Boosted Race Trees for Low Energy Classification

Georgios Tzimpragos
UC Santa Barbara
gtzimpragos@cs.ucsb.edu

Advait Madhavan
National Institute of Standards and Technology & University of Maryland
advait.madhavan@nist.gov

Dilip Vasudevan
Lawrence Berkeley National Lab
dilipv@lbl.gov

Dmitri Strukov
UC Santa Barbara
strukov@ece.ucsb.edu

Timothy Sherwood
UC Santa Barbara
sherwood@cs.ucsb.edu

Abstract

When extremely low-energy processing is required, the choice of data representation makes a tremendous difference. Each representation (e.g. frequency domain, residue coded, log-scale) comes with a unique set of trade-offs — some operations are easier in that domain while others are harder. We demonstrate that race logic, in which temporally coded signals are getting processed in a dataflow fashion, provides interesting new capabilities for in-sensor processing applications. Specifically, with an extended set of race logic operations, we show that tree-based classifiers can be naturally encoded, and that common classification tasks can be implemented efficiently as a programmable accelerator in this class of logic. To verify this hypothesis, we design several race logic implementations of ensemble learners, compare them against state-of-the-art classifiers, and conduct an architectural design space exploration. Our proof-of-concept architecture, consisting of 1,000 reconfigurable Race Trees of depth 6, will process 15.2M frames/s, dissipating 613mW in 14nm CMOS.

CCS Concepts • **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Machine learning*; • **Hardware** → *Hardware accelerators.*

Keywords in-sensor processing, classification, decision trees, race logic.

ACM Reference Format:
Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. 2019. Boosted Race Trees for Low Energy Classification. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304036

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304036

1 Introduction

There is little need to provide motivation for hardware support for machine learning today. In recent years, we have seen an explosion of activity in everything, from vision and speech processing to control and medical diagnostics. All of this is driven, in part, by an excitement that machine learning has the potential to unlock new insights in applications from the largest data center to the smallest embedded system. If machine learning is the engine, then raw data is the fuel, and most approaches consume a great deal of it. However, in embedded applications, where the learning and sensing are close in both time and space, the exact type of data to be consumed is something that needs to be considered carefully.

Typically, a sensor gathers analog information from the physical world and then converts it into a conventional digital signal. For example, a camera captures incident photons and, through the photoelectric effect, uses their energy to guide the charging of a cell. The voltage on the cell is read out to an Analog-to-Digital Converter that converts the measured voltage into a stream of zeros and ones, which is then stored into a memory. While this binary-represented integer is perfectly efficient for storage as bits in a memory and for typical general purpose computing operations, it is unclear that this is the most efficient solution. We posit that there are other encodings that, while still capturing the relative values of the data to be encoded, are more efficient for our target application.

One such possible representation is pure analog signalling. There is a long history of machine-learning-like computing with analog devices. While pure analog design is always an option, there are at least two important reasons to consider alternatives. First, well-understood analog design rules always lag far behind digital rules in technology node. High density, high performance, low energy CMOS parts can be hard to achieve because of this gap. Second, while analog design in these aggressive technology nodes is certainly possible, noise concerns often drive analog designs to use larger gates than their digital counterparts. If we can get the analog-like efficient behavior, where the computation matches the capabilities of the underlying devices, while keeping the noise tolerance and layout simplicity of digital designs, we would be well-served.

One class of logic that attempts to fit this space in a more domain specific role is race logic [22, 24]. The key idea behind race logic is to encode values as a delay from some reference. All signals, unlike pure analog approaches, are supposed to be 0 or 1 at all times. However, the time at which $0 \longrightarrow 1$ transition happens encodes the value. Computations can then be based on the observation of the relative propagation times of signals injected into a configurable circuit. In prior work, it was shown that these techniques could efficiently solve a class of dynamic programming algorithms, and both synchronous and asynchronous versions have been evaluated [23, 24]. However, many open questions remain including both the limits of its computational abilities and the scope of its utility.

In this paper, we explore the intersection of the three paragraphs above: race logic as a sensor-friendly yet machine-learning-ready encoding. We argue that race logic fits nicely with temporally-coded sensing systems, such as 3D depth perception systems and dynamic vision sensors, where the time that a signal gets "excited" to a logic-level "1" depends on the magnitude of the sensor reading. Furthermore, we demonstrate that the structure of race logic makes it a natural encoding for decision tree-based approaches, and show how a programmable accelerator can support this family of machine learning methods seamlessly from a popular software framework down to synthesizable implementation.

To experimentally validate this hypothesis, we complete an end-to-end evaluation that includes energy, throughput, and area utilization estimates for an ASIC design, a fully functional RTL implementation working in both simulation and on FPGA (Zedboard Zynq-7000 development board), SPICE models of the underlying primitives on which our system is built, a fully automated toolchain linking *scikit-learn* software structures down to device configurations, and error versus energy analysis across a set of decision tree ensembles and design parameters. Even without accounting for the extra energy savings of using an encoding more natural to the sensor, the presented system dramatically reduces the total energy usage required for classification with very low latency.

Overall, the main contributions of this paper are:

- an optimized implementation of the recently introduced INHIBIT operator for race logic and a demonstration of its utility to classification problems,
- two different classes of architecture that leverage the properties of race logic temporal, yet still digital, nature to achieve low-energy classification with tree-based ensemble learners,
- a fully automated toolchain, linked to the open-source *scikit-learn* framework, for (a) the configuration of our programmable accelerator, or (b) the generation of a custom RTL-level design and (c) its functional verification through cross-checking against *scikit-learn*'s inference functions,
- detailed sub-component power and area models for architectural design space exploration and trade-off analysis.

We start, in the next section, with a discussion of an extended set of race logic core functions useful for more general construction. The background required to better understand tree-based ensemble methods such as Random Forests, AdaBoost, and Gradient Tree Boosting is provided in Section 3. Section 4 introduces our algorithm-architecture co-design approach and a custom-optimized implementation of a decision tree in race logic, while Section 5 describes the end-to-end system, dubbed "*Race Trees*", in detail. The toolchain responsible for optimizing, implementing, and configuring our architecture directly from a high-level machine learning framework is presented in Section 6. To evaluate performance and analyze the various design trade-offs, Section 7 provides a comparison with state-of-the-art classifiers and an architectural design space exploration. Finally, concluding remarks are given in Section 8.

2 Generalized Race Logic

The core idea behind race logic is to do computation through the purposeful manipulation of signal delays (either synchronously or asynchronously) rather than final logic levels. The total time that a signal takes to propagate through the system encodes the "value" of that signal. While traditional and well-understood CMOS logic can be used to build such systems, the encoding is extremely important to the efficiency of the system. As with any information representation change, some operations become easier to perform, while others become more difficult. Race logic's temporal coding base operations consists of four primary functions: MAX, MIN, ADD-CONSTANT, and INHIBIT.

While all four operations could be implemented using a traditional multi-bit binary encoding, an alternate and more efficient encoding is to use the time of rise for an edge [1] [22]. A value of "2" is represented in this encoding as a signal that is low until time 2, and then high from then on. Under this representation, a range of magnitudes can be encoded on a single wire with a single edge. Smaller delays in rise time encode smaller magnitudes, while larger magnitudes are encoded as longer delays.

In this logic, a MAX function should "go high" only when all of the inputs to the system have arrived (i.e. gone "high"). Therefore, a single AND gate between two wires is the only thing needed to implement MAX in the race domain. Figure 1 displays the symmetric nature of this function. The input that arrives first has to wait for the second one to arrive

[1] Race logic values are still "binary" in the sense that any given signals will only ever be "high" or "low", it is just that the transition time between the two is what encodes the value, so "high" no longer means "1" but instead means "the value has already arrived"

before the output responds. In the case of MIN, a first arrival selection operation, a single OR gate is all that is needed.[2]

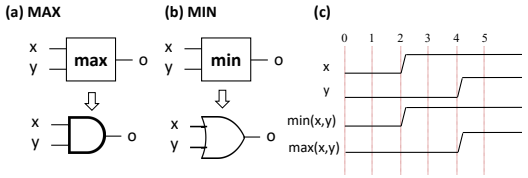


Figure 1. Panels (a) and (b) show the implementation of MAX and MIN functions in race logic. Panel (c) represents an example waveform for $x = 2$ and $y = 4$.

