Optimizing Performance and Energy Across Problem Sizes Through a Search Space Exploration and Machine Learning

Lana Scravaglieri, Mihail Popov, Laércio Lima Pilla, Amina Guermouche, Olivier Aumage, Emmanuelle Saillard

Optimizing Performance and Energy Across Problem Sizes Through a Search Space Exploration and Machine Learning

Lana Scravaglieri
Inria
Bordeaux, France
lana.scravaglieri@inria.fr

Laércio Lima Pilla
Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800
Talence, France
laercio.pilla@inria.fr

Amina Guermouche
Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800
Talence, France
amina.guermouche@inria.fr

Olivier Aumage
Inria
Bordeaux, France
olivier.aumage@inria.fr

Emmanuelle Saillard
Inria
Bordeaux, France
emmanuelle.saillard@inria.fr

Mihail Popov
Inria
Bordeaux, France
mihail.popov@inria.fr

***Abstract*—HPC systems expose configuration options that help users optimize their applications' execution. Questions related to the best thread and data mapping, number of threads, or cache prefetching have been posed for different applications, yet they have been mostly limited to a single optimization objective (e.g., performance) and a fixed application problem size. Unfortunately, optimization strategies that work well in one scenario may generalize poorly when applied in new contexts.**

In this work, we investigate the impact of configuration options and different problem sizes over both performance and energy. Through a search space exploration, we have found that well-adapted NUMA-related options and cache prefetchers provide significantly more gains for energy ($5.9\times$) than performance ($1.85\times$) over a standard baseline configuration. Moreover, reusing optimization strategies from performance to energy only provides 40% of the gains found when natively optimizing for energy, while transferring strategies across problem sizes is limited to about 70% of the original gains.

In order to fill this gap and to avoid exploring the whole search space in multiple scenarios for each new application, we have proposed a new Machine Learning framework. Taking information from one problem size enables us to predict the best configurations for other sizes. Overall, our Machine Learning models achieve 88% of the native gains when cross-predicting across performance and energy, and 85% when predicting across problem sizes.

I. Introduction

New High Performance Computing (HPC) systems come with an increasingly higher number of cores spread across multiple sockets and connected together with node-to-node communications links. While these links provide the programmer with a convenient shared memory view of the memory banks connected to distinct sockets, they also cause non-uniform latency and bandwidth among memory accesses. These Non-Uniform Memory Access (NUMA) effects can result in long latency, low bandwidth remote accesses, and congestion on the memory controller. Therefore, the programmer or the Operating System (OS) must carefully map the applications threads and data to avoid hurting performance.

To optimize latency, modern systems rely on complex memory hierarchies and hardware prefetching. The former attempts to reduce access time on frequently accessed data, while the latter attempts to fetch relevant pieces of data ahead of time. The CPU implements them using complex cache policies and data access pattern detectors to predict which cache line to evict or to fetch ahead of time. Yet, an excessively aggressive hardware prefetching can cause last-level cache interferences and defeat NUMA mapping optimizations [39]. It is therefore mandatory to jointly optimize NUMA mapping directives and hardware prefetcher configurations to achieve an efficient execution. Moreover, because of the software and hardware complexity and diversity in HPC, a unique strategy for NUMA mapping and prefetcher configuration cannot fit all usages in an optimal manner. Instead, a large parameter space must be searched to determine a successful parameter combination for executing a given application efficiently.

NUMA [14], [16], [17], [33], [39], [47] and prefetch [3], [19], [21], [22], [28], [39] optimizations have been extensively explored. They focus on selecting mapping policies that adjust the thread and data mappings across nodes or by configuring specific hardware prefetchers. To guide the selection of policies or prefetchers, studies use prediction models. They collect information (e.g., performance counters) about the application and supply it to Machine Learning models. Such models achieve substantial performance gains (i.e., $2\times$).

A limitation of these studies is that the behavior and performance of applications vary significantly across problem sizes [12] as illustrated by Figure 1 (horizontal axis). We observe how, for the same application, the optimal NUMA/prefetch configuration changes across sizes (i.e., the best performance optimization for size B, shown as a *green*

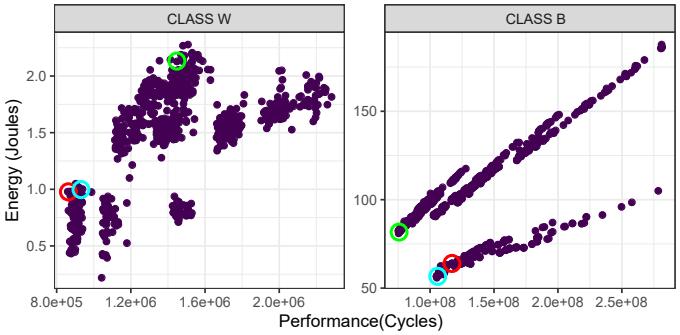

Fig. 1. [lower is better] Execution of x_solve, a parallel region from NAS-BT across different NUMA and prefetch configurations using 2 different sizes: CLASS W and B. X-axis: execution time per iteration. Y-axis: energy consumption per iteration. Each dot represents a possible execution with different NUMA and prefetch settings. Energy and performance behaviors change significantly across problem sizes. The red, blue, and green circles refer to the best settings for CLASS W/performance, CLASS B/energy, and CLASS B/performance respectively. The best settings change significantly depending on both the size and performance/energy.

circle, is $1.55\times$ faster than the best optimization for size W, the *red circle*, applied on B). This is sometimes expected, as an application executed with a small size can fit in the smaller but faster cache levels. It is thus essential to factor the impact of problem sizes when tuning an application in order to ensure that NUMA and prefetch optimizations are still valid when that application is later executed on new problem sizes.

Regarding the objective of optimization, energy consumption has become an important factor to consider. Due to their environmental impact as well as operating costs, HPC systems are constrained to operate on a predefined power budget. This further complicates the optimization process by adding energy as a new dimension to consider, as illustrated in Figure 1 (vertical axis). We similarly observe that the best configuration for performance is sub-optimal for energy and vice versa (i.e., for size B, the best energy optimization, shown as a *blue circle*, provides $1.47\times$ energy gains/saving over the best performance optimization *green circle*).

Applications exhibit different patterns of sensitiveness in terms of bandwidth, latency, and energy consumption, and are executed on complex systems with distinct combinations of NUMA factors (i.e., ratio of local to remote access latency), prefetcher capabilities, and topologies (e.g., number of nodes or cores). These characteristics cause different performance and energy bottlenecks depending on the application, system characteristics, prefetcher, problem size, and mapping. As a result, selecting the best mapping/prefetching combination strategy is a challenging task that requires modeling the aforementioned factors and the optimization target (i.e., performance and energy). Known previous works have solely modeled NUMA and prefetching strategies with a single size and with the unique goal of performance [39], [41]. These simpler models are easier to build, but may generalize poorly to alternate goals such as energy consumption optimization, or to applications executed in the real world across a number of different problem sizes.

In this work, **we are the first to co-optimize NUMA and prefetch strategies across multiple problem sizes and optimization goals**. We both quantify how distinct sizes affect the effectiveness of joint NUMA/prefetch optimization strategies, and characterize how tuning for performance impacts the level of energy efficiency and how that efficiency compares against optimizing for energy upfront (and vice versa). We show in this context that tuning performance only provides 40% of the potential energy savings. Moreover, applying optimizations from one size to another provides 95% and 51% of the potential performance and energy gains, respectively.

To exploit these significant optimization opportunities, we propose a **Machine Learning (ML) framework whose novelty consists in predicting NUMA/prefetch performance or energy optimizations across sizes**. Our model provides 96% and 74% of the potential performance and energy gains respectively. To achieve this accuracy, the model collects a few reaction-based performance and energy measurements as features [39], [49] on one problem size and uses them to predict configurations for another size.

