DiVA
http://www.diva-portal.org

Postprint

This is the accepted version of a paper presented at *International Conference on Supercomputing*.

Citation for the original published paper:

Sánchez Barrera, I., Black-Schaffer, D., Marc, C., Moretó, M., Stupnikova, A. et al. (2020)
Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning
In: *ICS '20: Proceedings of the 34th ACM International Conference on Supercomputing*
https://doi.org/10.1145/3392717.3392765

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-431441

Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning

Isaac Sánchez Barrera
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
isaac.sanchez@bsc.es

David Black-Schaffer
Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

Marc Casas
Barcelona Supercomputing Center (BSC)
marc.casas@bsc.es

Miquel Moretó
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
miquel.moreto@bsc.es

Anastasiia Stupnikova
Uppsala University
Uppsala, Sweden
Anastasiia.Stupnikova.1308@student.uu.se

Mihail Popov
Uppsala University
Uppsala, Sweden
mihail.popov@it.uu.se

ABSTRACT

Both NUMA thread/data placement and hardware prefetcher configuration have significant impacts on HPC performance. Optimizing both together leads to a large and complex design space that has previously been impractical to explore at runtime.

In this work we deliver the performance benefits of optimizing both NUMA thread/data placement and prefetcher configuration at runtime through careful modeling and online profiling. To address the large design space, we propose a prediction model that reduces the amount of input information needed and the complexity of the prediction required. We do so by selecting a subset of performance counters and application configurations that provide the richest profile information as inputs, and by limiting the output predictions to a subset of configurations that cover most of the performance.

Our model is robust and can choose near-optimal NUMA+Prefetcher configurations for applications from only two profile runs. We further demonstrate how to profile online with low overhead, resulting in a technique that delivers an average of 1.68× performance improvement over a locality-optimized NUMA baseline with all prefetchers enabled.

CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*; • **General and reference** → **Performance**; *Measurement*; • **Software and its engineering** → *Memory management*; • **Computing methodologies** → Cluster analysis; Supervised learning; Cross-validation; Model verification and validation; **Model development and analysis**.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICS '20, June 29-July 2, 2020, Barcelona, Spain
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7983-0/20/06.
https://doi.org/10.1145/3392717.3392765

KEYWORDS

page mapping, thread mapping, machine learning model, performance optimization, NUMA, prefetching

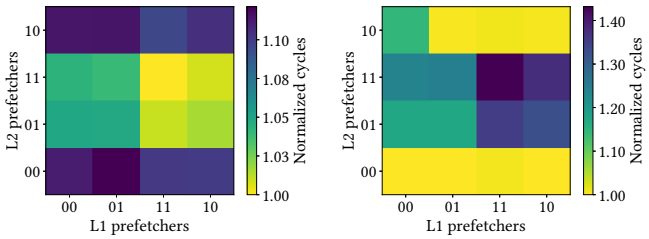
ACM Reference Format:
Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov. 2020. Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392765

1 INTRODUCTION

High Performance Computing systems increase memory bandwidth by providing local DRAM for each processor node with coherent communications between nodes, and reduce memory latency by providing prefetchers, which identify access patterns and fetch the data ahead of time. Local DRAM increases overall bandwidth, but results in *non-uniform memory access* (NUMA) behaviors: latency and bandwidth depend on the node accessing the data and the node where the data is stored. Prefetchers reduce latency, but increase bandwidth and cache evictions if the predictions are inaccurate.

For both NUMA and prefetching, appropriately configuring the system can lead to significant performance gains. NUMA optimizations have been explored extensively [9, 14, 15, 29, 39] and focus on adjusting the thread and data placement across the nodes to minimize latency and maximize bandwidth. These optimizations are typically done via the operating system's memory manager and thread scheduler. Configuring prefetchers via hardware registers [20, 22] improves performance by adjusting where data is prefetched and how aggressively it is fetched to match the application and system cache hierarchy.

However, previous prefetcher studies have assumed fixed NUMA configurations, and, likewise, previous NUMA studies have assumed fixed prefetcher configurations. This leaves open the question as to what benefits can be achieved by co-optimizing for both NUMA (thread and data placement) and prefetchers.



(a) All data in one node **(b) Optimized for locality**

Figure 1: Normalized cycles (lighter is faster) for `x_solve` from BT showing opposite effects of prefetchers depending on NUMA configuration, using 32 contiguous threads. (16 prefetcher configurations: 0 enabled, 1 disabled).

In this work we study the interactions between NUMA and prefetching. To appreciate the complexity of this optimization, consider Figure 1, which shows how part of the BT benchmark [1] is affected by prefetcher configurations (16 squares) depending on the NUMA configuration (left: all data in one node, right: optimized for locality). As the figure shows, while the middle row of prefetcher configurations improves performance with the left NUMA configuration, the same prefetcher configurations hurt performance for the right NUMA configuration. The full complexity of this problem is shown in Figure 2: 16 prefetcher configurations for each of 4 page mappings across 5 combinations of thread/node parallelism and mapping for 57 parallel benchmark regions. It is clear from this figure that, while the benchmarks exhibit diverse behaviors across the NUMA+Prefetcher configurations, there are clear patterns that we can leverage for efficient optimization.

To address this large search space we develop models that can choose near-optimal NUMA+Prefetcher configurations for applications. Our models use input profiles (performance counter values) collected by executing parallel regions from the application under a few specific NUMA+Prefetcher configurations. This provides valuable information about the applications (as they are profiled on multiple NUMA+Prefetcher configurations) at low cost (as only a few configurations need be profiled). We demonstrate through cross-validation that the resulting models are capable of accurately predicting good NUMA+Prefetcher configurations for unseen applications, and we show how they can be gathered online at runtime.

Our contributions (summarized in Figure 3) are:

- Demonstrating that the co-optimization of NUMA and prefetcher configurations can lead to a 1.77× average speedup over a locality-optimized NUMA baseline with all prefetchers enabled (Section 3), but that it requires the impractical evaluation of 288 distinct configurations per parallel region (Section 2).
- The design of a prediction model (Section 4) that requires the evaluation of only 2 distinct configurations and achieves an average of 1.68× (95% of the optimal performance) speedup over a locality-optimized NUMA baseline with all prefetcehrs enabled (Section 5).
- A methodology for applying our model at online at runtime that handles inter-region configuration conflicts. (Section 6).

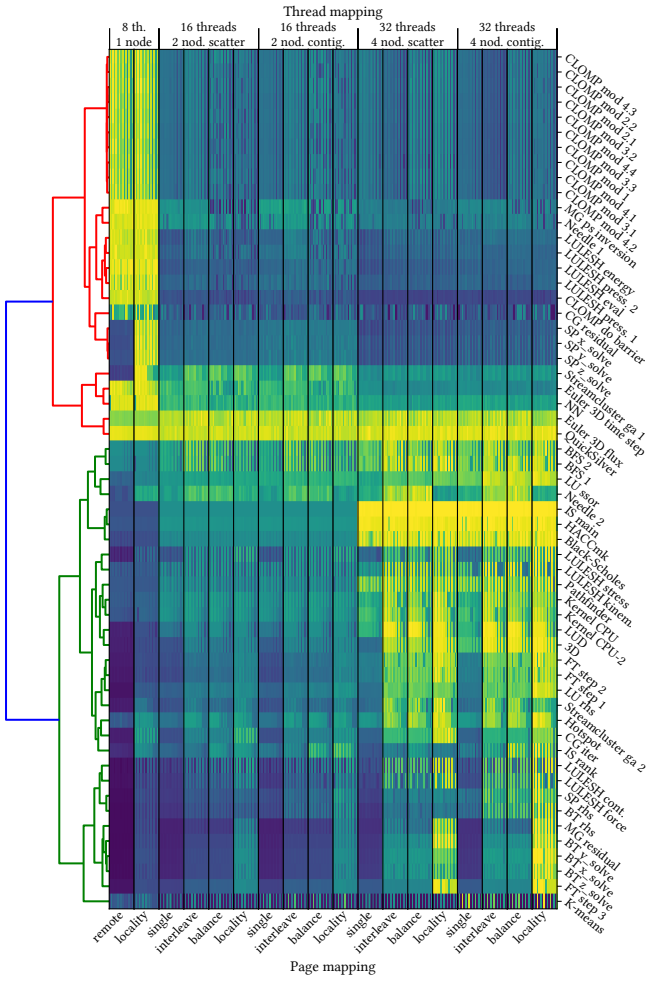


Figure 2: Normalized speedups (lighter is faster) of parallel regions showing complex sensitivities to NUMA+Prefetcher configurations on a Sandy Bridge system. Regions are clustered according to similar speedup behaviors. Each vertical line on a NUMA configuration (thread and page mapping) is 1 of 16 prefetcher configurations.

