ABSTRACT

Title of Thesis: METAREASONING APPROACHES TO THERMAL MANAGEMENT DURING IMAGE PROCESSING

Michael K. Dawson Jr.
Master of Science

Thesis Directed by: Professor Jeffrey Herrmann
Department of Mechanical Engineering

Resource-constrained electronic systems are present in many semi- and fully-autonomous systems and are tasked with computationally heavy tasks such as image processing. Without sufficient cooling, these tasks often increase device temperature up to a predetermined maximum, beyond which the task is slowed by the device firmware to maintain the maximum. This is done to avoid decreased processor lifespan due to thermal fatigue or catastrophic processor failure due to thermal overstress. This thesis describes a study that evaluated how well metareasoning can manage the central processing unit (CPU) temperature during image processing (object detection and classification) on two devices: a Raspberry Pi 4B and an NVIDIA Jetson Nano Developer Kit.

Three policies that employ metareasoning were developed; one which maintains a constant image throughput, one which maintains a constant expected detection precision, and a third that trades between throughput and precision losses based on a user-defined parameter. All policies used the EfficientDet series of object detectors. Depending on the policy, these networks were either switched between, delayed, or both. This

thesis also considered cases that used the system's built-in throttling policy to control the temperature.

A policy was also created via reinforcement learning. The policy was able to adjust the detection precision and program throughput based on a set of states corresponding to the possible temperatures, neural networks, and processing delays.

All three designed metareasoning policies were able to stabilize the device temperature without relying on thermal throttling. Additionally, the policy created through reinforcement learning was able to successfully stabilize the device temperature, though less consistently. These results suggest that a metareasoning-based approach to thermal management in image processing is able to provide a platform-agnostic and programmatic way to comply with constant or variable temperature constraints.

METAREASONING APPROACHES TO THERMAL MANAGEMENT DURING IMAGE PROCESSING

by

Michael K. Dawson Jr.

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2022

Advisory Committee:
Professor Jeffrey Herrmann, Chair/Advisor
Assistant Professor Yancy Diaz-Mercado
Associate Professor Mark Fuge

Dedication

To my parents, Mike and Beth, and my sister, Molly, with love.

Acknowledgments

This thesis was made possible by the help of those around me.

First, I would like to thank my advisor, Dr. Jeffrey Herrmann, for supporting my work and pushing me to think of new ideas. His patience and thoughtfulness encouraged me to continually refine my work. I'd also like to thank Dr. Fuge and Dr. Diaz-Mercado for their membership in my thesis committee.

I'd like to thank my colleagues in the Keystone Program, especially Kevin Calabro, for supporting me through my first year in graduate school. My experience with ENES100 and those who support is a constant source of inspiration and motivation. Thank you, also, to the Laboratory Teaching Fellows for their help in working with Linux and Python.

I want to acknowledge the Maryland Robotics Realization Laboratory for their generous loan of an NVIDIA Jetson Nano Test Kit.

Finally, I'd like to thank the U.S. Army Research Laboratory for its funding throughout my second year in graduate school.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ARM	Advanced RISC Machine
COCO	Common Objects in Context
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
EDL	EfficientDet-Lite
FPS	Frames per Second
GHz	Gigahertz
GPIO	General-purpose Input/Output
GPU	Graphics Processing Unit
IPD	Initial Pause Duration
MR	Metareasoning
MSTE	Mean Square Temperature Error
MTTF	Mean Time to Failure
PAC	Pause Adjustment Coefficient
RAM	Random Access Memory
RL	Reinfocement Learning
RPi	Raspberry Pi
RISC	Reduced Instruction Set Computer
SoC	System on a Chip
TAC	Throughput Adjustment Coefficient
TF	TensorFlow
TFLite	TensorFlow Lite

Chapter 1: Introduction

Many autonomous systems rely on image processing for object recognition. These applications require a computer to take an input image or video feed and output what objects are present and what their locations are in the image or video. Though occasionally performed via processes such as edge detection, neural networks have become increasingly popular as a method for image processing in recent years. Tesla autopilot, for example, uses video feeds from cameras around the vehicle to recognize people, stop signs, and many other objects critical to the driving experience [1]. This need to quickly process images using neural networks, which require millions of computations, increases the processor's temperature [2].

Any processes performed on a computer processor will increase its temperature. Normally, the processor's temperature will fluctuate as the computational load varies, but when a computationally heavy task such as image processing is performed continuously, the temperature may increase enough to damage the processor [3]. This is a common risk associated with heavy compute loads. Thus, it is important to have a thermal management policy that can maintain the processor's temperature within an acceptable range.

This thesis describes the design and performance of three metareasoning policies which maintain a desired temperature while modifying detection precision and throughput.

The metareasoner can monitor the device temperature in order to control the image processing procedure by switching between different neural networks and adjusting the processing frequency via pauses of varying length to adapt to changes in both the ambient and device temperatures. The metareasoner maintains a record of expected detection precision, temperature, and detection frequency to adjust the policy parameters for on-the-fly adjustments to the detection algorithm in accordance with a desired trade-off between precision and throughput.

1.1 Motivation

Object detection is an important subset of image processing in various fields, as it provides the benefits of human-like recognition capabilities applied at scale. Networks trained in object detection are used in tasks such as pedestrian detection, facial and text recognition, sign detection, and more [4]. When characterized by the rate and overall quantity of images to be processed, object detection tasks can be split into two groups. The first group of tasks requires a constant throughput of images and therefore puts a constant thermal load on the active processor. The second group of tasks involves processing images on-demand, placing the active processor under acute thermal stress. The first scenario is associated with thermal fatigue and failure due to cyclic changes in temperature, while the second is associated with thermal overstress and failure due to one-time significant changes in temperature [5].

When designing systems that will perform tasks that fall into the first group, designers will typically consider the thermal capacity, or maximum sustainable heat generation, of

their compute devices. Therefore, they can structure their object detection task such that it operates at a certain temperature within the limits of the compute device. Systems that must perform tasks in the second group, however, must rely on the device's built-in thermal throttling technique, which augments the device's functionality to limit temperature increase. The method by which the device maintains a maximum temperature, as well as the maximum temperature itself, varies from device to device. For example, a Raspberry Pi 4B has a thermal throttling temperature of approximately 82 °C [6], while an NVIDIA Jetson Nano Developer Kit has a throttling temperature of 97 °C [7].

It is therefore advantageous for system designers to be able to specify a desired temperature for a certain task across multiple devices with different hardware. This allows thermal reliability models to be based on software parameters rather than hardware specifications because a consistent thermal load can be expected, regardless of task precision or throughput.

1.2 Contributions

This thesis details the design and performance of a novel metareasoning approach to thermal throttling during image processing. Four policies are presented: three designed based on empirical evidence, and another learned by a reinforcement learning agent. These policies monitor the device's temperature, the task's throughput, and the recent detection accuracy to make decisions about which neural network to use and whether or not the overall throughput should be decreased or increased. These policies, shown in Table 1.1 are designed to solve the issue of decreased performance due to spikes in

computational load for tasks that fall into group two [8], as well as provide a platform-agnostic method for temperature-aware object detection for consistent thermal reliability studies.

Table 1.1: Metareasoning policies.

Metareasoner Category	Policy
Designed	Policy 1 Policy 2 Policy 3
Learned	Policy 4

1.2.1 Designed Metareasoner

Rather than utilizing dynamic voltage and frequency scaling (DVFS), these policies attempt to adjust for temperature changes by inserting pauses after each image is processed. On longer time scales, this mimics the effect of DVFS without reducing the total computational capacity of the central processing unit (CPU), without the drawbacks associated with on-demand tasks during DVFS which are noted by Wang et al. [8]. If throughput is of a higher priority than precision, the metareasoner will also switch to progressively smaller neural networks, each requiring fewer computations to process an image.

The results of our experiments showed that these designed metareasoning policies are able to maintain a chosen desired temperature while prioritizing either detection precision or throughput on multiple hardware platforms.

1.2.2 Learned Metareasoner

This policy performed the same function as the designed metareasoner, but learn an optimal policy through Q-learning. This metareasoner determines the state of the system and learns a desired action based on an estimated future reward associated with the squared temperature error.

The policy resulting from reinforcement learning training was also able to stabilize the temperature, though less consistently than the designed policies.

1.3 Overview

Chapter 2 details background information about the problem of thermal management and reviews related studies. It also provides information about different image processing techniques, metareasoning, and reinforcement learning. Chapter 3 describes the experimental methodology for the designed metareasoning policies. This includes the experimental apparatus, policy descriptions, results, and discussion. Chapter 4 details the experimental methodology for the learned metareasoning policy, including the policy descriptions, results, and discussion. Chapter 5 summarizes the results of the performed experiments and draws relevant conclusions.

Chapter 2: Background

2.1 Thermal Management

Central Processing Units (CPUs), or processors, are made of transistors. When transistors are in their "off" state, they have a high resistance compared to the "on" state. In their "on" state, current flows through the transistor, encountering a low, but non-zero resistance. When this happens, energy is dissipated in the form of waste heat.

The processor has a limited range of temperatures at which it can operate. Each processor has a minimum operating temperature recommended by the manufacturer. Below this temperature, damage to the device's circuitry may occur due to thermal contraction. On the other end, each processor also has a maximum specified operating temperature. Above this temperature, the risk of permanent damage increases. Thermal expansion can physically damage components by separating circuits. Additionally, silicon's resistivity decreases as temperature increases, which causes more current to flow and more heat to be generated, and so on. This thermal overstress can lead to the CPU becoming nonfunctional.

Even if the temperature is not high enough to significantly damage the device, periodic increases and decreases in temperature, known as thermal fatigue, can eventually lead to failure of the CPU [5].

Therefore, the fundamental problem of thermal management arises from two conflicting characteristics of a processing unit: (i) when active, the device generates heat at a rate that is proportional to the number of computations performed (which increases its temperature), but (ii) the device lifetime is inversely correlated with the mean operating temperature. This has led to the development of thermal throttling strategies as processing units are pushed towards ever-higher computation rates. These strategies fall into two main categories depending on what actuator they are using to control the temperature.

2.1.1 Internal Management

Internally-managed systems rely on changes to the input to the processing unit to lower the number of computations performed, thereby lowering the operating temperature. A popular strategy for achieving this is dynamic voltage and frequency scaling (DVFS) [9]. This method adjusts the input frequency to the processing unit to slow computation frequency. The slower computation frequency also leads to a lower operating temperature and power consumption.

2.1.2 External Management

Alternatively, the temperature of the processor can be controlled by changing the environment in which it operates. Typically, this is done by providing increased cooling through increased conductive or convective heat transfer.

Convection can be increased passively by increasing the surface area of the processor through fins, or actively by increasing the convection coefficient through increased mass

flow [10].

Benoit-Cattin et al. [11] showed that dynamic active cooling can increase the efficiency of image processing on a Raspberry Pi. Their approach, which used an external fan that was controlled by the Raspberry Pi 4B, increased the image processing throughput.

2.2 Image Processing

One of the main tasks that CPUs are required to perform is image processing. This task can take many forms, but the one which is of particular interest in recent years is object detection [4, 12].

Object detectors are a variant of image classifiers which themselves are a type of neural network. These neural networks take as input the gray scale or color values of an input image and perform a series of linear and non-linear operations on them until a set of values is output. For image classifiers, these outputs are a set of probabilities associated with a "class" of image. This network learns which class is correct via supervised learning, a process in which the network is provided with a set of inputs and associated, labeled outputs. In this process, it modifies the weights between each node to determine the value which provides the correct output most often. An object detector modifies the output of an image classifier to additionally locate where each object, predicted with at least a certain confidence, is in the scene.

2.3 Metareasoning

Metareasoning is a higher-level form of programmatic thinking that can lead to improved performance in autonomous agents. Metareasoning achieves this by monitoring the agent and its decision-making environment to determine how to approach its current decision [13, 14]. Without intervention, many programs operate toward the maximum confines of their environment. Autonomous rovers, for example, may travel at high speeds while searching an unknown environment, unknowingly shortening their battery life and leading to increased time between charging and shorter lifespans. Metareasoning provides a way for designers to help autonomous agents better understand the larger context of their mission so they can perform better in metrics for which they would otherwise not be able to account. In the case of this thesis, this means providing the program with knowledge of the device temperature, processing throughput, and detection precision so that it can optimally perform its task under external constraints.

I am unaware of any work explicitly detailing metareasoning as a method for thermal management specific to image processing. There is, however, some use of metareasoning in related areas - particularly, image processing.

Nguyen et al. [15] used "frame skipping" to achieve a desired output video quality. Given a set input frame rate, their system was designed to discard certain images if a consistent output frame rate was not maintainable due to limited computational resources on the client-side of the gaming system. Although this was not explicitly considered metareasoning by the authors, their implementation involved a "decision engine" which would "decide the optimal frame rate given the current system status." Thus, this is a type

of metareasoning.

Lee et al. propose "Virtuoso," a metareasoner which picks an object detector and tunes its parameters to optimally perform video processing based on user-defined energy, accuracy, and speed requirements [16]. This method finely adjusts neural networks so that they perform as best as possible.

Although DVFS is a proven methodology for power and thermal management, it does not perform optimally with tasks that require short bursts of high CPU utilization [8]. Additionally, not all devices can be cooled externally, so the work by Benoit-Cattin et al. cannot be applied to systems that rely only on passive cooling. The approach used by Nguyen et al. is designed to maintain a desired frame rate, but it does not account for temperature. Virtuoso also does not account for temperature. While the energy constraints could be intelligently defined by the user, the metareasoner is not explicitly aware of the temperature of the processor.

Thus, the method in chapter 3 sought to determine whether a metareasoning approach would be able to control a processor's temperature, which would be useful in situations where other techniques are infeasible or expensive.

2.4 Reinforcement Learning

Reinforcement learning is a field of machine learning in which an agent learns which actions to perform in a certain state based on the reward associated with each state. One method of doing this Q-learning, wherein an agent in an environment learns an optimal pairing of states and actions, known as a policy [17].

Das et al. researched the effectiveness of thermal management via reinforcement learning in 2014. Their approach sought to control the frequency of the CPUs as well as the assignment of certain processing threads to certain CPU cores. Their approach offered a two to three times improvement in the mean time to failure (MTTF) for the processor, depending on whether it was applied within or across different programs [18].

This approach addresses many of the same goals in this thesis, but is unable to affect the decisions made within a program, only how the CPU distributes and paces the program tasks. Therefore, the approach demonstrated in chapter 4 of this thesis is unique in that it allows the reinforcement agent to take actions within the program, particularly switching between neural networks.

Chapter 3: Designed Policies

3.1 Research Question

A trade-off is present between processing speed and detection precision of neural network object detectors [19, 20, 21]. Their performance is additionally constrained by the maximum temperature at which the device is capable of operating. Considering that temperature is a key factor in the performance and lifespan of electronic devices, is there a way to use metareasoning to sacrifice (i) speed, (ii) precision, or (iii) both to maintain a certain temperature across devices?

3.2 Experimental Approach

To answer this question, two different policies were developed to reduce the processing throughput or detection precision. A third policy was developed as a generalization of the first two policies which is able to trade off between both throughput and precision while maintaining temperature.

3.2.1 Hardware

3.2.1.1 Raspberry Pi 4B

Tests were performed on a Raspberry Pi 4B. This device was chosen because of its low price, popularity, and our familiarity with the default operating system, Raspbian.

The Raspberry Pi uses a Broadcom BCM2711 system on a chip (SoC), which contains a quad-core Cortex-A72 processor [22]. The temperature measured by the SoC's internal sensor is read from the Linux "/sys" directory [22, 23].

A 3D-printed case was created for the device [24]. A small 5V fan was installed in the case and controlled with the GPIO Python package [25]. The final setup is shown in Figure 3.2.

Figure 3.1: Raspberry Pi 4B in its case with fan.

3.2.1.2 NVIDIA Jetson Nano Developer Kit

Tests were additionally performed on an NVIDIA Jetson Nano Developer Kit. This device was chosen for its excellent ability to perform neural network operations, as well as its popularity in the machine learning field.

The Nano has a Quad-core ARM A57 SoC which operates at 1.43 GHz. It additionally has a 128-core Maxwell GPU; GPUs are notable for their competence in performing operations in parallel, a feat which is desirable in machine learning [26]. For the purposes of consistency between devices, the GPU was not used to accelerate object detection on the Nano.

Figure 3.2: The NVIDIA Jetson Nano Devloper Kit [26].

3.2.2 Software

Object detection was performed via the EfficientDet architecture, created by Tan et al. in 2019 [27]. This family of efficient object detectors provides sufficient recognition precision with low latency on resource-constrained devices. The authors provide eight individual architectures, D0-D7, which each accepts a different input image resolution.

As the network increases in size, its detections are increasingly accurate but take longer to perform. This is shown in Table 3.1 [28].

EfficientDet D0-D4 were used during testing. Each was trained on the COCO2017 dataset [29] in TensorFlow by the TensorFlow Team [30]. The models were then optimized to be compatible with TFLite, TensorFlow's mobile neural network framework. They are therefore referred to as EfficientDet-Lite[0-4] (EDL-[0-4]). The important characteristics for these neural networks is shown in table 3.1.

Table 3.1: EfficientDet D0-D4 (for TFLite) characteristics. Measurements performed by the TensorFlow Team on a Pixel 4 using the 2017 COCO validation dataset.

Model Architecture	Size (MBs)	Latency (ms)	Average Precision	Identifier
EfficientDet-Lite0	4.4	37	25.69%	$Network_4$
EfficientDet-Lite1	5.8	49	30.55%	$Network_3$
EfficientDet-Lite2	7.2	69	33.70%	$Network_2$
EfficientDet-Lite3	11.4	116	37.70%	$Network_1$
EfficientDet-Lite4	19.9	260	41.96%	$Network_0$

Note: Adapted from *Object Detection with TensorFlow Lite Model Maker* by the TensorFlow Team, 2022.

Each test used the same fundamental script with changes only occurring depending on parameter values and the metareasoning policy. This program is shown in Algorithm 1.

Three different metareasoning policies were tested. The first, policy 1, adjusts the throughput of the image processing to maintain a desired temperature. The second, policy 2, changes which neural network performs image processing while maintaining a constant throughput to maintain a desired temperature. The third, policy 3, does what both policy 1 and 3 do, but uses a parameter to determine how much throughput should be increased

Algorithm 1 Main Program

$T_s \leftarrow 50°C$ ▷ Set start temperature
Require: $T \leq T_s$
$t_0 \leftarrow 0s$ ▷ Set test start time
$t_s \leftarrow [300, 600]s$ ▷ Set test duration for policy [1,2]
$T_d \leftarrow [70, 55]°C$ ▷ Set desired temperature for [RPi, Nano]
$p_d \leftarrow p_d$ ▷ Set initial pause duration
$p_a \leftarrow p_a$ ▷ Set pause adjustment coef.
$TAC \leftarrow TAC$ ▷ Set throughput adjustment coef.
$Strategy \leftarrow Strategy$ ▷ Set switching strategy
$Network \leftarrow Network_4$ ▷ Set starting network
$N_{d_{max}} \leftarrow N_{d_{max}}$ ▷ Set maximum network duration
$Th_{max} \leftarrow 1/N_{d_{max}}$ ▷ Set maximum throughput
$P_{min} \leftarrow P_{min}$ ▷ Set minimum network precision
$P_{max} \leftarrow P_{max}$ ▷ Set maximum network precision
while $t < t_s$ **do** ▷ t is the program time
    Process image
    Record N_d ▷ The processing duration
    Record T ▷ The current temperature
    Perform Policy [1 | 2 | 3]
    Record L_d ▷ The loop duration
    Record T
end while

for each unit of precision decrease.

Because these tests depend on the temperature of the SoC, it was important that all tests began at the same temperature and have similar ambient thermodynamic environments. While idle, the temperature of the SoC in a 22°C room and without active cooling is approximately 57°C. Between tests, the device was cooled via convective heat transfer by the 5V fan attached to the case. The cooling cycle, which ran before each test if the temperature was greater than the set starting temperature, activated the fan until a specific temperature was reached. So this cycle was activated even if the device was near its idle temperature, the starting temperature was chosen to be 50°C, which is below the idle temperature.

The Nano has excellent passive cooling in the form of fins on its SoC. Its idle temperature is measured to be approximately 35 °C. Because the idle temperature is so low, 50 °C was chosen to be the starting test temperature. If the temperature is too low before the start of a test, EfficientDet-Lite4 is run continuously until the start temperature is reached. Between tests, cooling occurs via natural convection.

The desired temperature for each device was chosen so that it was reachable during each test. This way, metareasoning would actively modify the image processing loop during at least some portion of the test. The desired temperature for the RPi was chosen to be 70 °C so that even the least computationally intensive network, EfficientDet-Lite0, was able to reach the desired temperature. The desired temperature for the Nano was chosen to be 55 °C. Because the Nano is more efficiently able to remove heat from its CPU than the RPi thanks to its large set of fins, its temperature rises more slowly. Therefore, the desired temperature had to decrease so that it could be reached within the same time constraints.

We tracked multiple variables during each test, measured once after the image was processed and again after the pause was inserted. The most important of these were the following: SoC temperature, SoC CPU use, pause duration, processing duration, loop duration, and average expected precision.

3.2.3 Throughput Adjustment Policy - Policy 1

To alleviate computational load, the computational rate must be decreased. The proposed method does this on a "macro" level by inserting a pause of varying duration during an image processing loop. While paused, the device idles at a low CPU load.

On average, this decreases the computational frequency, therefore lowering the SoC heat output and the measured temperature.

Algorithm 2, inserts a pause with a duration proportional to the overshoot of the desired temperature. Because only one network is used, average precision remains constant while throughput drops over time to maintain a constant temperature.

Algorithm 2 Throughput Adjustment Policy (Policy 1)

$p_d \leftarrow p_d + p_a \times (T - T_d)$
if $p_d < 0s$ **then**
    $p_d \leftarrow 0s$
end if
Wait p_d seconds

For Algorithm 2, 25 tests were performed for each network. The values of both p_a and p_d were incremented to create the set of tests, which are shown with their identifiers in Table 3.2. For each value of p_d, a test was performed where p_a was zero. This represents a program where the loop length is static and metareasoning is inactive.

Table 3.2: Policy 1 test matrix.

p_d (s)	p_a (s/°C)				
	0	0.05	0.1	0.15	0.2
0	1	2	3	4	5
0.5	6	7	8	9	10
1	11	12	13	14	15
1.5	16	17	18	19	20
2	21	22	23	24	25

In these tests on the RPi, the temperature reaches a maximum of approximately 82°C. This occurs because, whenever the temperature of the Raspberry Pi 4B is above 75°C, the CPU operating speed is lowered via DVFS [6]. In practice, the observed throttling temperature is 82°C.

Meanwhile, on the Nano, the cooling is sufficient that the thermal throttling temperature of 97 °C is not reached [7].

Because 125 tests had to be performed and iterated upon, a test duration of five minutes was used. This allowed for a full set of tests to be performed over the course of a day.

3.2.4 Network Switching Policy - Policy 2

Algorithm 3, switches between the variants EfficientDet on an ordinal scale. Switching occurs if a certain temperature threshold is met. The threshold for each network depends on the switching strategy. Neither p_a nor p_d needed to be varied because they were not used in the program. Therefore, each test was performed for a duration of 10 minutes because fewer tests needed to be perfomed.

Algorithm 3 Network Switching Policy (Policy 2)

Wait p_d seconds
Perform Switching Strategy [1 | 2 | 3]
$p_d \leftarrow N_{d_{max}} - N_d$ ▷ N_d is the next network duration
$P_{avg} \leftarrow$ Precision ▷ Average expected precision across the current and last 4 frames
if $P_{avg} = P_{min}$ **then**
    $p_{d,old} \leftarrow p_d$
    Perform Policy 1 ▷ Calculate new pause duration
    $p_d \leftarrow p_{d,new} + p_{d,old}$
end if

Three different switching strategies were implemented. The first strategy iterates through the set of networks in order of descending or ascending computational complexity, the second, the networks to a linearly scaling range of temperatures, and the third assigns the networks to a range of exponentially scaling temperature ranges. The summary of switching strategies which were tested are listed in Table 3.3.

Table 3.3: Policy 2 strategies.