Furthermore, since the arrival time of the rising edge is what encodes information, delaying the $0 \rightarrow 1$ transition by a fixed amount of time is equivalent to constant addition (ADD-CONSTANT). Delaying a race logic-encoded input can be performed in multiple ways depending upon the implementation. In conventional synchronous digital logic, a sequence of flip-flops can be used, as shown in Figure 2. Asynchronous delay elements constructed out of current starved inverters have also been demonstrated to provide an alternative, more energy efficient method for performing the desired delay operation [24].

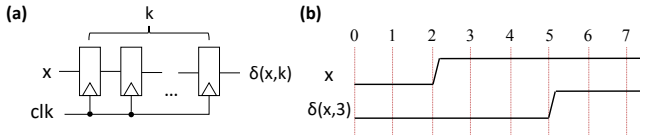


Figure 2. In race logic, adding a constant value k to a variable x is equivalent to delaying the rising edge of x by k clock cycles. Panel (a) shows how this delay can be achieved in conventional synchronous digital logic with the use of a shift register. Panel (b) shows an example waveform for $x = 2$ and $k = 3$.

In addition to the original MIN, MAX, and ADD-CONSTANT functions above, it was recently proposed that this set could be meaningfully extended with the addition of an INHIBIT function [37, 38]. This additional functionality is inspired by the behaviour of inhibitory post-synaptic potentials, as seen in the neurons of the neo-cortex. The inhibition of signals is particularly useful in the context of decision trees and requires some explanation.

The INHIBIT function has two inputs: an inhibiting signal and a data signal (that gets inhibited). If the inhibiting signal arrives first, the output is prevented from ever going high, which corresponds to ∞ in the race logic world. On the other hand, if the data signal arrives before or at the same time as the inhibiting signal, the former is allowed to pass through unchanged. In other words, the inhibiting input acts as a gate that only allows an earlier arriving or coincident data signal to pass. Figure 3 shows (a) the symbol used for i inhibiting j, (b) the function's state diagram as a Mealy machine, (c) and (d) possible fully digital-logic compatible implementations, and finally (e) a waveform depicting INHIBIT's functionality through two examples.

[2]The presented approach also enables the reverse encoding, where shorter delays represent larger magnitudes, due to the duality of race logic operators; in that case, first arrival will stand for MAX whereas last arrival will denote MIN.

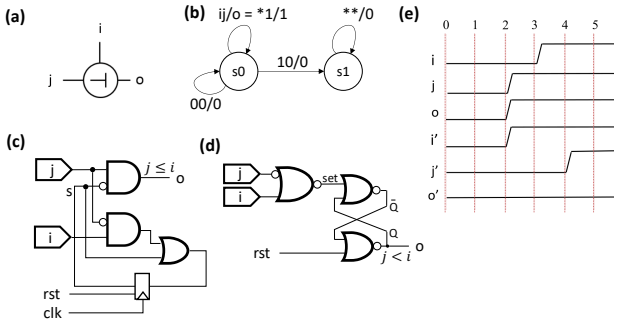


Figure 3. Panel (a) introduces the symbol that from now onwards we will use to represent the INHIBIT operator. Panel (b) presents the state diagram of the corresponding Mealy machine. Each transition edge is labeled with the value of inputs i and j, and the value of the output. The machine starts in state $s0$, which denotes that input j has not been inhibited by i, while state $s1$ indicates the opposite; j has been inhibited. Panels (c) and (d) show the implementation of the operator in a purely digital context. Finally, the waveform in Panel (e) depicts INHIBIT's functionality through two examples: (1) $i = 3$ and $j = 2$, and (2) $i' = 2$ and $j' = 4$.

While the implementations above can be fully realized with existing design tools as standard RTL, even more efficient implementations of the INHIBIT operator are possible with a little customization. Figure 4 (a) shows an INHIBIT built with four transistors. In this implementation, a pass gate, controlled by an inhibiting input i through the inverter, is used in the data signal path. If i arrives before j, the pass gate turns off, blocking the flow of j. Otherwise, j can pass through without getting masked.

A more compact version of the INHIBIT gate can also be designed. In that case, we leverage the one sided nature of race logic to reduce the transistor count further. Since we care only about rising edges, the only important signal to pass through is the logic 1. This allows us to use a single PMOS pass gate as an inhibit cell, as shown in Figure 4 (b).

Since both our inputs i and j are using the rising edge, two possible orderings of arrival exist. We verify the functional correctness for both of these orderings in the Cadence Analog Design Environment with a SPICE model using the 180nm process standard cell library. In Figure 4 (c), input j arrives before the controlling input i, and hence it passes

through without being blocked. The late arrival of the inhibit signal couples some charge into the output capacitance, which is visible as a small spike. The charge does not leak from the output capacitor after the inhibit input arrives though. Hence, the circuit's functionality is not affected. The case where i arrives before j is shown in Figure 4 (d). Here, the output does not change after input j arrives, as the controlling input i turns off the pass gates that "controls" the signal transmission. Similar to the previous case, a small amount of charge is injected into the output when the late signal arrives, which can be seen as a small negative voltage, but this is well within the noise margin and does not cause any misoperation.

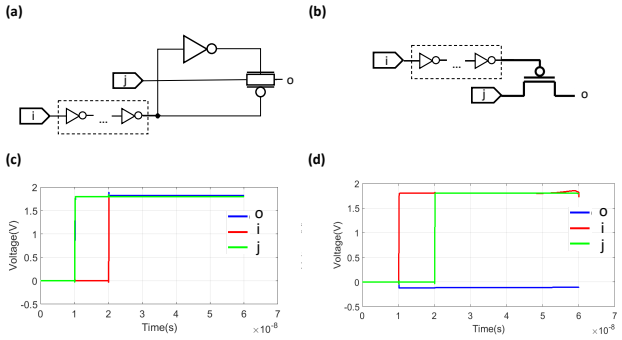


Figure 4. Panel (a) shows a 4 transistors INHIBIT operator implemented with a pass gate and an inverter. An even more compact design using only a single PMOS pass gate is presented in Panel (b). Panels (c) and (d) provide simulation results for the corresponding SPICE model in a 180nm process. More specifically, Panel (c) covers the case where j arrives before i, whereas Panel (d) shows how the controlling input i blocks j, when the former arrives first.

The fact that such asynchronous implementations of the inhibit gate do not need a clock brings area and energy gains, but can also lead to glitches when not used carefully. According to the shown SPICE simulations, there seems to be no problem when data and inhibiting signals are spaced by at least one clock cycle. But what happens when they both arrive at the same time? To ensure that the system does not glitch in such a case and the gate's functionality is preserved, the inhibiting input must be delayed by enough time so the data signal can pass clearly. This delay can be implemented with a chain of inverters present inside each INHIBIT cell. This method, though functionally correct, makes each cell larger. To avoid this overhead, a solution tailored to our *RaceTrees* architecture is presented in Section 4. To demonstrate the feasibility of such a compact INHIBIT cell, SPICE simulation results for a simple *RaceTree* are also provided.

Together this set of four operations allow us to deliberately engineer "race conditions" in a circuit to perform useful computation at low energy. The resulting systems use only one wire per operand (leading to both smaller area and capacitance) and at most one bit-flip per wire to perform the desired operation (resulting in less switching activity and dynamic power). While not all computations are amenable to such an encoding, those that are have the potential to operate with *very* little energy. An open question answered in this work is if such logic is amenable to any general learning or classification tasks.

3 Decision Trees & Ensemble Methods

While monolithic neural networks receive the lion's share of machine learning attention from the architecture community, decision trees have proven to be incredibly useful in many contexts and a promising solution towards explainable, high-performing AI systems [16, 17].

A decision tree, as its name denotes, creates a hierarchy of decisions, which consists of a set of "leaves" (that correspond to target class labels) and a set of decisions to be made (i.e. "branches") that lead one to those labels. Each branch splits the data according to some attribute being above or below a specified level. Thus, when making a decision tree we must take into consideration which feature we select as an attribute for each node and what is the threshold for classifying each "question" into a yes or no answer.

While decision trees can in theory chain together huge numbers of decisions to provide an accurate classification, in reality this is often limited by an important trade-off between misclassifying outliers and overfitting. This trade-off is particularly challenging when dealing with complex classification tasks where the best label is a function of *many* dimensions. This is where ensemble methods come in handy.

