

**PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks**

Soroush Bateni* Husheng Zhou * Yuankun Zhu Cong Liu
The University of Texas at Dallas

**Abstract.** The revolution of deep neural networks (DNNs) is enabling dramatically better autonomy in autonomous driving. However, it is not straightforward to simultaneously achieve both timing predictability (i.e., meeting job latency requirements) and energy efficiency that are essential for any DNN-based autonomous driving system, as they represent two (often) conflicting goals. In this paper, we propose PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system. PredJoule achieves both latency guarantees and energy efficiency through a layer-aware design that explores specific performance and energy characteristics of different layers within the same neural network. We implement and evaluate PredJoule on the automotive-specific NVIDIA Jetson TX2 platform for five state-of-the-art DNN models with both high and low variance latency requirements. Experiments show that PredJoule rarely violates job deadlines, and can improve energy by 65% on average compared to five existing approaches and 68% compared to an energy-oriented approach.

**I. INTRODUCTION**

Autonomous driving is expected to transform the auto industry. The recent advances in Deep Neural Networks (DNNs) enable autonomous driving systems to adapt their driving behaviors according to dynamic environments. For instance, DAVE-2 [17], released by NVIDIA in 2016, can predict steering angles from the driving scenes captured by front-centered cameras of the autonomous vehicle.

Adopting inherently compute-intensive DNNs in often resource- and energy-constrained automobiles creates hard computational challenges. Adding sufficient DNN layers to guarantee high accuracy may easily explode the computation demand [8]. Currently, embedded system designers mainly rely on applying GPU-accelerated hardware platforms to satisfy the need, since GPUs can enable orders of magnitude faster and more energy-efficient execution of many general-purpose workloads (i.e., GPGPU). For instance, NVIDIA and Audi, one in NVIDIA’s long list of automotive partners, recently announced their plans to deliver DNN-based highly automated vehicles by 2020 [4].

Unfortunately, it is not straightforward to process DNN-based workloads in an autonomous driving system equipped with GPU-accelerated platforms, due to the need of satisfying two (often) conflicting goals: timing predictability and energy efficiency. Timing predictability (i.e., meeting job latency requirement) is one of the most important tenets in certification required for autonomous driving systems. The functional correctness of an automobile hinges crucially upon temporal correctness (e.g., performing object detection within a strict latency boundary to signal automatic brake requests). On the other hand, automobiles demand low energy consumption, due to their strict size, weight, and power (SWaP) requirements. Regrettably, timing predictability and energy efficiency are often in conflict. This is because the former requires reserving sufficient resources for guaranteeing latency even in the worst case; while the latter often desires allocating just enough resource that barely meets the needs of the current job.

While several recent works have been done on optimizing latency and energy simultaneously in multicore systems [6], [13], [20], [21], [23], [25], [44], and not much research targets DNN workloads and explores their unique characteristics in performance optimization. A unique characteristic of DNNs is that they consist of multiple layers each of which exhibits different computation complexity and characteristic. Our extensive measurements also show that the energy usage pattern differs dramatically among different layers and under different system configurations. Existing layer-oblivious energy/latency optimization techniques [49], [14] that target the general workload may not work for DNN-based autonomous driving since they do not explore and exploit per-layer characteristics. Moreover, most existing solutions consider energy optimization in environments with soft timing constraints [23], [72], [18]. They optimize latency only on a best-effort basis but clearly cannot achieve the required timing predictability. The resulting state of affairs is rather unsettling: the revolution of DNN is enabling dramatically better autonomy and services in autonomous driving, but the required timing predictability and energy efficiency cannot be reached simultaneously in any DNN-based autonomous driving system.

**Contributions.** In this paper, we present PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system. PredJoule is designed to address a critical point not addressed in the design space: providing timing predictability while minimizing energy in neural networks.

Neural networks have specific power and performance characteristics that make existing energy optimization methods

*These two authors contributed equally.

Work supported by NSF grants CNS 1527727 and CNS CAREER 1750263.
incompatible due to their layer-oblivious design. PredJoule can specifically observe and adapt to the layer-based nature of neural networks and optimize the energy efficiency of each layer according to the inherent Uncertainty of that layer. Uncertainty characterizes each layer in terms of that layer’s energy sensitivity in response to performance change. To bridge the gap between timing predictability and energy efficiency, PredJoule utilizes a progress tracker component that tracks execution deficit at each layer’s execution boundary and identifies the best DVFS (dynamic voltage and frequency scaling) configuration for processing the next layer by considering both Uncertainty of that layer and the corresponding execution deficit.

PredJoule can handle a very large DVFS configuration space that is necessary for a GPU-enabled heterogeneous “big.LITTLE” multicore platform commonly deployed in automotive systems. Our configuration space consists of tens of thousands of individual frequency and core enable/disable modes. Based on a learning-based methodology, PredJoule first observes and records the nuances of each layer when new configurations are tried. Then, after a specific learning period, PredJoule will be able to meet timing requirements consistently by consulting the history. Moreover, PredJoule does not need to try the entire configuration space. Rather, for layers with lower Uncertainty, it is possible to try as few as 10 new configurations before finding an optimal setting. However, our method will not skip configurations for layers that have extremely high Uncertainty. Moreover, if the system is ahead of schedule, PredJoule will backtrack and learn new configurations for better energy efficiency.

