Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach

Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, Thomas Ropars

► To cite this version:

Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, Thomas Ropars. Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach. ICPP 2019 - 48th International Conference on Parallel Processing, Aug 2019, Kyoto, Japan. hal-02135545v1

HAL Id: hal-02135545
https://inria.hal.science/hal-02135545v1

Submitted on 21 May 2019 (v1), last revised 30 Jul 2019 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach

Nicolas Denoyelle
ndenoyelle@anl.gov
Argonne National Laboratory
Inria, LaBRI, Univ. Bordeaux – France
Atos Bull Technologies

Brice Goglin
brice.goglin@inria.fr
Inria, LaBRI, Univ. Bordeaux – France

Emmanuel Jeannot
emmanuel.jeannot@inria.fr
Inria, LaBRI, Univ. Bordeaux – France

Thomas Ropars
thomas.ropars@univ-grenoble-alpes.fr
Univ. Grenoble Alpes – France

ABSTRACT

Nowadays, NUMA architectures are common in compute-intensive systems. Achieving high performance for multi-threaded application requires both a careful placement of threads on computing units and a thorough allocation of data in memory. Finding such a placement is a hard problem to solve, because performance depends on complex interactions in several layers of the memory hierarchy. In this paper we propose a black-box approach to decide if an application execution time can be impacted by the placement of its threads and data, and in such a case, to choose the best placement strategy to adopt. We show that it is possible to reach near-optimal placement policy selection. Furthermore, solutions work across several recent processor architectures and decisions can be taken with a single run of low overhead profiling.

KEYWORDS

high-performance-computing, numa, machine-learning, threads, placement, data, multicore-processors

1 INTRODUCTION

With the advent of NUMA (Non Uniform Memory Access) architectures, optimizing applications execution time is known to be a extremely difficult endeavor. Not only it requires to carefully write the application such that the threads efficiently use the available resources (e.g., cores, caches and memory) but, even if the code is highly optimized, launch time optimizations and decisions may have a great impact on performance.

In this paper, we focus on this second aspect of the problem. We assume that a multi-threaded application is already statically optimized and we look at the factors impacting its execution time that can be set at launch-time. The two main factors targeted here are the thread placement policy (which thread is mapped on which core) and the memory allocation policy (where memory pages are allocated). Indeed, data locality and memory contention are affected by these policies and can have a huge impact in several levels of the memory hierarchy and thus, on application performance.

The goal of this paper is to provide a methodology and models to answer two questions: 1) Is an application sensitive to the placement of its threads and data on a given NUMA architecture? 2) What is the best placement and allocation policy for this application on that architecture?

To achieve this goal, we follow several lines of study. First, to capture applications characteristics, we use two methods that allow capturing different kinds of metrics and have a different cost: instrumenting the application binary (costly and characterizing applications regardless of the underlying hardware) or relying on hardware counters (cheap and embedding hardware specific responses). Second, we study several meta-models used in machine learning to model the impact of placement on applications. Third, we target many different applications and benchmarks that embrace a large spectrum of use-cases. Fourth, we use testbeds spanning different Intel processor generations. We study predictions on a given architecture and predictions across multiple architectures.

The two main results of this article are the following:

- On multiple Intel platforms, it is possible to build a model that can decide whether a new application is sensitive to locality and we reach an accuracy close to 80%.
- For applications that are sensitive to locality, it is further possible to build models taking good placement decisions (with respect to the studied placement strategies) and to achieve speedups that are close to the optimal.

Moreover, the following additional results have been obtained while conducting this study:

- It is possible to build a model on a processor family, and to obtain good predictions on another processor family.
- Although fast to collect metrics and unable to precisely capture the algorithmic characteristics of applications, hardware counters provide sufficient information to take decisions about placement.

The remainder of the paper is organized as follows. In Section 2, we describe the context and the state of the art. The methodology is presented in Section 3. The experimental testbed is described in Section 4. In Section 5, we present and study the results. Finally, concluding remarks are given in Section 6.

ICPP '19, August 5-8, 2019, Kyoto Research Park (KRP), Japan
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 CONTEXT

2.1 Objectives and assumptions

This study considers the placement of threads and data of a multithreaded application running in a modern NUMA platform. As illustrated in Figure 1, such platforms are composed of several multicore processors. Each core has access to some private caches while a last-level cache (LLC) is shared between all cores on the same processor. The processors are connected through a high performance interconnect (e.g., Intel Quick Path Interconnect). The memory is shared between all processors and cache coherence is ensured by the hardware. Part of the memory is associated with each processor. A processor together with its local part of the memory is called a NUMA node. Local accesses to data, *i.e.* inside a node, is fast whereas accesses to a remote node, whether the data is in a remote cache or in a remote memory, is slower.

Several phenomena can impact the performance of multi-threaded applications in such platforms. When deciding on the placement of threads and data, two main points need to be considered: locality and contention. Locality refers to the fact that for a thread to perform well, the data it accesses should be as close as possible. The best case is when the data is in some cache that the thread can directly access. The worst case is usually when the data is in another NUMA node. Contention refers to the fact that if too many threads use some shared resources at the same time, performance degradation is to be expected. The LLC, the interconnect links, and the memory controllers are examples of resources that can suffer from contention. Another difficulty comes from the fact that threads of a multithreaded application collaborate to run a computation, and so, use shared memory to synchronize and communicate. The challenge is then to optimize locality by taking into account accesses of threads to private and shared variables while avoiding contention on shared hardware resources.

In this study we consider the case where a single multithreaded application runs on a NUMA platform. The application uses all available cores. Our study is limited to cases where Simultaneous Multi-Threading (SMT) is disabled to be able to focus on the locality and contention issues that occur at the level of the LLC and below (enabling SMT would raise additional questions, *e.g.* related to the contention on the computing resources inside one core).

