DeepP: Deep Learning Multi-Program Prefetch Configuration for the IBM POWER 8

Manel Lurbe, Josué Feliu, Salvador Petit, *Member, IEEE*, Maria E. Gómez, and Julio Sahuquillo, *Member, IEEE*

Abstract—Current multi-core processors implement sophisticated hardware prefetchers, that can be configured by application (PID), to improve the system performance. When running multiple applications, each application can present different prefetch requirements, hence different configurations can be used. Setting the optimal prefetch configuration for each application is a complex task since it does not only depend on the application characteristics but also on the interference at the shared memory resources (e.g., memory bandwidth). In his paper, we propose *DeepP*, a deep learning approach for the IBM POWER8 that identifies at run-time the best prefetch configuration for each application in a workload. To this end, the neural network predicts the performance of each application under the studied prefetch configurations by using a set of performance events. The prediction accuracy of the network is improved thanks to a dynamic training methodology that allows learning the impact of dynamic changes of the prefetch configuration on performance. At run-time, the devised network infers the best prefetch configuration for each application and adjusts it dynamically. Experimental results show that the proposed approach improves performance, on average, by 5.8%, 6.7%, and 15.8% compared to the default prefetch configuration across different 6-, 8-, and 10-application workloads, respectively.

Index Terms—IBM POWER8 processor, prefetch configuration, inter-application interference, machine learning, deep learning

1 Introduction

Hardware data prefetching is a speculative technique that fetches data in advance to the processor. Thanks to its performance benefits, especially in the so called *prefetch friendly* applications, this technique has been widely applied to hide the long memory latency and improve the system performance both in single-thread processors [1], [2], [3], [4], [5] and in multicores [6], [7], [8], [9], [10], [11].

Current high-performance processors are deployed with a set of configurable prefetches aimed at capturing different memory behaviors. The IBM POWER family of processors implements the most complex and powerful prefetchers deployed in current servers. These processors allow the user to update the prefetch setting at run-time, which is especially challenging in the case of multicores, where main memory bandwidth contention can become a severe performance bottleneck. Recent research has focused on dynamically selecting the best prefetch configuration for the running workload at run-time. Some works have concentrated on parallel workloads [12], [13], [14] and other approaches target multi-program single-threaded workloads [15] which is particularly complex because each application has different memory characteristics and prefetch requirements.

Basically, these approaches rely on run-time *heuristics* to select the best prefetch setting for the running workloads. These heuristics measure hardware events of the target platform, analyze how their values correlate to performance at run-time, and based on experimental observations, they look into for insights about undesired and desired behavior. For instance, it can be found that when a given metric (e.g., the consumed memory bandwidth) exceeds a given threshold, the overall performance drops. To avoid this scenario, the approach would opt for reducing the prefetch aggressiveness when the aggregate bandwidth consumption is above that threshold. Heuristic-based approaches present two major drawbacks. On the one hand, the lack of flexibility. Heuristic approaches rely on a set of thresholds whose values are obtained experimentally through a wide set of "trial and error" experiments. Threshold values depend on the workload and underlying hardware. Thus, these approaches select the values that work well for the largest number of experimental workloads. On the other hand, heuristics require costly sampling when multiple settings are evaluated which can hurt performance and become prohibitive. In fact, to be practical, heuristic-based approaches only consider few metrics (e.g., IPC and memory bandwidth consumption) that require monitoring a minimal set of hardware events. An alternative option to deal with this drawback is the use of artificial intelligence techniques such as deep learning as used in this work.

• *Manel Lurbe, Salvador Petit, Maria E. Gómez, and Julio Sahuquillo are with the Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, 46022 València, Spain. E-mail: malursem@inf.upv.es, {spetit, megomez, jsahuqui}@disca.upv.es.*
• *Josué Feliu is with the Departmento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia, Spain. E-mail: josue.f.p@um.es.*

Manuscript received 24 Aug. 2020; revised 15 Dec. 2021; accepted 22 Dec. 2021. Date of publication 4 Jan. 2022; date of current version 8 Sept. 2022.
This work was supported in part by Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, in part by Generalitat Valenciana under Grant AICO/2021/266, and in part by Ayudas Contratos predoctorales UPV - subprograma 1 (PAID-01-20). The work of Josué Feliu was supported by a Juan de la Cierva Formación Contract under Grant FJC2018-036021-I.
(Corresponding author: Manel Lurbe.)
Recommended for acceptance by J. C. Hoe.
Digital Object Identifier no. 10.1109/TC.2021.3139997

Deep neural networks act as universal approximators and allow modeling non-linear relationships from a large amount of data. In other words, thanks to the capability to identify non-linear relationships between performance events, deep learning can be used to help some microarchitectural components (e.g., the branch predictor, or the cache replacement algorithm) to improve its performance, and hence the overall system performance.

Some tentative works have focused on machine learning to improve the prefetch configuration of commercial processors [16], [17], [18]. The works deal with single parallel applications running in Intel and IBM processors. However, to the best of our knowledge, no work has focused yet on multi-program workloads.

In this paper, we propose *DeepP* (Deep learning Prefetching) which, to the best of our knowledge, is the first neural network-based proposal aimed at estimating the performance of an application when running on a multi-program workload depending on its prefetch configuration. More precisely, our proposal uses the neural network to predict, for each running application, its expected IPC for each prefetch configuration with the aim of selecting the best performing prefetch setting dynamically at run-time. Compared to heuristic-based solutions, the proposed approach also gets rid of exploration phases and specially-tuned hardware thresholds.

To build a neural network, samples extracted from *training* executions are required. These samples are typically obtained keeping the same prefetch configuration across the entire execution of a given benchmark [17]. We refer to this type of training as *static-prefetch* training. Static-prefetch training requires additional executions to explore different prefetch configurations. Moreover, this type of training cannot catch the effect of changing the prefetch configuration at run-time. To address these problems, we propose a methodology that includes one or more *dynamic-prefetch* trainings. In a dynamic-prefetch training, a previously trained network is used to drive additional training executions to get more samples that help learn the effect of modifying the prefetch configuration dynamically. This enables the final neural network to adapt to dynamic changes along execution, leveraging the opportunities when a change of the prefetch configuration can improve performance.

In this work, we devise two neural networks. The first, *single-core* network, focuses on the execution of single applications and is built as a first step to explore the potential of neural networks to select the best prefetch configuration. The second, *multi-core* network, also takes into account the inter-thread interference at the shared resources among co-running applications. More precisely, it considers the memory bandwidth consumed by the co-runners, shown recently as the main critical shared resource for prefetching in the IBM POWER8 [15].

This paper makes three main contributions:

- We demonstrate that a neural network can be trained to predict the performance (IPC) of co-running applications depending on the prefetch configuration with high accuracy and minimal overhead regardless of the number of applications, which allows scaling in performance with the number of applications.
- We propose a multi-program aware neural network for the IBM POWER8 that establishes a correlation between inter-application interference and performance by taking as input the aggregate memory bandwidth consumption of co-runners.
- We devise a training methodology that does not only use executions with fixed prefetch configurations (i.e., static-prefetch training) but also includes dynamic-prefetch training as a way to improve traditional training methodologies.

The experimental results that DeepP provides performance gains, on average, of 5.8%, 6.7%, and 15.8% across a set of random 6-, 8-, and 10-application workloads compared to the default prefetch configuration. We would like to remark that the proposal can be adapted to other systems by adjusting intra-core and interference-related events as well as the prefetch configurations.

2 Related Work

2.1 Heuristic Based Solutions

Some recent research [12], [13], [14], [15], [19], [20] has focused on heuristic-based solutions with the aim of achieving the best performance from the prefetching capabilities of commercial machines. All these works focus on IBM POWER processors, whose prefetcher is much more powerful and complex, and further performance gains are expected.

In [19], authors propose AREP (Adaptive Resource-Efficient Prefetching). This work focuses on finding a single best prefetching configuration that maximizes throughput for multi-core scenarios. This solution explores different prefetch configurations of the intel i7 2600K processor. In [20], a spatial bit-pattern prefetcher is proposed, which uses memory bandwidth utilization to adjust prefetch aggressiveness on Intel Skylake processors, its performance improvements are directly related with the available memory bandwidth. In [12], authors propose a strategy to select at run-time the best prefetch setting with the goal of enhancing the performance of parallel workloads when running a single application on an IBM POWER8 processor. In contrast, the strategy proposed in [13] is aimed at selecting the best prefetch setting for single-threaded multi-program workloads in the IBM POWER7 processor. This approach, however, does not scale with the number of cores. This shortcoming is addressed by IBS [14] in a simple way by activating or deactivating the prefetcher of each individual core in the IBM POWER7. However, unlike approach, IBS only considers two prefetch settings: the most aggressive one and disabling the prefetcher.