Ensemble methods are a class of learning algorithms that construct a number of base learners and integrate them in some way to obtain a prediction about a new set of data points. Although the performance of each individual learner may be weak, ensemble methods increase accuracy, and decrease variance and bias through classifiers grouping or chaining. A visualization of this concept can be found in Figure 5.

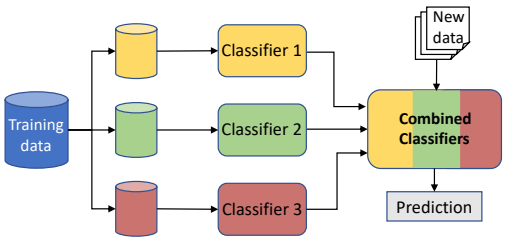


Figure 5. Ensemble methods combine multiple weak learners to obtain a better predictive performance.

Random forest is one of the most popular ensemble learning methods. Rather than growing a single tree, the algorithm

creates an ensemble of tree predictors, each looking at a random subset of features [5]. To classify a new object, the input vector has to walk through each of the trees in the forest. The decisions of all these learners are then combined and the forest chooses the classification having the most "votes". An early example of this approach is bagging (bootstrap aggregating). As stated by L. Breiman, "bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor" [4]. This method can be improved even further with boosting. Boosting classifiers are built one-by-one and try to reduce the remaining bias of the combined estimator. In other words, boosting strives to improve the ensemble by focusing new learners on those areas where the system performs worst. One of the most well-known examples of this technique is AdaBoost; short for "Adaptive Boosting".

The AdaBoost algorithm, introduced in 1995 by Freund and Schapire [14], solved many of the practical difficulties earlier boosting algorithms faced. The algorithm begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

Another popular algorithm, following the same fundamental boosting idea, is Gradient Boosting [15]. Unlike AdaBoost, where "shortcomings" are identified by high-weight data points, in Gradient Boosting the system's existing inefficiencies are identified by gradients. Gradient Boosting builds an additive model in a forward stage-wise fashion, minimizing complicated loss functions through gradient descent. One of the most successful derivatives of this method is XGBoost (extreme Gradient Boosting) [8], an optimized implementation of gradient boosted decision trees, that has recently been dominating applied machine learning and Kaggle competitions for structured or tabular data.

4 Rethinking Decision Trees

Besides the differences in the construction/training phase of the described ensemble learners, the inference process is similar for all these approaches. Interestingly, as we will see, this model of decisions fits nicely with the race logic encoding and the set of supported operations. To maximize efficiency, we proceed with an algorithm-architecture co-design methodology.

Normally, decision trees are thought of in a top down fashion – one starts at the root and branches down the tree to find the relevant answer. Each node implements a discriminant function for classification and each leaf is associated to a given class; therefore, classifying an input pattern reduces to a sequence of decisions.

In race logic, time is used as a computational resource. Existing race logic implementations, such as the DNA global sequence alignment engine [22], perform computation by

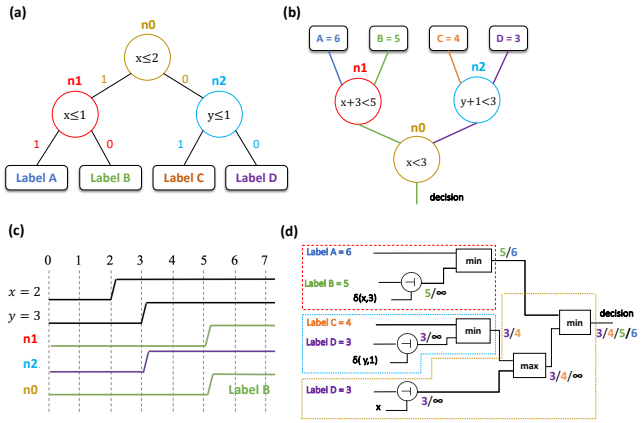


Figure 6. Panel (a) presents a decision tree that from now onward will be used as an example case. Panel (b) shows its "reverse" equivalent as well as the "flow" of the four temporally encoded labels for $x = 2$ and $y = 3$. Panel (c) displays the corresponding waveform for the given example. Panel(d) depicts the race logic implementation of this "reverse tree". The leaf label associated with the *False* branch of a node plays the role of j in the INHIBIT operator, whereas the record's attribute a serves as the inhibiting input i. In respect of the node's threshold t and race logic limitations (e.g. we cannot subtract or add variable numbers), the attribute a routed to an INHIBIT's controlling input has to be adjusted accordingly; e.g. $y < 2$ has to be rewritten as $y + 1 < 3$.

observing the relative propagation times of signals injected into the circuit. Following this example of computation, one approach to implement decision trees is by virtually turning them upside down, i.e. they can be thought of as trees that route possible answers from the leaves to the root. Initially, a unique delay-encoded label is assigned to each leaf. These labels then race against one another, and where two of them "meet" only one is allowed to propagate further. In the end, only the label associated with the "correct" leaf survives [3].

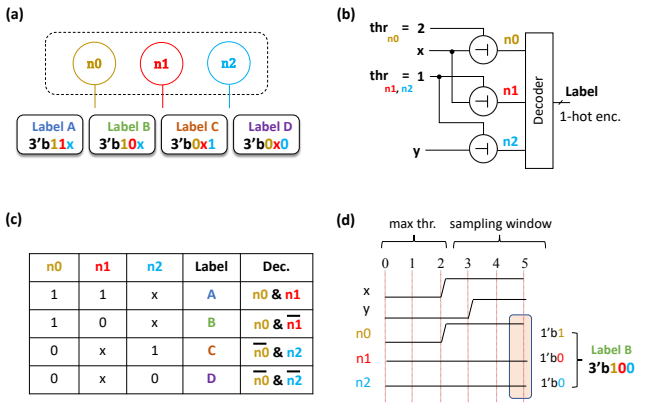
A decision tree that from now onward will be used as an example case is presented in Figure 6 (a). Figure 6 (b) shows the "flow" of the four temporally encoded labels in the "reverse" tree for $x = 2$ and $y = 3$, and Figure 6 (c) displays the corresponding waveform. Its race logic implementation is depicted in Figure 6 (d). The upper two blocks, colored in red and blue, correspond to the tree's internal nodes ($x \leq 1$ and $y \leq 1$) and are implemented with the use of one INHIBIT and one MIN operators, while the bottom one, colored in

[3]Leaf labels can be thought of as packets that are routed through a "reverse tree" network. Some packets are discarded along the way, but the packet at the output has been routed unchanged. Externally, the packets are assigned values that are purely symbolic and contain no useful numerical content – in much the same way that numbers in sudoku are used symbolically, but not numerically. However, internal to the network, as part of the routing architecture, the packet values do take part in numerical operations.

yellow, is slightly more complicated as the label coming from its *False* path can take more than one values; either one of *Labels C* and *D* in the given example.

As already discussed, when off-the-shelf digital circuits are used, shift-registers play the role of delay elements (ADD-CONSTANT). Therefore, a 2D "matrix", where each row (their number is equal to the number of input-features) corresponds to a shift register whose length depends on the trees' depth and features' resolution, would be required to delay input features by an appropriate number of clock cycles in a way that they will serve as the controlling inputs to the INHIBIT operators; e.g. to implement $y + 1 < 3$, feature y must be delayed by 1 clock cycle. However, these clocked components are relatively "costly", and ideally their usage should be constrained.

An alternative way to look at decision trees is as a set of independent and parallel rather than sequential decision rules that lead to a final prediction when combined accordingly. Each leaf now can be represented as a logical function of the binary decisions encountered at the nodes on its path to the root. In other words, each path from the tree root to a leaf corresponds to a unique conjunction of attribute tests. The big idea behind the parallel execution of all these independent *if* clauses is shown in Figure 7 (a). For example, the leftmost leaf is reached only when both $n0$ and $n1$ return *True*, while the output of $n2$ is indifferent. The order that the outcomes of these conditions reveal does not affect the final decision.



n0	n1	n2	Label	Dec.
1	1	x	A	n0 & n1
1	0	x	B	n0 & $\overline{n1}$
0	x	1	C	$\overline{n0}$ & n2
0	x	0	D	$\overline{n0}$ & $\overline{n2}$

Figure 7. A decision tree can be viewed as a set of independent decision rules that lead to one and only one leaf when combined accordingly. Hence, these functions can be executed in parallel, as shown in Panel (a). Panel (b) depicts the race logic implementation of this "flattened" decision tree with the use of INHIBITs, where thresholds play the role of the gates' controlling inputs. Panel (c) presents the truth table that defines decoder's functionality, which associates the nodes' outcomes with one of the leaf labels. Panel (d) displays the resulting waveform for $x = 2$ and $y = 3$.

