University of Glasgow | School of Computing Science

Machine Learning in the Linux Kernel for CPU Idle State Selection

Kajetan Puchalski (2457183P)

March 28, 2024

ABSTRACT

Modern processors are equipped with a number of power saving states which can be entered whenever the system has nothing to schedule on the given core - the idle states. They provide an attractive alternative to simply keeping the CPU active (i.e. busy-looping) by enabling a considerable decrease in power usage. However, while the CPU does make those states available, it is up to the operating system's kernel to decide which states should be used at what time. Selecting the most optimal state is a non-trivial problem as it would require knowing the future. To know which state is the most optimal, it is necessary to know the sleep length. The sleep length is only known at wake-up time, and not at sleep time when the decision needs to be taken.

The current state of the art approaches in the Linux Kernel rely on self-correcting heuristics which use past performance alongside system indicators to estimate which state is most likely to be optimal.

This paper shows the viability of employing Machine Learning in order to prepare and train a model that could then be used as a replacement for the currently existing heuristics. Firstly, it was established that it was feasible to train a model on collected kernel trace data and for that model to surpass the accuracy of the mainline heuristics on yet-unseen data. Secondly, it was established that such a model could be converted into a C header file and used directly inside the CPUIdle subsystem of the Linux kernel. The final governor was able to achieve an average accuracy of 0.87, an increase of 3.5% over the 'TEO' governor and 10% over the 'menu' governor. It was concluded that this ML-based approach has merit and there are no obstacles to using a Machine Learning model as a replacement for traditional CPUIdle governors.

1. INTRODUCTION

Idle states are available in every modern CPU. First introduced in the Intel DX4 (486DX4) as low-power mode circa 1995, they provide an alternative to wasteful busy-looping that enables substantial power saving. It is important to understand that the goal of these power saving states is by no means to provide suspend-like functionality, or even something that the users themselves would describe as "idle". The idle sleep length is measured in nanoseconds and it is very unusual for a sleep length to even approach a second. The purpose of this functionality is specifically to save power while a given CPU core does not have any instructions assigned to it by the system for execution. On a physical level, these idle states work by cutting the clock signal and power from the idle cores, reducing voltage and such.

The specific sequence of actions taken inside a CPU upon entering an idle state varies a lot between different manufacturers and CPU designs but it can typically be found described in detail in the reference manuals.

The amounts and types of available idle states vary a lot between different CPU architectures as well. Notably, Arm-based CPUs tend to only have 2 idle states available to them - shallow and deep [21]. In the Arm architecture, state0 is usually referred to as WFI (wait-for-interrupt) and already provides reasonable power savings with very little entry and exit cost. The deeper state - state1 has considerably higher entry and exit costs in exchange for providing better power saving, assuming that the CPU can be kept asleep for a long enough time to offset these entry and exit costs. On the other hand, CPUs using the x86 architecture tend to have considerably more states available - usually 6+ depending on the manufacturer and processor design [17]. In the x86 architecture, they are usually referred to as C-states. In contrast to Arm, the x86 state0 equivalent i.e. C0 does not save any power at all. Instead, it is a so-called polling state which simply means that selecting this state will instruct the CPU to perform busy-looping. The higher states e.g. C1-C6 will progressively cut the clock, voltage etc. as expected.

The CPU makes these states available but it is up to the operating system's kernel to request that a given state be entered into. This is the entire raison d'être of the CPUIdle subsystem. There is an implicit tradeoff involved in any idle state decision. Deeper states are able to save more power but they also incur higher entry and exit costs in the process, both in terms of the power usage and the latency. Because entering and exiting an idle state incurs a power cost in and of itself, that cost needs to be considered alongside the power usage while sleeping in said state in order to determine whether a given state would be an improvement over a shallower one for a given sleep length. In the Linux kernel, that tradeoff is modeled as a single number - the target residency. The target residency of a given idle state represents the lowest amount of time that the CPU would need to stay in said state in order for it to save more power than the closest shallower state would have. If the CPU is woken up by the system prior to the sleep length reaching (or exceeding) the target residency, more power will be wasted compared to a shallower state alongside increased latency. Thus, making the decision wrong from every perspective. It is clear to see that finding a way to make 100% accurate idle state decisions is an intractable problem as it would require knowing the future. At the point of the decision being made the operating system cannot know for sure when the particular CPU will be woken up next.

As of the time of writing this paper, the state-of-the-art solutions in the Linux kernel amount to self-correcting heuristics based on past correctness and overall indicators of system activity. These heuristics are referred to as **idle governors**, function as interchangeable components within the wider CPUIdle subsystem and can be switched at runtime using the kernel filesystem interface (sysfs) [27]. The accuracy they can achieve oscillates about 70%-80% depending on the governor and the workload in question.

In the modern era when most computing is done on mobile devices and so much attention is paid to sustainability, optimising power usage is more important than ever. Consequently, improving the accuracy of idle state predictions is particularly relevant in the context of mobile devices, such as Android phones. Since the vast majority of devices running the Linux kernel these days are mobile Android devices [14], a lot of developer time and effort is invested into mobile-specific kernel optimisations. For these reasons, in order to ensure that this research is as applicable and as relevant as possible, the test platform of choice for this project was determined to be a **Pixel 6** mobile phone running **Android 13** with a close-to-mainline **6.3 Linux kernel**. This is the device that all the training data was collected from, as well as the device that the final model implemented in the kernel was evaluated on.

To provide some more detail on why the Pixel 6 and similar Android phones make for an interesting test platform for research like this, it is useful to discuss what exactly the Pixel 6 CPU looks like. The Pixel 6 CPU is the original **Google Tensor G1** [34], model number GS101. It is a 5nm chip implementing the ARMv8.2-A ISA [4] manufactured by Samsung. Most importantly for this paper, as most if not all mobile chips these days, it employs the ARM big.LITTLE [5] heterogenous computing architecture. That is to say, the CPU is made up of several different types of cores. Some cores are more suitable for power saving and low-intensity tasks (LITTLE) while others consume more power but also offer much faster processing capabilities (big). The Tensor G1 chip is an octa-core CPU, meaning it is made up of 8 cores in total:

- 2x ARM Cortex-X1 at 2.8 GHz - The 'big' cluster
- 2x ARM Cortex-A76 at 2.25 GHz - The 'mid' cluster
- 4x ARM Cortex-A55 at 1.8 GHz - The 'little' cluster

It is up to the operating system scheduler to choose which tasks are allocated to which cores in order to exploit the potential of this type of architecture to the fullest. When it comes to CPUIdle, the most important thing to keep in mind is that each of the 3 different types of cores are effectively a different type of CPU and thus exhibit slightly different idle state-related behaviour. Luckily the interface for controlling the states, as well as the states themselves remain the same. The main difference is that the target residencies and idle state latencies are different for each of them. Fortunately, the way that the Linux Kernel code is written makes it very straightforward to manage these types of differences. The idle governor will effectively run and keep its counters on every core in separation, thus ensuring that the variations between different types of cores are handled appropriately.

2. BACKGROUND

2.1 The Linux CPUIdle subsystem

The Linux CPUIdle subsystem [22] manages the idle functionality of the CPU. Conceptually, it can be split into three parts - framework, drivers and governors. Firstly - the idle framework. The framework is used on every system that runs Linux and provides a standardised set of callbacks that drivers and governors can implement. Secondly - the drivers. The drivers are either device or architecture-specific and provide a bridge between the standardised idle framework and the hardware. Finally - the governors. The governors are heuristics which determine which state should be selected whenever the system decides that a given CPU should be put into idle sleep. This paper is only concerned with improvements on the governor level. It is important to note that a governor has no control over when (and whether) a CPU is put into idle sleep or woken up from it. Those decisions are taken by the scheduler, communicated through the framework and executed by the drivers. The only role of a governor is to determine *which* of the available states will be selected.