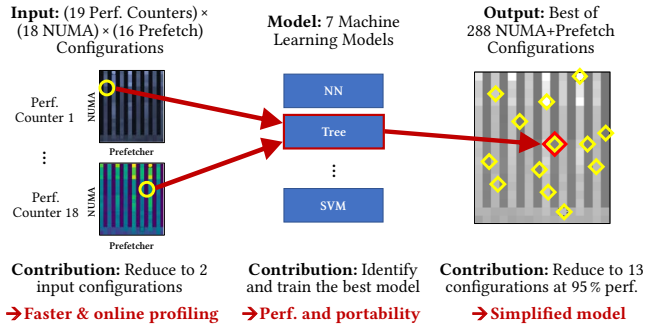


Figure 3: Developing an effective online model: reducing the input configurations for profiling, selecting the machine learning model, and reducing the possible outputs.

2 SEARCH SPACE

The main challenge in optimizing applications for both NUMA and prefetching is efficiently exploring the large number of configurations. In this work we consider 18 (or 20, depending on the system) state-of-the-art NUMA optimizations for thread- and page-placement together with the 16 hardware prefetcher configurations available on our system. This leads to a search space of 288 (or 320) NUMA+Prefetcher (thread+page+prefetching) configurations, which would take over 4 months of execution time to explore directly for our benchmarks.

2.1 NUMA Configurations

We focus on standard fork-join parallel HPC applications, e.g., OpenMP parallelized for-loops, as this results in predictable thread assignments. The **thread mapping** defines how the application's execution threads are assigned to the hardware cores available in the system[1]. Similarly, the **page mapping** defines how the application's memory pages are assigned to the processor nodes' DRAM.

Except in the case of a single-node system, where all NUMA configurations are equivalent, the different combinations of thread- and page-mapping can give very different results in terms of data locality, performance, and energy consumption for each application.

2.1.1 Thread Mapping. We consider three thread mapping parameters: **degree of parallelism** (number of threads used), **NUMA degree** (number of NUMA nodes used), and **assignment algorithm** (how threads are assigned to cores on the nodes). We consider two thread assignments: contiguous and scattered. Both evenly distribute the threads across the nodes. However, **scattered** uses round-robin to place the threads (e.g., if using 4 threads and 2 nodes, threads 1 and 3 are mapped to the first node and threads 2 and 4 to the second) while **contiguous** places them iteratively (e.g., if using 4 threads and 2 nodes, threads 1 and 2 are mapped to the first node and threads 3 and 4 to the second). For the configurations where we use only a subset of the cores, the remainder are idle. We pin the threads to the specific cores to keep the mappings stable throughout the execution.

2.1.2 Page Mapping. We consider 7 different page mappings, some of which require detailed profiler/programmer information and others which can be applied automatically by the system.

The automatic policies include: **first touch** (each page is allocated on the node that first accesses it), **single node** (all pages are allocated on one single node), and **interleaved** (pages are distributed in a round-robin fashion among the available nodes). We additionally consider two additional policies when all threads are in a single node: **local** (pages are allocated on the same node), and **remote** (pages are allocated on a different node from the execution). Remote mapping is typically a bad configuration because it increases access latency and reduces bandwidth, but is useful for exposing NUMA sensitivity. It is important to note that even though the first touch policy does not require any profiling or support from the programmer, the result of this mapping is highly-dependent on the application code, the thread mapping, and the scheduling algorithm.

[1]We ignore hardware multithreading in our policies and experiments for simplicity.

The mappings that require detailed profiler/programmer support include: **locality** (each page is allocated in the node of the cores that will access the page the most), and **balance** (pages are spread across the nodes in such a way that the total amount of memory accesses to each node is approximately the same). These mappings require profiling the application's access pattern and implicitly assume that the patterns are reasonably stable across different runs and inputs, which has been shown to be a fair assumption for these benchmarks [34, 40].

2.2 Prefetcher Configurations

The hardware prefetcher configurations provided by Intel[2] for post-Nehalem microarchitectures provide 4 bits (16 combinations) to control four prefetchers [8, 18, 26]:

- **DCU IP-correlated prefetcher**: A stride prefetcher that brings data from the L2 into the L1 (data cache unit, or DCU) by correlating prefetches with the instruction pointer (IP, or program counter).
- **DCU prefetcher**: A next-line L1 cache prefetcher.
- **L2 adjacent cache line prefetcher**: For every access it brings the previous or next line (64 B) that completes a memory block aligned to 128 B.
- **L2 streamer prefetcher**: Detects data streams and fetches the next predicted lines to the L2 cache. Similar to the DCU IP-correlated prefetcher, but not using the instruction pointer.

2.3 Faster Evaluation: Sampling with Codelets

As executing the full applications for the complete set of NUMA+Prefetcher configurations is impractical, we instead use a technique called *codelet execution* [11, 34]. Codelet execution extracts hot regions from the application as small, representative *codelets* and uses them to characterize the application's performance. Codelets are on average 66× faster [35] to evaluate than running the full application.

Codelet execution is faster because it only executes a few instances of each region (instead of hundreds during the original run). To ensure that the codelet execution matches the behavior of the region within the application, codelets implement a short warmup phase that configures the system state (e.g., caches) to match the application's native execution.

Codelets have been shown to be quite accurate for both micro-architectural evaluation [12] and NUMA configuration studies [35]. This is because parallel regions typically exhibit similar behavior [40]. For our fork-join applications, we extract codelets for instances of each important OpenMP parallel region. This results in 57 codelets, which take approximately 2 days to execute across all configurations, but would take over 4 months with full executions.

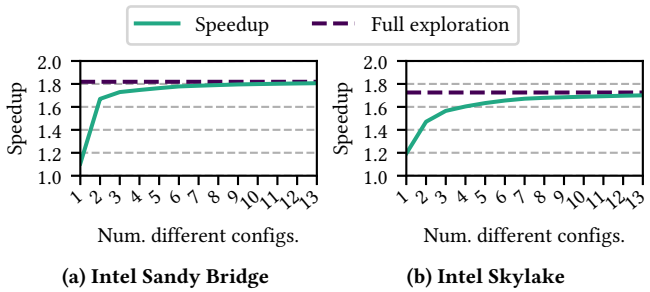
3 CHARACTERIZATION

To understand how we can simplify the search space, we first explore all NUMA+Prefetcher configurations via codelet exection. This brute-force exploration allows us to identify common behaviors across codes and configurations, which we can then use to build efficient models for choosing the best configuration.

[2]See https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

Table 1: Comparison of speedups between different optimization searches against an optimized default (pages: locality, threads: scatter, prefetchers: on).

Optimization	Speedup (geometric mean) Sandy Bridge	Skylake	Average
Only NUMA	1.73	1.59	1.66
Only prefetchers	1.19	1.19	1.19
NUMA then prefetchers	1.78	1.66	1.72
Prefetchers then NUMA	1.75	1.67	1.71
Coupled search	1.82	1.73	1.77



(a) Intel Sandy Bridge **(b) Intel Skylake**

Figure 4: Maximum attainable speedup with respect to the number of configurations.

3.1 Experimental Setup

For our experiments we use two machines: a four-node Intel Sandy Bridge EP E5-4650 with 128 GB of RAM and a dual-node Intel Skylake Platinum 8168 with 188 GB of RAM. Codelets are generated using LLVM Clang version 3.8 [23, 33]. Our benchmarks come from Rodinia [7] and the NAS C Parallel Benchmarks [1, 34] version 3.0, along with LULESH version 2 [19, 21] and CLOMP [3].

