Memory Mapping and Parallelizing Random Forests for Speed and Cache Efficiency

Eduardo Romero
romerogainza.1@osu.edu
The Ohio State University
USA

Angela Li
li.10011@osu.edu
The Ohio State University
USA

Christopher Stewart
cstewart@cse.ohio-state.edu
The Ohio State University
USA

Kyle Hale
khale1@iit.edu
Illinois Institute of Technology
USA

Nathaniel Morris
morris.743@osu.edu
The Ohio State University
USA

ABSTRACT

Memory mapping enhances decision tree implementations by enabling constant-time statistical inference, and is particularly effective when memory mapped tables fit in processor cache. However, memory mapping is more challenging when applied to random forests—ensembles of many trees—as the table sizes can easily outstrip cache capacity. We argue that careful system design for parallel and cache efficiency can make memory mapping effective for random forests. Our preliminary results show memory-mapped forests can speed up inference latency by a factor of up to 30×.

ACM Reference Format:
Eduardo Romero, Angela Li, Christopher Stewart, Kyle Hale, and Nathaniel Morris. 2021. Memory Mapping and Parallelizing Random Forests for Speed and Cache Efficiency. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3458744.3474052

1 INTRODUCTION

Memory mapping, depicted in Figure 1, is a depth-first alternative to breadth-first decision tree traversal. In this approach, a representation of each path in the tree is encoded as a lookup table address and the corresponding leaf-node result is stored in the table at that address. By default, memory mapping uses one table to represent all outcomes captured by a decision tree, so each table entry (i.e., address) must specify a value for every feature in the tree. In Figure 1, there are four paths through the tree but eight possible combinations of values for features.

AI inference on memory-mapped trees proceeds as follows: First, features from the input data are sorted and converted to a table address, then the address is used to lookup a classification result. Compared to breadth-first tree traversal, memory mapping avoids conditional control flow, thus alleviating pressure on the branch predictor and offering more opportunities for the CPU's prefetcher.

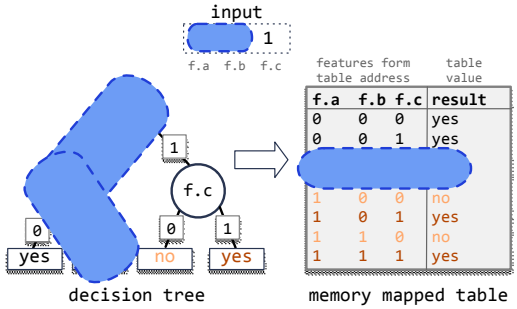

Figure 1: Memory mapping encodes each path in a tree as a table lookup address that directly maps to a classification result. Features that are irrelevant for particular paths result in "don't cares" in the table address.

However, memory mapping increases storage demands and potentially increases cache misses. These storage demands unfortunately increase exponentially with the size of the feature space and, generally, complex data sets will necessitate a large feature space for sufficient classification accuracy. Furthermore, ensemble models, such as random and deep forests [22] use many decision trees to achieve high accuracy and, as a result, further increase storage demands. Such storage demands can be mitigated by partitioning the memory-mapped table, considering subsets of features in each table. A "dictionary" that assigns samples to partitions based on the value of certain features can reduce storage demands, however, increasing the number of partitions (dictionary size) also incurs some of the same inefficiencies as bread-first traversal such as branching.

Forest Packing [6] creates tables from subsets of trees in the forest and reduces storage demands by avoiding repetition of leaf nodes and by using memory mapping *only* for frequently used (hot) paths. These paths can be retrieved from processor cache to reduce the number of tree traversals. However, in forests that comprise many trees and features, input data is unlikely to match hot paths. Ranger [21] batches multiple inputs together for inference, improving execution efficiency. However, AI inference and local explanation workloads increasingly demand low response times and cannot wait to batch queries. Further, while both Forest Packing [6] and Ranger [21] include efficient C++ implementations, neither consider bit-wise data structure optimizations.

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3474052

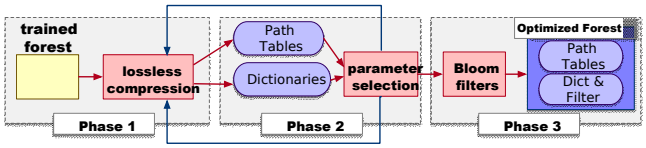

Figure 2: Cache-aware data management (1) compresses a given trained forest, (2) explores parameters to reduce storage demand and latency and (3) uses filters to reduce memory lookups.