In a decision tree, each node has a fixed threshold value t, assigned to it from the training phase, which is then getting compared to an attribute a of the provided input record as we conduct inference. Its function can be expressed as follows:

$$node(a,t) = \begin{cases} True\ Branch, & \text{if } a \leq t \\ False\ Branch, & \text{otherwise} \end{cases}$$

Considering that the outcome of each node is a binary decision, decision trees can be considered as networks of threshold functions [2]. The implementation of these functions in conventional digital logic would be trivial. However, their size and performance are directly related to the resolution of the associated attribute and threshold values. This is not the case for a race logic implementation though as a range of magnitudes can be encoded on a single wire with a single edge. To better understand how such a threshold function can be implemented in race logic the definition of INHIBIT, where i is the controlling input over j, should be revisited:

$$i \dashv j = \begin{cases} j, & \text{if } j \leq i \\ \infty, & \text{otherwise} \end{cases}$$

Interestingly, the two above definitions have significant similarities, especially when considering that ∞ denotes no $0 \rightarrow 1$ transition, and that the maximum threshold value across all tree nodes is known at design time. Figure 7 (b) depicts the race logic implementation of the flat decision tree found in Figure 7 (a). The decoder that associates nodes' decisions with one of the leaf labels implements the truth table presented in Figure 7 (c). Figure 7 (d) shows the resulting waveform for $x = 2$ and and $y = 3$. In that case, x is going to pass through $n0$, but both x and y will be masked in $n1$ and $n2$. Moreover, the maximum threshold value is statically known, equal to 2. Hence, we can read the outcome of $n0$, $n1$, and $n2$ conditions at any time after that. In that way, the transition from the temporal to the binary domain happens seamlessly and without the need of any additional hardware resources. To recap, this approach requires only one INHIBIT gate per tree node, the number of the potentially power-hungry shift-registers is minimized, and no additional logic is required for the transition from the temporal to the binary domain.

Figure 8 provides SPICE simulation results for the same example tree implemented with single transistor INHIBITs. To avoid glitches, a chain of four inverters is used per threshold, when operating at 1GHz, to cause the required delay. The way *Race Trees* are designed allows us one more optimization. We observe that in our case only thresholds are serving as inhibiting signals. Therefore, the inverters that are typically "inside" the INHIBIT cell can be "pulled out" and placed at the signal path between the delay-encoded thresholds and INHIBITs. This change enables the sharing of the chains of inverters between various INHIBIT cells,

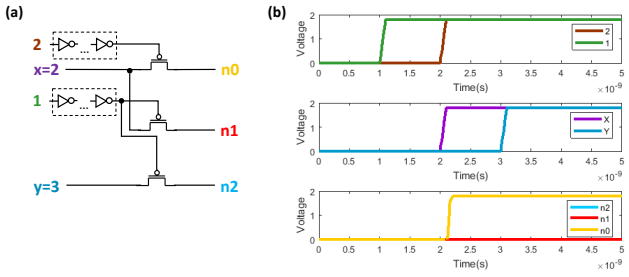


Figure 8. SPICE simulation results, using a 180nm CMOS library, for a *RaceTree* implemented with single transistor INHIBITs. To reduce the overhead of the chains of inverters, required for the delay of inhibiting inputs (to avoid glitches in case both i and j arrive at the cycle), rather than embedding them in each INHIBIT we place them at the threshold lines and share them between various cells.

which results in both a glitch free operation of the system and a more compact design.

5 System Architecture

In this section, we present our end-to-end architecture shown in Figure 9. Initially, we provide our view on our system's integration with sensors that encode data in the time domain, and then describe in detail the architectural decisions necessary for the development of a programmable race logic accelerator targeting tree-based learners. Besides the trees themselves, the system consists of a block for the temporal encoding of the referred threshold values as well as some decoding logic for accessing the contents of the trees' leaves once the final decision has been made. Finally, the voting circuitry, which is directly related to the notion of ensemble methods, adds all trees' votes and finds the class with the highest sum.

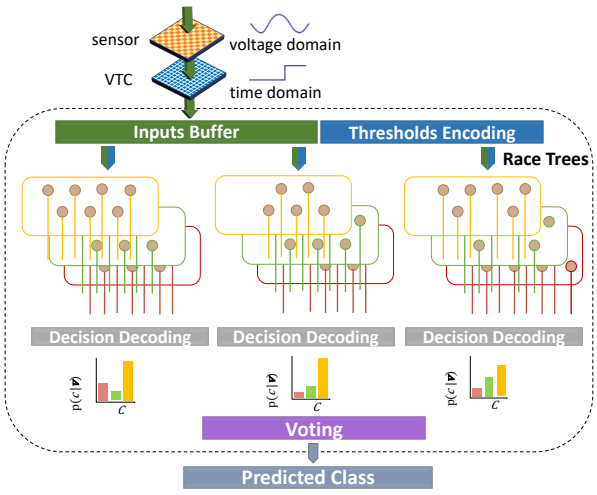


Figure 9. Overview of the presented architecture.

An open-source implementation of race logic primitives and example *Race Trees* can be found on github [4] for quick use and reference.

5.1 From Sensor to Delay Coded Input

Whenever a different encoding is considered, the cost of "translation" in and out of that encoding has to be taken into account. However, race logic is such a natural direct target for sensors that we instead consider a case where classification tasks are tightly integrated with the sensor. Since sensory input (e.g. light, sound, humidity, temperature, chemical potentials) is analog in nature, most sensors begin with a measured voltage or current. Aside from simple filtering and de-noising tasks, most signal processing is done after Analog-to-Digital Conversion (ADC).

Analog-to-Digital Converters are key components for a plethora of applications, and their design optimization, both in terms of area and power, has been an active research topic for years. Since design of analog circuits gets more challenging with reduced supply voltages and short channel effects in deep sub-micron process nodes, high fidelity analog amplifiers consume a large amount of power. With power being a critical constraint, techniques that relax this fidelity and offload it to digital components allow for more efficient designs. One such set of techniques involves performing temporal signal processing, by first converting the analog signal into the temporal domain. Such an approach leverages the small delays and high temporal resolution of nano-meter scale devices to get superior performance over their voltage domain counterparts [26]. Once faithful temporal coding has been achieved, various (mostly digital) time-to-digital conversion strategies allow outputs to be converted to the digital domain for further processing.

While the typical digital domain, with its well-understood binary encoding, may be very well-suited for storage and general-purpose manipulation of data, this is not always the case for designs with constrained area and power budgets. For example, there exists a variety of temporal and rate coded architectures, such as spiking neural networks [34], where either the arrival time of the first spike or the firing rate of spikes, which is defined by the intensity of the provided stimulus, encode the values of data. Given that information in the time domain is now considered useful for computation, the above-mentioned time-to-digital conversion (TDC) of ADCs [9, 18] is redundant and can be skipped. Examples of systems providing directly time-encoded outputs, without TDCs, range from visual sensors, such as Dynamic Vision Sensors (DVS) [21, 35], Asynchronous Time-based Image Sensors (ATIS)[29], Time To First Spike (TTFS)[30] and Time-of-Flight (ToF) [12, 27] cameras, to sound sensors; e.g. the AER (Address Event Representation) Ear [6], which is a silicon-based cochlea that converts auditory signals from

[4] https://github.com/UCSBarchlab/RaceLogic

various frequency bands to precise temporally coded outputs.

Race logic is built on the idea of encoding information in timing delay; it uses edges rather than spikes though. Therefore, the presented architecture can work with any temporally-coded input provided to it. Considering the nature of the above-described sensing systems, we expect that with minimal changes in their interfaces [25], raw time-coded data can be directly fed into our race logic accelerator, and enable a tighter and more efficient sensor-accelerator integration.

5.2 Race Trees Architecture

As already discussed, a decision tree can be thought as a set of independent and parallel decision rules that lead to a final prediction when combined accordingly. For the realization of each tree node an INHIBIT operator is used, where a delay-coded threshold serves as the gate's controlling input. Figure 10 provides more details regarding the implementation of a decision tree of depth 2 in a configurable way.

