Accelerating Random Forest on Memory-Constrained Devices Through Data Storage Optimization

Camélia Slimani, Chun-Feng Wu, *Member, IEEE*, Stéphane Rubini, Yuan-Hao Chang, *Senior Member, IEEE*, and Jalil Boukhobza, *Senior Member, IEEE*

Abstract—Random forests is a widely used classification algorithm. It consists of a set of decision trees each of which is a classifier built on the basis of a random subset of the training data-set. In an environment where the memory work-space is low in comparison to the data-set size, when training a decision tree, a large proportion of the execution time is related to I/O operations. These are caused by data blocks transfers between the storage device and the memory work-space (in both directions). Our analysis of random forests training algorithms showed that there are two major issues : *(1)* Block Under-utilization: data blocks are poorly used when loaded into memory and have to be reloaded multiple times, meaning that the algorithm exhibits a poor spatial locality; *(2)* Data Over-read: the data-set is supposed to be fully loaded in memory whereas a large proportion of data are not effectively useful when building a decision tree. Our proposed solution is structured to address these two issues. First, we propose to reorganize the data-set in such a way to enhance spatial locality and second, to remove the assumption that the data-set is entirely loaded into memory and access data only when effectively needed. Our experiments show that this method made it possible to reduce random forest building time by 51 to 95% in comparison to a state-of-the-art method.

Index Terms—Random forests, memory hierarchy, I/O accesses reduction, embedded systems

1 Introduction

According to the International Data Corporation, the volume of data created between 2020 and 2024 will surpass the one created over the last 30 years [1]. This is mainly due to the billions of end-point devices used in transportation, medicine or entertainment [2]. These devices collect and analyze huge amounts of data to extract meaningful information [3]. Analysis are traditionally performed on the cloud to exploit high computation power. However, this requires to send collected data on the cloud, that is exposing them to security threats, when applications deal with medical [4], industrial [5] or transportation data [6], and increasing the network traffic and energy consumption [7]. According to [8], 70% of energy consumption of an embedded device is spent on communication. Since, these devices may be battery-backed, it may be cheaper to process data than to send it [8].

While this is not considered as an issue for several applications, it can cause serious problems for critical data applications.

Processing data directly on embedded devices can be a solution for such an issue. Nevertheless, this solution struggles from the limitation in size of the main memory. In fact, memory capacity can hardly scale as fast as the volume of data to process [9]. We cite the RaspBerry Pi Zero as an example of embedded device used to perform several learning tasks [10], [11]. To overcome these constraints, some studies propose to trade accuracy for lighter models [12], [13]. However, some applications require high accuracy such as healthcare applications [14], industry 4.0 [15] and autonomous driving [8]. Thus, our objective in this study is to train models that are as accurate as the original methods at a lower cost.

One of the wide spread data analysis algorithms is classification. The objective is to assign a category to a recorded information according to observed features [3]. A classification model is trained on a learning set (data-set) of elements characterized by features and labeled with their real classes [3]. Among the classification algorithms, *Random Forests* (RF) [16] is a powerful and widely used one. It individually trains a set of decision trees, each of which is trained on a subset of elements called a *bootstrap*. A decision tree is a set of conditions based on the feature values which group elements that are in the same class together. Thus, the decision tree building process consists in identifying the best features that make it possible to obtain this grouping [17]. This process is performed iteratively by trying different sets of features.

- *Camélia Slimani and Stéphane Rubini are with the University of Brest, Lab-STICC, CNRS, F-29200 Brest, France. E-mail: {camelia.slimani, stephane.rubini}@univ-brest.fr.*
- *Chun-Feng Wu is with the National Yang Ming Chiao Tung University. E-mail: cfwu417@cs.nycu.edu.tw.*
- *Yuan-Hao Chang is with the Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan. E-mail: johnson@iis.sinica.edu.tw.*
- *Jalil Boukhobza is with ENSTA Bretagne, Lab-STICC, CNRS, F-29200 Brest, France. E-mail: jalil.boukhobza@ensta-bretagne.fr.*

Manuscript received 22 December 2021; revised 27 September 2022; accepted 12 October 2022. Date of publication 19 October 2022; date of current version 10 May 2023.
This work was supported by Bretagne region, France.
(Corresponding author: Camélia Slimani.)
Recommended for acceptance by R. Marculescu.
Digital Object Identifier no. 10.1109/TC.2022.3215898

During the tree building process, most RF implementations assume that the data-set can entirely fit into the main memory. However, embedded systems, which are usually resource-constrained, cannot easily meet this assumption. Practically, the size of the data-set is usually larger than the memory capacity, and thus, significant data movements may take place between memory and storage devices. Our experiments measured that building a decision tree with a memory work-space 8 times smaller than the size of the data-set is 25 times slower than the case where the data-set fits entirely into the memory [18]. We analyzed the traditional RF algorithm as proposed in [16] and identified two main issues that cause the increase of data movements between memory and storage when the memory pressure increases. *(1) Block Under-Utilization:* A given data-set block is accessed on multiple steps of the decision tree building process, which makes the algorithm present a poor spatial locality. As a consequence, the same data block is loaded several times throughout the whole algorithm execution. *(2) Data over-read:* A given data-set block may contain several data elements that are not part of the bootstrap, they are thus uselessly moved back and forth from/to the memory to storage, which highly increases the volume of I/O operations.

To address these issues, we propose an RF building algorithm based on two principles:

1) *Enhancing spatial locality by data-set reorganization:* This solution consists in reorganizing the data-set on the storage device in a way that data that are likely to be accessed together during the tree building process (on the same decision tree nodes) are stored in neighboring blocks. This idea is motivated by a random forest decision trees property that we observed experimentally on a set of data-sets. In fact, if a pair of data elements are successively accessed during one decision tree building, they have a high probability to be successively accessed for the building of other decision trees. Thus, those data elements can be stored in neighboring blocks.
2) *Accessing required data elements on-demand:* Instead of loading the whole data-set before starting to build a decision tree, we propose to take into account the available memory work-space to only load the effectively required data and adapt the decision tree building process accordingly. We identified three specific cases according to the memory work-space available: *(1)* loading the whole data to build a full sub-tree; *(2)* loading the necessary data to build only one node of the sub-tree; or *(3)* subdividing the node's data elements into chunks that can fit in memory and iterate over their elements separately to test splitting features. In the latter case, the results obtained from each the chunk processing are aggregated.

The proposed optimizations were tested by modifying the framework *Ranger* [19]. This framework already implements some memory optimizations for random forests. A comparison of the proposed method with the original version of Ranger showed that the proposed method (that combines the two optimizations) reduces the RF building time by up to 95% for high memory constraints, and this without alteration of algorithm accuracy. The two optimizations individually evaluated gave the following results: The data-set reorganization reduces the execution time by 35 to 74% and on-demand data-access reduces it by 20 to 77%.

TABLE 1
Notation Table

Notation	Description
D	Data-set
N	Number of elements of the data-set
t	Number of decision trees
M	Memory work-space size
B	Number of elements in an I/O block
d	Number of features of an element
F	Set of features to be tested
$S(i)$	Memory size occupied by i features
R	Execution time reduction

The remainder of this paper is organized as follows: Section 2 gives a brief background about random forests. In Section 3, we motivate our study. The proposed method is detailed in Section 4, and we evaluate it in Section 5. Section 6 gives an overview of state-of-the-art methods and a positioning of our proposal. Finally, Section 7 concludes this work and gives some perspectives.

2 Background

In this section, we start by describing the Random Forest building algorithm, then, we briefly describe swap mechanism that is responsible for I/Os in RF algorithm.

2.1 Random Forest Building Algorithm

In this Section, we introduce RF and the decision tree building process. Table 1 gives the notations used.

RF is a supervised machine learning algorithm used for classification and regression [17]. It is composed of T decision trees. The input of the learning phase of an RF is a *data-set*, that is a set of N observed data elements characterized by d features and labeled with their real classes. Each decision tree is trained on a subset of this data-set according to the method explained in the next section. The objective of a decision tree is to predict the class of an element knowing its observed features. RF builds a set of decision trees to limit the prediction error. The final predicted class of the elements is the predominant class among the predicted classes of the T trees of the forest. In what follows, we focus on binary decision trees, where the number of children nodes of a node is 2.

A binary decision tree is a tree-like graph where each node represents a Boolean condition that depends on a feature of the learning set. This Boolean condition allows to split the elements of the learning set into two subsets: a subset of elements for which the condition is 'true'; and another one for which the condition is 'false'. The topmost node of the tree is called the *root node*. All the elements of the learning set are assigned to this node at the beginning of the decision tree building process. The bottom nodes of a tree, called *leaf nodes*, are *pure* meaning that all elements assigned

TABLE 2
Data-Set Example

Label	f_1	f_2	f_3	f_4	Class
A	$\binom{0}{1}(1)$	$\binom{0}{0}(5)$	$\binom{0}{1}(9)$	$\binom{0}{0}(13)$	$\binom{0}{1}(17)$
B					
C	$\binom{0}{0}(2)$	$\binom{1}{0}(6)$	$\binom{0}{0}(10)$	$\binom{1}{0}(14)$	$\binom{0}{0}(18)$
D					
E	$\binom{1}{0}(3)$	$\binom{1}{1}(7)$	$\binom{1}{1}(11)$	$\binom{1}{0}(15)$	$\binom{1}{1}(19)$
F					
G	$\binom{1}{0}(4)$	$\binom{0}{1}(8)$	$\binom{0}{0}(12)$	$\binom{1}{1}(16)$	$\binom{0}{0}(20)$
H					

Pairs of data values are grouped into I/O blocks. These blocks are numbered (.) so as to be used in our explanations in Sections 3.1 and 4.2.3.

to them are from the same class. Thus, the objective of the training process is to find the sequences of conditions that allow to obtain pure nodes.

