful in describing thread behaviors on the cores and are inclusive of thread interference effects. The statistics are collected after a thread completes an execution quantum and are composed of the accesses and misses of the different structures of the cache hierarchy as well as the instruction mix executed. These 12 thread statistics (given as ratios) are: **(1)** DL1, **(2)** L2, and **(3)** L3 data cache miss ratios, instruction mix ratios including **(4)** loads, **(5)** stores, **(6)** floating point operations, **(7)** branches, and **(8)** generic arithmetic operations, **(9)** IL1 divided by DL1 loads, **(10)** L2 divided by DL1 misses, **(11)** L3 divided by DL1 misses, and **(12)** L3 divided by L2 misses.

The 12 statistics are saved as part of a thread's context after each quantum it executes, overwriting the values from the previous quantum. Many conventional CPUs come with hardware support for collecting similar statistics and in future work we will seek to further explore the set of statistics needed in order to mitigate collection and processing overheads while maintaining or improving the accuracy of our performance predictor.

Next quantum thread behavior predictor (NQP). Several novel approaches have been proposed which predict program behavior based on various statically or dynamically collected program statistics [7,26]. However, to keep overheads low and for simplicity, we use a next quantum thread behavior predictor (NQP) that always predicts the next behavior to be the same as the immediately anterior quantum behavior. The statistics forwarded by the NQP, therefore, are based on those collected during the thread's previous execution quantum.

Figure 3 helps to visualize the behavioral periodicity which the NQP must predict for. It shows the IPC variability of the perlbench and gamess benchmarks throughout their simulated execution on an Intel Nehalem x86 using a 1ms execution quantum. 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.

We measured the NQP accuracy results using the mean percentage error for the SPEC2006 benchmark suite. These applications were simulated executing on an Intel Nehalem x86 configuration using a 1ms execution quantum. The errors are calculated using Eq. 1 by measuring the IPC differences from quantum to quantum.

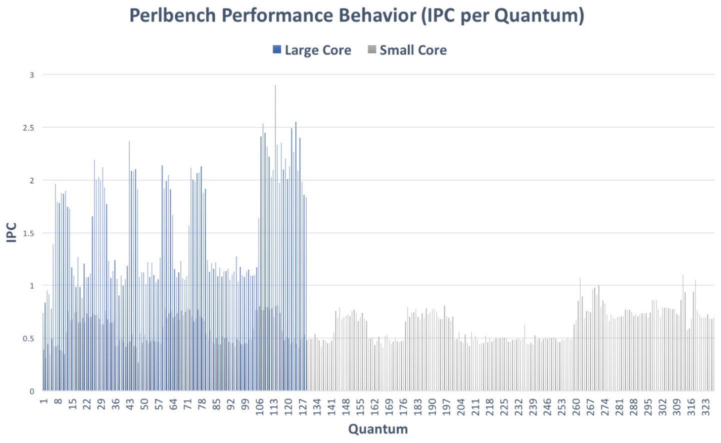
$$\begin{aligned} error_i &= \frac{|y_i - t_i|}{t_i} \\ \mu_{error} &= \frac{1}{n} \times \sum_{i=1}^{n} error_i \end{aligned} \tag{1}$$

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 NQP results in average errors of 10% for all SPEC2006 applications on both cores. However, the results vary between individual benchmarks with some outliers (e.g. cactusADM and soplex) exhibiting higher errors. These error variations can have a significant impact on the ability of the DNN predictor to properly predict and maximize system throughput.

DNN system performance predictor. The key component behind the DLM is a DNN system performance predictor which takes as input a set of parameters from as many individual threads as there are hardware cores and then outputs an estimated system IPC value. The system we target is a heterogeneous CPU architecture composed of 4 cores with 2 different core types (1 large core and

(a) The IPC per quantum behavior of perlbench.



(b) The IPC per quantum behavior of gamess.