Recently, Navarro *et al.* [15] consider different levels of aggressiveness and the inter-thread interference to propose a much more scalable and best performing approach for the IBM POWER8. All these works require exploration phases in order to check the performance of each studied configuration and select the best performing one. In addition, some thresholds need to be defined that act as a filter to apply the devised strategy.

2.2 Machine Learning Based Solutions

Research works using machine learning can be mainly classified in two main categories depending on whether they pursue to improve the hardware prefetcher architecture, or to dynamically select the best prefetch setting in commercial processors.

Regarding the first group, a new prefetch architecture is modeled and evaluated using simulation tools. ML can be used as a tool to dynamically identify patterns in data streams and capture correlations [18], [21], [22]. This way allows identifying the memory access patterns more precisely, so improving the prefetching accuracy.

Regarding the second group, some ML based techniques have been proposed to configure the hardware prefetchers at run-time in commercial machines [16], [17], [23]. Some of these works, like [23] and [16], focus on Intel processors. These processors implement an small set (typically four: two in the L1 cache and two in the L2 cache) of independent on-off prefetchers. Since these prefetchers are independent and orthogonal, the problem lies on selecting which ones should be turned on and which ones turned off. In [23] a decision-tree with thresholds on the hardware events is built to decide which ones of the four prefetchers should be activated.

In [16], authors propose a strategy for effective prefetching based on machine-learning for parallel applications (i.e., PARSEC benchmarks). The approach trains several machine-learning algorithms to select which one of the four built-in prefetchers should be used (i.e., switched on or switched off).

Finally, the work in [17], similarly to our work, addresses the dynamic tuning of the best prefetch knobs in an IBM POWER8 processor but, unlike this work, they focus on individual parallel workloads. In other words, the work concentrates on individual applications, thus there is not inter-application interference. Typically, the threads of an individual parallel application have an homogeneous behaviour, which simplifies the approach since a single prefetching configuration is adequate for the whole application (i.e., all threads share the same prefetcher configuration). This is not the case of multi-program workloads where each application exhibits a different behaviour and can interfere in the prefetch behaviour of the others. Other ML approaches for other computer components can be found in [24] [25] [26] [27] [28] [29].

Related research work has also employed ML and performance counters pursuing alternative goals. For instance, in [30], authors use ML to model the correlation between performance counters and IPC in bug-free microarchitectures to identify performance bugs in new designs and in [31] authors use decision trees to identify execution phases from performance counters and estimate the performance of each phase using linear regression.

2.3 Other Prefetching Engines

Other approaches have focused on devising a new prefetching engine. Domino prefetcher [32] is a temporal data prefetching technique that logically looks up the history with both 1 and 2 last miss addresses to find a match for prefetching. This work aims to improve the effectiveness of temporal prefetching techniques. Bingo prefetcher [33] proposes a spatial data prefetcher in which short and long latency events are used to select the best access pattern for prefetching. Access Map Pattern Matching [34] detects multiple prefetch candidates from pattern matching of memory access footprints. ISB [35] is a prefetcher that targets irregular sequences of temporally correlated memory references, in other words, a prefetcher that combines address correlation with PC localization. SPP [36] presents a solution that can read prefetch acces patterns using a history signature, then it is able to detect when a data access pattern crosses a page boundary, and quickly resume prefetching on the new page, and is able to balance agressive prefetching using path confidence. This implementation doesn't use program or core counters, only with OS physical address space, but only improves performance for the state of the art over 6.4%. Other approaches can be found in [37] [38] [39].

3 Background on Neural Networks

Artificial Neural Networks (NNs) are computing systems resembling biological neural networks. Artificial neural networks are composed of artificial neurons or *nodes*, which perform a computation that is based on the behavior observed in neurons of a biological brain. Typically, the computation performed by a node in a NN follows the equation:

$$output = activation\_function\left(b + \sum_{i=1}^{n} x_i \times w_i\right) \quad (1)$$

That is, the output of a node is calculated by adding the different inputs x_i multiplied by their corresponding weights w_i and the bias b. Then, the result of the sum is fed to an *activation function*. The purpose of this function is twofold. On the one hand, it determines if the output must progress as an input to the next node. On the other hand, it normalizes the sum result in a range, which is commonly between 0 and 1 or -1 and 1.

In an artificial neural network, nodes are organized in rows, referred to as layers. A node in a layer gets its weighted inputs from the previous layer and its outputs act as inputs to the next layer. The first layer takes the inputs of the NN, and is known as the *input layer*, while the last layer provides the outputs of the NN, and is referred to as the *output layer*. In between both layers, there can be several *hidden layers*. A set of input values for the input layer is referred to as a *dataset*. Artificial neural networks are usually used to extract abstract features of the dataset that can be used to categorize it by a given classification.

A deep neural network or *deepnet* is a type of NN that includes multiple hidden layers between the input and the output. Each hidden layer is in charge of extracting a distinct group of dataset features, which are fed as inputs to the next layer. In this way, each hidden layer analysis is based on the features of the dataset found by the previous layers, and therefore, the analysis can be more complex. For example, the first layers of the neural network can be in charge of detecting if the input image is a portrait of a person, while later layers can identify the person among a set of possible matches. This, for instance, is the basis of advanced smart photo albums.

To detect the features of a dataset, neural networks are trained with dataset samples. This process is known as *learning* and involves adjusting the weights w_i and biases b for each NN node to modify the NN output. Learning can be *supervised* or *unsupervised*. In this work, we focus on supervised learning, which means that each sample includes a *label* that represents the correct NN output for the sample. The output of an untrained neural network for a given sample highly differs from the corresponding label. The difference between the output and the label (i.e., the *error*) is passed to a *cost function*. A typical cost function is the mean-squared error, which is the average squared error between the NN output and the label for each trained sample.

Learning is an iterative process whose goal is to minimize the result of the cost function. In each step of this process, the neural network is fed with a different sample. Then, the NN output is compared to the label to obtain the error, and the cost function is calculated. After a given number of samples, a method known as *backpropagation* is used to adjust the weights w_i and biases b for each node to compensate the errors and minimize the cost function. The iterative process finishes when a local minimum of the cost function is found in the trained sample space.

Once the neural network has been trained, it can be used to estimate or *infer* the label for unlabeled datasets. This is known as *inference*. If the NN has been properly trained, then the NN will be capable to provide a label for the dataset that matches or falls close to the real one.

4 Experimental Platform

All the experiments in this work have been performed using an IBM Power System S812L system, with a 10-core IBM POWER8 processor [40]. Each core runs at 3.69 GHz. The processor implements a three-level cache hierarchy. The first level (32KB L1 I-cache and 64KB L1 D-cache) and the second level (L2 512KB) are private per core, and the L3 80MB cache is shared among all the cores. The system has installed a single 32GB DRAM module and runs Ubuntu 18.04 with the Linux kernel 4.15.

4.1 The IBM POWER8 Prefetch Engine

The IBM POWER8 implements one of the most sophisticated prefetch engines available in recent commercial processors. This prefetcher engine includes a stream prefetcher that tracks streams by their effective address. This prefetcher can be set through a 25-bit special purpose register, namely *Data Streams Control Register* (DSCR), that admits up to 2^{25} prefetch configurations.

The 25 bits are divided into twelve fields [40]. Some fields are single-bit and allow enabling or disabling a particular feature of the prefetcher. Other fields contain two or three bits that allow tuning the level (aggressiveness) of the field. The values of all these twelve fields together set up the prefetch configuration. The impact of the fields on the system performance widely varies among them; moreover, the impact of a given field depends not only on its value but on the remaining enabled fields.

Recent work [17], [12], [15] found that *depth* and *urgency* are the fields that impact on performance the most The former configures the level of the prefetch depth and is related to the number of cache lines that are brought into the on-chip caches. *Urgency* indicates how quickly this depth can be reached. A high urgency value indicates that the cache line should be prefetched as close as possible to the processor (i.e., into L1 or L2 cache).