All speedups presented in the paper are against an execution using all cores (32 for the Sandy Bridge and 48 for the Skylake), scattered among all nodes, and a locality-optimized page mapping with all prefetchers on. This is an optimized configuration which tries to increase the bandwidth (scattered thread mapping and locality page mapping) and reduce the latency (locality page mapping and prefetcher activation) over a simple first touch policy.

3.2 Performance Opportunities

We first look at the overall speedups that can be obtained from optimizing NUMA and/or prefetching in Table 1. NUMA optimizations alone (1.66× on average between Sandy Bridge and Skylake) are more significant than prefetcher optimizations alone (1.19× on average). The reduced NUMA sensitivity on Skylake is likely due to its faster interconnection links between nodes. Moreover, Sandy Bridge has more nodes, further increasing the severity of NUMA effects. Interestingly, the benefit from optimizing prefetchers is the same for both systems.

A greedy optimization for NUMA and prefetching (e.g., choosing one first, then picking the best choice for the second) delivers still better performance, but the best order depends on the system. Exploring all combinations of NUMA and prefetcher configurations (i.e., a coupled optimization) delivers slightly higher performance of 1.77× vs. 1.72× for the greedy NUMA-first optimization (on average). However, the majority of this benefit comes from one benchmark, K-means, which is able to find a particularly efficient configuration with the coupled optimization.

Figure 2 shows the per-region (rows) speedups (lighter is faster), across all NUMA and prefetcher configurations (columns), for Sandy Bridge. The results for Skylake follow a very similar structure. On the left side, the regions have been clustered using Ward's method on the normalized speedup vectors, showing that many regions share very similar speedup patterns across the configurations.

This representation allows us to see which regions benefit from similar NUMA+Prefetcher optimizations. For instance, BT (`x_solve`, `y_solve` and `z_solve`) has very similar behavior to MG residual, as they have similar access patterns (when the pages are appropriately mapped). As a result, they are clustered together. Similarly, all CLOMP regions, except the barrier, show similar sensitivities and are grouped together (Figure 2, top 10 rows).

Our clustering shows that many benchmarks share common behaviors, suggesting that clustering behaviors or optimizations may be effective.

3.3 NUMA+Prefetcher Configuration Diversity

Figure 2 showed that there is significant *similarity in region behavior* across the NUMA+Prefetcher configuration space. In Figure 4 we explore the similarity of optimized NUMA+Prefetcher configurations across the parallel regions. This figure shows the speedup we can achieve with a limited number of configurations compared to the full exploration. Here we see that, by only allowing the subset of the 11 best NUMA+Prefetcher configurations, we can achieve over 98 % of the maximum speedup, and when considering the best 13, we can achieve 99 %. These results come about for two reasons: the first is that regions have similar behaviors, as seen previously, and the second is that in many cases different NUMA+Prefetcher configurations give essentially the same performance benefit.

3.4 Takeaway

Our analysis shows that many benchmarks behave in similar ways and that we can achieve nearly all of the speedup potential with a very small set of NUMA+Prefetcher configurations. This suggests that it will be possible to build a model that can recognize application behaviors (since there is a limited number of them) and predict a very good NUMA+Prefetcher configurations (as only a few are needed to cover most of the benefit).

4 PREDICTION MODEL

A brute-force approach to optimizing NUMA+Prefetcher configurations would take as input the performance of all possible configurations and choose the best one as output. This guarantees the best performance but comes with the very high overhead of evaluating all configurations. To address this overhead, we train a *prediction model* that takes far fewer configurations as input but can still choose among a large enough subset of all possible NUMA+Prefetcher configurations as output to achieve good performance.

Building an effective model requires co-designing the subset of the input configurations to evaluate and the output configurations

Table 2: Prediction model parameters. Single/multi-label models use the same parameters.

Model	Parameter
ANN	lbfgs, alpha=0.0001, hidden_layer_sizes=(7,)
Tree	—
SVM	gamma=scale, decision_function_shape=ovo
LR	random_state=0, solver=lbfgs, multi_class=multinomial
Clustering	Hierarchical Ward (Euclidian distance)

to choose from while training the model to accurately map between them. The subset of inputs reduces the profiling overhead but still allows us to observe application differences, while the subset of output configurations allows us to simplify the model, while still obtaining high performance.

4.1 Machine Learning Models

We use the Python scikit-learn [31] package to train multiple types of models (Artificial Neural Network (ANN), Logistic Regression (LR), Tree, Support Vector Machine (SVM), and Clustering) using the parameters in Table 2. From Section 3 we saw that:

(1) Only 13 NUMA+Prefetcher output configurations are needed to cover 99% of the potential performance gains (Section 3.3).
(2) Many codes behave in a similar way across different NUMA+Prefetcher optimizations (Section 3.2).

These two observations guide our training of different types of models. To take advantage of 1), we use supervised learning (ANN, LR, SVM, Tree) to directly predict from among only the 13 overall most efficient configurations, instead of across all possible configurations (288 for Sandy Bridge and 320 for Skylake). We also train Tree and ANN models using multi-labels: all configurations that perform within 95 % of the best configuration are labeled as best, instead of just labeling the best one.

To take advantage of 2), we use unsupervised clustering to group regions by similar optimization choices. Unlike supervised learning, clustering cannot directly assign a configuration to a code. Therefore, we select the centroid of each cluster (in the feature space) and use its configuration across all the other regions within the same cluster. We expect efficient features to gather together regions that share the same optimal configuration. During validation we measure the features of the new regions and assign them to an existing cluster, and use its centroid-selected configuration.

Finally, we note that some models take much more time to train than others. For example, creating a single-label Tree model is 100× faster than a multi-labeled ANN. This directly affects the number of input feature pairs we can explore in training.

4.2 Model Generation and Inputs

As shown in Figure 5, the models we train take as input hardware performance counter values for a region executed with different NUMA+Prefetcher configurations and predict the best NUMA+Prefetcher configuration for that region. We identify the best configuration (correct prediction) through brute-force measurement (a) of the execution time for all NUMA+Prefetcher configurations for each region (see Section 2). Our training input features consist of hardware performance counter measurements for each region for all NUMA+Prefetcher configurations (b). With these input features (c) and the correct prediction data (d), we can train a variety of models (e) to predict the best NUMA+Prefetcher configuration[3].

4.2.1 Model Inputs. Hardware performance counters provide precise information on the interaction between the application and system. However, while these metrics are valuable for characterizing applications, they are not standard across systems. To address this, we use Likwid 3.0 [41] to abstract them to higher-level *performance groups*. We select memory-system related measurements, resulting in 19 performance counters from the Likwid NUMA, L2 CACHE, and L3 CACHE groups, as well as energy/power measurements from RAPL [10].

However, using only 19 data points for each application obtained with a single NUMA+Prefetcher configuration is not sufficient to train an accurate model (see analysis in Section 5.3). We therefore collect performance counter data for *all NUMA+Prefetcher configurations*, which increases our training set by a factor of 288. This is particularly valuable as we observe that the same performance counter profiled with different NUMA+Prefetcher configurations can return significantly different values, giving us information on the impact of changing the NUMA+Prefetcher configuration.

This technique of amplifying application data by changing the execution environment and observing the reaction is inspired by previous work in compilers [6, 43]. That work measured execution times across different compiler settings and built models to predict configurations based on the programs' reactions. We extend this concept by considering more diverse metrics given by the hardware performance counters (e.g., local accesses, bandwidth). This allows us to feed our prediction model with more diverse information to improve its accuracy. We call the resulting features **reaction-based performance counters** as they show the reaction of hardware performance counters to NUMA+Prefetcher configurations.

Unfortunately, while using reaction-based performance counters as input features to our models increases our data for training, collecting them can require up to 5472 executions for each region (19 counters × 288 NUMA+Prefetcher configurations)[4] to measure the features and another 288 per code for measuring the performance. However, these executions are a one-time cost for generating training data for the model.