For this workshop paper, we outline a novel system design for managing memory-mapped random forests. Our approach transforms the data layout *after* a breadth-first forest is fully trained, producing a memory-mapped forest that executes AI inference in microseconds, orders of magnitude faster than competing, breadth-first approaches used in practice.

Specifically, we make the following contributions:

- We present a method to manage storage and processing demands for memory-mapped forests using clustering and bloom filters.
- We present a method for exploring parameters, searching for inference latency given a forest and available cores.
- In early work, we show that our approach significantly reduces response time for AI inference relative to state-of-the-art methods.

The remainder of this paper is structured as follows: Section 2 presents our design. Section 3 presents empirical evaluations of our approach. Section 4 describes related work and Section 5 draws conclusions.

2 DESIGN

For our approach we focus on binary trees where nodes are features and edges indicate values associated with a feature (0 or 1). A matching path from root to leaf nodes means that each feature-value pair in the path is also present in the input data. Each tree has exactly one matching path for a given input. The terminal node (leaf node) of the path represents the inference result (classification). Results from each tree are aggregated to produce a final classification.

Our approach: Figure 2 presents our memory mapping approach for forests that safely[1] transforms trees, manages storage demand and speeds up inference and explanation workloads. Our approach accepts three inputs: a trained forest, number of available CPU cores, and cache capacity of each core. The output is a collection of structures that we call *dictionaries*[2] and memory-mapped trees ready for inference and corresponding to the original forest.

Our approach consists of three phases. Phase 1 splits the entire forest into several tree paths and clusters similar paths (among all trees) into tables to reduce storage demands. Phase 2 evaluates the outcome of Phase 1 against the expected size of each table (storage) and of each dictionary (latency) and searches for parameters that reduce inference latency. Phase 3 speeds up path matching by quickly filtering out tables that comprise only non-matching paths, thereby avoiding unnecessary memory accesses.

[1]Informally, *safety* means that transformations preserve classification results for all inputs.

[2]These are not traditional dictionaries in the sense of associative maps with $O(1)$ lookup.

2.1 Phase 1: Clustering and Compression

Recall from Figure 1 that naïve memory mapping forms a table address from *every* feature present in a tree and the possible values that feature can take on (here only 0 or 1). In that figure, even though the highlighted path formed by $(f_a, 0) \rightarrow (f_b, 1)$ does not include f_c, f_c still must be present in the address formed from the features, since addresses (*path table* indexes) are a fixed length. This results in two duplicate entries in the path table with the same result, where f_c is treated as a "don't care" in the address. This approach wastes space and inflates storage demand exponentially, since it necessitates 2^n table entries for n binary features, thus making memory mapping untenable for forests comprising complex trees. Note here that n comprises *all* distinct features used in *all* trees in the forest (not all trees use the same features), so it can grow quite quickly. We propose an alternative memory-mapping approach to manage storage demands. The key insight we leverage for our approach is that several trees within a forest may share paths, presenting an opportunity for compression.

Figure 3 shows how our approach transforms an input random forest comprising two decision trees into a set of data structures we can use for fast, cache-friendly inference. First, paths (consisting of a series of feature/value pairs) for each tree in the forest are enumerated and sorted lexicographically Figure 3(1). The sorted paths from the trees are then merged into a single, sorted list of paths *for the entire forest* Figure 3(2). Clusters are formed by incrementally adding paths from this sorted list. This is done until a tunable threshold is met.