Both fields have three bits (ranging from 0 to 7), which allow setting the target level. Setting this field to zero represents the default configuration in both fields and is equivalent to a *medium* (4) depth or urgency. Setting the depth to 1 disables prefetching and values from 2 to 7 configure different levels of depth from shallowest to deepest. Urgency levels vary from 1 (not urgent) to 7 (most urgent).

The names used in this work to refer to the studied prefetch configurations follow the UXPY scheme, where X and Y refer to urgency and depth, respectively. In particular, we study the configurations U1P2, U1P4, U1P7, U2P4, U2P7, U4P2, U4P7, U7P7. In addition, we also study the OFF (prefetcher disabled), and DEF (default) configurations.

4.2 Hardware Events and Performance Counters

Most of the inputs of the neural networks devised in this work are or are based on hardware events' counts gathered with performance counters. To configure and read these counters, we have used the *libpfm4* library [41].

The experimental processor allows monitoring among more than one hundred events. However, only six events (i.e., the number of hardware counters) can be monitored in a single period of time or *quantum*. Moreover, two of these counters are dedicated (i.e., they cannot be configured) to measure the number of executed cycles an committed instructions, which means that just four additional events can be monitored at a time. If the number of events to be monitored is higher than four, a possible solution is to distribute the monitoring along multiple quanta, each quantum gathering a subset of the target events. For instance, in order to monitor 22 events (including cycles and instructions) 5 quanta $((22 - 2)/4)$ are required. In other words, to read n events m intervals are required so that $n = 2 + m \times 4$.

Considering the low number of hardware events that can be monitored in a single quantum, training a NN with samples containing a large amount of events requires these samples to be collected through a high number of quanta. For instance, to gather the values corresponding to more than one hundred events, at least 25 quanta will be required. This fact has an important downside. Since the application behavior (and thus its performance) is likely to change in such a long execution interval, it would be unlikely to get a coherent sample because the hardware events gathered at the beginning of the sampling would be stale at the end. Therefore, a NN considering a high number of events would be useless to configure prefetching at run-time.

A possible solution to face the previous problem is to reduce the duration of the quantum. However, in our experiments we observed that measurements taken in short intervals are less stable and prone to suffer deviations (i.e., sudden peaks or drops). Another alternative approach to reduce the measurement time is using *event multiplexing*. This method, supported directly by *libpfm4*, allows the same counter to capture several hardware events in the same quantum by sampling the events for short periods of time and scaling the gathered values to the quantum length. However, this approach presents the same problems as reducing the quantum length.

Therefore, we discarded both alternative approaches and choose a typical 200ms quantum duration. Taken these observations into account, the devised approach requires a careful selection of events to avoid increasing too much the measurement time. The selected events will be discussed in Section 6.2.

5 Experimental Methodology and NN Design

To carry out the experiments, we implement a user-level manager that: i) performs process handling, ii) monitors performance counters, iii) queries a prefetch configuration approach (e.g., our devised NNs or an heuristic approach) for the best predicted prefetch settings, and iv) configures the prefetch engine according to the prediction.

The applications launched in the experiments are taken from the SPEC CPU2006 and SPEC CPU2017 suites. Depending on the specific experiment, an application can be executed in isolation or jointly with other co-running applications in a multi-program workload. Each application is allocated to a different physical core in multi-program workloads to avoid the SMT interference effect.

5.1 Manager Execution Flow

To run a multi-program mix, the manager launches its applications and maps each one to a different core. We set the execution of each application to the number of instructions it commits when running 180 seconds in isolation with the default prefetch configuration. The number of instructions that each application commits in this period is referred to as the target number of instructions for that application. Throughout all the paper, we assume applications complete when they commit their target number of instructions regardless of their prefetch configuration. When an application completes, the manager records its performance and then relaunches such application. This way ensures the workload is constant along its entire execution and allows all benchmarks to be evenly represented in the performance metric. The workload execution ends when all the applications in the workload have completed.

In the first quantum of the workload execution, the manager sets the default prefetch configuration for all applications.

After this quantum, the manager enters in a loop, which it repeats until the workload completes. The main loop has two parts. During the first part the manager collects, for each application, the values of the inputs required by the prefetch configuration approach to predict the best prefetch setting for the next quantum i. As explained above, this implies employing the m previous quanta to gather the inputs $I_{i-1}, I_{i-2}, \ldots I_{i-m}$, or simply $I_{i-1 \ldots i-m}$.

In the second part, for a given application, the input set $I_{i-1 \ldots i-m}$ is fed to the prefetch configuration approach, which provides the prediction. If the predicted best setting for the application differs from the current one, then the prefetch configuration is updated.

5.2 Neural Network Design

To design the neural networks we have used the BigML platform [42]. This platform provides an API for uploading training data, analyzing the data to find relevant inputs for the neural network, aid in designing the network (e.g., choosing an appropriate number of layers), training, and generating inference code. We have uploaded the training data from benchmark executions to the BigML platform to train and design the networks evaluated in this work. After that, the inference code from the designed networks is downloaded to perform the inference in our experimental platform.

The proposed NN *infers* the best performing prefetch configuration for the next quantum i. More precisely, the NN infers the IPC_i that a target application will achieve in the next quantum i for a given prefetch configuration $DCSR_i$. This configuration is provided as input to the NN together with the inputs gathered in the m previous quanta $I_{i-1 \ldots i-m}$. Notice that multiple NN inferences, as much as studied prefetch configurations, need to be made to find out the best prefetch configuration for each application. Once all the inferences are made, then the prefetch setting with the maximum predicted IPC is applied to the target application in the next quantum.

We measured that invoking the NN to obtain an IPC prediction for a prefetch configuration just takes by 0.1ms in our experimental platform. Overall, in a 10-benchmark workload execution, the overhead accounts approximately for 0.5% of the execution time; thus the incurred overhead can be considered negligible. This overhead can be further reduced in a production environment since the IPC predictions can be performed in parallel in different cores.

6 Learning Methodology

This section discusses the methodology followed to build the NN model, which covers three main issues:

- Training benchmarks.
- Model inputs.
- Model training methodology.

In this work, we present two NNs, the first one, a more simple NN, focuses on the execution of single threaded applications in a single core. The second one and more complex NN, predicts the performance of an application considering both its prefetch configuration and the interference caused by the applications co-running on other cores of the system.

In spite of the complexities of the devised NNs differ, the training methodology for both NNs is rather similar. The following subsections explain the three steps of the proposed methodology for the single-application single-core NN. After that, the last subsection explains the main differences to build the multi-core NN.

6.1 Training Benchmarks

To obtain the data inputs to train the neural networks, a set of benchmarks needs to be used. These benchmarks should be representative for the goal of study. To help choose representative benchmarks, several approaches have been proposed [14], [15] that categorize applications depending on the prefetch behavior. In this work, we follow the approach proposed in [15], which classifies applications into three main categories. Below, we summarize these categories in order to make the paper self-contained:

- *Prefetch unfriendly applications.* This category groups benchmarks whose performance does not improve with any prefetch configuration compared to no prefetching. Fig. 1a presents an example (`gamess`) of this type of application.
- *Prefetch friendly applications.* Their performance improves when prefetching is enabled. This category is divided into two main subcategories. *Prefetch*

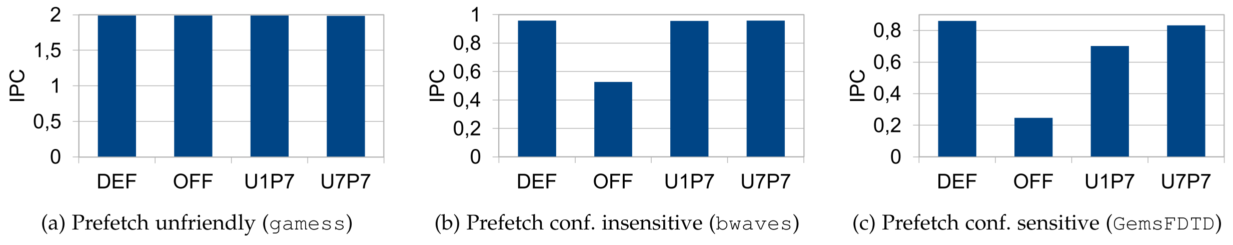


Fig. 1. Impact of prefetch configuration on performance of applications belonging to different prefetch sensitivity categories.

configuration insensitive. As long as prefetching is enabled, minor performance differences can be appreciated among the distinct prefetch configurations. This is the case of `bwaves` (Fig. 1b), whose performance improves due to prefetch, but the benefit is very similar across the studied prefetch settings. *Prefetch configuration sensitive*. Different prefetch settings result in different performance gains. An example of this type of application is `GemsFDTD` (Fig. 1c).