In this context, deciding on the appropriate threads placement and data allocation policy is a complicated problem that we aim to solve. Since placement decision algorithms can be expensive [14], we are also concerned by applications need for such an optimization. We seek for solutions to answer these two questions for unseen applications and/or unseen platforms. Such properties are important as they imply that obtained solutions could work for users with different machines and undisclosed applications and would be resilient to modifications in applications. Therefore, our overall goal is to build models that are able to decide, after a single run of an application for a given set of inputs[1] on a single machine (the minimum effort required to collect metrics), for a target (eventually different) platform: i) whether the application's performance is sensitive to placement and ii) if yes, what is the best placement policy for this application.

[1]In the following, the run of an application is always studied for a given set of inputs.

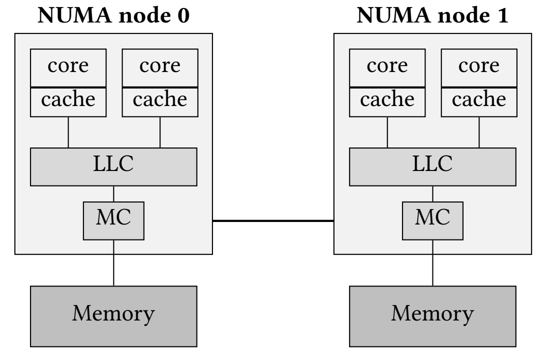

Figure 1: Abstract representation of an architecture with 2 NUMA nodes. LLC stands for Last-Level Cache. MC stands for Memory Controller

Our study focuses on compute-intensive applications [1, 2, 12]. We make a few assumptions about the characteristics of applications to which our proposed technique is applied, but these assumptions are common to many compute-intensive applications. First, our study focuses on applications that follow the fork-join execution model, *i.e.*, threads are not dynamically created as a consequence of some events. Hence the same number of threads is created in all the runs of an application. Furthermore we assume that threads can be uniquely identified using IDs assigned based on their creation order, as it is for instance the case in OpenMP applications. The other major assumption is that for a given set of inputs, the thread with the same ID in two runs of the application execute similar computations. This implies that our approach cannot work with *e.g.* task programming paradigms out of the box.

We only study static policies: the placement of threads and the policy for memory allocation are decided before launching the application and remain for the whole application lifespan. To make our problem tractable, we limit our study to two placement policies for threads and to two for data. For threads, the choice is between: i) having the threads with consecutive IDs placed on the same processor (called *Compact* hereafter) which favor neighbors sharing, and ii) having the threads assigned to processors in round-robin (*Scatter*) mitigating neighbors contention. These two policies are for instance supported by most OpenMP runtime libraries. For data, the choice is between the two standard allocation policies: i) memory is allocated on the node of the first thread that *touches* the corresponding page (*First-touch*) targeting data locality; (ii) memory is allocated in round robin on nodes (*Interleave*), mitigating contention in memory controllers.

The approach we follow is to collect a set of metrics during the run of applications. These metrics are collected using transparent techniques for the applications, either based on hardware counters or on information obtained through binary code instrumentation. These metrics, together with the preferred placement strategy in a given platform, are then used to train classification algorithms to try to answer the questions raised above.

2.2 Related work

Placement of threads and data in NUMA platforms has extensively been studied in the literature. Locality and resource contention issues are identified as the two main problems to tackle [21, 25, 34].

Our work follows an *offline profiling* approach to obtain information about interactions between threads and about memory accesses. Some previous works have also used this approach. Diener et al [20] propose a solution based on binary code instrumentation to obtain information about the interactions between threads. Other works try to collect information about memory accesses using the monitoring capabilities of modern processors [26, 27]. In this paper, we consider metrics coming both from information obtained through binary instrumentation and hardware counters.

Several works also tried to do *online* profiling, that is, to collect information during the execution and to adapt thread or memory page mapping based on this information [9, 13, 15, 18, 23]. To the best of our knowledge, only few of them also tackle the issue of characterizing the need for placement optimization [13]. Some rely on hardware monitoring features of processors to take placement decisions [9, 15, 23]. Online profiling has the advantage of being able to adapt to changes in the behavior of the application during the execution. On the other hand, offline profiling allows collecting more extensive and precise information about memory accesses and threads interactions and thus, can lead to better placement decisions for applications that have a similar behavior during the whole execution [21].

Most works on thread and/or data placement focus solely on optimizing locality [21]. However both locality and contention should be taken into account to obtain a solution that can select a good placement for a large variety of applications [15, 17]. A few solutions try to deal with locality and contention only through thread placement. However they mostly focus on cases where the number of threads is less than the number of cores [23] or where multiple applications are executed simultaneously on the platform [5, 30]. Recent studies show that best results are achieved when both data and thread placement are considered [9, 15, 19, 20]. This is confirmed in our study (see Figure 2). We observe that all the combinations of thread placement (Compact or Scatter) and data placement (First Touch or Interleave) policies are at least once the best strategy for an application.

In existing works, different metrics have been used to take placement decisions. There is no general agreement on the best metrics to use. Among the metrics that can be collected using hardware counters, cache miss rate, mostly at the LLC level, is often considered as a good indicator of contention [5, 9, 33]. However other studies show that using this metric can lead to wrong placement decisions [16]. Simple performance metrics such as MIPS (Millions Instructions per Second) have also been considered [23]. At the level of memory accesses and NUMA nodes, metrics such as memory read ratio, local access ratio or memory controllers load imbalance have also been studied [15]. More advanced metrics can be obtained with a high accuracy using binary instrumentation. Such approaches allow, among other things, collecting information about memory access patterns [33] and about interactions between threads [13, 20]. Metrics of interests regarding interactions between threads include the volume of data exchanged between threads, the frequency of the exchanges or the locality of the exchanges. As highlighted in previous studies, such interactions should be tracked at the level of cache lines [13, 20]. Our study considers a representative set of all these metrics (see Section 3) both derived from hardware counters and binary instrumentation. When tracing interactions between threads based on memory accesses, we implemented mechanisms at the granularity of cache lines.

