Empirical Model-Based Performance Prediction for Application Mapping on Multicore Architectures

Abdoulaye Gamatié, Xin An, Ying Zhang, An Kang, Gilles Sassatelli

Empirical Model-Based Performance Prediction for Application Mapping on Multicore Architectures

Abdoulaye Gamatié[a], Xin An[b], Ying Zhang[b], An Kang[b], Gilles Sassatelli[a]

[a] *LIRMM, CNRS - University of Montpellier, 161 rue Ada, 34095 Montpellier, France*
[b] *Anhui Province Key Laboratory of Affective Computing and Advanced Intelligent Machine, Hefei University of Technology, No.193 Tunxi Road, 230009 Hefei, China*

Abstract

Application mapping in multicore embedded systems plays a central role in their energy-efficiency. The present paper deals with this issue by focusing on the prediction of performance and energy consumption, induced by task and data allocation on computing resources. It proposes a solution by answering three fundamental questions as follows: i) how to encode mappings for training performance prediction models? ii) how to define an adequate criterion for assessing the quality of mapping performance predictors? and iii) which technique among regression and classification enables the best predictions? Here, the prediction models are obtained by applying carefully selected supervised machine learning techniques on raw data, generated off-line from system executions. These techniques are Support Vector Machines, Adaptive Boosting (AdaBoost) and Artificial Neural Networks (ANNs). Our study is validated on an automotive application case study. The experimental results show that with a limited set of training information, AdaBoost and ANNs can provide very good outcomes (up to 84.8% and 89.05% correct prediction score in some cases, respectively), making them attractive enough for the addressed problem.

Keywords: Resource allocation, Application mapping, Model-based performance prediction, Machine learning

*Corresponding authors: A. Gamatié and X. An
Email addresses: abdoulaye.gamatie@lirmm.fr (Abdoulaye Gamatié), xin.an@hfut.edu.cn (Xin An), yingzhang9403@163.com (Ying Zhang), kangan03@mail.hfut.edu.cn (An Kang), gilles.sassatelli@lirmm.fr (Gilles Sassatelli)

1. Introduction

Multicore and manycore architectures have become the *de facto* solutions to meet the energy-efficiency requirement in modern computer systems. The aim is to provide the systems with higher performance levels at the cost of minimal power consumption. Typically, for high-performance and embedded computing systems, this amounts to maximize the number of floating-point operations per second (FLOPS) and the millions of instructions per second (MIPS) respectively, per consumed Watt. Nevertheless, the advantage of multicore architectures comes with a non-trivial resource allocation challenge on which depend the energy-efficiency gains. As a matter of fact, the mapping and scheduling of both tasks and data on available processing cores and memory have a strong impact on performance and power consumption.

Existing mapping methodologies [1] adopt either design-time or runtime optimization approaches to improve the behavior of both homogeneous and heterogeneous multicore systems. At runtime, the mapping management may incur data/tasks migrations onto the available computation resources. This is orchestrated in various ways: either centralized or distributed. Generally speaking, the problem of finding optimal mapping and scheduling solutions is known to be NP-hard. Some pragmatic approaches that address this problem exploit heuristics combined with domain-specific knowledge to explore nearly optimal solutions [1]. Having the relevant information on system behavior according to variable runtime situations is one major challenge in such adaptive system management [2]. Collecting these information (e.g., CPU usage, memory and communication interconnect usage) is often tedious and intrusive to the system, especially when targeting fine-grained data.

Given the important progress made recently in machine learning techniques, particularly in deep-learning [3], we envision opportunities to apply them when dealing with application mapping in multicore systems. Machine learning has gained an increasing attention in system design, including computer architec-

tures [4] or compilers [5]. To predict the performance of mappings, *supervised* machine learning techniques are considered in this work. They enable to build class or value prediction models while minimizing a *loss* function denoting the prediction error percentage on the training data set. On the other hand, *unsupervised* machine learning techniques enable to identify clusters of similar behavior or to determine insightful feature representations from raw data sets. Beyond these techniques, which are usually applied off-line, other approaches such as *reinforcement learning* and *evolutionary algorithms* enable online learning.

1.1. Context of this Study

We consider the dynamic resource allocation question in multicore systems, as illustrated in Figure 1. Application workloads are described by hierarchical *task graphs*, where each task consists of a *runnable* graph [6]. Runnables are basic entities defining task behaviors in terms of runtime and communication. A mapping *performance predictor* is coupled loop-wise with a *mapping heuristics module*, which implements typical mapping selection techniques (e.g., evolutionary algorithms) on a given multicore *execution platform*. A component, called *workload mapper*, is in charge of applying the selected mapping decisions at runtime. It acts as a centralized processing element that realizes every mapping suggested by the *mapping heuristics module*.

The dynamic resource allocation question has been thoroughly covered in a recent book [7], considering application domains such as high-performance computing, cloud computing and embedded computing. Several approaches have been discussed: allocation and optimization inspired by control automation theory, search-based allocation heuristics such as genetic algorithms, distributed allocation based on swarm intelligence, and value-based allocation. These approaches are typical candidates for implementing the above *mapping heuristics module*.

The performance predictors, investigated in the current work, are the ideal complements of the above *mapping heuristics module*. Indeed, the predicted

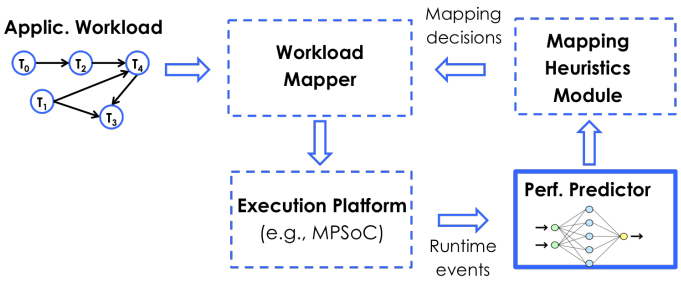


Figure 1: Dynamic resource allocation in multicore systems

performances, e.g., execution time, speedup or energy-efficiency, help in taking efficient mapping decisions at runtime. Note that in place of performance predictors, alternative candidates are performance evaluation tools, such as multi-core system simulators, analytic methods or worst-case performance estimation methods. However, these solutions may come with an overhead in the global execution time due to their inherent simulation time; or to their pessimistic over-approximations. To avoid this issue, here, we rather investigate an empirical approach that leverages prediction models trained on raw data generated off-line from different system execution scenarios. The models are built with machine learning techniques capable of extracting useful insights from system behavior. When invoked, they are expected to predict estimates of mapping performances in little-to-no time (e.g., for usage in fitness functions of genetic algorithm-based heuristics). These estimates must be relevant enough to enable the *mapping heuristics module* to take efficient decisions. While the current work does not aim at any new mapping heuristics, its main purpose is to speedup the decision loop shown in Figure 1, by reducing the computation complexity associated with the *performance predictor* leveraged by the *mapping heuristics module*.

1.2. Problem Formulation

The problem dealt with in this paper is defined as follows:

Definition 1 (Mapping Performance Prediction Problem). *Given an application to execute on a multicore platform, we are interested in its mapping issue onto the available cores. Here, the mapping is addressed at the granularity of the runnables. We consider machine learning techniques to predict the performance induced by the possible mapping choices, while meeting the following requirements:*

1. ***accuracy:*** *the successful prediction percentage reaches at least 80%;*
2. ***feasibility:*** *data used for learning are obtained at minimal and costless intrusion in systems;*
3. ***responsiveness:*** *predictions are performed in short delays.*

Intuitively, the above prediction issue is a regression problem, i.e., given a mapping scenario, we would like to predict its induced performance numbers. However, if we partition the domain of all possible values into sub-domains and predict the sub-domain to which the performance numbers of a given mapping scenario belong to, the above problem can be formulated then as a classification problem. Each sub-domain is seen as a class (or a label). For example, one may want to map an application according to three target performance ranges or classes: high, medium and low. A classification technique would be therefore preferred. Accordingly, if we refine the number of classes into more classes, fine-grained and more accurate predictions could be obtained.

1.3. Our Contribution

We address the above mapping problem by considering two off-line supervised machine learning approaches: on the one hand, classification through *Support Vector Machine* (SVM) [8] and *Adaptive Boosting* (AdaBoost) [9] techniques, and on the other hand, *regression* by using *Artificial Neural Networks* (ANNs) [10]. These approaches have been widely applied with a great success in machine learning problems [11]. SVM has been very popular in machine learning thanks to its ability to apply in both classification or regression problems, even though it is often used in the former. AdaBoost provides an original

vision combining different learners that enable accurate classifications while acting together. On the other hand, ANNs have been proved powerful enough to solve various regression problems. Compared to classification techniques, finding a good compromise between accuracy and training cost is however more challenging with ANNs due to their tedious parameterization.

To solve the mapping problem, three fundamental questions are identified and answered throughout this paper: *i) how to encode mappings for training performance prediction models? ii) how to define an adequate criterion for assessing the quality of mapping performance predictors?* and *iii) which technique among regression and classification enables the best prediction rates?* In this paper, we mainly consider execution time and energy consumption as target performance metrics to predict.

Based on these questions, the main contributions of the current paper are summarized as follows:

- different representations trade-offs are analyzed regarding mapping encodings for prediction model training. The aim is to identify a simple representation, which is compact and informative enough to be tractable with the selected machine learning techniques. Three mapping encoding variants are compared. They all capture the positions of execution entities and data in a given multicore system, under the form of vectors or matrices of topological coordinates.

- a custom metric for assessing the prediction accuracy is proposed, which fits well the mapping problem formulated above. The usual accuracy measure relies on the difference, i.e., error percentage, between predicted values and actual values: the lower this difference the better the prediction. It is not necessarily well-adapted for the mapping problem, especially when considering the potential imprecision affecting the values predicted by regression. The proposed metric relies on a relative comparison: it checks whether the performances induced by a pair of mappings are relatively comparable in the same way w.r.t. to their actual and predicted values.

For instance, if the actual performance of a mapping (computed here with a multicore system simulator) is actually better (or worse) than that of another mapping, then this also holds for their respective predicted performances. We refer to this metric as the *percentage of successful tendency prediction* (or PSTP for short).

- a comparative study of the considered supervised machine learning approaches is carried out on an automotive application case study, composed of several tens of execution entities. A suitable mapping encoding is selected from the above analysis and the PSTP metric is applied to evaluate the considered classification and regression based machine learning techniques. The training process is done off-line and the resulting prediction models are usable for online prediction. Our results show that, under some conditions, AdaBoost and ANNs can enable respectively up to 84.8% and 89.05% prediction accuracy w.r.t. PSTP, which is relevant enough for steering efficient resource allocation decisions.

The above contributions rely on our preliminary work, published in a conference [12], now extended with the following new results: a formalization of used design concepts (Sections 4.1, 4.2 and 5.2); the application of two additional supervised machine learning techniques (Sections 6.2 and 6.3); the improvement of the ANN-based evaluation (Section 6.4); and a comparison of all three techniques w.r.t. an application case study (Section 7).