Since the neural network is used to predict the best performing prefetch configuration, the IPC of the selected benchmarks must be both sensitive to prefetching and to the prefetch configuration as well. Otherwise, we found that if the performance of a high percentage of the training applications is not affected by prefetching or the prefetch configuration, the training will not be able to find out a correlation between the prefetch settings and the IPC, resulting in poor NN output accuracy.

Due to this reason, in this work all the benchmarks we selected for training the single-core NN are prefetch friendly (both sensitive and insensitive to the prefetch configuration). These benchmarks are shown in Table 1. From those benchmarks a pool of samples was created taken from the execution quanta of all the applications. A random subset (80%) of this pool was used to train the NN and the remaining 20% was reserved to evaluate the NN accuracy. After that, six new applications have been used to validate the proposal. These six applications represent 20% of all applications (the remaining 80% are used for training and testing).

In contrast, when training the multi-core NN, as explained in Section 6.4, we consider all the types of benchmarks in order to consider the interference among them. This interference can mofify the benchmark behavior, for instance, a configuration insensitive benchmark in single core can move its behavior to configuration sensitive due to the inter-application interference in multi-core execution. This interference mainly rises in the main memory bandwidth contention, therefore this event has been considered as input of the multi-core NN.

TABLE 1
Prefetch Friendly Training Benchmarks

SPEC CPU2006	SPEC CPU2017
gcc	gcc_r
mcf	mcf_r
libquantum	xalancbmk_s
omnetpp	x264_s
xalancbmk	deepsjeng_r
bwaves	leela_s
milc	xz_r
zeusmp	cactuBSSN_r
cactusADM	lbm_r
leslie3d	imagick_r
soplex	namd_r
GemsFDTD	parest_r
lbm	povray_r

6.2 Model Inputs

Once the benchmarks and workloads to train the NN have been selected, we determine its inputs. These inputs refer to the hardware events that are considered by the neural network to infer the performance (IPC) of a prefetch configuration.

As explained in Section 4.2, a key design issue is the number of these events. If the number of the considered events is too high, the NN would require a noticeable number of quanta to gather them, which negatively affects sampling consistency. However, a too small number, the model could miss events of interest to output an accurate IPC, thus dropping prediction accuracy. We found that an amount of events to 16 hardware events, which can be gathered in just 4 quanta ($m = 4$), represents a good trade-off. This time interval is small enough to ensure that most samples are consistent along the interval while letting the model to achieve enough prediction accuracy.

The events that should be selected depend on the goal of study and the target machine. The purpose of this work is to select the best performing prefetch setting for each application, regardless of it runs individually (single-core NN) or it co-runs with other applications in other cores (multi-core NN). To this aim, we should select events that a variation in the event count represents a variation in the application's performance. Moreover, the selected events should also present differences among the distinct studied prefetch settings, in order to help discerning the best prefetch configuration for each application.

Taking these issues into account, we select three types of events:

- *Cache hierarchy performance*. These events measure the activity of the application in the cache hierarchy. This category basically includes (load and store) miss events along the cache hierarchy (L1D, L1I, L2, and L3).
- *Memory bandwidth*. These events are related to the main memory bandwidth consumption, since the available bandwidth affects the prefetch performance.
- *Core*. This category includes core-related events that help the model to discern if a given prefetch configuration can improve performance or not, hence improving the NN prediction accuracy. For instance, if the prefetcher is properly working, the number of

TABLE 2
Categories, Events Name, and Description

Category	Event Name	Description
Cache Hierarchy	e1. PM_L1_ICACHE_MISS	Demand iCache misses
	e2. PERF_COUNT_HW_CACHE_L1D:READ:MISS	L1 cache load misses
	e3. PERF_COUNT_HW_CACHE_L1D:WRITE:MISS	L1 cache store misses
	e4. PERF_COUNT_HW_CACHE_LL:WRITE:MISS	Last level cache store misses
	e5. PM_DATA_FROM_L3	The processor's data cache was reloaded from local core's L3 due to demand loads plus prefetches
	e6. PM_MEM_PREF	Main memory prefetch accesses
	e7. PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS	L1I cache prefetch accesses
	e8. PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS	L1 cache prefetch accesses
	e9. PM_L1_ICACHE_RELOADED_PREF	Counts all Icache prefetch reloads (includes demand turned into prefetch)
	e10. PM_DATA_FROM_MEMORY	The processor's data cache was reloaded from a memory location due to demand loads plus prefetches
	e11. LLC_LOADS	Last level cache loads
Memory Bandwidth	e12. PM_MEM_READ	Main memory accesses
	e13. LLC_LOAD_MISSES	Last level cache load misses
Core	e14. PM_DISP_HELD_IQ_FULL	Dispatch held due to issue queue full
	e15. PM_BR_MPRED_CMPL	Number of branch mispredicts
	e16. PERF_COUNT_HW_BRANCH_INSTRUCTIONS	Retired branch instructions

execution cycles where the issue queue is full (event PM_ DISP_HELD_IQ_FULL) will be reduced. As another example, the amount of branch mispredictions (event PM_BR_MPRED_CMPL) gives information of whether prefetches are triggered from wrong path instructions and thus are likely to be useless.

Table 2 summarizes the 16 events finally considered to build the neural network and their description, broken down into the three mentioned categories.

Different events were considered and discarded when they provided scarce or no contribution to prediction accuracy. Events were chosen in a 3-stage refinement process. In the first stage, a subset of 12 events (3-quantum intervals) were selected related to cache hierarchy, prefetch and core related performance (event e1 to e10, event 12, and event 15). This stage reached an average accuracy by 70%. In a second stage, we mainly checked core related events (e14 and e16) in addition to e13 to improve accuracy. This supposed extending the model to 4-quantum intervals but accuracy was improved up to 80%, that rose up to 85% with dynamic training. Finally, we added e11 and improved training. In between these stages, we checked the contribution to accuracy of other events. This stage improved accuracy in some individual applications up to 97%, and on average above 90%.

In addition to these events, we also take into account the number of instructions and cycles executed in the 4 previous quanta. As explained in Section 4.2, obtaining these values does not require additional programmable counters since the number of cycles and instructions is measured by dedicated counters that are always enabled. We consider these metrics because the purpose of the NN is to predict the IPC (that is, the relation between instructions and cycles) and we found that providing previous IPCs to the model helps to increase its accuracy.

Finally, previous quanta prefetch configurations are also used as inputs by the model to establish correlations between these configurations with the achieved performance and the discussed inputs.

6.3 Model Training Methodology

To predict the IPC for the next quantum i with a given prefetch configuration, say IPC_i and $DCSR_i$, the NN uses the inputs gathered in the 4 previous quanta $I_{i-1\ldots i-4}$. These inputs include both event-related inputs (which comprise the previous IPCs) and the enabled prefetch configurations during the previous quanta. Therefore, a training sample S_i consists of IPC_i, $DCSR_i$, and $I_{i-1\ldots i-4}$. That is, a training sample is defined as the set $S_i = \{IPC_i, DCSR_i, I_{i-1\ldots i-4}\}$. Since the aim of the NN is to predict the IPC, we use IPC_i as the label for training.

To train the NN, a large amount of training samples is required, which are gathered from specific executions designed with this aim that will be referred to as *training executions*. During these executions, the inputs and the achieved IPC are obtained to compose the samples. For example, assume a short execution consisting of 12 quanta. These quanta would be distributed in 3 four-quantum groups (since $m = 4$): $\{Q_1, Q_2, Q_3, Q_4\}$, $\{Q_5, Q_6, Q_7, Q_8\}$, and $\{Q_9, Q_{10}, Q_{11}, Q_{12}\}$. This execution would provide two training samples at the end of the two first four-quantum group $S_5 = \{IPC_5, DCSR_5, I_{1..4}\}$ and $S_9 = \{IPC_9, DCSR_9, I_{5..8}\}$.