Strategy	Summary
1	"Upshift" or "downshift" between EDL variants when the temperature error is too low or too high, respectively.
2	Each variant of EDL is assigned to a temperature range between the start temperature and desired temperature. The five temperature ranges are equally sized.
3	Each variant of EDL is assigned to a temperature range between the start temperature and desired temperature. The five temperature ranges decrease in size exponentially as the temperature approaches the desired temperature.

Strategy 1 is shown in Algorithm 4. Each loop, this strategy iterates once through the network list in decreasing order of computation time while the measured temperature is greater than the desired temperature. Otherwise, it iterates through the list in the opposite direction from the last network used.

Algorithm 4 Switching Strategy 1

$i \leftarrow NetworkIndex$
if $T < T_d$ **then**
    if $i \neq 4$ **then**
        $Network \leftarrow Network_{i+1}$
    else
        $Network \leftarrow Network_i$
    end if
else
    if $i \neq 0$ **then**
        $Network \leftarrow Network_{i-1}$
    else
        $Network \leftarrow Network_i$
    end if
end if

Strategy 2 is shown in Algorithm 5 and creates five thermal zones between the starting temperature and the desired temperature. Whenever a threshold is crossed, the next network is switched to. For example, suppose that the first quintile extends from 50

°C to 55 °C. In the image processing loop, EfficientDet D4 is used to detect objects, then the temperature is measured to be 56 °C. Metareasoning will switch the neural network for the next loop to be EfficientDet D3 and insert a pause which will maintain the overall image throughput. Therefore, a small precision trade-off is made to preserve constant temperature and precision. A depiction of the quintiles used for the Raspberry Pi are shown in Figure 3.3.

Algorithm 5 Switching Strategy 2

$q_{lin} \leftarrow (T_d - T_s)/5$
$i \leftarrow 0$
while $T < (T_s + q_{lin})$ **do**
    $i \leftarrow i + 1$
    $q_{lin} \leftarrow (T_d - T_s)/5 * i$
    if $i = 4$ **then**
        Break
    end if
end while
$Network = Network_i$

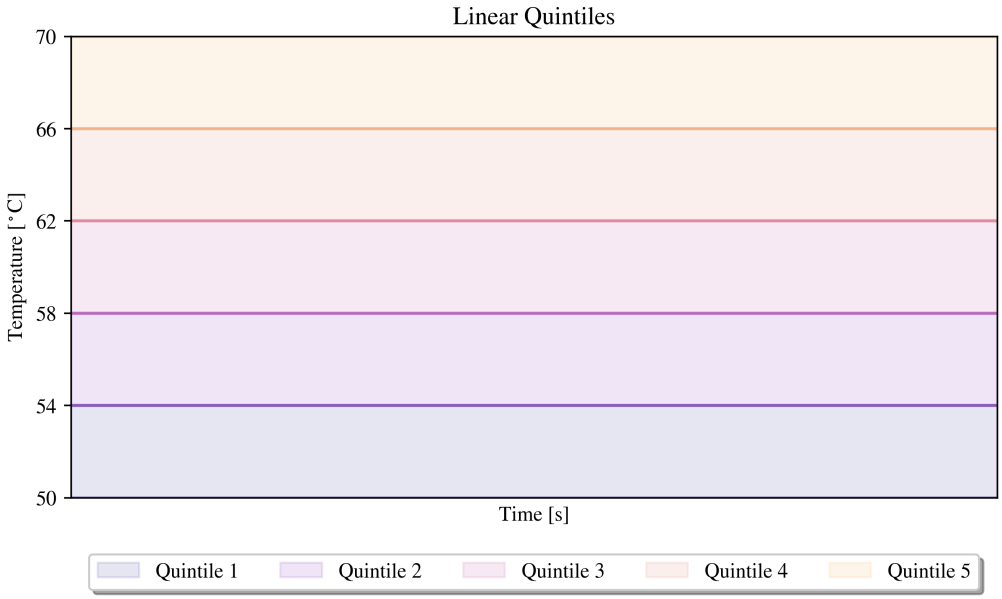


Figure 3.3: Linear quintiles for the Raspberry Pi tests using switching strategy 2.

Strategy 3, shown in Algorithm 6, instead uses five exponentially smaller thermal

zones. A depiction of these zones is shown in figure 3.4. Each quintile is half the size of the one below it, except for the uppermost two, which are the same size.

Algorithm 6 Switching Strategy 3

$q_{exp} \leftarrow (T_d - T_s)/2$
$i \leftarrow 0$
while $T < (T_s + q_{exp})$ **do**
    $i \leftarrow i + 1$
    $q_{exp} \leftarrow q_{exp} + (T_d - (q_{exp} + T_s))/2$
    if $i = 4$ **then**
        Break
    end if
end while
$Network = Network_i$

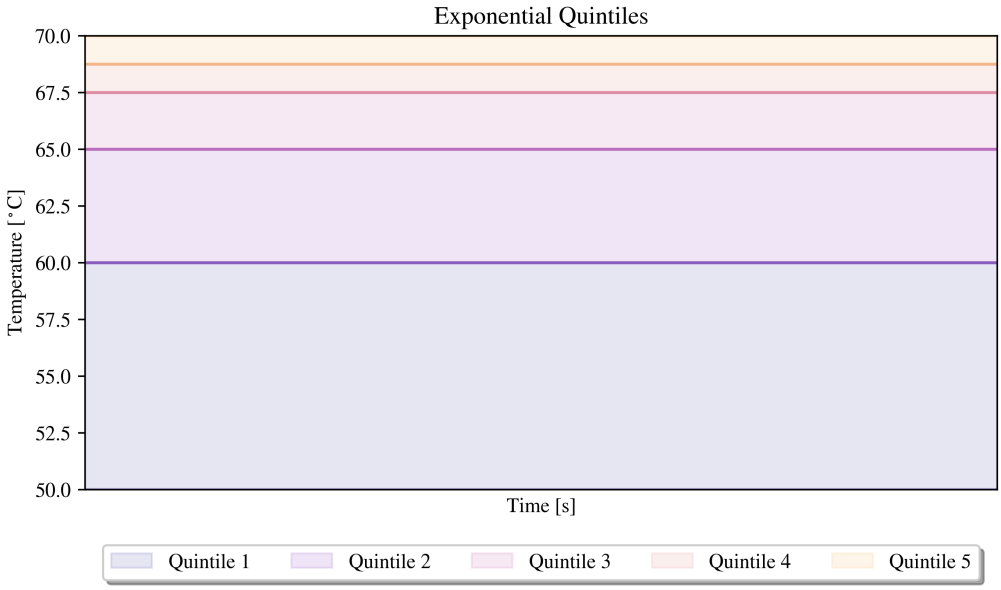


Figure 3.4: Exponential quintiles for the Raspberry Pi tests using switching strategy 3.

3.2.5 Hybrid Policy - Policy 3

Metareasoning policy 3, rather than prioritizing solely precision or throughput, provides a trade-off between both. The ratio of the trade-off is determined by a throughput

adjustment coefficient (TAC). It represents the preference for throughput loss rather than precision loss, with larger values representing increased throughput loss per unit of precision change. Its units are $1/s$ because it is a measure of the frames processed per second per precision loss. The TACs which were tested are shown in table 3.4.

Table 3.4: TAC variants.

TAC	Expectation
0	Throughput remains constant while precision decreases. Mimics policy 2.
1	Throughput decreases slightly as precision decreases.
2	Throughput decreases moderately as precision decreases.

In each loop, the switching strategy is used to determine which network will be used in the next cycle. Then, the average expected precision across the previous five frames is calculated. The difference between this value and the maximum expected precision is multiplied by the TAC to find the decrease in throughput between this loop and the next. Then, the pause duration is calculated from the difference. This policy is shown in Algorithm 7. Because the pause length assigned in policy 3 is independent of the temperature, this policy is best framed as a variant of policy 2.

Algorithm 7 Hybrid Policy (Policy 3)

Wait p_d seconds
Perform Switching Strategy [1 | 2 | 3]
$P_{avg} \leftarrow$ Precision
$Th_d \leftarrow Th_{max} - (P_{max} - P_{avg}) * TAC$
$ll_d \leftarrow 1/Th_d$ ▷ Calculate desired loop length from desired throughput
$p_d \leftarrow ll_d - N_d$
if $P_{avg} = P_{min}$ **then**
    $p_{d,old} \leftarrow p_d$
    Perform Policy 1
    $p_d \leftarrow p_{d,new} + p_{d,old}$
end if

To determine the expected duration of each network, five images were processed for each version of EfficientDet-Lite and the average processing duration was calculated.

In total, 12 tests are performed for policies 2 and 3. These are identified in Table 3.5.

Table 3.5: Policies 2 and 3 test matrix.

Strategy	TAC (1/s)			
	Policy 2	Policy 3		
	N/A	0	1	2
1	S1-TN	S1-T0	S1-T1	S1-T2
2	S2-TN	S2-T0	S2-T1	S2-T2
3	S3-TN	S3-T0	S3-T1	S3-T2

3.3 Results

3.3.1 Throughput Adjustment Policy - Policy 1

There are three main records of interest for policy 1. These are the temperature of the SoC, the loop length, and the CPU usage. To emphasize the effect of metareasoning on each program variant, Tests 1, 5, 20, and 25 are shown for each network.

3.3.1.1 Raspberry Pi 4B Temperature

The record of temperature over time for these tests on the RPi are shown in Figures 3.5, 3.6, 3.7, 3.8, and 3.9.

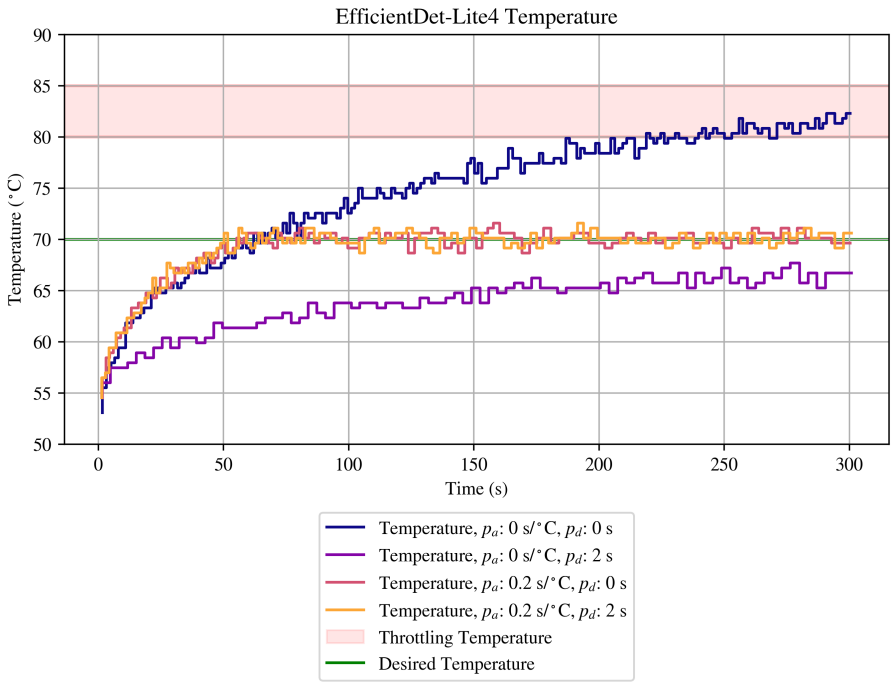


Figure 3.5: EfficientDet-Lite4 metareasoning effect on temperature on the RPi.

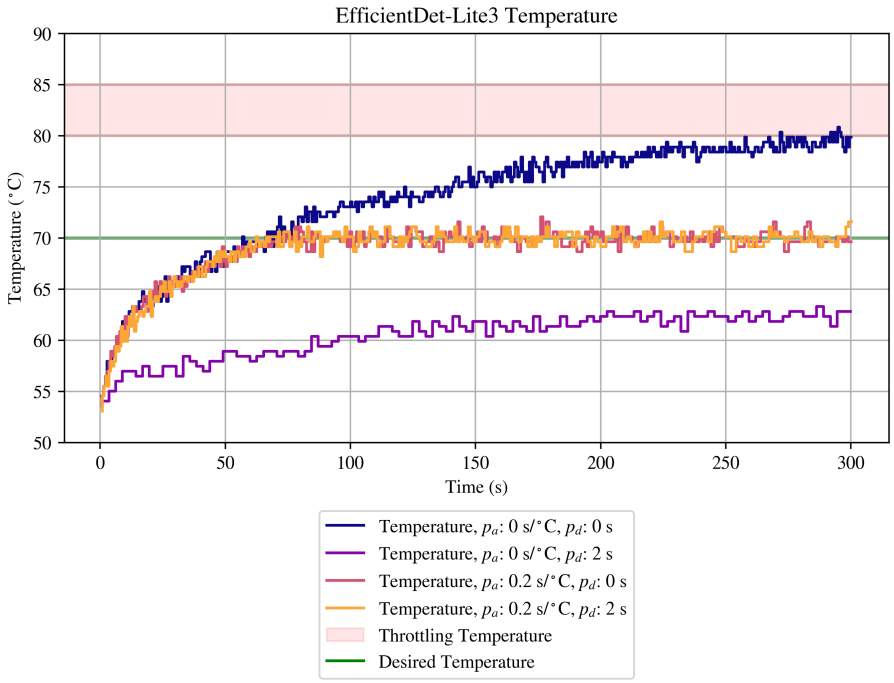


Figure 3.6: EfficientDet-Lite3 metareasoning effect on temperature on the RPi.

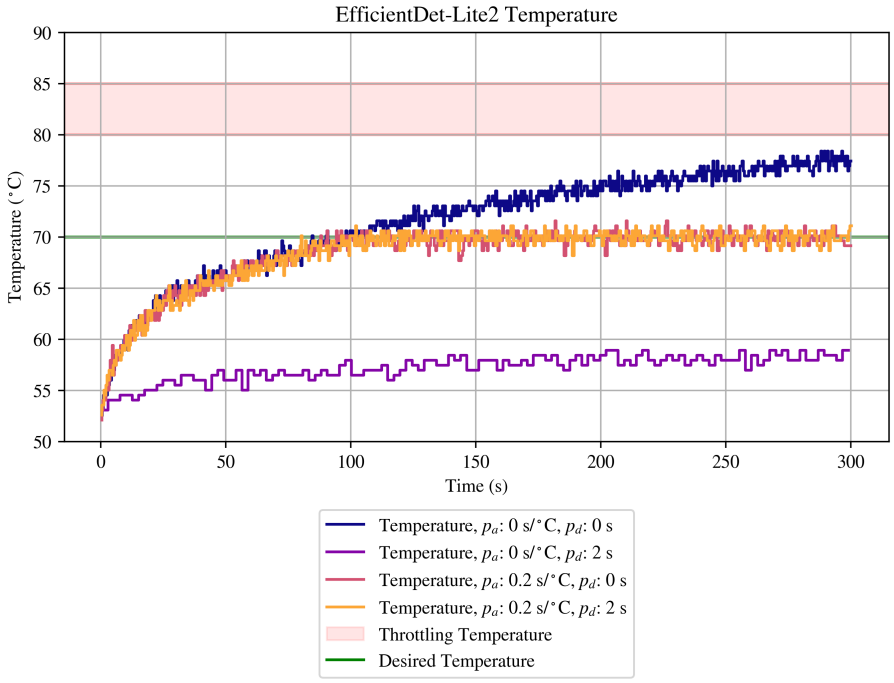


Figure 3.7: EfficientDet-Lite2 metareasoning effect on temperature on the RPi.

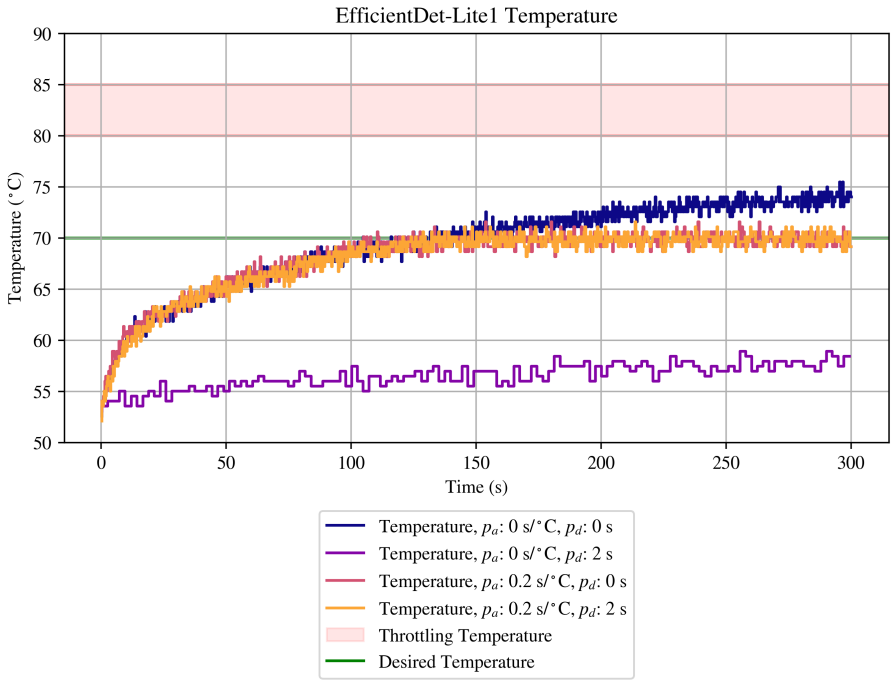


Figure 3.8: EfficientDet-Lite1 metareasoning effect on temperature on the RPi.

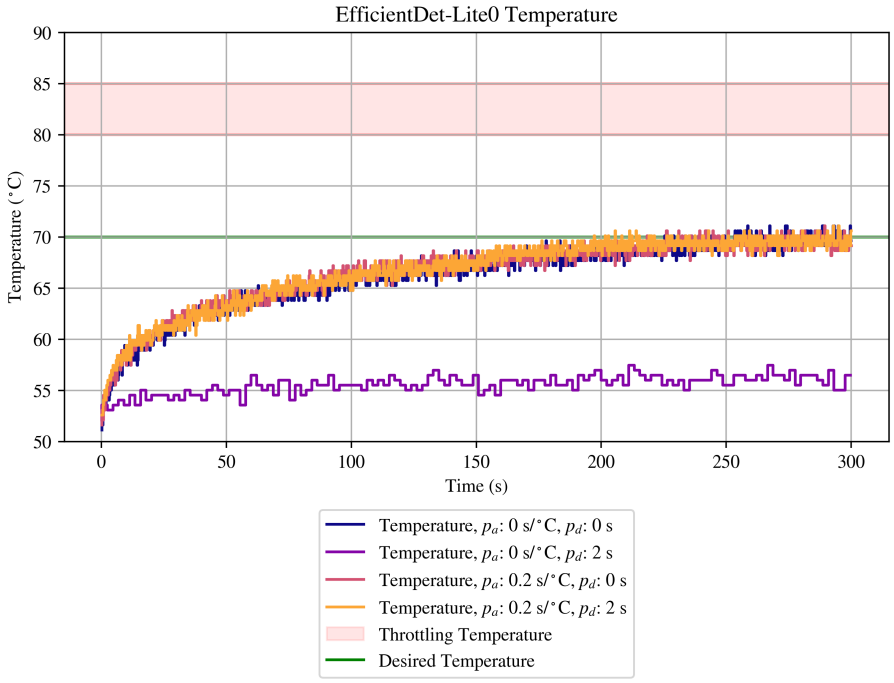


Figure 3.9: EfficientDet-Lite0 metareasoning effect on temperature on the RPi.

3.3.1.2 NVIDIA Jetson Nano Developer Kit Temperature

The record of temperature over time for these tests on the Nano are shown in Figures 3.10, 3.11, 3.12, 3.13, and 3.14.

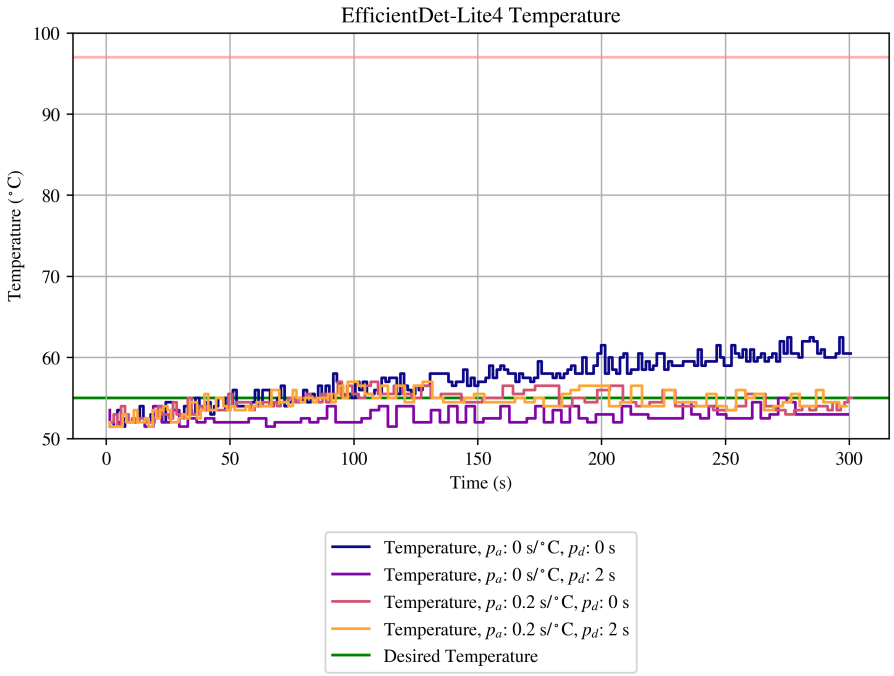


Figure 3.10: EfficientDet-Lite4 metareasoning effect on temperature on the Nano.

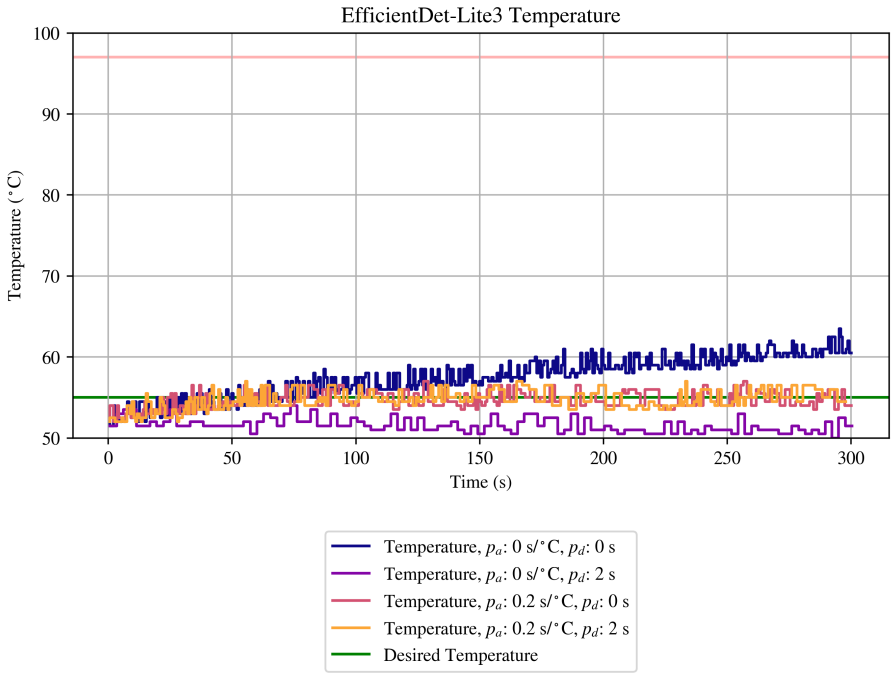


Figure 3.11: EfficientDet-Lite3 metareasoning effect on temperature on the Nano.