Some works have studied the use of statistical models to predict the performance of multi-threaded applications. Castro et al. [10] proposed a machine learning approach to take thread placement decisions in the context of software transactional memory. The work that is the closest to ours is the one by Wang et al. [32]. They have proposed solutions based on Artificial Neural Networks and SVMs, using a few simple metrics as inputs such as the number of L1 misses, to predict the optimal number of threads and the best scheduling policy for these threads for a new application on a given platform. Their results are encouraging as they are able to make good predictions on average and their solution works on multiple platforms. In this study, we adopt a similar approach but we tackle a different problem: we do not consider the scalability issue, and we rather emphasize on decisions regarding both threads and data placement. Furthermore, we also try to characterize the need for applying placement policies that may be expensive to compute and apply online. Finally, our study considers a much larger set of metrics and compares the interest of binary instrumentation against hardware counters.

3 METHODOLOGY

Our approach to predict the sensitivity of applications to placement and to identify the best placement policies for threads and data can be summarized as follows. We collect a large set of data to capture the main characteristics of applications through a single run with a default placement policy. This data is obtained via transparent methods, either based on binary instrumentation or hardware counters. It is further preprocessed to build non linear models out of several supervised classifiers. Finally a model selection is performed and validated on different applications and machines.

This section starts with a few definitions. Then, the collected metrics about the execution of applications are detailed. Finally, the classification algorithms, the pre-processing transformations, and the method for training, evaluating, and validating models are presented.

3.1 Definitions

As described previously, this study focuses on two placement policies for threads (*compact* and *scatter*) and for memory pages (*first-touch* and *interleave*). We name *placement policy* or *placement strategy*, the combination of a thread placement policy and a data allocation policy. Hence, there are four possible placement strategies. We define *compact-firsttouch* as the default strategy.

We define as speedup, the ratio of the application execution time when run with a specific policy over its execution time with the default policy. If the speedup or slowdown of an application with one placement policy exceeds 10% then we tag it as *sensitive to placement*. After running each application with all policies, we were able to tag them as sensitive to placement or not and to tag

LD_INSTANT	Load instructions per cycle.
SR_INSTANT	Store instructions per cycle.
L1_MISS_REL	Data cache misses in the first level cache per memory (load or store) instruction.
L2_MISS_REL	Same as L1_MISS_REL for level 2 cache.
L3_MISS_REL	Same as L1_MISS_REL for level 3 cache.
NODE_MISS_REL	Remote memory accesses per memory (load or store) instruction.

Table 1: Metrics derived from hardware counters.

them according to their preferred policy. These tags are further used as output targets for models.

3.2 Applications Characterization Metrics

A total of 29 metrics are computed per application, based on data collected while running each application with the default placement policy. This set of metrics, described below, is used as input to the models built in our study. For each application a call to a custom library is wrapped around a region of interest[2] to collect hardware counters value using the PAPI library [29] or to start instrumenting applications binary with the Pin tool [24].

Hardware counters and approaches based on binary instrumentation offer different trade-offs. Collecting hardware counters is a cheap operation compared to binary instrumentation as the former induces negligible side effects on execution compared to a 100X slowdown and heavy memory overhead for the latter. However, these two techniques enable to get different information about applications. With hardware counters, obtained information is less about the application algorithm than about the hardware response to the application execution. Through binary instrumentation, one can extract more precise information about the application algorithm (interactions between threads, memory access patterns) but the collected data are mostly agnostic to the platform on which the application executes.

3.2.1 Hardware counters. For each application, we collect the value of several hardware counters at the end of the execution using PAPI. We select abstract counters defined by PAPI, which are supposed to be equivalent across tested architectures. They capture information about the total number of cycles used by the application, about executed instructions, about cache misses at all levels of the cache hierarchy and about accesses to the NUMA nodes. To have a meaningful comparison of these numbers between applications that may have a very different execution duration, metrics derived from these counters are expressed relatively to the total number of cycles or of data accesses. The metrics are summarized in Table 1.

3.2.2 Binary Instrumentation. Runtime instrumentation of applications memory accesses is performed with the Pin framework. Accessed memory addresses are recorded at cache line granularity to maintain a representation of the memory state (e.g., ID of the threads that have accessed the cache line, last thread that modified the cache line, *etc.*). Such mechanisms enable to compute several statistics on applications execution, including per-thread memory footprint, inter-thread communications, *etc.* Metrics derived from these statistics are listed in Table 2. For those which are not straightforward, we provide a detailed explanation below.

A communication between threads i and j is defined as the first read from j after i writes to a common cache line. The sum over all cache lines of such an event is called the communication amount between these threads. In a similar way, the amount of sharing between each pair of threads is defined as the sum over cache lines of individual cache line sharing, where sharing amount is defined as the intersection of the number of accesses for each thread of the pair to a common cache line. Inter-thread communications and sharing matrices have been used in prior works [20] to derive metrics of interest toward a solution to threads and data mapping problems.

For such matrices, their *heterogeneity*, the *mean of their normalized values* and their *balance* aim at characterizing the impact of threads mapping. Heterogeneity represents the proportion of inter-threads exchanges over accesses to private data. Balance is defined as the average matrix value over the maximum matrix value and is high if all the matrix values are close to each other, and small if one thread is communicating/sharing more than the others.

Neighbors sharing/communication fraction is defined as the sum of adjacent thread pairs communications over the sum of matrix values, and quantifies whether communications are mainly focused on direct neighbors.

The need for balance at the scale of NUMA nodes is measured with *matrices cluster deviation.* It is defined as the standard deviation across NUMA domains (*i.e.* groups of consecutive thread IDs inside the same NUMA node) of the sum of values inside the corresponding domain. It is small when memory accesses are well balanced across thread groups whereas high values may be an insight that an interleave allocation policy would be a good choice.

When moving data from one thread to another in an occurring communication event, we count the number of hops in the machine topology (cores + caches + memories) as provided by hwloc [8], to walk from source core to destination core. The *average number of hops* per communication/sharing can then characterize locality among threads with default thread placement.

3.3 Modeling Methodology

Models built in this paper[3] follow a classic pipeline of sampling, transformations, learning and models selection. Obtained models are evaluated based on their ability to generalize their predictions to new applications, new platforms, or even both at the same time.