Following the flow diagram depicted in Fig. 2, we launched a wide range of training executions. This flow is composed of two main training stages (dark boxes): *static-prefetch* training and *dynamic-prefetch* training.

In the first stage, referred to as *static-prefetch* training, each application is executed several times, as many as the number of studied prefetch configurations. In each execution, the prefetch is set to a given configuration and remains unchanged (i.e., fixed or static) for the entire execution. Then, samples for training the NN are extracted from the execution following the scheme discussed above. The samples collected from these experiments are stored in a samples database and then used to train the first version of the NN referred to as *static approach*.

The *static approach* was originally devised with the aim of studying how accurate can be the NN prediction with this simple training method. An important shortcoming of the static approach, however, is that since the NN is only trained

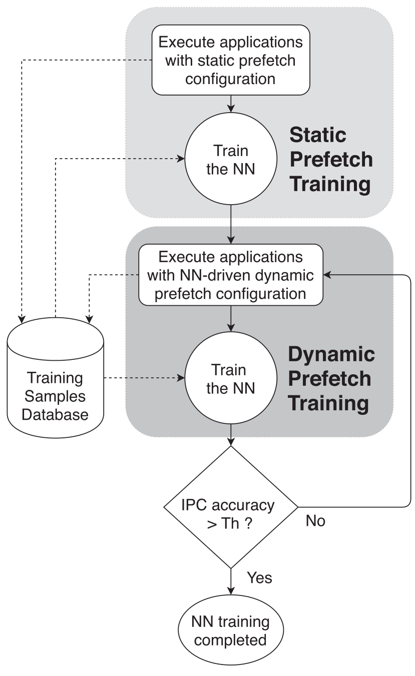

Execute applications with static prefetch configuration
Train the NN
Static Prefetch Training
Execute applications with NN-driven dynamic prefetch configuration
Training Samples Database
Train the NN
Dynamic Prefetch Training
IPC accuracy > Th ?
No
Yes
NN training completed

Fig. 2. Training flow.

with samples keeping the same prefetch configuration, it does not consider the impact on performance of changing the prefetch setting at run-time.

To provide the NN the capability to analyze this impact, one or more *dynamic-prefetch* training stages are performed. In any of these stages, to obtain the samples that will be used to train the NN, each application is executed once with a prefetch configuration that is changed along execution. This change is driven by the NN trained in the previous stage, which estimates, for each quanta group the IPC for the different studied prefetch configurations to select the best performing one, as explained in Section 5.2. Proceeding in this way, samples collected along an execution can be taken with different prefetch configurations; therefore, grasping the impact of dynamic prefetch configuration changes. As the samples obtained during executions performed in the static-prefetch training stage, samples obtained in a dynamic-prefetch training are stored in the samples database to train another NN with static and dynamic samples, which is done at the end of the stage.

A NN that has been trained with samples collected in one or more dynamic-prefetch trainings is referred to as $dynamic\ approach_{iter}$, where $iter$ refers to the number of dynamic-prefetch trainings performed to train the NN. Once the dynamic approach$_{iter}$ has been trained, it can be used in a next dynamic-prefetch stage to train the dynamic approach$_{iter+1}$, and so on. This implies a loop of multiple consecutive dynamic-prefetch trainings, as shown in the diagram. This loop is finished when it is detected that NN accuracy does not improve with more iterations.

In summary, the NN is trained through two sequential stages; the static-training prefetch stage, which is mainly used to initialize the training flow, and the dynamic-training one, which loops for some iterations to improve the prediction accuracy.

6.4 Multi-Core Neural Network

As mentioned above, in this work we devise two NNs, a single-core NN and a multi-core NN. Both networks provide the IPC as output. In the case of the multicore, the NN provides the predicted IPC for each application with the studied prefetch configurations. However, unlike the single-core NN, the multi-core NN considers the interference introduced by the applications co-running in the other cores. To this end, we performed three main updates to the described training methodology: i) the NN is trained with workload mixes, ii) memory bandwidth is considered as input, iii) training samples are obtained from executions of mixes. Below, these three updates are discussed.

Regarding the first update, *workload mixes* composed of multiple applications running together, each one on a different core, are used to train the NN instead of applications running alone. In order to study a wide diversity of inter-application interference, we built 25 workloads composed of 6, 8, and 10 benchmarks randomly selected that were used only to train the NN. We gather a dataset from the execution of all the workloads. A fraction of the dataset (i.e., 80% of the samples) was used in training and the remaining fraction (i.e., 20%) was used for testing the NN accuracy in the BigML platform.

Benchmarks were randomly selected from the entire SPEC CPU 2006 and SPEC CPU 2017 benchmark suites, considering both prefetch friendly and prefetch unfriendly benchmarks. Note that in the multi-core scenario, the performance of the applications does not only depend on their prefetch configuration but also on the dynamically varying bandwidth utilization of the co-running applications. Therefore, it makes sense to consider all types of co-running applications, which also mimics realistic scenarios.

The second update refers to consider the interference at the shared resources. In order to predict the performance of an application in multi-program execution, the model needs to consider both the target application's intrinsic behavior and the inter-application interference at the shared resources. In our experimental platform, the main memory is the most critical resource shared among cores. Note that, in contrast to Intel processors, where the entire LLC can be accessed by any core, in the POWER8 and POWER7 processors, a core mainly accesses to its corresponding LLC region [43]. In addition, previous work [44] has shown that the performance of individual applications when running in a multi-program workload is affected both by its bandwidth consumption and that of the co-runners. Therefore, we add two inputs to the multi-core NN: i) the bandwidth consumption of the target application in the previous quanta and ii) the sum of the bandwidth consumption of the co-runners as a whole. The bandwidth consumed by an application or core during a quantum is estimated considering both prefetch requests and LLC load misses.

The multi-core NN is accessed during inference with these additional inputs to predict the IPC of each application running on a core. Notice that this implies that to make predictions for all running applications at the end of each quantum group, the multi-core NN needs to be accessed NxM times, where N is the amount of applications in the mix and M is the number of considered prefetch configurations. An alternative approach could be a NN that selects all

TABLE 3
NN Description

Parameter	Single-core NN	Multi-core NN
# of inputs	21	23
# of layers	5	5
# of hidden layers	3	3
Activation function	ReLU	ReLU
# of neurons per layers	64,128,64	128,128,128
Optimizer	ADAM algorithm	ADAM algorithm
Learning rate	0.1%	0.5%
Dropout rate	25%	25%
beta1	0.9	0.9
beta2	0.999	0.999
epsilon	1e-8	1e-8

the prefetch settings together for all the cores. However, this design presents three main drawbacks. First, it will be much more complex since it needs to process in the order of $N\times$ much more input data and generate N prefetch configurations (one for each core) as output at the same time. Second, training time will be longer, since it requires $N\times$ executions to get the same amount of samples as the proposed approach. This is because, in a training execution of the alternative approach, a quantum produces just one sample while in the proposed approach a quantum generates N samples. Third, the trained NN will be only applicable to a system (mix) with a given number of cores (applications). In contrast, our approach is independent of the number of cores of the target system.

The third update refers to how training samples are gathered. For this purpose, the multi-core NN also applies static- and dynamic-prefetch trainings, but using executions of the multi-program workload mixes. In the static-prefetch training, each mix is executed several times, one for each of the studied prefetch configurations. For each execution, all the cores hold the same prefetch configuration. The collected samples, which are obtained for the execution of each application running in the mix are stored in the samples database and then used to train the (multi-core) static approach.

As in the training of the single-core NN, the flow diagram shown in Fig. 2 is followed. In other words, once the static approach has been trained, it is used in the first dynamic-prefetch training stage. In this stage, each mix is run to obtain training samples, but instead of setting a static prefetch configuration for the whole run, the static approach dynamically chooses each quantum a prefetch configuration for each application. The samples collected in these executions are stored in the samples database and used to train the first dynamic approach ($dynamic\ approach_1$). Samples collected with the $dynamic\ approach_1$ could be used later in further dynamic training iterations.

The devised networks are composed of 5 (1 input, 3 hidden, and 1 output) full connected layers. Table 3 summarizes the main parameters of the neural networks used in this work. The training platform provides three metrics to report the accuracy of the trained NN: mean absolute error, mean squared error, and R squared value. The mean absolute error, mean squared error, and R squared values for the multi-core NN are 0.08, 0.02 and 0.94, respectively. For single-core NN, these values are 0.05 (mean absolute error), 0.01 (mean squared error), and 0.97 (R squared value).