We can reduce the need for profiling by using the features from one system to train models for another system. For instance, we can use the reaction-based performance counter input features from Sandy Bridge together with the ground truth (best configuration execution time) from Skylake to develop a model for Skylake. This reduces the overhead to only the 288 performance measurements to train the Skylake model. We demonstrate the accuracy of this cross-training in Section 5.1.2.

[3] We do not consider first touch as a possible page mapping because it assigns pages based on how the developer designs the first accesses rather than on fixed rules. Therefore, first touch page allocation strategy differs across applications, making it an inconsistent choice for our model.
[4] On Intel machines this is reduced to 1440 executions as Likwid and RAPL measure multiple counters at the same time. For codelets we require an initial warmup execution prior to the profiling execution [11].

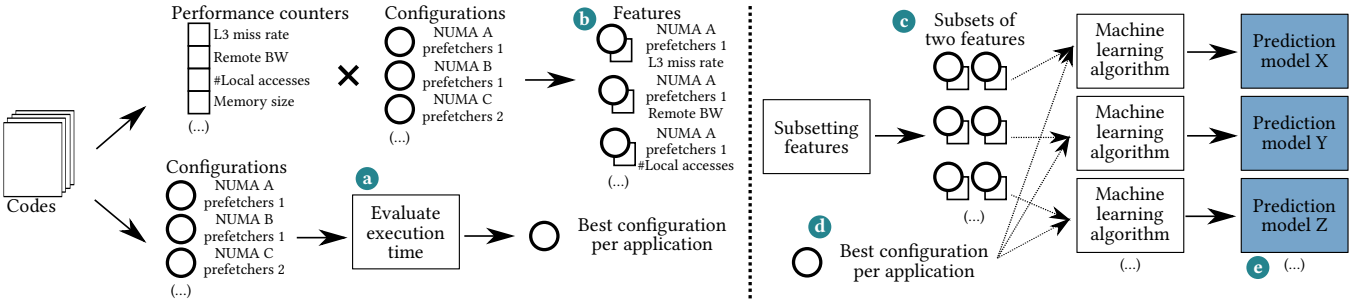


Figure 5: Model training. Left: Training data collection, including brute-force evaluation of execution time for all configurations. Right: Training each machine learning method across multiple subsets of the input features to identify the best combination of input features and model.

4.2.2 Feature Selection and Generation (Subsetting Features). To reduce the input needed for the model, we train models using only a *subset of the input features* for each region, specifically, two sets of performance counters and NUMA+Prefetcher configurations[5]. This allows us to benefit from the additional information provided by the reaction-based performance counters while keeping the profiling cost low: only two runs are required. To identify which subset of two features is most effective (i.e., contains helpful information for choosing configurations), we train models for many subsets of 2 input features and select the most efficient one. For each machine learning method, choosing the best model requires training for 20.4 million subsets[6]. As part of training, we use a standard 10-fold cross-validation (Section 5.1) to validate our models' robustness.

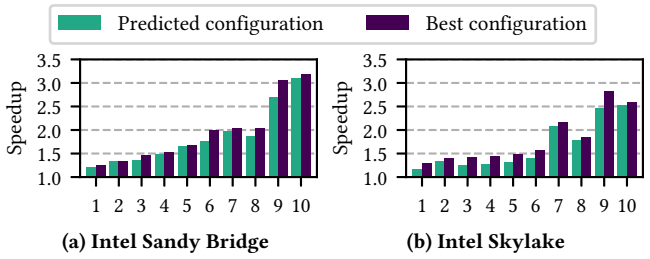


Figure 6: 10-fold cross-validation showing robust prediction across unseen regions and close to optimal predictions. Skylake trained with Sandy Bridge profiles. (See Section 4.2.2.)

5 PREDICTION RESULTS

We now evaluate the ability of our model to predict configurations for new regions. We use the same system setup and regions presented in Section 3.1: we collect the reaction-based performance counters (model input features) on Sandy Bridge and do a brute-force execution time exploration for all configurations on both Sandy Bridge and Skylake. With this data (input features and execution times), we train models for predicting NUMA+Prefetcher configurations for Sandy Bridge and Skylake. We then evaluate the resulting predictions on both systems vs. the ground-truth brute-force execution time exploration.

5.1 Model Evaluation

5.1.1 Model Validation. Each model takes as input two features and provides a prediction of the best configuration. To evaluate the quality of the models, we quantify the performance loss between the model-predicted configuration and the brute-force best configuration on *unseen codes* with cross-validation. Cross-validation shuffles all codes and splits them into groups (folds) of similar size. Each fold is then separately used as validation set for the model trained over remaining codes. If the training accuracy is consistently high across the different folds, then it indicates that the model is able to effectively generalize to unseen codes.

5.1.2 Cross-Validation. We illustrate in this section how cross-validation evaluates a model. We take as an example our best model for both Sandy Bridge and Skylake, Tree single-label (see Table 3).

Figure 6a presents the cross-validation for Sandy Bridge. It shows the results of training the model on the benchmarks in the other folds and then using the resulting model to predict the best configuration for the remaining fold. Here all but 2 folds (6 and 9) show small differences between the model prediction and the brute-force best choice, indicating that the model has generalized well. The comparison of the predicted configuration to the best configuration (dark) shows the model is effective, obtaining 95 % of the possible speedup. The reason why folds 6 and 9 show worse prediction results is because CG residual and K-means, respectively, have distinct behaviors (see Figure 2), and they were not included in the benchmarks used for training in these cases. However, the predicted configuration is still equivalent to or better than just optimizing for NUMA (page and thread mapping), and we expect that these mispredictions would be addressed by training on more codes.

Figure 6b shows the validation of the model which predicts configurations on Skylake based on performance counters collected from Sandy Bridge: i.e., the two input features are from reaction-based performance counters on Sandy Bridge, with only the output configuration performance measured on Skylake. This model has

[5]We empirically observed that two features provide good results from our models. Adding more features may achieve higher accuracy but causes a combinatorial explosion of the exploration space. We tried to only use subsets of many features collected within a single NUMA+Prefetcher configuration, but did not achieve good results.

[6]The exploration considers 21 NUMA configurations instead of 18, separating single, balance and interleave in 1-node settings, giving a total of $21 \cdot 16 = 336$ NUMA+Prefetcher configurations, with 19 counters giving $336 \cdot 19 = 6384$ total features. This results in $\binom{6384}{2} \approx 20.4\,\text{M}$ subsets of 2 features.

Table 3: Best model parameters for each ML method, including the reaction-based performance counters selected as the two profile inputs. (Thread mappings: remote, locality, single, interleave, balance, contiguous. Number of model evaluations in the 2-day training period.) While the DRAM Power performance counter was selected for many of the models, the second performance counter (and the NUMA+Prefetcher configurations) varied significantly.

		First feature (best model)						Second feature (best model)							
				NUMA configuration						NUMA configuration					
System	ML method	Perf. counter	HW pf.	# th.	# nodes	Th. map.	Page map.	Perf. counter	HW pf.	# th.	# nodes	Th. map.	Page map.	Model speedup	# eval. models
Sandy Bridge	**Tree_s**	**Package Power**	**1111**	**8**	**1**	**—**	**local**	**Package Power**	**1011**	16	**2**	**contig.**	**balance**	**1.76**	20 368 k
	LR_s	DRAM Power	0110	8	1	—	local	Package Power	1011	32	4	scatter	locality	1.43	3349 k
	Clustering	DRAM Power	0001	8	1	—	remote	DRAM Power	1101	8	1	—	remote	1.56	641 k
	ANN_m	DRAM Power	1011	8	1	—	remote	Remote DRAM BW	1011	16	2	scatter	interl.	1.65	146 k
	ANN_s	DRAM Power	0101	8	1	—	local	Energy	0001	32	4	scatter	interl.	1.66	284 k
	SVM_s	Core Power	0100	8	1	—	local	Core Power	1010	8	1	—	local	1.70	20 043 k
	Tree_m	Energy	1100	16	2	scatter	single	Package Power	1010	32	4	contig.	locality	1.74	11 075 k
Skylake	**Tree_s**	**Package Power**	**1101**	**16**	**2**	**contig.**	**locality**	**L3 miss ratio**	**1110**	**16**	**2**	**scatter**	**single**	**1.60**	20 368 k
	LR_s	DRAM Power	1101	8	1	—	remote	Package Power	1011	8	1	—	local	1.40	3349 k
	Clustering	DRAM Power	1001	8	1	—	local	L2 miss ratio	1111	8	1	—	local	1.40	643 k
	ANN_m	DRAM Power	1100	8	1	—	local	L2 miss ratio	0110	32	4	contig.	locality	1.51	144 k
	ANN_s	DRAM Power	0111	8	1	—	remote	DRAM Power	1101	32	4	scatter	locality	1.53	293 k
	SVM_s	DRAM Power	1001	8	1	—	remote	Local DRAM BW	1001	16	2	scatter	balance	1.52	20 236 k
	Tree_m	Energy	1111	16	2	contig.	interl.	Remote DRAM Volume	1010	8	1	—	local	1.59	11 094 k

higher variability between the folds and the best configuration, indicating the model is less-well generalized, and slightly less effective (92 % of the possible speedup). This is likely due to Skylake-specific behavior that is not visible in the Sandy Bridge performance counters used for training, and shows the trade-off for reducing the training overhead by reusing the Sandy Bridge input data.