Our contributions are:

- ⋆ Quantifying how problem sizes affect NUMA and prefetch optimizations. *Optimizations from one size are sub-optimal when applied on another size*.
- ⋆ Characterizing performance and energy in the context of NUMA and prefetcher configuration options. *Energy optimization has much more potential than performance but it needs to be studied separately*.
- ⋆ Analyzing the interaction between problem size, performance, energy, and the complicated optimization space.
- ⋆ Designing prediction models that suggest efficient NUMA and prefetch configurations across different problem sizes for both performance and energy.

II. Search Space Definition

This section describes the search space that we explore. A summary of its dimensions and options is shown in Table I. This optimization space is too large and complex for the use of conventional profiling methods. Its complexity comes from the interactions between different configuration options at software and hardware levels [33], and different problem characteristics related to the applications and optimization targets. The actual methodology employed for profiling the search space is detailed later in Section III.

A. Configuration Options

We divide the configuration options into two categories: user-level options, and administrator-level options (for which administrator privileges are required). We refer to a **dimension** as a specific configuration option. Dimensions are presented in this section in bold and followed by their abbreviations in small caps. A group of one or more dimensions is seen as a **subspace**. Finally, a **configuration** represents a setup for executing an application and refers to a specific point in the search space (i.e., each dimension has an assigned option).

1) User-level: NUMA: NUMA effects can be mitigated by optimized thread and data mapping decisions. For instance, the **number of threads** (PAR) to be used can be set when starting an application. More threads may take advantage of the available memory bandwidth to reduce the execution time when enough parallelism is available. However, they may also lead to increased synchronization overhead, or cache capacity exhaustion and conflict misses.

Other thread-related decisions impact the number of cores made available to the application and the mapping of its threads. Processors with **Simultaneous Multithreading** (SMT) capabilities let users map multiple threads (two on Intel HT) to the same physical core. One can also choose the **number of NUMA nodes** (NUMA) onto which to map the threads. For instance, selecting a subset of the NUMA nodes can let parts or whole processors go idle, reducing the power consumption. Given certain numbers of threads and cores, **thread mapping** (TMAP) policies decide which threads are assigned to which cores. We consider two policies: *scatter*, which spreads threads in round-robin to balance the resources' load; and *contiguous*, which fills NUMA nodes one after the other to favor locality. We refer to the subspace of these three dimensions combined as TH.

Data-related decisions include **data mapping** (DMAP) policies that influence the distribution of pages among NUMA nodes. We consider three policies: *first-touch* maps a page to the NUMA node of the first thread to access it; *locality* maps it to the node that accesses it the most; and *balance* spreads pages across nodes to balance the number of accesses per node. Together, data and thread mapping can be used to improve data locality, reducing the time and energy costs of moving data [44], and to spread memory accesses to avoid contention or to benefit from the aggregated bandwidth of multiple NUMA nodes (each with its own memory controller).

2) Administrator-level: Prefetching: Processors feature cache prefetching mechanisms that perform expensive data transfers ahead of time, usually having a positive impact on the performance of common workloads. However, prefetchers can fail to capture the specifics of more dynamic memory access patterns, causing interference, spurious cache evictions and general performance degradation. Thus, prefetchers can be disabled using privileged, administrator-level means. In the case of the Intel processor used in our study, each core supports four independently controlled **prefetchers** (PREF): the *DCU IP-correlated* prefetcher brings data from L2 to L1 based on a stride computed using the instruction pointer; the *DCU* prefetcher brings the next line to the L1 cache; the *L2 Adjacent Cache Line* prefetcher brings the previous or next cache line that completes a block aligned to 128 Bytes; and the *L2 Streamer* prefetcher tries to identify data streams and to bring the next predicted line to the L2 cache.

B. Problem Characteristics

1) Size-related Behavioral Variations: Applications can exhibit variations in their behaviors given different problem sizes. One may expect that larger sizes lead to larger memory footprints and longer execution times, while offering greater opportunities for parallel computation. It is known that this is not always the case, as sometimes the qualitative aspects of an application's data have a deeper effect than their quantitative aspects [18]. Nonetheless, it would be convenient to discover good configurations for small, cheap to process sizes that remain relevant for larger, more time-consuming problems. In order to verify if this is actually possible, we consider two problem **sizes** per application: *small* and *large*. Experimenting with more size variations could provide more insights, but it would also require many more experiments due to the multiplicative factor of the search space's dimensions. Additionally, finding representative problem sizes is not a trivial task: excessively small sizes might be unrepresentative of real world scenarios; excessively large sizes may require too much time for experiments; and sizes too close to each other may fail to cover the space of relevant application behaviors. We explain how we set our problem sizes and verified the resulting memory footprints in Section III-A3.

2) Optimization Target: We focus on two **target** goals: *performance* and *energy*. The minimization of applications' execution time has been the main objective on HPC systems for natural reasons. More recently, energy has become an additional concern due to its impact on the cost of ownership of systems [4] and for environmental reasons. Unfortunately, the energy consumed by these systems is not proportional to their workloads [4]. This is the case even for recent multicore processors, where optimizing thread mappings for performance does not result in an optimal energy consumption [24]. As a consequence, optimizing performance does not necessarily translate into energy consumption reductions. In this work, we optimize either performance or energy.

3) TS: Target-Size Optimization: We are interested in quantifying the effects that configuration options and size variations exert on our target goals. As problem size and target goal are not part of our search space, we refer to them together as Target-Size (**TS**). We use the notation <target, size> to represent a specific couple.

TABLE I
SEARCH SPACE DESCRIPTION. OPTIONS IN **BOLD** REPRESENT THE DEFAULT CONFIGURATION.

Dimensions	Options	No. opts	Abbreviation	
No. threads	64, **32**, 16	3	PAR	
SMT	Yes, **No**	2	SMT	TH
No. NUMA nodes	**2**, 1	2	NUMA	
Thread mapping	**Scatter**, Contiguous	2	TMAP	
Data mapping	**First-touch**, Locality, Balance	3	DMAP	
Prefetcher	DCU IP-correlated **On**, Off DCU **On**, Off L2 Adj. Cache Line **On**, Off L2 Streamer **On**, Off	16	PREF	

III. EXPLORATION METHODOLOGY

In this section, we present how we conducted the profiling of the previously described configuration space. The main challenge is its size and complexity: we addressed it with sampled

execution. We also assess the quality of our measurements as they condition both our analysis and the prediction capabilities of our Machine Learning models. Section IV presents the actual results analysis.

A. Experimental Setup

1) Hardware and Software: All experiments were performed on an Intel Xeon Gold 6130, with two NUMA nodes and 16 physical cores (32 virtual cores with Intel HT) per node, and 192 GB of DDR4 memory at 2666 MHz. To increase results stability, we used the base CPU frequency 2.1 GHz. We compiled all the applications with clang-6.0, -O3, and executed them on Linux 5.10.40-1.

All applications used in the experiments are codes written in C/C++ and annotated with OpenMP directives. They include established HPC benchmarks from the Rodinia Benchmark Suite [11] (v.3.1), the NAS Parallel Benchmarks (NPB v.3.0 in C) [2], [32], the PARSEC benchmarks [6], along with LULESH [20] (v.2) and CLOMP [8]. We have profiled 58 different parallel regions, each taking more than 5% of the total execution time, for their individual analysis: for faster evaluation, we do not factor inter-region configuration conflicts [33]. When comparing different configurations, values are normalized to the baseline configuration (options in bold in Table I). The baseline is a standard configuration that tries to increase bandwidth (scattered thread mapping) while preserving the developer's insights about the application (first-touch). We refer to performance improvements and energy savings as *gains* and calculate them as baseline divided by value.