The important thing to note about the clusters is that, at this stage, each cluster will have its own *compressed* memory-mapped path table (as opposed to a single large table for the whole forest). We can see the path clusters superimposed on the original forest (3) in the third column of Figure 3. By design, *each* cluster shares a unique set of feature/value pairs common to all paths in that cluster. In the figure, $(a, 0)$ is common for the green cluster, $(a, 1)$ is common for the yellow cluster, and $(h, 1)$ is common for the blue cluster. The commonality of these features *within the cluster* allows us to extract them out and use them as an identifier that determines membership of inputs in cluster-specific path tables Figure 3(5). This is accomplished using our "dictionary" data structure Figure 3(4). When an input vector arrives, its features are compared against entries in the dictionary to match it to an appropriate path table. For example, an input of `0100` would match the first dictionary entry (storing the common pair $(a, 0)$). The lookup would then be directed to the first (green) path table. Note that now we only have ten path table entries and three dictionary entries; "don't care" entries are eliminated entirely. This is in contrast to the 16-entry table that would be used in the naïve approach, shown on the right side of the figure. After clustering, and compression, our approach outputs a dictionary in which every entry maps to a unique path table. However, after this stage these tables must be recombined into one single table to help identifying false positives (details in §2.2). After recombination, this stage has one dictionary and one

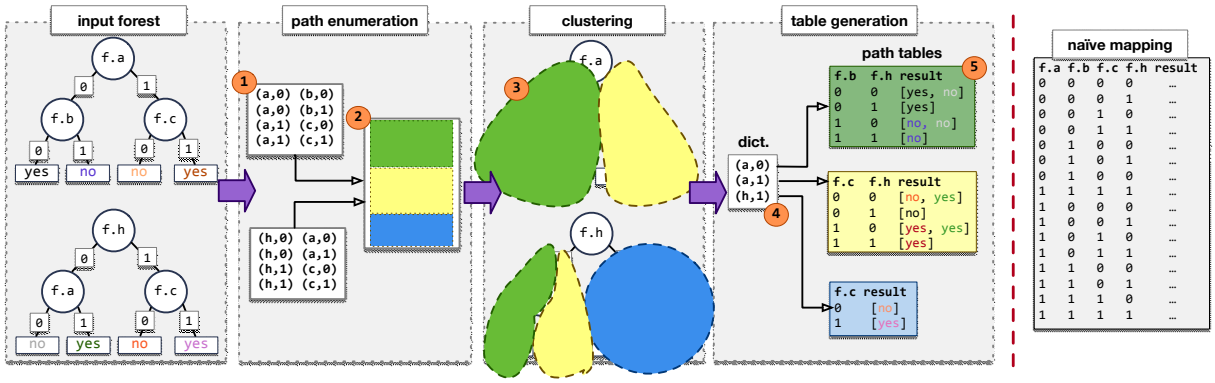

Figure 3: Compression of input forests

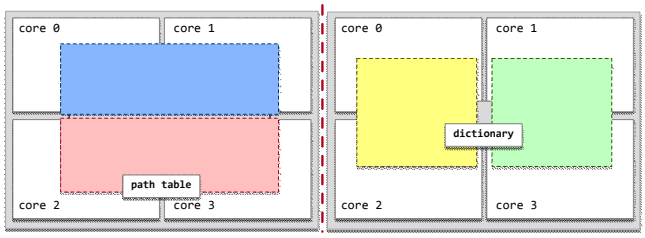

Figure 4: Four core example of parallelization. This example shows two partitions of the path table in and two partitions of the dictionary.

path table for the entire forest. The dictionary entries now point to subsets of entries in the unique memory-mapped table.

2.2 Phase 2: Parameter Selection

As described in the previous section, our approach makes use of dictionaries to reduce the storage demands of memory mapping. However, during inference each input must be compared to *every* entry in the dictionary. While this lookup does not require memory accesses and uses fast bit-wise operations in lieu of branching, a large number of dictionary entries can become the bottleneck during inference, particularly if the path table already fits in cache (fast memory accesses). Given the inverse proportionality between number of entries in the dictionary and the size of the table, the clustering threshold described in the previous section, which controls dictionary size, must be carefully tuned.

Furthermore, the possibility of scaling to multiple cores adds even more complexity. Figure 4 shows one possible division of a forest across four cores using our approach. In this example, both the path table and the dictionary are split into two partitions each. Partitioning path tables requires running two copies of the dictionary. For any input, a comparison is made with the key features of the dictionary entries. If there is a match, a memory lookup is attempted (a binary sequence is generated with the mapped features and the address is mapped to an index of the table using the entry ID), if the address searched is within the partition of the path table that corresponds to that core, a result is computed. Path table partitioning decreases storage demand per individual core but may only indirectly affect latency. Dividing the path table only improves latency if cache misses have a big impact on performance of the specific workload. Figure 4 also shows the division of the dictionary. When the dictionary is partitioned, a copy of each path table is made. During inference, each core compares the input with the key features of the available dictionary entries, and performs the corresponding memory accesses. The partitioning of the dictionary directly impacts latency, but the overhead of aggregating results must be considered. Different values for inter-core communication latency and different methods for result aggregation can lead to different partition strategies. The combination of path table size, dictionary size, the number of splits of these data structures across cores, and the overhead of partitioning complicates modeling the ideal strategy given a workload. Our algorithm searches the space given by these parameters by exploring different parameter settings, executing and selecting those partitioning strategies that lead to best results.