Organization of the paper. The rest of the paper is organized as follows: Section 2 discusses some related work; Section 3 introduces the machine learning techniques selected in this study; Section 4 describes our system design framework; Section 5 addresses how to effectively use the selected machine learning techniques for solving the mapping performance prediction problem; Section 6 evaluates the machine learning techniques on an application case study; Section 7 discusses some important outcomes resulting from these evaluations; finally, Section 8 gives concluding remarks and perspectives.

2. Related Work

Application mapping on multicore platforms has been studied for decades in literature [1]. To find out near-optimal mapping solutions, many mapping techniques adopt search-based approaches combined with some analyses to evaluate considered mappings w.r.t. the design requirements. The analyses typically rely on system-level simulations of application specification in C on FPGA platform [13], on analytical models [14, 15] for a fast evaluation of different mapping scenarios, or on UML-based model-driven design frameworks [16].

Some recent approaches distinguish themselves from others by advocating machine learning techniques to address the mapping problem. This trend is surveyed in [17]. The authors discuss the usual control methods employed to achieve the runtime management: mapping, dynamic voltage and frequency scaling (DVFS), and dynamic power management to optimize power/energy consumption. Then, cover a number of approaches relying on *reinforcement* learning and *supervised* learning. In [18], reinforcement learning is applied through a cross-layer system approach to predict the best energy-performance trade-off in multicore embedded systems. It relies on a biologically-inspired runtime power management framework implementing a Q-learning algorithm, which selects the voltage-frequency levels to minimize energy consumption. The Q-table is made up of state-action pairs, where a state represents the CPU cycle count and current performance, an action represents the appropriate voltage-frequency values to set up. Despite its attractive features, reinforcement learning is not easy to deploy in practice for various reasons (overhead of online learning, difficult setting of learning parameters, e.g., reward function – see [19]). For this reason, we rather consider supervised learning in this paper, as in the related work discussed next.

Generally speaking, when applying learning techniques to the mapping problem w.r.t. a given optimization goal (e.g., performance metrics), one usually needs to investigate either key parameters, such as the number of threads to be partitioned, the task/thread-core binding choices, which influence the opti-

mization goal; or simply the performance metrics of interest. He or she could then formulate the target problem as a learning problem with the corresponding learning features in order to predict the values of the parameters. Most of learning features found in existing works are: either *application-specific attributes*, such as number of loops and branch instructions; or *hardware resource-specific attributes*, such as cache and memory size and architecture; or system *runtime execution statistics*, such as cache miss and hit rates. Based on these criteria, we classify a selected related work as summarized in Table 1.

In [20], the authors propose a methodology named SMiTe to predict the performance interference on simultaneous multi-threading (SMT) processors. It employs a suite of software "stressors" to quantify applications' contention characteristics defined as sensitivity and contentiousness of shared resources, e.g., cache memories. A regression-based prediction model is then built by using measurements of such characteristics to predict the level of performance degradation that applications may suffer from co-locations. In [21], the authors develop statistical power models by using linear regression to estimate per-core power consumption. Only a small number of parameters such as the CPU cycles and L1 instruction/data cache access rates of each core are selected as the input features to train prediction models. The experimental results show that they could offer simple yet accurate enough power prediction models.

A machine learning based approach is proposed in [22] for the optimal mapping of streaming applications described by the StreamIt formalism onto dynamic multicore processors. To maximize the system performance, the authors employ a k-Nearest Neighbors (KNN) model to predict the best number of threads for streaming applications and a linear regression (LR) model to predict optimal number of cores for threads allocation. Input features are extracted by using correlation analysis. Fine-grained features such as number of distinct multiplicities and number of unconditionally executed blocks for KNN, average number of conditional blocks and average size of all blocks for LR have been used. In [23], the authors apply machine learning to predict execution time, memory and disk consumption of two bioinformatics applications deployed on

Table 1: Summary of discussed learning techniques for application mapping. LR= linear regression, KNN = k-Nearest Neighbors, DT = Decision Tree, MTL= Multi-Task Learning, CPI = Cycles Per Instruction. The symbol $*$ denotes application-specific features, the symbol $\circ$ denotes hardware resource-specific features, and the symbol $\bullet$ indicates system runtime execution behavior attributes.

References	**Optimization goals**	**Predicted parameters**	**Learning techniques**	**Typical input learning features**
[20]	server utilization	performance	LR	$\bullet$ sensitivity and contentiousness of shared resources, e.g., L2 cache
[21]	power	per-core power	LR	$\bullet$ CPI, L1 cache access rates
[22]	performance	numbers of threads & cores	LR, KNN	$*$ numbers of unconditional execution blocks, loops and vector operations, etc.
[23]	performance & resource usage	performance & resource usage	LR, KNN, DT, SVM, ANN	$*$ nucleotide sequence length, taxa size $\circ$ CPU clock, amount of cache and memory
[24]	performance & resource usage	straggler task	MTL	$\bullet$ CPU, memory, network and disk utilizations
[25, 26]	performance & energy	processor type & frequency	SVM	$*$ number of conditional branch instructions and number of successors to a basic block
[27]	energy	core type, voltage & frequency	LR, ANN, DT	$\bullet$ L1, L2 cache accesses and misses, branch mispredictions
[28]	resource allocation	performance	ANN	$\circ$ allocated cache space, off-chip bandwidth $\bullet$ recent cache access hits and misses
[29]	throughput	performance	ANN	$\bullet$ cache miss rates and instruction mix ratios

different hardware resources. Beyond KNN and LR, they address further techniques, e.g., SVM and Decision Trees (DT). The impact of application-specific attributes, such as the processed length of single nucleotide sequences and the taxa size of the input nucleotide datasets, as well as resource-specific attributes, e.g., as CPU speed, amount of memory, speed of memory, on the prediction accuracy is evaluated.

In [24], the authors propose multi-task learning (MTL) formulations to predict and avoid slow running (or straggler) tasks. They formulate the straggler prediction problem as a binary classification problem, and consider system-level counters such as CPU and memory usages as learning features. Further studies on the mapping of OpenCL kernels onto CPUs and GPUs use SVM models [25, 26]. The authors formulate the mapping problem as a classification problem, and devise SVM-based prediction models. These models are trained by using fine-grained static code features (e.g., number and types of instructions) and some runtime parameters extracted from a compiler. These approaches focus on the analysis of each OpenCL kernel program, based on which the most suitable type of processor (CPU or GPU) can be predicted for kernel mapping, w.r.t. given optimization criteria.

In [27], the authors apply machine learning to find out energy-efficient configurations for running multi-threaded workloads on heterogeneous multicore architectures. Machine learning models including Multi-Layer Perceptron (MLP), regression and tree-based classifiers, are built while taking into account fine-grained hardware performance counters information, e.g., cache misses and accesses, branch mispredictions at run-time from a multi-threaded application. These models aim at predicting parameter values such as core type, voltage and frequency for maximizing the energy-efficiency. While comparing the built machine learning models, the authors observed that complex predictors such as MLP achieve higher accuracy compared to simpler regression-based and tree-based classifiers, but they have higher overheads in hardware. In an earlier work [28], ANNs have been used for coordinating the dynamic allocation of shared multiprocessors-on-chip resources. The global resource allocation problem is for-

mulated based on monitored information about the execution of applications. Each ANN takes as input several fine-grain information related to the hardware resources, including L2 cache space, off-chip bandwidth, power budget, the number of read and write hits/misses in the L1 cache. Based on these information the performance of the application is predicted for better allocation decisions. In [29], the authors apply ANN-based machine learning to predict the performance of multiple threads running on heterogeneous cores. The aim is to maximize the throughput. For this purpose, fine-grained system execution information such as L1, L2 and L3 cache miss rates, instruction mix ratios are collected to feed the ANN models.

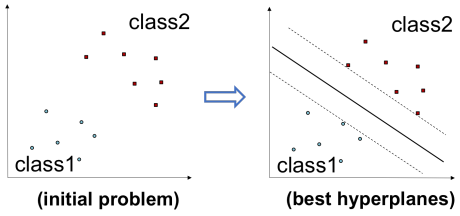
In this paper, we mainly concentrate on the accurate performance prediction for application mapping onto multicore architectures by considering low-cost and coarse-grained input training information, i.e., mapping locations of tasks and data, combined with global performance numbers associated with each mapping instance. To obtain high prediction accuracy, the aforementioned related work require fine-grained information as indicated via the input learning features in Table 1, and thus need to implement some non-trivial module to collect such data at runtime. On the other hand, these studies alleviate the performance prediction problem of mappings by either focusing on task/thread executions on some specific resources such as in [21, 25, 26, 24] without considering the communication aspects, or focusing on the prediction of threads and/or cores numbers or core configurations such as in [23, 22, 27] without investigating the explicit thread/task-core binding solutions. No microarchitecture-dependent information is required in our approach contrarily to approaches such as [29] or [30]. By considering a minimal information, we show how selected machine learning techniques, i.e., SVM, AdaBoost and ANN, can be applied to build relevant performance prediction models useful for mapping decisions in the flow depicted by Figure 1.

3. Selected Supervised Machine Learning

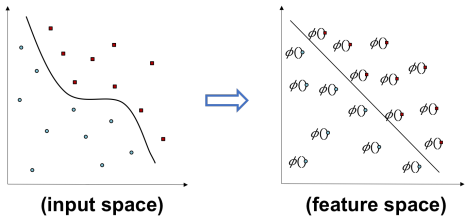
We briefly recall in the next the main principles of the three supervised machine learning techniques selected for our study. The tools used for applying these techniques are briefly presented.

3.1. Classification Techniques: SVM and AdaBoost

The Support Vector Machines (SVM) [8] technique is usually considered a must-try in machine learning approach [31]. Given a set of training examples, each marked as belonging to a class among a number of classes, the aim of SVM is to find the best classification[1] function to distinguish the class members.



(a) **linear** classification



(b) **nonlinear** classification

Figure 2: SVM applied to a 2-class learning problem: in case 2a the best classification function is denoted by the solid line; in case 2b the input space is transformed into a feature space with linearly separated dataset.

Figure 2a shows a two-class learning problem with a linearly separable dataset, and a corresponding linear classification function consisting of a hy-

[1] SVM can be also applied in regression problem, even though it is only used for classification in our work.

perplane that separates the members of the two classes. As there are many such linear hyperplanes, SVM enables to find the best function (e.g., the solid line in Figure 2a, right-hand side) by maximizing the margin between the two classes. Geometrically, this margin corresponds to the shortest distance between the closest data points to a point on the hyperplane. In addition to linear classification, SVMs can also perform a nonlinear classification by using *kernel trick* to deal with data sets that are not linearly separated. This is done by transforming the input space into a high-dimensional feature space in which the data set can be separated linearly as shown in Figure 2b. To perform such transformation, a kernel function denoted by ϕ is required. The most widely used kernel functions are *Radial Basis Function* (RBF), *linear* and *polynomial*. Let x and y be two vectors in the input space, the simplest linear kernel is defined by their inner product plus an optional constant, whereas RBF and degree-d polynomial kernels are respectively defined as:

$$K(x, y) = exp(-\frac{||x - y||^2}{2\sigma^2}) \tag{1}$$

and

$$K(x, y) = (x^{\mathsf{T}} y + c)^d \tag{2}$$

where σ and c are free parameters trading off the influence of higher-order versus lower-order terms.