2) Enforcing the Configuration Options: The options listed in Table I are implemented using different mechanisms in the system: we enabled or disabled the four prefetchers by updating the Model Specific Register (MSR) `0x1A4`. We set parallelism and threads to cores using `OMP_NUM_THREADS` and `KMP_AFFINITY` environment variables. Nonetheless, as this is done through the runtime environment, the actual thread mapping is only applied when the runtime is loaded at the first parallel region of the application. This could cause performance variation between runs because the OS could initially map threads differently between runs, which would also lead to different *first-touch* data mappings [33]. To address this issue, we used a compiler pass to insert empty parallel regions at the beginning of each application to force the desired thread mapping. Pages are mapped to nodes using the OS function `move_pages`. In order to identify pages for the *locality* and *balance* page mapping policies, we profiled each parallel region for each number of threads using Numalize (a Pin-based memory access pattern tracking tool [17]). This profiling also returns the memory footprint for each case. We avoided recalculating offsets across executions by disabling the Address Space Layout Randomization. Finally, the OS migrates pages to maximize locality. We disabled it because we already implemented a data policy that maximizes locality with Numalize and we do not want the automatic migration to disturb our decisions during the search space exploration.

3) Setting Varied Problem Sizes: Problem sizes are selected differently depending on the benchmark suite. The NAS Parallel Benchmarks are compiled with a selection of problem sizes (i.e., classes) that affect the size of data structures as well as the number of iterations to compute. The other benchmarks rely either on generated inputs or arguments set at launch time. We present the commands to replicate our experiments in Table II.

We empirically explored different options to define *small* and *large* problem sizes for the different applications. We considered classes A and B for NPB, and the largest possible input sizes for Rodinia and LULESH along with smaller cases. Because problem sizes are not consistent across benchmark suites, we profiled the memory footprint of each parallel region and reported it in Figure 2 when executed with 16 threads. Note that changing the parallelism affects the memory footprint: the footprint increases on average across all the applications by 35% and 13% for small and large sizes, respectively, when executed with 64 threads. In other words, increasing parallelism increases the memory footprint of an application, especially if the application has a small footprint.

We define the problem sizes from an application point of view: small and large are defined relative to each other for a given application. We do so because we assume that problem sizes are intrinsic to the applications. Thus, executing a region with small size can result in a bigger footprint or execution time than another region executed with large size (e.g., *bt xsolve* and *kmeans* for footprint). While the generated footprints are different, this approach enables us to preserve the benchmark diversity across sizes.

B. Execution Sampling and Stability

The full search space that we evaluate is composed of 1152 configurations. We reduce it by pruning configurations that produce the same execution (e.g., scatter or contiguous thread mapping without SMT executed on one NUMA node results in exactly the same thread mapping) to 768. Exhaustively exploring it enables us to find the best configuration (Section V presents how ML predicts the best configurations without such exploration for new applications) but also to characterize the applications through their behavior changes across configurations. Nevertheless, evaluating all the benchmarks across the entire space takes a huge amount of resources: running a single parallel region of Streamcluster takes approximately 4 minutes, resulting in 50 hours of runtime (we have 58 regions).

We cannot afford this exploration but we also do not want to further prune the configuration space as we may miss optimizations opportunities (e.g., counter intuitive data placements increasing remote accesses can improve performance [33]). An idea would be to fully explore the space on one size, and then sample the relevant configurations and only evaluate them on the other size. Unfortunately, we have no guarantee that the configurations will remain relevant across sizes as illustrated in Figure 1. Therefore, instead of sampling the space, we sampled the applications execution. Parallel applications using the fork-join models are known to have regular behaviors where the same parallel region is called hundreds of times

TABLE II
EXECUTING/COMPILING CODES ACROSS THE 2 SIZES. *THREADS* REFERS TO THE NUMBER OF THREADS AT EXECUTION.

Application	Compilation flag or execution option for the small input size	Compilation flag or execution option for the large input size
BT, CG, EP, FT, IS, LU, MG, SP	make CLASS=A	make CLASS=B
Blackscholes	./blackscholes THREADS in_4.txt out	./blackscholes THREADS in_10M.txt out
clomp_v1.2	./clomp THREADS -1 16 400 32 1 100	./clomp THREADS -1 16 6400 32 1 100
lulesh2.0.3	./lulesh2.0 -s 40	./lulesh2.0 -s 48
srad	./srad 100 0.5 383 300 THREADS	./srad 100 0.5 502 458 THREADS
myocyte	./myocyte.out 100 10 1 THREADS	./myocyte.out 100 50 1 THREADS
nn	./nn list20k_2.txt 5 30 90	./nn filelist_4 5 30 90
cfd	./euler3d_cpu ../../data/cfd/fvcorr.domn.193K	./euler3d_cpu ../../data/cfd/missile.domn.0.2M
kmeans	./kmeans -n THREADS -i ../../../data/kmeans/204800.txt	./kmeans -n THREADS -i ../../../data/kmeans/kdd_cup
hotspot	./hotspot 512 512 2 THREADS ../../data/hotspot/temp_512 ../../data/hotspot/power_512 output.out	./hotspot 1024 1024 2 THREADS ../../data/hotspot/temp_1024 ../../data/hotspot/power_1024 output.out
bfs	./bfs THREADS ../../data/bfs/graph65536.txt	./bfs THREADS ../../data/bfs/graph1MW_6.txt
particlefilter	./particle_filter -x 128 -y 128 -z 10 -np 5000	./particle_filter -x 128 -y 128 -z 10 -np 10000
pathfinder	./pathfinder 50000 100	./pathfinder 100000 100
lud	./lud_omp -n THREADS -s 6000	./lud_omp -n THREADS -s 8000
hotspot3D	./3D 512 4 100 ../../data/hotspot3D/power_512x4 ../../data/hotspot3D/temp_512x4 output.out	./3D 512 8 100 ../../data/hotspot3D/power_512x8 ../../data/hotspot3D/temp_512x8 output.out
lavaMD	./lavaMD -cores THREADS -boxes1d 8	./lavaMD -cores THREADS -boxes1d 10
streamcluster	./sc_omp 10 20 128 1000000 100000 5000 none output.txt THREADS	./sc_omp 10 20 128 1000000 200000 5000 none output.txt THREADS
nw	./needle 1024 10 THREADS	./needle 2048 10 THREADS

with the same behavior [42]. Executing a few calls of the region can be used to extrapolate its behavior [33], [39]. Not all regions have regular behaviors across calls [10], but this is a reasonable compromise to fully explore the space. To sample the execution of a region, we profile each region call and interrupt the application run as soon as we register 10 calls. We return the resulting median performance: we use the median instead of a sum across calls to not overestimate cache warmup effects during the first call. In the native execution, the cold cache effects are negligible when a region is called hundreds of times. On the contrary, they would have a disproportionate impact if we were to sum the execution times over 10 calls. For energy, we used Likwid [43] to report the aggregated consumption of the whole system across the 10 calls. We note that despite the execution sampling, it still took us over 3 weeks of continuous execution to evaluate the search space.

To ensure stability, we run 3 meta-repetitions for each execution. We selected the number of meta repetitions as a trade-off between a sufficient level of confidence and an acceptable execution cost. The performance variation across meta-repetitions was 3.1% and 1.8% when we sum or select the median across the 10 calls respectively (thus illustrating how cold cache effects create performance instability). By taking the median as metric in this study, we ensure that our performance results are stable. The average energy variation across the 3 meta-repetitions is 13%, indicating that energy measurements are much less stable than performance. We are unsure why performance is more stable than energy: we speculate they have different refresh-rates.

Figure 3 groups regions that share similar performance or energy variations across the meta repetitions into buckets. For instance, there are 54 regions which have, on average across all the configurations, a performance variation under 5% against only 27 for energy using the small size. We pruned from our study regions with variation exceeding 25%. This is a high stability threshold which results in optimizing regions with significant energy variation across executions. However, optimizing these regions result in huge energy gains (i.e., more than 700% on average), making the stability threshold minor in comparison.

To fairly compare the optimization impact across sizes in Section IV, we only keep *regions that are stable across both sizes and targets*. To ensure that we do not measure changes because we optimize different applications, we kept stable regions and preserved the same cross-validation folds in Section V.