2.3 Phase 3: Improving Path Table Selection

The use of dictionaries to compress a path table renders many entries in the dictionary irrelevant for a particular input. For the dictionaries to be effective, the decision to access a path table given the features in a dictionary entry must be a fast one. To solve this, we use bloom filters [3], a probabilistic data structure used to query set membership. Unlike perfect hashing that correctly labels inputs and non-members, bloom filters can report false positives; some non-members can be labeled members but members are never labeled as non-members (i.e. no false negatives). When non-members greatly outnumber members, like in our compressed forest with many dictionary entries, bloom filters can afford fast, resource-lean membership lookups.

During inference, for every dictionary entry, we use bit-wise operations to simultaneously decide if the dictionary entry is relevant to the input, and compute the location of the path table that would be accessed if the dictionary entry is relevant to the input. As shown in Figure 3, our dictionaries distinguish between common and uncommon features.

Given input features, we use common and uncommon features to create a binary sequence representing a mapping of the input in the features present in the dictionary entry. Then, we use a bit-mask

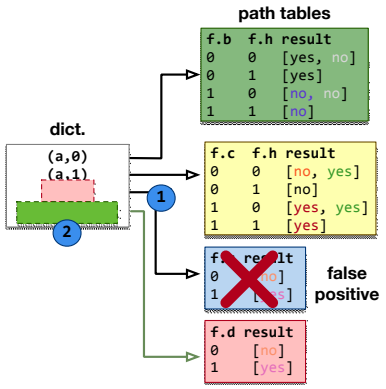

Figure 5: False positives.

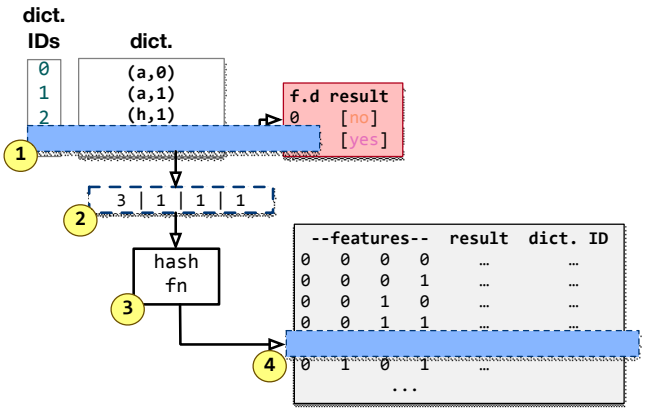

Figure 6: Recombining tables.

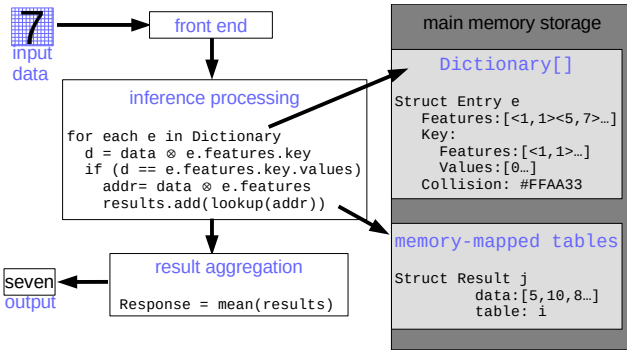

Figure 7: Workflow for AI inference using our approach. This example shows digit recognition where input data is a 28 x 28 image.