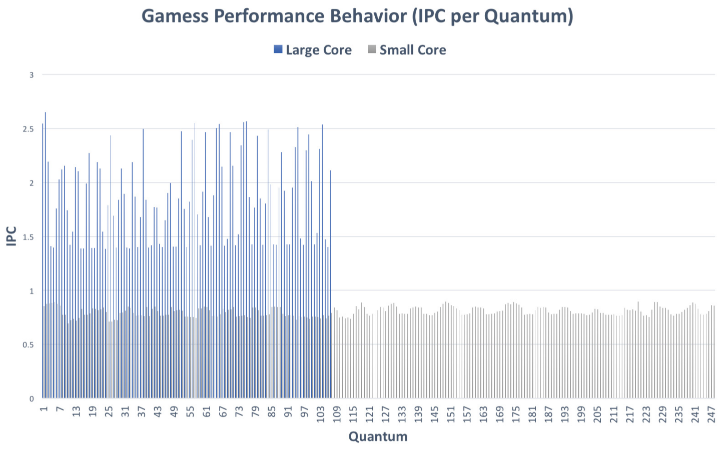
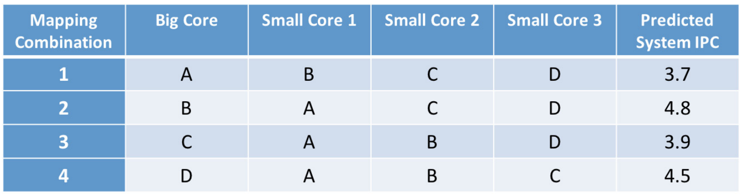


Fig. 3. The IPC per quantum behavior of four SPEC benchmarks when running on the large core compared to the small core.

3 small cores, described in Sect. 4.1). The DNN predictor takes as input the 12 normalized parameters from the 4 threads (selected for execution by the CS) for a total of 48 input parameters.

The order in which the threads are inputted to the DNN correspond to which physical core they would be mapped to with the first 12 thread parameters as corresponding to the thread mapped to the large core, and the next 36 parameters corresponding to the threads mapped to the three small cores. This way, we are able to estimate what the system IPC would be for different mapping combinations.



Mapping Combination	Big Core	Small Core 1	Small Core 2	Small Core 3	Predicted System IPC
1	A	B	C	D	3.7
2	B	A	C	D	4.8
3	C	A	B	D	3.9
4	D	A	B	C	4.5

Fig. 4. An example of how the DLM uses the DNN to predict for 4 different mapping combinations once it is passed the 4 threads selected by the CS (A, B, C, and D).

An example of this is given in Fig. 4. Here the CS has selected 4 threads (A, B, C, and D) from a larger pool of available threads to execute next quantum. There are 4 different combinations which we can map the 4 threads onto the hardware where each combination will have a different thread mapped onto the large core. The different mapping combinations represent the different ordering of the thread parameter inputs to the DNN. For instance, combination 1 will have the first 12 inputs correspond to thread A, the next 12 to thread B and so on. We can also consider all mapping permutations but since the only shared structure is the L3, there should be negligible differences in performance and interference effects. In the example, the DNN predictions for the 4 different combinations are given in the last column. Combination 2 has the highest estimated system and will be chosen as the optimal mapping scheme for the upcoming quantum.

We have implemented the DNN performance predictor using Python and the machine learning library scikit-learn [20]. An extensive exploration into the DNN architecture was conducted before settling upon the chosen design. Due to space concerns and the objective of this work being the proof of concept of the DLM, only a brief summary of the DNN design study is provided here.