Since the Linux Kernel is designed to support both single and multi-core processors, kernel implementations treat every core as a separate logical CPU and use the term *CPU* to mean *CPU core*. Usage of the term *CPU* throughout this paper should be understood in that context.

As of the time of writing, there are four idle governors available in the kernel - menu, TEO, ladder and haltpoll.

The oldest one - **ladder** - was developed around 2001 and was the default governor until menu came about. It remains the default governor on systems where the scheduler tick cannot be stopped on idle CPUs.

On the other hand, on systems where the scheduler tick can be stopped on idle CPUs (usually referred to as tickless systems), the default governor is currently **menu** [29]. Menu was developed around 2006 in order to leverage the aforementioned ability to stop the scheduler tick on systems which support doing so. Stopping the scheduler tick on idle CPUs can be very beneficial as it avoids having to periodically wake idle CPUs up when the system knows that a given CPU has no actual work to do. Thus, tickless systems with an appropriate idle governor are in general able to be more energy-efficient.

The **TEO** (Timer Events Oriented) [28] governor then was developed around 2018 in order to address certain deficiencies in the "menu" decision-making process. The main idea behind TEO is that timer events are the primary source of wakeups in the system which is orders of magnitude more frequent than any other source. For that reason, the governor is structured primarily around said timer events with additional heuristics further aiding the decision-making process. In 2023, the TEO governor was extended to take scheduling data into account within its heuristics (specifically the average CPU utilisation) in order to avoid entering unnecessarily deep idle states during periods of high system activity.

The last idle governor - **haltpoll** - was developed around 2019 to work in conjunction with the haltpoll CPUIdle driver. It is a special-case governor as opposed to the other three general-purpose ones and is only intended for certain virtualised workloads which can benefit from its specific functionality.

2.2 The Linux Kernel

When discussing and working on any improvements to the Linux kernel, it is necessary to consider the type of software ecosystem that the kernel exists in given that it is very different from any other software project that one could work on.

To start with, the kernel is written almost entirely in C11 [25]. Recently some support for Rust was introduced and some components written in Rust are slowly appearing in the kernel codebase [40], but thus far at least this has not been the case for anything CPUIdle-related and so it is outside of the scope of this paper. More importantly, the kernel does not use or include the C standard library - the only facilities available to the kernel developer are the language itself and the library helper functions that are already part of the kernel. This includes the available data types.

Most importantly for the purposes of this paper, the use of floating point operations inside kernel subsystems is heavily discouraged. That is because the kernel keeps the FPU [3] switched off by default unless it is explicitly requested by a userspace application. Using floating point operations inside the kernel would require explicitly managing saving and restoring FPU registers which adds unnecessary overhead and is generally looked down upon unless it cannot be avoided [20]. This consideration is crucial for implementing the inference step of any trained Machine Learning model in kernel C - it is close to necessary to ensure that it will only use fixed-point arithmetic and integer data types.

The aforementioned issue becomes even more important in light of the main consideration - the overall performance. Kernel code always needs to be written with performance in mind. Particularly relevant for this paper, the code inside an idle governor will be called thousands of times per second on every CPU core available in the system. The CPU cannot enter idle sleep until the idle selection logic finishes executing. Overheads that could be negligible for a userspace program can and will make the entire system unusable if introduced into a kernel path as vital as the idle path. Kernel code also needs to keep its memory footprint to the minimum - any memory that the kernel allocates will not be available to userspace programs. Due to these constraints, it is vital to carefully consider any overhead that a program might introduce prior to attempting to execute it inside the kernel.

Unsurprisingly given the aforementioned considerations and the overall attitudes present within the kernel community, the kernel does not offer any Machine Learning-related facilities. There is no kernel Machine Learning library, subsystem, helper functions or anything of the sort. In order to perform any sort of Machine Learning tasks within the kernel, a potential model will have to be prepared and trained using external tools of some kind and then the inference step will have to be implemented in pure kernel C.

2.3 Machine Learning

For the purposes of this paper, the most relevant Machine Learning models to focus on are classifiers. Classifiers are models that can be trained to predict categorical labels (classes) given a set of input features. Those are the models that this paper is focused on for the simple reason that they fundamentally align with what a CPUIdle governor does - predict a state label given some input data available to the system.

Many different classifier models have been developed over the years, the most commonly used and notable types are discussed in the remainder of this section.

Logistic Regression [42] works by utilising regression in order to perform classification tasks. The training phase attempts to find a linear trend separating the classes within the dataset. Once found, this linear trend can then be used in the inference step to classify future yet-unseen datapoints simply by determining which side of the trend line they fall on. Naturally, the primary drawback of this approach is that it cannot be effective in cases when there is no linear trend in the underlying data to begin with.

Support Vector Machines (SVM) [10] are models that find a hyperplane separating the data points in order to classify them into separate categories. Linear by default, they can leverage the kernel trick in order to perform nonlinear classification. The kernel trick involves projecting the input data into a higher dimensional space where a hyperplane separating the data points can be found more easily than in the lower dimensional space.

Naive Bayes [16] models, as the name suggests, are models that leverage Bayes Theorem in order to attempt to make predictions on unseen data. They are referred to as *naive* because the models makes the assumption that all of their input features are completely unrelated - which is usually not the case. However, quite surprisingly, these types of models still tend to perform well at certain tasks in spite of that fundamental assumption being incorrect.

Neural Networks, such as the **MLP** (Multilayer Perceptron) [36] model are networks made up of an arbitrary number of layers with an arbitrary number of nodes. These nodes, referred to as neurons, mimic the way that neurons inside a human brain work and are effectively just nonlinear functions of their input. Such neural networks are universal function approximators. That is to say, given any function with an arbitrary number of inputs and outputs, there exists a neural network that could approximate this function without explicitly implementing it. The challenge of course is that such a network might not be feasible to find and operate.

Finally, **Decision Trees** [8] are models which operate by creating chains of decisions taken based on the values of the features present in the input data. They are best visualised as long chains of if-else statements where at every step a value of a feature is compared to some constant (found in the training stage) and then the result of that comparison determines which branch of the tree is taken next. Once a leaf node is reached, the classifier returns the corresponding class label as its decision. Decision Trees are most often used in ensembles, i.e. sets of several trees trained and used in parallel. The decisions taken by each tree in the ensemble are then aggregated as "votes" and the class voted on by the highest number of trees is returned as the classifier's final decision. The most popular example of such a decision tree ensemble is the **Random Forest** [7] classifier.

As an extra consideration, it also worth discussing the option of using Deep Learning [11] models. While very popular and frequently utilised for a variety of modern use cases, Deep Learning models are extremely heavy compared to the type of code present in and required by the Linux Kernel. Such models have tens of thousands or even millions of parameters which would all need to be kept in the system memory at all times.

Compared to kernel execution timeframes, they also tend to take unacceptably high amounts of time to generate predictions. Additionally, most modern applications rely on running these types of models on GPUs which is simply not an option for kernel code. Succinctly put, the overhead of Deep Learning models is simply considerably too high for them to ever be feasible to use in a kernel context.

3. PROPOSED APPROACH

This paper describes research into exploring and evaluating the feasibility of leveraging Machine Learning in order to improve the accuracy of idle state predictions. The fundamental idea is rather simple. Preexisting heuristics provide a way to select an idle state based on a number of system counters and statistics about the governor's own accuracy. Because the selections are effectively educated guesses, it is reasonable to assume that these heuristics could be improved upon in some ways. Since it is straightforward to collect data from a Linux system at runtime e.g. using ftrace, this problem can easily be formulated in the form of supervised learning. Pre-collected data from various system counters sampled at each state selection can be used as input data and it can also be matched with the sleep time collected at each wakeup in order to determine what, in hindsight, the optimal state selection for that set of input data should have been. Such obtained input data and labels can then be used to train, as well as evaluate the model using usual Machine Learning infrastructure and frameworks, e.g. Python and scikit-learn [37]. Once a satisfactory model is trained, the next logical step is to find a way to export it into kernel-compatible C and integrate it into the kernel as a new CPUIdle governor alongside the already existing ones. Such a modified kernel could then be installed on a given device, the governor switched to at runtime and evaluated using the same tracing infrastructure that was previously used for collecting trace data in the data collection step.