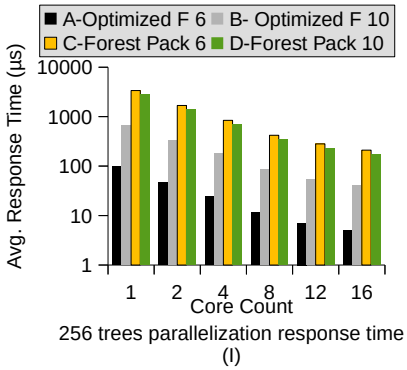

Figure 8: Log scale comparison of execution time in our optimized forests against Forest Packing.

representing the common features to decide membership of the input in the table mapped by that entry.

The approach described above suffices when the dictionary entries map entire subtrees. However, because we allow for grouping of paths from different trees or subtrees, and because of the greedy algorithm for clustering, false positives are possible. To correct for this, we use the entry ID of the dictionary entry when hashing the binary sequence into the recombined memory mapped table.

3 EVALUATION

In this section, we compare our results to Forest Packing. We start with a Forest trained by Scikit-learn. Our evaluation uses Python scripts for front-end processing. The front-end communicates to inference processing engines on a UNIX domain socket. We implemented our design in C/C++, allowing for low level control of data layout and bit operations.

We tested our approach on multiple processors with varying cache size and clock frequency. Unless noted otherwise, we used an Intel Xeon(R) E5-2650 v4 @ 2.20GHz with 12 cores, 30 MB of LLC and 132 GB of memory. We also used (1) Intel Xeon(R) E5-2620, (2) Intel Xeon E5504, and (3) AMD Ryzen 5 3500. Our default platform runs stock Linux kernel version 3.10.0, Python 3.6.8 and Scikit-learn 20.

For our comparison, we used the MNIST with 10K samples for testing. Input data to MNIST are 28×28 images of a handwritten digit. Each pixel is a feature (784 in total). The output classifies the image as a digit (0–9).

Figure 8 examines response time across number of cores. We use the experiments and results reported in [6] on MNIST to compare our results with Forest Packing with multiple cores available. The average response time for our design is measured by the running 10000 samples on MNIST dataset without batching. Both forests are comprised of 256 trees. Two versions of each forest are compared, with the maximum depth of each tree set to 6 and 10 respectively. On those two settings, we achieve speedups of 34× and 4×.

Additionally, we observed 40× speedup for our approach over Scikit-Learn on our experiments while also achieving a reduction of both Branch Misses and Cache Misses of 99%.

4 RELATED WORK

Recent work has advanced explainable ML models [2, 7, 8, 10, 13, 16, 17] and shown that decision trees have unique properties for explain-ability [11, 12]. Further, edge computing applications, e.g., unmanned aerial vehicles for crop scouting [4, 5], federated learning with active cameras [19], and interactive classrooms [14, 15], increase the demand for explainable AI for end users. While trees within forests are explainable, random forests are more complex and require additional structure to track salient features. There is also an innate performance to quality trade off with tree height, improving accuracy but making forests more complex [9].

Forest Packing is the closest related work [6]. Browne et al. also seek to speed up response time for AI inference services. Their approach implicitly memory maps data by storing trees in depth-first order. Nodes in the same path are loaded into the same cache line and checked against input data. Paths are organized by how frequently they are accessed in testing data, prioritizing cache lines for hot paths. However, testing data may not reflect the statistical path distribution observed when a forest runs inference as a service. However, for complex data used on a wide range of services, hot paths will likely differ. By explicitly memory mapping paths, our approach forests can cache whichever paths are used most frequently by a service. Another key difference is that our approach does not follow pointers from node to node. By using a dictionary, our approach reduces branch mispredictions and separates compute and cache capacity concerns. Further, our approach is less dependent on system scheduling and instruction-level parallelism.

Prior work has explored deterministic finite automaton (DFA) on custom hardware [18, 20] and for processing XML and JSON files [1]. These efforts share our goals of efficiently using processor resources for DFA workloads. our approach includes unique implementation optimizations targeted at random forests.

5 FUTURE

In this workshop paper, we presented an approach to optimize trained decision forests, reducing inference latency by compressing via memory mapping. Our approach improve an otherwise inefficient process by considering cache allocation and many core processor configurations and by using efficient data structures to minimize branch mispredictions and memory lookups that negatively affect inference latency. Our approach includes a novel method of compressing forests that combines memory mapping with clustering to optimize the number of memory lookups and storage demands. In future work, we will further explore the system design, providing more rigorous models to guide optimizations. We will also generalize our method for searching the space of hyper-parameters to find minimal latency.