We implement PredJoule on an automotive-specific embedded platform: the NVIDIA Jetson TX2 with a GPU-enabled “big.LITTLE” multicore architecture. We evaluate its timing predictability and energy efficiency against 5 state-of-the-art approaches for 5 modern DNN models: CoffeeNet [27], AlexNet [31], GoogleNet [43], VGGNet [41], and ResNet [19]. Our results demonstrate that PredJoule achieves:

- **Efficacy with a range of behaviors**: The tested workloads include five DNN models with both high and low variance latency requirements (Sec. V-B).
- **Timing predictability**: PredJoule never misses a deadline in all test cases after convergence (see Secs. V-B and V-C).
- **Energy efficiency**: PredJoule outperforms all other tested methods by a significant margin in most cases, reducing energy by 65% of five existing approaches on average and 68% of an energy-oriented approach (see Secs. V-B and V-C).
- **Adaptability to interference**: PredJoule automatically reacts to fluctuations introduced by interfering workloads (see Sec. V-D).
- **Runtime overhead**: the design and implementation of PredJoule is efficient, incurring a rather small runtime overhead (see Sec. V-E).

II. SYSTEM MODEL

In this section, we describe the basic principles behind DNNs and explore the abstract architecture of a typical GPU-enabled automotive hardware.

**DNN background.** A neural network is constructed from a neuron. A neuron takes a set of inputs \( I \) and will calculate a state \( s \) according to the “learned” weight of each input \( w_i \).

A neural network is a set of neurons that are directly connected to the inputs of the neural network, the output neurons whose outputs represent the final output of the neural network, and finally, the internal neurons that use the output of other neurons as their input and send their outputs to another neuron. The input neurons and the output neurons form the input and output layers of a neural network, respectively; while the internal neurons form the hidden layers of a neural network.

For reasons related to complexity, internal neurons may not be interconnected arbitrarily. Rather, based on their broad functionalities, they will be split into layers. Neural networks with many hidden layers are usually called deep neural networks. Note that current DNN-based autonomous driving systems almost always rely on GPUs to learn weights \( w_i \) and produce outputs.

**Heterogeneous platforms and DVFS.** Before presenting the system model and our design, we first elaborate on the power model of the target system. Contemporary literature has mainly employed a simplistic power model. Most only consider CPU frequency [6], [13], [20], some have included single-cluster core count [25], and even fewer have considered GPU frequency [59].

However, the recent trend in computing systems has resulted in increasingly more power efficient systems. This trend is especially noticeable in embedded systems such as those used in autonomous driving. This implies that many modern systems have specifically voltage-gated subcomponents that go far and beyond what traditional literature has considered. For example, for a contemporary System on a Chip (SoC) used as an energy and space saving alternative to separate component layouts (as seen in state-of-the-art automotive platforms such as NVIDIA TX2 and PX2 [34]), there is at least 5 different DVFS adjustments:

- Small Core Activated Mask \( (M_s) \)
- Small Core Frequency \( (F_s) \)
- Big Core Activated Mask \( (M_b) \)
- Big Core Frequency \( (F_b) \)
- GPU Frequency \( (F_{GPU}) \)
- Memory frequency \( (G_m) \)

Our encompassing configuration space is defined as a tuple \( (M_s,F_s,M_b,F_b,F_{GPU},G_m) \). For a system that lacks dynamic memory or GPU adjustment or lacks a big.LITTLE architecture, the corresponding variable would be set to 0.

Moreover, state of the art literature always considers a definite sorting order for the configuration space. For example, for two CPU-only configurations \( c_i \) and \( c_j \), it is possible to assume that \( c_j \) is faster than \( c_i \) if the frequency is higher.
However, our extensive testings show that such an order does not necessarily apply in general. A lower frequency setting may counter-intuitively result in smaller latency or less energy saving. We thus define a partial order as a starting point but also introduce a history-based learning approach in Sec. IV-C to account for errors.

The partial order follows a similar rule as other prominent literature [49]. A DVFS configuration $c_i$ is considered faster than $c_j$ if each subcomponent of $c_i$ is larger than that of $c_j$. However, this ordering is not sufficient to completely sort the configuration space. For example, if all but one attribute of $c_i$ are larger than $c_j$, there is still no way to sort $c_i$ after $c_j$. For that, we introduce partial ordering instead that uses weights to compare configurations. For example, the priorities for TX2 and GPU layers are $G_m, F_s, F_{GPU}, M_b, F_b$, and $M_s$ in that order and weights are assigned accordingly.