Generalization to new applications employs a *one-versus-all* training scheme to train models. It consists in removing two applications from the applications set, then training the model on all applications but the two removed (*i.e.* the *training set*), and finally performing a prediction on the first removed application (*i.e.* the *test set*) and the second removed application (*i.e.* the *validation set*). This training process is repeated for all applications pairs and the models average performance on predictions for all applications is reported.

Splitting the dataset in this way is necessary to perform validation after the model selection stage. Indeed, chances are that

[2]The region of interest of applications corresponds to the outer most computational part, called once during application lifespan, and positioned in between data initialization and cleanup.

[3]The complete implementation of the modeling methodology with the dataset described in Section 4 has been published on Code Ocean platform [28] for reproducibility.

sharing/communications_CC	Average normalized amount of sharing/communications per thread
sharing/communications_CB	Sharing/communication matrix balance
sharing/communications_CH	Sharing/communication matrix heterogeneity
sharing/communications_NB	Neighbors sharing/communication fraction
sharing/communications_clusterSD	Inter NUMA cluster sharing/communications deviation
sharing/communications_hop_element	Average number of topology hops per sharing/communication
avg_sharing_degree	Average number of threads sharing a cache line
sd_sharing_degree	Deviation of number of threads sharing a cacheline
avg_write_ratio	Average writes over memory access per cache line
sd_write_ratio	Deviation of writes over memory access per cache line
avg_shared_write_ratio	same as avg_write_ratio per shared cache line
sd_shared_write_ratio	same as sd_write_ratio per shared cache line
avg_write_degree	Average number of threads writing a cache line
sd_write_degree	Deviation of number of threads writing a cache line
footprint	Number of cache lines accessed
sd_thread_footprint	Deviation of number of cache lines accessed per thread

Table 2: Metrics obtained through binary instrumentation.

choosing a model among thousands based on a few tenths of predictions may draw a lucky one or an overfitting one. Using a validation set aims at reducing this risk. More precisely the model selection methodology is the following: The 1% top performers on the test set are selected, then the best performer on the validation set among remaining models is elected as the final model. This selection decreases the risk of presenting overfitting models.

When modeling for generalization to new machines, models are trained with all applications on a single platform. Each model is then used for predictions on another platform (the test set), and again on another one (the validation set). In the third scenario (a new application on a new machine), we combine both training schemes, *i.e.* we remove two applications from the training set, use the training set on a single machine then predict the two unused applications each on a different unseen machine.

The modeling process includes several preprocessing stages on collected data. These steps are designed to fit more complicated models and perhaps obtain a better quality on the predictions. When needed, they are calibrated on the training set, then applied with the same settings on the test and validation sets. For instance, when normalizing data, the center and amplitude of the dataset is computed on the test set, then normalization of other sets utilizes the same center and amplitude. These steps are: the normalization of inputs, (optional) singular value decomposition [22], and (optional) polynomial transformation [3] of inputs, up to degree two.

Pre-processed applications metrics are given as input to learning classifiers to match applications label, *i.e.* sensitivity to placement or preferred placement policy. We use the classifiers as black boxes: we use the default classification functions without tuning their hyper-parameters. Classification building blocks used are the following: Random-Forest [6] from R package `randomForest`, logistic regression [4] from native R `glm` function, Support Vector Machines [11] from R package `kernlab` and Artificial Neural Network with one hidden layer of four neurons from R package `nnet`. We chose to use classifiers as black boxes because our main goal is to demonstrate the feasibility of the approach and not to find the most optimal classification algorithm for our problem. Furthermore, using default classification functions shows that one can apply our methodology without having a high degree of expertise in machine learning.

Training all models requires several tens of hours on a quad-core desktop computer. Training time is linked to the number of parameters. For the Random Forest algorithm, it grows exponentially relatively to this number. In order to explore all parameters despite this limitation, the number of parameters fed to the models is capped. Thus, the exploration of the parameters space is achieved by randomly sampling parameters, and training models with many different parameters sets. The input sampling, pre-processing step, and type of learning kernel are the models "hyper-parameters" which are explored. We train a model for each possible combination of pre-processing and learning kernel and for many input parameters sets.

4 EXPERIMENTAL TESTBED

Our evaluation includes a set of 27 proxy-applications from the Coral [12], Parsec [2] and NAS [1] benchmarks suites (see Table 3). Most applications are HPC applications except for the Parsec suite which embeds other types of applications. Three computing systems are used to run applications. Some of their features are detailed in Table 4. Machines of this testbed originate from the same vendor and have consecutively been released for similar server computing systems. The three machines have hyper-threading disabled and are configured to virtually split processors into two NUMA nodes[4]. It is worth mentioning that the micro-architecture differences are greater between Broadwell and Skylake [31] than between Haswell and Broadwell [7]. In particular, Skylake has larger private L2 caches but smaller LLCs, and it replaces the ring interconnection between cores inside a chip with a mesh network, both with potential implications on locality and contention.

Applications	run parameters
bodytrack, canneal, freqmine, swaptions	input native
fir, del_dot_vec_2d, energy_calc_alt, vol3d, couple, pressure_calc_alt, pic_2d	*NA*
lulesh2.0	-b 4 -s 100 -i 40 -r 100
MILCmk	nmax = 256*1024*16
HACCmk	count=200
lu, cg, ep, mg, sp, bt, ft, sp	class A, B

Table 3: The set of applications and their parameters.

[4] *Cluster-on-Die* for Haswell and Broadwell, *Sub-NUMA-Cluster* for Skylake.

Microarchitecture	Model	Sockets × Cores	NUMA Nodes	Shared L3	Private L2
Haswell	Xeon E5-2680 v3	2 × 12 (2.5GHz)	4 × 32GB	4 × 15MB	24 × 256kB
Broadwell	Xeon E5-2650L v4	2 × 14 (1.7GHz)	4 × 16GB	4 × 18MB	28 × 256kB
Skylake	Xeon Gold 6140	2 × 18 (2.3GHz)	4 × 24GB	2 × 25MB	36 × 1024kB

Table 4: Experimentation platforms.