A decision tree is built according to the *bagging Algorithm* [16]. To illustrate this process, we use the following example. We assume a data-set, given in Table 2, composed of 8 elements (labeled from A to H) characterized by 4 features $\{f_1, f_2, f_3, f_4\}$. Fig. 1 shows the steps of building a decision tree on the basis of this data-set:

1) *Bootstrap creation (step 1):* A subset of the data-set, called *bootstrap*, is formed by a random sampling with replacement (i.e., each element can be sampled multiple times). Note that the overall number of samples is N which equals the number of elements in the data-set. Elements of the bootstrap are assigned to the root node. Fig. 1 shows the sampled bootstrap, and its assignment to the root node N_0.
2) *Split trial (step 2):* The second step consists of splitting the elements of the bootstrap according to boolean conditions based on a random subset F of features. Once F is formed, the elements of the bootstrap are distributed according to their value for each feature, resulting in $|F|$ potential trees. In Fig. 1, the sampled features are f_1 and f_4. Elements of the bootstrap are distributed according to each of these features resulting in two possible trees.
3) *Effective split (step 3):* This step consists of choosing the best splitting feature among the previous subset F, that is the one that allows to group the most elements of the same class together. Then, the node is effectively split according to the best feature. In Fig. 1, f_1 is chosen since it already gives a pure leaf node, which is N_1, as all the elements assigned to it are from the same class.
4) Step 1, 2 and 3 are repeated with the resulting nodes if they do not satisfy a stopping condition that is theoretically the pureness, and a maximum tree depth or a minimum number of elements per node in practice.

For the sake of simplicity, in the given example, the features of the used data-set take binary values. In more general cases, the features take real values. Thus, split trials (step 2) include an additional step which consists in choosing the best split value for each tested feature. It should be noted that the general case is taken into consideration in the proposed mechanism in the same way Ranger Framework does [19].

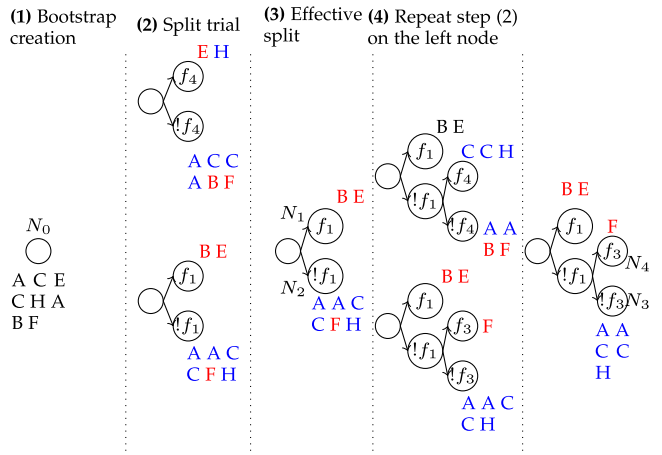


Fig. 1. Bagging algorithm illustration.

2.2 Swap Mechanism and I/Os

Swapping is an operating system mechanism that consists in a backup on disk for memory pages. This mechanism is transparent to programs [20]. Swap space is implemented as a partition or a file in the operating system. Page movements between the main memory and the secondary storage are triggered by Page Frame Reclaiming Algorithm (PRFA), which objective is to free User mode page frames for a future use [20].

The Swapping mechanism can considerably slow down the execution of an application when the available memory workspace is not sufficient for the application. This slowdown is correlated to the performance difference between technologies used to respectively implement main memory and secondary storage. In what follows, we will describe the impact of this mechanism when building a random forest.

3 Motivation

In this section, we highlight the main issues when building an RF. To do so, we illustrate the I/O operations that occur using a motivational example. Then we measure the proportion of I/O time in comparison to the overall decision tree building time, using experiments on real data-sets.

3.1 Motivational Example

We assume a simplified system composed of a main memory work-space and a secondary storage space[1]. The Bagging algorithm assumes that the data-set fits in memory; this means that it is read once and kept in memory during all the processing[2]. We use the example presented in Table 2. We are interested in I/Os that occur during the step 2 of the bagging algorithm. For simplicity, we suppose a memory work-space that can hold four values of features ($M = 4$) and an I/O block of two feature values ($B = 2$).

Table 3 shows the following information: the labels of elements that need to be accessed when trying to split nodes

1. Data organization to optimize the processor cache accesses is out of the scope of this paper; we only focus on the number of I/O operations since they are several order of magnitudes slower than memory operations and they significantly slow down the decision tree building.
2. Data are loaded per column for an optimal feature value extraction [19]

TABLE 3
I/O Access Pattern for Splitting Each Node

Node	Elements	Feature	Accessed blocks	Average usage per block
N_0	$\{A, A, B, C, C, E, F, H\}$	f_1	(1),(17),(2),(18),(3),(19),(4),(20)	
		f_4	(13),(17),(14),(18),(15),(19),(16),(20)	75%
N_1	$\{A, A, C, C, F, H\}$	f_3	(1),(17),(2),(18),(3),(19),(4),(20)	
		f_4	(13),(17),(14),(18),(15),(19),(16),(20)	50%
N_2	$\{B, E\}$	/	(2),(18),(3),(19)	50%
N_3	$\{A, A, C, C, F, H\}$	/	(1),(17),(2),(18),(4),(20)	50%
N_4	$\{F\}$	/	(3),(19),(15),(19)	50%

of Fig. 1 according to testing features, the accessed storage blocks (as numbered in Table 2) and the average percentage of used data for each node. The first line of the table shows for example, the accesses resulting from N_0 division. The features to be tested are f_1 and f_4. When trying the division using feature f_1, the blocks that need to be accessed are those containing values of features f_1 of elements of node N_0 $((1), (2), (3), (4))$, and blocks containing values of the effective class $((17), (18), (19), (20))$.

We observe the following issues:

1) Low spatial locality: In the given example, to process node N_0, all the blocks that contain data values of the features to be tested are accessed and all of them are 50% effectively used. The elements that are needed to process a node are distributed on multiple data blocks which are poorly exploited (50%). Thus, the original data-set organization exhibits a low spatial locality.
2) Useless data movement: Since the data-set is loaded into memory before starting to build the decision trees, the elements that are not part of the bootstrap are distributed among data blocks that are moved from secondary storage to main memory when the useful data are needed. In the given example, when trying to split node N_0, the block (2) that contains the f_1 feature of element C is moved into the main memory. Yet, this block also contains the feature f_1 of element D although it is not needed. This makes the average percentage of used data per block 50% for several nodes in this example as shown in Table 3.
3) Multiple accesses to the same blocks for an individual node: The blocks that contain the class data are accessed as many times as features to be tested (i.e., $|F|$ times). In the given example, each class block is moved to the main memory twice, since there are two features to be tested. This behavior increases even more the the number of I/O operations.

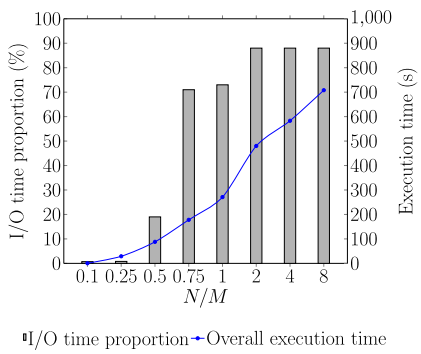


Fig. 2. Execution time and I/O time % according to N/M.

3.2 Experimental Measurement of I/O Time

For this experiment, the data-set Wearable which volume is 4.58 MB (see Table 7) was used. The evaluation platform was the same as the one used in the evaluation part (see Section 5.1.2). The Random Forest used in this experiment is Ranger Framework described in Section 5.1. We used different memory configurations expressed by the proportion of data-set volume N over the volume of available memory work-space M, that is N/M.

Fig. 2 shows the obtained results. We observe that when the proportion N/M is lower than 0.5, the system spends less than 20% of the time performing I/O operations. This proportion rapidly grows to reach 88% when N/M is equal to 8. The same phenomena is observed for execution time. We observe that when $N/M = 0.75$, or $N/M = 1$, the I/O time proportion is still high despite the fact that the data-set can fit into memory. This is because several data structures (such as indexes) must be maintained in memory while building decision trees.

For instance, building a tree when $N/M = 8$ is almost 25 times slower than building it when $N/M = 0.25$. As discussed earlier, this performance drop is mainly due to useless I/Os and a low spatial locality on memory accesses.