DNN Workload Model. A traditional system model would model a DNN as a single task and deal with latency and energy requirements of it with a resolution that is not optimal according to our observations (shown in Sec. III). Instead we define a layer-based DNN model that includes sub-layers of any DNN as individual tasks. Thus, for a DNN $x = \{l_1, l_2, ..., l_n\}$, the system would at least contain $n$ tasks $T = \{t_1, t_2, ..., t_n\}$ corresponding to $n$ layers. A DNN instance $x$ is expected to have a deadline at $d_x$. For instance, for a 30 FPS (frame per second) camera, running DNN-based object recognition would impose a 33ms deadline on processing each frame.

III. MOTIVATION

In this section, we present two fundamental observations that have motivated our design and implementation of an efficient system solution specifically tailored for DNNs.

A. DNN-specific Energy Usage Patterns

We have performed an extensive set of experiments seeking to investigate the energy usage patterns in a neural network. We setup the default configuration of Alexnet on a Jetson TX2 as a constant factor and measure the energy consumption and the compute latency under different configurations.

The effect of GPU and CPU frequency on energy consumption for general workloads have been extensively explored before [1, 29, 25, 39, 49]. However, on an NVIDIA Jetson TX2, we found no clear linear pattern between lowering/boosting the frequency and decreasing/increasing the energy usage when running a DNN instance as a single entity. Rather interestingly, memory energy usage decreases exponentially with higher frequencies. This behavior is more related to the inherent structure of memory. Thus, making memory operations shorter by increasing the memory frequency would result in a better energy efficiency.

In the case of GPU, the inherent parallelism causes the execution time of Alexnet to exponentially decrease when the frequency is increased, outpacing the increased power usage that results from the higher frequency. However, there is also not a clear trend when GPU frequencies are set to be higher than 1000MHz. For example, it is possible for a configuration with GPU frequency of 1000MHz to be more energy efficient compared to one with a 1300MHz frequency.

We next investigate the per-layer energy usage patterns in neural networks. For demonstration purposes, we only show 2 different layers among all 24 default layers in Alexnet in Fig. 1. For this experiment, we combine all the frequency changes but only present a subset of 192 configurations in Fig. 1 for clarity. We record the average energy usage on an NVIDIA Jetson TX2. As is depicted in the figure, each layer demonstrates a different response to the configuration change. Interestingly, the same type of layer, Relu, behaves vastly different (almost opposite) when positioned at different depths. This stark difference is due to the fact that by the time Alexnet reaches Relu6, the data has been shrunken to a point that Relu6 is dominated by computation rather than memory. The same behavior is observed among all layers, with almost no consensus throughout the neural network on an optimal configuration for energy usage. This trend is observed even for the entire 14111 configurations tested.

**Observation 1:** If we consider DNNs as a blackbox, choosing the best configuration would not be as simple as solving a linear equation (as for other more straightforward workloads [29]) since our measurements show that Alexnet behaves optimally only by changing the DVFS configuration for each layer. In fact, a layer’s optimum configuration can be the absolute worst for other layers, making the process of finding an optimal configuration rather challenging and interesting for neural networks. Moreover, we find that layer type alone cannot be a deciding factor in finding the optimal configuration.

B. Energy-Performance Relationship

While choosing an optimum energy configuration for neural networks is an interesting challenge on its own, addressing this problem without considering latency can cause safety-critical timing issues. An optimal energy configuration for a layer can result in unreasonably long processing times.

Thus, we have also considered layer-based performance in terms of latency. We would like to investigate how a potential solution might be able to balance between energy usage and performance such that stringent latency requirements are met while energy gets minimized. Specifically, we are interested
in understanding how each layer might respond to system configuration change in terms of latency and energy efficiency. Fig. 2 shows the same setup used for Fig. 1. In this case, we include the latency metric for the first normalization layer Norm1 and the Relu6 layer in Fig. 2. Values of energy and latency are normalized by dividing each value by the maximum of their respective measurement.

Naturally, when the system jumps from one configuration to another, there will be changes to both latency and power. The critical point is to understand how would the energy consumption respond to a decrease or increase in latency. As seen in Fig. 2 for Norm1, a decrease in latency due to configuration changes will almost always result in a decrease in energy consumption and vice versa. On the other hand, for a layer like Relu6, this trend is not clear. In other words, we could be certain to a point that increasing the speed for Norm1 will result in better energy efficiency. However, we cannot be certain that such a benefit exists for Relu6. For Relu6, it is possible that a slight increase in speed can cause significant energy loss.

We call the trend present in Fig. 2 the trailing effect because energy consumption trails the latency. In Section IV, we present an idea that can separate the layers based on this trailing effect. Moreover, we will show the degree of which the trailing effect exists for various neural networks and layers in Sec. V.

**Observation 2:** We have investigated the impact of DVFS configurations on latency performance with relation to energy, observing that a trailing pattern often exists between energy and performance. This trailing relationship motivates us to explore potential solutions that may exploit per-layer characteristics for more efficiently saving energy while maintaining timing requirements. For instance, compared to Relu6, layer Norm1 could be a much better candidate if we need a certain speedup while executing the network in order to catch the corresponding deadline.