Once the 12 input parameters were chosen, we evaluated numerous DNNs by modifying the hyperparameters of each including using different numbers of hidden layers, hidden units, activation functions, and training regularization techniques. We sought to balance prediction accuracy with implementation feasibility and made use of learning curves to gain insight into how many training samples the DNN needs to start predicting consistently for unseen data and how accurate these predictions are. Each training data sample consists of 48 input parameters and 1 target system IPC value. These are collected after each scheduling quantum which has resulted in the execution of 4 threads on the 4 cores. The algorithm used for training is a stochastic gradient based optimizer with L2 regularization which is readily used in machine learning models. During training, the weights of the neural network are adjusted after each full iteration of a batch of training data, always aiming to minimize the mean square error (mse) between the predicted output and the target output.

At the end of the design study, we settled upon a DNN implementation consisting of 48 total inputs, 5 hidden layers of 25 hidden units each, and a single output unit that use a rectified linear activation function. Figure 5 plots the learning curves of the training and 10-fold cross-validation results of the chosen

DNN. It highlights how, as the quantity of training data grows, so too does the accuracy and generalizability of the predictor when executing all the applications from SPEC2006. The score is measured in terms of correlation between the predicted system performance and the observed system performance using an R^2 coefficient. In particular, the figure shows that after about 15000 quanta, the correlation between the predicted performance and the observed performance on the data used to train is very high (about 0.96) and after about 35000 quanta, the correlation of stabilizes for the unseen validation data at about 0.64. The difference between the training and validation curves illustrates that the model has high variance which may indicate overfitting but can be explored in future work by adding more regularization and fine tuning the input parameters, hyperparameters, and sample data. Since our model is capable of online learning, however, the prediction errors introduced by running new applications will gradually settle at lower levels after training dynamically. Online learning works by continuing to train our DNN periodically after a certain number of new data samples are gathered. For our online DNN implementation, we have chosen to keep training our predictor every 20 execution quanta (i.e. a micro-batch of 20 samples). This requires needing to save only 20 quantum samples of data a time. The frequency of online training is related to the average number of quanta the benchmarks take to complete. A larger micro-batch could be used for longer applications or when the system is exceedingly busy in order to lower overheads.

3.2 Overheads

Schedulers typically add overheads due to the mapping calculations and resulting context swaps after each scheduling quantum. Since the DLM is triggered 4 times as often as the CS (1 ms vs 4 ms quantum), the DLM can also cause context swaps before the next CS quantum. A minimum of 0 and maximum of 4 extra context swaps can be issued by the DLM before the next CS quantum. However, the DLM will only trigger a swap if the resulting mapping is beneficial to overall system performance. The overheads due to the NQP and performance predictor amount to less than 4000 floating point operations per predicted mapping combination and less than 16000 in total. However, not only can a large quantity of these calculations be done in parallel, but this overhead is still orders of magnitude less than it costs to swap contexts and load the caches. Online training also adds overheads but is only done after every 20 quanta (or the chosen frequency of micro-batch training) and can be hidden by running it in the background when a core is idle.

Storing the 64-bit weights of the DNN requires about 21 KB of memory. The introduction of new statistical fields to save for each thread is also a minor overhead (96 bytes per thread) as is the memory needed to store the online training data (¡8 KB for 20 samples of 4 threads' worth of parameters). Lowering these overheads is a topic for future work but are still reasonable for a viable implementation of the scheduling model.

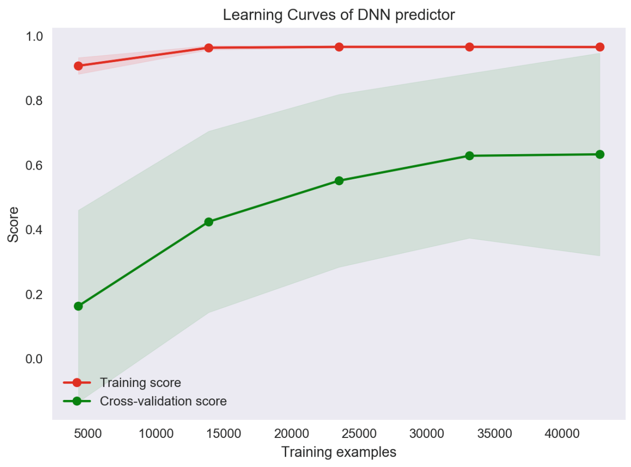


Fig. 5. The learning curve of the online DNN predictor. As the amount of training data increases the predictor becomes more generalized to account for different applications and behaviors. Higher y-axis numbers are better.