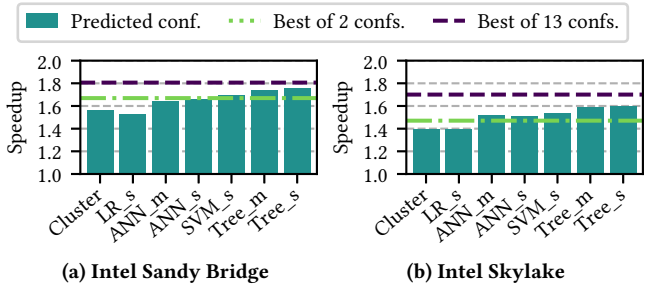
Our models' geometric mean performance gains, over a locality-optimized baseline with all prefetchers enabled, are 1.76× (Sandy Bridge) and 1.56× (Skylake) across all the folds. This shows that our best models provide significant speedups while remaining robust across new unseen benchmarks.

For Skylake, the selected input features are **L3 Miss Ratio** and **Package Power**, but with very different prefetcher and NUMA configurations (see Table 3). Package Power measures the power consumption of an entire node, including cores, last level cache, and memory controller. It is interesting to note that for Sandy Bridge the selected input features also measure Package Power, but profile it on two very different configurations (i.e., different prefetchers, core counts or nodes). In other words, the performance change across configurations is a useful information to guide the configuration prediction. This illustrates how the reaction-based performance counters allow us to include sensitivities to system configurations as model inputs.

5.2 Comparing Machine Learning Methods

5.2.1 Performance Gains. We evaluate the 7 different machine learning methods described in Section 4.1. For each method and system, Figure 7 reports the geometric mean performance gain across all folds of the most efficient model.

For performance analysis we provide two baselines that use our brute-force evaluation to pick the best configuration across subsets of the whole NUMA+Prefetcher search space. We define *best of 2* and *best of 13* as optimization strategies to compare with. Best of 13 selects the best optimization for each region from the 13 output NUMA+Prefetcher configurations that our models choose among. As illustrated in Figure 4, best of 13 provides similar performance



(a) Intel Sandy Bridge **(b) Intel Skylake**

Figure 7: Geometric mean performance gains of the most effective prediction model for each Machine Learning method. _s/_m refer to single label/many-labeled training.

to a full brute-force exploration and is therefore used as oracle. Best of 2 picks the best configuration from only the two most efficient overall configurations. That is, best of 2 has the same input overhead as our models: the user runs two configurations and picks the best one. This allows us to separate out the contribution of the model from the choice of model inputs.

Figure 7 shows that there is a significant difference across the the models: 1.2× between the best (Tree) and worst (Logistic Regression). However, the methods show similar relative results across the systems. In both cases, Tree (both single and many-labeled) gives the most efficient model while Logistic Regression is the worst. Finally, SVM and Tree outperform the best of 2 brute-force approach on both systems, demonstrating our model capabilities.

5.2.2 Understanding Why Some Methods Are More Efficient. To understand why Tree is the most efficient method, we need to describe in details how we train. Each method iterates over the subsets of 2 input features, trains a model for those input features, and then does a cross-validation for the resulting model. For each method, the best performing set of input features is chosen. This process takes longer for methods that are slower to train, which therefore limits the number of input feature subsets that can be evaluated.

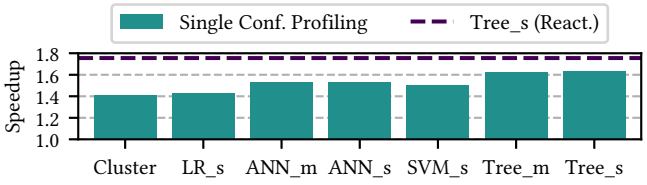


Figure 8: Single-configuration profiling vs. the best Reaction-based Performance Counter approach (Tree_s React.).

For each method we allocated a budget of 2 days of training per system. This limits the number of input subsets explored for the methods that take longer to train. Table 3 shows the size of this effect.

We observe that Tree_s was able to evaluate the whole search space of input subset features, thus exploring 100× more input subsets than ANN many-labeled in the 2 day training time. As a result, it is possible that the more time-consuming methods did not have a chance to evaluate some particularly good combinations of performance counters. To partially mitigate this, we explored over 10 000 random subsets, as well as trying the best input subset from Tree_s with the ANN. Unfortunately, these approaches did not improve the performance, suggesting that *input features need to be selected together with the method for best efficiency*. This is particularly interesting as we see that some performance counters such as DRAM Power (included in 5 out of 7 models) carry valuable information, but are not used in Tree_s.

We conclude that Tree is not necessarily the best method, but its combination of faster training time and good prediction allows it to beat other models that may be more accurate, but take longer to train. Therefore, investing more time to train more time consuming methods such as ANN may further improve the gains[7].

5.3 Reaction-Based Performance Counters Improve Modeling

In the previous results, our models were all trained using the reaction-based performance counters (e.g., inputs were a performance counter measured on a specific NUMA+Prefetcher configuration). To quantify the value of this measuring the performance counter on the default NUMA+Prefetcher configuration, we compare our results with models that are only allowed to choose their 2 input features from a single NUMA+Prefetcher configuration.

Single-configuration profiling drastically reduces the number of model inputs to only $19 \cdot 19 = 361$. As a result, each machine learning model was able to evaluate all possible 2-input feature subsets. Figure 8 compares single-configuration-trained models[8] to our reaction-based performance counter models. We see that training on only a single configuration significantly hurts performance (down to 93 % of the Reaction-Based Performance Counter approach for Tree_s), even thought it is able to explore all possible input combinations. This showcases that the overhead of having to evaluate codes on multiple configurations pays off in better modeling, which leads to better performance.

[7]We did not consider Deep Neural Networks due to our small input training set of only 57 parallel regions.

[8]Single-configuration for Sandy Bridge: 32 threads, 4 nodes, scatter, locality, and all prefetchers on.

5.4 Takeaway

We have shown that our Tree-based model can *robustly* predict combined NUMA+Prefetcher configurations that deliver a *geometric mean speedup of 1.74×* (compared to 1.82× for an oracle), even against a locality-optimized baseline with all prefetchers enabled. We observe that the Tree model wins out due to combination of faster training (which allows greater exploration of input combinations) and good prediction accuracy. Further, we have shown that the reaction-based performance counters provide significant benefits by exposing sensitivity to system configurations.

In this study we also observed that the Package Power performance counter appears to provide the best indication of overall configuration fitness, as it is chosen as an input to the best-performing machine learning models. This choice is unexpected, as IPC (instructions per cycle) or MPKI (misses per thousand instructions) are typically chosen to represent overall performance. Our analysis is that Package Power performs better as it takes into account both CPU activity (as IPC does) and cache behavior (as MPKI does), as well as CPU idle times, link utilization, and DRAM accesses, making it a robust overall metric.