Since the mapping problem addressed in this paper is a non-linear classification problem, choosing the suitable kernel function ϕ is very important to find the best SVM classification models.

The Adaptive Boosting (AdaBoost) algorithm [9] is one of the most important ensemble methods [32]. Its main idea is to construct a strong learner by combining multiple weak or base learners. It is adaptive in the sense that consequent weak learners are adjusted iteratively in favor of those instances misclassified by previous classifiers.

Given a weak or base learning algorithm and a training set as shown in Figure 3 (left-hand side), where the symbols + and − represent instances that

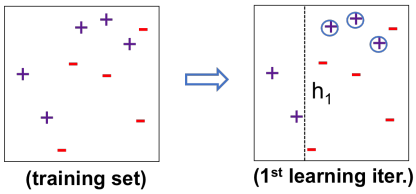


Figure 3: Constructing a strong learner by combining weak learners generated iteratively in AdaBoost.

belong to two different classes, AdaBoost works as follows. First, it assigns equal weights to all the training examples. Let D_i denote the weights distribution at the i^{th} learning round. From the training set and D_1 the algorithm generates a weak learner denoted by h_1 as shown in Figure 3 (right-hand side) by calling the base learning algorithm. Then, the weights of the incorrectly classified instances denoted by circles are increased, and an updated weight distribution D_2 is obtained. From the training set and D_2, AdaBoost generates again another weak learner. This process is repeated for a fixed number of rounds, and the final model is derived by combining the weighted outputs of the previously generated weak learners. The weights of the weak learners are determined during this training process. It has been proven that even when the base learners are weak, as long as the performance of each one is slightly better than random guessing, the final model can converge to a strong learner [33].

3.2. Artificial Neural Networks (ANNs)

We consider the *feed-forward neural networks*, also known as Multi-Layer Perceptron (MLP) [10], consisting of: one input layer of neurons, one output layer of neurons, and one or several hidden layers of neurons. An example of such a network is illustrated in Figure 4. The connections between the neurons of different layers are weighted. The weights of the connections, denoted by w_k, are adapted during the training process. Given an input mapping M_i, the output of the network $o = pred(M_i)$ should match as much as possible the expected value $eval(M_i)$. Once the network is trained enough, it is used as a predictor for unseen mappings.

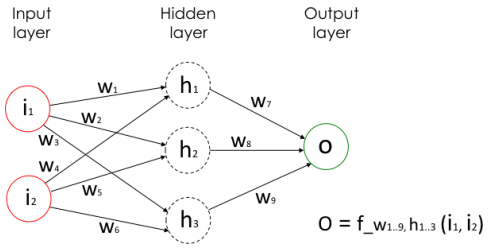


Figure 4: Multi-layer Perceptron with one hidden layer.

The MLP network features interesting approximation properties: any continuous function can be approximated closely by an MLP [34] with a single hidden layer. However, the number of neurons in the hidden layer may be large and cannot be determined algorithmically. To learn a function, an input vector of values is fed to the network through the input layer. The algorithm used to adapt the weights during the training phase is *back-propagation*. The weights are adapted in order to minimize the error between the output value calculated by the network and the actual value of the function computed at that input vector. This learning process is repeated for every input vector. Its outcome, i.e., whether or not the network approximates well the function, is dependent on the initial values of the weights and on the number of the neurons in the hidden layer. To obtain a suitable network, the process needs to be performed multiple times by varying the weights and/or the number of hidden layers and their included neurons until suitable parameter values are found, w.r.t. the expected accuracy of the approximated function.

3.3. Considered Machine Learning Tools

There are several machine learning tools nowadays. Two of them are considered in this work: the `scikit-learn v0.9.1`[2] package and the `Weka v3.8.0` toolset [35]. The former is used to train classification models with SVM and AdaBoost, while the latter is applied for training regression-based prediction models with ANNs.

[2] `http://scikit-learn.org`

For SVM-based classification with the `scikit-learn` package, the main parameters one needs to tune are the following:

- *kernel function*: one can choose among `linear, poly, rbf` and `sigmoid`;
- *gamma*: kernel coefficient for `poly, rbf` and `sigmoid` functions;
- C: penalty parameter of error term.

For AdaBoost-based classification, the tuning parameters in the `scikit-learn` package are as follows:

- *base_estimator*: the base estimator from which the boosted ensemble is built;
- *n_estimators* : the maximum number of estimators;
- *learning rate*: it is used to shrink the contribution of each classifier;
- *algorithm*: either `SAMME.R(default)` or `SAMME`. The former uses the probability estimates to update the additive model, while the latter uses the classifications only. The `SAMME.R` algorithm enables a faster training.

For regression-based prediction with the `Weka v3.8.0` toolset, we consider its associated `MLPRegressor` package: a multilayer perceptron with a single hidden layer. This package exploits the optimization capability provided in `Weka`, by minimizing the given loss function plus a quadratic penalty with the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method. The ANN tuning parameters of the `MLPRegressor`-based prediction are described as follows:

- *number of hidden neurons* (large numbers induce long learning durations);
- *ridge parameter*: used to determine the penalty on the size of the weights;
- *seed value* for initializing the weight values of the networks;
- *activation functions*: Sigmoid or Softplus;
- *loss function*: squared error or approximated absolute error;

- a *tolerance* parameter for the delta values;
- *conjugate gradient descent* (rather than BFGS) for accelerating the training process;
- *parallel calculation* of loss function and gradient when training on multiple CPU cores.

The application of the above machine learning techniques to the case study addressed in Section 6 will consist in finding the parameter values that provide precise-enough performance predictions.

4. Multicore System Design

We present the design concepts used in this study for the description and simulation of multicore systems. These concepts enable to specify applications through a task graph oriented representation (see Section 4.1). Existing application parallelization tools [36] [37], combined with designers' analysis, help to derive such task graphs. Network-on-Chip based multicore system models are used for application mapping and execution with a simulator (see Section 4.2). Finally, the encoding of the resulting mappings is addressed (see Sections 5.1 and 5.2) for performance prediction.

4.1. Application Design Concepts

We define the modeling concepts dedicated to application description. These concepts are inspired by the Amalthea formalism [6], which has been introduced for automotive software design.

Definition 2 (Runnables and labels). *We consider the following notions:*

- *a runnable r is a function representing the smallest unit of code schedulable by an operating system, and associated with non functional attributes, e.g., execution time;*

- *a label l is a symbolic concept representing a memory location, associated a size attribute.*

The value of a non functional attribute of a runnable r can be either a point-wise value $v \in \mathbb{R}$ or an interval of values (lwb, upb), $lwb, upb \in \mathbb{R}$ or a probabilistic distribution. This enables to specify various value approximations. For instance, considering the execution time of a runnable, a point-wise value can be used to capture an average/worst-case/best-case execution time. An interval captures a variation of execution time between worst-case and best-case scenarios, while a probabilistic distribution will describe a probabilistic law characterizing the execution time behavior. The unit of label size is *byte*.

In the sequel, we respectively denote by $\mathcal{R}$ and $\mathcal{L}$ the sets of all runnables and labels. Runnables and labels are combined to build a *task*, which corresponds to an aggregate execution entity.

Definition 3 (Tasks). *A task $t = (R, L, dep, release)$ is a labeled directed graph of runnables such that the set of runnables $R \subseteq \mathcal{R}$ represents the graph vertices; $L \subseteq \mathcal{L}$ is a non-empty set of labels associated with the edges connecting the runnables $r \in R$; $dep \subseteq R \times (L \cup \emptyset) \times R$ defines the edges of the graph; and $release$ is an attribute specifying whether the release mode of task t is either periodic or sporadic or aperiodic, together with the corresponding periodicity value.*

From the above definition of *dep*, the edge connecting two different runnables within a task can be either associated with a label or not: a labeled edge expresses a data communication between connected runnables, while non-labeled edges model precedence between connected runnables.

Example 1. *The task $t = (R, L, dep, release)$ where $R = \{r_0, r_1, r_2, r_3, r_4\}$, $L = \{l_1, l_2\}$, $dep = \{(r_0, l_1, r_2), (r_1, r_4), (r_1, l_2, r_3), (r_2, r_4), (r_4, r_3)\}$ and $release = \langle aperiodic, --\rangle$ represents an aperiodic task, composed of five connected runnables. Here, only two runnable connections correspond to data communications achieved*

through labels l_1 and l_2. The specification of task t is the same as for the task T_4 shown graphically in Figure 5.

Upon the release of a task, all its associated runnables are scheduled for execution. Let us denote by $\mathcal{T}$ the set of all tasks. Tasks are combined together to build applications as described in the next.

Definition 4 (Application). *An application $a = (T, dep)$ is a directed graph of tasks such that $T \subseteq \mathcal{T}$ and $dep \subseteq T \times T$.*

Concretely, applications are graphically described by using Amalthea notations [6], which capture the design concepts defined above.

Example 2. *An application model composed of five tasks with various release modes is illustrated in Figure 5. The periodic task T_0 has a period of $5ms$. It interacts with the periodic tasks T_1 and T_2. Task T_3 denotes a sporadic task with a minimum inter-release interval specified as (lwb, upb). Task T_4 is an aperiodic task with a release mode defined according to a given distribution law. A zoom in this task shows a sub-graph of five runnables $R_{i,i\in 0..4}$. Runnables R_0 and R_2 communicate via the label L_1: R_0 writes L_1 while R_2 reads L_1. The size of L_1 represents the exchanged data volume.*

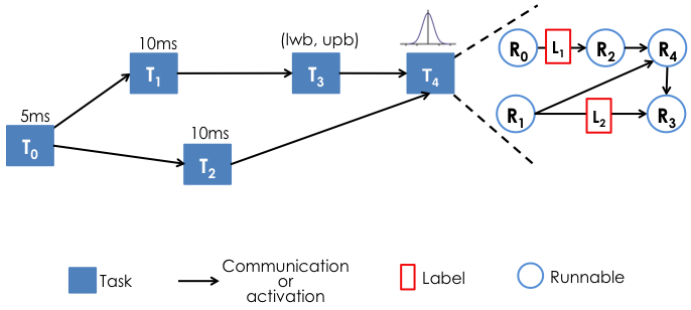


Figure 5: A simple application model in Amalthea

In the remainder of the paper, for the sake of simplicity we will use the notation $X.a$ in order to refer to an attribute a of a concept X. For instance, given a task t, the runnable r_i in task t is denoted by $t.r_i$.

4.2. Application Mapping on Execution Platforms

We consider execution platforms composed of multiple cores that exchange data via a communication interconnect, e.g., a crossbar. Each individual core is composed of a CPU and a local memory. Let $\mathcal{C}$ denote the set of all cores.

Definition 5 (Execution Platform). *An execution platform $p = (C, I)$ is defined as a subset $C \subseteq \mathcal{C}$ of cores, interconnected by an interconnect I as communication infrastructure.*

With the high number of cores in target execution platforms, the chosen communication interconnect is Network-on-Chip (NoC), as it scales better compared to bus and crossbar.