IV. SEARCH SPACE CHARACTERIZATION

This section presents the insights we have gathered through the exploration of the search space for the different parallel

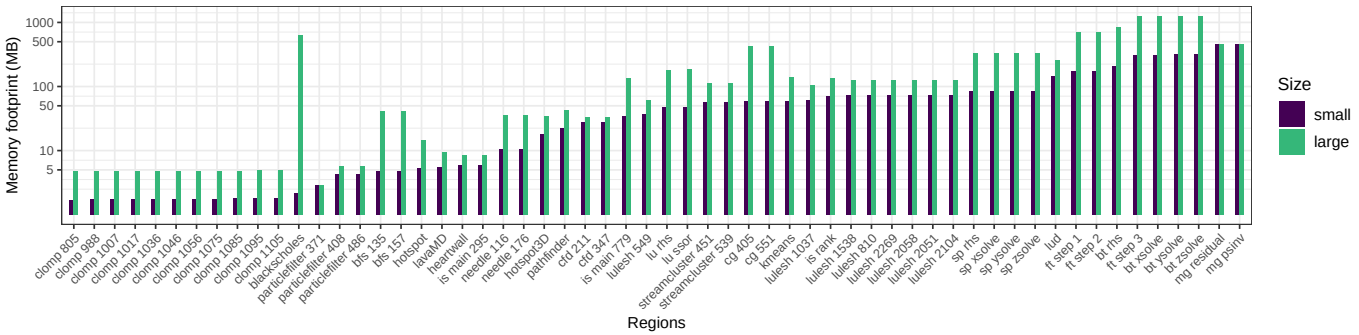

Fig. 2. Memory footprint across regions executed with 16 threads. We consider different memory footprints to quantify their impact on NUMA/prefetch.

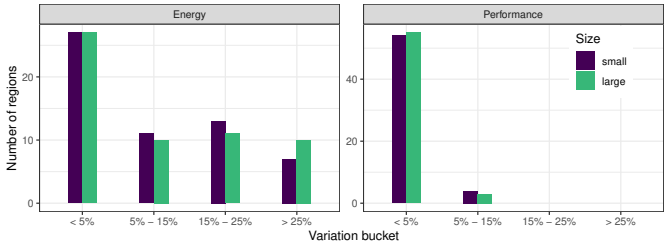

Fig. 3. Histogram presenting the variation of performance and energy across 3 meta repetitions. We set each region into a bucket depending on its variation. We then count the number of regions per bucket.

applications. We start by studying how the different dimensions of the search space affect our optimization goals (time and energy) across different inputs. We then quantify how changing the target goal or using a different problem size affects the overall gains. These changes justify the need for Machine Learning models to predict the best configurations which are later presented in Section V.

A. Quantifying Potential Gains Across the Search Space, per <target, size>

Quantifying the gains over the baseline provided by the different dimensions of the search space helps to answer different questions regarding the relative importance of the configuration options with respect to each other, such as: *Does a single option provide all the gains? Should some options be explored with a higher priority? Can we safely ignore some options to reduce the search space?*

Figure 4 presents the gains achieved when exhaustively exploring the effect of different configuration options. Each bar represents the gains of exploring a subspace: we average the gains across all the regions and select the best configuration per region within the considered subspace. On the top of the figure, we observe that an average speedup of $1.85\times$ can be achieved for performance, while on the bottom we see that the average energy consumption can be improved by a factor of $5.90\times$. This difference between target optimization achievements highlights that the baseline configuration is much more effective to optimize performance than energy. Indeed, 68% of the configurations improve energy on average across all regions against only 30% for performance: most of the configurations are more energy efficient than the default. Regarding performance in the context of problem sizes, we see that the optimizations always have a bigger impact when large inputs are used. Conversely, energy consumption improvements are mostly similar across problem sizes when TH and PAR are not considered together. Only when both are optimized do we see better energy savings for large inputs. Interestingly, if we change the baseline to the optimized configuration *contiguous, 32 threads using 2 nodes, with data locality, and all prefetch on*, we still observe that the energy gains are $4.1\times$ and $8.9\times$ over small/large, while the performance gains are $1.6\times$ for both sizes. The takeaway is that efficiency across TS cannot be achieved with a single configuration, independently on how optimized it is.

The results in Figure 4 also indicate that *no single option provides all the gains nor can be completely ignored*. We further consider how interesting each dimension of the space is by comparing its size (i.e., the number of options, efforts to explore it) against the gains it provides. Some spaces are larger compared to others (e.g., PREF 16 vs SMT 3). Moreover, some dimensions (e.g., PREF) even require root access which limits their optimization. Therefore, we note that *some parameters such as* TH*,* SMT*, or* PAR *are more attractive to be explored in priority on a budget than others* such as PREF or dimension combinations. Nevertheless, **we need to explore all the dimensions to achieve the highest gains**. Finally, the figure legend also compares our exploration with related work: **we are the first to explore such a large space for energy optimization across sizes**.

B. Reusing Search Space Insights from One <target, size> to Another

The pertinence of applying the insights and findings from one TS to another TS relates to the transferability of the overall gains. We say that gains are *transferable* from problem A to B (with different problem sizes or targets) if, for each parallel region, using the best configuration found in A in the context of B (the *cross* configuration) leads to gains similar to the best configuration found in B (the *native* configuration). We call *cross-optimization* the process of evaluating whether — and to quantify to which extent— gains are transferable from one problem to another one. When cross-optimization fails to apply from a problem A into a problem B, both problems must be optimized individually.

We illustrate the cross-optimization outcome on average across all the regions for the four combinations of problem characteristics in Figure 5. Each TS on the X-axis is profiled with different optimizations. Each color represents the TS used for guiding these optimization (i.e. we cross-optimize using the configurations from the color TS). Each Y-axis value represents the gains of the cross configurations selected using color TS divided by the native gains (e.g., on <perf, large>, the native and cross configurations from <energy, large> provide $1.91\times$ and $1.32\times$ gains respectively, resulting in $\frac{1.32}{1.91}(69\%)$, efficiency between cross and native configurations).

Overall, we can take three lessons from these results:

- *Cross-optimizing problem sizes while considering performance*: We observe that gains are mostly transferable, leading to around 95% of the native gains.
- *Cross-optimizing from energy to time*: all results are kept between 69% and 75% of the native gains. This gap of 25% or more erases most of the previously achieved gains. For instance, the speedup of $1.91\times$ achieved by <perf, large> is reduced to $1.32\times$ (69%) and $1.44\times$ (75%) when cross-optimizing, meaning that these gains are not transferable.
- *Optimizing for energy*: changing input sizes or cross-optimizing from time to energy leads to non-transferable

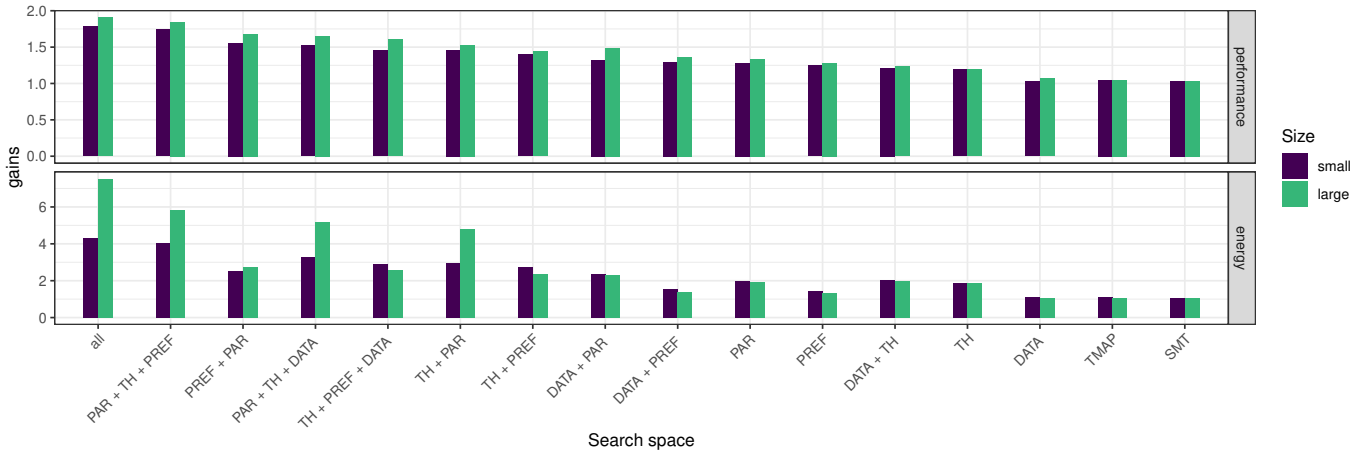

Fig. 4. [higher is better] X-axis: Subspaces. Y-axis: gains (performance or energy) calculated as ratios with the baseline. Note that subspaces containing PREF require root access to be evaluated. This table is also convenient to compare with different related works. The key difference is the size of the exploration and therefore the resulting gains. In particular, the following subspaces were previously explored: TMAP+DMAP [14], PAR+NUMA [47], TMAP [49], PREF [26], NUMA+PAR+DMAP [34]. The closest works are both Sanchez et al. [39] and Tehrani et al. [41]: we explore the same space except that our work further adds SMT. *More importantly, they only consider performance on one size, while we also consider energy across two sizes.*