4 Contribution

In this section, we present an adaptive RF algorithm whose objective is to reduce I/O operations for memory constrained environments. We start by giving an overview of the method, then we describe in detail the principles we used to address the issues shown in the previous section.

4.1 Overview of the Method

As shown in the previous section, the original RF algorithm generates a substantial amount of I/O operations when the memory work-space is limited. The proposed method is structured around two optimizations to address above mentioned issues[3]. Fig. 3 gives an overview of the method described hereafter.

3. No prefetching and eviction algorithms have been designed. The default ones in Linux page cache were used. We achieved no updates at the operating system level, only the application part was upgraded.

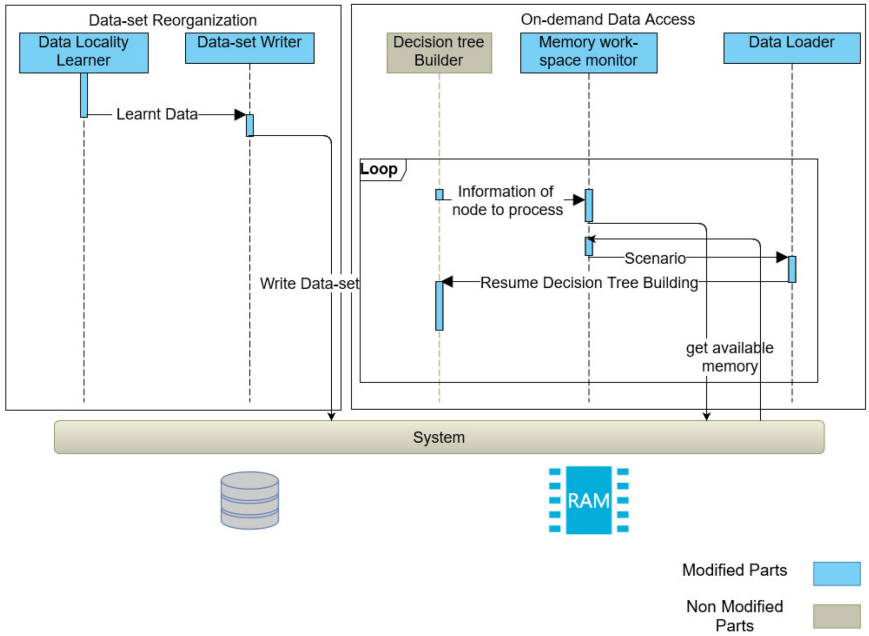


Fig. 3. Overview of the method.

1) *Data Reorganization:* The objective is to enhance spatial locality of the algorithm. To do so, we propose to reorganize the data-set blocks in a way that each of them contains elements that are likely to be accessed during the split of the same node, before building the RF decision trees. We decompose the data reorganization into two functions:
   a) *Data Locality Learner:* The objective of this module is to determine groups of data elements that are likely to be successively accessed. The learning method is detailed in the next section.
   b) *Data-set Writer:* Each group of elements that are likely to be accessed together, are stored (written) in neighbor blocks on the storage device. At the end of this process, we obtain a copy of the data-set stored in a reorganized way on the storage device. This reorganized data-set is used to build the RF decision trees. The original data-set copy can be discarded or kept according to application needs. Rewriting the whole data-set may seem costly, but as it is read several time, this operation is profitable.
2) *On-demand Data Access Decision Tree Builder:* We decompose this second optimization into three modules:
   a) *Decision Tree Building Module:* This module effectively performs the decision tree building steps of the bagging algorithm. The method does not prefetch all the data into the memory work-space at the beginning of the RF building. Rather, it performs on-demand data loading according to the memory space available. To do so, it relies on two other modules: the memory work-space monitor and the data loader.
   b) *Memory Work-Space Monitor:* This module compares the volume N of data to process for a given node and the available memory work-space M. It selects between three scenarios related to the available memory: all node's data can fit in memory (*scenario 1*), memory can hold all values of features to be tested in the split trial step but cannot hold all the node's data (*scenario 2*), and memory cannot hold all the values of features to be tested (*scenario 3*).
   c) *Data Loader Module:*It smartly loads the useful data according to the scenario selected by the memory work-space monitor. Once data are loaded, they are processed by the decision tree building module to complete the split trial step. This module also forms an index of the data elements locations on the data-set file before starting decision tree building. This is done to easily locate useful data without reading them sequentially during data loading. The index is kept in memory during decision tree building. In terms of memory overhead, the proportion of index size as compared to data-set size equals $1/d$. This means that the size of the index becomes negligible as compared to the data-set size when the dimensions (d) of the data-set grows. As an example, for the real data-sets considered in the paper, the lowest and highest dimension of the used real data-sets are respectively 8 and 299 (see Table 2), which makes the proportion of index size as compared to data-set size vary between 0.3% and 12.5%.

Note that both optimizations can individually reduce the volume of I/O operations. In what follows, we detail each one of them.

4.2 Enhancing Spatial Locality by Data Reorganization

Data reorganization is based on the observation that decision trees of the same RF exhibit some similarity in the way

TABLE 4
Table of Contingency

X/Y	Y_1	Y_2	...	Y_s	Sum
X_1	n_{11}	n_{12}	...	n_{1s}	a_1
X_2	n_{21}	n_{22}	...	n_{2s}	a_2
...	...	...	...	...	...
X_r	n_{r1}	n_{r2}	...	n_{rs}	a_2
Sum	b_1	b_2	...	b_s	

they classify the elements. In what follows, we detail what similarity between decision trees is, and how did we exploit this property to enhance spatial locality.

4.2.1 Similarity Between RF Decision Trees

We will show that, in an RF, if a pair of data elements are classified in the same leaf node in one tree, they are likely to be classified together in another tree. We first formalize this property and evaluate its relevance.

Decision Trees Similarity. We assume a bootstrap that contains N elements. T_1 and T_2 are two decision trees built on the basis of the selected bootstrap. Each decision tree results in a set of leaf nodes. A leaf node groups elements of the bootstrap that share the same features. The union of all groups of elements (leaf nodes) gives the original bootstrap, and the intersection between a given pair of groups is empty. Thus, a decision tree results in a clustering of the bootstrap, where each group of elements affected to a leaf node represents a cluster. Let us assume that tree T_1 (resp. T_2) gives the clustering P_1 (resp. P_2). To know whether a pair of elements classified in the same leaf nodes of a decision tree are likely to be classified together in another one, we can compare the similarity of obtained clusterings P_1 and P_2.

In the literature, several metrics exist that evaluate similarity between clusters [21]. We relied on the *Adjusted Rand Index (ARI)*, which is a widely used one. It is calculated from the contingency matrix that is given in Table 4, where $n_{ij} = |X_i \cap Y_j|$ is the number of observations that are common to X_i and Y_j. The formula for ARI index [22] is :

$$ARI = \frac{\binom{N}{2} \cdot \sum_{i,j}^{r,s} \binom{n_{ij}}{2} - \sum_i^r \binom{a_i}{2} \cdot \sum_j^s \binom{b_j}{2}}{1/2 \cdot \binom{N}{2} \cdot \left[\sum_i^r \binom{a_i}{2} + \sum_j^s \binom{b_j}{2}\right] - \sum_i^r \binom{a_i}{2} \cdot \sum_j^s \binom{b_j}{2}} \quad (1)$$

Its value ranges between -1 and 1; a high ARI value indicates a high similarity.

Experimental Measurement of the ARI. In order to check whether this property is relevant, we measured the ARI index of the clusterings obtained using two decision trees, with multiple real data-sets picked from UCI Data-set Repository [23]. Table 5 shows the obtained ARI measures. The bootstraps used to build the two decision trees were formed by sampling the data-set without replacement, thus, the bootstraps contain all the data-set. We rely on the fact that the bootstrap samples have 63% of observations in common [24] and assume that the similarity property, if checked, would be verified in case of bootstraps that are different to some extent.

TABLE 5
Obtained ARI Measures

Data-set	Wearable	Adult	Covertype	Ecoli	Wine	Heart Failure	Poker
ARI	0.27	0.26	0.12	0.37	0.42	0.33	0.034

The results show that most of the ARI indexes range between 0.12 and 0.42 (except for Poker data-set). In order to tell if these values reflect significant similarity between the clusterings obtained using two decision trees, we compared them to state-of-the-art studies [22], [25]. These studies compare the similarity between clusetrings obtained by popular algorithms and the ground-truth clusters, and the ARI indexes range between 0.07 and 0.87[22], 0.05 and 0.8 [25]. Thus we admit that the similarities between decision trees are satisfactory enough.

In addition, we have generated two random clusterings of 50,000 data elements. The process to generate them consists of generating a data-set of 50,000 elements and randomly assigning each of them to a cluster. We did the assignment two times in order to generate two random clusterings. When measuring the ARI index of these two clusterings, the result is 0.0009, which is a few orders of magnitude lower than the indexes obtained from the above mentioned data-set elements clustering using two decision trees.