Figure 2 represents speedups[5] for the best and worst execution time of each application on the Skylake machine when trying the four placement strategies. A 100% speedup corresponds to the performance with the default placement policy. If the best or worst performance exceed the 10% threshold materialized with a dashed line, the application is tagged as placement-sensitive. On this machine, a majority of applications prefers the default policy. However, placement sensitive applications can perform significantly better or worse with alternative placement policies.

After running all applications with all placement policies, we are able to tag them as sensitive or not according to our definition (see Section 3.1). Table 5 presents statistics about applications sensitivity to placement with respect to the platform. On the diagonal, the percentage of applications that are placement sensitive on a given platform is reported. Out of the diagonal, the percentage of applications that remain in the same class when changing the platform is provided. From this table, it is clear that there is no easy assumption, neither for predicting the sensitivity of new applications nor for predicting the sensitivity of known applications on new machines. On average, there is a good balance between sensitive and insensitive applications. This observation motivates the need to identify placement-sensitive applications, in order to avoid useless placement computation. Also, up to 33% of them may change their status from one machine to another, which shows that predictions from one machine to another is not straightforward.

	Haswell	Broadwell	Skylake
Haswell	0.59	0.74	0.67
Broadwell	-	0.41	0.85
Skylake	-	-	0.48

Table 5: Prediction accuracy of a basic prediction mechanism for sensitivity to placement. On the diagonal, maximum achievable accuracy ratio when predicting a constant output on a given machine. Out of the diagonal, accuracy ratio when assuming that all applications remain in the same class when changing the machine.

Considering the subset of placement-sensitive applications, Table 6 presents the average achieved speedup when using one machine preferred policies to run on another machine. The diagonal shows the average of top achievable speedup, on each specific machines as a comparison point. Using conservative choices across machines is usually a good choice compared to using the default policy (*i.e.* speedup > 1). Nevertheless, our results (see Section 5) show that it is possible to improve from this strategy.

[5] Speedup as well as hardware counters values are median values over 6 identical runs.

	Haswell	Broadwell	Skylake
Haswell	1.16	1.03	1.06
Broadwell	1.06	1.06	1.08
Skylake	1.10	1.04	1.12

Table 6: Average speedup for placement-sensitive applications when applying the per-application best policy of machine A (rows) on machine B (columns).

5 MODELS PERFORMANCE AND REAL WORLD PRACTICABILITY

This section is devoted to observe models performance and conclude on practicability of placement-sensitivity detection and placement policy selection. For each of these two objectives, we describe the performance of models along all the proposed dimensions: generalization abilities to new applications and machines, with hardware counter versus instrumentation metrics.

5.1 Sensitivity to Data and Threads Placement

Following the proposed methodology, a set of models is trained to detect if an application is placement sensitive. For this objective, the metric of importance is accuracy, *i.e.* the average number of good predictions among all applications. In Figure 3 is represented an evaluation of models accuracy on the validation set when predicting applications sensitivity to data and threads placement. On the diagonal, models have been trained with applications running on a single machine and prediction are made on unseen applications on the same machine. The performance baseline represented by a horizontal dashed line is the accuracy obtained when always predicting that applications are sensitive. Outside of the diagonal, presented results are about models trained with all applications on a system (rows) to predict the same applications on another one (columns). In these cases, the baseline, represented by a horizontal dashed line, is the accuracy obtained when predicting the same output as on the original machine.

The analysis of Figure 3 delivers several conclusions about modeling sensitivity to placement for new applications or new platforms. First, in every scenario, sensitivity to placement can be predicted with around 80% accuracy. Models always improve compared to the baseline even in the case of cross-platform predictions. We also note that using hardware counters provides similar results as using binary instrumentation metrics. It implies that with a very low overhead application profiling, our models can fulfill the goal of predicting placement sensitivity with a good accuracy both for new applications and for new architectures.

We investigate the wrong predictions of new applications made by the best models per machine in Table 7. Considering the misclassified applications, the table presents the average maximum