For the temporal encoding of the various threshold values a shift register is used. Its content is initially set to 0, and a steady logical high signal is shifted in once the processing starts. To ensure that any delay-coded threshold value can be routed to the necessary nodes of the tree a configurable crossbar is used; colored in blue in our figure. As expected, its dimensions depend on the length of the shift register, which is defined by the input features' resolution, and the total number of the tree's nodes. Assuming that input features are coming from the sensor already delay-coded, another configurable network, colored in green in our example, must be used for their routing to INHIBITs' data signal ports.

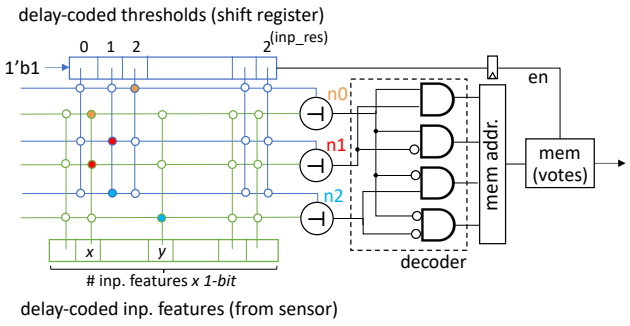


Figure 10. Programmable race logic accelerator for a decision tree of depth 2. The shown configuration corresponds to the tree presented in Figure 7. The length of the shift register, used for the temporal encoding of thresholds, is defined by the resolution of input features; e.g. if input resolution is 8 bits, a shift register of depth 256 has to be used. The memory block shown after the decoder is useful in the case of a trees ensemble, where a weighted voting scheme follows.

The outcome of a decision tree is the contents of the reached leaf node, and these contents are stored in a memory in our case. To access this memory, the output of the array of INHIBITs has to be transformed into a valid address. Taking into consideration the fact that each leaf is associated with a unique conjunction of the conditions along its path to the root, a simple decoder consisting of AND gates and inverters has to be built. The output of this block is an 1-hot encoded memory address. At this point, it should be noted that although INHIBITs are conceptually operating in the temporal domain, their sampled output will be a typical binary vector; the maximum number of clock cycles required to process a tree is defined by the length of the shift register and is known statically. Hence, only one memory read is required per input record at the time that the propagating wavefront reaches the end of the shift register. Prior to the next computation, our circuit must be reset.

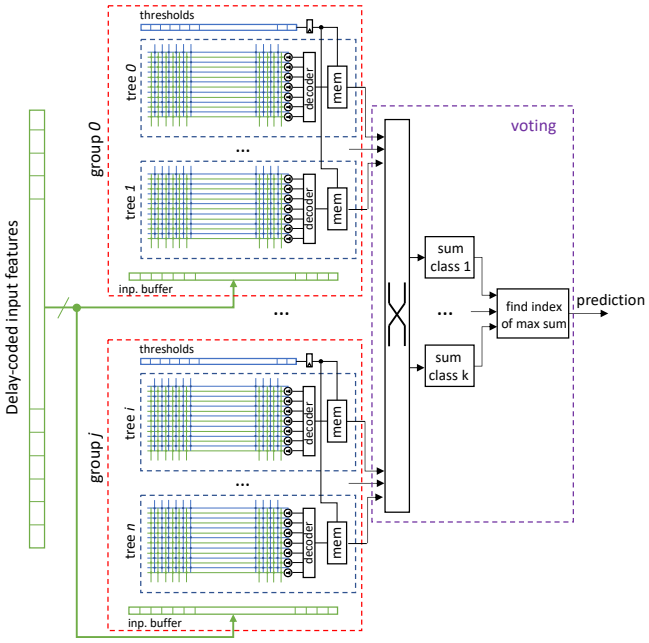


Figure 11. Lower-level diagram of the architecture presented in Figure 9. The shown circuit implements a configurable race logic accelerator for tree-based ensemble learners.

Figure 11 presents our system architecture for a tree-based ensemble learner. To keep the overhead of clocked components low, trees are now organized into groups and share the same shift register and buffer, used for the generation of the delay-coded thresholds and the local buffering of the delay-coded input features. Of course the "cost" of the crossbars is increasing with the size of these groups, and this trade-off should be further analyzed for maximum efficiency.

Since the data retrieved from memory are in regular binary encoding, the implementation of the weighted voting scheme

is based on typical binary adders. Once all prediction values of all trees have been summed, a comparison between them takes place to find the class with the highest value, which is going to be our system's final "guess". At this point, it should be noted that the shown crossbars can be replaced by any other more efficient configurable routing network without any effect in our system's functionality.

Finally, to make this architecture more generally applicable, we need to ensure that the tree structure itself can be changed. This is done by constructing only full trees with all their nodes accounted for. Nodes that are necessary for the correct functioning of the trained network are activated while the unnecessary ones are statically set to *False* at configuration time.

6 Race Trees Toolchain

When designing a new system, development time and usability are always major concerns. Ideally, the proposed architecture should be modular for systematic implementation, and an abstraction layer, hiding the hardware complexity from domain experts, should also exist. In the case of machine learning, the vast majority of development happens with the use of specialized software libraries. To address the above-mentioned challenges and assist evaluation, a fully automated toolchain, linked to the open-source *scikit-learn* framework, has been developed for *Race Trees*.

6.1 Implementation & Programming Methodology

One of the first decisions to be made when starting a machine learning project is selecting the learning method that will be used as well as its parameters, e.g. number of estimators, size of the trees. As a next step, training data are provided to the algorithm to learn from.

In this work, *scikit-learn* [28] is used for the training part. *Scikit-learn* is a free software machine learning library for the Python programming language and features various classification, regression and clustering algorithms. Initially, a set of user-defined parameters is expected, as usual, for the construction of the desired learning model. Considering that our goal is to map this model to hardware, and inspired by recent studies that demonstrate high accuracy number with even reduced precision inputs [39], features quantization is given to the user as an option. Once the model is trained, our tool analyzes the importance of input features, explores the learner's performance against lower resolution data, and proceeds with votes (content of tree leaves) quantization. Finally, the user has to choose one of the following options: (a) the generation of a configuration file for the crossbar networks of a programmable *Race Trees* accelerator (Figure 10) , or (b) the generation of a custom RTL-level design with hardwired units.

Figure 12 provides a quick visual overview of our design flow. The automatic implementation of our architecture in hardware directly from a typical software machine learning framework is based on the idea of hardware templates. Synthesizable RTL code is generated out of a simple graph-based intermediate representation with the use of a set of basic building blocks (e.g. parametric shift register and buffer, INHIBIT gate, etc.) and "glue" logic. More specifically, once the training process completes, the learning model contains information about the total number of nodes per tree, their interconnection, threshold values, and referenced input features, as well as the data of each leaf. Tree's hierarchy is used for tagging its nodes accordingly, which directly affects the structure of its decoder, while the threshold value and attribute assigned to each node define the connectivity between the corresponding INHIBIT's ports and the associated shift register and inputs buffer.

Regarding the implementation of the weighted voting scheme that characterizes ensemble methods, the values of the leaves' data/votes have to be normalized. In the software world, where these machine learning models usually live, leaves' votes are typically in floating point format. Thus, their conversion in a hardware-friendly fixed point representation is necessary. The number of bits used for their storage affects both the predictive performance of the learner and the size of the memory. To reduce the quantization error, the distribution of the votes' values is first analyzed to detect outliers (with the use of percentiles), and then the rest of the data are normalized according to the min-max feature scaling approach. At this point, it should be noted that race logic itself does not introduce any new sources of inaccuracy, as this kind of data quantization for the votes would be necessary for any reasonable hardware realization of the algorithm.

6.2 Area & Power Models

Considering that the main focus of this work is to stretch the boundaries of race logic, we create analytical and empirical power and area models for the basic components of the presented architecture. To do so, we synthesize the RTL of each sub-component of *Race Trees* individually, as well as the RTL of the whole design as generated by our flow. To obtain the desired area, power, and performance results, we use an open source tool [40] and the 14nm standard cell library found in [7]. The operational voltage and frequency are 0.55V and 1,000MHz, respectively. However, in our energy and throughput calculations we use a 500MHz clock, without scaling our power numbers, to compensate for the lack of a wire load model; under this assumption, each operation consumes twice its nominal energy [36]. For the switch fabric included in the analysis, scaled energy results for the 14nm node from [3] are used. For the interconnect switch block, we scale the power from 45nm [33] to 14nm.