4.2.2 Data-Set Reorganization Method

Our approach is to re-organize data-set blocks in such a way that each of them contains elements that are likely to be accessed during the split of the same node. To do so, we take advantage of the property discussed in the previous section. In fact, since the probability of similarity between leaf nodes is high, we use a decision tree trained on this data-set to extract information about elements clustering, and then exploit this information to reorganize the data-set before building the remaining decision trees. This induces an additional complete data-set write operation on the storage device, but it is profitable in view of the several trees to be built. For instance, the default number of trees in *Ranger* [19] and *Scikit-learn* [26] frameworks is 100. The steps are explained hereafter.

1) *Build the T_0 Tree:* This step is performed by the *Data Locality Learner* shown in Fig. 3. The objective of this first step is to build a decision tree that would allow to get a clustering of data-sets elements, where tree leaf nodes represent the clusters. In order to get a clustering of all data-set elements and not only the bootstrap, the decision tree T_0 (and only this one) is built on the basis of the whole data-set. As a consequence, the decision tree T_0 is deeper (more elements), thus slower to build, than ordinary decision trees. In addition, it is more subject to over-fitting. Thus, in our method, T_0 is a "disposable" tree used to cluster data-set elements; it is not saved in the RF decision trees that are used for the inference step, so as not to change the final generated forest. Our strategy accelerates the forest building and does not change it.

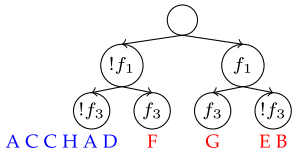


Fig. 4. T_0 tree.

2) *Re-write the data-set on the storage device:* Once the decision tree T_0 obtained, we dispose of the clustering of all the data-set elements. The information is used to rewrite the whole file in the storage device (we assume there is space on the storage device for such a write operation). A new data-set is written and that is organized in a way that data elements within a block belong to the same cluster (leaf node). This way, elements of a block are likely to be accessed altogether during the building of the other trees. If the size of a block cannot contain all the elements of a cluster, they are stored on multiple ones. The data-set was written sequentially to reduce the SSD wear-out issue, since sequential writes are less harmful than random ones [27]. Since the bootstrap is randomly sampled, the appearance order of elements in the data-set is not important. Thus, re-writing the data-set in a different order does not alter the accuracy of the decision trees.
3) *Effective Random Forests Trees Building:* Once the data-set blocks are written according to the clustering obtained from the tree T_0, the remaining decision trees of the RF are formed on the basis of the newly stored data-set.

Note that the building of tree T_0 and the writing of the new data-set are deliberately separated in the previous explanation for simplicity. In our implementation, see Algorithm 1, their operations are overlapping in a way that each time a leaf node is reached, its elements are written into neighbor blocks. The objective of this overlapping is to free the memory space dedicated to store the element indexes assigned to a leaf node, as soon as the blocks are written. Algorithm 1 shows that T_0 is built according to the decision tree building algorithm, except that after splitting a node (lines 2-5), the pureness of children nodes is checked (lines 6-7). If a resulting node is pure (leaf node), its elements are written into storage blocks (line 8).

Algorithm 1. T_0 Building and Data-Set Reorganization

Data: a matrix of data

1: **while** *there exists an impure node* n *in the leaves* **do**
2: Create the subset F of features
3: **for** $f \leftarrow 0$ **to** $|F| - 1$ **do**
4: Split Trial of node n according to f
5: $n_l, n_r \leftarrow$ Effective Split of n according to the best feature
6: **foreach** *node* cn *in* $\{n_r, n_l\}$ **do**
7: **if** cn *is pure* **then**
8: Write node elements into the same secondary storage blocks
9: **else**
10: Add node cn to impure nodes list

In terms of I/O cost, the T_0 tree building step implies the traditional decision tree I/O cost, plus the cost to write the new data-set (the volume is equal to the data-set volume).

4.2.3 Method Illustration

Here we illustrate the proposed optimization processing by applying it to the motivational example. Fig. 4 shows the tree T_0 built by the method. Once the decision tree T_0 is built, the elements are reorganized in the following order: $\{A, C, H, D, F, G, E, B\}$. Thus, the I/O accesses generated by the decision tree building are given in Table 6. In comparison to Table 3, we observe that for all the nodes, the average percentage of used data per block with the optimized data-set is higher or equal to the traditional method. The additional I/O cost of T_0 building and data-set rewriting is substantial in this small example, but for a realistic case this cost is largely amortized as it is shown in the evaluation part.

4.3 On-Demand Data Accesses

In this section, we give more details about the second optimization. We describe each of the three scenarios outlined in the overview section.

The objective of on-demand data accesses is to reduce the I/O operations by reading from the storage system, and keeping in memory, only effectively needed data for the next steps of the tree building process. Algorithm 2 sets the data loading method according to the available memory work-space. If the memory work-space makes it possible to load all nodes' data, then it is loaded; and the algorithm builds a full sub-tree (from the current node to the leaves) (scenario 1). If the memory space is too small to apply the scenario 1, then we check whether there is room for loading

TABLE 6
I/O Access Pattern for Splitting Each Node of the Motivational Example After Data-Set Reorganization

Node	Elements	Feature	Accessed blocks	Average usage per block
N_0	$\{A, A, C, C, H, F, E, B\}$	f_1	(1),(17),(2),(18),(3),(19),(4),(20)	
		f_4	(13),(17),(14),(18),(15),(19),(16),(20)	75%
N_1	$\{A, A, C, C, H, F\}$	f_3	(1),(17),(2),(18),(3),(19)	
		f_4	(13),(17),(14),(18),(15),(19)	66.66%
N_2	$\{B, E\}$	/	(4),(20)	100%
N_3	$\{A, A, C, C, H\}$	/	(1),(17),(2),(18)	75%
N_4	$\{F\}$	/	(3),(19),(15),(19)	50%

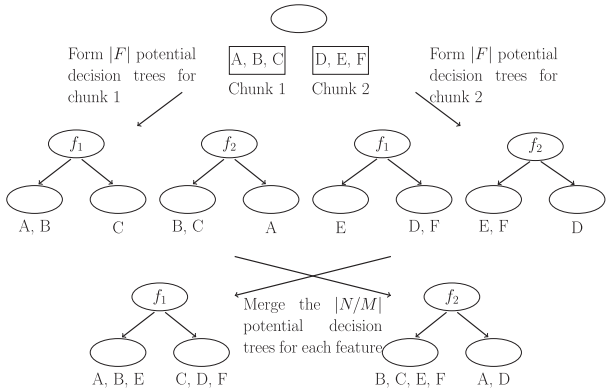


Fig. 5. Scenario 3 processing.

data to fully build the current node (scenario 2). In the case where there is not enough memory space for scenario 2, then we subdivide data into chunks. We split separately the node for each chunk, and then we aggregate the children nodes. Doing so, temporal locality is fully exploited as the algorithm loads each chunk once and goes through it several times to split the current node. The different scenarios are detailed in the following sections.

4.3.1 Scenario 1, Full Sub-Tree Building

This is the scenario where the available memory space can hold all the features and the effective class information of the current node elements. That is $M \geq |N_i| \cdot S(d+1)$ (see line 1 of Algorithm 2); the volume $|N_i| \cdot S(d+1)$ is the volume occupied by the d features of the $|N_i|$ elements of node i plus the effective class information (the function $S(i)$ returns the memory space occupied by the i feature values). In this case, one can load all these data into the memory work-space and keep them in until reaching the terminal nodes for the sub-tree whose root node is the current node. By loading all the features and not only those belonging to feature set F, we make sure that for the child nodes, the features needed are already loaded in memory. In other words, data features that will be needed in the children nodes are prefetched.

In order to fully take advantage of the loaded data, we need to process the nodes in Depth-First strategy by processing children nodes of the current node instead of a Breadth-First strategy, where the nodes of the same level are processed before moving to a deeper level; this way, data that are already in memory will be re-exploited for the full sub-tree without generating more I/O operations.

4.3.2 Scenario 2, Full Node Building

This is the scenario where the memory work-space cannot hold all the d features but is large enough to contain currently needed features, that is $|N_i| \cdot S(|F|+1) \leq M < |N_i| \cdot S(d+1)$ (see line 4 of Algorithm 2). In this case, we load the features of the elements that belong to the feature set F and keep them in memory until the node is fully processed.

By reducing the volume of data to the useful data only, we reduce the amount of I/O operations that occur during the step 2 of decision tree building and avoid swapping in and out due to memory pages that contain both useful and useless data (as outlined in the background section).

Note that in the scenarios 1 and 2, the useful data are loaded right after setting the loading method as shown in lines 3 and 6 of Algorithm 2.