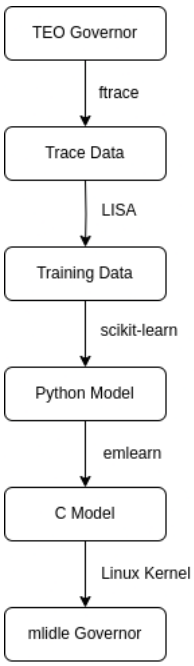

Figure 1: Methodology flowchart

4. METHODOLOGY

4.1 Model selection

With the considerations described in the background section in mind, the primary Machine Learning model selected as the best candidate for this task was the Random Forest classifier. Conceptually, this is the model that is most closely aligned with the task that an idle governor is supposed to perform - making decisions by going through a process of evaluating each input feature one by one then branching out.

It is unlikely that there is a linear, or even quasi-linear relationship between system indicators and the idle state selection and thus Logistic Regression was excluded from the get-go.

Support Vector Machines could potentially be an interesting option to explore and might give reasonable results, but unfortunately the emlearn project used to export the trained models into kernel C had no support for this type of model. Given that it would not have been possible to implement such a model in the kernel within the scope of this research, SVMs were excluded from the list of potential models.

Naive Bayes models tend to be used for different types of prediction - e.g. predicting whether a given message is a spam or not, predicting whether a patient might be suffering from a certain disease given their medical data and so on. Conceptually, this is fundamentally different than what an idle governor tries to achieve, which is to take a decision based on the ensemble of its input data. For that reason, Naive Bayes models were also excluded from this research.

From there, the two remaining candidates were Decision Trees and Neural Networks. While both of these approaches have the potential to perform with similar accuracy, Decision Tree models are easier to implement and operate in kernel C. That is primarily because a trained Decision Tree model can simply be implemented as a sequence of if-else statements in C without any more complicated functions or features. Furthermore, Neural Network models such as the MLP usually require their input data to be normalised in order to achieve optimal performance. This normalising step would contribute to the overhead as well and thus it is preferable to try and avoid it. Of the Decision Tree models, Random Forest is the most flexible and ubiquitous option and so for that reason it was the primary model selected for the purposes of this paper. A secondary model, namely the MLP, was also selected and trained as a potential alternative and for comparison purposes.

4.2 Data collection

After selecting the model, the test environment for data collection was prepared. Data used for training the model was collected using standard Linux kernel tracing facilities i.e. **ftrace** [24]. The data collection methodology was rather straightforward.

First, the TEO CPUIdle governor was extended to log all the possible decision data into *ftrace* at decision time. Further tracing was added at wakeup time to log the outcome of each sleep (i.e. final sleep time and selected state) as well. In order to ensure that decision data was correctly matched to wakeup data, a per-CPU sleep_id variable was introduced to allow easily connecting the traces in spite of them being collected at different times.

With such a setup, large amounts of data were collected from the system while running a variety of different workloads and benchmarks in order to cover a wide range of use cases. The primary focus was put on **Geekbench 6**, **JetNews**, **Speedometer 2** and **DrArm**.

The tooling used for this data collection was Arm software's workload-automation suite which is a set of convenience tools for running and instrumenting various workloads. Workload-automation collects ftrace data by running the standard Linux **trace-cmd** [39] command simultaneously with the workload while taking care of all the implementation details regarding instrumentation, starting and stopping the collection, pulling resulting traces from the test device onto the host machine and so on. The tooling supports other methods of collecting data as well, although for this research only *ftrace* was used.

All the aforementioned workloads were ran in sequence for a predetermined number of iterations. The number of iterations varied depending on the benchmark based on its execution time, its impact on the thermal situation of the device and so on. For instance, the Geekbench 6 workload was ran for fewer iterations than the other workloads due to it being a heavy benchmark which makes the device reach its thermal throttling point fairly quickly. Once thermal throttling kicks in on the device, the benchmark results become very unreliable because the code paths that the scheduler takes to make its decisions change in order to try and manage rising device temperature. On account of trying to manage thermal throttling and achieving consistently comparable test results, the phone was purposefully cooled down to the same starting temperature before running each worlkoad (although not in between specific iterations). This was achieved by rebooting the phone, keeping it in a stable 5°C temperature environment for exactly an hour and only then running the tests. This was the methodology used for initial data collection from TEO as well as all further governor evaluation tests. The numbers of iterations used for each of the workloads were as follows:

- Geekbench 6 - 3 iterations
- JetNews - 10 iterations
- Speedometer 2 - 10 iterations
- DrArm - 10 iterations

Once collected into a trace file, the data was processed using Python trace analysis tooling, primarily Arm software's Lisa and the author's own tool called workload-processor. The goal of the processing was to match the decision data with the outcome data and thus to have all the necessary training data contained within one Python (specifically Polars) dataframe. Having been processed in this way, the data was available as a Polars dataframe containing all the decision data and sleep outcome data alongside the simulated optimal state data. That is to say, the sleep length obtained at wakeup time was used to determine which state would have been the most optimal one to select. The most optimal state could easily be determined by comparing the actual sleep length with the target residencies for the given CPU core. In conjunction with the input decision data, this amounted to the full set of input features alongside their target labels needed to successfully train any supervised learning model.

4.3 Benchmark workloads

Geekbench 6 [35] is a cross-platform CPU-heavy benchmark. It also supports GPU benchmarking, although that part of it was not relevant or used for this research. The benchmark itself consists of two phases - the single-core phase and the multi-core phase. During each phase, the benchmark spawns many small tasks performing CPU-bound workloads such as file compression, clang compilation, text processing, object detection, ray tracing and more. The benchmark measures how long these tasks take to finish and computes a score based on the length of their runtime. In the multi-core phase, these tasks are spawned across every core in the system - effectively flooding the entire CPU in order to determine the upper bound of system performance.

JetNews [13] is a UI workload based on Google's official Jetpack Compose sample under the same name. It simply automates common UI tasks such as scrolling up and down, switching screens and such while measuring dropped and janky frames. *Janky frames* is an Android UI pipeline term denoting frames that finished rendering past their rendering deadline, thus resulting in *jank* that could have been visible to the user. In terms of system activity it is a very similar workload to e.g. browsing Twitter, which is why it is a useful workload to include for these types of tests.

Speedometer 2 [38] is a web browsing benchmark that measures performance of web frameworks. It uses several different web frameworks to simulate common user actions such as scrolling, opening menus, adding and removing list items and such. The responsiveness of these frameworks is measured and a final score is computed based on it. Speedometer results change a lot with different browser versions, which is to be expected, but as long as the browser version is fixed it can provide a good indication of how changes to the kernel might affect typical web browsing tasks.

DrArm [18] is a Unity [43] video game, developed internally at Arm for benchmarking and evaluation of gaming-based workloads. It is a fairly typical Unity game with a replay feature that automatically makes the game character perform a predetermined sequence of steps at every launch. This results in a repeatable, comparable setting which can be used to evaluate what impact certain kernel changes might have on an example gaming workload. Including this type of workload was beneficial because the Unity game engine does not use the default Android rendering pipeline.

For some light background, the vast majority of Android apps use the same rendering pipeline built into the Android framework [1]. This makes it easy for app developers to create their desired apps without needing to worry about particular details of rendering performance. This one-size-fits-all approach is completely appropriate for the vast majority of use cases, including certain simple mobile games. For instance, the aforementioned JetNews workload uses the Android rendering pipeline. On the other hand, developers who require a more flexible approach in order to extract even more performance from devices tend to opt for implementing their own rendering pipelines using the Android NDK [12] as opposed to using the Google-provided facilities. This is the case for the Unity engine and this is why it was a useful addition. Including a workload that uses a custom-made rendering pipeline allowed for determining in what ways these specific changes to the idle decision making process impact sensitive and performance-critical software frameworks that were not explicitly tuned with these changes in mind.