7 Experimental Evaluation

7.1 Individual Applications

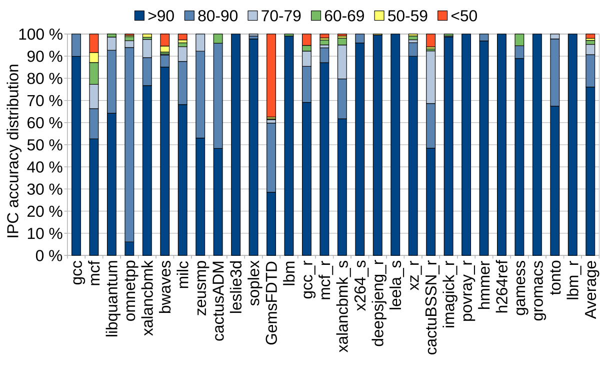
This section evaluates the performance of the NN we built working on individual applications without considering inter-thread interference. The main purpose of this network is to be used as workbench to work on improving the prediction accuracy for individual applications running in isolation.

To this end, two main steps were followed. In the first step, the NN was only trained with samples obtained in the static-prefetch trainig stage. In the second step, dynamic-prefetch training data was used to re-train (see Section 6.3) the NN combining static and dynamic data to improve prediction accuracy when the prefetcher configuration changes.

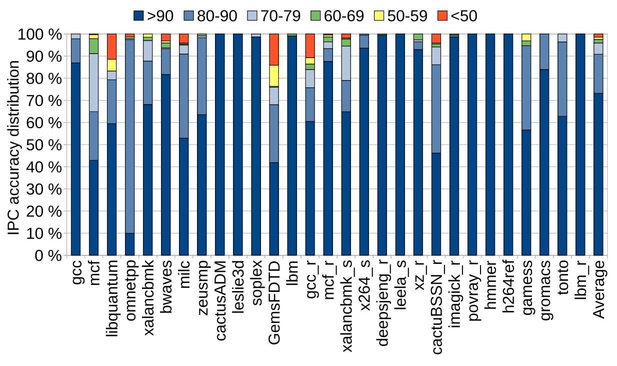
Fig. 3 compares the prediction accuracy of both training modes. Each bar shows the distribution of the accuracy (in percentage) of the predicted IPC over the real value.

It can be observed that the static approach already shows good IPC estimates across most of the applications. The dynamic approach, however, allows improving the accuracy for those applications that experienced higher IPC deviations (e.g., *mcf*, *omnetpp*, *zeusmp*, or *gemsFDTD*).

The proposed dynamic-prefetch training outperforms, in general, the accuracy of the static one in 20 out of 23 applications. On average, the prediction accuracy of the static approach exceeds 90% and 80% for 75% and 90% of the total predictions made, respectively. This accuracy is significantly improved by *dynamic-training*, which rises up to and nearly

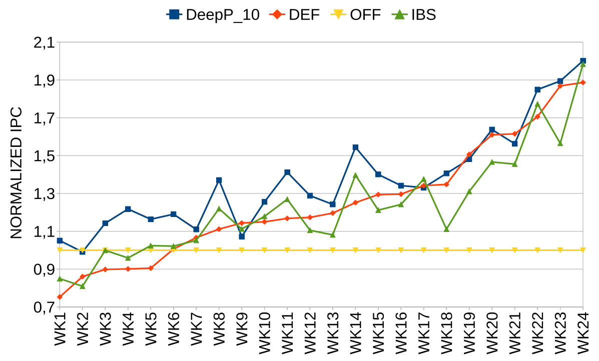


(a) Static

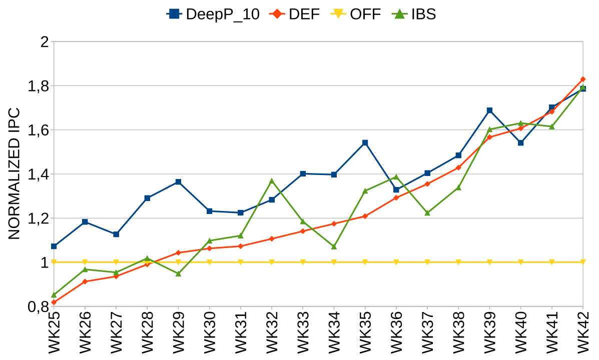


(b) Dynamic

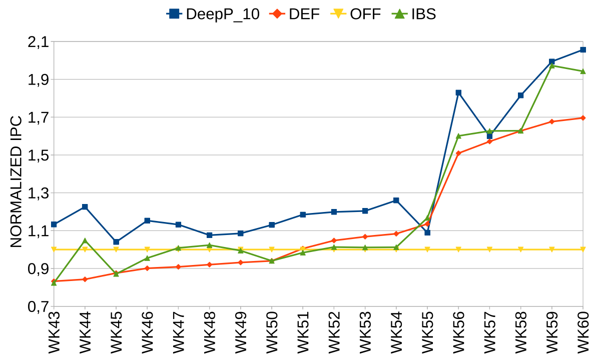
Fig. 3. Interval accuracy of the predicted IPC with respect to the reached IPC in the next quanta. The average prediction accuracy is 91.8% and 92.1%, for the Static and Dynamic approaches, respectively.



(a) Performance of the 6-application workloads.



(b) Performance of the 8-application workloads.



(c) Performance of the 10-application workloads.

Fig. 4. Performance achieved in the multi-program workloads for the studied prefetch configuration approaches. Performance is normalized over OFF.

to 90% of the predictions, respectively. We found that, in general, the accuracy significantly improves in some applications when iterating the training phase. Then, the IPC of these applications stabilizes in most cases and falls closer to the real IPC achieved in the next quantum. The presented values were obtained with three iterations.

7.2 Multi-Core Evaluation

This section compares the performance of DeepP to *DEF* and *IBS* [14] prefetch approaches. *DEF* refers to the default IBM POWER8 prefetch configuration, which is set across all the execution. IBS was proposed for the IBM POWER7 prefetcher, so we adapted this mechanism to work on our system, an IBM POWER8 machine. In the IBM POWER7, IBS uses two prefetch configurations (the most aggressive prefetch setting and OFF) and therefore we selected the most aggressive and the OFF configurations in our IBM POWER8 platform to implement IBS.

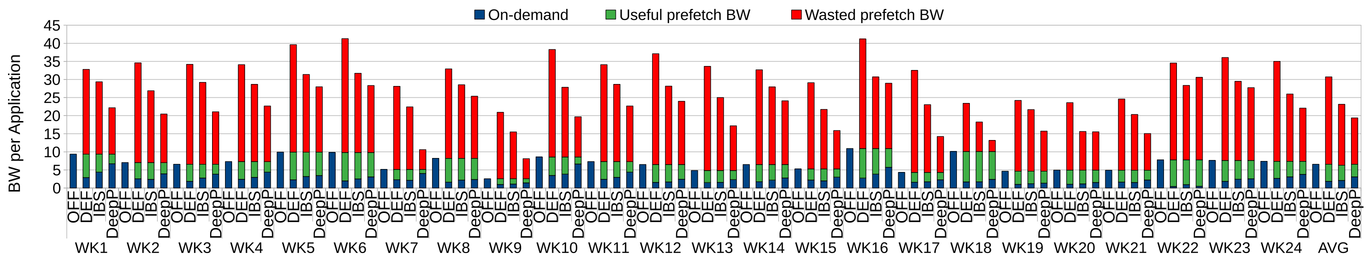
To evaluate DeepP we have used a set of 60 randomly generated multi-program mixes of different sizes, which help analyze a wide range of inter-application interference. We built 24 workloads composed of 6 benchmarks, 18 8-benchmark workloads, and 18 10-benchmark workloads. Two workloads of each size contain applications that were not used to train the NN. These workloads are referred to as WK19, WK21, WK40, WK42, WK57, and WK60.

Fig. 4 shows the harmonic mean [45] of the IPC of the benchmarks in each workload across the studied approaches normalized to no prefetching. To simplify the analysis, the workloads are sorted in ascending normalized performance order of the *DEF* prefetch configuration.