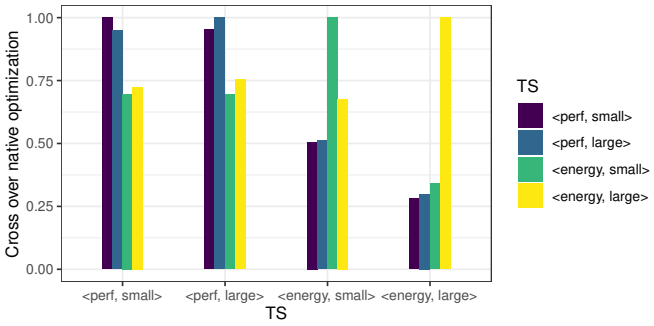

Fig. 5. [higher is better] X-axis: Profiled TSs. Y-axis: cross-optimizations gains compared to native gains. Colors represent the TSs configurations used to optimize each TS. Using a color that matches the profiled TS results in a native optimization. Otherwise, TSs are cross-optimized.

gains. In the most extreme case (for the native case with large problem sizes), gains are capped at 33%, meaning that the native optimization is at least three times better than any cross-optimization.

These results indicate that performance optimization insights do not apply to energy consumption optimization and vice-versa. Additionally, optimizing for energy consumption must be done with a problem size similar to the typical optimization target workload, due to the limited transferability of energy-related findings across multiple problem sizes. These facts motivate **the need of models for individual pairs of problem sizes and optimization goals**.

V. MACHINE LEARNING OPTIMIZATION

The results from the previous sections show that significant gains can be achieved, but only when all dimensions of the search space (Section IV-A) and a specific TS (Section IV-B) are considered. This requires thousands of runs for each new parallel region or TS of interest, which can be prohibitive. In order to overcome this limitation, we propose to use Machine Learning (ML) to train dedicated models per TS.

For a new parallel region and a given TS, we want models that avoid exploring the search space, predict efficient NUMA/prefetch configurations, and only use a few profiling runs. To do so, we train models by extracting, from each parallel region, a large set of characteristics (called *features*) along with the different NUMA/prefetch configurations performance or energy measurements (called *labels*). Labels are either performance or energy depending on the TS that we are predicting (i.e., we use energy measurements as labels if the target is energy). Features and labels are provided together to a supervised learning classification algorithm which creates a model that predicts labels for new, unseen parallel regions based only on their features.

Models can either collect labels and features using the same problem size or with different sizes. Using different sizes, *cross-predicting*, (small to large or vice versa) is more challenging as the information to predict the optimization might not be available on the profiling size. Yet, it is also much more attractive as we propose more generalized optimizations. **Predicting complex NUMA/prefetch configurations across sizes is one of the key novelties of this work**. For the rest of this section, we present the training and validation of Machine Learning models in Section V-A, the processing of labels and features in Sections V-B and V-C, and finally the results in Section V-D.

A. Training Machine Learning Models

Figure 6 presents the workflow of the ML models that we train and evaluate for a target TS. We start by profiling a set of training parallel regions. We collect features that we subset (details in Section V-C). In parallel, we also *evaluate* all the configurations (details in Section V-B) and return the most efficient ones per parallel region as labels. For simplicity

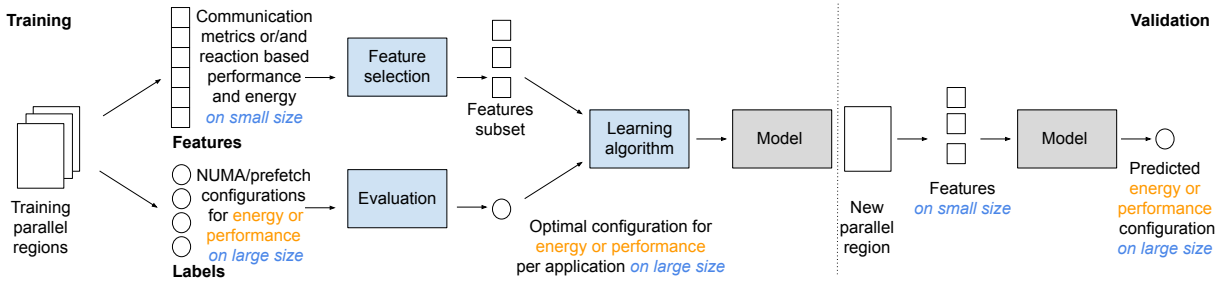

Fig. 6. ML workflow when predicting optimizations from small size to large size. Left: model training. Right: model validation. The model predicts NUMA/prefetch configurations for large size by only measuring a few features on small size. The same workflow is used for performance or energy models.

and clarity, the figure presents a TS set to size large using features collected with size small (in practice we consider all the possible size combinations, even a mix of both small and large for the features to assess how the models perform if they have access to all the information). We then group both labels and features and provide them to Machine Learning algorithms that generate prediction models. Different algorithms operate more or less efficiently depending on the available data. Based on the concept of *bucketing* [46] and following an approach similar to Sanchez et al. [39], we implemented different learning algorithms including Decision Tree (DT), Logistic Regressions, and Support Vector Machines with Scikit learn setup [30] (version 1.0). Similarly to Sanchez et al. [39], we observed on preliminary results that DT outperforms the other classifiers and have, therefore, selected it for the remaining steps of our work.

To evaluate and validate our models, we deploy them over new unseen parallel regions. We expect that, by just collecting features over these new regions, our models will predict the most efficient NUMA/prefetch configuration for each region. We evaluate our models using a standard 10-fold cross-validation [37]: for each TS, we train ten models and evaluate each one over a validation fold composed of 5 or 6 unseen parallel regions (i.e., 10%). To check that our models can operate over new unseen regions, all results presented in Section V-D are generated by aggregating all the regions from the ten validation folds.

B. Key Configurations for Classification (Labels)

Having a short list of configuration labels is particularly important for reducing the stress over the Machine Learning models [41]. Therefore, instead of exposing the entire search space as labels for evaluation to the learner, we only provide a small but impactful set of configurations that preserves the improvements found in the whole search space. We follow the same procedure previously employed by Sanchez et al. [39]. For a given TS, we first find the configuration that provides the biggest average gain. We then find the next configuration that further improves this gain the most, and we continue on this process until we have ten distinct configurations (we empirically observed that ten configurations was enough to provide high gains). This process is applied to each TS, resulting in four lists of ten configurations, one per TS. We observe that, for a given size, selecting the best configuration for each region among these ten configurations provides at least 99% of the performance and 97% of the energy gains of the exhaustive search across the whole space.

We further replicated the cross-evaluation of Section IV-B using only our configuration lists. In this scenario, the average difference between the cross-optimization using our lists and the whole search space is only 1%. These results indicate that our short configuration lists are effective proxies for the original search space.