4.4 Training set preparation

Because the training data was collected from several different workloads and benchmarks, it was prudent to prevent any specific workload becoming over represented in the training set. That is because the end goal was to train a general-purpose model that would perform well across a variety of workloads. By training the model either exclusively or primarily on one of the workloads we would risk the model becoming overly attuned to the characteristics of that one workload. Due to hardware constraints and also partly due to empirical testing, it was decided that the upper bound for the number of sleeps in the training data should be about *10 million*. This number has no particular meaning attached to it and was selected primarily because trying to train the model on more data was taking considerably more time while it also did not appear to increase the decision accuracy in any worthwhile fashion.

Early on during the experimentation phase, it was observed that the aforementioned DrArm Unity gaming workload exhibited different characteristics than the other 3 workloads. Namely, the baseline decision accuracy of the TEO idle governor was consistently around 95% across all different runs of this workload. For comparison, that same baseline accuracy for runs of the 3 other workloads was around 80%. This difference indicated that, at least as far as the idle state selection was concerned, this workload was almost entirely predictable and already well catered for by the existing TEO heuristics. The implication of this conclusion was that it was very likely that a decision model would learn close to nothing new from this training data. In order to verify this hypothesis, several models were trained on datasets both including and excluding the DrArm training data. No substantial difference was found between the accuracy of these two variants and thus it was concluded that it was safe to exclude DrArm data from the main training dataset. However, the data was still collected and retained in order to be available for use when evaluating the model later on.

Having excluded DrArm data from the training set, the training dataset was split between the 3 remaining workloads - Geekbench, JetNews and Speedometer. In order to ensure the aforementioned even split in the training data, it was decided that the target size of the training set would be *9 million* samples - *3 million* per workload. Once again, this number had no particular meaning or relevance attached to it beyond convenience and empirical testing results. The methodology described in this paper would likely work well with many different total numbers of samples.

4.5 Feature selection

As is commonly the case in Machine Learning contexts, feature selection is one of the most important aspects of training a successful model. It is particularly important for decision tree models. Due to their tree-like structure, decision tree models can grow exponentially in size when new features are added into their training set. If the newly added features do not provide substantial benefits, whatever benefits they might have can and frequently will become offset by the increased overhead of the model itself. Through empirical testing alongside feature importance evaluations using scikit-learn, 7 features were selected for inclusion in the final Random Forest model.

The Random Forest implementation in scikit-learn exposes the model's internal feature importances computed using the Mean Decrease in Impurity (MDI) method [7]. However, impurity-based importance metrics can be biased and unreliable when paired with high cardinality (i.e. non-categorical) features such as the ones in this training dataset. For this reason, feature importance was instead evaluated using scikit-learn's implementation of the Permutation importance method [2] which tends to be more reliable in scenarios such as the one discussed in this paper.

The 7 features selected for inclusion in the final model, along with their mean permutation importances, were as follows. The included permutation importance values indicate the impact that randomly shuffling the values of that feature had on the degradation of the model's baseline score. They were computed on the entire training set, although the values computed on the test set were very similar.

- Idle state1 hit metric - 0.0565 - Number of times in the recent past that state1 was correctly selected by the governor on this CPU based on the measured sleep lengths. Computed and decayed over time in the same way as in the mainline TEO [28] implementation.
- Time until the next timer event - 0.0378 - Time until the next timer event [30] is the only completely predictable and guaranteed source of an idle wakeup in the system, in addition to being the most common one. It is effectively the upper bound for how long the CPU will remain in idle sleep. It is readily available in the Linux Kernel and used by several idle governors.
- Idle state0 interception metric - 0.0163 - Metric representing the number of times in the recent past that state0 should have been selected instead of state1 on this CPU based on the measured sleep lengths. Computed and decayed over time in the same way as in the mainline TEO [28] implementation as of the writing of this paper.
- Average CPU utilisation - 0.0076 - Linux Kernel scheduler abstraction representing the degree to which a CPU is 'busy' or 'utilised' as a number between 0 and the maximum capacity of the CPU core [23]. Maximum capacity is always 1024 on the biggest core in the system and proportionally less on smaller cores. Alternatively, in the case of homogenous CPU architectures, maximum capacity is 1024 for every core in the system.
- CPU ttwu count delta - 0.0035 - Number of threads the scheduler attempted to wake up on this CPU between the last wakeup and current idle state selection. Computed as a delta of the raw count tracked by the Linux Kernel scheduler [26].
- CPU rq_cpu_time delta - 0.0030 - Amount of runtime recorded for tasks on this CPU's runqueue between the last wakeup and the current idle state selection. Computed as a delta of the raw time tracked by the Linux Kernel scheduler [26].
- CPU max capacity - 0.0021 - Maximum capacity of the given CPU core. Upper bound for the utilisation feature, allows the decision tree to distinguish between different types of cores in the system [23].

4.6 Model hyperparameter selection

When it comes to Random Forest models, the main two hyperparameters are the number of estimators (i.e. the number of Trees in the Forest) and the maximum depth of each Tree.

As a general rule, optimal maximum depths tend to oscillate around the number of features in the training set. This can easily be seen by considering what a fully grown Decision Tree looks like. If the maximum depth is lower than the number of features, some features will have to be excluded by the Tree and thus will not be taken into account. This can very easily lead to underfitting [9] the model. Conversely, if the maximum depth is much higher than the number of features, the model might start to learn the training dataset instead of just learning the generic patterns. This results in overfitting [9]. For those reasons, the target maximum depth of each Tree was determined to be a number just above the number of features, e.g. 8 for a training dataset containing 7 features. Furthermore, the size of each Tree grows exponentially with its depth and so deeper Trees are particularly ill-advised in this case on account of the overhead.

As for the number of estimators, various different numbers can work equally well. Due to the overhead and latency constaints, attemps were made to find the lowest possible number of estimators achieving satisfactory performance. This target number was empirically determined to be around 5.

Having selected the target hyperparameter values, the grid search with cross-validation [41] technique was used on a range of values neighbouring them in order to find the optimal hyperparameters for the model.

4.7 Model evaluation

In order to evaluate the model, the same benchmarks and workloads that were used to collect the training set were run again in order to obtain an evaluation set of the same size. The ability to do this was a considerable benefit of using kernel tracing to collect training data. In most Machine Learning problems, the training dataset is limited and the evaluation set needs to be taken out of it. In this case, the source of training and evaluation data is effectively infinite and so more new data that the model has not seen in the training stage can always be easily collected.

Having collected the evaluation data, the trained model was evaluated by comparing the prediction it generated for each datapoint with its optimal target label. An accuracy score obtained in such a way was then compared against the baseline accuracy scores obtained from idle governors currently implemented in the Linux mainline.

With such a methodology, it was possible to evaluate multiple trained models simply by running simulations in Python, without actually having to implement the models within a governor and run them on a real-world device. This method was likely not a perfect reflection of how the model would influence a live Linux system, but it nonetheless provided a very good and reliable indication of the type of accuracy score that could be expected from a governor based on said model.

4.8 Kernel governor implementation

In order to implement the ready model inside the Linux Kernel, the emlearn[32] project was used. Emlearn is a Machine Learning inference engine designed for exporting models trained in scikit-learn [37] or Keras [33] into portable C99 code. Using emlearn, this trained model could easily be converted into a C header file. This generated header file needed some manual adjustments, primarily adjusting the data types to what they should look like in Kernel C and including otherwise inaccessible imports. Given that emlearn is an open source project available on GitHub, one area of future work could be submitting a contribution to the project that would add a 'Kernel C' model export mode in order to avoid the need to adjust the generated header files by hand.