![Diagram](image.png)

**Fig. 2:** The trailing effect for two layers. Layer 4 has a pronounced trailing effect while layer 18 does not.

**Fig. 3:** Design overview of PredJoule.

**IV. DESIGN**

In this section, we outline the overall design of PredJoule, as illustrated in Fig. 3. To exploit per-layer characteristics in terms of latency and energy usage patterns, we introduce a concept of Uncertainty for each layer. Uncertainty indicates how uncertain the system should be about a layer’s energy to performance relationship (as discussed in Sec. III). Intuitively, the lower the value of Uncertainty, the better the chance of achieving the optimal power/performance ratio for that layer.

Consequently, the system should aggressively change the DVFS configuration to a faster one for layers that have low Uncertainty. Moreover, the controller should explore more configurations for layers with a high Uncertainty because the behavior of these layers is inherently unpredictable.

To maintain timing correctness, PredJoule introduces a progress tracker that provides an effective system status and interference control in the form of an execution deficit $\varepsilon$. The progress tracker then conveys $\varepsilon$ to a controller that is responsible for making configuration changes at runtime. The controller is able to react to $\varepsilon$ via configuration adjustments. However, PredJoule imposes a strict power-consumption tariff on the decision-making process of the controller. This is achieved by inputting the next layer’s Uncertainty value into the controller. By smartly integrating the consideration of both execution deficit $\varepsilon$ and the next layer’s Uncertainty, PredJoule is able to find the optimal configuration for each layer that yields controlled timing correctness with minimized energy consumption. Finally, we address issues of error-prone partial sorting and timing guarantee with a history module. If the history is reliable enough, our method will use history instead to adjust total execution time accurately. Note that the configuration adjustment decision is made only at each layer’s boundary (i.e., at the completion time of each layer’s execution).

**A. Uncertainty**

To capture the layer-based uniqueness in power and performance usage, we introduce a new concept: Uncertainty. As discussed in Sec. III, a unique characteristic of some DNN layers was the trailing effect. Layers that exhibit this effect can be good candidates for more aggressive configuration adjustments since the performance and energy behaviors can
be easily predicted. Intuitively, Uncertainty reflects the degree of certainty the system has on predicting how a layer’s energy usage would respond to the performance change due to configuration adjustments. A lower Uncertainty value implies a higher certainty.

Before formally defining Uncertainty, we first define normalized energy and latency performance. We normalize the latency and energy usage values by dividing each value by the maximum:

**Definition 1: Normalized energy and latency.**

\[
\text{Norm}(P_{c\in C}) = \frac{P_c}{\max_{c \in C} P_c}, \quad (1) \\
\text{Norm}(L_{c\in C}) = \frac{L_c}{\max_{c \in C} L_c}, \quad (2)
\]

in which \(\text{Norm}(P_{c\in C})\) and \(\text{Norm}(L_{c\in C})\) are the normalized energy and performance values under a configuration \(c, \) and \(C\) represents the entire configuration space.

**Definition 2: Configuration Uncertainty (CU).** For running a layer under a configuration \(c \in C,\) we define CU of the layer as:

\[
CU_{c\in C} = \frac{\text{Norm}(P_c)}{\text{Norm}(L_c)}. \quad (3)
\]

Since \(CU\) is on a per-layer and per-configuration basis, storing and processing it for hundreds of layers under thousands of configurations is expensive (if not impossible). Thus, we collect and summarize \(CU\) for each layer, and reduce the problem complexity by \(|C|\) (size of configuration space). We utilize the max function to define the Uncertainty:

**Definition 3: Uncertainty.** Given the set of configurations \(C\) for each layer, we define the Uncertainty as:

\[
U = \max_{c \in C} CU_c. \quad (4)
\]

According to the above definition, the Uncertainty of a layer indicates the maximum gap between the normalized energy consumption and the normalized latency. If the energy consumption is high (close to 1) and latency is low (close to 0), Uncertainty would become extremely large. This large value can indicate to the system that a low latency (implying a fast configuration) can yield a high energy consumption. Thus, the system needs to be cautious since it is uncertain about whether or not a boost in performance could cause a dramatic sacrifice in energy for that layer. On the other hand, for the case where Uncertainty is low, the system knows that an increase in latency performance would lead to a smaller energy consumption for the corresponding layer and configuration. Thus the system can be more certain on optimizing the latency and energy using this layer.

As an example, imagine a system with 7 computing layers. The Uncertainty of each layer along with their type is depicted in Table I. We note that real-world neural networks are usually much deeper, yet Table I contains almost all the crucial layers in a compact configuration. As is seen in the table, the first Convolutional layer abbreviated as \(Conv1\) has an Uncertainty that is low. This is due to the type of the layer. Moreover, this layer is at a shallow level of the DNN, which makes the trailing effect for this type of layer even more pronounced.

The second convolutional layer has an Uncertainty that is larger because it is deeper. The Uncertainty indicates that in this case, the power would trail the performance closely by up to 1.7 times. Other layers, particularly the Softmax layer, have high Uncertainty values, indicating that the relationship between power and performance is rather erratic for these layers.