To illustrate the variety in the configuration lists, Table III summarizes the number of times a given option was part of the best configurations list for each TS. Indifferently from dimension, all options besides two (i.e., contiguous thread mapping and DCU IP-correlated prefetcher on) are present among the best configurations at least 30% of the time (i.e., 12 over 40 configurations). This variety is further emphasized by the presence of 37 unique configurations. These results both emphasize the conclusions of Section IV (i.e., no dimension can be pruned without losses) and demonstrate the coverage of the search space by the key configurations lists. Therefore, we use, for a given TS, the associated list of configurations as labels for our models.

TABLE III
OPTIONS FOUND IN THE KEY CONFIGURATIONS FOR EACH TS. FOR INSTANCE, CONTIGUOUS IS INCLUDED 7 AND 9 TIMES IN THE 10 BEST CONFIGURATIONS WHEN OPTIMIZING <ENERGY, LARGE> AND <PERFORMANCE, LARGE>

Dimensions	Options	TSs			
		<perf, small>	<perf, large>	<energy, small>	<energy, large>
No. threads	64, 32, 16	5,3,2	4,2,4	1,5,4	3,3,4
SMT	Yes, No	4,6	3,7	6,4	4,6
No. NUMA nodes	2, 1	7,3	6,4	3,7	6,4
Thread mapping	Scatter, Contiguous	0,10	1,9	3,7	3,7
Data mapping	First-touch, Locality, Balance	2,4,4	3,4,3	5,3,2	4,3,3
Prefetcher	On for each of the 4 pref.	10,7,6,5	8,6,3,5	8,6,4,3	8,5,3,5

C. Application Characteristics (Features)

ML models use features to predict what configuration to use at deployment. As previously described, naively optimizing NUMA/prefetch requires exploring many different configurations. On the other hand, an application's features can be cheaply collected through light profiling in comparison. We investigate features that maximize information to make accurate predictions while minimizing collection overhead to reduce deployment costs. Traditional dynamic features (i.e.,

collected at execution) used to predict NUMA or prefetch include performance counters and communication metrics.

Communication metrics have been proposed in the context of thread and process mapping in NUMA systems [7], [15], [17]. They are based on communication matrices that represent how much data is shared between two threads (e.g., the number of accesses to the same page, the number of messages sent, the number of bytes exchanged). *We consider in our models six communication metrics profiled across the different degrees of parallelism resulting in a total of* 18 *features*: CA [17] represents the average communication cost of the matrix; CB [15] captures how balanced the matrix is; CC [7] calculates the dispersion of communication from the main diagonal of the matrix; CH [17] measures the average communication variance; NBC [7] indicates the fraction of communication between threads with close identifiers; and SP(16) [7] represents the fraction of communication happening among threads in 16×16 blocks in the matrix. We used Numalize to identify how threads access pages and then calculated the resulting communication metrics. Finally, we normalize each metric by the highest value recorded for that metric across all regions.

Performance counters are standard metrics that characterize an application's execution and are available across many systems. We can collect counters by either executing the application over a single configuration or across different ones. Collecting counters from a single configuration is more straightforward as we do not need to change the NUMA/prefetch configuration across executions. However, collecting counters across configurations has higher tuning potential in the context of compiler [49] or NUMA/prefetch performance optimization [39]. By profiling the same counter across different configurations, we measure its reaction to various contexts. This reaction is valuable information to guide optimizations, a so-called *reaction-based profiling* [39], [49]. *We also consider the* 1536 $(768 * 2)$ *performance and energy measurements as reaction-based features in our models*. Incorporating diverse counters may potentially characterize memory/compute-bound applications to improve our predictions. Nevertheless, we only consider counters tracking performance and energy because 1) reaction-based energy has already successfully optimized NUMA/prefetch performance [39], and 2) including more counters comes at an extremely high profiling cost as we need to execute the new counters across the whole space. To provide the reaction-based performance or energy features to the model, we take all the measurements and normalize them against the baseline configuration. The performance or energy changes between the baseline or the different configurations are the reaction-based information provided to the model.

In total, we consider 18 communication metrics and 1536 reaction-based performance and energy measurements for a total of 1554 features that can be collected on a given size. Using multiple features for a model increases its information and prediction potential. However, it creates noise as some information might not be relevant to the optimization. More importantly, it increases the profiling cost at deployment. To reduce deployment cost, we use *feature selection*: we train different models using only a subset of features at a time. Using feature subsets drastically complicates the training process: instead of training models which consider all the features at once, we must train different models, one per subset. Therefore, for each TS and subset of features, we evaluate ten models because of cross-validation. In addition, we consider features collected with small, large, or both sizes (we discuss mixing sizes in Section V-D). However, this enables us to deploy learning algorithms that predict the best NUMA/prefetch configuration using only a subset of features.

We empirically evaluated models with 1, 2, or 3 features. Moreover, because collecting communication metrics is more expensive than collecting counters, we consider models using only communication metrics, reaction-based counters, or combinations of both. While we can easily exhaustively evaluate all the combinations of 1 feature, training models with 2 or more features drastically increases the feature space: we need to train over a million models to explore all combinations of 2 features. Therefore, we reduced the feature space for each TS to $15,000$ by randomly evaluating combinations of features. We also used Genetic Algorithms (GA) to guide the feature selection and return the most efficient subset of features accordingly. This was implemented with *pyeasyga* [1] (version $0.3.1$) with population size 2500, 25 generations, 90% and 10% crossover and mutation probabilities respectively).

D. Machine Learning Results

Figure 7 presents the gains of the models described in Section V-A and compares them against the naive cross-optimizations from Section IV-B. Similarly to Figure 5, all values are normalized to the native optimizations. Cross-optimizations between TSs are presented using horizontal lines (e.g., the straight line <perf, small> indicates the gains achieved when selecting the best configuration per region for <perf, small>). The colors show the size used to collect the features (e.g., if the color is different from the predicted size, we are predicting configurations on a different size than the one used to collect the features). *small+large* presents models trained using features collected from both sizes. Finally, and as described in Section V-C, we consider different subsets of features. In particular, we evaluated models using 1 feature (exhaustively explored), 2 features ($15,000$ random subsets or using GA), and 3 features (only with GA).

ML models consistently outperform any cross-optimizations across all TSs. Cross-optimizing <energy, large> with <energy, small> only provides 34% of the peak gains. On the opposite, ML with 3 features achieves 73% of the gains. Even for <perf, small> where cross-optimization mostly preserves the gains (i.e., <perf, large> provides 95% of the peak gains), models with 3 features from small and large sizes provide 98% and 96% gains, respectively.

We also compute the absolute distance ($|native - cross|/max(native, cross)$) between the gains of native models and their cross-predicting counterparts: native models with 3 features provide 3.5% more gains than cross-predicting ones. This difference is very small when compared to some naive

cross-optimizations from different TSs (up to 60% losses when cross optimizing <energy, large>). This emphasizes how it is more difficult to optimize energy than performance. We also assess if collecting features from both sizes improves predictions over collecting them with a single size. Overall, both approaches return relatively similar gains, indicating that considering a single size is enough (i.e., the features collected on that size contain enough information to take optimization decisions).

We observe that reaction-based counters systematically outperform communication metrics when used as features. Not only do they provide more gains but even when we mixed communication metrics with reaction-based counters using 2 or 3 features, our feature selection process only returned reaction-based counters. Denoyelle et al. [14] compared performance counters (i.e. memory information) against communication metrics and concluded that both provide similar gains for NUMA predictions (while communication metrics are approximately 10× more costly to collect). **Our results indicate that reaction-based performance/energy provide more gains** than any of these two approaches and can be collected during execution without costly instruction instrumentation. However, they require exploring a large feature space [39] at training as illustrated by Table IV. This table shows the most efficient groups of 3 features for the most difficult to predict TS <energy, large> using either small or large sizes. We observe that we must profile performance or energy across quite different configurations.

TABLE IV
BEST REACTION-BASED COUNTERS COLLECTED EITHER WITH SMALL OR LARGE SIZES PREDICTING <ENERGY, LARGE>. WE NOTE THAT PERFORMANCE IS USED TO PREDICT ENERGY OPTIMIZATIONS.