When applications are mapped on a given execution platform, each task (or runnable) is assigned to a core CPU in charge of processing the corresponding functions. Label variables are assigned to memory locations in the cores. When a runnable and its accessing labels are mapped onto different cores, the corresponding communications become remote and require transactions via the NoC. Otherwise, the memory accesses are local and do not incur any NoC transaction.

Definition 6 (Application mapping on execution platform). *Given an application a and an execution platform p, a mapping m of a on p is defined as:*

$$(a.T \times p.C) \equiv_{def} (a.T.R \times p.C) \cup (a.T.L \times p.C) \tag{3}$$

i.e., the runnables $a.T.R$ and labels $a.T.L$ associated with each task T of the application a are mapped onto the cores $p.C$ of the platform p.

Figure 6 depicts a typical scenario where runnables are mapped onto the CPU part of the cores in an execution platform. The labels are mapped onto memory locations within cores. The bottom part of Figure 6 illustrates a multicore platform models where cores communicate with each other via a network interface (NI), connecting them to the NoC. Each core model includes a CPU (dark blue box) and a local memory (red dashed box).

The McSim-TLM-NoC (Manycore platform Simulation tool for NoC-based systems at Transactional Level Modeling) [38] [39] is an Amalthea-based simulator that is used to evaluate mapping scenarios. The multicore architecture considered in this simulator relies on an abstract cache-less core model [40] [41], which supports priority-preemptive runnable execution (and *Round-Robin scheduling* for runnables with the same priority level). The runnables mapping decisions are defined in the *mapping heuristics module* (see Figure 1). An example of mapping consists in allocating tasks that strongly communicate with each other on the same (or closest) cores, in order to reduce the overall communication traffic [40]. Each core in McSim-TLM-NoC is composed of two main units: an execution unit and a communication unit, which deal with their corresponding instructions within the executed runnables. The different cores communicate through either a bus, a crossbar or a mesh-oriented packet-based Network-on-Chip (NoC). In the current work, we use a NoC, where each node in the network includes a core and a local memory. An XY routing algorithm is applied for packet exchanges between nodes. The runtime and energy consumption information computed by McSim-TLM-NoC are estimated on the basis of instruction costs relying on ARM Cortex-A7 and Cortex-A15 CPUs. Further details on the simulator implementation can be found in [41].

McSim-TLM-NoC provides a clean and simple interface allowing to map runnables and labels onto platform resources, through custom mapping algorithms. Once the mapping is defined, the different runnables are scheduled and executed [40]. Contrarily to cycle-accurate simulators such as gem5 [42], McSim-TLM-NoC is fast enough to enable the evaluation of thousands of application mappings in a quite reasonable time. This enables to produce mapping examples usable as training data for performance prediction.

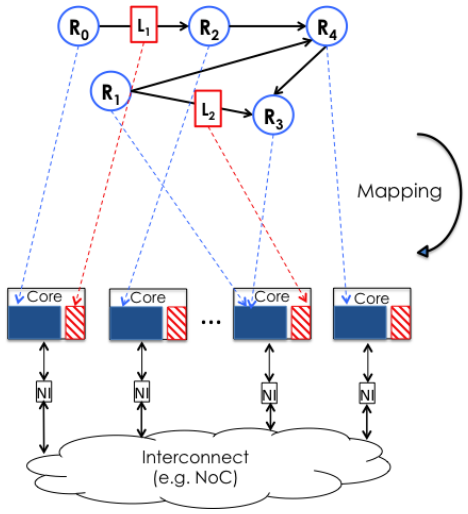


Figure 6: Application mapping on a multicore platform.

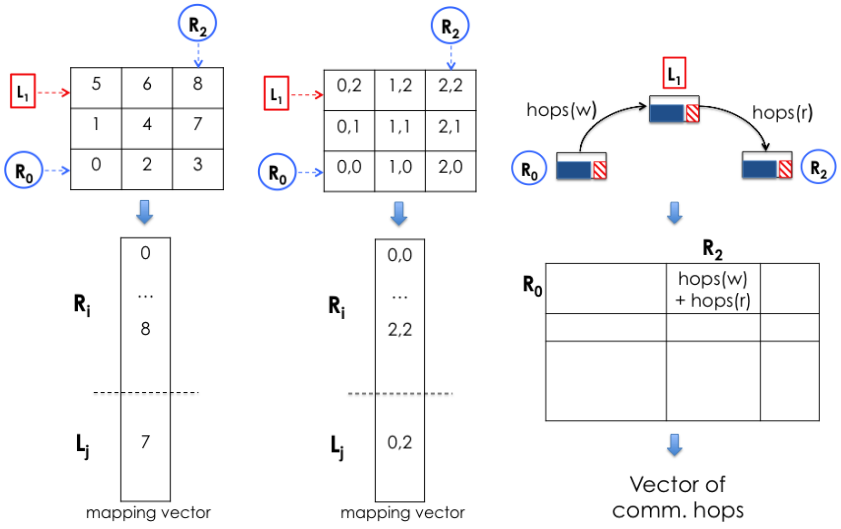
5. Application of Selected Machine Learning Techniques to Mapping Performance Prediction

The effective use of the selected machine learning techniques (see Section 3) to address the mapping performance prediction problem, requires some answers to two crucial questions: i) how to define a relevant mapping encoding for model training? ii) how to adequately assess the quality of the generated prediction models? These questions are addressed in the sequel.

5.1. Mapping Encoding for Training

We discuss three candidate mapping encodings, as illustrated in Figure 7:

- **Encoding 1** (Figure 7a). In this scenario, the vector describing a mapping has as many entries as there are runnables and labels in the model of an application. To build such a vector, the runnable and label identifiers are sorted in an arbitrary order once and for all. The cores of the platform are indexed using integers. Then:
  - each mapping vector component, corresponding to a runnable identifier, is initialized with the index value of the core on which this



(a) Encoding 1 (b) Encoding 2 (c) Encoding 3

Figure 7: Three application mapping encodings.

runnable is mapped. For instance, if the i^{th} component of such a vector V corresponds to a runnable R_i, then the value of $V[i]$ is equal to the index of the core on which R_i is mapped;

- in a similar way, for each label, the corresponding mapping vector component is initialized with the index of the core containing the memory on which the label is mapped.

- **Encoding 2** (Figure 7b). This scenario is similar to the previous one except that now core indexes are not single integers but two integers, corresponding to the Cartesian coordinates of cores within the two-dimensional space inherited from the mesh topology of the considered NoC interconnect. Here, the size of the vector representing the mapping is twice as large as in the first encoding approach.

- **Encoding 3** (Figure 7c). In this scenario, we encode a mapping through a square matrix. The number of columns and the number of rows of the

matrix are equal to the number of runnables in an application. Each row (and column) entry is associated with a runnable identifier.

Let us assume that a runnable R_i writes data to a label L and another runnable R_j reads data from L. The flits (i.e., the elements composing a packet exchanged in a NoC) sent by R_i to L have to perform $h(w)$ hops in the NoC depending on the mapping locations of R_i and L. When R_j reads data from L, the flits traveling from L to R_j perform $h(r)$ hops. Finally, the value at entry (i, j) of the encoding matrix is defined as:

$$(h(w) * n_w) + (h(r) * n_r) \tag{4}$$

where n_w is the number of flits written by R_i on L and n_r is the number of flits read by R_j from L. Finally, the matrix resulting from the encoding is transformed into a vector by putting its columns on top of each other or by aligning its rows next to each other.

Which mapping encoding to select? The first encoding may not render well the similarity or dissimilarity between different mappings. Typically given the scenario shown in Figure 7a, let us consider a first pair of mappings M1 and M2 such that M1 and M2 only differ by the location of one specific runnable. In M1 this runnable is mapped on core 5 (in the matrix shown on top of Figure 7a) while in M2 the runnable is mapped on core 3. The Manhattan distance between the vectors representing M1 and M2 is 2. Now, let us consider mappings M3 and M4 such that in M3 the same runnable is mapped on core 4 and in M4 this runnable is mapped on core 8. The Manhattan distance between the vectors encoding M3 and M4 is 4. By comparing with the Manhattan distance, mappings M1 and M2 appear to be more similar than mappings M3 and M4. However, M3 and M4 are topologically more similar since the locations of the runnable of interest are closer in that case than in the case of M1 and M2: cores 4 and 8 are closer to each other compared to cores 5 and 3. From this observation, the first mapping encoding scenario does not appear appropriate enough. So, we will consider the two other encodings.

The size of vectors in the second encoding is linearly proportional to the number of runnables and labels. In the third encoding, the size of the vector depends quadratically on the number of runnables. This can make the training of learning models more complex. Indeed, real-life applications can feature huge numbers of runnables and labels. Thus, the data needed to successfully train learning models can grow exponentially in the dimension of the input vector. Reducing the size of this vector is necessary to speed-up the training by accelerating the training algorithms and by reducing the size of the required training set of data.

From this remark, we finally select the second encoding scenario for our experiments in this paper, since it provides the best compromise in terms of relevance and tractability in size. Note that this encoding induces some constraint on the reusability of obtained prediction models for different applications. Indeed, the applications must have similar task graph structures, but the attribute values of the task, runnables and labels can vary. This restriction can be lifted however by building the prediction models at runtime, e.g., through an initial training phase during application execution where mappings are encoded and evaluated. Of course, this online learning process can have some cost, especially when achieved on the same execution platform as the application itself.

5.2. Mapping Prediction Model Assessment

The natural way to assess learned predictive models for both classification and regression problems is to calculate the *prediction accuracy*[3], i.e., ratio of correct predictions over total predictions, obtained with trained models on previously unseen test data instances. The higher the accuracy the better the model. F-measure [43] is another widely used metric to evaluate classification models,

[3]The prediction accuracy is different from the *loss* metric (generally a percentage), which is rather computed on training and validation data instances. The validation data set enables to tune the parameters of the prediction model under training phase. The loss can be seen then as a summation of the approximation errors made for predicted versus actual values/classes in the training or validation sets.

especially for imbalanced data. It is the harmonic average of the precision and recall metrics. A high F1-score indicates that the model has low false positives and low false negatives, and is thus able to correctly identify real threats and not disturbed by false alarms.

Another way to assess the relevance of mapping performance prediction could rely on a ranking of considered mappings according to their predicted classes or performance metrics. Let M_i and M_j denote two different mappings; let $eval(M_i)$ and $eval(M_j)$ be respectively their actual metric values; and let $pred(M_i)$ and $pred(M_j)$ denote their respective predicted classes or metric values. No matter the difference between the predicted and the actual classes or metric values of M_i and M_j, if $eval(M_i)$ and $eval(M_j)$ strictly compare similarly as $pred(M_i)$ and $pred(M_j)$, then the predictions become relevant enough to be exploited in the *mapping heuristics module* (see Figure 1). For instance, if $eval(M_i) > eval(M_j)$, then one should have $pred(M_i) > pred(M_j)$. We refer to this relative comparison as mapping metrics *tendency prediction*, i.e., how the predicted classes or performances of mappings "*tend*" to behave relatively to each other, w.r.t. actual metric values.