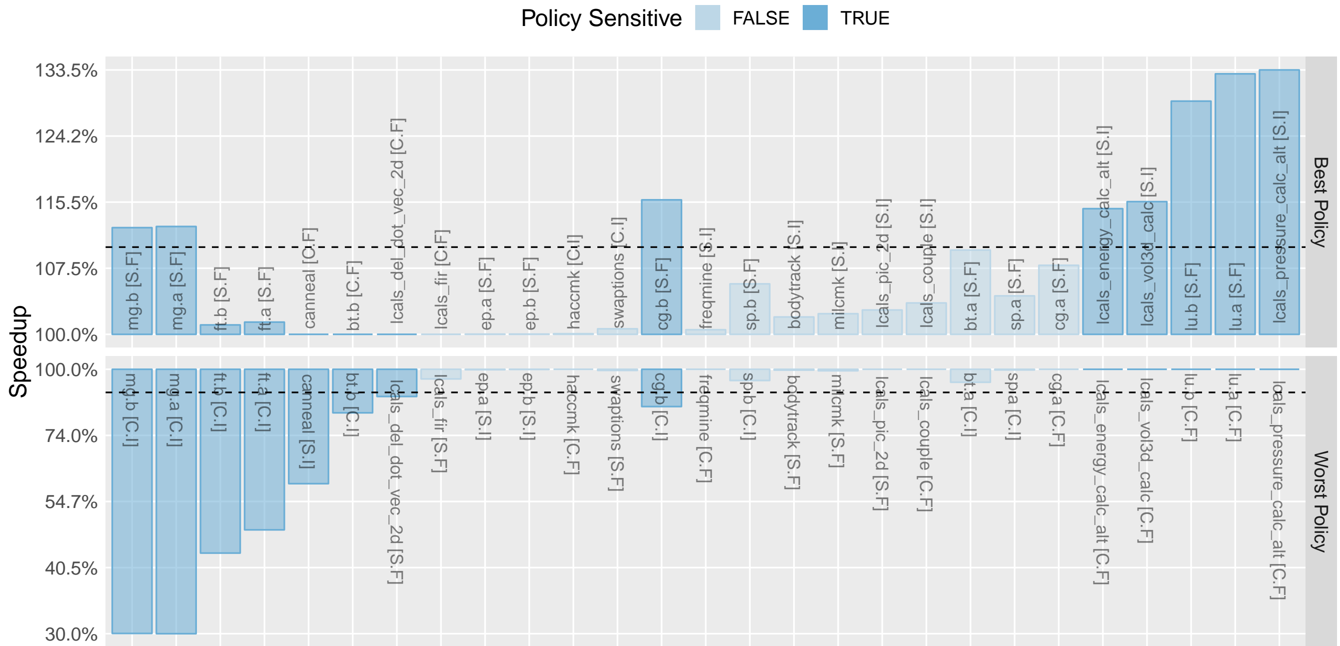

Figure 2: Applications speedups with best and worst placement policies and placement sensitivity on the Skylake platform. Policies are noted *[thread policy, data policy]*, with *C* for compact, *S* for scatter, *F* for firsttouch, and *I* for interleave.

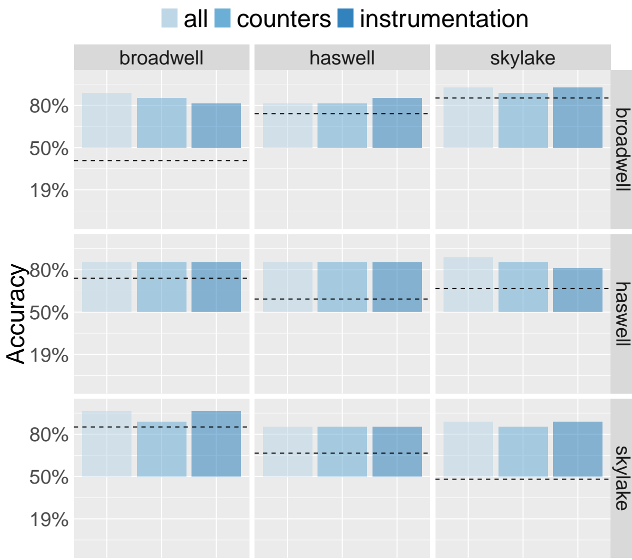

Figure 3: Accuracy on validation set, when predicting applications sensitivity to data and threads placement. On the diagonal, models predict new applications on a given machine (baseline: "always sensitive" predictions). Out of the diagonal, models predicts for already seen applications but on a new machine (baseline: same answer on training platform and on prediction platform).

distance between the speedup achievable with an application and the boundaries set by our definition of placement sensitivity. The results show our wrong predictions would have only little impact in practice as the misclassified applications are not very sensitive to placement.

machine	counters	instrumentation	all
Haswell	0.04	0.06	0.04
Broadwell	0.05	0.07	0.04
Skylake	0.05	0.07	0.02

Table 7: Average distance to the classification boundary of misclassified applications when predicting new applications only. Average distance to the classification boundary is reported per *machine* and type of model inputs (*counters, instrumentation, all*).

In Figure 4, is presented the performance of models on the test and validation sets, when predicting a new application on a new platform. Each column describes a result with training from a source machine and predicting for a different one. Violin plots in the first line exhibit models performance distribution on test set. Models of individual violins vary in the preprocessing performed, the learning algorithm used and the set of input parameters. A violin plot is drawn per set of input parameters: hardware counters, instrumentation metrics or both. In the second line, outcome of model selection is presented with the best models performance on the validation set (*i.e.* different machines and applications than for training and testing).

Additional conclusions come with Figure 4. Models performance on the validation set consistently reaches more than 75% accuracy, this time with both a new application and a new machine. Once again, the accuracy difference between using hardware counters

or others is not significant. The accuracy distribution on the test set (first row) shows that only few models reach the highest performance. Model validation (second row) reaches similar accuracy as the results obtained with the test set, which suggests that models are not overfitting.

Overall, detecting sensitivity to thread and data placement with the set of proposed parameters is not trivial. However, we managed to achieve it with at least 75% accuracy which significantly improves from basic strategies, and with errors on least sensitive applications. We also found that using hardware counters can provide similar performance as specific binary instrumentation. Thus, the most practical models, *i.e.* with low overhead application profiling are satisfying enough compared to more complex application characteristics. Moreover, these models are robust to new users out of the training context, *i.e.* with different applications and/or different processors with a close architecture.

5.2 Preferred Policy Classification

Similarly to placement sensitivity detection, models are trained toward the objective of choosing the preferred placement policy for sensitive applications. For this objective, we do not look at models accuracy but rather at predictions impact on applications execution time. We consider that mistakes may be acceptable when they have little impact. The performance of models is therefore presented as the geometric mean of applications speedup using predicted policies, *i.e.* it embeds a lower penalty if wrong choices have small impact on the execution time.

The performance of models when generalizing to new applications or to new platforms is explored in Figure 5. Similarly to Figure 3, results are organized in rows expressing the platform on which models were trained and columns on which the validation was done. On the diagonal, models were trained to predict new applications on a single platform, whereas out of the diagonal, models were trained to predict same applications on a different platform. In the first scenario, the application set is restricted to applications sensitive to threads and data placement on a single machine. The upper bound represented as a top dashed line correspond to average achieved speedup when always predicting best placement policies, and the baseline represented as the bottom dashed line correspond to always picking the default placement policy. In the latter scenario, out of the diagonal, the application set is restricted to applications which are sensitive to placement on at least one machine. In that case, the baseline is the achieved speedup when adopting a conservative policy selection.

Figure 5 exhibits very good models performance. Whether it is on a new application or a new platform, models predictions always outperform the baseline and consistently reach near-optimal placement policy choice. And this is true for all categories of input parameters. The only exception is when using hardware counters to predict for new application on the Haswell machine. In this case, the performance is only slightly improved compared to the baseline, which means that the 1% top models on the test set are all overfitting for this particular case.