Finally, to ease the burden of future development based on Uncertainty, we offer Table II. Table II contains an approximate value of Uncertainty for the most common DNN layers. The Uncertainty is based on the two most important factors: the type of the layer and the position of the layer. As discussed earlier, the type of the layer is not the sole contributor to a layer’s power/performance behavior. The position of that layer in the neural network is equally important. As the DNN progresses, the amount of information processed by each layer is reduced substantially. On the other hand, the computation required to produce the output for the next layer becomes much more significant. We note that these values of Uncertainty only apply to GPU-based DNNs. Since these types of DNNs are the most widespread version, Table II should cover most use cases in the real world.

**B. Progress Tracker**

We have explained our intuition behind using Uncertainty to capture the power/performance characteristics of each layer. We now introduce the other component, a progress tracker, that will be integrated into PredJoule to tightly control timing through tracking the execution deficit at each layer boundary.

We define a straightforward progress tracking strategy based on a schedule that is constructed from a randomly selected DVFS configuration. Consider the example shown in Fig. II. Assume there is a DNN instance \(T\) consisting of \(n\) layers \(\{T_1, ..., T_n\}.\) Let \(c_i\) denote the execution time of each layer \(i\) in this schedule. Let \(\varepsilon\) denote the end-to-end execution deficit in this schedule, i.e., \(\varepsilon = F - D,\) where \(F\) and \(D\) are the completion time and deadline of \(T,\) respectively. A positive

\[\varepsilon > 0\]

The depth is important because it would reflect on the computational characteristic of the layer. This reflection is mainly due to the input size for that layer. The deeper a layer is, the smaller the input size for that layer is, in most cases.
(negative) value of $\varepsilon$ implies that $T$ completes after (before) the deadline; the case where $\varepsilon = 0$ implies that $T$ completes exactly at its deadline under this random configuration. At each layer boundary, the system updates the execution deficit and will have to catch up in the case of a positive value. On the other hand, the system might exploit a negative execution deficit for an energy efficiency gain.

Our design tracks the progress of each layer’s execution under its runtime configuration $C$ using this random schedule as a reference, which considers the above-defined $\varepsilon$. Our design then compares each layer $l_i$’s execution time under $C$, denoted as $c_i$, with the execution time $c_i'$ under the random configuration. That is, from a per-layer perspective, after executing each layer $l_i$ under a certain configuration $C$, $T$ would have an updated deficit of $\varepsilon = c_i - c_i' + \varepsilon$. A positive (negative) per-layer deficit at each layer boundary implies that after executing layer $l_i$, the entire schedule is running behind (ahead) compared to the “ideal schedule”. From a latency perspective, a perfect scenario is to complete each layer such that the deficit at the corresponding layer boundary is equal to 0, which implies that the entire DNN instance would complete exactly at its deadline.

C. Integration

In this section, we describe how we integrate our idea of Uncertainty with the progress tracker to achieve timing predictability and energy minimization. Specifically, we consider the execution deficit at each layer boundary and use Uncertainty of the next layer to decide its configuration. There are two cases to consider after executing each layer $l_i$.

Case 1: $\varepsilon > 0$. If the deficit after executing $l_i$ is positive, the system will need to jump into a higher-speed configuration to catch up with the deadline, say from $C_i$ to $C_j$. This configuration adjustment depends crucially on Uncertainty of the next layer $l_{i+1}$ for energy saving reasons.

As discussed in Sec. IV-A, a lower Uncertainty indicates that the layer exhibits a more pronounced trailing effect in which a higher performance results in lower power usage, while a high Uncertainty indicates irregularities that could potentially be inefficient.

Since the configuration space is partially sorted, the jump from $C_i$ to $C_j$ can be achieved by adding a value to $i$. This would imply that the higher index value of a configuration is, the faster it will be (we first assume this and correct for any errors once the history becomes reliable). Moreover, a jump to a higher value for configuration index would be directly related to the inverse value of Uncertainty since a small Uncertainty should result in a big jump:

$$j = i + \alpha \frac{1}{U_{i \in L}},$$

in which both $c_i$ and $c_j$ are part of the configuration space $C$. The variable $\alpha$ acts as a user-defined gauge that can control the aggressiveness of Equ. [5].

Case 2: $\varepsilon \leq 0$. In the case that the deficit is less than or equal to zero, PredJoule will try to exploit the resulting negative $\varepsilon$ for gaining a more optimal energy configuration. In other words, if the Uncertainty of $l_{i+1}$ is high, it is possible to gain energy optimization by running slower. For example, an infinite Uncertainty implies that the slowest configuration should be used to be conservative. We reuse Equ. [5] with a reversed coefficient:

$$j = i - \beta U_{i \in L}.$$  

$\beta$ is the corresponding coefficient for a negative execution deficit.

A Learning-based Approach. An assumption made in Eqs. [5] and [6] is that the configuration space $C$ is perfectly sorted according to the speedup gained. However, in real-world, there could be oddities present in a partially sorted set. Moreover, Uncertainty tries to encapsulate how each layer would react to a configuration change into just one number, which will obviously result in a loss of detail that can potentially hamper the process of improving the energy usage. Finally, it is impossible to guarantee any latency requirement because the controller cannot predict the exact degree of speedup a new configuration would bring. If there were an exact speedup associated with the new configuration, the controller would have been able to intricately choose a configuration that would exactly match the current system need.