6 OPTIMIZING APPLICATIONS ONLINE

6.1 Online Profiling and Optimization

The method described up until now makes predictions using information from offline profiling: first, extract codelets of the OpenMP regions and execute them with the two selected NUMA+Prefetcher configurations while recording the appropriate hardware performance counters, then use the model to predict the best NUMA+Prefetcher configuration. However, as the application itself consists of repeated executions of the OpenMP regions, we can move this profiling online by carefully setting the NUMA+Prefetcher configurations between the OpenMP region executions as the application runs and profiling online. Doing so allows us to collect the required input information online with only the overhead of changing the configuration for the two regions and appropriate warmup.

Figure 9a illustrates the online profiling of OpenMP region A as the application executes. The first instance of region A is used to collect the access patterns (Address Trace) needed for the NUMA locality and balance policies[9]. We then need to execute the region for each of the two input NUMA+Prefetcher configurations required for the model (conf. 1 and conf 2. in the Figure). However, this incurs two sources of overhead: First, we may need to migrate threads and pages if the current configuration does not match the configuration we need to measure. And, second, we execute the OpenMP region once before measuring to warmup the caches, and this execution may experience significant cache misses due to the migration. After that setup, we profile the next execution of the region (Profile) and use the model to predict the best configuration. With the model prediction, we migrate pages and threads as needed (Migration), and then execute with the chosen configuration (Optimized execution). This approach assumes that parallel regions have similar behavior, which is quite common in our benchmark applications [34, 40].

[9]The overhead of collecting this information can be reduced to 12 % [40], but in this work we use Pin directly, which is around 10× slower.

Code region	A		A	A		A	A		A
Exec. step	Address Trace	Migration conf. 1	Warmup conf. 1	Profile conf. 1	Migration conf. 2	Warmup conf. 2	Profile conf. 2	*Migration conf. A*	Optimized execution

(a) Obtaining the best configuration of a region with a single run.

Code region		A	B	A	B		A	B	A	B		A
Exec. step	Migration conf. A	Warmup conf. A	Warmup conf. A	Profile conf. A	Profile conf. A	Migration conf. B	Warmup conf. B	Warmup conf. B	Profile conf. B	Profile conf. B	*Migration best conf.*	Optimized execution

(b) Optimizing whole applications based on the per-region configurations. A and B conflicts are addressed by cross-evaluating their configurations and selecting the best one overall.

Figure 9: Online evaluation of different configurations.

6.2 Whole-Application Optimization

While our prediction model finds the most efficient configuration for each parallel region, it does not consider how regions interact. This is a problem if the NUMA page optimizations for one region hurt the performance of another region as explicit page migration between regions is generally too costly [27]. Prefetching and thread configurations can be changed with little overhead between regions, making them less problematic.

Such inter-region configuration conflicts have been shown to have a small impact on overall performance[10]. Popov et al. [35] reported that only a few applications in the NAS and Rodinia benchmarks, such as BT ans SP, are significantly affected, and the overall reduction in speedup when accounting for inter-region conflicts was only 6%. We therefore evaluate the impact of inter-region conflicts on our online profiling approach for both BT and SP.

Our approach is similar to the one proposed by Popov et al. [35]. First, we select the best configuration for each region individually. We then check the configurations for conflicts (i.e., pages mapped to different nodes). If conflicts are found, we *cross-evaluate* the configurations of the conflicting regions and select the best overall performing configuration. Figure 9b shows the phases of this cross-evaluation: migration, warming, and profiling A and B with A's configuration (left) and, similarly, migration, warming and profiling A and B with B's configuration (right).

The cost and accuracy of online cross-evaluation is shown in Figure 10 for the `x_solve` region in BT (top) and the `y_solve` region in SP (bottom). The figures show cycle counts for each step in the cross-evaluation of those two regions. From left to right: the region is first executed twice with first touch to observe the non-optimized configuration behavior, followed by migration/warmup/execution of the region with the configurations for the `rhs`, `x_solve`, `y_solve` and `z_solve` regions. For example, Ⓐ/Ⓑ/Ⓒ in the top figure show Ⓐ the cycles for migrating pages for the `rhs` NUMA+Prefetcher configuration, Ⓑ the cycles for executing the `x_solve` region with the `rhs` configuration for cache warmup, and Ⓒ the cycles for the profiling of the `x_solve` region with the `rhs` configuration. The accuracy of the resulting online profile can be seen by comparing the online measured execution time for the region with the different configurations to the offline profiled one (e.g., for the `rhs` configuration, this is comparing the online measured time in blue, left, to the offline measured time, blue, right).

Table 4: Execution times (billions of cycles) of BT and SP region conflicts and online profiling. The overhead is quickly amortized since each region is called hundreds of times.

	Locality, 32 threads, scatter (baseline)	Ignoring inter-region	Offline best inter-region	Online best inter-region
BT	33.5	16.3 (2.1×)	22.9 (1.5×)	23.9 (1.4×)
SP	179.8	45.2 (4.0×)	53.5 (3.4×)	54.9 (3.3×)

Figure 10 shows that there are many more cycles spent in migration for BT (top) than SP (bottom). This is because BT's `x_solve` region has a different access pattern from its `y_solve` and `z_solve` regions [27], causing more pages to be migrated. Conversely, the three regions from SP all use a single NUMA node optimization, significantly reducing the migration cost.

For online profiling/optimization to be effective, the overhead needs to be less than the benefits. Our approach has three sources of slowdown: migration, warmup, and profiling[11]. These overheads are only paid once at the beginning of the application and we find they are quickly amortized by the speedups obtained.

Table 4 shows the execution time of the applications BT and SP with four configurations: our locality-optimized baseline applied to the whole application, ignoring inter-region conflicts and using each region's best configuration, the best overall configuration found offline, and the best overall configuration found online, including profiling overhead. For both applications there are enough page conflicts that the cost of migrating pages between regions to use each region's independently optimal configuration is prohibitive. However, our online approach finds the overall best configuration and delivers 95 % of the offline performance.

[10] Speedups reported so far are for regions and do not include these effects. In this section we evaluate two applications where these effects are significant.

[11] Note that the cross-evaluation profiling itself will incur slowdowns if the configurations being cross-evaluated are a poor match for the region being evaluated. E.g., `y_solve` takes over three times as long to complete when being profiled with the `rhs` configuration (Figure 10 bottom, "Profile rhs").

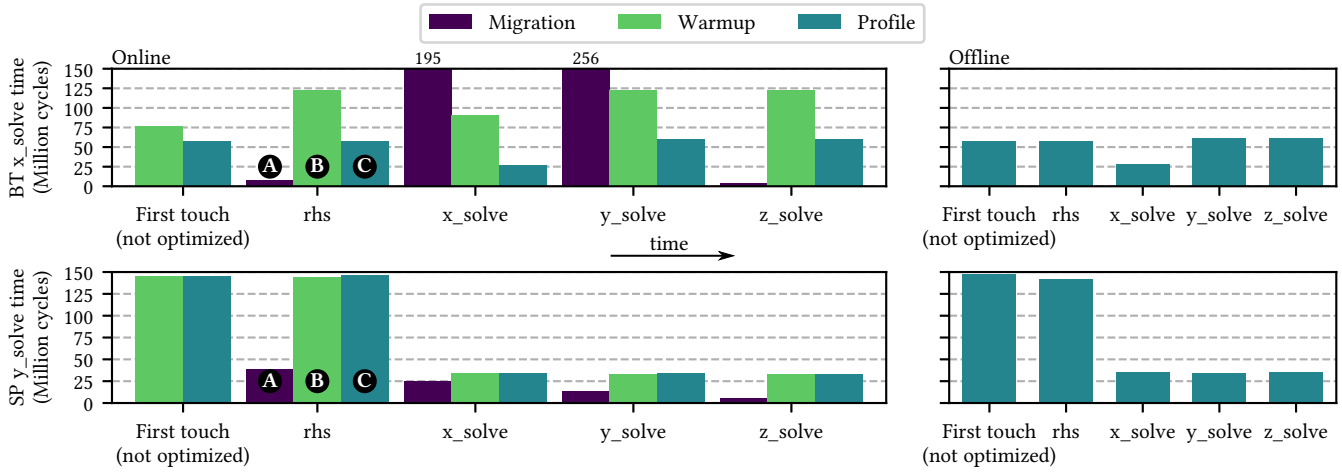


Figure 10: Left: Overhead costs of online cross-evaluation of the parallel regions `x_solve` (top) and `y_solve` (bottom) across the best individually chosen configurations for regions `rhs`, `x_solve`, `y_solve`, and `z_solve`. Cross-evaluation requires first migration A, then warmup B, and finally profiling C for each configuration. Right: Average execution time for the region when optimized for each configuration offline. Similar online/offline results show the accuracy of online profiling.