The biggest benefit of utilising emlearn is that it already supports exporting the models using fixed-point arithmetic. This made it very straightforward to be able to implement the model without the need to account for the aforementioned issues with using floating-point operations inside a kernel subsystem.

Having prepared the trained model in the form of a self-contained kernel-friendly C header file, a new CPUIdle governor was created using the current TEO implementation as a template. To be precise, the code present within TEO to collect and update its own statistics was kept mostly unchanged so that the statistics could be utilised as features for the model. Necessary code to collect and keep track of the new scheduler-related feature deltas was added. Finally, the entire heuristic part of the TEO state selection function was removed and replaced with a call to the prediction function of the model that was contained in the included header file. The prediction generated by the model was then treated as the decision taken by the new ML-based heuristic and the selection function continued on to perform its final checks of whether it was necessary to stop the scheduler tick or not before returning.

A major advantage of this type of approach is that the governor itself, called **mlidle** [1] in the implementation, only calls into the model which is located in a separate header file. This results in what is effectively a "plug and play" setup where it is very easy to swap out differently trained models without the need to modify the governor implementation. If some features need to be added or removed, only very minor changes to the governor are required so that it properly reflects what the model is expecting.

4.9 Governor evaluation

With the new governor implemented inside the kernel and the model bundled with it, final evaluation of the running governor was very straightforward. The only steps needed were to flash the kernel onto the device, reboot, switch to the new governor using sysfs at runtime and run these same workloads and benchmarks in the same way as it was done when collecting data for training and evaluation.

Having collected the data, the results were then compared against those obtained with traditional CPUIdle governors in order to assess how this proposed approach performed in a fully ready, real-world setting.

[1] All code written for this paper can be found on GitHub: https://github.com/mrkajetanp/ml-idle

4.10 The MLP Model

The final aspect of the methodology was the aforementioned MLP model, selected as the alternative model for evaluation and reference purposes. Several MLP models were trained on the same training sets as the Random Forest models and also evaluated in the same way through simulations in Python.

Unfortunately, the emlearn project as of the writing of this paper did not support fixed-point arithmetic for exporting MLP models. As a result, the MLP was never implemented and evaluated inside the kernel. Implementing fixed-point arithmetic support for these models in emlearn was outside of the scope of this project while having to manage FPU registers in the kernel would have increased the overhead beyond a point where the governors could be reasonably compared.

Nonetheless, it was informative to determine whether a simple neural network was also capable of learning the patterns necessary for making mostly-optimal idle state selection decisions.

5. RESULTS AND DISCUSSION

5.1 Prediction accuracy

The most important metric that this paper focuses on is the idle state prediction accuracy. This is because prediction accuracy is the most reliable, independent metric that is both easy to determine and easy to compare. Prediction accuracy is defined simply as the fraction of all state predictions that resulted in the most optimal state being selected.

Which state was the most optimal can easily be determined by comparing the final sleep time recorded at wakeup time with the target residencies of available idle states. The most optimal state will always be the deepest state with target residency lower than the recorded sleep time. In case of the test device for this paper being Pixel 6, and in case of Arm-based devices in general, this determination is very easy to make as it only requires comparing the recorded sleep time with the target residency of state 1 - if it is higher, the optimal state is state 1. Otherwise, it is state 0.

In theory, based on how the target residency is defined within the CPUIdle framework, governor accuracy should be a direct proxy indicator of at the very least power usage - increasing accuracy should always decrease power usage. In practice, somewhat unsurprisingly it is always more complicated than that. Firstly, power usage can be affected by close to every other subsystem of the kernel and so it is often difficult to determine whether changes in power usage are directly attributable to a change in idle state patterns, indirectly attributable or even coincidental. More importantly, the target residency values are hardcoded in the Device Tree [19] files that implement support for the given SOC (System-on-Chip). This means that they are effectively arbitrary numbers put in the kernel by the device vendor. The kernel assumes that the values are 100% truthful, but it does not necessarily have to be so. Certain vendors have a tendency to use those values as performance knobs and simply tune them to achieve the best possible performance with whichever CPUIdle governor happens to be in use during the development stage. Such tweaking then poses challenges when trying to develop and test a new governor, as in the case with the research described in this paper.

Even without any specific intent from the vendor side, these values are by no means straightforward to measure and determine with a degree of certainty. Obviously, the problem becomes even more pronounced once we consider the separation between chip designers, chip manufacturers and kernel developers implementing support for the chip. In most cases, all 3 of the aforementioned groups will be working for different companies in different parts of the world which makes communicating this type of information all the more difficult.

Despite all the caveats concerning the accuracy-first approach, centering this type of research around improving accuracy first and foremost is still the most reasonable approach. Attempting to account for potentially misleading residency values is unlikely to lead to anything useful and would almost certainly cause issues in the long run. If a model can successfully learn to optimise for increased prediction accuracy then it has the best chance of improving performance and power usage across a range of different devices and the best chance to be easy to retrain and adapt for previously unseen target platforms.

Workload	menu	TEO	RF-8	RF-10	MLP
Geekbench	0.816	0.784	0.810	0.814	0.481
JetNews	0.696	0.903	0.903	0.903	0.826
Speedometer	0.780	0.850	0.852	0.853	0.742
DrArm	0.869	0.951	0.951	0.951	0.762

Table 1: Simulated prediction accuracy

Table 1 shows the prediction accuracy values obtained with different CPUIdle governors and trained models. The *menu* governor was included for reference. For consistency, the accuracy values for trained models were obtained through Python simulations on the same pre-collected dataset as the TEO values. That is to say, the values for menu come from real-world tracing. The values for TEO, Random Forest & MLP models come from running a simulation on the collected TEO values. The intent of this approach was to compare 1:1 the decisions that would be made if the governor was presented with the exact same sequence of requests and sleep patterns as the TEO governor had been. This approach is not 100% reflective of real-world performance but still gives a good idea of comparative decision-making accuracy. Real-world empirical accuracy values obtained from running those trained models on the test device can be found later in this section. The following list describes the meanings of each of the columns.

- *menu* - The menu idle governor. Included in the Linux mainline, default on most devices.
- *TEO* - The TEO idle governor. Included in the Linux mainline, newer alternative to menu.
- *RF-8* - The Random Forest model trained on the aforementioned 7 features with 5 estimators and maximum depth of 8.
- *RF-10* - The Random Forest model trained on the aforementioned 7 features with 5 estimators and maximum depth of 10.
- *MLP* - The Multilayered Perceptron model trained on the aforementioned 7 features using the lbfgs [31] solver.

Two variants of the Random Forest model were included for completeness and to visualise the trade-off inherent in selecting the model's hyperparameters. While models with higher maximum tree depth can achieve better accuracy under certain circumstances, they also take up much more space and require much more C code to be represented. This obviously translates into kernel overhead. To visualise the difference, the 8-deep Random Forest model after exporting to a C header file took up 258 KB and contained 6100 lines of code. The 10-deep model on the other hand took up 1 MB and contained 21186 lines of code. That is to say, the size of the model roughly doubled with every extra level of depth for the trees.

As can be seen in table 1, on the whole the two trained Random Forest models achieved the best accuracy out of all the available options. It is particularly interesting to note how *menu* performed better than *TEO* on Geekbench but much worse on the remaining 3 workloads. Random Forest on the other hand was able to achieve the "best of both worlds" and obtain either best or close-to best results for all 4 tested workloads, no matter which *Random Forest* variant was considered. The *MLP* model, somewhat unsurprisingly performed worse than the *Random Forest* across the board. This was in line with the initial prediction of it being less suitable to the task, although theoretically it should still be possible to train a neural network to achieve better results than the ones recorded in the table. Interestingly, even that *MLP* model was able to outperform the mainline idle governors on certain workloads.