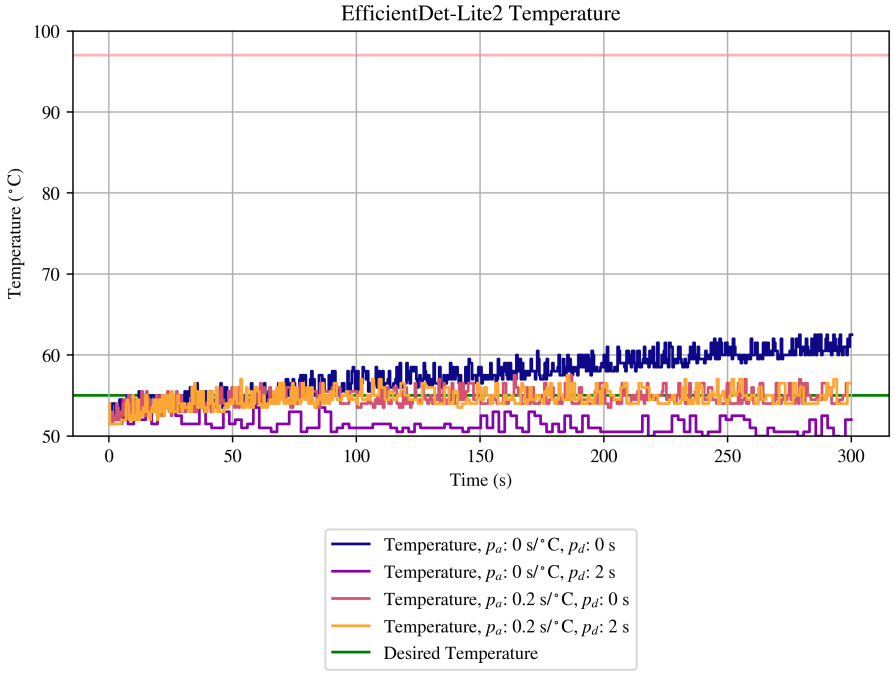


Figure 3.12: EfficientDet-Lite2 metareasoning effect on temperature on the Nano.

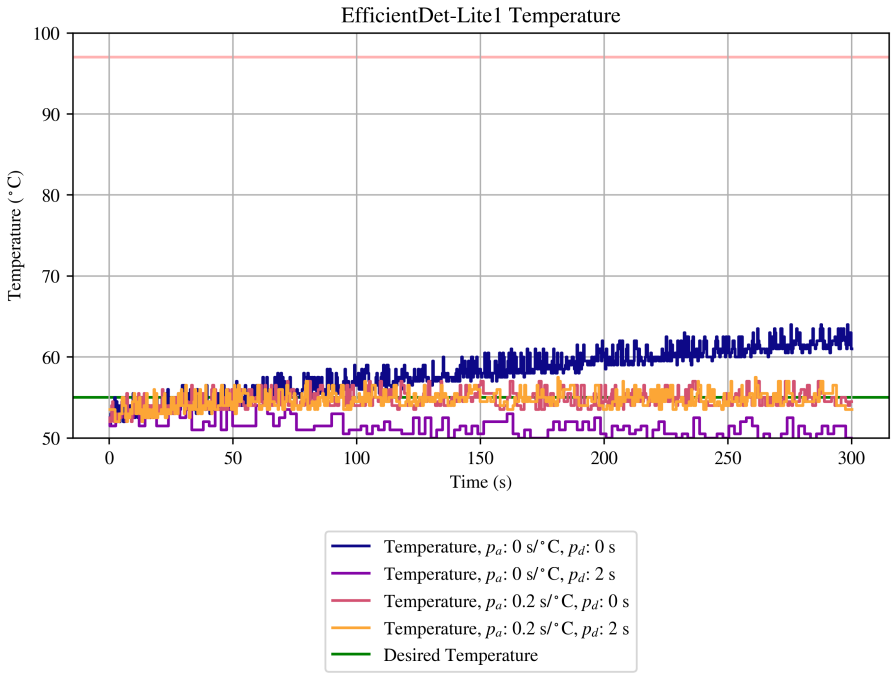


Figure 3.13: EfficientDet-Lite1 metareasoning effect on temperature on the Nano.

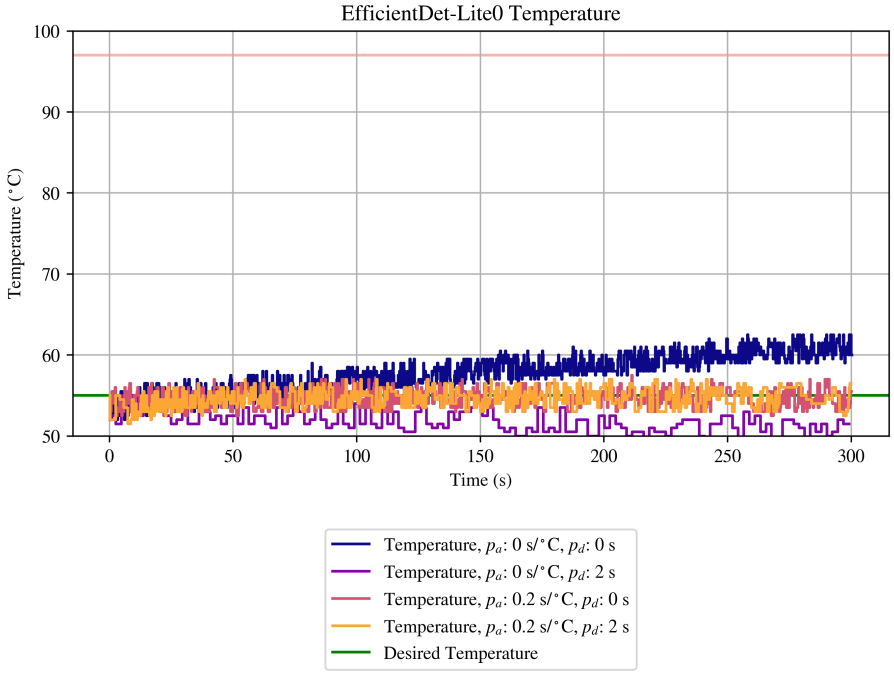


Figure 3.14: EfficientDet-Lite0 metareasoning effect on temperature on the Nano.

3.3.1.3 Raspberry Pi 4B Loop Length

The record of the program loop length over time for these tests on the RPi are shown in Figures 3.15, 3.16, 3.17, 3.18, and 3.19.

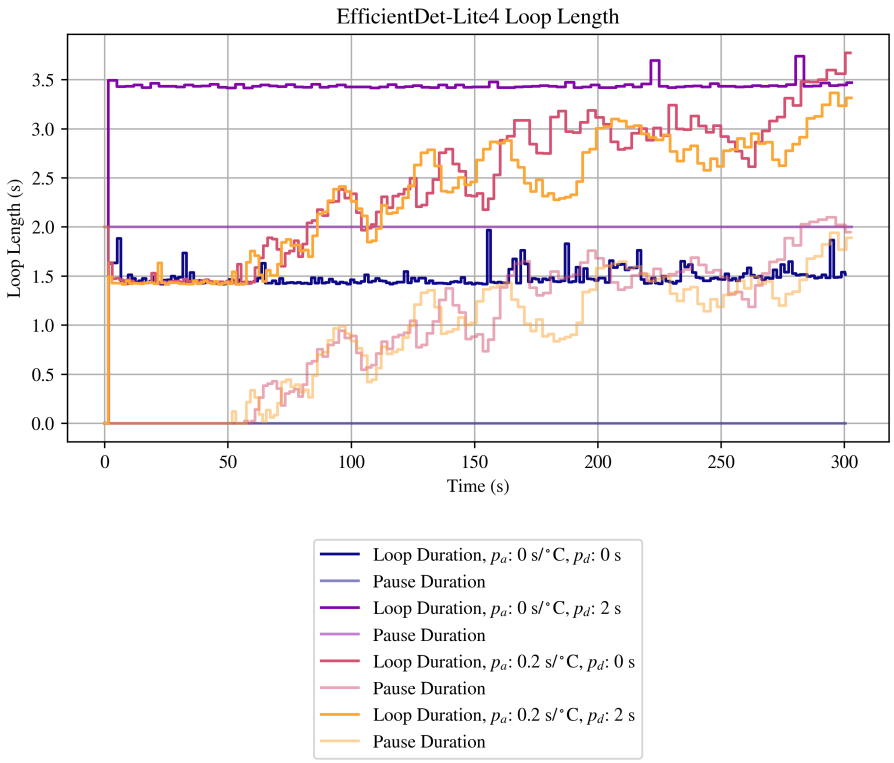


Figure 3.15: EfficientDet-Lite4 metareasoning effect on loop length on the RPi.

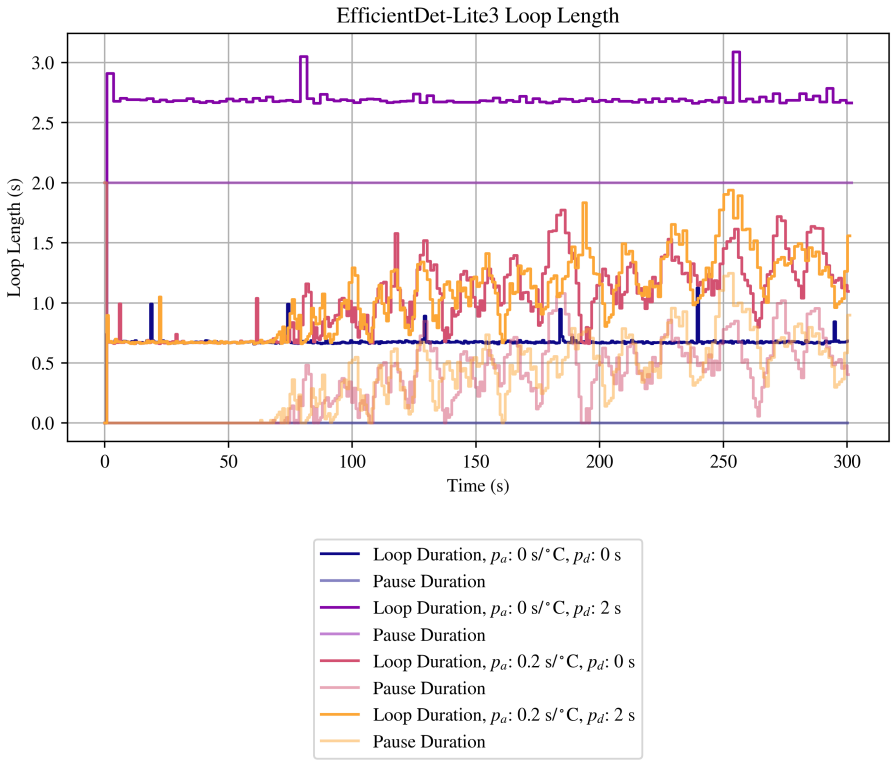


Figure 3.16: EfficientDet-Lite3 metareasoning effect on loop length on the RPi.

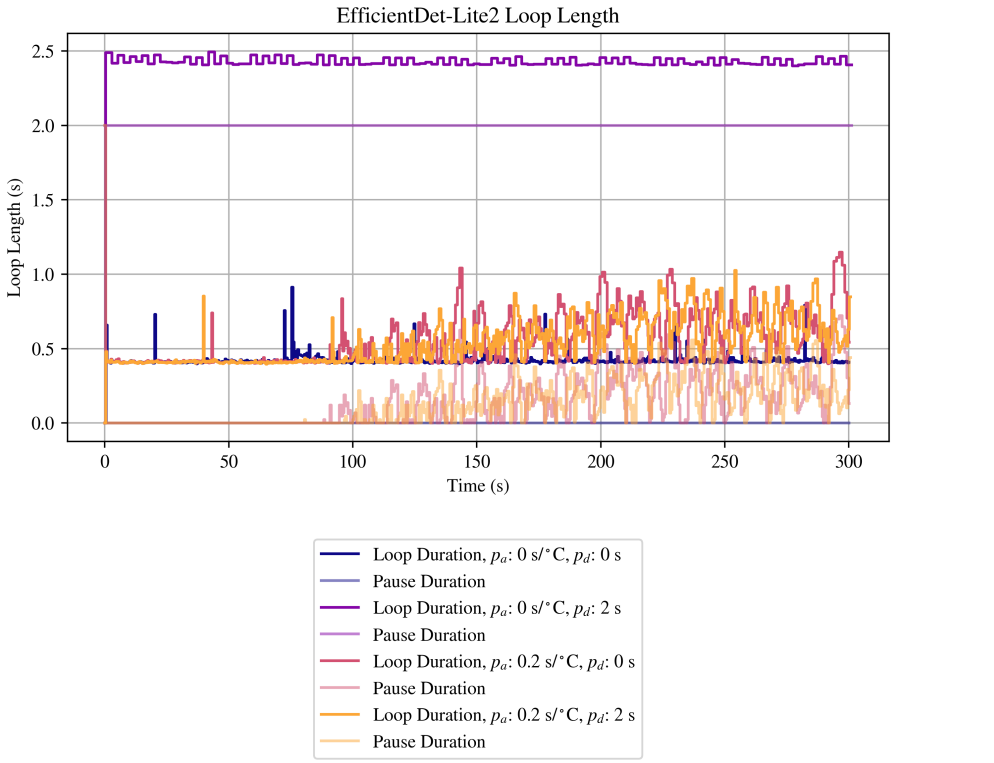


Figure 3.17: EfficientDet-Lite2 metareasoning effect on loop length on the RPi.

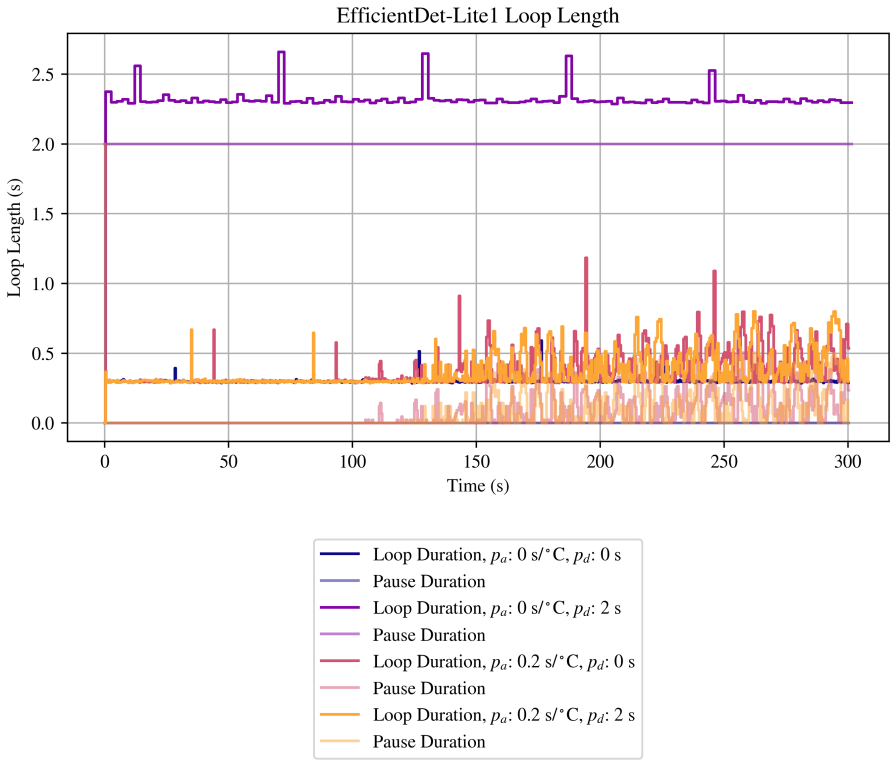


Figure 3.18: EfficientDet-Lite1 metareasoning effect on loop length on the RPi.

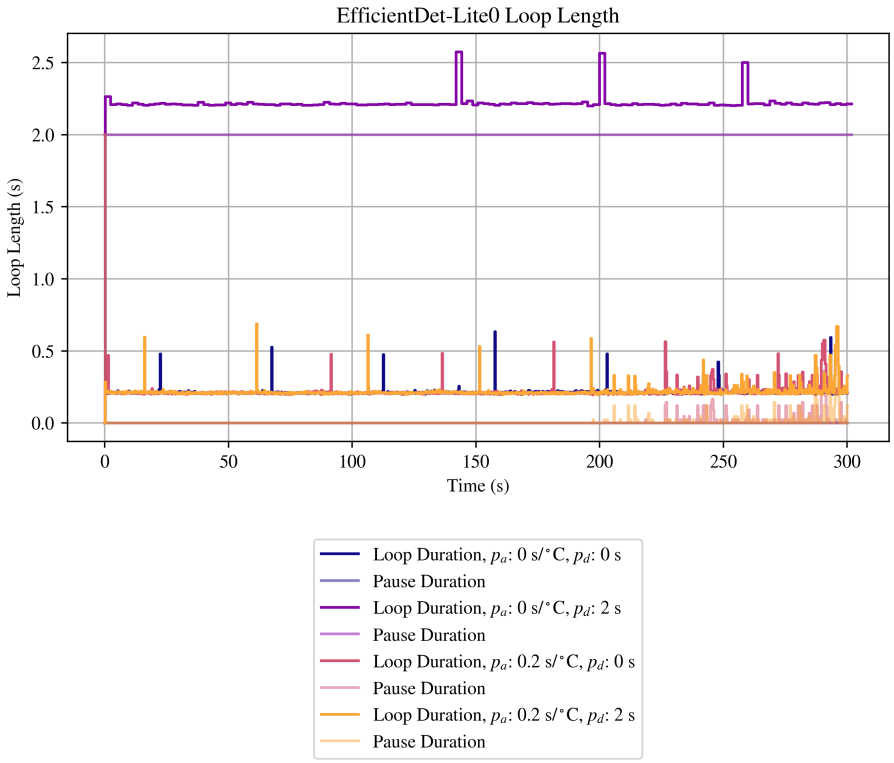


Figure 3.19: EfficientDet-Lite0 metareasoning effect on loop length on the RPi.

3.3.1.4 NVIDIA Jetson Nano Developer Kit Loop Length

The record of the program loop length over time for these tests on the Nano are shown in Figures 3.20, 3.21, 3.22, 3.23, and 3.24.

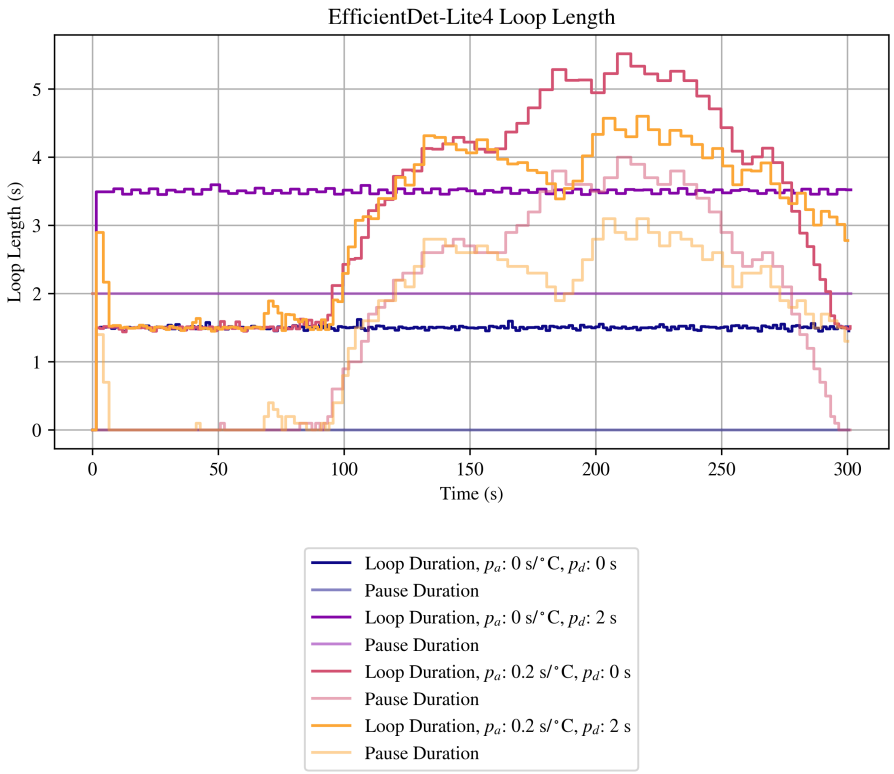


Figure 3.20: EfficientDet-Lite4 metareasoning effect on loop length on the Nano.

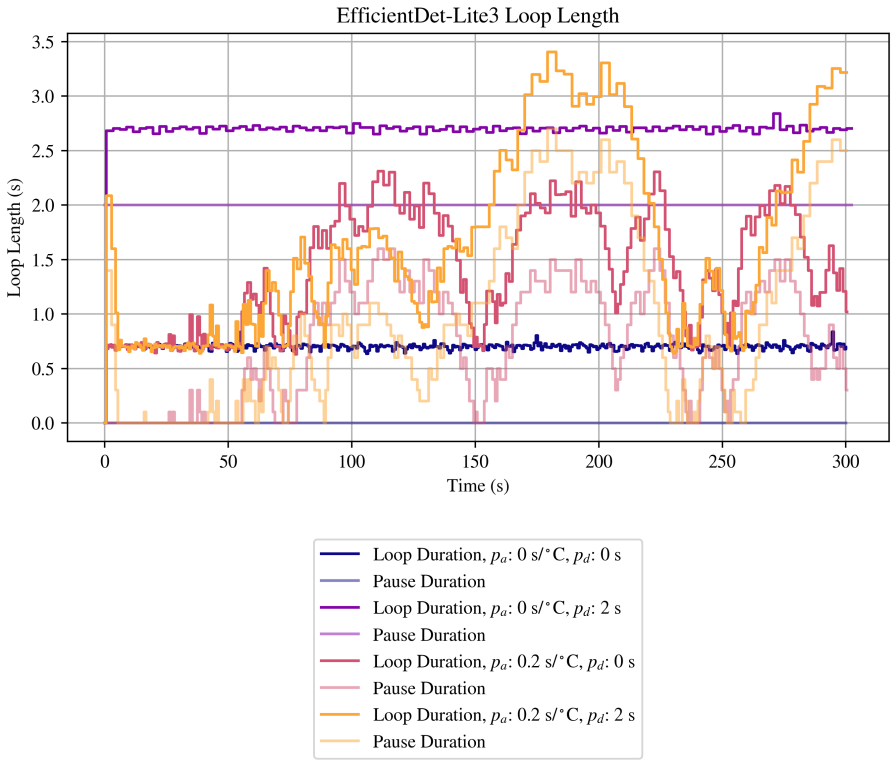


Figure 3.21: EfficientDet-Lite3 metareasoning effect on loop length on the Nano.

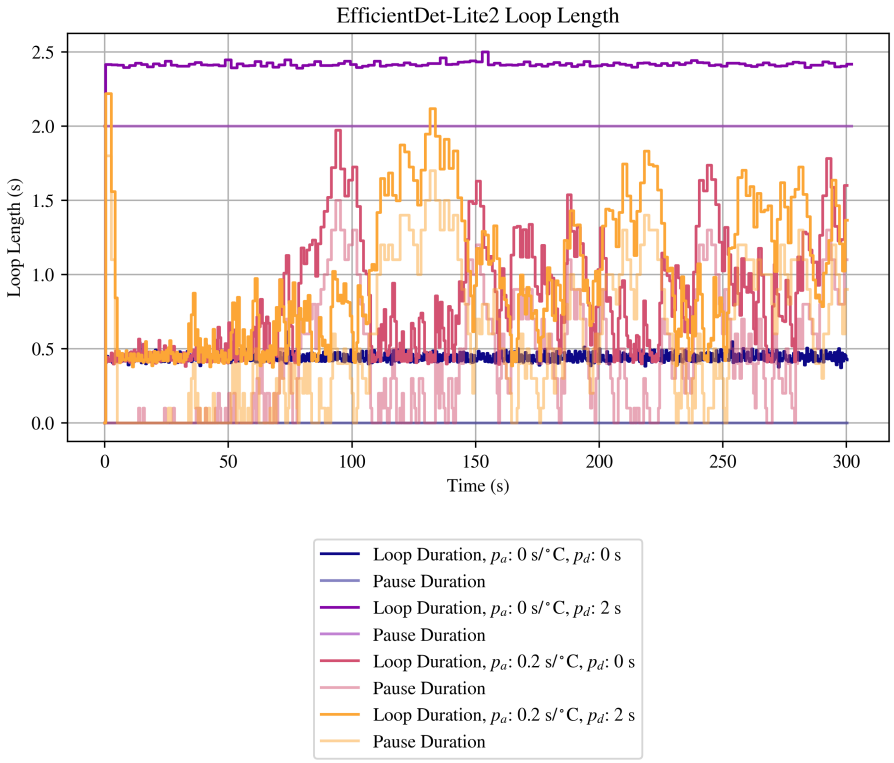


Figure 3.22: EfficientDet-Lite2 metareasoning effect on loop length on the Nano.