The overall efficiency remains valid in Figure 6. In this figure, the case of generalization to both new applications and new machines at the same time is explored. The distributions of models performance on test sets is presented in the first row. In the second row, the best model performance on the validation set after the model selection stage is exhibited. These performances are to be compared with the upper and lower-bound of achievable performance represented as top and bottom horizontal dashed lines, when always predicting respectively the best and worst placement policies for each application. The performance is also to be compared with the baseline where applications are run with the default placement policy. In columns, models are organized by training-to-prediction machine pairs. Models performances are presented considering three set of metrics for training: metrics based on hardware counters, metrics based on binary instrumentation, both.

In Figure 6 again, predictions on the validation set reach near-optimal speedup for every machine. The results highlight the generality and practicability of the models. Indeed, we were able to get very good speedups, even when both the application and the machine are unknown, and with a low overhead profiling (*i.e.* hardware counters). There is a little discrepancy with hardware counters models in the couple (Haswell, Skylake) which might be attributed to the greater difference between these two systems compared to other couples. The validation set performance is consistent with the performance achieved on test set and shows that selected models are not likely to be overfitting.

6 CONCLUSION

At the scale of multi-socket shared memory NUMA platforms, thread and data placement matters for performance. However, finding an optimal strategy, even with post-mortem analysis is far from trivial. This difficulty arises from the numerous interactions and trade-offs in the whole memory hierarchy. Although some of them have been tackled in previous studies, improving placement decisions while considering several architectures and several sources of applications characterization had yet to be done.

In this paper we implemented a thorough modeling methodology toward two main goals: i) assessing the ability for meta-models to predict if an application will at all be impacted by the placement of its threads and data, and in such a case, ii) finding the best placement policy. We successfully answered these two questions. We were able to train models that can predict the sensitivity of applications with an accuracy of more than 80%, wrong prediction being mostly done on applications that are barely sensitive to placement. In all cases, we were also able to select a good placement policy so as to achieve almost optimal performance among the studied policies.

Our results show that the obtained models can be easily used in practice and are general enough to be used in many use-cases. First, models comparisons show that good results are achieved by solely using hardware counters to collect metrics. It implies that models can be trained and make predictions based on a single run of applications and with a low overhead profiling technique. Second, our evaluation show that we are not only able to make accurate predictions for unseen applications on a given machine. We are also able to make prediction on an unknown machine, and even in this case, for new applications.

The results presented in this paper open new research directions. The proposed methodology was applied in the context of applications using fork-join parallelism and with simple placement

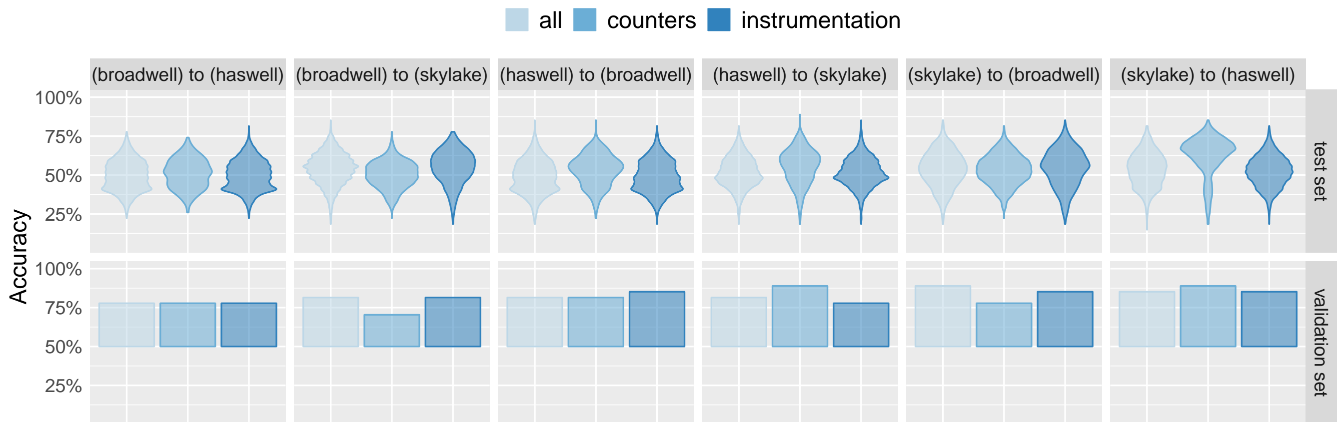

Figure 4: Models accuracy for placement-sensitivity predictions when generalizing both to new applications and new machines. The first row, show the models accuracy distribution on test set. The second row show the best models accuracy on validation set. Each column show-cases models trained on a particular platform with predictions on another platform.

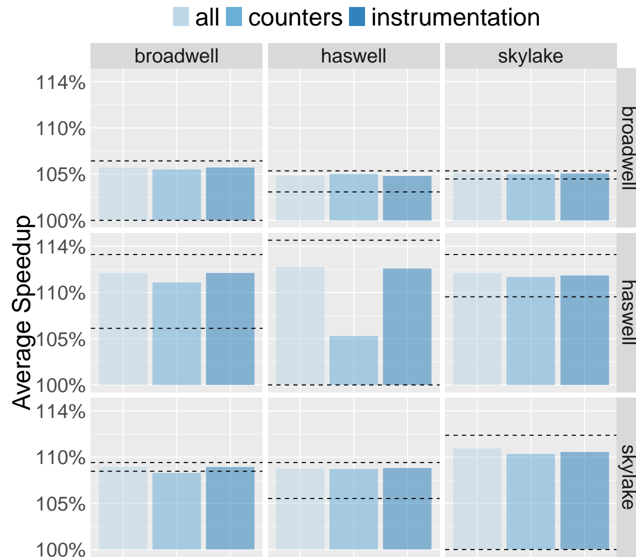

Figure 5: Speedup obtained with predicted placement policies on validation set. On the diagonal, models predict new applications (baseline: using the default placement policy). Out of the diagonal, models predict same applications on a new machine (baseline: conservative placement strategy, same choice on prediction platform as on training platform).

policies. In the future, we would like to extend the applicability of the approach by adding more policies or by building models for different parallelism paradigms, *e.g.* task-based programming. Our results also show that although hardware counters and binary instrumentation capture very different metrics, models based on each these sources of data achieve similar performance. Furthermore, in many cases only a small number of parameter sets allow making good predictions. The question of understanding what are the best metrics to use to take placement decision should be investigated.