Definition 7 (Consistent tendency prediction). *Let M_i and M_j denote two mappings; let $eval(M_i)$ and $eval(M_j)$ be respectively their actual metric values, and let $pred(M_i)$ and $pred(M_j)$ denote their respective predicted classes or metric values. A tendency prediction is said to be consistent if the values $pred(M_i)$ and $pred(M_j)$ are comparable in the same way as $eval(M_i)$ and $eval(M_j)$, i.e.:*

$$eval(M_i) \sim eval(M_j) \leftrightarrow pred(M_i) \sim pred(M_j) \tag{5}$$

where the operator $\sim$ belongs to $\{<, =, >\}$.

In general, when the prediction accuracy of a trained model is high, the tendency will be very consistently predicted. However, the inverse is not true. Thus, prediction accuracy is not necessary the most suitable assessment criterion for our learning problem. Instead, we introduce a simpler yet adequate measure relying on tendency prediction.

Definition 8 (Percentage of successful tendency prediction – PSTP). *Given a reference set T of testing mapping pairs, we define the percentage of successful tendency prediction (PSTP) as the percentage of mapping pairs $\langle M_i, M_j \rangle \in T$ that satisfies the formula (5).*

Accurate prediction models are expected to provide very high PSTP values. In practice, it is difficult to reach a maximum prediction accuracy, especially with regression techniques, because of the approximations applied for value prediction. For instance, given two different application mappings M_i and M_j, let us consider $eval(M_i)$ and $eval(M_j)$ are close values when executed on an actual platform. The comparison of their predicted values, $pred(M_i)$ and $pred(M_j)$, according to PSTP will be consistent only if the prediction accuracy is high enough to distinguish how they compare. However, when $eval(M_i)$ and $eval(M_j)$ are quite different, the comparison of $pred(M_i)$ and $pred(M_j)$ according to PSTP has higher chance to be consistent, even without a moderate prediction accuracy.

To assess the quality of built prediction models, it is worth evaluating PSTP on pairs of mappings $\langle M_i, M_j \rangle$ whose actual performance values differ by $\Delta\%$ (where $\Delta \in \mathbb{R}^+$). The idea behind this filtering of mapping pairs is to eliminate test cases for which the performance comparison is highly sensitive to the prediction accuracy. We thus define such Δ-filter PSTP measure as follows:

Definition 9 (Δ-filter PSTP). *Given a reference test set T of mapping pairs whose actual values differ by $\Delta\%$ (where $\Delta \in \mathbb{R}^+$), we define Δ-filter PSTP over T as the percentage of mapping pairs $\langle M_i, M_j \rangle \in T$ that satisfies the formula (5).*

Note that for classification techniques, given a mapping pair $\langle M_i, M_j \rangle$, their respective predicted classes $pred(M_i)$ and $pred(M_j)$ are, instead of real numbers for regression techniques, class labels representing sub-domains of performance values. To make them directly comparable as real numbers, we encode class labels as natural numbers $\lambda \in \mathbb{N}$ in a way that reflects the greater than/less than/equal to relationships for the sub-domains derived from the domain of performance values.

Given a mapping pair $\langle M_i, M_j \rangle$ whose actual performance values M_i and M_j differ by $\Delta\%$, let us assume that the number of target classes enables to assign M_i and M_j into different classes. If both mappings are, however, classified into the same class, representing the same sub-domain of performance values, it then indicates that the classifier is not accurate enough to distinguish them. We refer to such predictions as **unknown** tendency predictions, characterized as follows:

Definition 10 (Percentage of unknown tendency prediction – PUTP). *Given a reference test set T of mapping pairs to be classified, we define the percentage of unknown tendency prediction (PUTP) as the percentage of mapping pairs $\{M_i, M_j\} \in T$ that satisfy:*

$$eval(M_i) \sim eval(M_j) \rightarrow pred(M_i) = pred(M_j) \tag{6}$$

where the operator $\sim$ belongs to $\{<, >\}$.

Similarly to Δ-filter PSTP, the Δ-filter PUTP for classification is defined as follows:

Definition 11 (Δ-filter PUTP.). *Given a reference test set T of mapping pairs whose actual values differ by $\Delta\%$ (where $\Delta \in \mathbb{R}^+$), we define the Δ-filter PUTP over P as the percentage of mapping pairs $\langle M_i, M_j \rangle \in T$ that satisfies the formula (6).*

To summarize, the outcome of the classification of two different mappings M_i and M_j falls within one of the following cases:

- *correct* prediction: when the predicted classes are ranked consistently w.r.t. the actual mapping performances values $eval(M_i)$ and $eval(M_j)$;
- *wrong* prediction: when the predicted classes $pred(M_i)$ and $pred(M_j)$ are ranked in an opposite way w.r.t. the actual mapping performances values $eval(M_i)$ and $eval(M_j)$;
- *unknown* prediction (only for classification): when the predicted classes $pred(M_i)$ and $pred(M_j)$ are identical while they should be distinct w.r.t. the actual mapping performances values $eval(M_i)$ and $eval(M_j)$;

In our experiments, we will mainly use PSTP as accuracy assessment metric for defined prediction models. The coverage of this assessment on the testing mapping set will be evaluated with PUTP in applied classification techniques.

To formulate the mapping performance prediction problem as a classification problem, we partition the generated simulation data into a number of classes according to the metric value ranges. For execution time, it is done by taking the minimal and maximal execution times (denoted by $minExec$ and $maxExec$) as the possible range of execution times $[minExec, maxExec]$, and by dividing this range into sub-ranges of same length. The length is computed as follows:

$$length = (maxExec - minExec)/N \tag{7}$$

where N denotes a selected number classes. As a result, we obtain N intervals, as follows:

$$[minExec, minExec + length], ..., [maxExec - length, MaxExec] \tag{8}$$

denoted by $I_1, ..., I_N$. The data samples can thus be classified into N classes accordingly. An instance is classified in class C_i, if its execution time for instance, falls into the interval I_i. In this way, instead of predicting the execution time, we predict the class or interval a given mapping falls into. The larger the N gets, the more informative the prediction result gets.

Finally, the model training, we partition the *working mapping set* as follows: 65% of the mappings are used for training and the remaining 35% are used as unseen data for testing the quality of the prediction models. This partitioning is compatible with common practices in machine learning – e.g., see the partitioning suggested in `Weka` [35].

6. Comparison of Machine Learning Techniques on a Case Study

We consider an automotive application case study [12] in order to evaluate the quality of the prediction models derived using the selected machine learning techniques: SVM, AdaBoost and ANN. The application, referred to as DemoCar, corresponds to an engine control system, provided by Robert Bosch GmbH,

within the DreamCloud European FP7 project. As briefly mentioned in the introductory section, comparing the quality of our results w.r.t. existing mapping heuristics [1] is beyond the scope of this paper. Instead, we focus on the quality of performance value prediction, which is used by the *mapping heuristics module* to assess candidate mappings (see Figure 1).

The inputs of DemoCar application are typical in automobiles, e.g., engine speed, temperature, battery voltage. Its outputs are the triggered cylinder number, the ignition time and the desired throttle position. In total, there are 10 input message sources and 4 output message sinks. The considered Amalthea model of DemoCar is composed of 43 runnables and 71 labels. Out of these runnables, 22 runnables operate at high activation rate, 4 runnables operate at low activation rate, and 17 runnables get activated aperiodically upon some event occurrences.

In the following, we discuss the generation of DemoCar mapping instances for training and testing the target prediction models. Classification techniques are first presented. Then, ANNs are applied. Finally, we discuss the effectiveness and efficiency of the three techniques.

6.1. Experimental Setup

Generation of the DemoCar Application Mapping Instances. The mappings of DemoCar feature a multicore execution platform composed of 6 cores with a 2x3-mesh NoC architecture for communication. Here, each core model in McSim-TLM-NoC features an ARM Cortex-A15 CPU running at 1GHz. Current automotive on-chip multicore systems do not exceed this core count. Note that even though a homogeneous multicore execution platform is considered here, our proposal can also deal with heterogeneity by associating tasks/runnables with instruction costs pertaining to different target computing elements, in McSim-TLM-NoC. This would probably result in different performance/energy outcomes in the resulting mapping vectors. Then, the exact same training and prediction methods remain applicable, as illustrated in the homogeneous design considered here.

The mapping of labels is fixed and identical in all mappings generated in this study. Only the mapping of runnables on core CPUs is variable. This choice has been made for the sake of simplicity as we can straightforwardly evaluate the impact of changes in runnable mappings. Even though relevant, taking into account possible changes in label mappings would make the exploration space much larger. Given a number R of runnables to be mapped on a number C of cores, there are C^R possible mappings of the runnables on these cores. For DemoCar, this corresponds to 6^{43}, which is a very large exploration space. Within this space, we decided to compute with McSim-TLM-NoC simulator a maximum set of 30K mappings generated[4] randomly according to a uniform distribution (for a relevant coverage of the possible mapping space). We checked there is redundant and no outlier mapping instance within this set of mappings. Each mapping instance is associated with its corresponding execution time and energy consumption computed with the simulator.

Four different working mapping sets are considered for the training with all three supervised learning techniques: 3K, 5K, 10K and 30K mapping instances. This enables to explore how the quality of the prediction evolves with the size of working mapping sets.

Prediction Model Evaluation Scenarios. The PSPT measure introduced previously is used for assessing the generated prediction models. For this purpose, we consider the set $P_{(M_i,M_j)}$ of all possible pairs of mappings without redundancy resulting from the testing subset mappings. Then, we evaluate the following cases:

- **case-0**: PSTP over the set $P_{(M_i,M_j)}$. Given n the number of mappings in the test subset, the number of all possible pairs of mappings in $P_{(M_i,M_j)}$ is defined as: $(n * (n-1))/2$;

- **case-1**: Δ-filter PSTP over the set $P_{(M_i,M_j)}$, where $\Delta = 5$ (i.e., only pairs

[4] All mapping data sets used in the current study are available at `https://seafile.lirmm.fr/d/ca11a19a75c44013988f`.

of mappings such that the actual metric values of one mapping are 5% different from those of the other mapping);

- **case-2**: Δ-filter PSTP over the set $P_{(M_i,M_j)}$, where $\Delta = 10$;
- **case-3**: Δ-filter PSTP over the set $P_{(M_i,M_j)}$, where $\Delta = 20$.

For classification-based approaches, we also use the PUTP measure (see Definition 10 in Section 5) to evaluate the coverage of PSTP on the testing set. In the reported experiments, we consider different numbers of classes $N \in \{3, 9, 81, 512\}$. Figure 8 shows the distributions of the 30K mapping instances used later on, according to their induced execution times, and w.r.t. different numbers of classes. We notice that the resulting distributions are moderately unequal and may lead to *imbalanced data* problem [44], sometimes faced in classification problems. While the data set could be transformed to have more balanced distribution (e.g., by collecting more data or by applying sampling methods), we decide to keep the set unchanged. Advanced machine learning algorithms can deal with imbalanced data [44]. This is typically the case of Decision Trees.