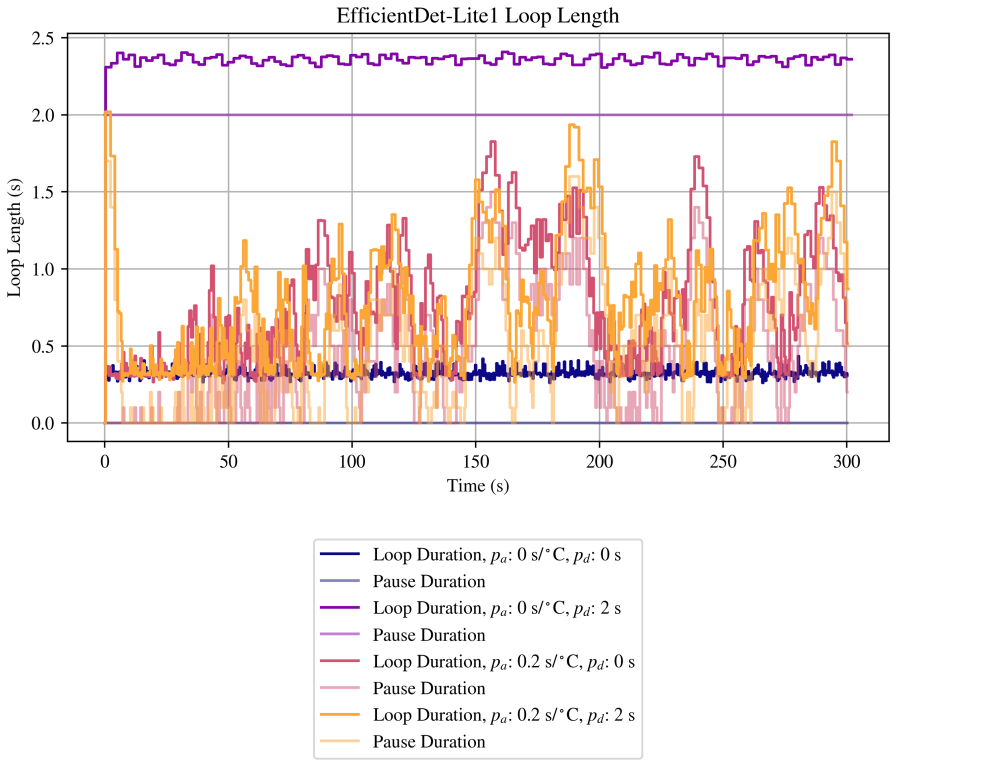


Figure 3.23: EfficientDet-Lite1 metareasoning effect on loop length on the Nano.

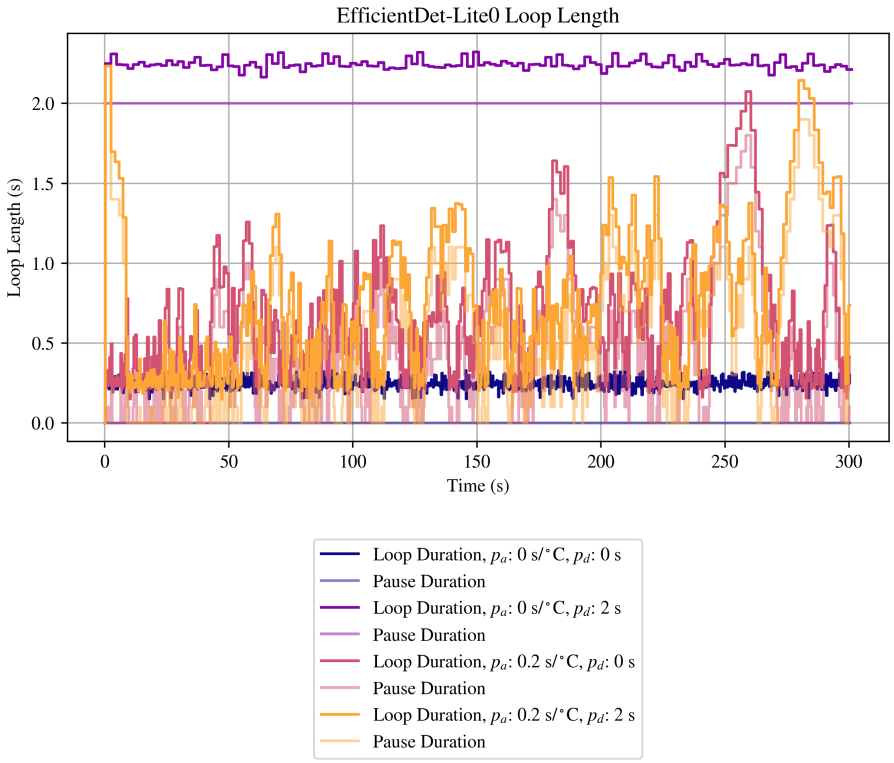


Figure 3.24: EfficientDet-Lite0 metareasoning effect on loop length on the Nano.

3.3.1.5 Raspberry Pi 4B CPU Usage

The record of the CPU usage over time for these tests on the RPi are shown in Figures 3.25, 3.26, 3.27, 3.28, and 3.29. Note that the CPU presented in these figures is a 20-second moving average of the raw CPU usage data. This is more representative of the trend in usage.

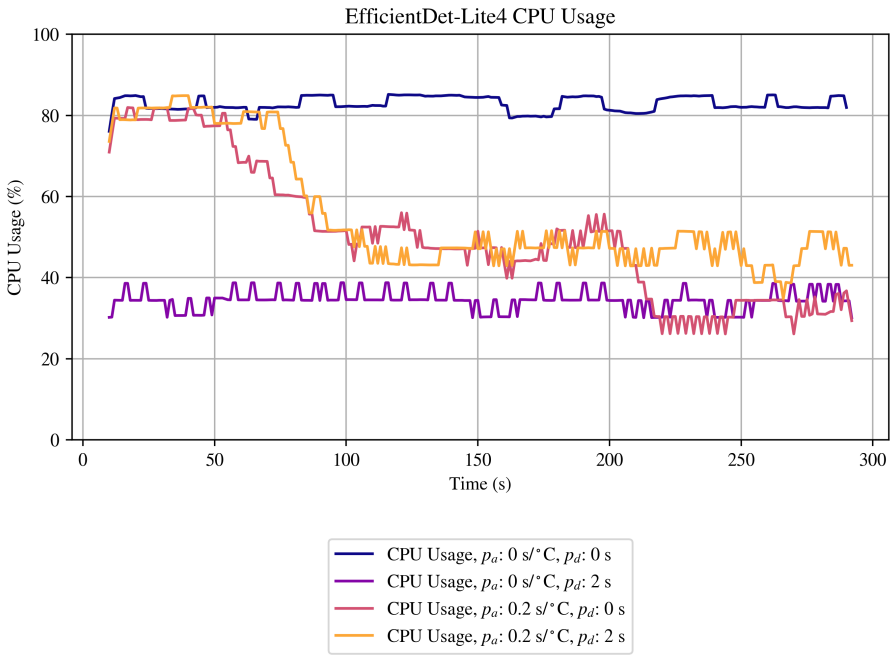


Figure 3.25: EfficientDet-Lite4 metareasoning effect on CPU usage on the RPi.

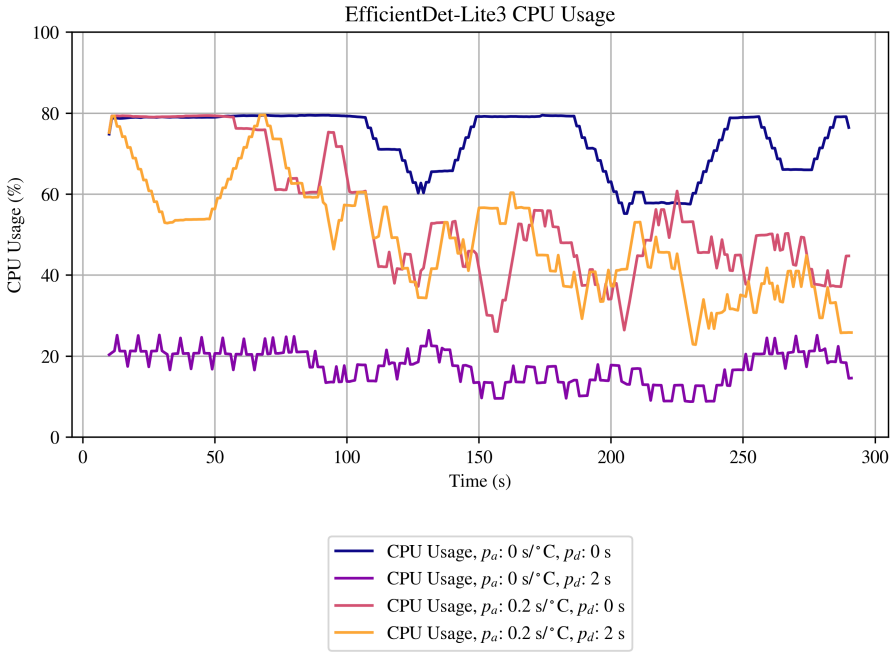


Figure 3.26: EfficientDet-Lite3 metareasoning effect on CPU usage on the RPi.

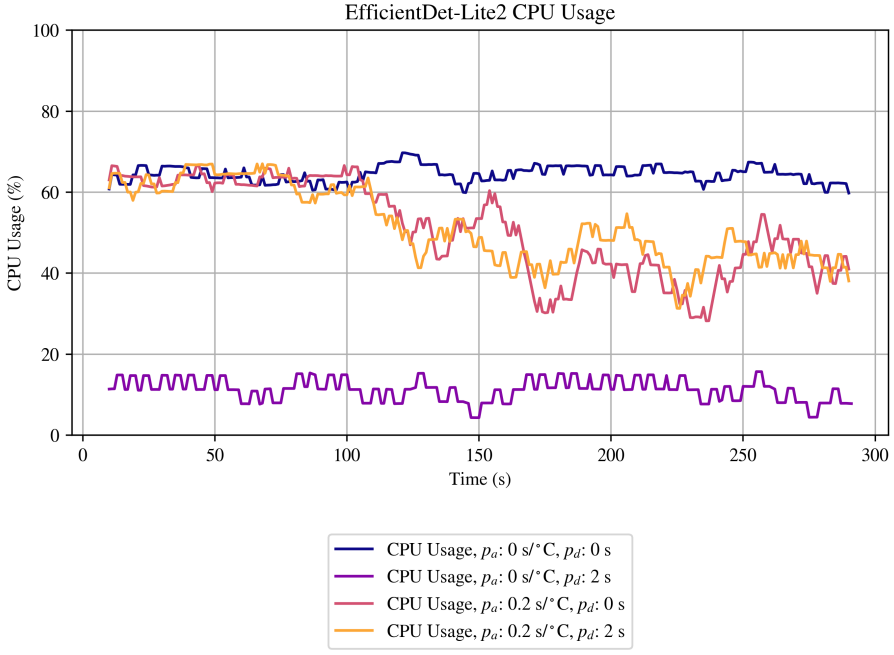


Figure 3.27: EfficientDet-Lite2 metareasoning effect on CPU usage on the RPi.

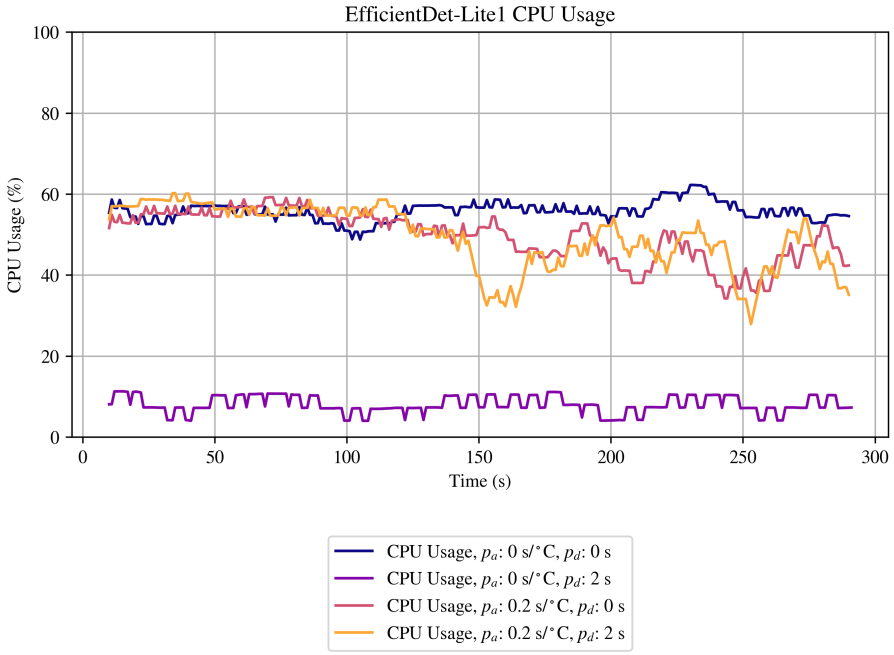


Figure 3.28: EfficientDet-Lite1 metareasoning effect on CPU usage on the RPi.

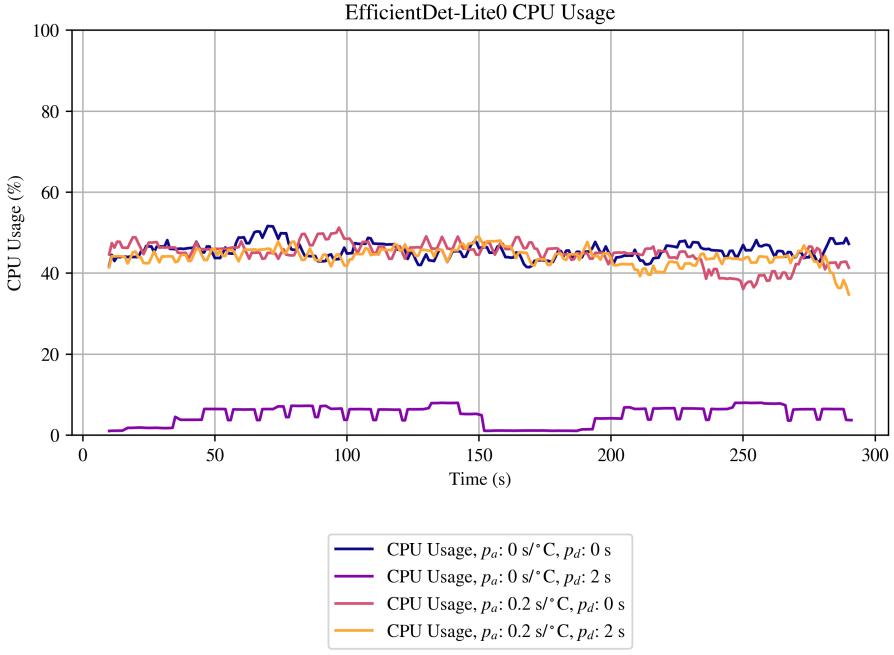


Figure 3.29: EfficientDet-Lite0 metareasoning effect on CPU usage on the RPi.

3.3.1.6 NVIDIA Jetson Nano Developer Kit CPU Usage

The record of the CPU usage over time for these tests on the Nano are shown in Figures 3.30, 3.31, 3.32, 3.33, and 3.34.

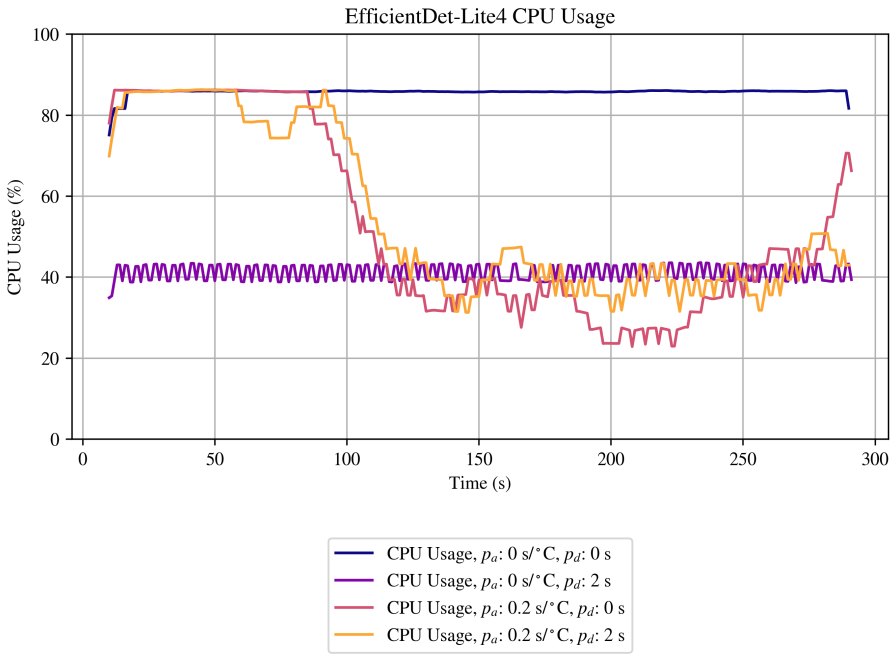


Figure 3.30: EfficientDet-Lite4 metareasoning effect on CPU usage on the Nano.

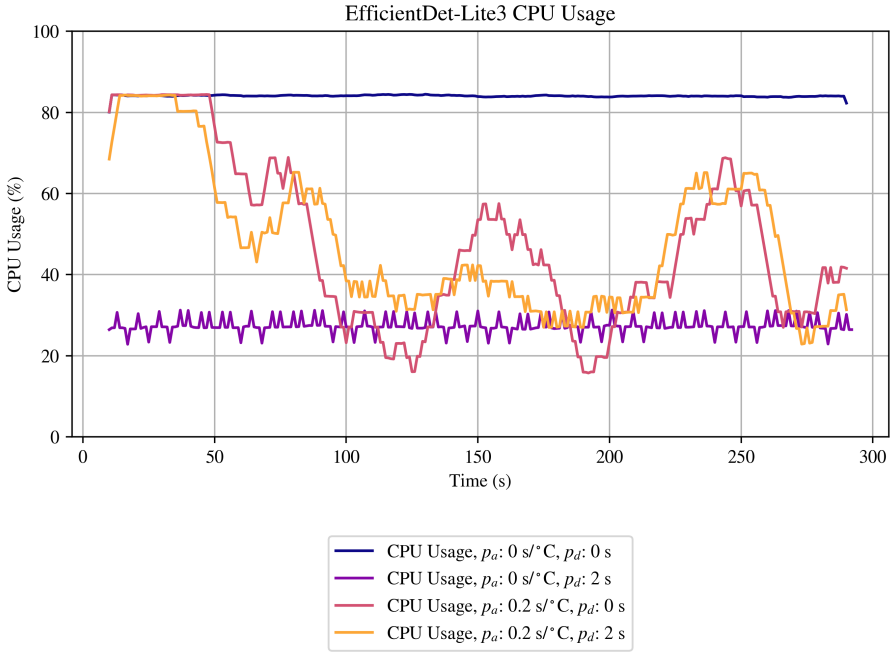


Figure 3.31: EfficientDet-Lite3 metareasoning effect on CPU usage on the Nano.

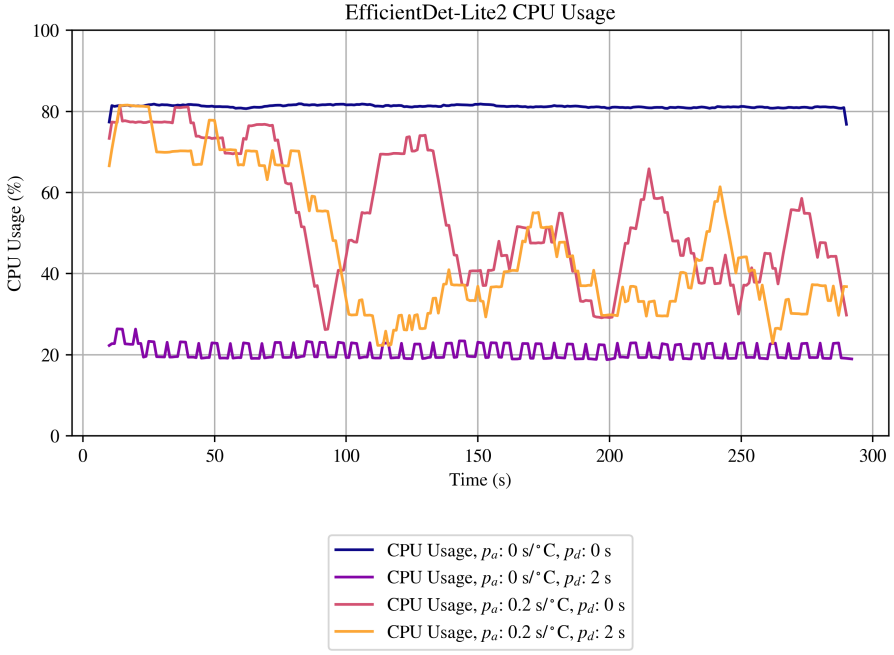


Figure 3.32: EfficientDet-Lite2 metareasoning effect on CPU usage on the Nano.

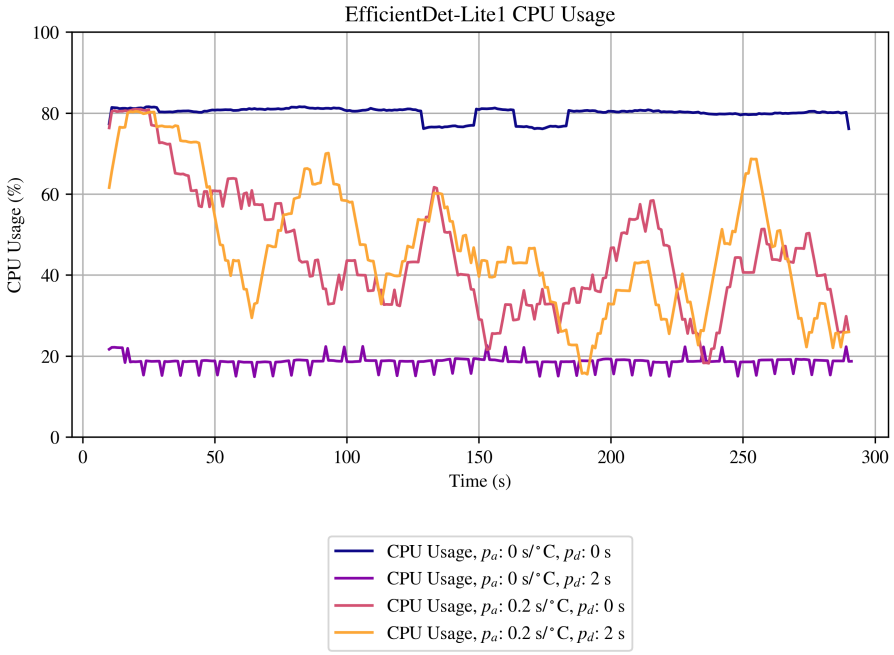


Figure 3.33: EfficientDet-Lite1 metareasoning effect on CPU usage on the Nano.

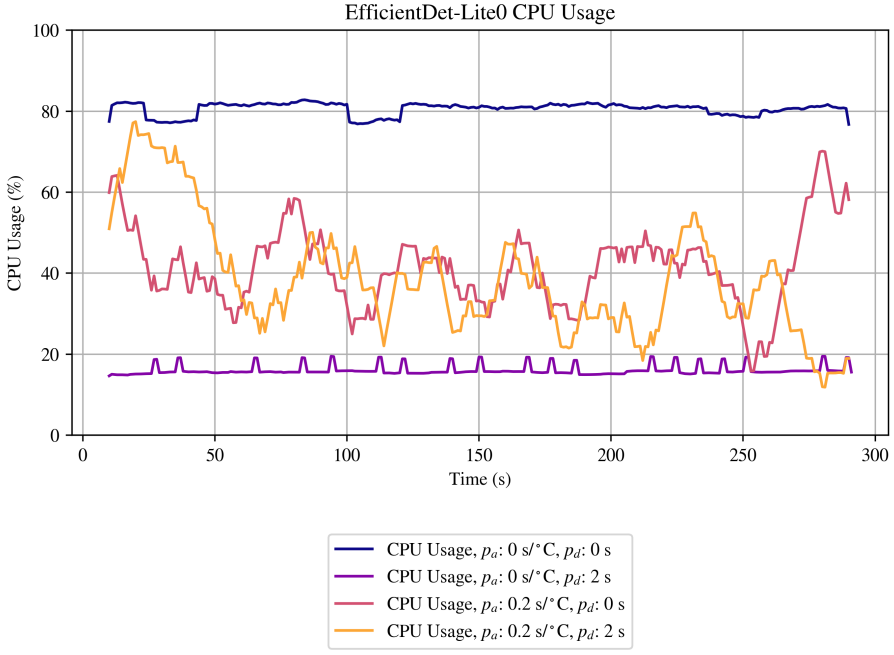


Figure 3.34: EfficientDet-Lite0 metareasoning effect on CPU usage on the Nano.