In this work, we present early and very promising results comparing our approach against state-of-the-art decision forest libraries. We achieved much faster inference speeds on the widely used MNIST benchmark. However, in future work, we would like to conduct a more detailed evaluation that considers multiple data sets, inference on a wide range of heterogeneous, parallelized systems. Also, we would like to further explore explainability benchmarks, an increasingly critical component of human-machine interactions in edge and Internet-of-Things systems. We hypothesize that memory-mapped forests can support single-pass explainability, greatly improving throughput for this workload compared to any other widely used machine learning platform, including neural networks.

REFERENCES

[1] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A scalable In-SRAM architecture for pushdown automata. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 921–932.

[2] Umang Bhatt, Alice Xiang, Shubham Sharma, Adrian Weller, Ankur Taly, Yunhan Jia, Joydeep Ghosh, Ruchir Puri, José MF Moura, and Peter Eckersley. 2020. Explainable machine learning in deployment. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. 648–657.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Jayson Boubin, Naveen Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. 2019. Managing Edge Resources for Fully Autonomous Aerial Systems. In *ACM Symposium on Edge Computing*.

[5] Jayson Boubin, John Chumley, Christopher Stewart, and Sami Khanal. 2019. Autonomic Computing Challenges in Fully Autonomous Precision Agriculture. In *IEEE International Conference on Autonomic Computing*.

[6] James Browne, Disa Mhembere, Tyler M Tomita, Joshua T Vogelstein, and Randal Burns. 2019. Forest Packing: Fast Parallel, Decision Forests. In *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM, 46–54.

[7] Seunghoon Hong, Tackgeun You, Suha Kwak, and Bohyung Han. 2015. Online tracking by learning discriminative saliency map with convolutional neural network. In *International conference on machine learning*. 597–606.

[8] Sarthak Jain and Byron C Wallace. 2019. Attention is not explanation. *arXiv preprint arXiv:1902.10186* (2019).

[9] Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. 2017. Obtaining and Managing Answer Quality for Online Data-Intensive Services. In *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*.

[10] Emre Kıcıman and Matthew Richardson. 2015. Towards decision support and goal achievement: Identifying action-outcome relationships from social media. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 547–556.

[11] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence* 2, 1 (2020).

[12] Ioannis Mollas, Grigorios Tsoumakas, and Nick Bassiliades. 2019. LionForests: Local Interpretation of Random Forests through Path Selection. *arXiv preprint arXiv:1911.08780* (2019).

[13] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley:. 2018. Model-driven Computational Sprinting. In *ACM European Conference on Computer Systems*.

[14] Rashmi Rao, Christopher Stewart, Arnuflo Perez, and Siva Renganathan. 2018. Assessing Learning Behavior and Cognitive Bias from Web Logs. In *IEEE Frontiers in Education*.

[15] Siva Renganathan, Christopher Stewart, Arnulfo Perez, Rashmi Rao, and Bailey Braaten. 2017. Preliminary Results on an Interactive Learning Tool for Early Algebra Education. In *IEEE Frontiers in Education*.

[16] Eduardo Romero, Christopher Stewart, and Nathaniel Morris. 2019. Fast Inference Services for Alternative Deep Learning Structures. In *ACM Symposium on Edge Computing*.

[17] C. Stewart, K. Shen, A. Iyengar, and J. Yin. 2010. EntomoModel: Understanding and Avoiding Performance Anomaly Manifestations. In *IEEE MASCOTS*.

[18] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards machine learning on the automata processor. In *International Conference on High Performance Computing*. Springer, 200–218.

[19] Blesson Varghese, Nan Wang, David Bermbach, Cheol-Ho Hong, Eyal de Lara, Weisong Shi, and Christopher Stewart. 2020. A Survey on Edge Performance Benchmarking. *Comput. Surveys* (2020).

[20] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, et al. 2018. AutomataZoo: A modern automata processing benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 13–24.

[21] Marvin N Wright and Andreas Ziegler. 2015. ranger: A fast implementation of random forests for high dimensional data in C++ and R. *arXiv preprint arXiv:1508.04409* (2015).

[22] Zhi-Hua Zhou and Ji Feng. 2017. Deep forest. *arXiv preprint arXiv:1702.08835* (2017).