Outside of the Speedometer workload, the difference between the accuracy of the two *Random Forest* variants was surprisingly small. It indicated that the benefits of increasing the maximum depth of the trees beyond that specific point were most likely not worth the added overhead. Given that the number of features the model was trained on was 7, these results appear to confirm the previously stated observation that the optimal maximum depth of the trees should be either exactly the same or just over the number of features in the dataset.

Workload	menu	TEO	mlidle-8	mlidle-10
Geekbench	0.816	0.782	0.804	0.805
JetNews	0.696	0.794	0.797	0.784
Speedometer	0.780	0.834	0.924	0.837
DrArm	0.869	0.952	0.953	0.951

Table 2: Measured prediction accuracy

Table 2 shows the prediction accuracy values obtained from running the fully implemented governor on the target device. The reported numbers represent the accuracy as seen by the CPUIdle framework. Following the same pattern as with the previous simulations, numbers 8 and 10 refer to the maximum depth of the decision tree.

In terms of the results, it is worth reminding that not all missed sleeps are equal. For this type of test platform, "too shallow" sleeps are much preferable to "too deep" sleeps. The reasoning for why that is was described in the Introduction section. With that in mind, it is important to highlight that for the *menu* governor and GB6, 15.7% of all sleeps were "too deep" misses and 2.5% were "too shallow" misses. For *mlidle-10*, that same split was 9.2% "too deep" and 10.2% "too shallow". Thus, *mlidle-10* was still preferable.

5.2 Power usage

The second most important metric to consider is power usage. As discussed in the previous section, in theory power usage on the device should decrease in line with the increases in prediction accuracy but in practice it is not always the case. Irrespective of that it is always beneficial to track the average power usage when benchmarking any sorts of changes to the kernel in order to make sure those changes do not come with any undesirable side effects. The test device, i.e. Pixel 6, comes with a built-in power meter. The power driver exposes the power meter via a sysfs [27] energy counter, enabling measurements of power usage with milisecond precision across various power rails available in the device. Table 3 contains aggregate measurements of power usage over time obtained while running the aforementioned numbers of iterations of each of the 4 test workloads. The exposed power meter provides its values as $power[mW] \times time[ms]$, i.e in microJoules. The unit for the values below is mW and the values were obtained by dividing the measurements over the difference in time.

Workload	menu	TEO	mlidle-8	mlidle-10
Geekbench	1754	1856	1768	1794
JetNews	173	167	171	170
Speedometer	1667	1720	1734	1741
DrArm	1960	1899	1864	1835

Table 3: Power usage [mW]

In line with the previously indicated expectations, the power usage results were not entirely clear cut in terms of matching up with the differences in prediction accuracy. For Geekbench, the theoretical relationship was preserved. The most accurate governor (menu) used the least power while the least accurate governor (TEO) used the most power. This was also the case for JetNews, albeit which exact governors were the most and least accurate differed between the workloads. Speedometer was the workload that went against this theoretical expectation - the least-accurate governor, i.e. menu was also the most power efficient one. The DrArm workload placed somewhere in between those two extremes with the mlidle governor achieving best power efficiency despite its accuracy being virtually the same, or even slightly lower, than the TEO governor. These types of differences are to be expected and often cannot be explained to a satisfying degree due to the sheer amount of different variables and factors that play into the entire picture of performance of an Android phone.

The differences between the two *mlidle* variants were not entirely conclusive either. For some workloads, the *mlidle-10* variant was using more power. For others, the *mlidle-10* variant was using less power.

The most important finding from the power usage readings, however, was that for the most part a relationship can be drawn between the idle state prediction accuracy and the power usage. This further reinforced the conviction that opting to focus this research around optimising prediction accuracy was the correct choice. Furthermore, the fact that no actually problematic spikes in power usage were found was a good confirmation of there not being any unforeseen power usage issues brought about by introducing even a relatively large ML model into the idle decision path.

5.3 Benchmark scores

In addition to accuracy and power usage, the third major metric to take into account are broadly-defined benchmark scores. Benchmark scores can be used to estimate the impact that a given kernel change could have on the performance of various types of workloads. What constitutes a benchmark score differs between various workloads. For the 4 workloads considered in this paper, Geekbench and Speedometer produce numerical benchmark scores after each iteration and so those can be used directly. JetNews and DrArm do not produce such scores as they simply simulate a type of system activity. In those cases, frame rendering data such as frames per second or "jank percentage" can be used as a replacement for more traditional benchmark scores.

Workload	menu	TEO	mlidle-8	mlidle-10
GB6 (single-core)	929	990	943	950
GB6 (multi-core)	2236	2333	2236	2277
JetNews - FPS	43.2	42.5	42.8	42.9
JetNews - Jank%	0.4	0.4	0.5	0.6
Speedometer	68	69.5	65.2	70.6
DrArm - FPS	49.9	48.4	44.9	46.2
DrArm - Jank%	0.7	0.5	0.6	0.6

Table 4: Benchmark scores

The various types of benchmark scores are even more of a mixed bag than the power meter readings. There is no clear theoretical relationship to be drawn in this case. In theory, since shallower states always have lower latency than deeper states, the easiest way to maximise performance would be to always select the shallowest available state. This would ensure that there is no unnecessary latency added to the system through CPUIdle. In practice however, this approach does not in fact maximise benchmark scores in the long run. This is because avoiding the opportunity to save power through deeper idle states leads to the device heating up faster, which in turn results in the kernel thermal throttling facilities kicking in. Thermal throttling, unsurprisingly, results in decreased performance. Conversely, selecting overly deep states will unduly increase latency while also wasting power. Thus, there is no precise rule as to how changes to idle selection patterns will affect performance of certain workloads. In practice it is largely random chance. On a case-by-case basis, sometimes certain idle decisions will result in better performance, sometimes worse and sometimes they will have no impact at all.

When it comes to the exact results obtained for this paper, they can be found above in Table 4. It is interesting how *mlidle-10* outperforms *menu* in both Geekbench 6 metrics but does not outperform *TEO*. On the other hand, when it comes to Speedometer, *mlidle-10* outperforms both the mainline governors. In JetNews, *mlidle-10* performs ever so slightly worse than *menu* but better than *TEO*. When it comes to the difference with *menu*, it is worth reminding that the power usage was lower with *mlidle* which arguably is the more important metric for a UI-based workload where a third of a frame per second is less relevant than preserving battery life.

Across the board, *mlidle-10* outperformed *mlidle-8*. This was all the more noteworthy given that the differences in prediction accuracy between those two were either negligible or in favour of *mlidle-8*.

5.4 Idle state residency

While not exactly a metric useful for evaluating comparative governor performance, CPU idle state residency represents an interesting window into the overall effects that a governor has on the system. Namely, the idle state residency is the aggregate amount of time that a given CPU (core) spent in a given idle state. By comparing idle state residencies between governors it is possible to determine whether a particular change to the decision making process makes the governor more likely to select state 1 or more likely to select state 0 and so on in a way that accounts not just for the sheer number of sleeps, but also for their duration.

Using pre-collected trace data, it is also possible to simulate 'optimal' idle state residency - i.e. what the idle state residency would be if the governor's decision accuracy was 1.0. By doing so it is possible to compare the actual residency with the ideal residency and see how far off the evaluated governor is. Of course, there is no telling how achieving that 1.0 accuracy would affect either performance or power usage.