Our experiments are performed on an Intel Core i5-6600 host operating at 3.4GHz. Most of the prediction model evaluations shown in the next sections concern execution time. Even though not reported, similar results are observed when focusing on energy. The main reason is that the evaluations obtained with McSim-TLM-NoC are often proportional for both metrics.

6.2. SVM-based Prediction Modeling

Figure 9 reports the performance of SVM models w.r.t. different numbers of classes. These results rely on the most favorable values selected for the *kernel function* function, *gamma* and C values: `rbf`, 1 and 1000 respectively. In particular, the kernel function value domain has been exhaustively explored. For each function, in average 30 combinations of *gamma* and C values are explored. The duration required to train every SVM model varies between less than 1 second to 13 minutes.

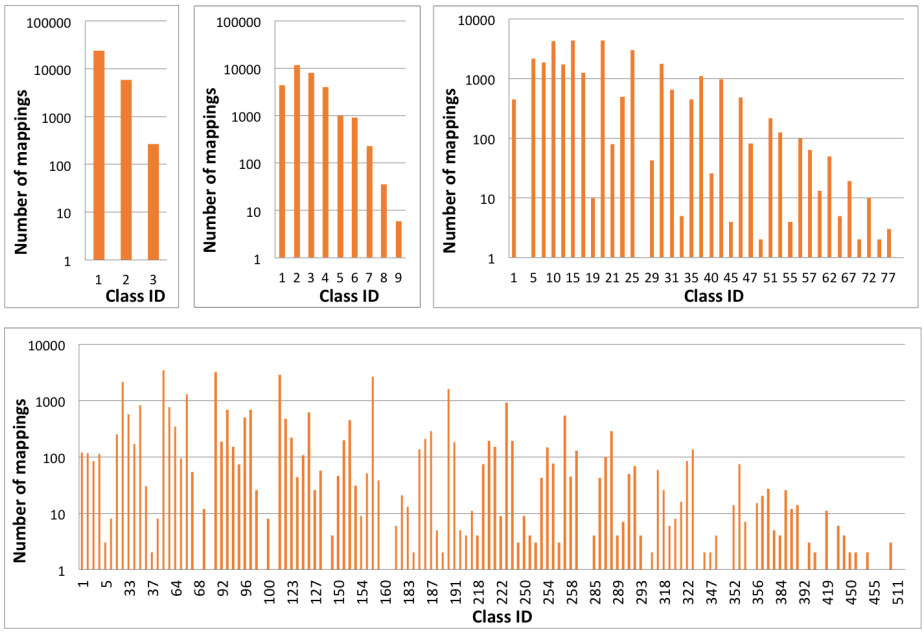
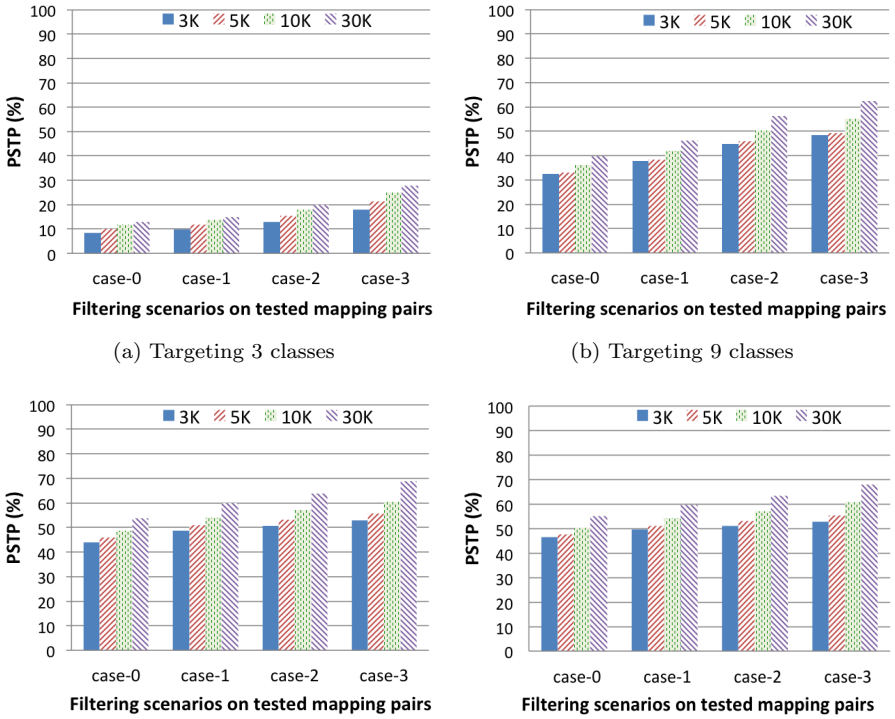


Figure 8: Distributions of 30K mapping instances w.r.t. execution time ranges. The Y-axis denotes the number of mapping instances (in log scale) corresponding to each class ID shown on the X-axis.

Let us consider Figure 9c. We obtain for **case-0** a maximum PSTP value of 53.6% while the minimum is 43.9%. It means that with a 30K working mapping set, the best SVM prediction model would enable correct mapping pair comparisons only on 53.6% of evaluated pairs. For **case-1**, **case-2** and **case-3**, after filtering the tested mapping pairs based on the difference in their execution times, we observe better results (i.e., Δ-filter PSTP where $\Delta = 5\%$, 10%, 20%). This respectively leads to 59.76%, 63.9% and 68.97% of correct mapping comparison. More generally, for each target number of classes shown in Figure 9, we observe the same trend: given a working mapping data set (i.e., 3K, 5K, 10K and 30K mapping sets), the PSTP gets better as 1) the number of training samples grows, and 2) the filtering ratio of the testing set of mapping pairs increases.

In addition, the PSTP values get better when the number of classes is refined.



(a) Targeting 3 classes

(b) Targeting 9 classes

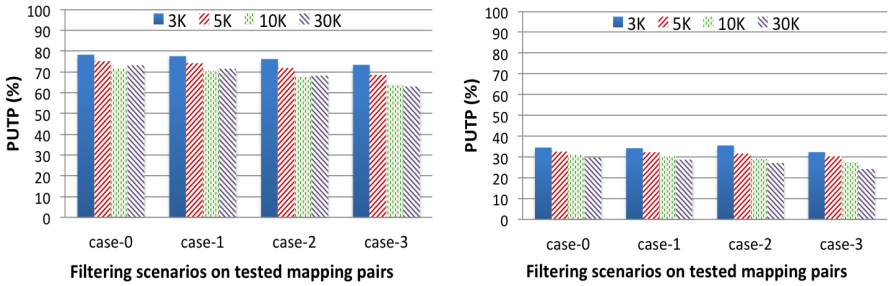
(c) Targeting 81 classes

(d) Targeting 512 classes

Figure 9: Performance of SVM models according to PSTP criterion w.r.t. different target classes.

The reason behind is that with a low number of classes, more mapping pairs tend to be classified into the same class. Then, the prediction model would not be able to predict their related tendency by comparing their execution time. This is characterized through the PUTP measures reported in Figure 10 accordingly.

For instance, the PUTP value obtained in Figure 10a explains why the PSTP's shown in Figure 9a are quite low. Let us take the best PSTP in Figure 9a, which is 27.9% and its corresponding PUTP in Figure 10a, which is 63.0%. It means that only 37.0% of tested mapping pairs have been classified into different classes, and among such pairs whose tendency can be compared, 75.4% were predicted correctly. However, a predictor that cannot compare mappings in most of the cases, even when providing good predictions whenever possible,



(a) Targeting 3 classes

(b) Targeting 9 classes

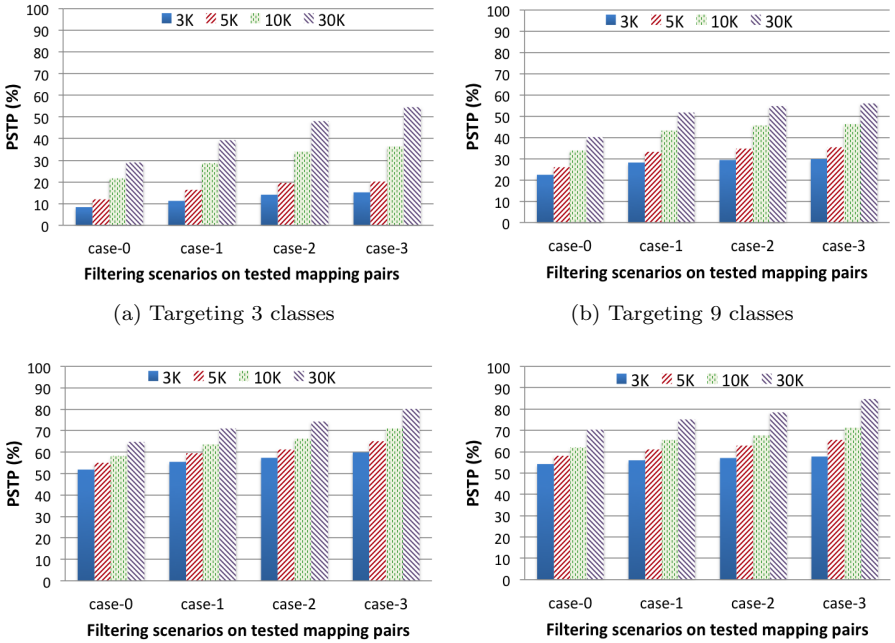
(c) Targeting 81 classes

(d) Targeting 512 classes

Figure 10: The percentage of unknown tendency predictions for SVM w.r.t. Figure 9.

is not preferable. Better predictions can be actually obtained by increasing the number of classes from 3 to 81, as observed in Figures 9 and 10: the PUTP decreases fast while the PSTP becomes better. When further refining the number of classes from 81 to 512, the PUTP decreases very slowly, resulting in a poor evolution of the PSTP values (decrease from 69.0% to 68.0%). This indicates that the benefit from class refinement holds up to certain partitioning granularity.

On the other hand, one possible reason behind the modest correct prediction scores obtained above with SVM technique can relate to the aforementioned possible data imbalance of the mapping instance sets (see Figure 8). In the next, experiments, we apply the AdaBoost algorithm, which includes Decision Trees as a weak learner, to check whether the correct prediction scores can be improved.



(a) Targeting 3 classes

(b) Targeting 9 classes

(c) Targeting 81 classes

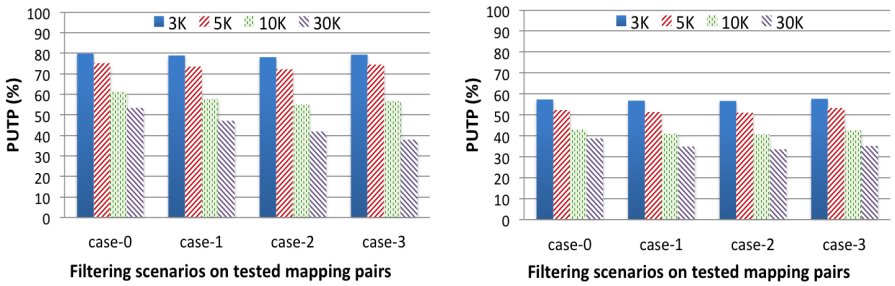
(d) Targeting 512 classes

Figure 11: Performance of AdaBoost models according to PSTP criterion w.r.t. different target classes.