Contemporary literature concerning this matter always resorts to some sort of offline benchmarking in order to mitigate this problem [25], [49], [34]. While offline benchmarking can be effective in isolation, it cannot work for a runtime
environment. The biggest reason is that an offline solution will not be able to adapt to any interference due to either workload or system environment changes. This is a major drawback even without considering the overhead of offline benchmarking for hundreds of layers run under thousands of configurations.

Rather, we would like to exploit a special characteristic of neural networks: All neural networks are constructed from a limited set of layers and keep a consistent configuration throughout multiple executions. Since the parameters, operations, and data width and types are consistent between each run, variations in input cannot affect the execution behavior of a DNN in a measurable way. This motivates us to design a learning-based approach that can learn and adapt more granularity over time. Such an approach is especially desirable in applications such as autonomous driving in which a learning procedure is always part of the development. For example, Tesla uses a “fleet learning” procedure \cite{44} to make vehicles learn driving from shared driving data.

For the learning phase, we introduce a history recorder to PredJoule. Each time the controller selects a new configuration according to Equ. 5 and Equ. 6, it will record the resultant speed and energy change for that specific layer. To make the history data structure and the lookup procedure simple, all the changes are recorded by dividing the newly measured power and speed of the system by a single base configuration: the random configuration first defined in Sec. IV-B. Note that history is layer-based.

However, the accuracy of the history entirely depends on the number of new configurations that have been explored by using Equ. 6 and 5. If the system has not been running for long or has settled on an optimal configuration after visiting only a few configurations, the history can be quite unreliable. The only solution in this scenario would be to continue to explore new configurations.

We represent this quality by a variable called system maturity denoted by $M$. For a DVFS configuration space $C$ and a history lookup table $H$, the maturity of the system is defined as:

$$ M = \frac{|H|}{|C|} $$

in which $M$ is always between 0 and 1. The $|H|$ and $|C|$ depict the number of entries in $H$ and $C$, respectively. The problem now becomes that of setting a threshold. The closer the $M$ is to 1, the more likely it is for PredJoule to choose a configuration already in the history. We set an extremely high value of the threshold (0.9999) because in our experiments, the system reaches enough maturity so that it can solely rely on history. While the deadline can be caught before then (as it almost always does in our experiments), the procedure continues until the system reaches enough maturity that can be used to rely on history.

With a positive deficit, the system will need to catch up. At the beginning of next execution, the controller (at line 5) will recognize the positive $\varepsilon$. Moreover, the maturity is just 1 (the random configuration) divided by the configuration space size (1411 in our case). The check thus fails at line 6. For this layer’s boundary, the system will use Equ. 5 in line 9 instead, with an $\alpha$ of 1. The system will jump to $c_{400+1,1,400}$, which is $c_{900}$. At the boundary of layer 1, the controller will record the speedup of $c_{900}$ (based on $c_{400}$). If $\varepsilon$ is still positive, the same procedure is applied. Let us assume that the system has caught up with the execution deficit. Then, at the boundary of layer 2, with a maturity of 1/1411 (for layer 2), the system will use Equ. 6 (at line 14) instead. This equation backtracks any jump in order to examine any configurations that have been missed. For layer 2 and a $\beta$ of 0.001, the new configuration will be $c_{400-0.001,10,400} = c_{360}$ ($\beta$ is always much smaller than $\alpha$ because catching deadline is much more urgent than exploration). This procedure continues until the system reaches enough maturity so that it can still rely on history.
approximation. With no approximation, MEANTIME may not be effective.

- **Poet-GPU**: Poet does not consider platforms containing GPUs. We thus have implemented a version of poet that can control GPU’s DVFS configuration in addition to CPU.

- **Race2Idle**: Race2Idle \([29]\) is the infamous simplistic approach to meet hard deadlines. The premise is simple: run everything at max frequency with all cores activated. While this method is easy to dismiss because it might be assumed to have bad energy efficiency, it beats Poet in some scenarios according to our observations.

- **Max-N**: NVIDIA has implemented several hardware DVFS modes that are preloaded on Jetson TX2 and are accessible using the nvpmodel command. Max-N is the high performance mode that allows for the maximum frequency of CPU and GPU. However, the frequencies are adjusted dynamically at runtime.

- **Max-Q**: Max-Q is the most energy efficient mode that disables big cores entirely. Moreover, it caps the maximum frequency on both CPU and GPU for energy saving purposes.

We test each of the six approaches using five different DNN models: GoogleNet, CaffeNet, AlexNet, VGGNet, and ResNet. We first show the general usability of PredJoule by comparing it against other methods on all five DNN models, under two deadline configurations of tight and relatively loose deadlines. Next, we use the largest and the most advanced among these neural networks, which is ResNet, to demonstrate detailed latency and energy results under two scenarios: running DNN workloads with and without interference. We end by measuring PredJoule’s overhead.