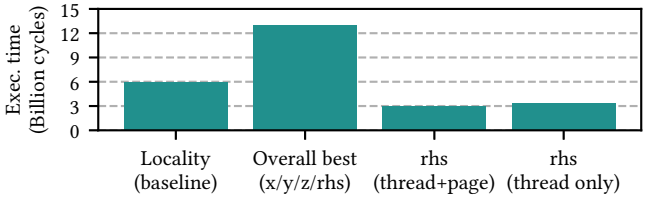


Figure 11: Execution cycles for `rhs` from SP. *Overall best* is the best configuration for the whole application, *thread only* uses the `rhs` optimal thread configuration with the overall best page mapping, and is only slightly slower than the combination `rhs` optimal thread- and page-mapping, but can be achieved without the overhead of migrating pages.

6.3 Per-Region NUMA Optimization

The overhead of page migration makes it too expensive to adapt on a per-region basis. However, it can be profitable to change the thread mapping on a per-region basis, once the overall best page mapping has been chosen. As an example, our method identifies a case in the SP benchmark where different regions should have different degrees of parallelism. This is shown in Figure 11, for the case of the `rhs` region when executed on the Sandy Bridge system.

In this case, the best configuration for `rhs` uses 32 threads and maps the pages across the whole system, while `x_solve`, `y_solve`, and `z_solve` optimally use only 8 threads and map all pages to just one node. Changing the page mapping between these regions costs more than the potential gains, but the reconfiguration overhead of changing the thread mapping is negligible. By reconfiguring threads, but not pages, we can avoid the costly page migration while retaining many of the benefits from the better thread mapping (3.9× improvement from changing just the thread-mapping on a per-region basis vs. 4.4× for changing both thread- and page-mapping on a per-region basis, compared to the overall best all-region configuration), which also outperforms the baseline locality-optimized mapping. The end result is that the page mapping is stable throughout the execution of the application while we change the thread mapping on a per-region basis, at a negligible cost.

A more sophisticated approach would be to consider which subsets of threads, pages, and prefetchers should be optimized across conflicting configurations, but we leave that for future work.

7 RELATED WORK

Our work uses online performance prediction models to optimize both NUMA configurations and prefetching. Surveys of these broad areas were done by Diener et al. [16] and Mittal [30], respectively, and an overview of closely related work is presented in Table 5.

Online methods [4, 9, 38, 45] profile and make optimization decisions as the application runs. However, this necessitates very low overhead profiling to avoid outweighing the optimization gains. **Offline** [2, 15, 40] methods avoid the need for low-overhead profiling, but require an additional execution and can be sensitive to changes in input data.

Direct Performance Evaluation methods measure the performance of configurations [37, 45] across a search space, which requires an effective search strategy. While accurate, this can be time consuming as it requires executing all or part of each application for each configuration. **Performance Prediction Models** use input features to predict the best configuration [24, 44]. Commonly used features include thread access patterns [13], performance counters [26, 42], static code properties [44], and page/thread or inter-thread communication and sharing [15].

NUMA - Prediction Models. Performance counters are commonly used inputs to performance models. Wang et al. [42] use Integer Programming to predict bandwidth usage and thread allocation across different degrees of NUMA nodes, while Denoyelle et al. [13] add thread and page mappings. They conclude that performance counters and thread access patterns provide similar results

Table 5: Related prediction approaches. (Castro et al. [5] explore Round-Robin and Scatter, which produce the same mappings on our systems with private L2s. Wang and O'Boyle [44] evaluated scheduling policies.)

			Optimizations			
	Inputs	**Models Evaluated**	**Parallelism**	**Thread Placement**	**Page Placement**	**Prefetch**
Denoyelle et al. [13]	Performance Counters, Thread Access Patterns	LR, SVM, Random Forest, ANN		Scatter, Contiguous	First touch, Interleave	
Wang et al. [42]	Performance Counters	Integer programming	Threads, Nodes			
Wang and O'Boyle [44]	Performance Counters, Static Code Metrics	ANN, SVM, Decision Trees		Scatter, Contiguous		
Castro et al. [5]	Microarchitecturally-independent metrics	Analytical	Threads			
Liao et al. [26]	Performance Counters	NN, Naive Bayes, ANN, Tree, Ripper Classifier, SVM, LR				Intel
Hiebel et al. [18]	Performance Counters	Smooth 0-1 Loss Approximation				Intel
This Work	**Reaction-based Performance Counters**	**LR, SVM, ANN, Tree, Clustering**	**Threads, Nodes**	**Scatter, Contiguous**	**Interleave, Locality, Balance, Single, Local**	**Intel**

as inputs. In both cases, inputs are collected by executing the application. Piccoli et al. [32] proposes migrating pages for locality by estimating array reuse during execution. Similarly, Carrefour [9] triggers page migration for page balance by monitoring memory performance counters. However, they only target a subset of the NUMA mappings we consider (specifically not locality and balance) and therefore miss opportunities.

NUMA - Performance Evaluation. Direct execution avoids the inaccuracies of modeling, but costs time to execute the application for each configuration. Exploration strategies have evaluated different thread mappings [36, 37] and degrees of parallelism [17]. CERE [33, 35] used codelets to more rapidly evaluate configurations, allowing the simultaneous exploration of thread and page mappings for further benefit.

Prefetching - Prediction Model. Liao et al. [26] proposed a tuning frameworks that predicts the best prefetching based on performance counters. Hiebel et al. [18] extended this prediction to target more fine grained phases of execution.

Prefetching - Evaluation. Adaptive prefetching schemes [20, 22, 45] have been proposed to directly evaluate different configurations. These approaches optimize only prefetching, which can lead to sub-optimal performance on NUMA systems, as seen in the K-means application.

Prefetching for NUMA. Disabling all prefetching on NUMA systems can improve performance for irregular access patterns [28]. Moreover, prefetching can also increase the contention and hurt performance [25]. By ignoring the full spectrum of NUMA effects on hardware prefetchers, previous works sacrificed significant performance potential, as described in Section 3.2.

8 CONCLUSION

In this work we have shown that there is a significant performance benefit from optimizing the NUMA configuration (parallelism, thread-, and page-placement) together with hardware prefetcher configurations (L1, L2). However, this benefit comes at the cost of a very large design space to explore. We tackled this problem by developing an efficient and robust performance model. We reduce the overhead of collecting data for the model by identifying two reaction-based performance counter configurations (combinations of NUMA+Prefetcher and performance counters) which allow our model to accurately predict the best configuration, and reducing the number of configurations the model has to choose among by analyzing how we can reduce the configuration space without losing performance. During training, we saw the importance of selecting the correct reaction-based performance counters as inputs for each specific model type and system and how we can do more efficient cross-system training by reusing input data. We then demonstrated how this approach can be applied for online profiling and optimization to deliver an average of 1.68× performance increase over a NUMA-locality-optimized baseline with all prefetchers enabled. Finally, we observed that in rare cases applications can suffer from inter-region page-mapping conflicts. Using the best overall configuration and changing parallelism across regions partly overcomes the performance loss and improves over the already-optimized baseline.

ACKNOWLEDGMENTS

This work has been supported by the European Research Council grant Nos. 715283 and 321253, the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows program (No. 2015.0153), the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), the European HiPEAC Network of Excellence, and the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

I. Sánchez Barrera has been partially supported by the Spanish Ministry of Education, Culture and Sport under Formación del Profesorado Universitario fellowship number FPU15/03612 and by the Spanish Ministry of Science, Innovation and Universities under Ayudas a la movilidad para estancias breves y traslados temporales fellowship number EST18/00799. M. Casas and M. Moretó have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowship numbers RYC-2017-23269 and RYC-2016-21104, respectively.