6.3. AdaBoost-based Prediction Modeling

To anticipate any imbalanced data issue as discussed in Section 6.1, we have selected Decision Trees as *base_estimator*. Different combinations of the other parameter values are then explored, and the best values are selected from around 100 explored combinations. Figure 11 reports the obtained prediction performance scores w.r.t. different numbers of classes, i.e., 3, 9, 81 and 512. These results are obtained by selecting 600 for *n_estimators*, 0.4 for *learning rate*, and `SAMME` as *algorithm*. The duration required to train the AdaBoost models on the different training sets varies between 6 to 93 seconds.

Comparing the prediction performance values of SVM and AdaBoost, we observe that the former outperforms the latter only when targeting 9 classes with a maximum PSTP of 62.6% versus 56.1%. However, when targeting 81



(a) Targeting 3 classes

(b) Targeting 9 classes

(c) Targeting 81 classes

(d) Targeting 512 classes

Figure 12: The percentage of unknown tendency predictions w.r.t. Figure 11.

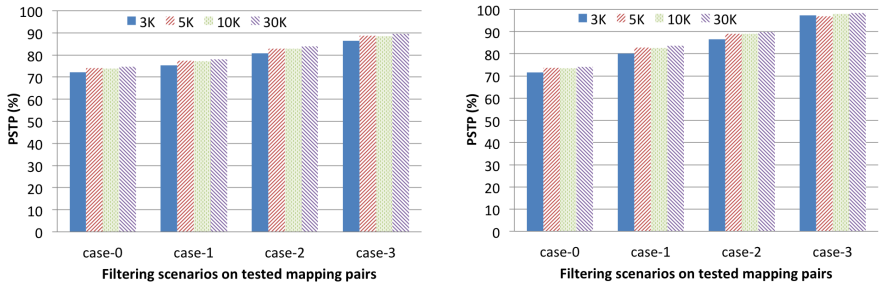
and 512 classes, AdaBoost models achieves 80.1% and 84.8% PSTP scores respectively. In the same time, with AdaBoost a prediction performance always keeps improving as the number of classes increases, in contrast to SVM beyond 81 classes (see Figure 9). To some extent, this indicates that the AdaBoost models distinguish more accurately different mapping instances.

This is confirmed in Figure 12, which reports the PUTP scores for AdaBoost experiments. Here, the PUTP keeps decreasing as the number of classes grows, and drops to less than 1% with 512 classes. This result is quite promising, in particular when considering that the training time is less than 2 minutes! Finally, another interesting observation here is that the PUTP also remains stable w.r.t. different working mapping sets.

6.4. ANN-based Prediction Modeling by Regression

We apply now a regression technique combined with MLP instead of classification, to the same mapping performance prediction problem.

From the explored parameter values, only the most promising ones are reported here. Four neurons are considered within the single hidden layer in selected ANN models. Depending on the size of the considered working mapping sets, the other parameters of the ANN models vary as follows: the ridge parameter, the seed and the tolerance parameter are respectively 13.1, 34 and 10^{-3} for 3K mappings; 13.1 (and 10 for energy prediction), 34 and 10^{-4} for 5K mappings; 0.21 (and 10 for energy prediction), 185 and 10^{-7} for 10K mappings, and 0.03, 185 and 10^{-7} for 30K mappings. The obtained prediction performance scores are depicted in Figure 13. Here, in addition to execution time, we also report the prediction of energy consumption. The training durations required for building the corresponding prediction models varies from 5 seconds to 4 minutes (note that during the initial ANN parameter exploration, some settings took even more than a day to complete, without giving better scores).



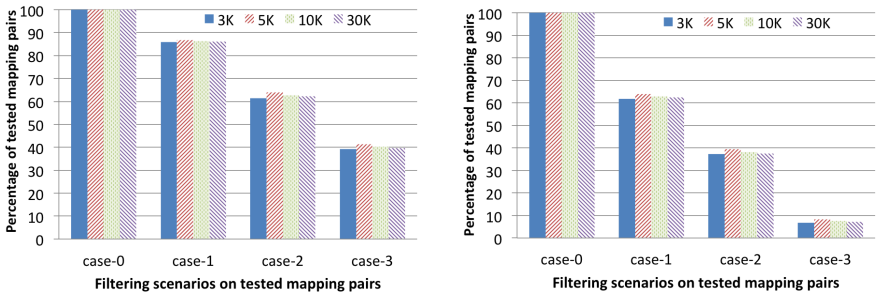
(a) Execution time prediction

(b) Energy prediction

Figure 13: Performance of ANNs according to PSTP criterion, w.r.t. different metrics.

In Figure 13a, we obtain for **case-0** a maximum PSTP of about 74.6% while the minimum PSTP value is 72.23%. In other words, it means that in the best case (i.e., 30K mappings) the obtained prediction model enabled a correct comparison for 74.6% of evaluated mapping pairs. In **case-1**, after filtering the set of mapping pairs, there remain around 85.87% of this set. The PSTP

on this reduced set of mapping pairs yields a better score that reaches up to 77.98%, which is slightly better than previously. By increasing the filtering of the set of mapping pairs, respectively in **case-2** and **case-3**, we observe better results, leading respectively to 83.88% and 89.47% of correct mapping tendency prediction.



(a) Execution time prediction

(b) Energy prediction

Figure 14: Percentage of tested mapping pairs with ANN-based prediction for **case-0**, **case-1**, **case-2** and **case-3**

The prediction performance scores for energy consumption (see Figure 13b) generally follow similar trends compared to execution time. In particular, the very high score observed in **case-3** results from the low number of tested mapping pairs after the Δ-filtering. As a matter of fact, the variation in the energy values of the generated mappings is not as large as for the corresponding execution time. Figure 14 shows the percentage of tested mapping pairs with ANN-based prediction for **case-0**, **case-1**, **case-2** and **case-3**, as a consequence of the Δ-filter PSTP assessment.

7. Gained Insights and Discussion

The different experiments presented above show how classification and regression can be used to deal with the prediction of performances multi-task application mapping on multicore architectures. First of all, despite the potential complexity of the addressed problem, the results obtained especially with the AdaBoost classification and ANN regression models are promising.

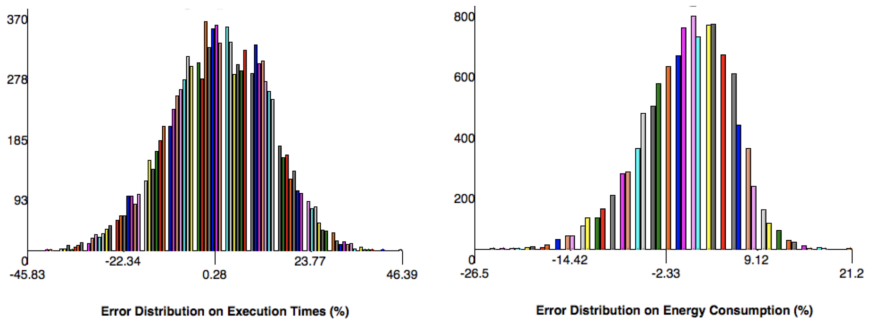
On the accuracy of evaluated techniques. Among the two evaluated classification techniques, AdaBoost provides better results than SVM. Despite the high success of the latter in literature, it seems that the diversity of learners combined by the former is beneficial when facing typical situations such as data imbalance, which is more tractable with Decision Trees supported in AdaBoost. On the other hand, the ANN-based regression technique provides the most accurate prediction models in terms of PSTP score.

In order to confirm the above observations about the three evaluated machine learning techniques, we carried out similar experiments with a different application, executed on a 2x3-mesh multicore architecture. This application, referred to as *light-weight DemoCar*, is composed of 18 periodic runnables and 61 labels [12]. We obtained similar trends as for the case study detailed in Section 6. While all these results are obtained on a 2x3-mesh multicore architecture, we still expect similar trends when comparing the three techniques for architectures comprising more cores. Nevertheless, their corresponding training costs may increase as there would be more possible mapping vector configurations to be taken into account.

Now, when focusing on the prediction errors about both execution time and energy values with ANNs, we obtain the distributions depicted in Figures 15a and 15b. Their respective mean values are 1.46% and 0.3%, while the standard deviations are 12.35 and 5.72. The number of mappings with an error less than 20% accounts for 90.0% and 99.8% of tested mapping sets w.r.t. execution time and energy consumption respectively. This makes the built performance predictors relevant enough for a meaningful mapping comparison.

On the implications about models integration in dynamic resource allocation. Our study on mapping performance prediction is motivated by the dynamic resource allocation flow illustrated in Figure 1. Here, the *mapping heuristics module* is responsible of taking efficient resource allocation decisions at runtime for enhanced energy-efficiency. For this purpose, it exploits mapping performance estimations or prediction to select the best resource allocation de-

cisions.



(a) Execution time (b) Energy

Figure 15: Prediction errors distribution for ANN-based generated models. The Y-axis denotes the number of mapping instances within the different error ranges reported on the X-axis.

In embedded real-time systems, the computing and memory resources are generally limited compared to high-performance or cloud computing systems. Therefore, it is difficult to envision a mapping design space exploration at run-time in embedded real-time systems as it will induce an overhead on the actually executed workload. An alternative pragmatic approach would pre-evaluate different mapping options off-line, which could be leveraged afterwards at runtime. For more effectiveness, one should make sure to cover *a priori* all relevant design options. This is not easy to guarantee. An alternative solution, as promoted in the DreamCloud European project, is to consider fast performance estimation tools such as the McSim-TLM-NoC [40] or the Interval Algebra simulator [45]. The current work opens an opportunity for an aggressive mitigation of the overhead related to the on-demand evaluation of mappings with these tools. For instance, the size and response time (for estimating the performance of a mapping) of the ANN-based prediction model defined for the DemoCar application are respectively about a few tens of kilobytes and microseconds (See Table 2). The size and response time of the Interval Algebra simulator are respectively about a few megabytes and milliseconds, while they are about a few seconds and megabytes respectively for McSim-TLM-NoC. AdaBoost is more costly than

Interval Algebra in both response time and model size. For applications with high reactivity constraints, the ANN-based prediction model appears then as the most preferable.

Table 2: Prediction models versus simulator for mapping performance estimation.

	Response time (μs)	Implement. size (KB)
Interval Algebra simulator [45]	2×10^3	4.13×10^3
McSim-TLM-NoC simulator [40]	1.2×10^6	440
ANN prediction model	**63**	**37**
AdaBoost prediction model	2.4×10^3	1.0×10^4

On the requirements about the solution to the problem addressed in this paper (see Definition 1). The AdaBoost and ANN prediction models can meet the *accuracy* requirement specified earlier in the problem definition, with their respective Δ-filter PSTP scores of 84.8% and 89.05%, when $\Delta = 20$. Concretely, these scores make the associated prediction models capable of identifying, when they exist, candidate mappings that can improve by 20%, e.g., the execution time, w.r.t. a reference mapping. While the above PSTP scores can be considered as reasonable enough for soft real-time automotive applications, higher scores would be however necessary for hard real-time tasks in order to make sure they meet their timing requirements.