**B. Generality**

First and foremost, we test the deadline miss ratio and energy usage of all six methods for running the five DNN models with two deadline settings: tight and loose deadlines. According to the specific computational demand, we set a tight (loose) deadline of 20ms / 20ms / 50ms / 50ms / 100ms (25ms / 25ms / 100ms / 100ms / 150ms) for CaffeNet, AlexNet, GoogleNet, VGGNet, and ResNet, respectively.

We test five different neural networks because each has a unique combination of layers with different sizes and depths. Fig. 5 shows the differences of the five neural networks in terms of the value of Uncertainty for different layers. As is evident in the figure, CaffeNet and AlexNet are relatively small networks. The most fluctuation for these neural networks is happening towards the end since that is where fully connected and softmax layers are located with small input sizes and large computations. While the same is true for VGGNet and ResNet, they are much more complicated, especially ResNet with over 200 layers. However, these extra layers are added in the name of improved accuracy. Finally, GoogleNet has a concentration of Relu layers in the middle that show up as high points of Uncertainty.

As is depicted in Fig. 6(a), PredJoule outperforms all other methods by a significant margin in most cases. It is observed that Race2Idle can perform as good as Max-Q or better. This is due to the fact that if a DNN is considered as a layer-oblivious blackbox, then it will have a low Uncertainty overall and will thus benefit from running faster. However, running faster will have a potentially adverse effect on layers with a high Uncertainty. Thus, our method can get much more efficient than Race2Idle by scaling back the DVFS for layers with high Uncertainty values. Finally, Poet does not perform well because it neither controls GPU nor being layer-aware. Our own port of Poet to GPU performs better than Poet because it controls GPU frequency for energy optimization. However, the effect of a layer-based design exploring layer characteristics is clearly depicted when comparing Poet-GPU to PredJoule.

The second scenario depicted in Fig. 6(b) shows the same...
TABLE III: Method deadline misses for various DNNs.

<table>
<thead>
<tr>
<th>Method</th>
<th>CaffeNet</th>
<th>AlexNet</th>
<th>GoogleNet</th>
<th>VGGNet</th>
<th>ResNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-N</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Max-Q</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Race2Idle</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Poet</td>
<td>1%</td>
<td>0%</td>
<td>4%</td>
<td>99%</td>
<td>16%</td>
</tr>
<tr>
<td>Poet-GPU</td>
<td>51%</td>
<td>9%</td>
<td>6%</td>
<td>99%</td>
<td>51%</td>
</tr>
<tr>
<td>PredJoule</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>3%</td>
<td>0%</td>
</tr>
<tr>
<td>After 50 iter.</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Fig. 7: Energy and latency of PredJoule compared to others for ResNet-100 over 50 iterations.

experiment but under a loose deadline setting. Having a loose deadline can show the efficacy of exploiting the system idle-ness under each method. For example, Race2Idle will suffer more when the system is idle for longer since its configuration does not change between scenarios. Other methods may suffer less because they switch to different settings accordingly. For two DNNs (VGGNet and ResNet), Max-N will be better than Max-Q in terms of energy efficiency. This scenario happens because the speed gain from running VGGNet and ResNet under Max-N will outweigh the added energy usage of it. Moreover, reactive methods such as Poet/Poet-GPU should be able to utilize looser deadlines to minimize energy since they make tasks run longer and may achieve better energy efficiency. While PredJoule benefits from a looser deadline compared to the tight case, Poet clearly cannot. This is due to the fact that Poet does not consider idleness as an adverse effect on energy and will not try to remedy it. Our method, on the other hand, will backtrack on high Uncertainty layers to find more efficient configurations. This effect is the most pronounced in VGGNet, which has a complicated structure.

Finally, we compare the deadline miss ratios of PredJoule against the other five methods in Table III. We run each task for 1000 iterations with the tight deadline and record a missed deadline. PredJoule can outperform others in almost all cases. For CaffeNet, AlexNet, and ResNet our method never misses a deadline. This is because PredJoule is able to recover the execution deficit inside the first iteration, rather than waiting until after the first iteration. However, our method misses 10% of deadlines for GoogleNet and performs worse than Race2Idle, Poet, and Poet-GPU in this specific scenario. This adverse effect is because GoogleNet contains many Relu layers that have a high Uncertainty. Thus, our method will take longer to adapt and meet the deadline, missing deadlines only at the beginning. We showcase the execution of our method compared to the others in a subsequent section. Finally, for the complicated VGGNet, our method only misses 3% of deadlines, which is much better than others. In fact, for VGGNet and our tightly set deadline, other methods will almost entirely miss their deadlines. This is a special case in which our layer-based design would shine the most, because VGGNet has many layers with low Uncertainty, even more so than CaffeNet and AlexNet. Nonetheless, our method still misses deadlines at the beginning and is not able to recover inside the first iteration. We believe our method can improve on timing predictability compared to competing approaches, as is evident by no deadline misses after 50 iterations.