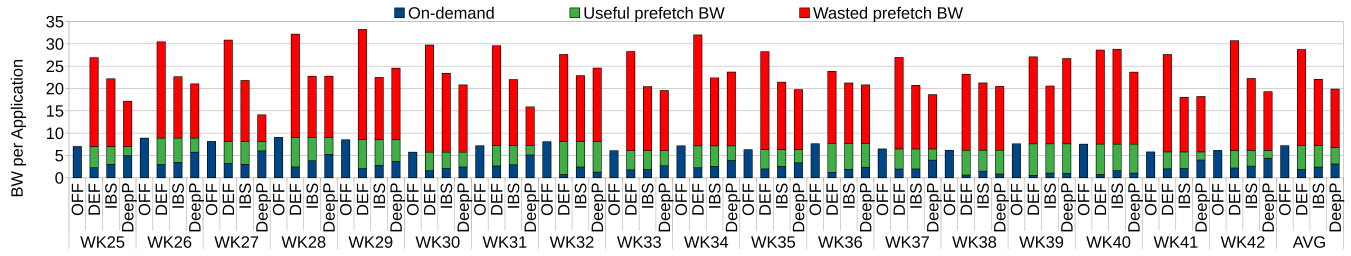
It can be observed that the three prefetch approaches provide significant performance gains over the OFF prefetch configuration in the 6-application and 8-application workloads. It can be appreciated that DeepP is the best performing approach across all the mixes with the only exception of just a couple of them. Its performance exceeds 30% in around 33% of the mixes (8 out of 24 in the 6-application mixes and 6 out of 18 in the 8-application mixes). DeepP differences rise over the DEF and IBS approaches as the number of benchmarks in the mix rises. An interesting observation is that *no prefetching* provides better performance in about in the 6-application workloads, and this percentage rises in the 10-application workloads. Moreover, performance gains achieved by the prefetch schemes dramatically drop in 10-application workloads, while DeepP is the only prefetch approach that clearly outperforms OFF across all the mix sizes.

The main reason that explains the performance results is the inter-application interference at the main memory bandwidth consumption. Fig. 5 shows the bandwidth averaged by application of each workload, quantified in terms of memory transactions per μs. We measured experimentally that the maximum available bandwidth in the system is about 185-195 transactions per μs. This bandwidth is split in three main components: *on-demand*, that is the main memory bandwidth due to regular memory instructions, *useful* that refers to bandwidth consumed by prefetches that are later requested by the processor, and *wasted* that refers to useless prefetches. As expected, the average bandwidth consumed per application is on average barely the same when prefetch is disabled (OFF) regardless of the number of applications in the mix. In contrast, the average bandwidth drops as the number of applications rises for all the remaining prefetching approaches. This fact is more accentuated in the aggressive default approach (DEF), which tends to demand all the system bandwidth (aggregated among the applications) regardless of the number of running applications. Consequently, bandwidth contention rises and adversely impacts on performance.

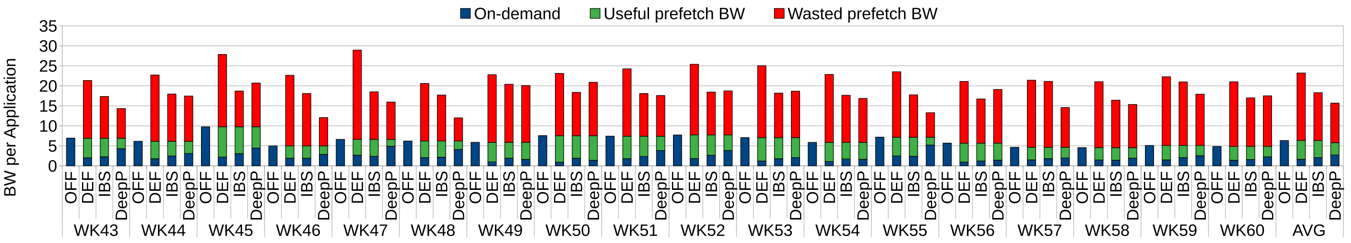
By setting the best prefetch configuration predicted by the deep network, DeepP is able to estimate which applications will benefit the most from enabling or increasing the prefetch aggressiveness, which allows i) reducing bandwidth utilization (i.e., less wasted bandwidth) and hence contention, and ii) saving prefetch bandwidth for being used by the applications that actually benefit from prefetching.



(a) Bandwidth (trans/us) of the 6-application workloads.



(b) Bandwidth (trans/us) of the 8-application workloads.



(c) Bandwidth (trans/us) of the 10-application workloads.

Fig. 5. Memory memory bandwidth consumption in the multi-program workloads for the studied prefetch configuration approaches. Bandwidth is split in according to it is consumed by on demand processor requests or prefetch requests, which in turn is broken down in bandwidth consumed by useless (wasted) and useful prefetches.

In the case of IBS, the bandwidth consumed by this approach, in general, falls in between the consumed by DEF and DeepP. This is mainly due to IBS only uses prefetch off and the most aggressive prefetch configuration, thus the prefetcher is significantly switched off to reduce the interference.

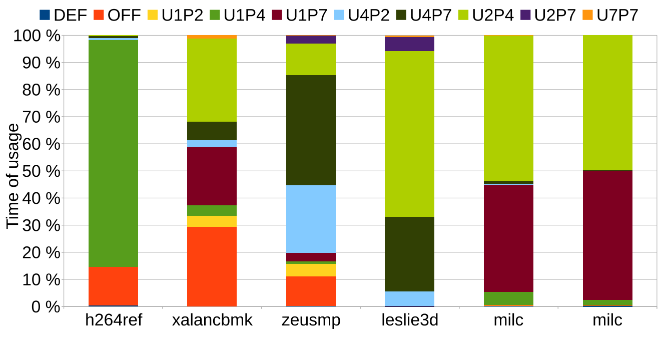
7.3 Mix Example

This section analyzes the performance and bandwidth of individual applications composing a mix to provide further insights in the DeepP's performance gains. For illustrative purposes, we analyze mix WK4. Fig. 6a depicts the percentage of time that each prefetch configuration has been active for each application along the execution of workload 4 with DeepP. The figure shows that three applications execute most of the time with $U1P7$ or $U2P4$. These configurations are less aggressive than the default prefetch configuration ($U4P4$), which allows reducing the pressure on the main memory bandwidth. Looking at Fig. 6b three main observations can be appreciated. First, the aggregate bandwidth consumed among all the applications in the mix is much lower in DeepP than in DEF which reduces the bandwidth contention. Second, in those applications whose performance improves significantly with prefetching ($leslie3d$), DeepP consumes more bandwidth than DEF. Notice that in these applications the aggressive $U4P7$ configuration is kept for an important fraction of the execution time. This fact, jointly with the reduced contention allows DeepP to outperform DEF in this application. Third, in the applications where prefetching brings scarce performance gains (e.g., the two instances of $milc$), DeepP consumes less bandwidth than DEF. In addition, this plot also illustrates the importance of correctly handling bandwidth, which can be especially appreciated in $xalancbmk$, an application that does not benefit from prefetch; nevertheless, the bandwidth contention introduced by co-runners makes its performance to drop. In other words, the higher bandwidth contention makes memory latency of *on demand* memory requests to increase. Comparing both plots of the figure, it can be noticed the importance of handling configurations with different aggressiveness. If this is not considered, as in IBS, there are less opportunities to share the memory bandwidth in a efficient way to improve performance.

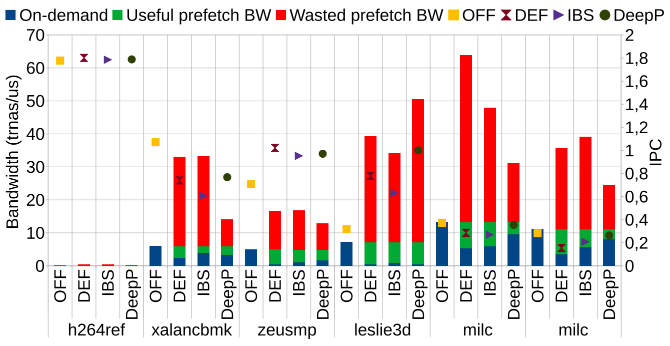
8 Conclusion

In this paper, we have proposed DeepP that uses a deep neural network to select the best prefetch configuration for each running application. To this end, the network estimates the performance of each application under the studied prefetch configurations. The neural network makes these estimates using a set of hardware events that relate to the intrinsic application behavior and the main memory bandwidth utilization of the co-runners.