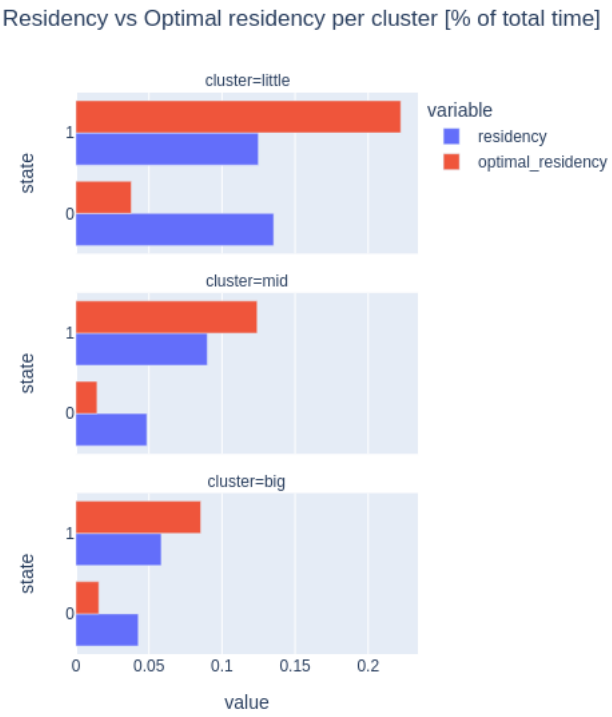

Figure 2: Simulated optimal idle state residency

Figure 2 shows the difference between the actual and simulated optimal idle state residency as computed on the entire training dataset. Since the training dataset was collected by running the TEO governor, it showcases the difference between the decisions taken by said governor and the theoretically most optimal decisions. TEO errs towards selecting state 0 too much because the penalty for doing so incorrectly is less than the penalty for incorrectly selecting state 1. Thus, the total measured residency of state 1 will always be lowered than the optimal residency would be.

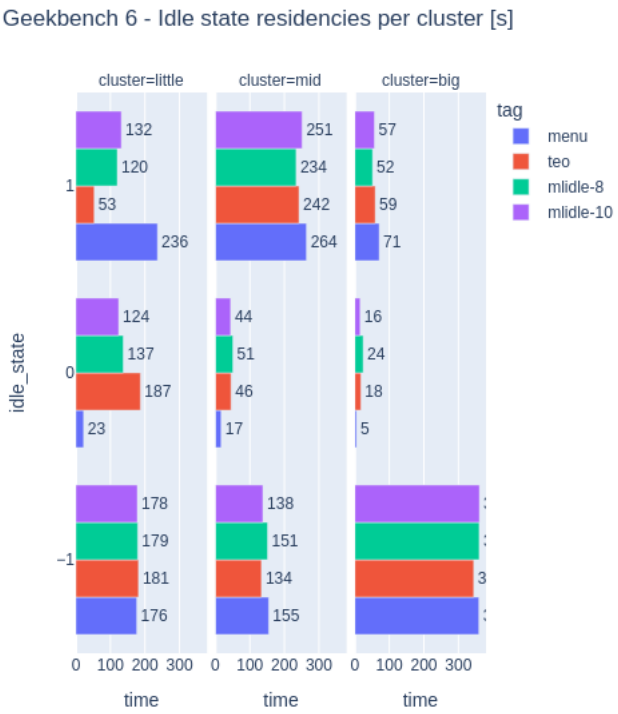

Figure 3: Geekbench 6 - Per cluster idle residencies

Figure 3 shows the idle residencies measured for all four tested governors across all three CPU clusters in the system. The *idle_state* column naturally shows the idle state in question while the *time* column shows the time spent in said state in seconds. The *-1* 'state' represents non-idle time, i.e. time when the cluster was actively running a task and not using CPUIdle. The values show an average of the residency values obtained for all 3 iterations of Geekbench. This plot serves particularly well to explain the difference in Geekbench 6 benchmark scores obtained by the respective workloads. Menu chooses state 1 by far the most and it also scores the lowest on this benchmark. TEO chooses state 1 the least on the whole and consequently scores the highest.

For brevity, similar plots for the remaining 3 workloads will be omitted. The plot in Figure 3 was included specifically to visualise how big of an effect idle governor changes have on the idle residency picture and what the idle residency picture looks like overall.

The results on said plot appear to confirm the overall hypothesis on what benefits *mlidle* brings in practice. It is able to achieve a kind of middle of the road performance point between menu and TEO. The new governor shifts the decision making for this workload more towards state 0 to achieve better performance than menu while also not going as far as TEO and achieving better power efficiency than TEO does in the process. Said results prove that far from simply mimicking existing governors, the methodology described throughout this paper is sound and can result in a trained model that shows entirely novel, previously unseen characteristics when used as a CPUIdle governor.

5.5 Explainability

When it comes to explainability, the best way to interpret the model's decision making process is to look back on the feature permutation importances previously mentioned in the *Feature selection* section. These features will be described below in the order of their importance.

1. state1 hit metric - Indicates to the governor that in the recent past it has had success selecting state 1 and so it might be a good idea to continue doing so.

2. time until the next timer event - This feature provides a hard ceiling for the governor and, in certain circumstances, allows it to make a very definitive decision on which state to select on its own.

3. state0 interception metric - In a way the opposite of feature 1. It serves to indicate to the governor when the assumption drawn from a high state1 hit metric becomes false, such as when the idle sleep pattern suddenly changes due to change in system activity.

4. CPU utilisation - Used less than the three previously discussed features, provides an even earlier indicator of a rise in system activity than state0 interceptions.

5. ttwu_count delta - Helps refine the decision based on the amount of threads woken up on this CPU since the last sleep.

6. rq_cpu_time delta - Helps refine the decision based on the runtime on this CPU since the last sleep.

7. max_capacity - Helps refine the decision by letting the governor distinguish between CPU clusters.

Even when it comes to the non-ML mainline idle governors, discussions of explainability tend to be slighlty more on the theoretical rather than on the tangible side. The idle landscape of the Linux Kernel in general is very fast with sometimes hundreds of sleeps per minute, various metrics and heuristics that are more aligned with the theory of how things should look rather than with the practice of what things do look like in reality. The metrics collected and used by TEO and mlidle for instance are kept and decayed in very specific ways based on hardcoded constants that were determined empirically by the developers and have no deeper meaning outside of the fact that these hardcoded values made the governor work fine in the testing stage.

Outside of those general observations, the picture becomes even less clear once differences between various workloads are considered. The decision tree of the governor in and of itself is static, but because the input features comprise statistics computed based on the decisions taken by the governor, a certain kind of a feedback loop is formed in the process. Where this feedback loop will lead is heavily dependent on the workload being run on the system at any given moment. On that account, it becomes neigh impossible to completely explain or predict why a certain decision was taken at a certain point in time. Simply put, too much randomness is involved in the entire process for it to be explainable apart from at a higher level.

5.6 Observations

With the approach described and established in this way, it should certainly be possible to train and produce other models with even better results, whether it be by tweaking the feature selection, tweaking the type of model, the hyperparameters or even all of the above. The most notable observation was that the behaviour of the trained *mlidle* models was not mimicking the heuristics present in either of the two mainline governors - it was a completely novel governor with its own set of benefits and drawbacks, even in spite of the kernel implementation being based around existing *TEO* metrics.

To some extent, this research also managed to answer a fundamental theoretical question in regards to the inherent randomness present within CPUIdle. That is to say, since none of the trained models were able to even get close to the ideal 100% accuracy, the conclusion is that not being able to achieve said accuracy is not exclusively the fault of mainline kernel heuristics being suboptimal. On the contrary, the heuristics are very close to being as good as they could be given the set of input data that they consider when making their decisions. The wider conclusion is that either some system data currently unaccounted for needs to be taken into account by potential future governors, or more likely it is just that due to operating system randomness there will always be a hard ceiling for how accurate a governor can get. Unfortunately that ceiling will also always be below 100%.

A final interesting observation could be made on the question of overheads. The primary concern that was raised when this research was initially being considered was about the overheads of Machine Learning potentially being too high for a kernel setting, even more so in a subsystem as performance-sensitive as CPUIdle. As it turned out, at least with the model of choice being a Random Forest, no issues specifically relating to overheads were observed at any stage of this research. In spite of the exported C models amounting to respectively either 6100 lines of code or even 21186 lines of code, close to all of those lines are just sequences of if-else statements which could be optimised by the compiler and the CPU branch predictor.