C. Detailed Latency/Energy Performance

We now showcase a detailed set of latency and energy results for all the six methods over time for running ResNet with a deadline of 100ms for 50 iterations. ResNet is the largest and the most advanced among our tested neural networks. As seen in Fig. 8, other than Race2Idle, our method is the only one that can meet the latency requirements of ResNet. Regarding energy efficiency, our method also outperforms the other methods by a considerable margin. This set of results demonstrate that our design and implementation of PredJoule can achieve two (often) conflicting goals of timing predictability and energy efficiency. The consistency for Max-Q and Max-N are also apparent in this figure. However, they cannot match the latency of our method. Moreover, PredJoule can be as much as 66% and 70% more efficient compared to Max-N and Max-Q.

D. Adaptability With Interference

While the energy and latency can be optimized under PredJoule when the neural network is run as a standalone application, this scenario may not always happen in practice. In many cases, the neural network is accompanied by other
tasks that run concurrently. For example, in autonomous driving, it is possible that an object tracking task is running in parallel to a route planning task. In this experiment set, we test our method’s adaptivity with the presence of interference.

Fig. 8 shows the detailed energy and latency results for running ResNet with a deadline of 100ms under our method over 30 iterations. At iteration 10, we manually add interfering workloads. The interference will finish execution at iteration 20, as depicted by the cross lines. As seen in the figure, when the interference is introduced, the added energy and latency will initially hurt the execution of ResNet. Nonetheless, PredJoule is able to recover rather quickly and meet the deadline within a few iterations. Similarly, the energy usage will only increase slightly when the adaptation happens. PredJoule is able to respond to interference because it actively updates the execution deficit $\varepsilon$ at each layer boundary. $\varepsilon$ shall include any interference because it is always calculated in a real-time fashion in our implementation. Also, PredJoule is able to recover inside 10 iterations, making the execution more responsive.

E. Overhead

While it might be tempting to ignore the overhead for energy saving solutions, we believe it is crucial for two reasons. First, our method aims at meeting latency requirements. Thus, a solution with a high overhead may defeat this purpose. Second, most energy-related literature might assume that frequency change and DVFS, in general, are free. Our findings show that DVFS may not have a negligible overhead, especially for Linux since it requires multiple file accesses. In this set of experiment, we evaluate the runtime overhead incurred under PredJoule to assess whether our design and implementation is efficient in practice.

We compare our method against Poet for running ResNet with a deadline of 100ms, in terms of runtime overhead due to DVFS configuration adjustment (note that Poet is shown to be a rather efficient runtime DVFS solution [25]). As is depicted in Fig. 9, PredJoule is highly efficient even compared to Poet. This efficiency comes from two factors: First, PredJoule implements an efficient implementation of DVFS for Linux that relies solely on file handles and does not need to perform costly file accesses. Second, Poet iterates through all configurations before finding an optimal configuration. As is evident in Fig. 9 at around iteration 60, Poet has found an optimal configuration. However, the fluctuation in overhead still remains as Poet does a preliminary calculation before moving on to the next iteration. On the other hand, PredJoule mostly uses Equ. 5 and 6 which are done in constant time. For occasions that the history might be needed, we implement an efficient reverse history that gives the desired speed up in $O(1)$ time. Once our method has found an efficient configuration, the overhead becomes minimal since there is no need for DVFS operations any longer.

VI. RELATED WORK

The problem of optimizing energy while meeting hard or soft real-time constraints has received much attention in the literature [14], [22], [3], [42], [6], [13], [20], [23], [25], [21], [34]. Despite various manipulation mechanisms in detail, these works mostly study simplified workload models (e.g., the well-studied sporadic independent task model [35] or approximate applications where accuracy can be traded for performance and/or energy).

State-of-the-art research on optimizing energy efficiency for DNNs can be categorized into three categories: hardware approaches, DNN model optimization, and runtime approaches. Hardware approaches aim to optimize basic computations used in DNNs (e.g., convolution, matrix multiplication) [11], [15], [45], [33], [37], [13], [49], [4] through developing efficient hardware acceleration solutions. Model optimization approaches seek to compress or prune DNN models prior to execution [10], [48], [26], [30], [38], [47], [1]. Runtime approaches include offloading partial or entire DNN workloads to remote servers [23], [24], [9], [9], [46], and performing runtime approximation which trades performance with accuracy [32], [18]. To the best of our knowledge, none of these works simultaneously consider timing correctness and energy efficiency by using native DVFS.

Different from all these works on optimizing latency and energy efficiency, PredJoule represents a system solution that can achieve timing predictability while minimizing energy for running DNN workloads. A unique contribution of PredJoule is to explore dramatically different performance/energy characteristics of DNNs through developing a layer-based approach. This allows the system to smartly identify the best configuration for running each layer, such that timing can be tightly controlled on a per-layer basis while achieving the most energy saving by considering each layer’s performance/energy characteristics.

VII. CONCLUSION

This paper presents PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system. PredJoule presents a layer-based approach that controls the timing and optimizes energy efficiency through exploiting each layer’s performance/energy characteristics. Our experimental results demonstrate that PredJoule is able to achieve timing predictability with no deadline miss and much higher energy efficiency than prominent methods.
REFERENCES