Algorithm 2. Data Loading

Data: N_i : Node to be processed, M: Available memory work-space , D : Data-set file , F : set of features to be tested
1: **if** $|N_i| \cdot S(d+1) \leq M$ **then**
2: Scenario $\leftarrow$ scenario 1
3: Read from D all the features of elements of N_i into the matrix X
4: **else if** $|N_i| \cdot S(|F|+1) \leq M$ **then**
5: Scenario $\leftarrow$ scenario 2
6: Read from D features belonging to feature set F of elements of N_i into the matrix X
7: **else**
8: Scenario $\leftarrow$ scenario 3

4.3.3 Scenario 3, per-Chunk Split Trial

This strategy is used when the useful data cannot fit in memory. The case occurs when $M < N_i \cdot S(|F|+1)$, meaning that the memory work-space cannot hold all the features of the feature set F plus its class. In that situation, variable *Scenario* is set to *scenario 3* in line 8 of Algorithm 2. The idea is to process data per chunk that can fit in memory. In other words, we perform the split trial on each chunk. This means that we divide the step 2 of decision tree building algorithm to the steps shown in Algorithm 3 and Fig. 5, summarized as follows:

1) *Load one chunk of elements* of node i that can fit into memory: a chunk is a subdivision of $\frac{M}{S(|F|+1)}$ elements of node i. $S(|F|+1)$ is the volume occupied by useful data on a single element, and then $\frac{M}{S(|F|+1)}$ is the number of elements that can be kept in memory. The first step of the algorithm is to read the values of features in the feature set F for each element of the chunk from the data-set file (see line 2 of Algorithm 3). This step is performed using an initially built index that helps finding the positions of elements in the data-set file.
2) *Process each chunk independently:* For each chunk, the method builds the $|F|$ potential trees by distributing the elements of the chunk according to the $|F|$ features; this is the application of split trial on the chunk (see lines 4 and 5 of Algorithm 3). As shown in Fig. 5, elements of each chunk are distributed according to the features to test $\{f1, f2\}$ to obtain 2 potential trees.
3) Repeat the steps (1) and (2) until all the chunks of the current node elements are processed.
4) *Combine the obtained trees:* This step aims at merging the $|F|$ potential trees obtained from each chunk (see line 6 of Algorithm 3). Merging potential trees of the chunks according to one feature of the set F is performed as follows:
   1) Group the child nodes that correspond to the same test value over the chunks. In Fig. 5, we observe that for each feature ($f1$ and $f2$), the

right and left nodes of the resulting potential trees according to each chunk are merged to obtain the final potential trees.

2) Compute the number of elements per class in each group: the number of elements per child node are summed over the chunks to obtain the number of elements per class and per node of the final potential tree. No additional I/Os are incurred in this step.

The choice of the best feature can thus be deduced as in the traditional method.

The division and combination methods described above do not alter the final obtained trees, since analytically, the pureness measurement is the same as in the traditional method.

Algorithm 3. Process the Node per Chunk

Data: D: Data-set file, F set: splitting features set
Result: Potential $|F|$ trees

for $c \leftarrow 0$ **to** $\frac{|N_i| \cdot S(|F|+1)}{M}$ -1 **do**
  Load from D chunk c
  // **Process each chunk**
  for $f \leftarrow 0$ **to** $|F| - 1$ **do**
    Split trial of chunk c according to f
Combine the obtained sub-trees

4.4 New RF Algorithm

The proposed RF algorithm relies on the two previously detailed optimizations. It is given in Algorithm 4. The first step consists of reorganizing the data-set using Algorithm 1. Once the new data-set reorganized, decision trees are built using on-demand data accessed by taking into consideration the available memory work-space.

Algorithm 4. RF New Algorithm

Data: Original Data-set
Build Tree T_0 **and Reorganize Data-set** (Algorithm 1)
for $t \leftarrow 0$ *to* $T - 1$ **do**
  Create a bootstrap
  while *there exists an impure node n among the children of the current* **do**
    Create the subset F of features
    Data loading (Algorithm 2)
    if *Scenario* $\neq$ *scenario 3* **then**
      Split trial using traditional method
    else
      Process the node per chunk (Algorithm 3)
    Choose the best feature that gives the purest child nodes
    Effectively split the node n according to the best feature
  Add the built tree to the RF
return RF

5 Evaluation

In this section, we start by describing the evaluation methodology. Then we show the obtained results and discuss them.

TABLE 7
Used *Data-Sets*

Type	*Data-set*	Number of features	Number of observations	*Data-set* file size (Mo)
Real	Covertype	54	581012	239.36
	Wearable	8	75128	4.58
	Adult	14	48842	5.21
	Ecoli	8	336	0.02
	Heart Failure	299	13	0.03
	Wine	178	13	0.01
	Poker	11	25010	2.1
Synthetic		16	100000	48.82
		32	5000	
		64	2500	
		182	1250	

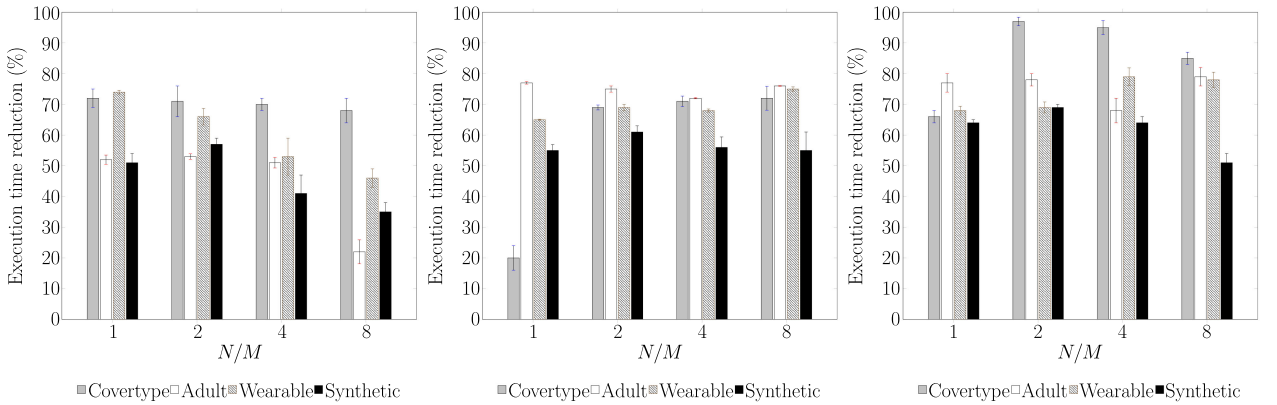
5.1 Evaluation Methodology

The proposed method was compared to the Ranger Framework [19] which is widely used and referenced in literature. Ranger is chosen as a reference method since it is popular and already integrates optimizations that aim to reduce memory footprint when building RF. More details about the reference study are given in the related work section.

5.1.1 Experiments

Our objective with these experiments is to measure the execution time reduction performed by the proposed optimizations in comparison to state-of-the-art method, then, we evaluate the impact of parameters that are specific to each proposed optimization on the efficiency of the method. The experiments performed are the following.

- *Experiment 1:* In this experiment, we measure the overall performance of the proposed strategy and compare it to the reference method. We also show the impact of each optimization on the overall performance for our strategy. We do so for different memory configurations (N/M). N/M values used for this experiment are $\{1, 2, 4, 8\}$, the number of decision trees to build was set to 25.
- *Experiment 2:* The objective of this experiment is to analyze the execution time reduction performed by the first optimization, that is data-set reorganization with the increase of the number of trees in the RF. To perform this experiment, we set the memory constraint to $N/M = 4$ and vary the number of decision trees to build T such as $T = \{25, 50, 100, 150\}$.
- *Experiment 3:* In order to check whether the prefetching has an effect on the obtained results, we measured execution time reduction obtained by data-set reorganization as compared to Ranger Framework on "Adult" data-set with enabled and disabled prefetching.
- *Experiment 4:* The objective of this experiment is to evaluate the efficiency of the second optimization, that is on-demand data access, for each scenario in terms of execution time reduction. To do so, for both the proposed method and the traditional method, we performed the following: 1) we identified for



(a) Optimization 1 (b) Optimization 2 (c) Combination of optimizations 1 and 2

Fig. 6. Experiment 1 results.

each node the scenario to use in terms of memory space, 2) we summed the execution times per scenario for the on-demand data access optimization and the reference method, 3) we computed and compared the average execution time for each scenario under the following memory constraints $N/M = \{1, 2, 4, 8, 16\}$. The used data-set in this experiment is "Wearable" (see Table 7).

- *Experiment 5:* In this experiment, our objective is to evaluate the performance of the second optimization compared to the reference method for different number of features. To do so, we used the same volume of learning data while varying the number of features: $d = 16, 32, 64$ and 128. For each value of d, we varied the memory constraints such as N/M takes the following values: $1, 2, 4,$ and 8.
- *Experiment 6:* The objective of this experiment is to estimate the energy reduction performed by the data-set reorganization of RaFIO as compared to Ranger Framework. We used pyJoules toolkit [28], that measures the energy footprint of a piece of code. The experiment was run using "Adult" data-set.
- *Experiment 7:* In order to check that running our experiments on a virtual machine does not have impact on the obtained results, we measured the execution time reductions with "Adult" data-set on another physical machine which features are the following: Dell Inspiron 3543 with an Intel Core i5-5200U, 8GB RAM and 512GB HDD. We also ran the same measures on a Linux virtual machine hosted in this physical one and compared the obtained results.
- *Experiment 8:* In order to check for the efficiency of RaFIO on an embedded device, we measured the execution time reduction of RaFIO as compared to Ranger on a BeagleBone Black [29] which features are an AM335x 1GHz ARM Cortex-A8 core, 512MB DDR3 RAM, 4GB 8-bit eMMC on-board flash storage. We plugged a 16GB SD card, 14 GB were used to store data-sets and 2 GB as a swap space[4]. The data-set "Adult" was used in this experiment. We ran the experiment with the Linux page cache read-ahead prefetching both enabled and disabled.