4 Evaluation

4.1 Methodology

This work uses the Sniper [3] 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 core types are based on the Intel Nehalem x86 architecture running at 2.66 GHz. Each core type has a 4 wide dispatch width, but whereas the large core has 128 instruction window size, 16 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, 8 cycle branch misprediction penalty (based on a one-bit history predictor), and a 6 entry load/store queue. The 1-large 3-small multi-core system configuration is based on the experimental framework used in previous work [15,27] that we evaluate our proposal against. 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 [9] benchmark suites to evaluate and train our scheduling model. This is an industry-standardized, CPU-intensive benchmark suite, stressing a system's processor and memory subsystem. The entirety of the benchmark suite is used with the exception of some applications which did not compile in our platform (dealII, wrf, sphinx3). All 26 benchmarks are run from start to finish and the simulation ends after all the benchmarks finish. This is done to emulate a busy system which must execute a diverse set of applications. This setup is also useful in demonstrating the ability of the DLM to improve system throughput.

We evaluate the performance for three different conventional schedulers (round-robin [15], fairness-aware [27], and CFS [19]) with and without the use of a fully trained DLM. That is to say we compare how much each conventional scheduler may be improved (in terms of system throughput) by using the DLM instead of its typical mapping mechanism. To account for context switch overheads due to architectural state swapping, we apply a 1000 cycle penalty which is consistent with the value utilized in the round robin study. The additional cache effects from the context switches are captured by the simulation.

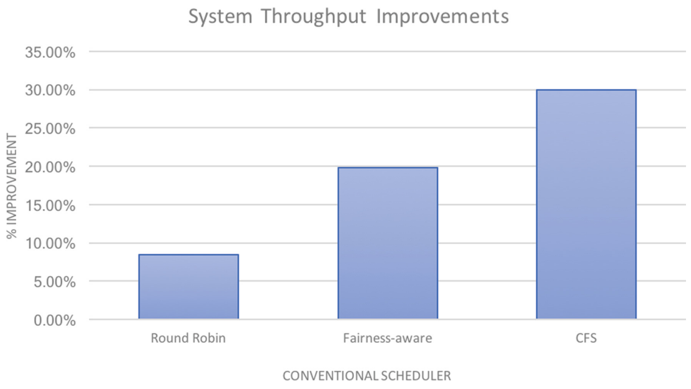


Fig. 6. Average system throughput (IPC) improvements when using the DLM for all SPEC2006. Higher numbers are better.

Figure 6 compares the system throughput improvements achieved for all 3 schedulers when using a DLM after running SPEC2006. The results show an average percentage throughput increase of 8%, 20%, and 30% for the round-robin, fairness-aware, and CFS schedulers respectfully. These improvements are significant especially for a preliminary study with a simple deep neural network predictor. They also highlight how effective the DLM is at benefitting all 3 different state-of-the-art schedulers. The improvements demonstrate the ability of the DLM to find more optimal mappings than the schedulers can by themselves. It achieves this thanks to two main factors. The DNN predictor allows the DLM to make highly accurate predictions for different mapping combinations while the 1ms quantum provides the opportunity to detect and adjust the mapping for variations in thread behaviors. The differences in the throughput gains for the 3 schedulers are also consistent with how they perform relative to one another without the DLM. On a heterogeneous system, the round-robin scheduler has been shown to perform better than the fairness-aware scheduler, which in turn performs better than the CFS.