3.3.2 Network Switching Policy - Policy 2

3.3.2.1 Raspberry Pi 4B

For policy 2, temperature, loop length, and CPU usage are still important measurements. Additionally, however, network used in each loop must be tracked. Therefore, a measure of network usage frequency as a function of time is also shown. Note that each bin in this representation covers units of 10 seconds. Data overviews for policy 2 on the RPi are shown in Figures 3.35, 3.36, and 3.37.

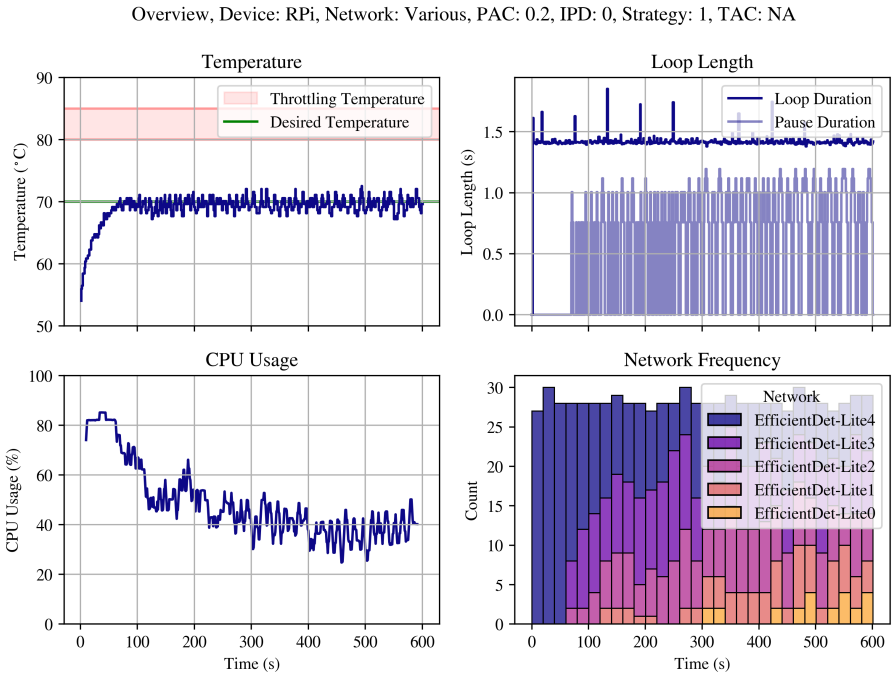


Figure 3.35: Overview of test for metareasoning policy two with switching strategy 1 on the RPi.

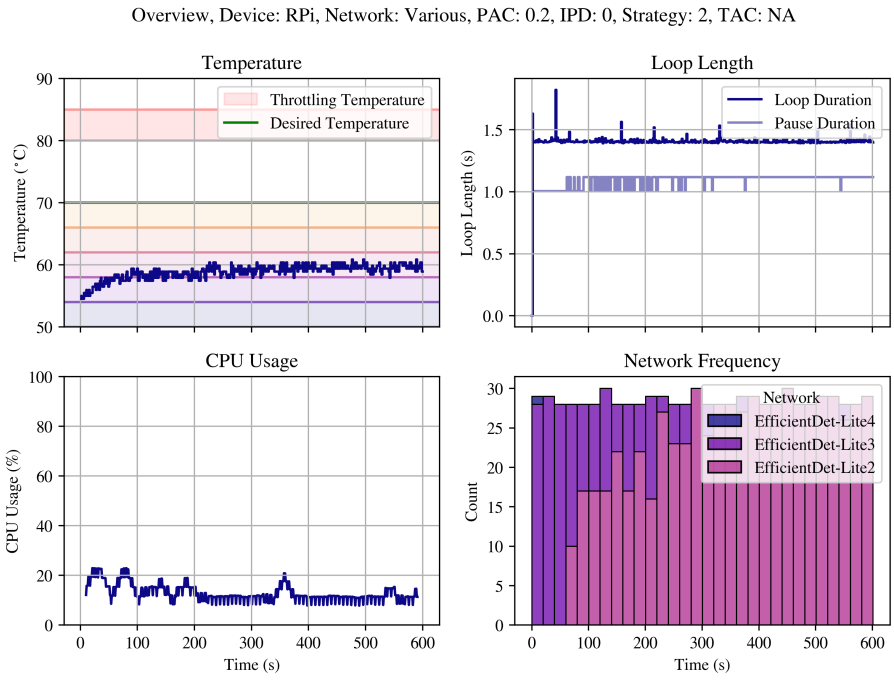


Figure 3.36: Overview of test for metareasoning policy two with switching strategy 2 on the RPi.

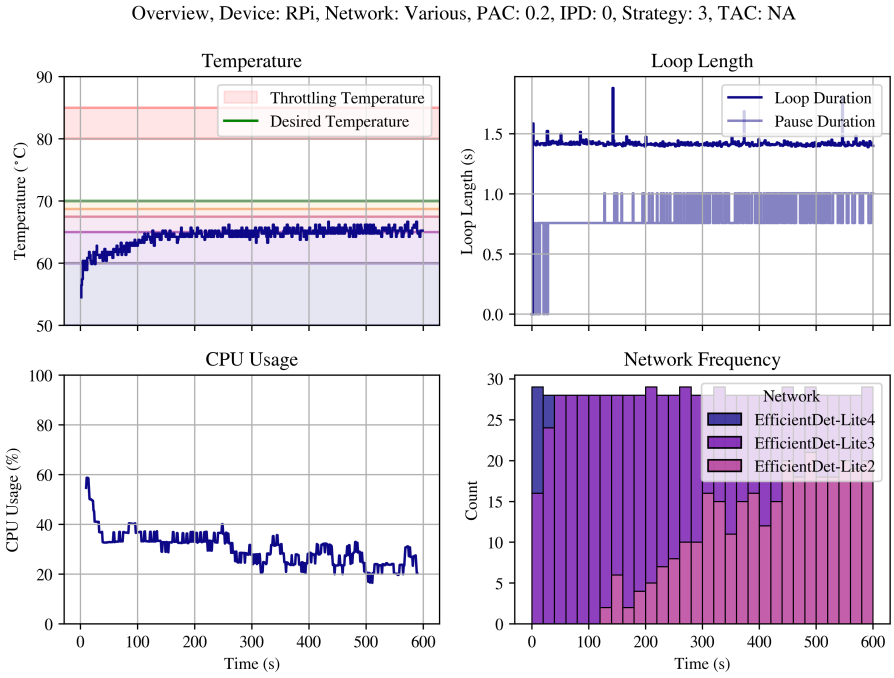


Figure 3.37: Overview of test for metareasoning policy two with switching strategy 3 on the RPi.

3.3.2.2 NVIDIA Jetson Nano Developer Kit

Data overviews for policy 2 on the Nano are shown in Figures 3.38, 3.39, and 3.40.

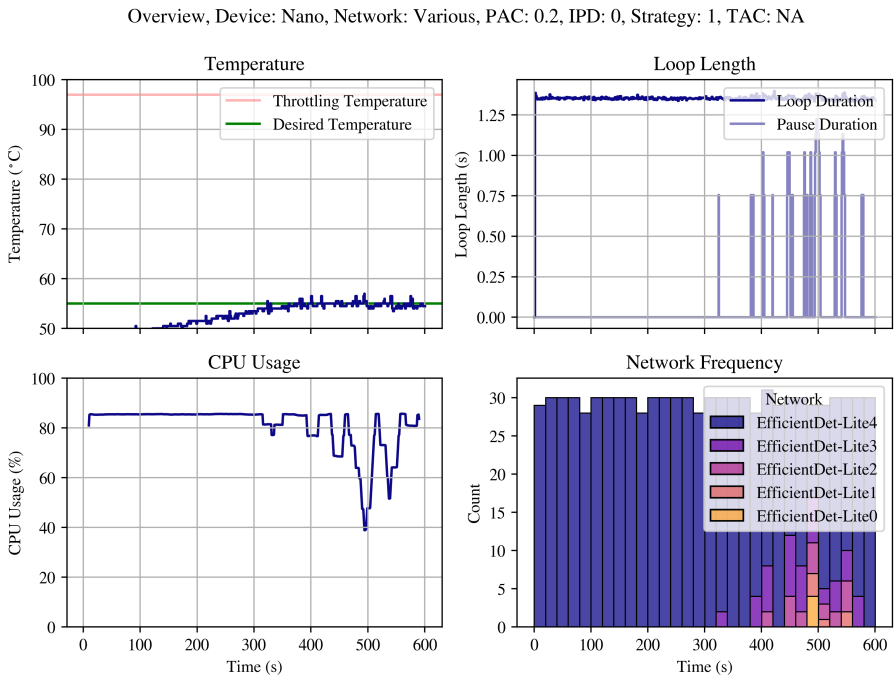


Figure 3.38: Overview of test for metareasoning policy two with switching strategy 1 on the Nano.

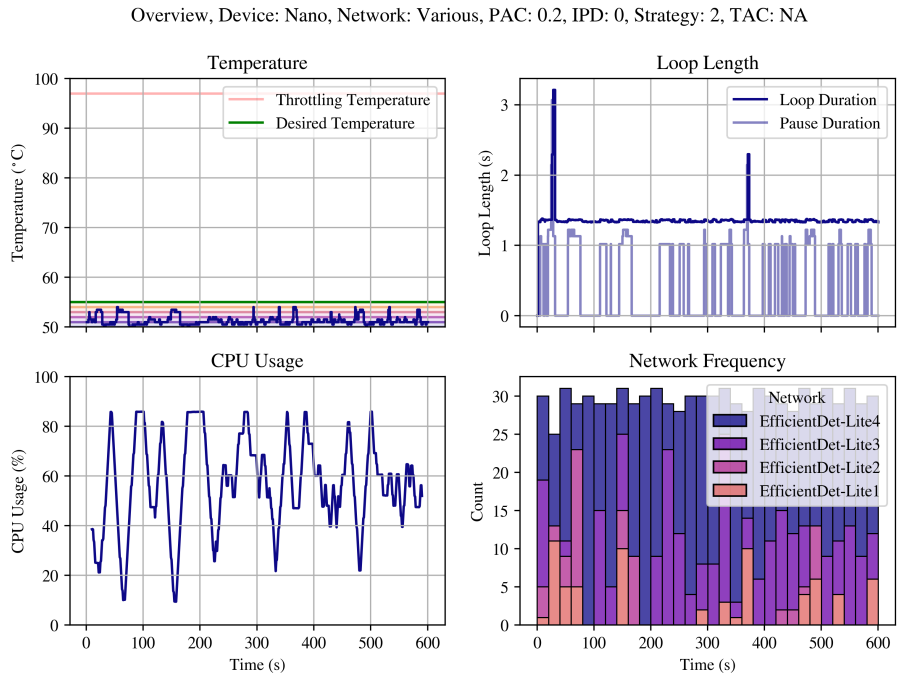


Figure 3.39: Overview of test for metareasoning policy two with switching strategy 2 on the Nano.

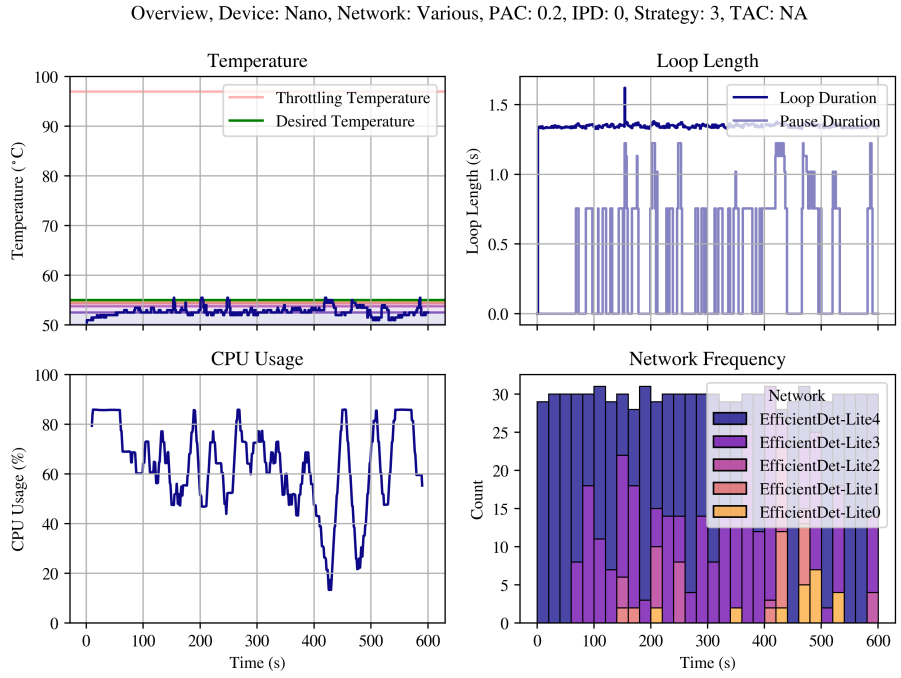


Figure 3.40: Overview of test for metareasoning policy two with switching strategy 3 on the Nano.

3.3.3 Hybrid Policy - Policy 3

3.3.3.1 Raspberry Pi 4B

Data overviews for policy 3 and switching strategy 1 on the RPi are shown in Figures 3.41, 3.42, and 3.43. Overviews for policy 3 and switching strategy 2 and shown in Figures 3.44, 3.45, and 3.46. Overviews for policy 3 and switching strategy 3 on the RPi are shown in Figures 3.47, 3.48, and 3.49.

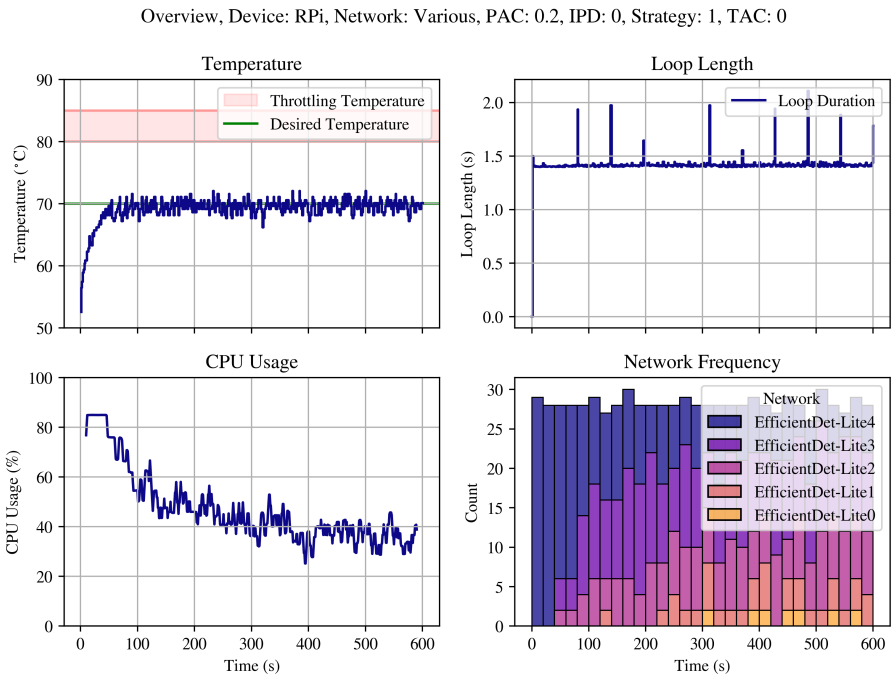


Figure 3.41: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 0 on the RPi.

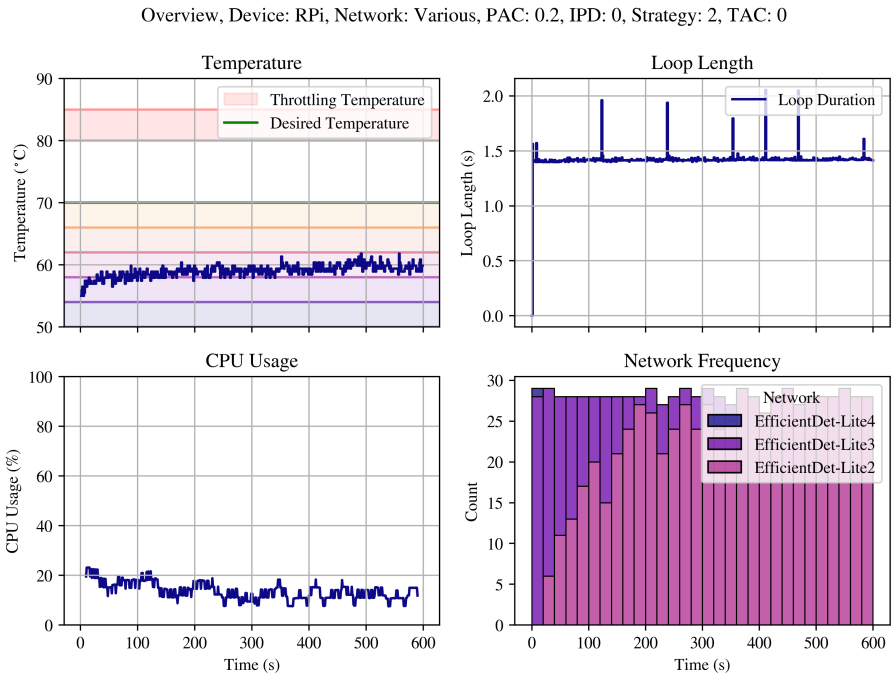


Figure 3.42: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 0 on the RPi.

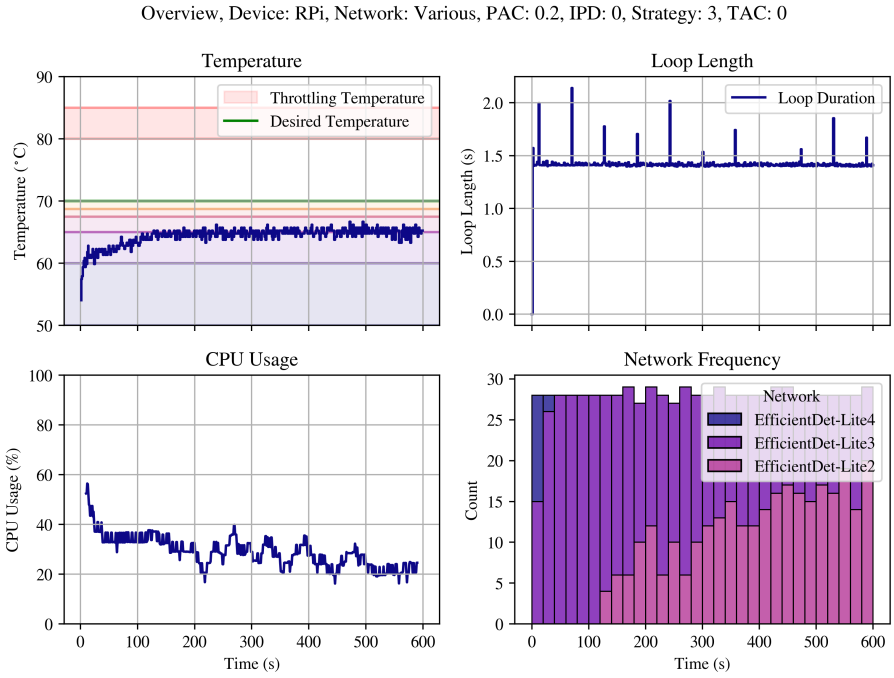


Figure 3.43: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 0 on the RPi.

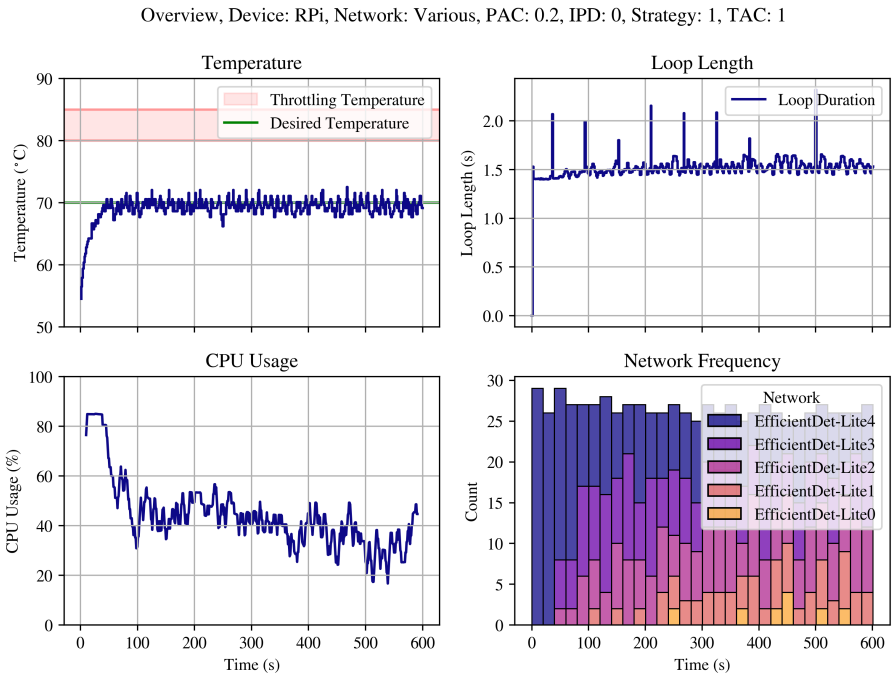


Figure 3.44: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 1 on the RPi.

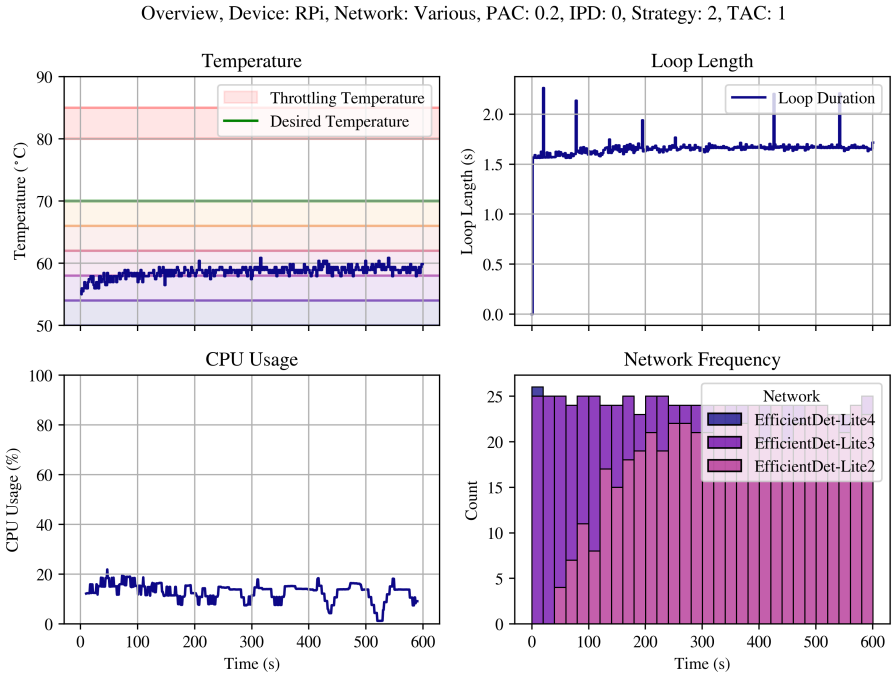


Figure 3.45: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 1 on the RPi.