ACKNOWLEDGMENTS

Some experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LaBRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS (and ANR in accordance to the programme d'investissements d'Avenirs, see https://www.plafrim.fr/). The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.

REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. *The NAS Parallel Benchmarks.* Technical Report. The International Journal of Supercomputer Applications.
[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08).* 72–81.
[3] Christopher M Bishop. 2006. *Pattern recognition and machine learning.* springer, Chapter 1, 4–11.
[4] Christopher M Bishop. 2006. *Pattern recognition and machine learning.* springer, Chapter 4, 179–207.
[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. 2010. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)* 28, 4 (2010), 8.
[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[7] Broadwell chip changes from Haswell 2014. Broadwell chip changes from Haswell. https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client). *Accessed 03/22/2019.*

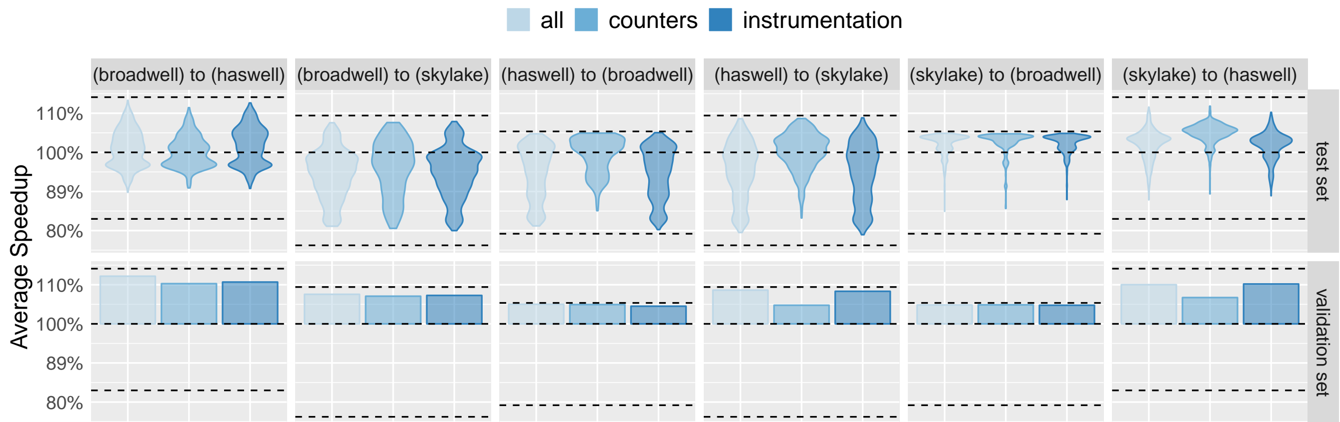

Figure 6: Speedup obtained when predicting new applications preferred execution policy on new machines. Results are detailed per training-to-prediction machine pairs and kind of input metrics. The first row displays all models distribution on the test set, while the second row displays the best model performance for each group on the validation set.

[8] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*.

[9] 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* (2010).

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

[11] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

[12] Coral 2018. The Coral Benchmarks Codes. https://asc.llnl.gov/CORAL-benchmarks/. *Accessed: 2018/04/16*.

[13] Eduardo H.M. Cruz, Matthias Diener, Marco A.Z. Alves, and Philippe O.A. Navaux. 2014. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *J. Parallel and Distrib. Comput.* 74, 3 (2014), 2215 – 2228.

[14] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, and Philippe OA Navaux. 2019. EagerMap: a task mapping algorithm to improve communication and load balancing in clusters of multicore systems. *ACM Transactions on Parallel Computing (TOPC)* 5, 4 (2019), 17.

[15] 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 (ASPLOS '13)*. 381–394.

[16] Andreas de Blanche and Thomas Lundqvist. 2015. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing* (2015), 1–33.

[17] Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. 2015. Locality vs. Balance: Exploring data mapping policies on NUMA systems. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 9–16.

[18] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2015. Communication-aware Process and Thread Mapping Using Online Communication Detection. *Parallel Comput.* 43, C (March 2015), 43–63.

[19] Matthias Diener, Eduardo HM Cruz, Philippe OA 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*. 277–288.

[20] Matthias Diener, Eduardo H.M. 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 (2015), 18 – 36.

[21] 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. *ACM Computing Survey* 49, 4, Article 64 (Dec. 2016), 38 pages.

[22] V. Klema and A. Laub. 1980. The singular value decomposition: Its computation and some applications. *IEEE Trans. Automat. Control* 25, 2 (April 1980), 164–176.

[23] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. 2011. autopin–automated optimization of thread-to-core pinning on multicore systems. In *Transactions on high-performance embedded architectures and compilers III*. 219–235.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. 190–200.

[25] Zoltan Majo and Thomas R. Gross. 2011. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. *SIGPLAN Not.* 46, 11 (June 2011), 11–20.

[26] Jaydeep Marathe and Frank Mueller. 2006. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. 90–99.

[27] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *J. Parallel and Distrib. Comput.* 70, 12 (Dec. 2010), 1204–1219.

[28] Methodology Implementation 2019. Methodology Implementation for Reproducibility of the Paper Experiments. https://codeocean.com/capsule/8469721/tree. *Accessed: 2019/04/20*.

[29] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[30] Petar Radojkovic, Vladimir Cakarevic, Javier Verdu, Alex Pajuelo, Francisco J Cazorla, Mario Nemirovsky, and Mateo Valero. 2013. Thread assignment of multithreaded network applications in multicore/multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2013), 2513–2525.

[31] Skylake chip changes from Broadwell 2015. Skylake chip changes from Broadwell. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). *Accessed 03/22/2019*.

[32] 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 (PPoPP '09)*. 75–84.

[33] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. 129–142.

[34] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 4.