A key design issue of DeepP is the training methodology of the neural network, which uses dynamic-prefetch training



(a) Configurations used during the execution.



(b) Memory bandwidth and IPC of each application.

Fig. 6. Performance analysis of workload WK4. System throughput (IPC harmonic mean) is 0.57 for DeepP, 0.42 for DEF, 0.47 for OFF and 0.45 for IBS.

samples to improve the accuracy of the predictions. Results show that the neural network accuracy exceeds 90% in more than 92% of the performance estimates when selecting the best prefetch configuration. Regarding performance, using the network to dynamically set the best prefetch configuration of the applications in multi-program workloads DeepP achieves performance gains, on average, by 5.8%, 6.7%, and 15.8% across a set of random 6-, 8-, and 10-application workloads compared to the default prefetch configuration.

DeepP and its learning methodology can be easily adapted to other systems like Intel and ARM. In this regard, both the prefetch configurations and the set of hardware events used by the NNs should be adapted. As for future work, we plan to explore advanced NNs such as Convolutional Neural Networks (CNNs) or Long Short-Term Memory (LSTM) to seek potential accuracy improvements. For instance, CNNs might help find symbiosis among neighbour cores, and LSTM help identify patterns along the execution to improve the prefetch configuration selection.

References

[1] J. Baer and T. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. Supercomput.*, 1991, pp. 176–186.

[2] J. Fu, J. Patel, and B. Janssens, "Stride directed prefetching in scalar processors," in *Proc. 25th Annu. Int. Symp. Microarchit.*, 1992, pp. 102–110.

[3] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. 24th Annu. Int. Symp. Comput. Archit.*, 1997, pp. 252–263.

[4] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: An application-driven study," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 195–206.

[5] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, Jan./Feb. 2005.

[6] S. H. Pugsley, *et al.*, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *Proc. 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 626–637.

[7] S. Byna, Y. Chen, andX. Sun,"Taxonomy of data prefetching for multicore processors," *J. Comput. Sci. Technol.*, vol. 24, no. 3, pp. 405–417, 2009.

[8] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. 42nd Annu. Int. Symp. Microarchit.*, 2009, pp. 316–326.

[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *Proc. 38th Int. Symp. Comput. Archit.*, 2011, pp. 141–152.

[10] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware dram controllers," in *Proc. 41st Annu. Int. Symp. Microarchit.*, 2008, pp. 200–209.

[11] M. Torrens, "Improving prefetching mechanisms for tiled cmp platforms," Ph.D. Thesis. Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain, 2016.

[12] C. Ortega *et al.*, "Intelligent adaptation of hardware knobs for improving performance and power consumption," *IEEE Trans. Comput.*, vol. 70, no. 1, pp. 1–16, Jan. 2021.

[13] V. Jiménez *et al.*, "Adaptive prefetching on POWER7: Improving performance and power consumption," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, pp. 4:1–4:25, 2014.

[14] V. Jiménez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. J. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *Proc. 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 39–50.

[15] C. Navarro, J. Feliu, S. Petit, M. E. Gómez, and J. Sahuquillo, "Bandwidth-aware dynamic prefetch configuration for IBM POWER8," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1970–1982, Aug. 2020.

[16] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," in *Proc. 17th Int. Conf. High Perform. Comput. Commun.*, 2015, pp. 383–389.

[17] M. Li, *et al.*, "PATer: A hardware prefetching automatic tuner on IBM POWER8 processor," *Comput. Archit. Lett.*, vol. 15, no. 1, pp. 37–40, 2016.

[18] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 285–297.

[19] M. Khan, M. A. Laurenzanoy, J. Marsy, E. Hagersten, and D. Black-Schaffer, "AREP: Adaptive resource efficient prefetching for maximizing multicore performance," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 367–378.

[20] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual spatial pattern prefetcher," in *Proc. 52nd Annu. Int. Symp. Microarchit.*, 2019, pp. 531–544.

[21] M. Hashemi *et al.*, "Learning memory access patterns,"in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1919–1928.

[22] L. Peled, U. Weiser, andY. Etsion,"Neural network prefetcher for arbitrary memory access patterns," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–27, 2019.

[23] S. Liao, T. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, pp. 1–10.

[24] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 39–50.

[25] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proc. Int. Conf. Mach. Learn.*, 2020, *arXiv:2006.16239*.

[26] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez,"Perceptron-based prefetch filtering," in *Proc. 46th Annu. Int. Symp. Comput. Archit.*, 2019, pp. 1–13.

[27] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. Int. Symp. Microarchit.*, 2019, pp. 413–425.

[28] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," in *Proc. 2nd Int. Sci. Community Assoc. Int. Workshop AI Assisted Des. Archit.*, 2019, *arXiv:1906.09889*.

[29] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *Proc. 53rd Annu. Int. Symp. Microarchit.*, 2020, pp. 118–130.

[30] E. C. Barboza, S. Jacob, M. Ketkar, M. Kishinevsky, P. Gratz, and J. Hu, "Automatic microprocessor performance bug detection," in *Proc. 27th Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 545–556.

[31] E. O.-A.-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham, "Using model trees for computer architecture performance analysis of software applications," in *Proc. 8th Int. Symp. Perform. Anal. Syst. Softw*, 2007, pp. 116–125.

[32] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 131–142.

[33] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad , "Bingo spatial data prefetcher, HPCA," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 399–411.

[34] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proc. 23rd Int. Conf. Supercomput.*, 2009, pp. 499–500.

[35] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. Int. Symp. Microarchit.*, 2013, pp. 247–259.

[36] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. 49th Annu. Int. Symp. Microarchit.*, 2016, pp. 1–12.

[37] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proc. 47th Annu. Int. Symp. Comput.*, 2020, pp. 118–131.

[38] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proc. 36th Annu. Int. Symp. Comput.*, 2009, pp. 69–80.

[39] P. Michaud, "Best-offset hardware prefetching," in *Proc. Int. Symp. High Perform. Comput.*, 2016, pp. 469–480.

[40] B. Sinharoy *et al.*, "IBM POWER8 processor core microarchitecture," *IBM J. Res. Develop.*, vol. 59, no. 1, pp. 1–21, 2015.

[41] S. Eranian, "What can performance counters do for memory subsystem analysis?" in *Proc. ACM SIGPLAN Workshop Memory Syst. Perform. Correctness*, 2008, pp. 26–30.

[42] "BigML.com," Accessed: Apr. 2020. [Online] Available: https://bigml.com/

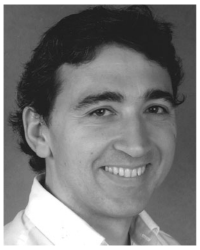
[43] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, pp. 7–15, Mar./Apr. 2010.

[44] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 237–248.

[45] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

Manel Lurbe received the BS and MS degrees in 2019 and 2020 respectively, in computer engineering from the Universitat Politècnica de València (UPV), Spain, where he is currently working toward the PhD degree with the Department of Computer Engineering. His research focuses on prefetching strategies in commercial multicore processors and deep learning.

Josué Feliu received the MSc and PhD degrees in computer engineering from the Universitat Politècnica de València, Spain, in 2012 and 2017, respectively. He is currently a postdoctoral researcher with the Universidad de Murcia. His research interests include scheduling strategies and microarchitecture for multicore and multi-threaded processors. He was the recipient of IEEE TCSC Outstanding PhD Dissertation Award in 2017.



Salvador Petit (Member, IEEE) received the PhD degree in computer engineering from the Universitat Politècnica de València (UPV), Spain. Since 2009, he has been an associate professor with DISCA Department UPV, where he has taught several courses on computer organization. He has authored more than 100 refereed conference and journal papers. His research interests include multithreaded and multicore processors, memory hierarchy design, GPU architecture, and resource management.

Maria E. Gomez received the MS and PhD degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1996 and 2000, respectively. She joined the DISCA Department, UPV, in 1996, where she is currently a full professor. She has authored or coauthored more than 70 conference and journal papers. She was on program committees for several major conferences. Her research interests include processor architecture and interconnection networks.

Julio Sahuquillo (Member, IEEE) received the BS, MS, and PhD degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain. He is currently a full professor with DISCA Department, UPV. He has taught several courses on computer architecture. He has authored more than 150 refereed conference and journal papers. His research interests include processor microarchitecture, memory hierarchy design, GPU architecture, and system resource management.