Besides identifying target components for further optimization, the constructed models can also be used for the exploration of different design options and their effects. For

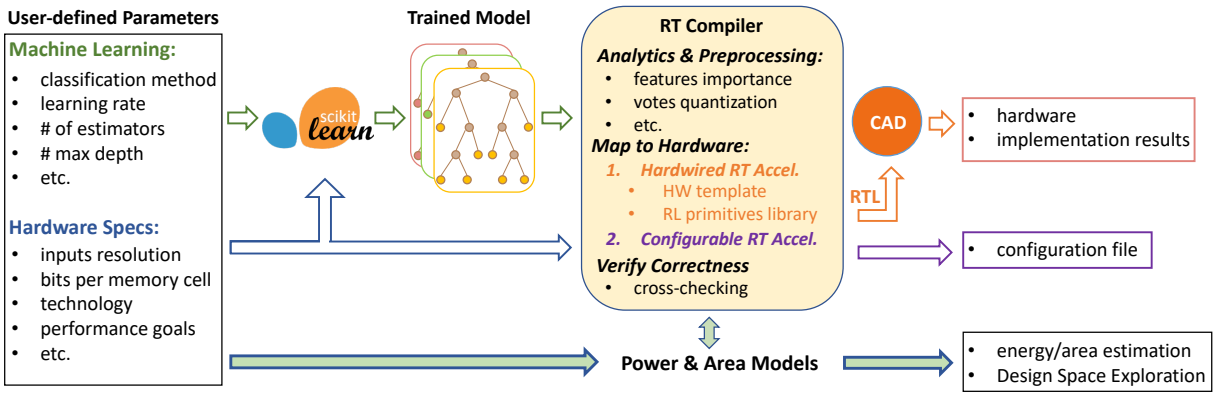


Figure 12. Overview of the developed toolchain. For the training part, the open-source *scikit-learn* framework is used. The tool (a) provides the user with the options to quantize input features and/or trees' votes, (b) support the automatic generation of model-specific *Race Trees* circuitry or the configuration of a programmable architecture directly from *scikit-learn* structures, and assists (c) trade-off analysis and (d) design evaluation through an analytical architectural models and cross-checking with software models, respectively.

example, they do not only help the user understand the accuracy versus system performance trade-off for various learners better, without having to implement each of them, but also assist in the design of the hardware itself. For example, defining the size of each group of trees, as shown in Figure 11, is an architectural decision that should compromise the overhead imposed by the clocked components and the complexity of the required configurable routing networks, and its optimal value can be computed with the use of known modeling frameworks [11].

6.3 Evaluation of Correctness

For the development and verification of a *Race Trees* design, PyRTL [10], a Python embedded hardware design language, is used. The fact that both *scikit-learn* and PyRTL are built around Python makes their integration easier and assists verification. More specifically, once the PyRTL code for a specific model is complete, software modules, such as the *predict* function of the *scikit-learn* library, can be used as the golden reference for the design's functional verification. However, *scikit-learn* does not provide/expect signals in the temporal domain. To simulate input stimuli, Python's generators are used to encode input data while the results can be directly pulled out of the simulation tracer to facilitate cross-checking.

7 Performance Evaluation

In this section, we first provide a comparison between the presented system and other state-of-the-art solutions, and then perform a trade-off analysis to get deeper insight into the design space.

7.1 Comparison with State-of-the-Art Classifiers

In recent years, an explosion of hardware accelerated machine learning activity has resulted in a wide variety of ASIC architectures to compare against. Based on our experience and literature survey, MNIST is the most commonly used dataset in the context of a proof-of-concept prototype. Hence, we proceed with an MNIST-based evaluation that facilitates the comparison of the proposed approach with state-of-the-art low-power classifiers. Figure 13 plots a few of these solutions on an accuracy versus energy plot.

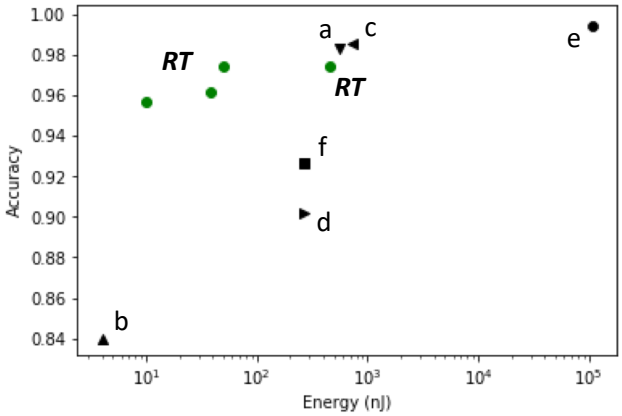


Figure 13. Accuracy vs energy scatter plot for state-of-the-art machine learning accelerators: a [41], b [19], c [31], d [20], e [13], f [13]. For easier comparison, all results have been scaled to 28nm. Green dots represent *Race Trees*.

One approach that has tried to tame this massive design space is Minerva [31], which is represented in Figure 13 by the letter *c*. Minerva is an automated co-design approach that

accounts for algorithmic, architectural, and circuit level constraints in an attempt to efficiently accelerate Deep Neural Networks. More specifically, Minerva first performs design space exploration at the algorithmic and architectural level, followed by resolution tweaking and pruning of certain unnecessary energy hungry operations. Moreover, it looks at circuit level optimizations, such as SRAM fault mitigation, before reporting accurate chip level performance metrics. This broad design space exploration and multi-level optimizations allow Minerva to be highly accurate, still energy efficient, and make it a good starting point for comparison.

Another interesting implementation, represented by the letter *b* in Figure 13, is the sparse event-driven neuromorphic object recognition processor developed by J. J. Kim et al. [19]. This solution is composed of the locally competitive algorithm (LCA) inference module [32] (for feature extraction) and a task-driven classifier. As can be seen, this spiking neuron architecture allows for a very low energy cost ($\approx 20.7nJ/pred$ in 65nm). However, it achieves only 84% accuracy, which is the lowest among the displayed solutions.

At the other extreme, a high performance sparsely connected neural network running on the IBM TrueNorth chip (28nm) utilizes 64 ensembles and hits very high accuracy numbers (99.42%) at the the expense of $108\mu J/pred$ [13]. A more energy efficient version with a single ensemble is also reported and achieves 92.7% accuracy at $268nJ/pred$. These implementations are represented in Figure 13 by the letters *e* and *f*, respectively. A few other solutions with comparable accuracy and energy performance, represented by the letters *a* [41] and *d* [20], are displayed, too.

Table 1. Synthesis results for hardwired *Race Trees* produced by Yosys [42] with a cell library in 14nm CMOS and power results using [40].

# Trees	Depth	Inp. res. (bits)	Mem. bits (per vote)	Accuracy	Latency (CCs)	Power (mW)	Area (mm^2)	Freq. (MHz)
1,000	6	8	8	97.48%	273	521	0.46	1,000
1,000	6	4	4	97.45%	33	475	0.45	1,000
200	8	4	4	96.18%	31	384	0.33	1,000
200	6	4	4	95.72%	31	125	0.13	1,000

Our *Race Trees* are represented with green dots. As already described, our architecture can serve any tree-based ensemble method. The technique used for this comparison is Gradient Boosting, whose derivatives have recently gained popularity by winning various Kaggle and other data science competitions. A classifier consisting of $1,000$ *Race Trees* of depth 6 gets 97.45% accuracy, while still maintaining a fairly low energy expenditure at $31.35nJ/pred$. A more efficient approach, utilizing only a fifth of the number of ensembles, still achieves a performance of 95.7% with energy numbers as low as $7.8nJ/pred$. By increasing the depth of trees to 8, the accuracy increments by 0.5% at the expense of $16.1nJ$ of additional energy per prediction. At this point, it should be noted that we are not performing any parameter fine-tuning to improve the learner's performance, and that race logic does not introduce any new sources of inaccuracy. The reported implementation results for the hardwired designs are produced by Yosys synthesis suite [42] with a standard cell library in 14nm CMOS [7], while the energy and area of our programmable architecture are estimated with the use of the models described in Section 6.2. More information about the designs under test can be found in Tables 1 and 2.