I. Sánchez Barrera wishes to thank the Uppsala Architecture Research Team and other members from the IT Department of the Uppsala University for their welcome and all the great moments during the visit to Uppsala. In particular, thanks to Anastasiia, Chris, David, Gustaf, Hassan, Johan, Kim, Marina, Mehdi, Mihail, Paul, Per and Ricardo.

REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks — Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[2] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. 2015. TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures. In *Proceedings of the 2nd Workshop on Visual Performance Analysis* (Austin, Texas, USA) *(VPA '15)*. ACM, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2835238.2835239

[3] Greg Bronevetsky, John Gyllenhaal, and Bronis R. De Supinski. 2008. CLOMP: Accurately Characterizing OpenMP Application Overheads. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism* (West Lafayette, Indiana, USA) *(IWOMP '08)*. Springer, Berlin, Heidelberg, 13–25. https://doi.org/10.1007/978-3-540-79561-2_2

[4] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming* 38, 5 (2010), 418–439. https://doi.org/10.1007/s10766-010-0136-3

[5] Marcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. 2011. A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications. In *Proceedings of the 2011 18th International Conference on High Performance Computing* (Bangalore, India) *(HiPC '11)*. IEEE, USA, 10. https://doi.org/10.1109/HiPC.2011.6152736

[6] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. 2006. Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea) *(CASES '06)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/1176760.1176765

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization* (Austin, Texas, USA) *(IISWC '09)*. IEEE, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[8] Henry Cook, Miquel Moretó, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanović. 2013. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/2485922.2485949

[9] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[10] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design* (Austin, Texas, USA) *(ISLPED '10)*. ACM, New York, NY, USA, 189–194. https://doi.org/10.1145/1840845.1840883

[11] Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.* 12, 1, Article 6 (April 2015), 24 pages. https://doi.org/10.1145/2724717

[12] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. 2014. Fine-Grained Benchmark Subsetting for System Selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, Florida, USA) *(CGO '14)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2581122.2544144

[13] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. 2019. Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. ACM, New York, NY, USA, Article 39, 10 pages. https://doi.org/10.1145/3337821.3337893

[14] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. KMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, Alberta, Canada) *(PACT '14)*. ACM, New York, NY, USA, 277–288. https://doi.org/10.1145/2628071.2628085

[15] Matthias Diener, Eduardo HM Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O.A. Navaux. 2015. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88-89 (June 2015), 18–36. https://doi.org/10.1016/j.peva.2015.03.001

[16] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. 2016. Affinity-Based Thread and Data Mapping in Shared Memory Systems. *Comput. Surveys* 49, 4, Article 64 (Dec. 2016), 38 pages. https://doi.org/10.1145/3006385

[17] Juan J Durillo, Philipp Gschwandtner, Klaus Kofler, and Thomas Fahringer. 2019. Multi-Objective region-Aware optimization of parallel programs. *Parallel Comput.* 83 (2019), 3–21. https://doi.org/10.1016/j.parco.2018.03.010

[18] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. 2019. Machine Learning for Fine-Grained Hardware Prefetcher Control. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. ACM, New York, NY, USA, Article 3, 9 pages. https://doi.org/10.1145/3337821.3337854

[19] Rich Hornung, Jeff Keasler, and Maya Gokhale. 2011. *Hydrodynamics Challenge Problem.* Technical Report LLNL-TR-490254. Lawrence Livermore National Laboratory. 1–17 pages. https://computing.llnl.gov/projects/co-design/spec-7.pdf

[20] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/2370816.2370837

[21] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory. 1–9 pages. https://computing.llnl.gov/projects/co-design/lulesh2.0_changes1.pdf

[22] Muneeb Khan, Michael A. Laurenzanoy, Jason Marsy, Erik Hagersten, and David Black-Schaffer. 2015. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)* (San Francisco, California, USA) *(PACT '15)*. IEEE, USA, 367–378. https://doi.org/10.1109/PACT.2015.35

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, California, USA) *(CGO '04)*. IEEE, USA, 12. https://doi.org/10.1109/CGO.2004.1281665

[24] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. 2007. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP '07)*. ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/1229428.1229479

[25] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. 2017. Analysis of NUMA Effects in Modern Multicore Systems for the Design of High-Performance Data Transfer Applications. *Future Generation Computer Systems* 74, C (Sept. 2017), 41–50. https://doi.org/10.1016/j.future.2017.04.001

[26] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon, USA) *(SC '09)*. ACM, New York, NY, USA, Article 56, 10 pages. https://doi.org/10.1145/1654059.1654116

[27] Zoltan Majo and Thomas R. Gross. 2012. Matching Memory Access Patterns and Data Placement for NUMA Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California, USA) *(CGO '12)*. ACM, New York, NY, USA, 230–241. https://doi.org/10.1145/2259016.2259046

[28] Zoltan Majo and Thomas R. Gross. 2013. (Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization* (Portland, Oregon, USA) *(IISWC '13)*. IEEE, USA, 11–22. https://doi.org/10.1109/IISWC.2013.6704666

[29] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. 2019. Toward Efficient In-memory Data Analytics on NUMA Systems. *arXiv e-prints* (2019), 15. arXiv:1908.01860v2 [cs.DB]

[30] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *Comput. Surveys* 49, 2, Article 35 (Aug. 2016), 35 pages. https://doi.org/10.1145/2907071

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. arXiv:1201.0490 [cs.LG] http://jmlr.org/papers/v12/pedregosa11a.html

[32] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, Alberta, Canada) *(PACT '14)*. ACM, New York, NY, USA, 369–380. https://doi.org/10.1145/2628071.2628077

[33] Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. 2017. Piecewise holistic autotuning of parallel programs with CERE. *Concurrency and Computation: Practice and Experience* 29, 15, Article e4190 (2017), 15 pages. https://doi.org/10.1002/cpe.4190

[34] Mihail Popov, Chadi Akel, Florent Conti, William Jalby, and Pablo de Oliveira Castro. 2015. PCERE: Fine-Grained Parallel Benchmark Decomposition for Scalability Prediction. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium* (Hyderabad, India) *(IPDPS '15)*. IEEE, USA, 1151–1160. https://doi.org/10.1109/IPDPS.2015.19

[35] Mihail Popov, Alexandra Jimborean, and David Black-Schaffer. 2019. Efficient Thread/Page/Parallelism Autotuning for NUMA Systems. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona, USA) *(ICS '19)*. ACM, New York, NY, USA, 342–353. https://doi.org/10.1145/3330345.3330376

[36] Petar Radojković, Paul M. Carpenter, Miquel Moretó, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2016. Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach. *IEEE Trans. Comput.* 65, 1 (Jan. 2016), 256–269. https://doi.org/10.1109/TC.2015.2417533

[37] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2150976.2151002

[38] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. 2018. Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. ACM, New York, NY, USA, 207–217. https://doi.org/10.1145/3205289.3205310

[39] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware Monitors for Dynamic Page Migration. *J. Parallel and Distrib. Comput.* 68, 9 (Sept. 2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[40] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. 2018. NumaMMA: NUMA MeMory Analyzer. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, Oregon, USA) *(ICPP 2018)*. ACM, New York, NY, USA, Article 19, 10 pages. https://doi.org/10.1145/3225058.3225094

[41] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for X86 Multicore Environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops* (San Diego, California, USA) *(ICPPW '10)*. IEEE, USA, 207–216. https://doi.org/10.1109/ICPPW.2010.38

[42] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture* (Barcelona, Spain) *(HPCA '16)*. IEEE, USA, 419–431. https://doi.org/10.1109/HPCA.2016.7446083

[43] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[44] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, North Carolina, USA) *(PPoPP '09)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/1504176.1504189

[45] Carole-Jean Wu and Margaret Martonosi. 2011. Characterization and Dynamic Mitigation of Intra-Application Cache Interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Austin, Texas, USA) *(ISPASS '11)*. IEEE, USA, 2–11. https://doi.org/10.1109/ISPASS.2011.5762710