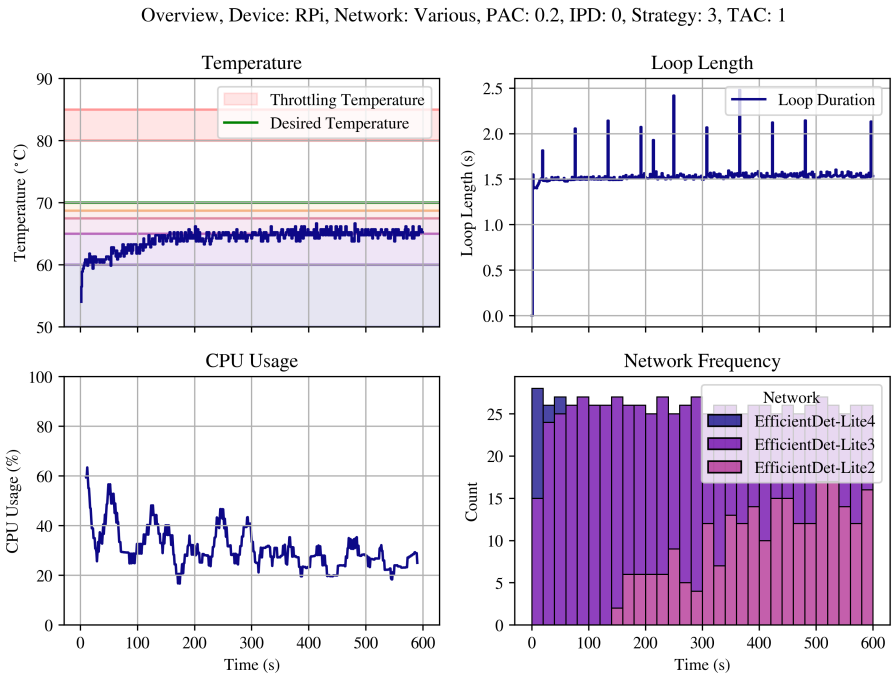


Figure 3.46: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 1 on the RPi.

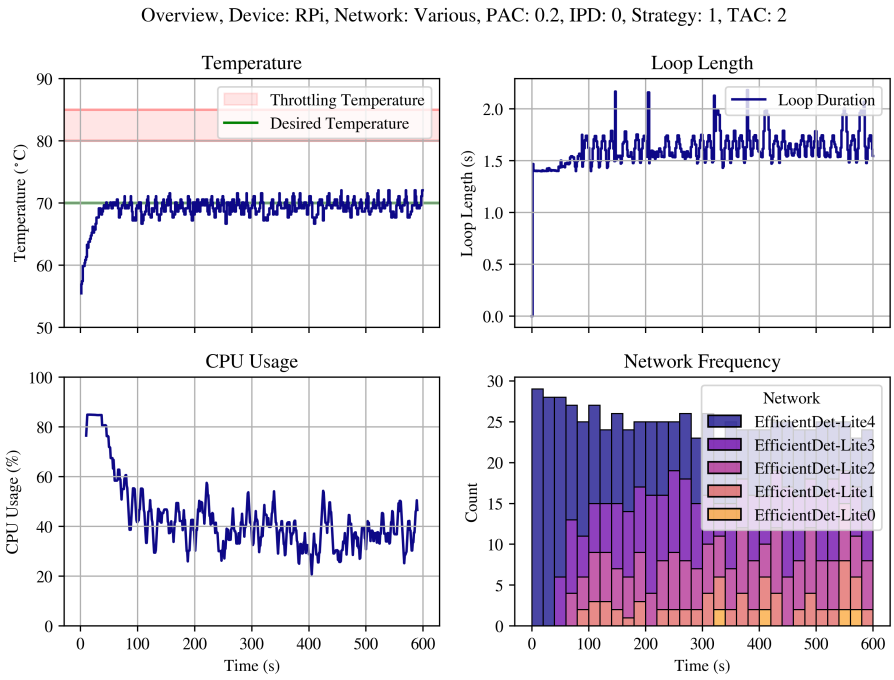


Figure 3.47: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 2 on the RPi.

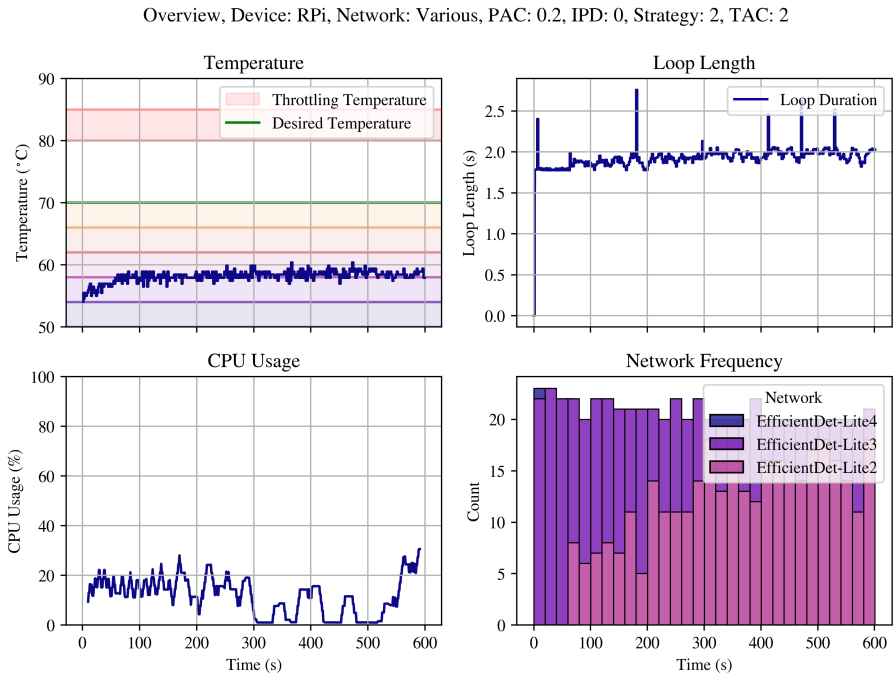


Figure 3.48: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 2 on the RPi.

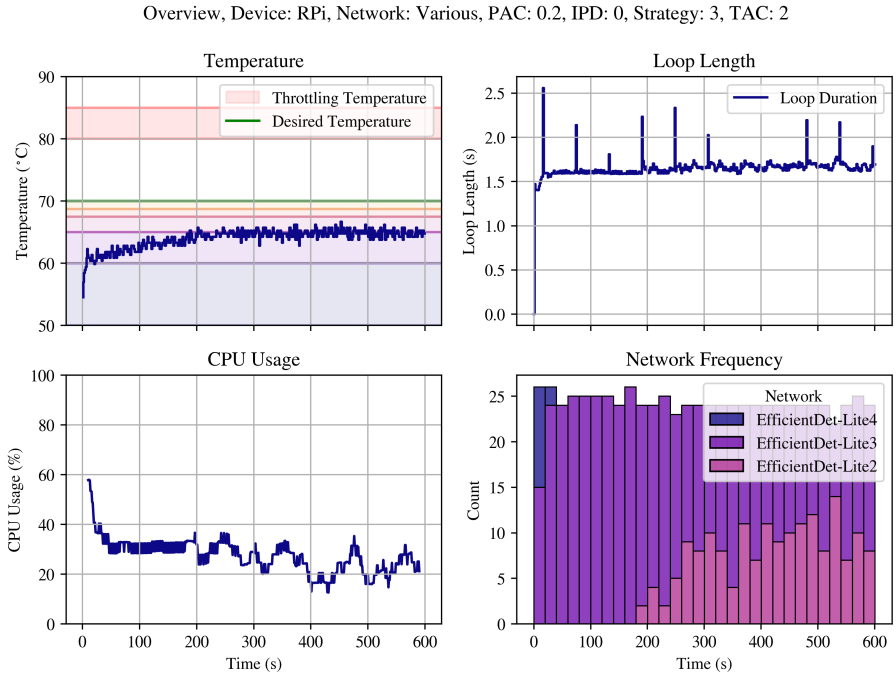


Figure 3.49: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 2 on the RPi.

3.3.3.2 NVIDIA Jetson Nano Developer Kit

Data overviews for policy 3 and switching strategy 1 on the Nano are shown in Figures 3.50, 3.51, and 3.52. Overviews for policy 3 and switching strategy 2 on the Nano are shown in Figures 3.53, 3.54, and 3.55. Overviews for policy 3 and switching strategy 3 are shown in Figures 3.56, 3.57, and 3.58.

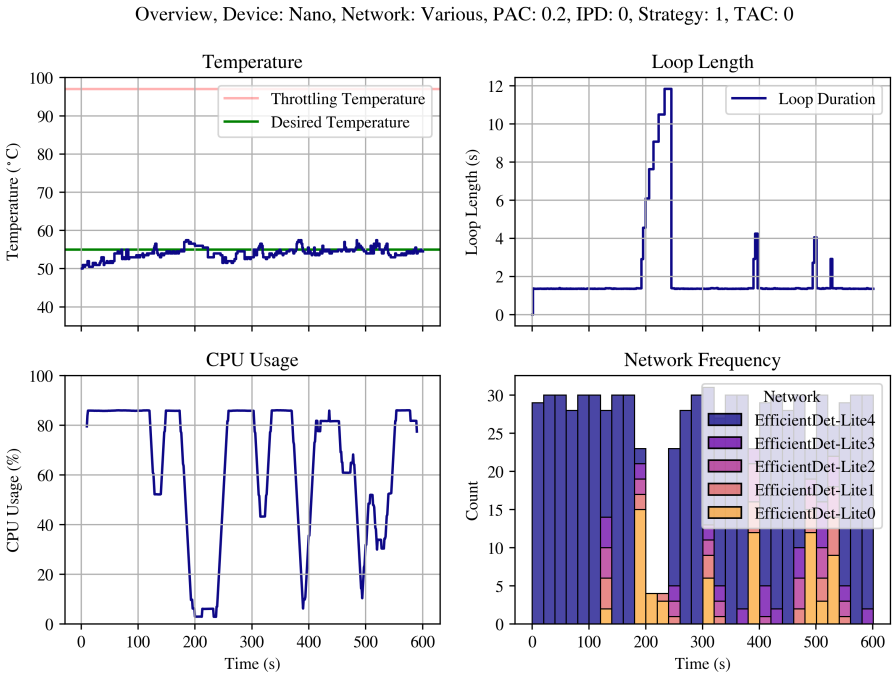


Figure 3.50: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 0 on the Nano.

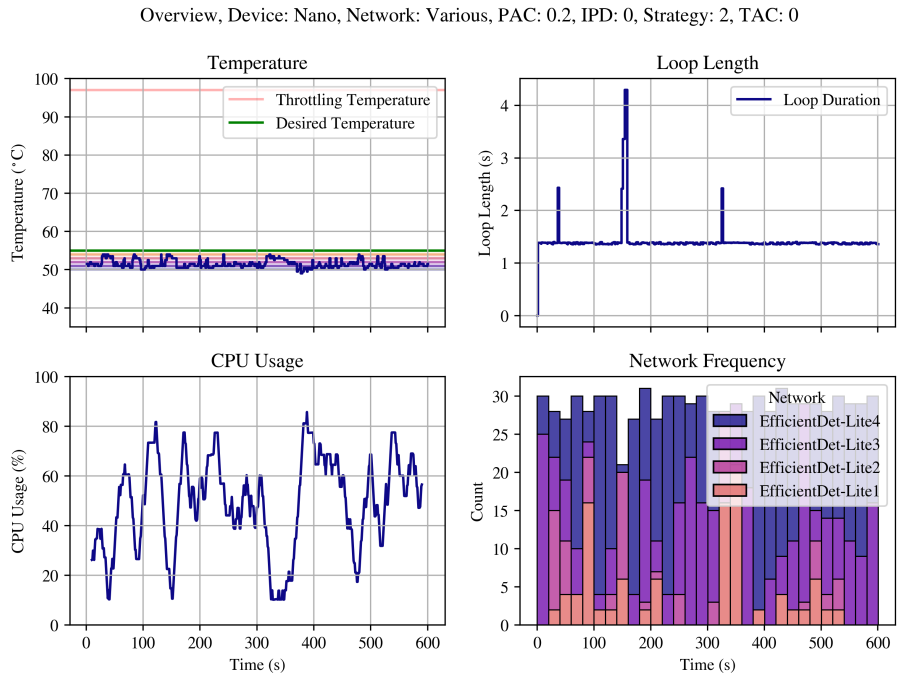


Figure 3.51: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 0 on the Nano.

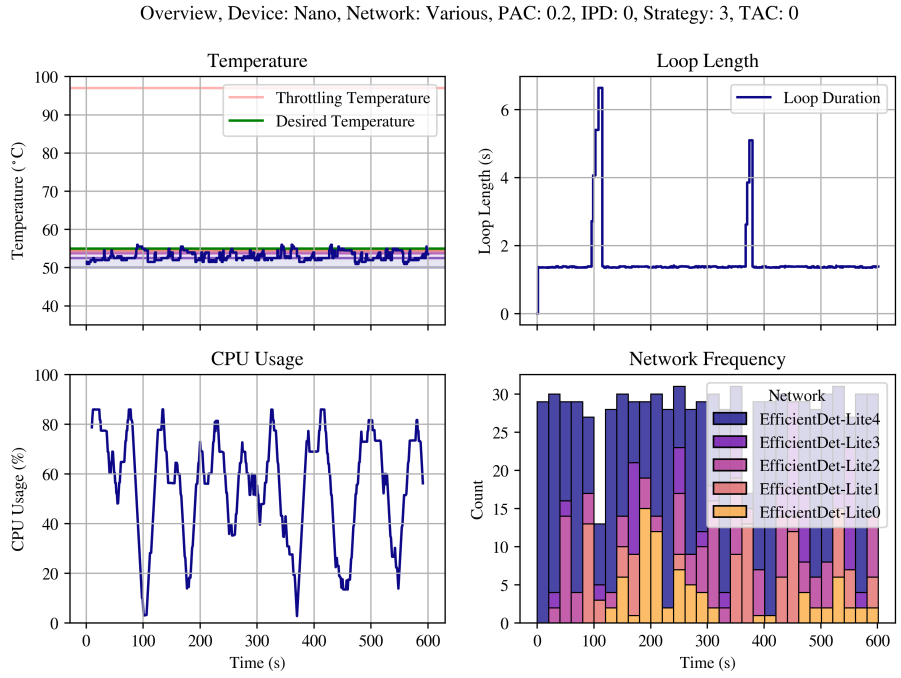


Figure 3.52: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 0 on the Nano.

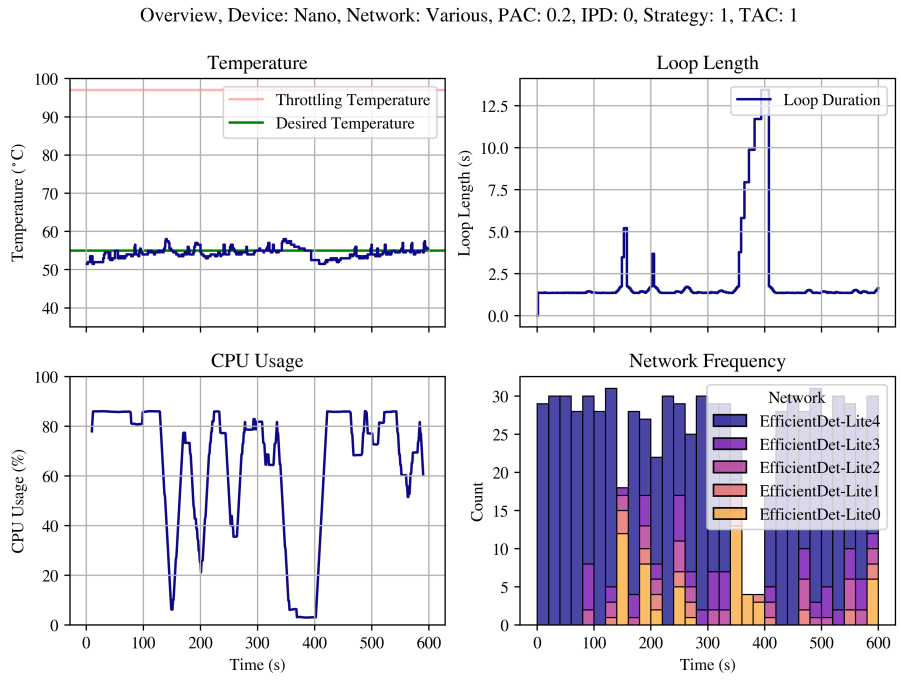


Figure 3.53: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 1 on the Nano.

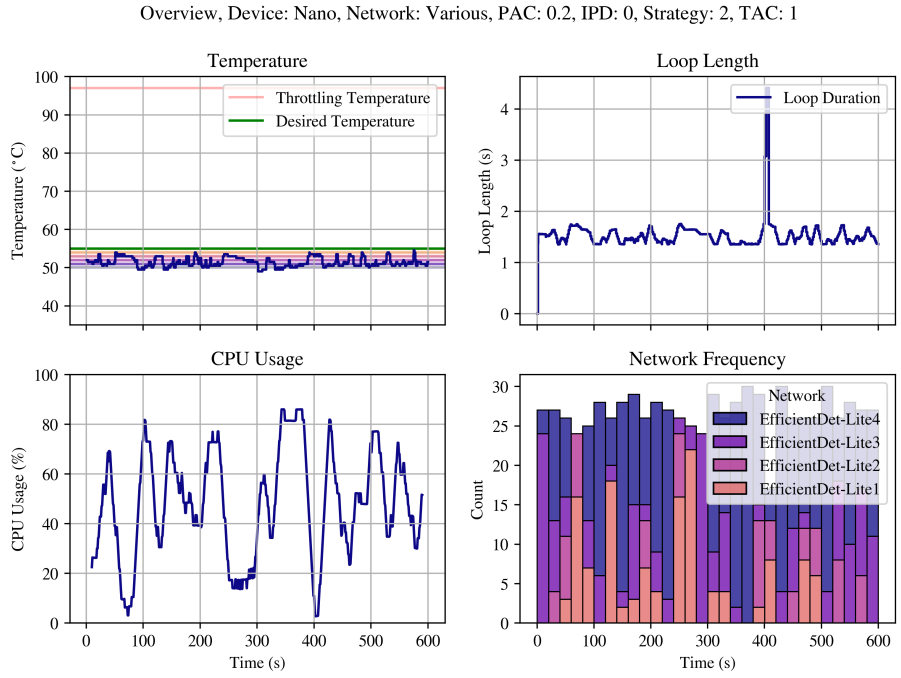


Figure 3.54: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 1 on the Nano.

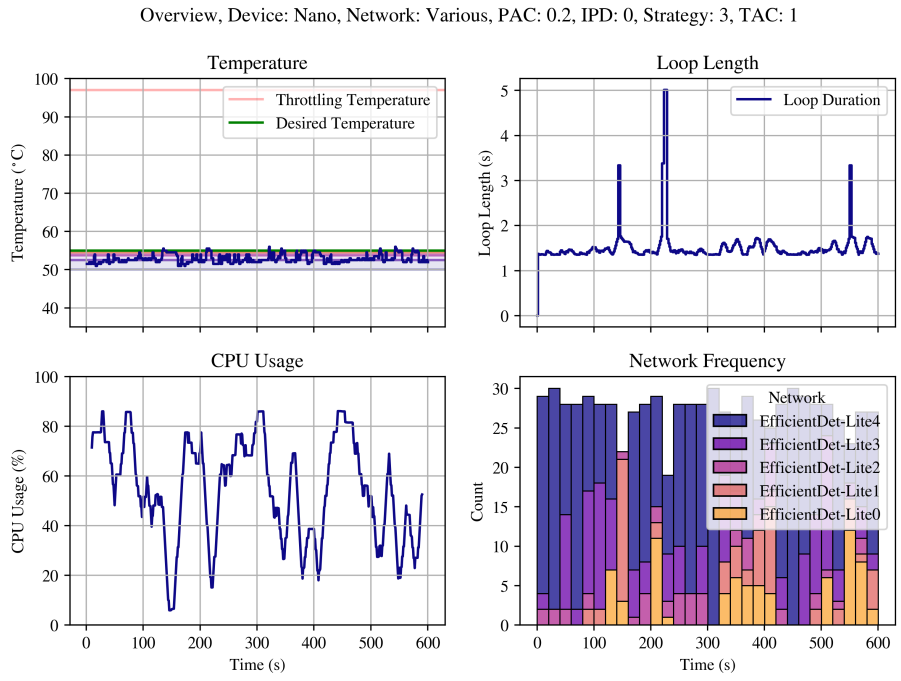


Figure 3.55: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 1 on the Nano.

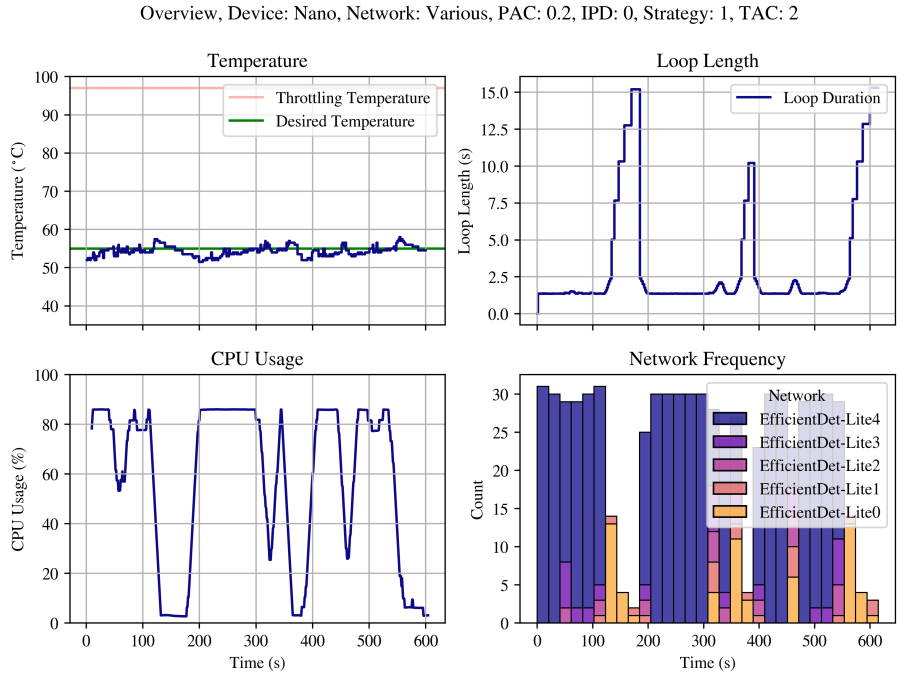


Figure 3.56: Overview of test for metareasoning policy 3 with switching strategy 1 and a TAC of 2 on the Nano.

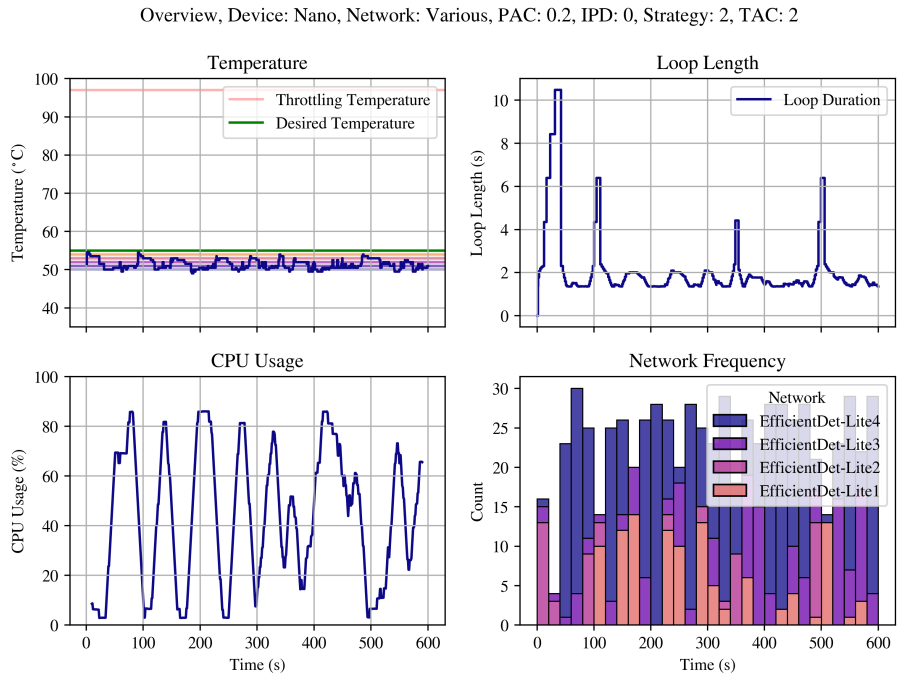


Figure 3.57: Overview of test for metareasoning policy 3 with switching strategy 2 and a TAC of 2 on the Nano.