Target	No. threads	SMT	No. NUMA nodes	Thread mapping	Data mapping	Prefetcher
Size small						
performance	32	no	2	scatter	first-touch	off/off/on/off
energy	32	no	2	contiguous	locality	on/on/off/on
energy	32	no	2	scatter	locality	on/off/on/on
Size large						
energy	16	no	1	contiguous	balance	off/off/on/off
energy	64	no	2	scatter	locality	on/on/on/off
performance	32	no	2	contiguous	first-touch	on/on/on/off

We further analyzed our results for <energy, large>: Fig-

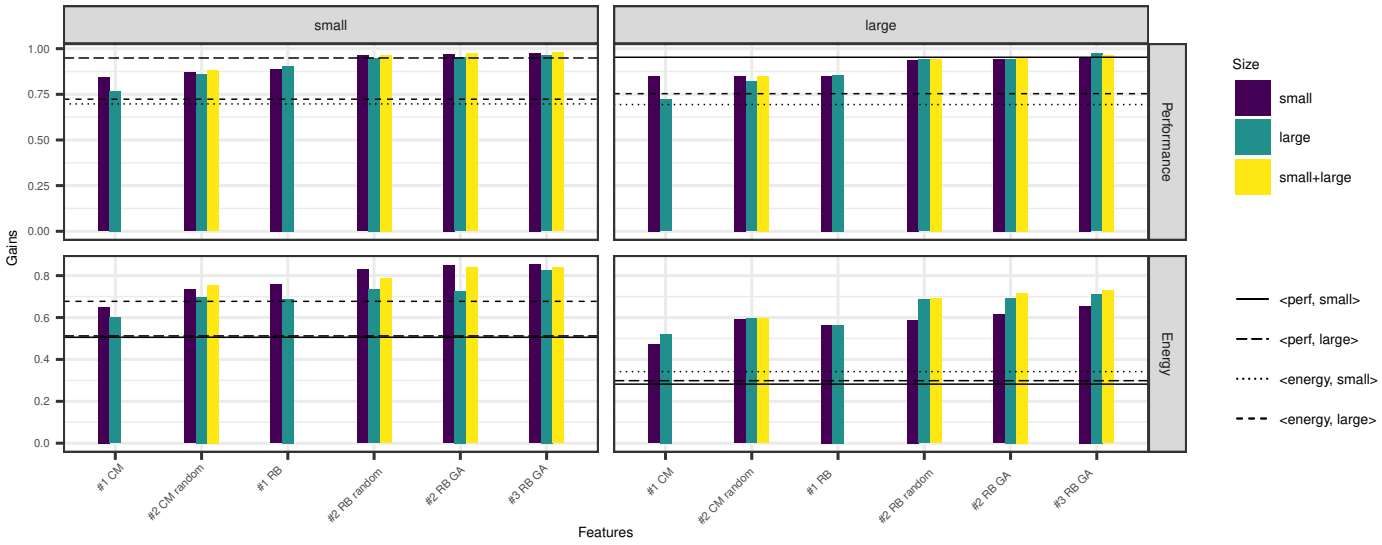

Fig. 7. [higher is better] X-axis: Different subsets of features used for training. CM and RB refer to Communication Metrics and Reaction Based profiling respectively while #[1-3] shows the number of used features. Y-axis: gains (performance or energy) normalized to native optimizations. Colors indicate the size used to collect the features. Dashed lines show the gains when we transfer configurations across TSs as described in Section IV.

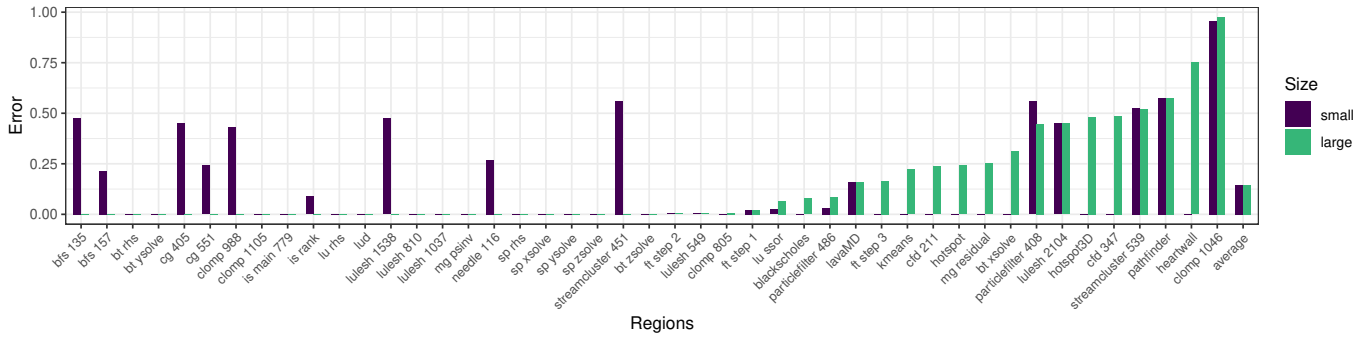

Fig. 8. [lower is better] X-Axis: regions. Y-axis: absolute difference calculated between the gains predicted by our model and the native exploration gains for <energy, large>. Colors indicate from which size features were collected.

ure 8 presents prediction errors calculated using the absolute distance ($|peak-predicted|/max(peak, predicted)$) for each region between its peak gains and the model prediction gains using small or large sizes with the 3 features presented in Table IV. The average distance across all the regions is 15%. 4 regions have over $30\times$ energy gains and our model mispredicts some of them. As a result, removing such outlier regions increases the gains of our models to over 85% of the peak gains: we believe that adding more regions with such gains would help our models to better recognize them. **To summarize, ML gives most of the gains across different TSs while only collecting 3 measurements on a single problem size.**

VI. Related Work

Search space exploration. Standard techniques for finding the best NUMA and prefetch-related configuration options often require exploring a large optimization space. Khan et al. [22] and Jimenez et al. [21] evaluate different prefetchers at runtime to select the most efficient ones for performance on Intel and IBM systems, respectively. Radojkovic et al. [35] explore different thread mappings to optimize performance. Popov et al. [33] couple thread and data mapping options to boost performance gains in NUMA systems, while Sanchez et al. [39] extends these options with prefetch settings. *Our work takes this state-of-the-art search space and incorporates an SMT dimension, while also considering different problem sizes and optimization targets.*

Native search limitation. While exhaustive explorations lead to good configurations for some applications, they require thousands of runs [33]. This overhead is present every time a new application is considered. Additionally, one may also find applications that benefit very little from optimizations beyond the standard configuration [17]. In order to avoid these overheads, models, either based on Machine Learning or designed by domain experts, can directly predict relevant configurations or prune the search space.

Features selection. Different models use distinct means to collect application characteristics (features) for their decisions, as collecting this information is seen as far less costly than the space exploration. There is a wide spectrum of feature extraction techniques, each with its own benefits and collection costs. These techniques include compiler analysis [31], [41], performance counters [9], [25], [26], [47], reaction-based profiling [23], [49], instrumentation [17] (e.g. communication metrics, pin memory profiling), or even a combination of techniques [14], [19], [39]. *Our work experiments for the first time how a combination of instrumentation and reaction-based profiling can feed a Machine Learning model.*

Configurations selection. Different approaches predict configurations across the diverse subspaces. Piccoli et al. [31] are able to optimize data placement using compiler loop analysis. Similarly, Tehrani et al. [41] demonstrate that compiler Intermediate Representation analysis can provide 70% of the gains from reaction-based profiling. Guo et al. [19] use both compiler information and performance counters to reduce energy overheads due to prefetching. Counters are also used to optimize prefetch [26] and NUMA effects [25] through thread and memory migration. Using performance counters, Wang et al. [47] further predict the number of NUMA nodes to use to optimize performance, while Castro et al. [9] select thread mappings for Software Transactional Memory applications. Denoyelle et al. [14] unify NUMA optimizations by considering both thread and data placements using instrumentation and performance counters together. Reaction-based profiling is employed by Sanchez et al. [39] in order to optimize performance for different applications based on NUMA and prefetch configurations. Other examples of reaction-based prediction include compiler optimization [23], [49].