The total number of execution clock cycles required per prediction is defined by the following equation:

$$latency = 2^{inp\_res} + log_2(\#ests) + \#classes$$

The first term is related to the maximum possible value of the thresholds, which also defines the length of the shift register used for their temporal encoding, whereas the second and third terms are associated with the voting part (argmax of the sums of the predicted probabilities). This means that our *Race Trees* accelerator can classify $1.83M$ images/s when 8 bit inputs are used and up to $16.1M$ images/s for 4 bit inputs.

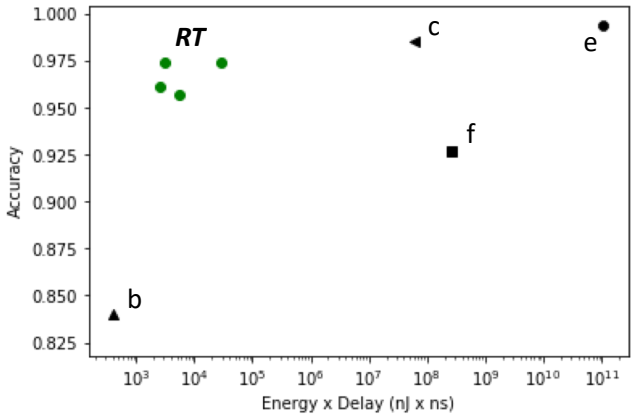


Figure 14. Accuracy vs energy-delay product scatter plot for state-of-the-art machine learning accelerators: b [19], c [31], e [13], f [13]

Though application specific, a change of information representation allows problems to be reformulated in a way that can be performed both quickly and efficiently, and race logic does just that. On one hand, the temporal domain representation, which allows multiple values to be encoded on the same wire, coupled with simple digital primitives and a low logic depth, allow high speed. On the other hand, a single edge transition per computation running down a spatially laid out architecture allows for superior energy efficiency. A comparison of *RaceTrees* to other machine learning implementations in an accuracy vs energy-delay product scatter plot is presented in Figure 14, and shows that *RaceTrees* achieve both lower delay and energy per operation than their counterparts.

Table 2. Estimated power and resource utilization for various sub-units of the *Race Trees* architecture.

# Trees	Depth	Inp. res.	Mem. bits	Groups	Tech. node	Thresholds		Inp. Buffers		Trees & Dec.		Memory		Voting		Progr. Intercon.		Total	
						P (μW)	A (mm^2)	P (μW)	A (mm^2)	P (μW)	A (mm^2)	P (μW)	A (mm^2)	P (μW)	A (mm^2)	P (pW)	A (mm^2)	P (mW)	A (mm^2)
1,000	6	8	8	10	14nm	7,237	1.1e-2	21,999	3.5e-2	529,603	0.5	111,492	0.05	2,008	0.03	0.252	–	673	0.63
1,000	6	4	4	10	14nm	359	7.5e-4	21,999	3.5e-2	529,603	0.5	59,289	0.03	1,673	0.03	0.016	–	613	0.59
200	8	4	4	10	14nm	359	7.5e-4	21,999	3.5e-2	452,701	0.38	35,715	1.5e-2	321	5.8e-3	0.013	–	511	0.44
200	6	4	4	10	14nm	359	7.5e-4	21,999	3.5e-2	105,921	0.1	11,857	5.6e-3	321	5.8e-3	0.003	–	140	0.15

7.2 Design Space Exploration & Performance Analysis

The main focus of this work is to stretch the boundaries of race logic and demonstrate the effectiveness of classification accelerators that leverage its properties. Thus, we believe that it is important to understand where and why the majority of power is consumed. Table 2 provides area and power estimates for each of our system's main blocks for four Gradient Boosting classifiers of different size.

To obtain these results, we synthesize each sub-component of the *Race Trees* architecture individually, as described in Section 6.2. In this analysis, for the realization of the INHIBIT operators, we use off-the-shelf CMOS components (Figure 3), rather than any of the custom implementations presented above (Figure 4). The cost of the programmable interconnect is calculated based on the results from [3]. As a sanity check, we compare the sum of area and power results of all these components for each classifier against the complete synthesized design results, shown in Table 1. The differences observed are expected as all the trees are now considered fully-grown to cover the most general case.

In Figure 15, the energy vs accuracy trade-off for a variety of Random Forest, AdaBoost, and Gradient Boosting implementations, is illustrated. The number of estimators, trees' max depth, and the learning rate were considered as parameters for all three algorithms. For the cases of Random Forest and AdaBoost, the numbers of estimators, trees' max depth and learning rate range from 20 to 200, 6 to 10, and 0.3 to 0.8, respectively. In the case of Gradient Boosting Classifiers, the upper bound for the number of estimator was set to 100 (consisting of 10 trees each, so 1,000 trees in total). In fact, we observe that the prediction performance of the latter does not change significantly, for the MNIST dataset at least, as the depth of trees increases.

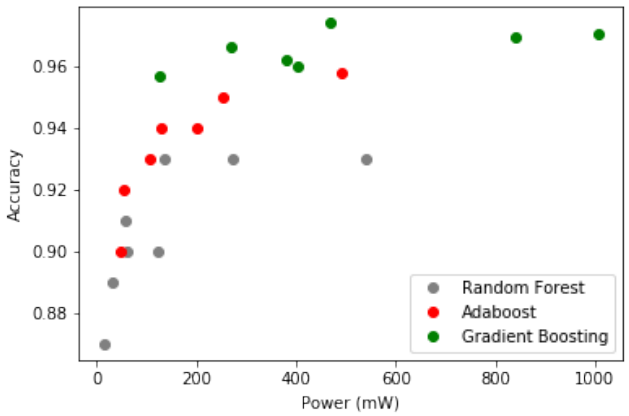


Figure 15. Accuracy vs power scatter plot for various configurations of Random Forest, AdaBoost and Gradient Boosting Classifiers.

8 Conclusion

As machine learning techniques continue to find new and compelling application across a wide range of computing tasks, the desire to bring that computational power into even our lowest power devices will only continue to grow. Applying these complex algorithms without resorting to the use of significant amounts of energy remains an important challenge, and the choice of data representation is an important cross-layer factor that impacts everything from the sensor to the end product of learning.

While each representation embodies a different set of trade-offs based on the algebraic operations that are either easy or hard in that domain, machine learning algorithms themselves have a degree of malleability that has yet to be fully exploited. We show that the natural relationship between modern decision tree algorithms, new advances in race logic, and the underlying sensors themselves provide such an opportunity. Although, it is rare to come across a change that appears simultaneously beneficial across all three of these different layers, sensor, architecture, and learning algorithm, a delay code seems to be exactly such a rarity. Others have already shown the analog advantage of avoiding the final step of converting input signals to a pure digital representation — instead leaving them as a variable delay which can be trivially converted to a race encoding. At the algorithm level, little is needed in the way of changes other than being mindful of the depth and configuration of the existing decision tree algorithms. We present an interface to the user that is identical to that used already by the well-loved *scikit-learn* package. At the architecture level the improvements for these considerations are dramatic both in hardwired and programmable configurations. The resulting computation has a shallow critical path and induces exceedingly few bit transitions as the computation propagates through *Race Trees*.

To demonstrate that this approach could be useful in practice we design and analyze, at the hardware level, two different approaches across a set of inputs. Because each and every wire of the entire accelerator, besides clock, flips from

0 to 1 at most once across the entire computation, the required energy per image classification can be as low as 7.8 nJ, while still achieving a 95.7% accuracy and 16.1M predictions/s throughput for the MNIST dataset. Assuming fully-grown race trees, the routing crossbars are the only parts that need to be configured to support arbitrary branching over features.

While the resulting system already performs admirably as measured by energy, area, and performance, there is still room for further exploration and improvement. The evaluation here has yet to take advantage of a) the asymmetric nature of the logic level transitions (meaning that only 0 to 1 transitions are performance sensitive), b) the true malleability afforded by the decision tree algorithms and the co-design it enables, and c) the ability of delay and INHIBIT operations to be even more efficiently implemented by less traditional technologies. Lastly, the integration of race logic-based accelerators with other circuits operating purely on the time-domain, such as the recently proposed vector-matrix multiplier presented in [1], is another interesting path for exploration towards the construction of more complicated, energy-efficient machine learning systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1763699, 1740352, 1730309, 1717779, 156393.