5.1.2 Experimental Setup

In the previously listed experiments, we measured the overall execution time, that is the whole tree building time (including processing and I/Os). The additional time for building the T_0 decision tree for the data-set reorganization optimization and the new data-set writing time were included in the execution time measure. Each measure was ran 5 times. In Section 5.2, we show the averages of the obtained results with a 5% confidence interval.

In our experiments, we used 3 real data-sets provided on UCI Machine Learning Repository [23], plus a synthetic one. Details about the used data-sets are given in Table 7. Note that Ecoli, Heart Failure, Wine and Poker data-sets were used to evaluate the similarity property in Table 5. In Experiment 4, we used 4 synthetic data-sets to vary the number of features. They were generated using the classification data-set generator provided in the Scikit-Learn Framework [26].

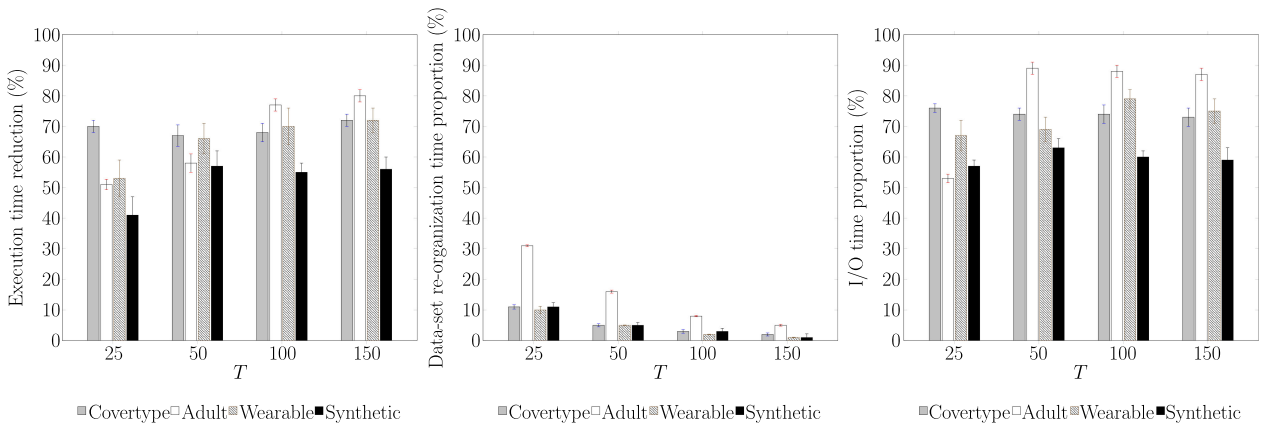
The measures were performed on a Linux Virtual Machine configured with 1 core and 4 GB of RAM memory, and 10 GB storage formatted with ext4 in order to simulate embedded platforms with varying memory constraints. The host machine is a Dell Latitude 5590 with an Intel Core i7-8650U, 8 GB RAM, and INTEL SSDSCKKF512G8 SATA 512 GB storage. The memory constraints (N/M) were applied using the *cgroup* mechanism [30]; it allows to limit the volume of memory that can be allocated to a process. The cgroup mechanism was used to emulate small memory footprints instead of a real edge device. This was done to explore a larger configuration space. The I/O block size is set to the default Linux I/O block size (4KB). This hardware setup is used for all the experiments.

5.2 Results and Discussion

5.2.1 Experiment 1

Figs. 6a, 6b and 6c respectively show the execution time reductions performed by the first optimization, the second and the combination of both of them in comparison to Ranger. We observe that the data-set reorganization (Optimization 1) is slightly more efficient with low N/M values.

4. both the SD card and eMMC flash memories were tested, they gave similar results. SD card configuration is presented in this paper



(a) Execution time reduction (b) Data-set writing time proportion (c) I/O time reduction

Fig. 7. Experiment 2 results.

In fact, even if the data-set organization gives a data-set organized such as all the elements of a block belong to the same nodes, when N/M is very high, the number of elements affected to a node is likely to be greater than M. Thus, the volume of I/Os when dividing the node is high. Optimization 2 performance allows more efficiency with the high values of N/M than Optimization 1. This is because on-demand data accesses allow to reduce the volume of I/Os since only the effectively needed data are loaded. Fig. 6c shows that the combination of the two optimizations allows to reach, on average, the highest execution time reduction. In fact, this combination makes the blocks kept in memory contain only effectively needed elements, which are ordered such as data locality is higher. We observe that for some measures, the combination of the two optimizations is less performing than Optimization 2. We believe that it is due to two main reasons: (1) the random nature of bootstrap creation and the feature selection, we think that the variability of the results is high, (2) we believe that in some rare cases, the data reorganization achieved may not fit with some specific feature selection patterns. In most cases, the overall optimization is higher than both optimizations, in some cases it is between those (some interference between optimizations), but in all measures, it was never lower than both optimizations.

Overall, the average execution time reduction performed respectively by Optimization 1, 2 and their combination are: 58%, 64% and 78%.

5.2.2 Experiment 2

Fig. 7a shows the execution time reductions performed by the first optimization (Data Reorganization) for different number of decision trees. We observe that execution time reduction ranges between 41 and 80%.

Fig. 7b shows the proportion of time necessary to reorganize the data-set (T_0 building and data-set writing times) compared to the overall random forest building time of the proposed method (T_0 and the T RF trees building time). We observe that the higher the number of trees in a random forest, the smaller this proportion.

Fig. 7c, represents the I/O time reduction achieved by the proposed method compared to the reference one. We observe that the I/O time reduction follows the curve of execution time reduction.

We observe a slight increase in the execution time reduction (Fig. 7a) between $T = 25$ and $T = 50$. This is because the proportion of time necessary for building decision tree T_0 becomes less important in comparison to the remaining trees building time. The remaining measurements show that the execution time reduction remains almost stable for each data-set, because the T_0 building time becomes negligible in comparison to the overall RF building time. Thus, since a data-set reorganization performs a given execution time reduction for one tree, the execution time reduction for all the RF is theoretically the same when neglecting T_0 building time, which is consistent with the obtained experimental results.

5.2.3 Experiment 3

Fig. 8 shows execution time reductions performed by RaFIO as compared to Ranger Framework when the prefetching is enabled and disabled. We observe that there is a negligible difference between the obtained results with and without prefetching enabled. In fact, we can explain this by the data read operations pattern which is not fully sequential.

5.2.4 Experiment 4

Fig. 9 shows the average processing time reduction performed by the second optimization (on-demand data accesses) compared to the reference method for each scenario. Fig. 10 shows the weighted execution time reduction;

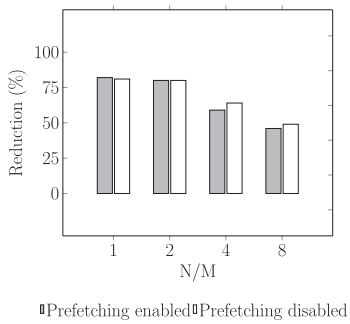


Fig. 8. Experiment 3 results.

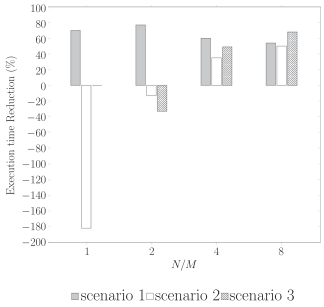


Fig. 9. Average execution time reduction performed.

it is obtained by multiplying the execution time reductions given in Fig. 9 by the proportions of number of nodes concerned per each scenario. For $N/M = 1$, since the memory is large enough to contain the whole data-set, scenario 3 never happens. Scenario 2 occurs with the topmost nodes that contain a large number of elements whilst memory work-space contains other structures used by the program. We observe from the figure that the strategy used by the proposed method in this scenario is less efficient than the reference method. This is because when the method loads useful data, it uses data location index and accesses all the data blocks whilst the amount of data to swap-in is low for the reference method since the data-set almost fits in memory. However, since there are very few nodes concerned by this scenario (only the topmost ones), this loss is negligible compared to the reduction achieved in scenario 1 as shown in Fig. 10.

When $N/M = 2$, we observe that the strategy used by our method with scenarios 2 and 3 is less efficient in comparison to the reference method. The reason is the same as with scenario 2 when $N/M = 1$, the volume of I/Os caused by chunks loading out-passes the swap occurring with the reference method. However, Fig. 10 shows that these losses are negligible in comparison to the reduction that it performs for scenario 1.

For $N/M = 8$, we observe that the three scenarios reduce the execution time by respectively up to 35%, 20% and 17% as compared to the traditional method.

In general, scenario 1 optimization is efficient when memory constraints are weak (a memory work-space size that is close to the bootstrap size) whereas scenarios 2 and 3 are efficient with stronger memory constraints (high values of N/M). The usefulness of scenario 2 and 3 is also related to the tree nature. The shallower the tree (with high memory constraints) the more effective scenario 2 and 3 are.