6. CONCLUSIONS

Throughout this paper, it was shown that the proposed methodology for training Machine Learning models to predict CPU idle states is sound, works as expected and produces governors with actually novel characteristics. It was shown that those models when trained to maximise accuracy were able to improve on the accuracy of the Linux mainline governors. Averaged out across the 4 workloads, the *mlidle* governor was able to achieve an accuracy score of 0.87 compared to 0.84 for *TEO* and 0.79 for *menu*. While the exact measurements and test results are important to consider and keep in mind, they are not the most important conclusion from this research. The primary goal of this research was to establish whether the approach in and of itself was feasible and could lead to improvements in certain areas. From the results it is clear to see that the answer to this fundamental question is yes.

All in all, the research described in this paper could be considered a success and a good starting point for potential future research in related areas.

7. FUTURE WORK

The results of this research clearly open the door for a large body of potential future work. At its simplest, now that a viable methodology has been established, future research could be conducted to prepare and train potentially more accurate idle state prediction models. This could be achieved either by tweaking the currently available model types or by finding alternative system metrics for potential governors to consider in the first place. It could also be interesting to explore applying Reinforcement Learning [44] to the task of improving idle state selection accuracy.

7.1 Alternative CPU Architectures

Research could (and most likely should) be conducted with the goal of adapting this approach for non-Arm, e.g. x86-based devices. For the most part the methodology will work exactly the same, the only small challenge will be brought about by changing the feature selection in order to appropriately account for there being considerably more than 2 idle states available on these types of devices. This research focused on the Arm platform due to the availability of the test device and the ease of conducting experiments but nothing within the methodology is Arm-specific. Similar models, with small adjustments, could certainly be trained for any other platform and architecture.

7.2 ML in other Kernel Subsystems

There is nothing standing in the way of applying this same approach to any other subsystem of the Linux kernel. An especially interesting area of future work could be conducting research into training and applying Machine Learning for instance in the area of scheduling algorithms, whether it be process scheduling or I/O scheduling. Some research already exists into different kinds of applications of Machine Learning within the Linux Kernel, some even specifically using *emlearn* such as [6]. Other research exists as well in the area of applying Reinforcement Learning [44] to the Linux Kernel, such as [15].

7.3 Improving emlearn

There is certainly potential for future work to be done on extending the *emlearn* [32] project to support other types of models and to streamline its application to this type of work. Firstly, as things stand all of the exported C models had to be manually adjusted before it was possible to include them in the kernel. This was necessary because the "kernel way" of writing C is somewhat different from the userspace one, as was described earlier in the Background section. On that account, *emlearn* could easily be extended to optionally support such "kernel mode" model exports.

Improvements to *emlearn* could be made in the area of the availability and support for more *scikit-learn* models too. As things stand, very few models are fully supported. Models such as the aforementioned MLP are supported only partially, notably without fixed-point arithmetic, which makes them presently unsuitable for these types of applications. Such limited support is completely understandable given that it is an open-source project with a single maintainer and thus focusing future work on improving the project would certainly be of great benefit to the wider community. It is clear even at this stage that the project holds great value and great potential for the future.

8. REFERENCES

[1] Alessio Balsini. Scheduling for the Android display pipeline. https://lwn.net/Articles/809545/. Retrieved March 2024.

[2] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer. Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347, 04 2010.

[3] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The ibm system/360 model 91: Floating-point execution unit. *IBM Journal of Research and Development*, 11(1):34–53, 1967.

[4] Arm Ltd. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487. Retrieved March 2024.

[5] Arm Ltd. big.LITTLE Processing with ARM Cortex-A15 Cortex-A7. https://web.archive.org/web/20131017064722/http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf. Retrieved March 2024.

[6] G. Bertoli, L. Alves Pereira Junior, O. Saotome, A. Santos, F. Verri, C. Marcondes, S. Barbieri, M. Rodrigues, and J. Oliveira. An end-to-end framework for machine learning-based network intrusion detection system. *IEEE Access*, PP:1–1, 07 2021.

[7] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[8] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984.

[9] K. Burnham and D. Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer New York, 2007.

[10] C. Cortes and V. N. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 2004.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[12] Google. Android NDK. https://developer.android.com/ndk. Retrieved 2024-03-18.

[13] Google. JetNews. https://github.com/android/compose-samples/tree/master/JetNews. Retrieved March 2024.

[14] gs.statcounter.com. Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share. Retrieved March 2024.

[15] J. Han and S. Lee. Performance improvement of linux cpu scheduler using policy gradient reinforcement learning for android smartphones. *IEEE Access*, 8:11031–11045, 2020.

[16] D. Hand and K. Yu. Idiot's bayes: Not so stupid after all? *International Statistical Review*, 69:385 – 398, 05 2007.

[17] Intel®. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2b-manual.html. Retrieved March 2024.

[18] Joe Rozek. Using Adaptive Performance for the "Amazing Adventures of Dr. Arm". https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/how-to-use-adaptive-performance. Retrieved March 2024.

[19] Linaro Limited. DeviceTree. https://www.devicetree.org/. Retrieved March 2024.

[20] Linus Torvalds. Kernel floating-point. https://yarchive.net/comp/linux/kernel_fp.html. Retrieved March 2024.

[21] Linux Kernel Community. Arm - Idle States. https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/idle-states.txt. Retrieved March 2024.

[22] Linux Kernel Community. CPU Idle Time Management. https://docs.kernel.org/admin-guide/pm/cpuidle.html. Retrieved March 2024.

[23] Linux Kernel Community. CPU Performance Scaling. https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html. Retrieved March 2024.

[24] Linux Kernel Community. ftrace - Function Tracer. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html. Retrieved March 2024.

[25] Linux Kernel Community. Programming Language. https://docs.kernel.org/process/programming-language.html. Retrieved March 2024.

[26] Linux Kernel Community. Sched Stats. https://www.kernel.org/doc/Documentation/scheduler/sched-stats.txt. Retrieved March 2024.

[27] Linux Kernel Community. sysfs - The filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html. Retrieved March 2024.

[28] Linux Kernel Community. TEO CPUIdle Governor. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/cpuidle/governors/teo.c. Retrieved March 2024.

[29] Linux Kernel Community. The menu Governor. https://docs.kernel.org/admin-guide/pm/cpuidle.htmlthe-menu-governor. Retrieved March 2024.

[30] Linux Kernel Community. Timers. https://www.kernel.org/doc/html/latest/timers/index.html. Retrieved March 2024.

[31] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[32] J. Nordby, M. Cooke, and A. Horvath. emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices, Mar. 2019.

[33] ONEIROS. Keras. https://keras.io/. Retrieved March 2024.

[34] Peter Kostadinov. Google's Tensor G1, G2, and G3 chipsets explained. https://www.phonearena.com/news/Googles-Tensor-explained-Core-Pixel-features-amplified_id134064. Retrieved March 2024.

[35] Primate Labs Inc. Geekbench 6.

https://www.geekbench.com/. Retrieved March 2024.

[36] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back propagating errors. *Nature*, 323:533–536, 10 1986.

[37] scikit-learn developers. scikit-learn. https://scikit-learn.org/stable/user_guide.html. Retrieved March 2024.

[38] Speedometer. Speedometer 2.0. https://browserbench.org/Speedometer2.0/. Retrieved March 2024.

[39] Steven Rostedt. trace-cmd: A front-end for ftrace. https://lwn.net/Articles/410200/. Retrieved March 2024.

[40] Steven Vaughan-Nichols. Rust in Linux: Where we are and where we're going next. https://www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/. Retrieved March 2024.

[41] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[42] J. Tolles and W. J. Meurer. Logistic Regression: Relating Patient Characteristics to Outcomes. *JAMA*, 316(5):533–534, 08 2016.

[43] Unity. The Unity Game Engine. https://unity.com/. Retrieved March 2024.

[44] M. van Otterlo and M. Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.