The average total percentage prediction error of the DLM (calculated using Eq. 1) for the experiments was 12%. The DLM errors are slightly higher than those of the DNN shown earlier (see Sect. 3.1) because the DLM includes errors from both the NQP and the DNN. This error rate is within reasonable margins and our results are notable when considering that the DLM still showed such significant throughput benefits.

5 Related Work

Much of the previous work using machine/deep learning for scheduling has been to classify applications, as well as to identify process attributes and a program's execution history. This is the approach of [18] which used decision trees to characterize whole programs and customize CPU time slices to reduce application turn around time by decreasing the amount of context swaps. The work presented in [13] 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 s, use only a handful of benchmarks, and do not implement the predictor inside of a scheduler.

The studies by Dorronsoro and Pinel [6,21] investigates using machine learning to automatically generate desired solutions for a set of problem instances and solve for new problems in a massively parallel manner. An approach that utilized machine learning 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. [25]. Predicting L2 cache behavior is done using machine learning for the purpose of adapting a process scheduler for reducing shared L2 contention in [23].

In the work done by Bogdanski et al. [2], choosing parameters for task scheduling and loadbalancing is done with machine learning. However, their prediction is whether it is beneficial to run a pilot program that will characterize a financial application. They also assume that the computational parameters of the workload stay uniform over certain periods of time. Nearly all of these approaches deal with either program or process level predictions and target homogeneous systems.

Characterizing and exploiting program behavior and phases has been the subject of extensive research. Duesterwald et al. [7] and Sherwood et al. [24] showed that programs exhibit significant behavioral variation and can be categorized into basic blocks and phases which can span several millions to billions of instructions before changing. Work done in [26] has taken advantage of the compilers ability to statically estimate an applications varying level of instruction level parallelism in order to estimate IPC using monotonic dataflow analysis and simple heuristics for guiding a fetch-throttling mechanism.

A heterogeneous system containing various cores of the same ISA but of different types was proposed by Kumar et al. in [11]. Their process consists of deciding on the core that will perform in the most power efficient manner each time a new phase or program is detected using sampling techniques. Moncrieff et al. [17] and Menasce and Almeida [16] 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 and Yang [14]. A separate study [1] aims to create a contention-aware scheduler that maximizes throughput by learning and mimicking the decisions of an oracle scheduler.

Chronaki et al. [4,5] 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. A statistical method using extreme value theory is used in [22] to determine the probabilities for optimal task assignment in massively multithreaded processors.

6 Future Work and Conclusion

In this paper we have presented a preliminary study which pioneers applying DL to heterogeneous scheduling. We outlined a scalable scheduling model that decouples thread selection and mapping routines. The thread selection mechanism of a conventional scheduler is used in conjunction with a deep learning mapper (DLM) to maintain fairness and increase system performance. The DLM uses a deep neural network to predict the system performance for different mapping options at the scheduling quantum granularity. This lightweight deep neural network can provide highly accurate predictions for a diverse set of applications while continuing to train dynamically. The validation of our approach shows that even a simple DL based mapper can significantly improve system performance for state-of-the-art schedulers by 8% to 30% for CPU and memory intensive applications.

We seek to expand the scope of our work in the future by further exploring thread behavioral statistics, alternative DL models, improving the NQP, and scalability issues. We would also like to study ensemble models could be used to further widen the scope of the DLM for dealing with irregular applications.

We hope that the novelty of this work has helped to highlight the value that using deep learning can offer towards improving system performance.

Acknowledgments. This work has been supported in part by the European Union (FEDER funds) under contract TIN2015-65316-P.