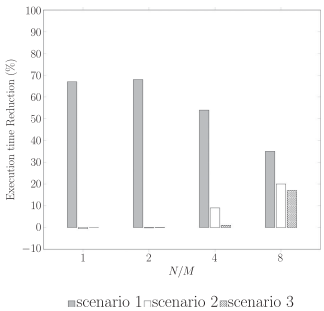


Fig. 10. Weighted execution time reduction.

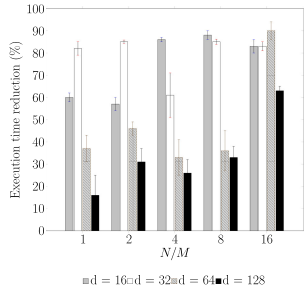


Fig. 11. Experiment 5 results.

5.2.5 Experiment 5

In this experiment, we give the execution time reductions obtained with our method compared to the traditional Ranger implementation for different number of feature values (d). The results are shown in Fig. 11. Overall, our method reduces the execution time by a factor laying between 15% - 90%. Two observations can be drawn from this experiment: *(1)* the proposed method is more profitable for a smaller number of features. This is because we have kept the same data-set file size for all the generated data-sets. As a consequence, when increasing the number of features, the number of elements decreases and consequently the depth of the built tree decreases too. Since the number of elements of the tree is small when it comes to a higher number of features, the number of nodes to build is smaller. Node split is the step that causes more I/O operations; thus, the execution time reductions are smaller than that for a lower number of features. *(2)* The second observation is that for the same number of features, the execution time reduction decreases between some successive values of N/M. This happens for $d = 16$ between $N/M = 1$ and $N/M = 2$; $d = 32$, $d = 64$ and $d = 128$ between $N/M = 2$ and $N/M = 4$. This is because for higher values of N/M, the on-demand data accesses method starts using the scenario 3; this implies to perform more I/O operations than those in scenarios 1 and 2.

5.2.6 Experiment 6

Fig. 12 shows the execution time, RAM main memory and CPU core energy reduction Energy and CPU Core energy reductions. We can observe that the energy optimization realized with RaFIO is correlated to the execution time reduction. The respective average reductions are 72 (for time), 74 (memory energy) and 72 (CPU core energy)%.

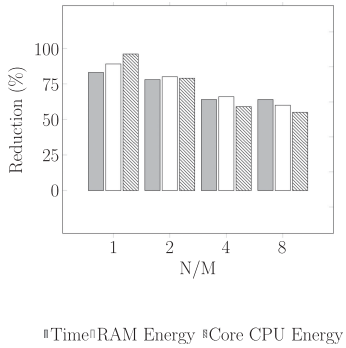


Fig. 12. Experiment 6 results.

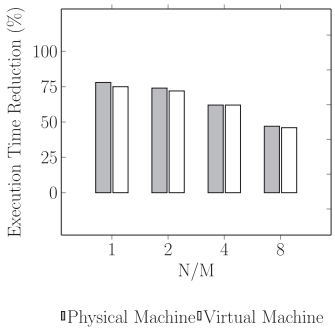


Fig. 13. Experiment 7 results.

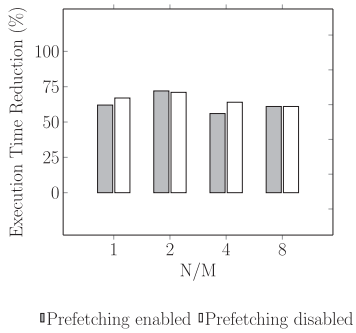


Fig. 14. Experiment 8 results.

5.2.7 Experiment 7

Fig. 13 shows the time reduction brought by our contribution compared to the traditional algorithm. We can see that the difference between running the experiment within and out of a virtual machine is negligible.

5.2.8 Experiment 8

The obtained results are given in Fig. 14. We observe that the execution time reductions are comparable to the ones obtained in Experiment 1. As in Experiment 3, we also observe that the page cache read-ahead prefetching does not have a significant impact on the results. Overall RaFIO proved to be as efficient on the tested embedded platform as on a virtual machine.

6 Related Work

Some studies have been conducted to optimize RF algorithm in order to tackle the fast expansion of data volumes in a big data context. We can coarsely classify these studies into the following categories.

The first one investigates parallel implementation at the forest level, that is decision tree buildings are performed in parallel [31].

The second category of optimizations acts on bootstrap sampling method. Authors of [32] and [33] propose methods based on bootstraps that contain less elements and sampled without replacement in comparison to the original method. This allows to reduce both computation and volume of data to span in comparison to the original method. In [34], the authors exploit the fact that the bootstrap is formed using a random sampling with replacement. In fact, this means that two consecutive bootstraps formed to build two trees are likely to contain common elements. Thus, the idea proposed by the authors is to pin in memory a proportion of elements between two consecutive tree buildings. This allows to elegantly reduce the volume of I/O operations by avoiding to load elements that were already in memory during previous tree building. This method can be coupled to our method in the case low N/M (near to 1) values. In the case of high N/M values, the I/Os occur at the node division level, thus, coupling this method to ours would increase even more the proportion N/M.

Another category of optimization proposed for RF acts at the decision tree building level. In [35], the authors proposed a hybrid depth-first, breadth-next decision tree building strategy. These two building modes are employed according to the number of elements per node. This method allows a better exploitation of cache memory. Our previous work [18] showed that the on-demand data access (Optimization 2) reduces decision tree building time by 10 to 70% in comparison to this method. Ranger Framework [19] also falls in this category of optimizations. It employs two different data structures that are respectively suitable for the nodes that have a high or low number of elements.

Other studies such as [36], [37] propose methods to optimize decision tree storage to reduce the memory footprint of the method, and thus, data movements between main memory and secondary storage, when performing inference. Such methods can be combined with RaFIO to store the built forest in an optimized manner.

On the one hand, different studies have shown the importance of a good spatial locality for diverse applications as for machine learning algorithms [38], distributed algorithms [39], GPU caches [40]. On the other hand, different machine learning techniques are used to predict data locality [39]. Our objective in this study is to take advantage of decision tree knowledge (the ML algorithm itself) to enhance spatial locality. To the best of our knowledge, no similar method have been proposed in the case of Random Forests. We combine this spatial locality enhancement with an on-demand data access to eliminate useless data accesses.

7 Conclusion

In this paper, we proposed an I/O-aware Random Forest algorithm. It is motivated by the fact that training a random forest in a memory constrained environment causes a substantial I/O volume. Our experiments show that by building a decision tree when the volume of the data-set is 8 times the volume of available memory, the I/O time represents 88% of the overall building time.

The proposed algorithm relies on two principles : *(1)* Data reorganization, the objective is to deduce from the first built tree which data are likely to be accessed within the same time window. This information is used to write a new data-set to enhance the spatial locality when building the remaining decision trees. *(2)* On-demand data accesses, the objective of this optimization is to remove the assumption that the data-set is memory resident, thus, avoiding useless data swaps to the secondary storage. In the proposed algorithm, the data is accessed at the decision tree node level, such as to only load data that is effectively needed. The benefit of our contribution depends on the performance ratio

between the main memory and the secondary storage used, which is always high, even with optimized storage devices.

Our experiments show that the data-set reorganization allows to reduce execution time by 63% on average when the volume of data to process is 4 times the available memory work-space. On-demand data accesses optimization allows to reduce a decision tree building time by 71% on average in comparison to the state-of-the-art method evaluated. The combination of the two optimizations allows to reduce execution time by nearly 75% in comparison to the reference method.

As a future work, we aim to apply the data-set reorganization to other ensemble learning methods, since they are based on training multiple models on the same data-set. In addition, the on-demand data access paradigm instead of a memory resident data-set assumption can be generalized to other machine learning algorithms.

REFERENCES

[1] D. Reinsel, J. Rydning, and J. F. Gantz, "Worldwide global datasphere forecast, 2020–2024: The COVID-19 data bump and the future of data growth," Apr. 2020. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=US44797920

[2] H. Abbas et al., "Special session: Embedded software for robotics: Challenges and future directions," in *Proc. Int. Conf. Embedded Softw.*, 2018, pp. 1–10.

[3] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, "Machine learning for Internet of Things data analysis: A survey," *Digit. Commun. Netw.*, vol. 4, no. 3, pp. 161–175, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S235286481730247X

[4] T. Muhammed, R. Mehmood, A. Albeshri, and I. Katib, "UbeHealth: A personalized ubiquitous cloud and edge-enabled networked healthcare system for smart cities," *IEEE Access*, vol. 6, pp. 32 258–32 285, 2018.

[5] P. C. M. Arachchige, P. Bertok, I. Khalil, D. Liu, S. Camtepe, and M. Atiquzzaman, "A trustworthy privacy preserving framework for machine learning in industrial IoT systems," *IEEE Trans. Ind. Informat.*, vol. 16, no. 9, pp. 6092–6102, Sep. 2020.

[6] X. Li, J. Tan, A. Liu, P. Vijayakumar, N. Kumar, and M. Alazab, "A novel UAV-enabled data collection scheme for intelligent transportation system through UAV speed control," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 4, pp. 2100–2110, Apr. 2021.

[7] N. Kukreja et al., "Training on the edge: The why and the how," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2019, pp. 899–903.