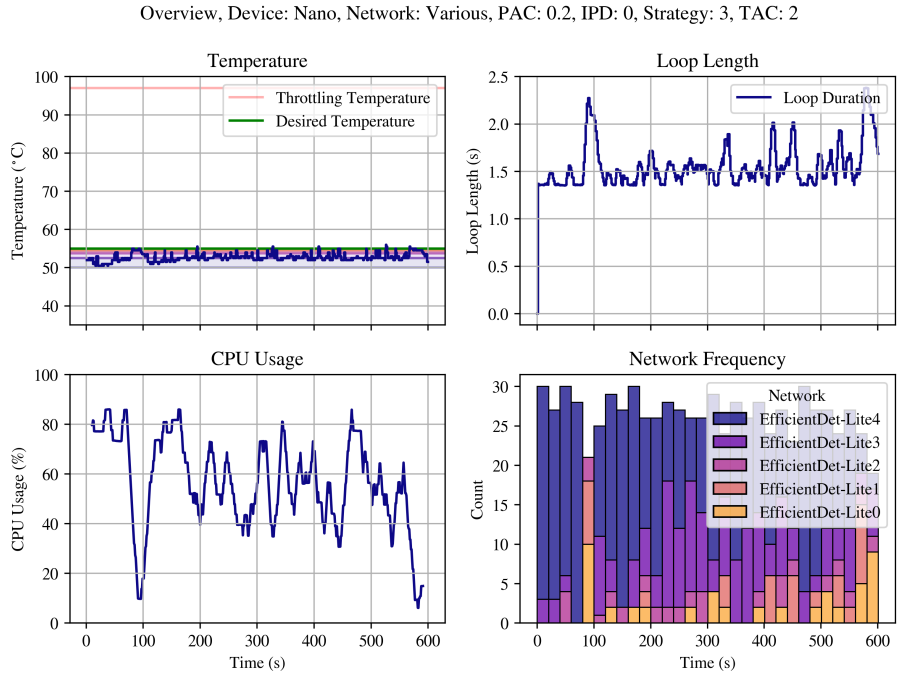


Figure 3.58: Overview of test for metareasoning policy 3 with switching strategy 3 and a TAC of 2 on the Nano.

3.4 Discussion

The results for policy 1 demonstrate the successful control of device temperature via metareasoning. In each figure, the control test increases beyond the desired temperature, the static pause length test remains at a temperature below the desired temperature, while tests with a non-zero pause adjustment coefficient stabilize the desired temperature regardless of the initial pause duration.

The mean squared temperature error (MSTE) on the RPi, that is the squared difference squared between the current temperature and desired temperature, is shown in Figures 3.59, 3.60, and 3.61. In these figures, the mean squared temperature error is plotted as a function of the value of p_a and p_d for each test.

The control test, where metareasoning is inactive, for each set of tests in policy 1 corresponds to $p_a = 0s/°C$ and $p_d = 0s$. It is notable that, in these tests, the MSTE is highly dependent on the network being used. Notice in Figure 3.59 the control error of approximately 60 °C, whereas the control error is approximately 25 °C in Figure 3.61. This is likely due to the trend that, despite using the same portion of the CPU, more "complex" networks tend to produce more heat than "simpler" networks.

Similarly, for tests where $p_a = 0s/°C$, the MSTE increases as p_d incerases. This increase is accelerated for "simple" networks. EDL-0 shows an increase in MSTE from 25 °C to 225 °C over a range of $0s \leq p_d \leq 2s$, while EDL-2 shows an increase from 25 °C to 180 °C over the same p_d range.

Especially notable is the effect that p_a has on the MSTE. For all tests where $p_a > 0s/°C$, the average temperature error remains between 10 °C and 30 °C, depending on

the network. This is true regardless of the initial pause duration, p_d.

Similar trends are seen on the Nano, shown in Figures 3.62, 3.63, and 3.64.

Figure 3.59: EfficientDet-Lite4 and EfficientDet-Lite3 temperature error trend as a function of p_a and p_d on the RPi.

Figure 3.60: EfficientDet-Lite2 and EfficientDet-Lite1 temperature error trend as a function of p_a and p_d on the RPi.

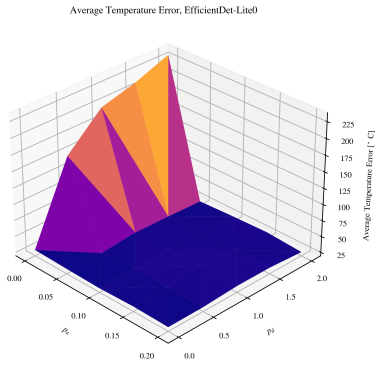
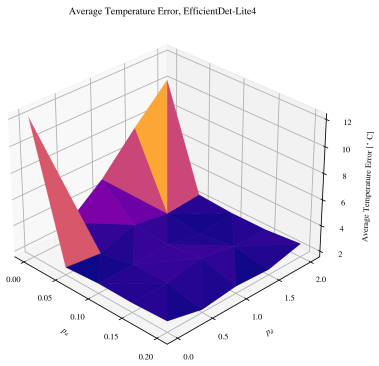


Figure 3.61: EfficientDet-Lite0 temperature error trend as a function of p_a and p_d on the RPi.



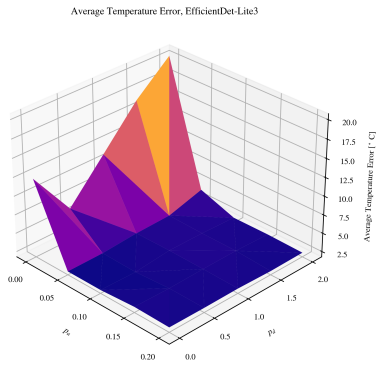
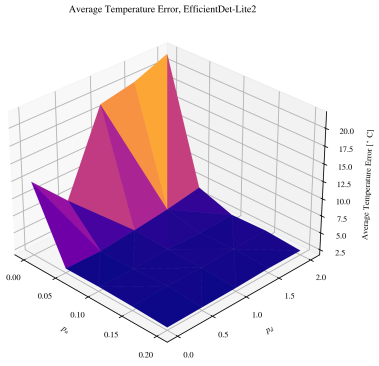


Figure 3.62: EfficientDet-Lite4 and EfficientDet-3 temperature error trend as a function of p_a and p_d on the Nano.



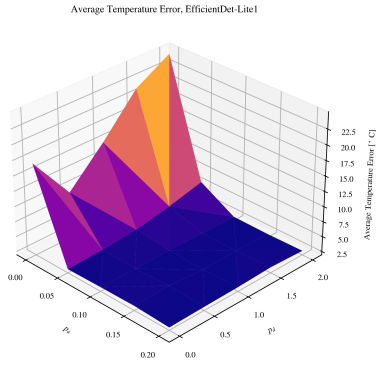


Figure 3.63: EfficientDet-Lite2 and EfficientDet-Lite1 temperature error trend as a function of p_a and p_d on the Nano.

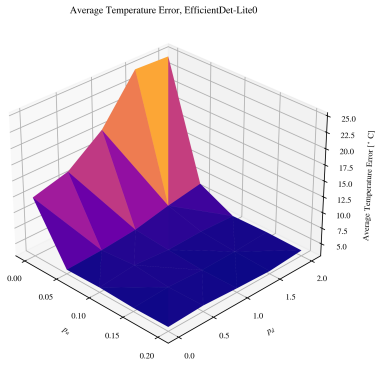


Figure 3.64: EfficientDet-Lite0 temperature error trend as a function of p_a and p_d on the Nano.

The results for policy 2 show that a constant throughput is successfully maintained over the course of the tests for each switching strategy on both the RPi and the Nano. Note that there are occasional spikes in the loop length, the inverse of the throughput. These likely correspond to periodic background tasks performed by the Raspberry Pi which request additional CPU usage, therefore increasing the processing duration. The program is running on the Raspbian operating system, which is additionally tasked with communicating with remote viewers via the RealVNC (Virtual Network Computing) software, for example.

While switching strategy 1 optimizes the image processing loop to use maximum precision for as long as possible, seen in Figure 3.35, strategies 2 and 3 operate at lower temperatures for a longer duration, sacrificing a surplus of detection precision for lower average temperature, seen in Figures 3.36 and 3.36. Switching strategy 3 more closely approaches the success of switching strategy 1.

Results for policy 2 are similar on the Nano, with switching strategy 1 successfully optimizing precision while constrained by a minimum throughput and maximum temperature, shown in Figure 3.38. Again, spikes in the program loop length can be seen in the Nano, likely caused by similarly periodic background tasks.

The results for policy 3 show that the strategy is successful in generalizing policy 2. For example, Figures 3.41, 3.42, and 3.43, where $TAC = 0$, successfully mimic metareasoning policy 2. For tests where $TAC = 1$, results are similar to that of $TAC = 0$ but the throughput slowly increases over time, representing the 1:1 sacrifice of precision to throughput. This is seen in Figures 3.44, 3.45, and 3.46. Test results where $TAC = 2$ show an even larger throughput loss than when $TAC = 1$, as expected. These results are

shown in Figures 3.47, 3.48, and 3.49.

Testing on the Nano, showed similar success as on the RPi, but note that for all switching strategies, there are occasional climbs in loop length when precision reaches its minimum. This corresponds to a contingency that if the average expected detection precision is at the minimum possible, policy 1 augments the pause duration created by policy 3. Therefore, when the minimum precision is reached for 5 consecutive program loops the pause length begins to increase beyond what policy 3 is capable of creating. This is clearly visible in Figures 3.50, 3.51, 3.52, 3.53, 3.56, 3.57, and 3.58.

Figures 3.65, 3.66, and 3.67 show the precision vs throughput graphs for each value of TAC and for each switching strategy on the RPi. Figures 3.68, 3.69, and 3.70 show the same for the Nano.

Note that for strategy 2 on the RPi, EDL-4 was only used once before the temperature entered the next thermal zone, therefore, the average precision remains relatively low throughout the test. This is clearly visible in Figure 3.66. Also, for each strategy on the Raspberry Pi, the occasional spikes in the loop length are apparent in Figures 3.65, 3.66, and 3.67.

Policy 3 has a notable caveat in its parameter assignment. As seen in Algorithm 7, the desired loop length is assigned the inverse of the desired throughput, which itself is assigned by subtracting the product of the precision loss and TAC from the maximum throughput. With high TACs, it is therefore possible to obtain a negative desired throughput and a corresponding loop length with negative duration. The pause length is then assigned a negative value which, to prevent the breaking of the loop, is set to zero. Therefore, the TAC parameter for policy 3 must be tuned to a value such that when the precision reaches

its minimum the desired throughput is a positive, if small value.

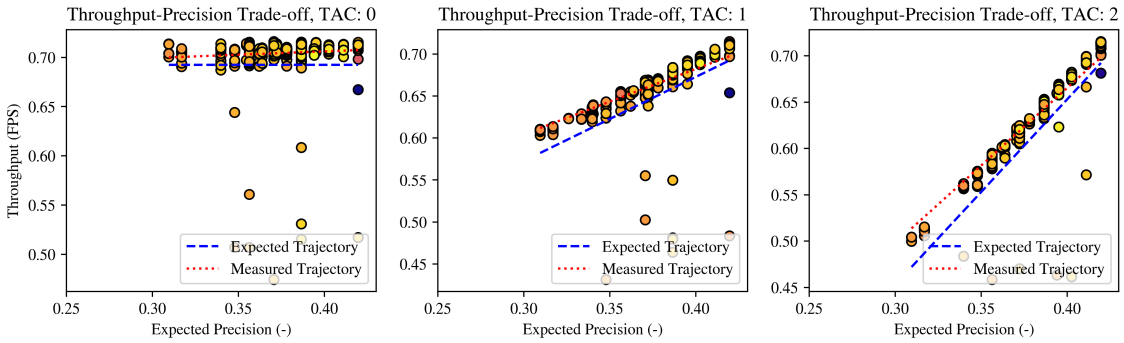


Figure 3.65: Throughput vs precision for policy 3, strategy 1 on the RPi. Lighter colors are associated with higher temperatures.

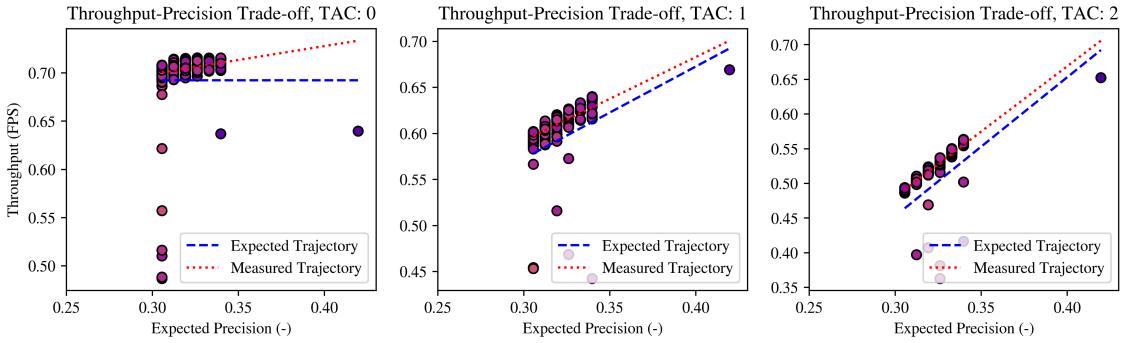


Figure 3.66: Throughput vs precision for policy 3, strategy 2 on the RPi. Lighter colors are associated with higher temperatures.

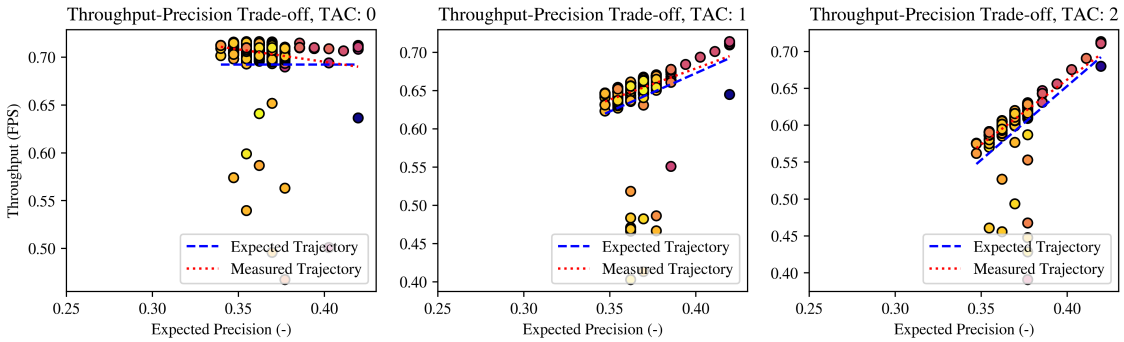


Figure 3.67: Throughput vs precision for policy 3, strategy 3 on the RPi. Lighter colors are associated with higher temperatures.

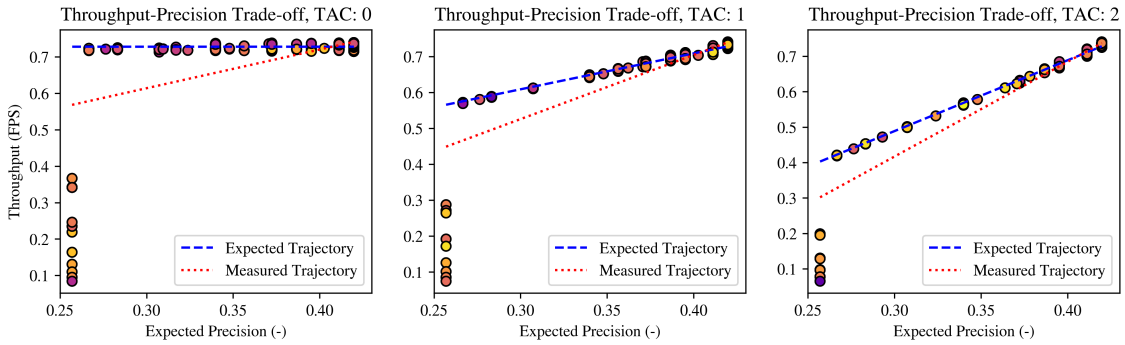


Figure 3.68: Throughput vs precision for policy 3, strategy 1 on the Nano. Lighter colors are associated with higher temperatures.

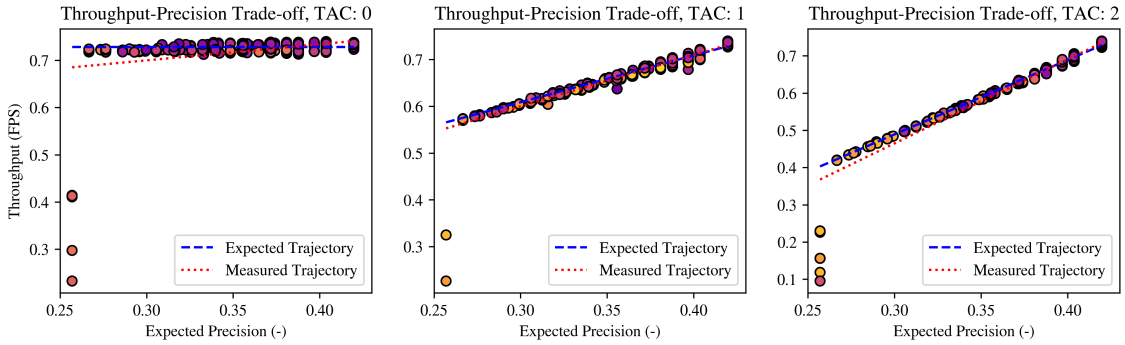


Figure 3.69: Throughput vs precision for policy 3, strategy 2 on the Nano. Lighter colors are associated with higher temperatures.

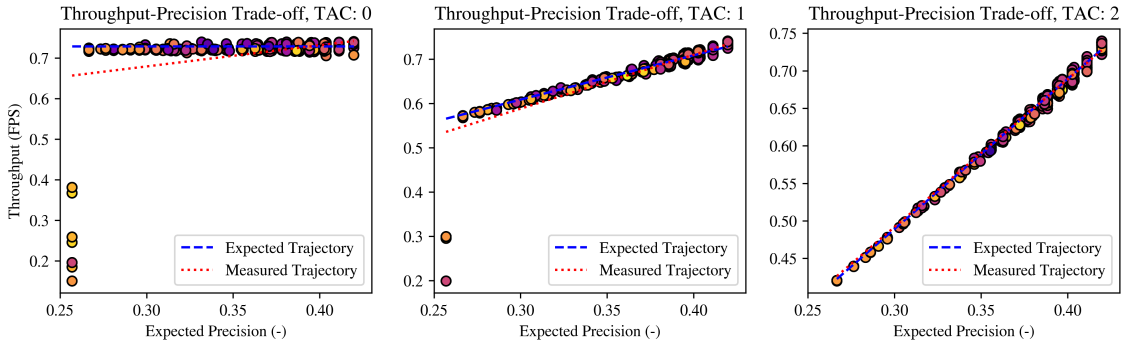


Figure 3.70: Throughput vs precision for policy 3, strategy 3 on the Nano. Lighter colors are associated with higher temperatures.

There is a clear trade-off between each metareasoning policy. Policy 1 favors high detection precision while sacrificing throughput. For use cases where the fraction of true positives must be high, though speed is not as important, policy 1 is a good choice. If the object detection is only required intermittently, then this policy will provide high precision detection with minimal loss in throughput. It additionally does not require any user-specified parameters, as any non-zero pause adjustment coefficient value will similarly reduce the temperature error.

Policy 2 favors high throughput consistency while sacrificing detection precision. This policy is therefore favorable where a program pipeline requires near constant input but the detection precision is less important. For example, commercial applications of object detection often provide video output with a near-constant frame rate for the benefit of the customer, but do not always require consistently high precision. The policy requires the specification of the switching strategy, though switching strategy 1 would likely be the best choice in most scenarios.

Policy 3 provides a generalization of both policies 1 and 2, but requires more user input. Given the image processing networks which are chosen, an appropriate TAC must be defined so as to avoid creating a negative desired throughput. Additionally, like policy 2, a switching strategy must be chosen. Given this input, though, a highly-tailored version of the policy could be created for the user.

It should be noted that in the cases of policies 2 and 3, the program operates in a fashion similar to that of a hybrid control system. Branicky highlights that the internal stability of hybrid systems is not guaranteed by the stability of the component systems [31]. Even if each component system is internally stable, it is sometimes possible to

create a switching strategy that makes the hybrid system unstable. Our testing has shown that the hybrid systems designed for policies 2 and 3 are stable.

3.5 Summary

Results from testing on the Raspberry Pi 4B and NVIDIA Jetson Nano Developer Kit demonstrate that all three designed metareasoning policies, given the right input parameters, are able to stabilize the temperature of their device during processing. Policy 1 is able to do so for any value of pause adjustment coefficient, no matter the initial pause duration. Policy 2 is able to do so with any switching strategy, though only switching strategy 1 is able to maximize the average detection precision given minimum throughput and maximum temperature constraints. For most input parameters, policy 3 is able to stabilize the device temperature. Careful consideration of the TAC is required before using this policy for consistent temperature stability. In the case of EDL-[0-4], a TAC between zero and four on the Raspberry Pi can provide a throughput-precision trade-off tailored to the user's preference.

Chapter 4: Learned Policies

4.1 Research Question

Chapter 3 describes how different metareasoning policies can control the temperature of an electronic device during image processing. Given this information, is it possible to create a policy that may combine policies 1 and 2 using reinforcement learning?

4.2 Reinforcement Learning Policy - Policy 4

This system can be framed as a reinforcement learning problem because it has an agent which performs actions in an environment [17]. The agent is the image processing program. The environment is the SoC temperature, pause duration, and current network. The program is then able to take actions in its environment by modifying the pause duration or active network. Therefore, a reinforcement learning approach can be taken to address this problem. The full training program, which will train policy 4, is described in Algorithm 8. For each device, a five-hour training session was provided for policy 4 to be learned.

Algorithm 8 Reinforcement Learning Policy (Policy 4) Training

$T_s \leftarrow 50°C$ ▷ Set start temperature
Require: $T \leq T_s$
$t_0 \leftarrow 0s$ ▷ Set test start time
$t_s \leftarrow 300s$ ▷ Set test duration
$T_d \leftarrow [70, 55]°C$ ▷ Set desired temperature for [RPi, Nano]
$p_d \leftarrow 0s$ ▷ Set initial pause duration
$p_a \leftarrow 0.2s/°C$ ▷ Set pause adjustment coef.
$Network \leftarrow Network_4$ ▷ Set starting network
$N_{d_{max}} \leftarrow N_{d_{max}}$ ▷ Set maximum network duration
$t_{max} \leftarrow 1/N_{d_{max}}$ ▷ Set maximum throughput
$\epsilon \leftarrow 0.9$
$\gamma \leftarrow 0.9$
$\alpha \leftarrow 0.2$
Initialize Q-table ▷ Table of values of each state-action pair
while $t < t_s$ **do** ▷ t is the program time
    $T \leftarrow T$ ▷ Measure temperature
    Process image
    Record N_d ▷ The processing duration
    Record T ▷ The current temperature
    Get state s_t
    Take action a_t
    Wait p_d seconds
    Record L_d ▷ The loop duration
    Record T
    Get new state s_{t+1}
    Get reward r_t
    Update Q-table
end while
Save Q-table

4.2.1 Q-Learning

Q-learning was chosen as the type of reinforcement learning to be used for this problem. Q-learning is a value iteration method which updates the value, $Q(s_t, a_t)$, of a state-action pair, s_t, a_t, given the reward, r_t, received for entering the new state and the value of the new state-action pair, $Q(s_{t+1}, a)$ [17]. The update equation is shown in equation 4.1:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a_t} Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{4.1}$$

where α is the learning rate and γ is the discount factor [17].

4.2.2 States

The state s_t of the system can be described by the temperature, T, of the processor, the duration of the pause, p_d, and the current network, N. These three values can characterize every possible state that the agent can be in during normal operation. Particularly, the operating ranges of T, p_d, and N are:

- T: [40, 95.1] °C, in intervals of 0.1 °C
- p_d: [0, 10] seconds, in intervals of 0.1s.
- N: [0, 5]

The number of possible states for the system is the number of points in the grid composed of these two coordinate vectors. Therefore, there are a total of 300,500 states in which the system can be.