References

1. Anderson, G., Marwala, T., Nelwamondo, F.V.: Multicore scheduling based on learning from optimization models. Int. J. Innov. Comput. Inf. Control **9**(4), 1511–1522 (2013)
2. Bogdanski, M., Lewis, P.R., Becker, T., Yao, X.: Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. In: 2011 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 496–501. IEEE (2011)
3. Carlson, T.E., Heirmant, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. IEEE (2011)
4. Chronaki, K., Rico, A., Badia, R.M., Ayguade, E., Labarta, J., Valero, M.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 329–338. ACM (2015)
5. Chronaki, K., et al.: Task scheduling techniques for asymmetric multi-core systems. IEEE Trans. Parallel Distrib. Syst. **28**(7), 2074–2087 (2017)
6. Dorronsoro, B., Pinel, F.: Combining machine learning and genetic algorithms to solve the independent tasks scheduling problem. In: 2017 3rd IEEE International Conference on Cybernetics (CYBCON), pp. 1–8. IEEE (2017)
7. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. In: 12th International Conference on Parallel Architectures and Compilation Techniques, PACT 2003, Proceedings, pp. 220–231. IEEE (2003)
8. Greenhalgh, P.: big.little processing with arm cortex-a15 & cortex-a7 (2011). http://www.arm.com/files/downloads/bigLITTLE_Final_Final.pdf
9. Henning, J.: SPEC CPU2006 benchmark descriptions. In: Proceedings of the ACM SIGARCH Computer Architecture News, pp. 1–17 (2006)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
11. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, Proceedings, pp. 81–92. IEEE (2003)
12. LeCun, Y., Kavukcuoglu, K., Farabet, C.: Convolutional networks and applications in vision. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 253–256. IEEE (2010)
13. Li, C.V., Petrucci, V., Mossé, D.: Predicting thread profiles across core types via machine learning on heterogeneous multiprocessors. In: 2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 56–62. IEEE (2016)
14. Liu, J.W., Yang, A.T.: Optimal scheduling of independent tasks on heterogeneous computing systems. In: Proceedings of the 1974 Annual Conference, vol. 1, pp. 38–45. ACM (1974)
15. Markovic, N., Nemirovsky, D., Milutinovic, V., Unsal, O., Valero, M., Cristal, A.: Hardware round-robin scheduler for single-ISA asymmetric multi-core. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 122–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_10

16. Menasce, D., Almeida, V.: Cost-performance analysis of heterogeneity in supercomputer architectures. In: Proceedings of Supercomputing 1990, pp. 169–177. IEEE (1990)
17. Moncrieff, D., Overill, R.E., Wilson, S.: Heterogeneous computing machines and Amdahl's law. Parallel Comput. **22**(3), 407–413 (1996)
18. Negi, A., Kumar, P.K.: Applying machine learning techniques to improve Linux process scheduling. In: TENCON 2005, 2005 IEEE Region 10, pp. 1–6. IEEE (2005)
19. Pabla, C.S.: Completely fair scheduler. Linux J. **2009**(184), 4 (2009)
20. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
21. Pinel, F., Dorronsoro, B.: Savant: automatic generation of a parallel scheduling heuristic for map-reduce. Int. J. Hybrid Intell. Syst. **11**(4), 287–302 (2014)
22. Radojković, P., Čakarević, V., Moretó, M., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M., Valero, M.: Optimal task assignment in multithreaded processors: a statistical approach. ACM SIGARCH Comput. Architect. News **40**(1), 235–248 (2012)
23. Rai, J.K., Negi, A., Wankar, R., Nayak, K.: A machine learning based meta-scheduler for multi-core processors. In: Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts, pp. 226–238. IGI Global (2012)
24. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003)
25. Shulga, D., Kapustin, A., Kozlov, A., Kozyrev, A., Rovnyagin, M.: 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, pp. 345–348. IEEE (2016)
26. Unsal, O.S., Koren, I., Khrishna, C., Moritz, C.A.: Cool-Fetch: a compiler-enabled IPC estimation based framework for energy reduction. In: Eighth Workshop on Interaction between Compilers and Computer Architectures, INTERACT-8 2004, pp. 43–52. IEEE (2004)
27. Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A., Eeckhout, L.: Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, pp. 177–188. IEEE Press (2013)