On the other hand, the mapping instances used to train the built prediction models are simple enough to be extractable in a costless manner from system executions. Only information about task/data allocation on target cores and memories, together with the induced global performance numbers, are required. This is easily captured via the proposed mapping encodings for fast learning, confirming that our approach favors the *feasibility* requirement.

Finally, the *responsiveness* requirement is met by the selected prediction models. For instance, the average performance prediction time for a mapping is 63 μs on the desktop machine used to carry out the previous experiments, which is quite reasonable.

8. Conclusions and Perspectives

In this paper, we applied machine learning to deal with the performance and energy consumption prediction of applications mapped onto multicore platforms. Our solution relies on simple coarse-grained information, i.e., the mapping coordinates of application tasks, and thus avoids intrusion into a system to obtain training parameters. Two supervised machine learning approaches are investigated: classification based on SVM and AdaBoost, and regression based on ANNs. They have been experimented on an automotive application case study to evaluate their efficiency and effectiveness. The results show that, under some conditions, AdaBoost and ANNs can achieve very promising prediction accuracy with up to 84.8% and 89.05% respectively, which confirms the effectiveness of these two models for learning the multicore system behaviors.

In the future, we would like to deepen our current proposal with methods enabling to overcome the possible learning scalability issue while enhancing the current prediction scores. One possible idea is to enrich the mapping encoding with more information about system characteristics. This could help the machine learning models to better learn the system behavior. For instance, making explicit the data dependency information between runnables or the number of NoC traversal hops may contribute to a better performance prediction. This enhancement may come at the cost of large size input data for networks as there will be additional information to encode. and huge mapping encoding vectors could be difficultly tractable. Complementary techniques such as unsupervised machine learning (e.g., feature or attribute selection, which enables to keep only the most relevant features w.r.t. the learning problem) could be considered to mitigate this possible risk.

Acknowledgements

This work has been supported by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project, and

by the Chinese NSFC under grant 61502140. It has been initiated within the European Community's Seventh Framework Programme (FP7/2007-2013) under the DreamCloud project, grant agreement no. 611411. The authors would like to thank Roman Ursu and Manuel Selva who participated in earlier discussions about the current work.

References

[1] A. K. Singh, M. Shafique, A. Kumar, J. Henkel, Mapping on multi/many-core systems: Survey of current and emerging trends, in: Design Automation Conference, 2013, pp. 1:1–1:10.

[2] A. Jantsch, N. D. Dutt, A. M. Rahmani, Self-awareness in systems on chip - A survey, IEEE Design & Test 34 (6) (2017) 8–26.

[3] Y. LeCun, Y. Bengio, G. E. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[4] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, A. Cristal, M. Valero, A general guide to applying machine learning to computer architecture, Supercomputing Frontiers and Innovations 5 (1).

[5] Z. Wang, M. O'Boyle, Machine learning in compiler optimisation, CoRR abs/1805.03441.

[6] AMALTHEA Project Consortium, Amalthea - An Open Platform Project for Embedded Multicore Systems, `http://amalthea-project.org/` (2015).

[7] L. S. Indrusiak, P. Dziurzanski, A. K. Singh, Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing, River Publishers Series in Information Science and Technology, River Publishers, 2016.

[8] C. Cortes, V. Vapnik, Support-vector networks, Machine Learning 20 (3) (1995) 273–297.

[9] Y. Freund, R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1) (1997) 119–139.

[10] C. Fyfe, Artificial neural networks, in: B. Gabrys, K. Leiviskä, J. Strackeljan (Eds.), Do Smart Adaptive Systems Exist?, Vol. 173 of Studies in Fuzziness and Soft Computing, Springer Berlin Heidelberg, 2005, pp. 57–79.

[11] S. B. Kotsiantis, Supervised machine learning: A review of classification techniques, Informatica (Slovenia) 31 (3) (2007) 249–268.

[12] A. Gamatié, R. Ursu, M. Selva, G. Sassatelli, Performance prediction of application mapping in manycore systems with artificial neural networks, in: MCSoC, IEEE Computer Society, 2016, pp. 185–192.

[13] T. Stefanov, A. Pimentel, H. Nikolov, Daedalus: System-level design methodology for streaming multiprocessor embedded systems on chips, Handbook of Hardware/Software Codesign (2017) 1–36.

[14] X. An, A. Gamatié, É. Rutten, High-level design space exploration for adaptive applications on multiprocessor systems-on-chip, Journal of Systems Architecture - Embedded Systems Design 61 (3-4) (2015) 172–184.

[15] X. Y. Zhu, M. Geilen, T. Basten, S. Stuijk, Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (6) (2016) 905–918.

[16] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, J.-L. Dekeyser, A model-driven design framework for massively parallel embedded systems, ACM Trans. Embed. Comput. Syst. 10 (4) (2011) 39:1–39:36. doi:10.1145/2043662.2043663.
URL http://doi.acm.org/10.1145/2043662.2043663

[17] A. K. Singh, C. Leech, B. K. Reddy, B. M. Al-Hashimi, G. V. Merrett, Learning-based run-time power and energy management of multi/many-core systems: Current and future trends, J. Low Power Electronics 13 (3) (2017) 310–325.

[18] D. Biswas, V. Balagopal, R. Shafik, B. M. Al-Hashimi, G. V. Merrett, Machine learning for run-time energy optimisation in many-core systems, in: Proceedings of the Conference on Design, Automation & Test in Europe, DATE '17, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2017, pp. 1592–1596.
URL http://dl.acm.org/citation.cfm?id=3130379.3130749

[19] A. Irpan, Deep reinforcement learning doesn't work yet, https://www.alexirpan.com/2018/02/14/rl-hard.html (2018).

[20] Y. Zhang, M. A. Laurenzano, J. Mars, L. Tang, Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47, IEEE Computer Society, Washington, DC, USA, 2014, pp. 406–418. doi:10.1109/MICRO.2014.53.
URL http://dx.doi.org/10.1109/MICRO.2014.53

[21] S. Sankaran, Predictive modeling based power estimation for embedded multicore systems, in: Proceedings of the ACM International Conference on Computing Frontiers, CF '16, ACM, New York, NY, USA, 2016, pp. 370–375. doi:10.1145/2903150.2911714.
URL http://doi.acm.org/10.1145/2903150.2911714

[22] P.-J. Micolet, A. Smith, C. Dubach, A machine learning approach to mapping streaming workloads to dynamic multicore processors, SIGPLAN Not. 51 (5) (2016) 113–122. doi:10.1145/2980930.2907951.
URL http://doi.acm.org/10.1145/2980930.2907951

[23] A. Matsunaga, J. A. B. Fortes, On the use of machine learning to predict the time and resources consumed by applications, in: Proceedings of the

2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 495–504. doi:10.1109/CCGRID.2010.98.
URL https://doi.org/10.1109/CCGRID.2010.98

[24] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, R. Katz, Multi-task learning for straggler avoiding predictive job scheduling, J. Mach. Learn. Res. 17 (1) (2016) 3692–3728.
URL http://dl.acm.org/citation.cfm?id=2946645.3007059

[25] Y. Wen, Z. Wang, M. F. P. O'Boyle, Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms, in: 21st International Conference on High Performance Computing (HiPC), 2014, pp. 1–10.

[26] B. Taylor, V. S. Marco, Z. Wang, Adaptive optimization for opencl programs on embedded heterogeneous systems, in: Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2017, pp. 11–20.

[27] H. Sayadi, N. Patel, A. Sasan, H. Homayoun, Machine learning-based approaches for energy-efficiency prediction and scheduling in composite cores architectures, in: 2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017, IEEE Computer Society, 2017, pp. 129–136. doi:10.1109/ICCD.2017.28.
URL https://doi.org/10.1109/ICCD.2017.28

[28] R. Bitirgen, E. Ipek, J. F. Martinez, Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach, in: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, IEEE Computer Society, Washington, DC, USA, 2008, pp. 318–329.

[29] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, A. Cristal, A machine learning approach for performance prediction and

scheduling on heterogeneous cpus, in: 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2017, pp. 121–128.

[30] C. V. Li, V. Petrucci, D. Mossé, Exploring machine learning for thread characterization on heterogeneous multiprocessors, Operating Systems Review 51 (1) (2017) 113–123.

[31] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, D. Steinberg, Top 10 algorithms in data mining, Knowledge and Information Systems 14 (1) (2008) 1–37.

[32] T. G. Dietterich, Machine-learning research – four current directions, AI MAGAZINE 18 (1997) 97–136.

[33] R. E. Schapire, The strength of weak learnability, Machine Learning 5 (2) (1990) 197–227. doi:10.1023/A:1022648800760.
URL `https://doi.org/10.1023/A:1022648800760`

[34] Wikipedia, Artificial Neural Networks: approximation theorem, `https://en.wikipedia.org/wiki/Universal_approximation_theorem`.

[35] Machine Learning Group at the University of Waikato, `Weka` - Data Mining Software in Java, `http://www.cs.waikato.ac.nz/~ml/weka` (2017).

[36] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, H. Kunieda, Maps: An integrated framework for mpsoc application parallelization, 2008, pp. 754–759.

[37] A. Mallik, S. Mamagkakis, C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, P. Marwedel, F. Capman, S. Collet, N. Mitas, D. Kritharidis, Mnemee – an automated toolflow for parallelization and memory management in mpsoc platforms, in: 48th ACM/IEEE Design Automation Conf. (DAC'11), San Diego, CA - USA, 2011.

[38] K. Latif, C. Effiong, A. Gamatié, G. Sassatelli, L. Zordan, L. Ost, P. Dziurzanski, L. Soares Indrusiak, An Integrated Framework for Model-Based Design and Analysis of Automotive Multi-Core Systems, in: FDL: Forum on specification & Design Languages, Work-in-Progress Session, Barcelona, Spain, 2015.
URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-01418748

[39] DreamCloud Project Consortium, McSim - Manycore platform Simulation tool for NoC-based platform at a Transactional Level Modeling level, https://github.com/DreamCloud-Project (2016).

[40] K. Latif, M. Selva, C. Effiong, R. Ursu, A. Gamatié, G. Sassatelli, L. Zordan, L. Ost, P. Dziurzanski, L. Indrusiak, Design space exploration for complex automotive applications: An engine control system case study, in: 8th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Prague, Czech Republic, 2016.

[41] CNRS, Deliverable D5.2 – Abstract Simulation Platform, European DreamCloud FP7 Project - http://www.dreamcloud-project.org/results (2015).

[42] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, M. Robert, Full-system simulation of big.little multicore architecture for performance and energy exploration, in: 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC), 2016, pp. 201–208. doi:10.1109/MCSoC.2016.20.

[43] C. J. van Rijsbergen, Information Retrieval, Butterworth, 1979.

[44] H. He, E. A. Garcia, Learning from imbalanced data, IEEE Trans. on Knowl. and Data Eng. 21 (9) (2009) 1263–1284.

[45] L. S. Indrusiak, P. Dziurzanski, An interval algebra for multiprocessor resource allocation, in: SAMOS, IEEE, 2015, pp. 165–172.