Advait Madhavan acknowledges support under the Cooperative Research Agreement between the University of Maryland and the National Institute of Standards and Technology, Physical Measurement Laboratory. Award 70NANB14H209 through the University of Maryland.

Dilip Vasudevan was supported by the Advanced Scientific Computing Research (ASCR) program and funded by the U.S. Department of Energy Office of Science. Lawrence Berkeley National Laboratory operates under Contract No. DE-AC02-05CH11231.

Last but not least, the authors would like to thank James E. Smith, Georgios Michelogiannakis, David Donofrio, John Shalf, and the anonymous reviewers for their helpful comments.

References

[1] M. Bavandpour, M. R. Mahmoodi, and D. B. Strukov. 2019. Energy-Efficient Time-Domain Vector-by-Matrix Multiplier for Neurocomputing and Beyond. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2019), 1–1. https://doi.org/10.1109/TCSII.2019.2891688

[2] A. Bermak and D. Martinez. 2003. A compact 3D VLSI classifier using bagging threshold network ensembles. *IEEE Transactions on Neural Networks* 14, 5 (Sept 2003), 1097–1109. https://doi.org/10.1109/TNN.2003.816362

[3] Shekhar Borkar. 2010. Future of Interconnect Fabric: A Contrarian View. In *Proceedings of the 12th ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP '10)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1811100.1811101

[4] Leo Breiman. 1996. Bagging Predictors. *Mach. Learn.* 24, 2 (Aug. 1996), 123–140. https://doi.org/10.1023/A:1018054314350

[5] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[6] Vincent Chan, Shih-Chii Liu, and Andr van Schaik. 2007. AER EAR: A matched silicon cochlea pair with address event representation interface. *IEEE Transactions on Circuits and Systems I: Regular Papers* 54, 1 (2007), 48–59.

[7] S. Chen, Y. Wang, X. Lin, Q. Xie, and M. Pedram. 2014. Performance prediction for multiple-threshold 7nm-FinFET-based circuits operating in multiple voltage regimes using a cross-layer simulation framework. In *2014 SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–2. https://doi.org/10.1109/S3S.2014.7028218

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754

[9] Kyoungrok Cho, Sang-Jin Lee, Omid Kavehei, and Kamran Eshraghian. 2014. High fill factor low-voltage CMOS image sensor based on time-to-threshold PWM VLSI architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 7 (2014), 1548–1556.

[10] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

[11] W. Cui, Y. Ding, D. Dangwal, A. Holmes, J. McMahan, A. Javadi-Abhari, G. Tzimpragos, F. Chong, and T. Sherwood. 2018. Charm: A Language for Closed-Form High-Level Architecture Modeling. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 152–165. https://doi.org/10.1109/ISCA.2018.00023

[12] O Elkhalili, OM Schrey, P Mengel, M Petermann, W Brockherde, and BJ Hosticka. 2004. A 4/spl times/64 pixel CMOS image sensor for 3-D measurement applications. *IEEE Journal of Solid-State Circuits* 39, 7 (2004), 1208–1212.

[13] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. 2015. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*. 1117–1125.

[14] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. Syst. Sci.* 55, 1 (Aug. 1997), 119–139. https://doi.org/10.1006/jcss.1997.1504

[15] Jerome H. Friedman. 2000. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* 29 (2000), 1189–1232.

[16] Nicholas Frosst and Geoffrey E. Hinton. 2017. Distilling a Neural Network Into a Soft Decision Tree. *CoRR* abs/1711.09784 (2017). arXiv:1711.09784 http://arxiv.org/abs/1711.09784

[17] David Gunning. 2017. Explainable Artificial Intelligence.

[18] Xiaochuan Guo, Xin Qi, and John G Harris. 2007. A time-to-first-spike CMOS image sensor. *IEEE Sensors Journal* 7, 8 (2007), 1165–1175.

[19] Jung Kuk Kim, Phil Knag, Thomas Chen, and Zhengya Zhang. 2015. A 640m pixel/s 3.65 mw sparse event-driven neuromorphic object recognition processor with on-chip learning. In *VLSI Circuits (VLSI Circuits), 2015 Symposium on*. IEEE, C50–C51.

[20] Jaeha Kung, Duckhwan Kim, and Saibal Mukhopadhyay. 2015. A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*. IEEE, 85–90.

[21] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. 2008. A 128×128 120dB 15μs Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE journal of solid-state circuits* 43, 2 (2008), 566–576.

[22] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2014. Race logic: A hardware acceleration for dynamic programming algorithms.

ACM SIGARCH Computer Architecture News 42, 3 (2014), 517–528.

[23] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2016. Energy efficient computation with asynchronous races. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[24] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2017. A 4-mm 2 180-nm-CMOS 15-Giga-cell-updates-per-second DNA sequence alignment engine based on asynchronous race conditions. In *Custom Integrated Circuits Conference (CICC), 2017 IEEE*. IEEE, 1–4.

[25] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan. 2018. Low-Cost Sorting Network Circuits Using Unary Processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2018), 1–10. https://doi.org/10.1109/TVLSI.2018.2822300

[26] Shahrzad Naraghi. 2009. Time-Based Analog To Digital Converters.

[27] Cristiano Niclass, Mineki Soga, Hiroyuki Matsubara, Satoru Kato, and Manabu Kagami. 2013. A 100-m Range 10-Frame/s 340×96-Pixel Time-of-Flight Depth Sensor in 0.18-μm CMOS. *IEEE Journal of Solid-State Circuits* 48, 2 (2013), 559–572.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Christoph Posch, Daniel Matolin, Rainer Wohlgenannt, Michael Hofstätter, Peter Schön, Martin Litzenberger, Daniel Bauer, and Heinrich Garn. 2010. Biomimetic frame-free HDR camera with event-driven PWM image/video sensor and full-custom address-event processor. In *Biomedical Circuits and Systems Conference (BioCAS), 2010 IEEE*. IEEE, 254–257.

[30] Xin Qi, Xiaochuan Guo, and John G Harris. 2004. A time-to-first spike CMOS imager. In *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on*, Vol. 4. IEEE, IV–824.

[31] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[32] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen. 2008. Sparse Coding via Thresholding and Local Competition in Neural Circuits. *Neural Computation* 20, 10 (Oct 2008), 2526–2563. https://doi.org/10.1162/neco.2008.03-07-486

[33] S. Satpathy, K. Sewell, T. Manville, Y. Chen, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw. 2012. A 4.5Tb/s 3.4Tb/s/W 64x64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm CMOS. In *2012 IEEE International Solid-State Circuits Conference*. 478–480. https://doi.org/10.1109/ISSCC.2012.6177098

[34] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. 2017. A Survey of Neuromorphic Computing and Neural Networks in Hardware. *CoRR* abs/1705.06963 (2017). arXiv:1705.06963 http://arxiv.org/abs/1705.06963

[35] Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. 2013. A 128×128 1.5% Contrast Sensitivity 0.9% FPN 3 μs Latency 4 mW Asynchronous Frame-Free Dynamic Vision Sensor Using Transimpedance Preamplifiers. *IEEE Journal of Solid-State Circuits* 48, 3 (2013), 827–838.

[36] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science – VECPAR 2010*, José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[37] James E Smith. 2017. Space-Time Computing with Temporal Neural Networks. *Synthesis Lectures on Computer Architecture* 12, 2 (2017).

[38] James E. Smith. 2018. Space-Time Algebra: A Model for Neocortical Computation. In *Proceedings of the International Symposium of Computer Architecture (ISCA '18)*.

[39] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *CoRR* abs/1703.09039 (2017). arXiv:1703.09039 http://arxiv.org/abs/1703.09039

[40] Dilip Vasudevan, Anastasiia Butko, George Michelogiannakis, David Donofrio, and John Shalf. 2017. Towards an Integrated Strategy to Preserve Digital Computing Performance Scaling Using Emerging Technologies. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, Cham, 115–123.

[41] Paul N Whatmough, Sae Kyu Lee, Hyunkwang Lee, Saketh Rama, David Brooks, and Gu-Yeon Wei. 2017. A 28nm SoC with a 1.2 GHz 568nJ/prediction sparse deep-neural-network engine with > 0.1 timing error rate tolerance for IoT applications. In *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*. IEEE, 242–243.

[42] Clifford Wolf and Johann Glaser. 2013. Yosys–A Free Verilog Synthesis Suite. In *Submitted to: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip), Linz, Austria*, Vol. 10.