4.2.3 Actions

The agent must be able to interact with its environment, so it was provided four possible actions to perform. Action 1 leaves the pause duration and current network unchanged. Actions 2 and 3 iterate up or down and adjust the pause duration to maintain a constant throughput. This is what metareasoning policy 2 does. Action 4 increases the

pause duration in the same way that metareasoning policy 1 does. Note that p_d could not take on a value. The actions which can be taken are the following:

1. $p_d \leftarrow p_d$, $Network_i \leftarrow Network_i$
2. $p_d \leftarrow 1/t_{max} - N_d$, $Network_i \leftarrow Network_{i+1}$
3. $p_d \leftarrow 1/t_{max} - N_d$, $Network_i \leftarrow Network_{i-1}$
4. $p_d \leftarrow p_d + p_a(T - T_d)$, $Network_i \leftarrow Network_i$

4.2.4 Reward Schedule

The environment provided a reward to the agent corresponding to the error between the current temperature, T, and the desired temperature, T_d. The reward schedule for this environment is shown in equation 4.2:

$$r_t = \begin{cases} 1/D^2 - 1 & D < 1 \\ 0 & D = 0 \\ -D^2 + 1 & D > 1 \end{cases} \tag{4.2}$$

where $D = \mid T - T_d \mid$.

Note that the threshold between positive and negative rewards was chosen to be 1 °C difference between the measured temperature and desired temperature. This is because a 1 degree temperature error was considered acceptable, but any error beyond that was not preferable. This schedule was chosen because it provides a strong incentive to stay around the desired temperature, T_d.

4.2.5 Exploration vs Exploitation

To encourage the agent to explore its environment and assign values to state-action pairs that it has not yet encountered, it will occasionally choose a random action rather than the one which would provide the maximum value. The frequency of this exploration is controlled by the parameter epsilon, ϵ [17].

Before the agent chooses an action, a random number between zero and one is generated. If epsilon is greater than that number, a random action is chosen. Otherwise, the action which has the most value is chosen.

4.2.6 Validation

The learned policy was validated by using a snapshot of the q-table at the end of training as a static policy where the program is run under the same conditions. This program is shown in Algorithm 9.

Algorithm 9 Reinforcement Learning Policy (Policy 4)

$T_s \leftarrow 50°C$
Require: $T \leq T_s$
$t_0 \leftarrow 0s$
$t_s \leftarrow 300s$
$T_d \leftarrow [70, 55]°C$
$p_d \leftarrow 0s$
$p_a \leftarrow 0.2s/°C$
$Network \leftarrow Network_4$
$N_{d_{max}} \leftarrow N_{d_{max}}$
$t_{max} \leftarrow 1/N_{d_{max}}$
$\gamma \leftarrow 0.9$
Load Q-table ▷ Load the trained Q-table
while $t < t_s$ **do**
    $T \leftarrow T$
    Process image
    Record N_d
    Record T
    Get state s_t
    Take action a_t
    Wait p_d seconds
    Record L_d
    Record T
end while

4.3 Results

Results for both Q-table training and validation are necessary to understand the effectiveness of policy 4. The Q-table is shown as a 3D plot that presents each action as a color located according to its corresponding state values for pause duration, temperature, and neural network. The policy map for the Raspberry Pi 4B is shown in Figure 4.1, while the map for the Nano is shown in Figure 4.3.

Validation tests were performed for 600 seconds. The RPi validation test is shown in Figure 4.2, while the validation test for the Nano is shown in Figure 4.4.

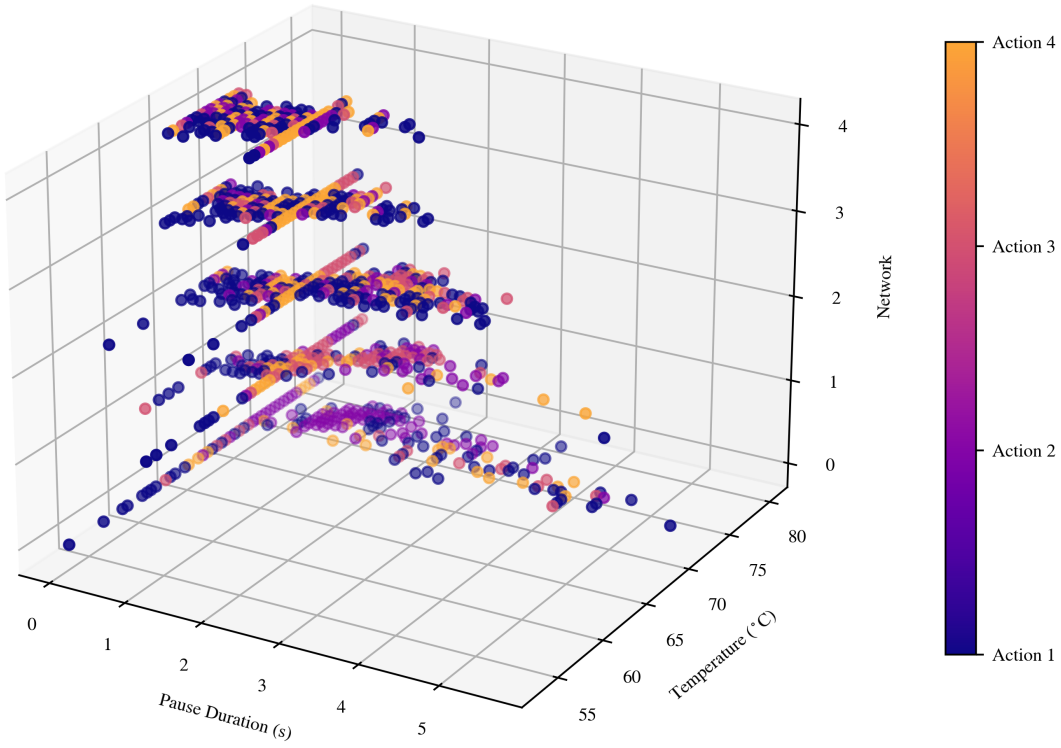


Figure 4.1: Raspberry Pi policy 4 map.

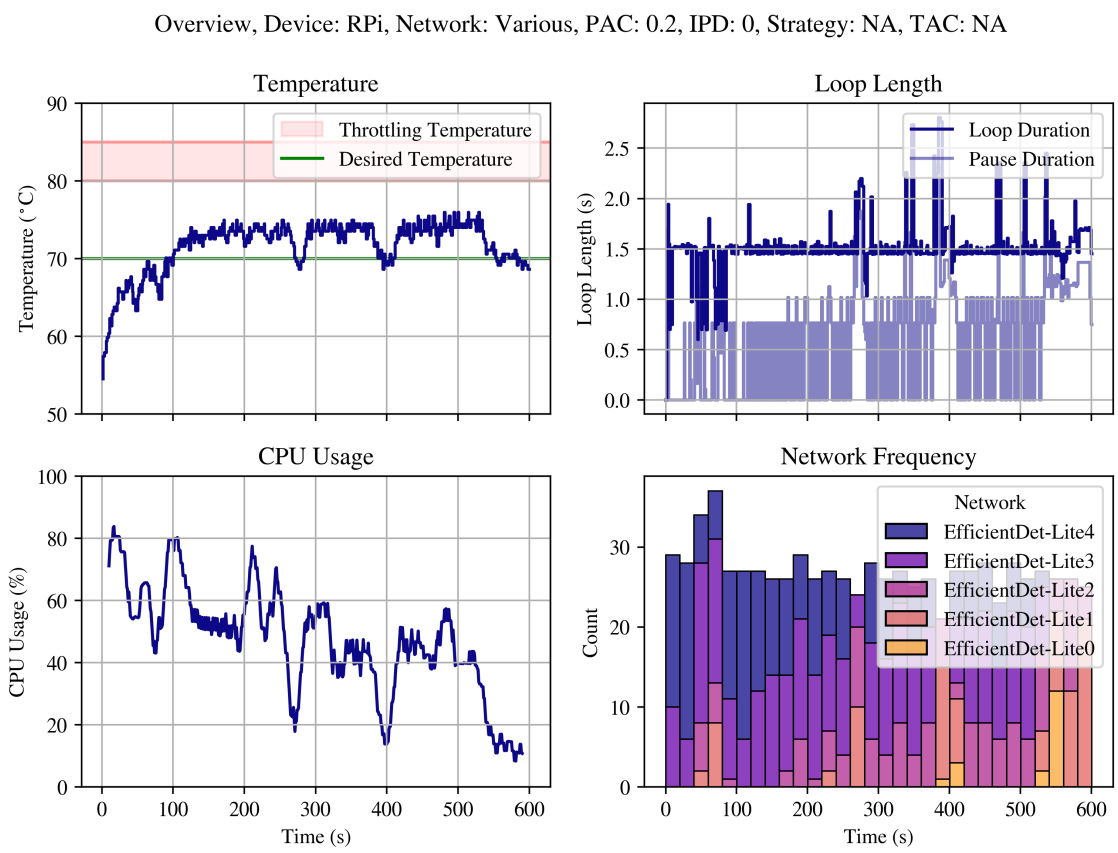


Figure 4.2: Raspberry Pi policy 4 validation test.

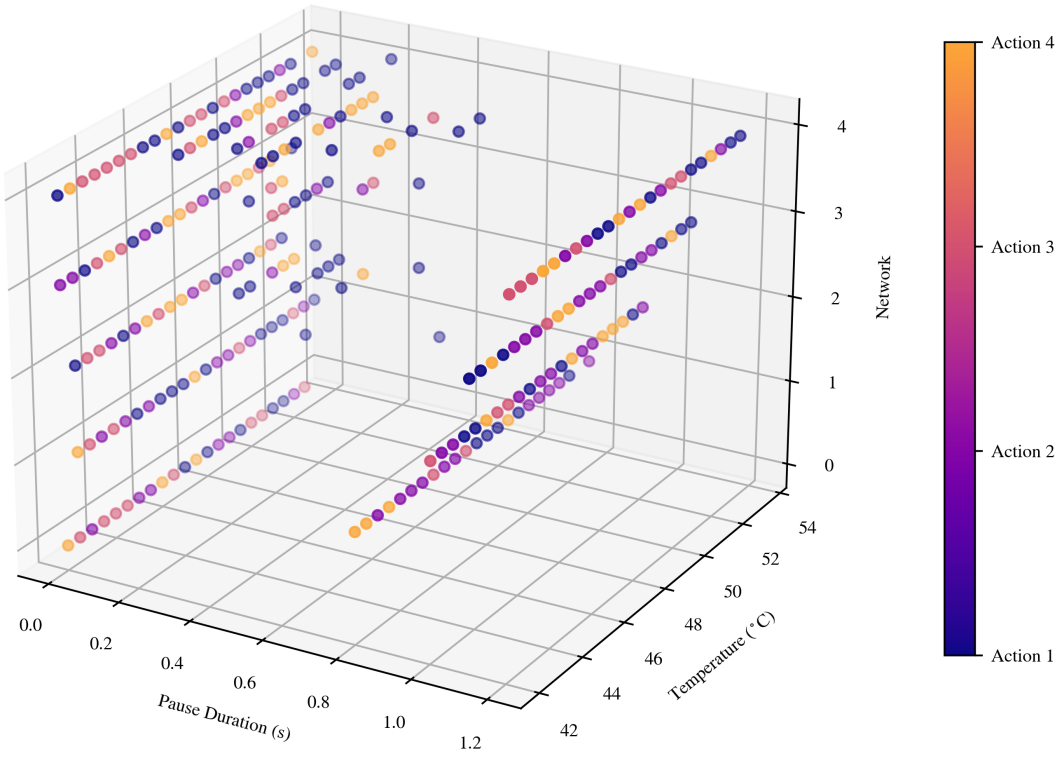


Figure 4.3: Raspberry Pi policy 4 map.

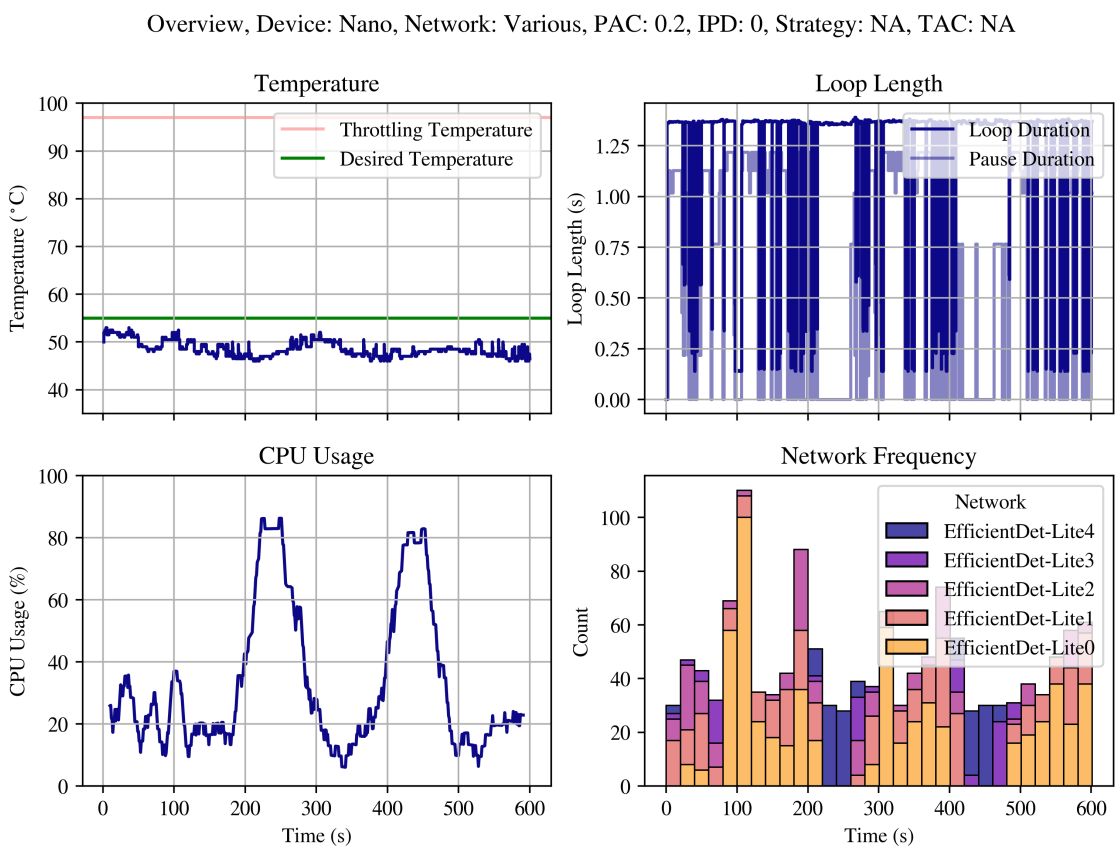


Figure 4.4: Raspberry Pi policy 4 validation test.

4.4 Discussion

The results from the Q-table training provide policy maps full of information. Figure 4.1, the RPi policy map, shows concentrations of state-action pairs around a temperature of 70 °C, the desired temperature. This is likely caused by the drastic change in the provided reward as the temperature gets further from the desired temperature. If the reinforcement learning agent "steps" away from the desired temperature by more than 1 °C, it will be assigned a large negative award. Therefore, the agent tended to stay near the desired temperature unless it was exploring.

Additionally, the number of desirable actions learned tends to positively corresponds to the pause duration associated with each network. Therefore, as the network index increases from 0 to 4, the pause duration where learned state-action values are most dense corresponds to the maximum loop length minus the expected network duration. Variations in the pause duration associated with each state-action value tend to increase as the desired temperature is reached.

Qualitatively, action 2 seems to be the best choice for many states when EDL-4 is in use and the temperature is higher than the desired temperature. This means that when the temperature was too high and the network EDL-4 is in use, the agent learned to switch to EDL-3. This is analogous to what is performed during metareasoning policy 2.

Figure 4.2 shows that the policy was successful in maintaining a certain temperature, but not consistently. The policy trends toward using less computationally intensive neural networks while occasionally adjusting the pause length to bring the temperature from highs around 75 °C to below the threshold. This is not an optimal strategy, but mimics

trends seen in policy 1 and 2 to some success.

The Q-table trained on the Nano, shown in Figure 4.4, is vastly different from the table trained on the RPi. The values of far fewer state-action pairs were learned, while those that were were concentrated around either the static pause lengths assigned to maintain a constant throughput, or zero. Additionally, temperatures above 54 °C were never reached in training, so the policy acts randomly above that temperature.

The validation test in Figure 4.4 is indeed equally different from the RPi validation test. While it can be said that the policy maintained a temperature less than the threshold, it does not optimize for throughput or precision. It therefore performs less efficiently than either policy 1 or 2.

Chapter 5: Conclusion

This thesis proposed four different metareasoning policies for managing the SoC temperature on electronic devices during image processing. These policies directly modified the image processing program during each loop. Policy 1 did so by adding pauses to the program, with each pause duration dependent on the difference between the current and desired temperatures. Policy 2 did so by switching between different neural networks and adding a static pause length for each so as to maintain a static image throughput. It did so using one of three different switching strategies. Policy 3 combined the principles of both policies 1 and 2 by switching between neural networks and adding pauses with increasing length depending on the decrease in detection precision and a user-defined parameter, TAC. Policy 4 was trained via Q-learning to take actions associated with both policy 1 and policy 2.

For each policy, a range of tests were performed with a range of parameter values. Tests were performed on a Raspberry Pi 4B and an NVIDIA Jetson Nano Developer Kit. These tests were performed in the same thermal environment, with a controlled start temperature, and with as few simultaneously running programs as possible.

Testing results show that policies 1 and 2 are successful at maximizing either detection precision or minimizing throughput variance for a range of parameter values. Results for

policy 3 show that the TAC parameter can be tuned to maintain a constant temperature while performing a user-defined trade-off between detection throughput and precision. Results for policy 4 show some success in maintaining a constant temperature, though neither device-specific policy maximized detection precision or maintained a consistent throughput while doing so.

Policy 1 would be best used when operating as a part of a system that mandates a certain average expected detection precision. If an autonomous agent must maintain an average detection precision of at least 0.4, but does not mandate a constant throughput, policy 1 using EDL-4 would be satisfactory, where the other policies would not be.

Policy 2 would be best used where consistent throughput is always prioritized over consistent detection accuracy. One example use case would have a human viewing the output object defections in real-time and making judgments based on the defections. Humans generally perform better at consistently high frame rates, so maintaining a constant throughput would be important in this scenario [32, 33].

Policy 3 would be best used in cases similar to that of policy 2. If a certain throughput loss tolerance can be found through use-case studies involving real-time human viewing, an appropriate TAC can be assigned and a higher average expected precision can be maintained than in policy 2. A summary of policy descriptions and use cases is shown in Table 5.1.

Future work in this area may involve more variants on the policies listed in this thesis. Insight could be gathered from testing policies 1 and 3 with a wider range of input parameters. Policy 2 may benefit from the addition of more neural networks, the testing of networks with purposes other than object detection, or additional switching strategies.

Table 5.1: Policy summary.

Policy	Summary	Use Case
Throughput Adjustment (1)	Adjusts throughput to maximize detection precision while remaining below a temperature threshold.	Consistent detection precision is valued more than consistent detection precision.
Network Switching (2)	Switches between neural networks to maximize detection precision while remaining at a desired throughput and below a temperature threshold.	Consistent throughput is valued more than consistent detection precision.
Hybrid (3)	Switches between neural networks to maximize detection precision while increasing throughput and remaining below a temperature threshold.	A tolerance for throughput loss as a function of detection precision is known.

An additional direction of research may involve modifying policy 3 so that it is viable at a larger range of TAC values. Policies 1, 2, and 3 could benefit by being formalized as control systems for the purposes of optimal parameter selection, or, in the case of policies 2 and 3, internal stability. Finally, there is much potential for reinforcement learning or other machine learning strategies to improve upon policy 4. Additional training time, larger action set, precision-based rewards, or increased variation in training parameters could lead to a more consistently successful version of this policy.

Bibliography

[1] Tesla Inc. Autopilot, 2022. https://www.tesla.com/AI.

[2] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[3] V. Lakshminarayanan and N. Sriraam. The effect of temperature on the reliability of electronic components. In *2014 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6, 2014.

[4] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, Jieping Ye, and Senior Member. Object detection in 20 years: A survey. 5 2019.

[5] Qianfan Xin. Durability and reliability in diesel engine system design. *Diesel Engine System Design*, pages 113–202, 1 2013.

[6] Raspberry Pi Ltd. Raspberry Pi Documentation - Frequency Management and Thermal Control.

[7] NVIDIA. Nvidia jetson nano thermal design guide. 2021.

[8] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien An Lai, Masazumi Matsubara, and Calton Pu. Impact of dvfs on n-tier application performance. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, page 732–737, New York, NY, USA, 2002. Association for Computing Machinery.

[10] J. H. Lienhard, IV and J. H. Lienhard, V. *A Heat Transfer Textbook*. Dover Publications, Mineola, NY, 5th edition, December 2019.

[11] Théo Benoit-Cattin, Delia Velasco-Montero, and Jorge Fernández-Berni. Impact of thermal throttling on long-term visual inference in a cpu-based edge device. *Electronics*, 9(12), 2020.

[12] Victor Wiley and Thomas Lucas. Computer vision and image processing: A paper review. *International Journal of Artificial Intelligence Research*, 2:22, 6 2018.

[13] Michael T. Cox and Anita Raja, editors. *Metareasoning: Thinking About Thinking*. MIT Press. OCLC: ocn611551144.

[14] Samuel T. Langlois, Oghenetekevwe Akoroda, Estefany Carrillo, Jeffrey W. Herrmann, Shapour Azarm, Huan Xu, and Michael Otte. Metareasoning structures, problems, and modes for multiagent systems: A survey. *IEEE Access*, 8:183080–183089, 2020.

[15] Duc V. Nguyen, Huyen T. T. Tran, and Truong Cong Thang. A delay-aware adaptation framework for cloud gaming under the computation constraint of user devices. In *MultiMedia Modeling: 26th International Conference, MMM 2020, Daejeon, South Korea, January 5–8, 2020, Proceedings, Part II*, page 27–38, Berlin, Heidelberg, 2020. Springer-Verlag.

[16] Jayoung Lee, PengCheng Wang, Ran Xu, Venkat Dasari, Noah Weston, Yin Li, Saurabh Bagchi, and Somali Chaterji. Virtuoso: Video-based intelligence for real-time tuning on socs, 2021.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An Introduction (2nd edition 2018)*, volume 3. 2018.

[18] Anup Das, Bashir M. Al-Hashimi, Rishad A. Shafik, Akash Kumar, Geoff V. Merrett, and Bharadwaj Veeravalli. Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems. *Proceedings - Design Automation Conference*, 2014.

[19] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[20] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3296–3297, 2017.

[21] Jayoung Lee, Pengcheng Wang, Ran Xu, Venkat Dasari, Noah Weston, Yin Li, Saurabh Bagchi, and Somali Chaterji. Benchmarking video object detection systems on embedded devices under resource contention. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, EMDL'21, page 19–24, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Raspberry Pi Ltd. Raspberry Pi Documentation - Processors.

[23] Vivek Gite. How to find out Raspberry Pi GPU and ARM CPU temperature on Linux, 2021. https://www.cyberciti.biz/faq/linux-find-out-raspberry-pi-gpu-and-arm-cpu-temperature-command/.

[24] Malolo. Malolo's screw-less / snap fit customizable Raspberry Pi 4 Case and Stands, 2019. https://www.thingiverse.com/thing:3723561.

[25] Ben Croston. RPi.GPIO: A module to control Raspberry Pi GPIO channels. http://sourceforge.net/projects/raspberry-gpio-python/.

[26] NVIDIA Corporation. Nvidia jetson nano developer kit — nvidia developer, 2022.

[27] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. 2019.

[28] TensorFlow Team. Object detection with tensorflow lite model maker.

[29] COCO Consortium. COCO - Common Objects in Context, 2017. https://cocodataset.org/home.

[30] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[31] M.S. Branicky. Multiple lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):475–482, 1998.

[32] Benjamin F. Janzen and Robert J. Teather. Is 60 fps better than 30? the impact of frame rate and latency on moving target selection. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, page 1477–1482, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Jessie Y. C. Chen and Jennifer E. Thropp. Review of low frame rate effects on human performance. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 37(6):1063–1076, 2007.