Machine Learning models. We also note that more or less computation-intensive ML models are used for prediction, ranging from simple decision trees or support vector machine [39], [48], [49], to complex deep learning methods [41], [45]. Multiple models can also be considered as illustrated by Roy et al. [38]. They auto-tune OpenMP parameters and system frequencies using multiple light-weight Bayesian Optimization models. *In our work, we use decision trees with reaction-based performance and energy along with instrumentation and observe that reaction features outperform communication metrics collected from instrumentation.*

Energy and problem size. While performance tuning is still commonly studied considering a single problem size, works have emerged to explore multiple problem sizes or to consider energy and performance. Indeed, performance and energy are not proportional across different search spaces including thread scheduling, cache reconfiguration, and frequency [24] or in the context of heterogeneous systems [13], [40]. Reddy et al. [36] proposed a data partitioning algorithm for NUMA that optimizes both performance and energy. Berned et al. [5] further combine performance and energy using thread mapping across three problem sizes by natively exploring the space. Similarly, Marques et al. [27] combine concurrency throttling with turbo-boosting while Papadimitriou et al. [29] consider Voltage/Frequency scaling with core allocations.

To the best of our knowledge, in the context of NUMA (i.e., considering both thread and data) and prefetch-related settings optimization, this paper is the first 1) that investigates the impact of problem sizes on the effectiveness of configurations and thus on models; 2) that quantifies the gap due to configuration transposition between performance or energy; 3) that proposes a model to predict overall efficient configurations and succeeds in doing so using reaction-based features.

VII. Conclusion

In this study, we have explored the search space of NUMA and prefetch-related configurations to optimize both performance and energy across two different problem sizes per application. Our research has shown the impact of these configuration options in all cases with performance and energy gains of $1.85\times$ and $5.9\times$ over a baseline. Unfortunately, the configuration combinations achieving these gains are conditioned by the problem size and optimization target, i.e., we

cannot simply transpose optimization strategies discovered for performance and expect to obtain high energy gains as well. Similarly, we observed that the best optimizations for energy on one problem size do not apply to another problem size. To address these limitations, we proposed a Machine Learning framework for modeling and predicting both performance and energy optimizations by quickly profiling a single problem size. This approach has achieved over 85% of the peak gains from the search space exploration.

As future work, we envision larger search spaces that consider heterogeneous systems, frequency scaling, as well as the selection of compiler optimization settings. Indeed, preliminary results show that the same application compiled with different compiler options experiences different NUMA effects. Finally, to further reduce the overhead of our model at deployment, we contemplate the evaluation of static features for modeling.

REFERENCES

[1] "Pyeasyga," https://pyeasyga.readthedocs.io/en/latest/readme.html, accessed: 2022-07-29.

[2] 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, "The NAS Parallel Benchmarks — Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165.

[3] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[4] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[5] G. P. Berned, T. S. Medeiros, M. Serpa, F. D. Rossi, M. C. Luizelli, P. O. Navaux, A. C. S. Beck, and A. F. Lorenzon, "Combining thread throttling and mapping to optimize the edp of parallel applications," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2021, pp. 177–180.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[7] C. Bordage and E. Jeannot, "Process affinity, metrics and impact on performance: An empirical study," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '18. IEEE Press, 2018, pp. 523–532.

[8] G. Bronevetsky, J. Gyllenhaal, and B. R. De Supinski, "Clomp: accurately characterizing openmp application overheads," in *International Workshop on OpenMP*. Springer, 2008, pp. 13–25.

[9] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *2011 18th International Conference on High Performance Computing*. IEEE, 2011, pp. 1–10.

[10] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, p. 6, 2015.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, Oct. 2009, pp. 44–54.

[12] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 448–459.

[13] M. D'Amico and J. C. Gonzalez, "Energy hardware and workload aware job scheduling towards interconnected hpc environments," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[14] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in numa architectures: A statistical learning approach," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: ACM, 2019, pp. 39:1–39:10.

[15] M. Diener, E. H. Cruz, M. A. Alves, M. S. Alhakeem, P. O. Navaux, and H.-U. Heiß, "Locality and balance for communication-aware thread mapping in multicore systems," in *European Conference on Parallel Processing*. Springer, 2015, pp. 196–208.

[16] M. Diener, E. H. Cruz, M. A. Alves, P. O. Navaux, and I. Koren, "Affinity-based thread and data mapping in shared memory systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 64, 2017.

[17] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88, pp. 18–36, 2015.

[18] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez, "Profiles of upcoming hpc applications and their impact on reservation strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, 11 2020.

[19] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, "Energy-efficient hardware data prefetching," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 2, pp. 250–263, 2009.

[20] R. Hornung, J. Keasler, and M. Gokhale, "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[21] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making data prefetch smarter: Adaptive prefetching on power7," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2012, pp. 137–146.

[22] M. Khan, M. A. Laurenzanoy, J. Marsy, E. Hagersten, and D. Black-Schaffer, "Arep: Adaptive resource efficient prefetching for maximizing multicore performance," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 367–378.

[23] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using predictivemodeling for cross-program design space exploration in multicore systems," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE, 2007, pp. 327–338.

[24] S. Khokhriakov, R. R. Manumachu, and A. Lastovetsky, "Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance," *Applied Energy*, vol. 268, p. 114957, 2020.

[25] R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "Cimar, nimar, and lmma: Novel algorithms for thread and memory migrations in user space on numa systems using hardware counters," *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022.

[26] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 56.

[27] S. M. V. Marques, M. S. Serpa, A. N. Muñoz, F. D. Rossi, M. C. Luizelli, P. O. Navaux, A. C. S. Beck, and A. F. Lorenzon, "Optimizing the edp of openmp applications via concurrency throttling and frequency boosting," *Journal of Systems Architecture*, p. 102379, 2022.

[28] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 35, 2016.

[29] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus," in *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 133–146.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. v, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *The Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[31] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in numa architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 369–380.

[32] M. Popov, C. Akel, F. Conti, W. Jalby, and P. de Oliveira Castro, "Pcere: Fine-grained parallel benchmark decomposition for scalability predic-

tion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1151–1160.

[33] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/parallelism autotuning for NUMA systems," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 342–353.

[34] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/parallelism autotuning for numa systems," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 342–353. [Online]. Available: http://doi.acm.org/10.1145/3330345.3330376

[35] P. Radojković, P. M. Carpenter, M. Moretó, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread assignment in multicore/multithreaded processors: a statistical approach," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 256–269, 2016.

[36] R. Reddy Manumachu and A. L. Lastovetsky, "Design of self-adaptable data parallel applications on multicore clusters automatically optimized for performance and energy through load distribution," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 4, p. e4958, 2019.

[37] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation." *Encyclopedia of database systems*, vol. 5, pp. 532–538, 2009.

[38] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1280–1295.

[39] I. Sánchez Barrera, D. Black-Schaffer, M. Casas, M. Moretó, A. Stupnikova, and M. Popov, "Modeling and optimizing numa effects and prefetching with machine learning," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–13.

[40] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. Chong, "Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques," *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 6, pp. 1633–1646, 2015.

[41] A. TehraniJamsaz, M. Popov, A. Dutta, E. Saillard, and A. Jannesari, "Learning intermediate representations using graph neural networks for numa and prefetchers optimization," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 1206–1216. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00120

[42] F. Trahay, M. Selva, L. Morel, and K. Marquet, "Numamma: Numa memory analyzer," in *International Conference on Parallel Processing*, 2018.

[43] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[44] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.

[45] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.

[46] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, mar 2020.

[47] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 419–431.

[48] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *ACM Sigplan notices*, vol. 44, no. 4. ACM, 2009, pp. 75–84.

[49] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.