[8] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey," *Electronics*, vol. 8, no. 11, 2019, Art. no. 1289. [Online]. Available: https://www.mdpi.com/2079–9292/8/11/1289

[9] O. Mutlu, S. Ghose, and R. Ausavarungnirun, "Recent advances in overcoming bottlenecks in memory systems and managing memory resources in GPU systems," 2018, *arXiv:1805.06407*.

[10] G. S. Nagpal, G. Singh, J. Singh, and N. Yadav, "Facial detection and recognition using OpenCV on raspberry Pi zero," in *Proc. IEEE Int. Conf. Adv. Comput. Commun. Control Netw.*, 2018, pp. 945–950.

[11] A. Alarcón-Paredes, V. Francisco-García, I. P. Guzmán-Guzmán, J. Cantillo-Negrete, R. E. Cuevas-Valencia, and G. A. Alonso-Silverio, "An IoT-based non-invasive glucose level monitoring system using raspberry Pi," *Appl. Sci.*, vol. 9, no. 15, 2019, Art. no. 3046.

[12] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the Internet of Things," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1935–1944.

[13] J. Shotton, T. Sharp, P. Kohli, S. Nowozin, J. Winn, and A. Criminisi, "Decision jungles: Compact and rich models for classification," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 234–242.

[14] K. H. Lee and N. Verma, "A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals," *IEEE J. Solid-State Circuits*, vol. 48, no. 7, pp. 1625–1637, Jul. 2013.

[15] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *J. Ind. Inf. Integr.*, vol. 6, pp. 1–10, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2452414X17300043

[16] L. Breiman, "Random Forests," *Mach. Learn.*, vol. 45, pp. 5–32, 2001.

[17] C. Zhang and Y. Ma, *Ensemble Machine Learning: Methods and Applications*. Berlin, Germany: Springer, 2012.

[18] C. Slimani, C.-F. Wu, Y.-H. Chang, S. Rubini, and J. Boukhobza, "RaFIO: A random forest I/O-aware algorithm," in *Proc. 36th ACM Symp. Appl. Comput.*, 2021, pp. 521–528.

[19] M. Wright and A. Ziegler, "ranger: A fast implementation of random forests for high dimensional data in C++ and R," *J. Statist. Softw.*, vol. 77, pp. 1–17, 2017.

[20] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Sebastopol, CA, USA: O'Reilly, 2005.

[21] C. Guyeux, S. Chrétien, G. B. Tayeh, J. Demerjian, and J. Bahi, "Introducing and comparing recent clustering methods for massive data management in the Internet of Things," *J. Sensor Actuator Netw.*, vol. 8, no. 4, 2019, Art. no. 56. [Online]. Available: https://www.mdpi.com/2224–2708/8/4/56

[22] J. Santos and M. Embrechts, "On the use of the adjusted rand index as a metric for evaluating supervised classification," in *Proc. Int. Conf. Artif. Neural Netw.*, 2009, pp. 175–184.

[23] D. Dua and C. Graff, "UCI machine learning repository," 2019. [Online]. Available: http://archive.ics.uci.edu/ml

[24] D. Rosenberg, "Bagging and random forests," Mar. 2015.

[25] M. Rodriguez et al., "Clustering algorithms: A comparative approach," *PLoS One*, vol. 14, 2016, Art. no. e0210236.

[26] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[27] J. Boukhobza and P. Olivier, *Flash Memory Integration: Performance and Energy Issues*, 1st ed. Amsterdam, The Netherlands: Elsevier, 2017. [Online]. Available: http://www.sciencedirect.com/science/book/9781785481246

[28] Welcome to Pyjoules's documentation!, "Welcome to pyJoules's documentation! - pyJoules 0.2.0 documentation.," 2021. Accessed: Nov. 3, 2022. [Online]. Available: https://pyjoules.readthedocs.io/en/latest/index.html

[29] G. Coley, "System reference manual • Beagleboard/Beaglebone-Black Wiki," *GitHub*, 2021. Accessed: Nov. 3, 2022. [Online]. Available: https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual

[30] P. Menage, "Cgroups," 2004. Last Accessed: Jul. 28, 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[31] R. Genuer, J.-M. Poggi, C. Tuleau-Malot, and N. Villa-Vialaneix, "Random forests for big data," *Big Data Res.*, vol. 9, pp. 28–46, 2017.

[32] P. Bickel, F. Götze, and W. V. Zwet, "Resampling fewer than n observations: Gains, losses, and remedies for losses," *Statistica Sinica*, vol. 7, no. 1, pp. 1–31, 1997.

[33] A. Kleiner, A. Talwalkar, P. Sarkar, and M. Jordan, "A scalable bootstrap for massive data," *J. Roy.Statist. Soc. Ser. B Statist. Methodol.*, vol. 76, no. 4, pp. 795–816, 2014.

[34] Y. T. Ho, C. Wu, M. Yang, T. Chen, and Y. Chang, "Replanting your forest: NVM-friendly bagging strategy for random forest," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp.*, 2019, pp. 1–6.

[35] A. Anghel, N. Ioannou, T. P. Parnell, N. Papandreou, C. Mendler-Dünner, and H. Pozidis, "Breadth-first, depth-next training of random forests," 2019, *arXiv:1910.06853*.

[36] M. Madhyastha, K. Lillaney, J. Browne, J. Vogelstein, and R. Burns, "PACSET (packed serialized trees): Reducing inference latency for tree ensemble deployment," 2020, *arXiv:2011.05383*.

[37] M. Madhyastha, K. Lillaney, J. Browne, J. T. Vogelstein, and R. Burns, "BLOCKSET (block-aligned serialized trees) reducing inference latency for tree ensemble deployment," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2021, pp. 1170–1179.

[38] I. Chakroun, T. V. Aa, and T. Ashby, "Guidelines for enhancing data locality in selected machine learning algorithms," *Intell. Data Anal.*, vol. 23, no. 5, pp. 1003–1020, 2019.

[39] E. Kayraklioglu, E. Favry, and T. El-Ghazawi, "A machine learning approach for productive data locality exploitation in parallel computing systems," in *Proc. IEEE/ACM 19th Int. Symp. Cluster Cloud Grid Comput.*, 2019, pp. 361–370.

[40] S. Lal and B. Juurlink, "A quantitative study of locality in GPU caches," in *Proc. Int. Conf. Embedded Comput. Syst.*, 2020, pp. 228–242.

Camélia Slimani received the engineering degree (with Hons.) in computer science from Ecole nationale Supérieure d'Informatique (ESI), Algiers, Algeria, the MSc degree (with Hons.) in computer science from the University of Bretagne Occidentale, France, in 2018, and the PhD degree in computer science from the University of Bretagne Occidentale, France, in 2022. She is now a postdoctoral researcher with ENSTA Bretagne, France. Her research interests include the optimization of machine learning algorithms in terms of I/Os in the context of memory constrained environments.

Chun-Feng Wu (Member, IEEE) received the MS degree from the Department of Computer Science, National Tsing-Hua University, in 2016, and the PhD degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2021. Currently, he is an assistant professor with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu, Taiwan. Previously, He was a postdoctoral scholar with the Department of Computer Science, Harvard University, Cambridge from 2021 to 2022. He served in R&D alternative service with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, from 2017 to 2021. His primary research interests include memory/storage systems, embedded systems, operating systems and the next-generation memory/storage architecture designs.

Stéphane Rubini received the graduate degree in electrical and computer engineering from the Ecole Nationale d'Ingénieurs de Brest (ENIB), in 1991, and the PhD degree in computer science from the University of Rennes I, in 1995. Currently, he is an associate professor of computer science with the Department of Computer Science, University of Brest, France. He is part of the Lab-STICC Laboratory. His research interests comprise the design of dedicated computer architectures on reconfigurable hardware targets, and most generally the matching between software algorithms and hardware. He has contributed to the hardware/software development of several processing machines for filtering genomic banks. He is currently working on the modeling and exploitation of complex memory architectures in the context of embedded systems.

Yuan-Hao Chang (Senior Member, IEEE) received the PhD degree in computer science from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei. He is currently a research fellow with the Institute of Information Science, Academia Sinica, Taipei. He is a senior member of the ACM. His research interests include memory/storage systems, operating systems, embedded systems, and real-time systems.

Jalil Boukhobza (Senior Member, IEEE) received the electrical engineering (with Hons.) degree from the Institut Nationale d'Electricite et d'electronique (I.N.E.L.E.C) Boumerdes, Algeria, in 1999, and the MSc and PhD degrees in computer science from the University of Versailles, France, in 2000 and 2004, respectively. He is a professor with the ENSTA-Bretagne, a French State Graduate, Post-Graduate and Research Institute. He was a research fellow with the PRiSM Laboratory (University of Versailles) from 2004 to 2006. He was an associate professor with the University Bretagne Occidentale, Brest, France, from 2006 to 2020 and is a member of Lab-STICC. He has also been working with the Technology Research Institute (IRT) bcom since 2013. His main research interests include storage system design, performance evaluation and energy optimization, and operating system design. He works on different application domains such as embedded systems, cloud computing, and database systems.