Implementation of A Thermal Head Room based P-State Driver in Linux

by

Harika Kilari

A thesis submitted to the Graduate Faculty of
Auburn University
in fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 10, 2016

Keywords:
Thermal Head Room, P-State Driver, Linux Kernel, Peak Temperature and Power Consumption

Approved by

Sanjeev Baskiyar, Chair, Associate Professor, Computer Science and Software Engineering
Anthony Skjellum, COLSA Professor, Computer Science and Software Engineering
Xiao Qin, Professor, Computer Science and Software Engineering

Abstract

Processor overloading causes excessive heat dissipation and high temperatures which may cause unreliable operation and low lifespan. To handle such risks, processors have thermal protection modules, limit the temperature within permissible temperature ceiling via thermal throttling. It is a computer architecture technique which lowers the operating frequency and voltage (or, p-states) dynamically to conserve power and/or reduce heat dissipation at the cost of performance. Intel's Proportional, Integral and Derivative (PID) controller uses p-states to control the temperature. We implemented a previously proposed Thermal Headroom Based p-state Driver in the Linux to reduce thermal violations above the target temperature. Thermal headroom is the resulting difference between adjacent peak (above target) to trough (below target) temperatures. When temperature rises, the thermal headroom driver cools the CPU by reducing its p-state until the temperature falls near the target temperature. Once there is enough thermal headroom, p-state is increased to boost performance. We conducted an evaluation using the SPEC CPU 2006 benchmark suite. The thermal headroom was successful in reducing thermal constraint violations, which in return could lower peak temperatures and energy consumption by 1 - 4℃ and 0.5 - 6 KJ respectively compared to the PID based controller.

Acknowledgments

I take pleasure in thanking many people for making my thesis possible. Firstly, I would like to thank Dr. Sanjeev Baskiyar for his guidance and support throughout this work. I owe much gratitude to Dr. Xiao Qin and Dr. Anthony Skjellum for serving as members of my advisory committee.

I thank my father K.S.S. Prasad and mother K. Padmaja for providing me with a good life and their determination to educate me. I also want to thank my sister Dr. Mounika for believing in me and for her constant encouragement.

I would like to thank my friends Sagar and Sonakshi for their constant support and encouragement. I thank Ravi Uppu and Shehenaz Shaik for their help throughout my thesis work. I express my deepest gratitude to my family and my friends for their love, compassion and support in my endeavor. Without them, I know that none of this would have ever been possible. Finally, I am truly grateful to GOD for giving me such good opportunity in life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Dynamic Thermal Management

Dynamic Thermal Management (DTM) is a heat management technique in computer architecture, where the system uses different cooling strategies to bring down the temperature of system. All electronic devices and circuits generate excess heat and thus require thermal management to improve reliability and prevent premature failure. DTM is required to remove the extra heat produced by computer components, to keep components within permissible operating temperature limits. All modern day processors are designed to cut out or reduce their voltage (which translates to power usage) and/or clock speed if the internal temperature of the processor exceeds a specified limit.

Components are often designed to generate as little heat as possible, and computers and operating systems may be designed to reduce power consumption and consequent heating according to workload, but more heat may still be produced than can be removed without attention to cooling. A 10℃ increase in overall temperature reduces the life span of electronic devices by half [1]. The cost for cooling and packaging also increases with increasing power [2], [3]. As the demand for high performance is growing, nowadays thinner and smaller computing systems such as laptops, ultra-books are being used for running HPC applications, which in turn demands innovative ways to monitor and reduce system heat dissipation. Thermal aware strategies have emphasized distributing the performance both temporally and spatially to mitigate temperature [4].

1.2 Dynamic Voltage Frequency Scaling (DVFS)

Dynamic Voltage Frequency Scaling (DVFS) is a power management technique in computer architecture, where the voltage used in a component is increased or decreased, depending upon circumstances. Dynamic voltage scaling to increase voltage is known as overvolting; dynamic voltage scaling to decrease voltage is known as undervolting. As the demand for high performance is growing, nowadays thinner and smaller computing systems such as laptops, ultra-books are being used for running HPC applications, which in turn demands innovative ways to monitor and reduce system heat dissipation and power consumption. Based on temperature, the BIOS controls the system fan. It monitors fan speed and can do thermal throttling accordingly.

Thermal throttling adjusts the duty cycle of the processor clock or reduces the operating frequency, voltage. Such controls significantly impact performance [5]. When the maximum specified CPU temperature settings are exceeded performance loss occurs due to the intervention of BIOS. The efficiency of some electrical components, such as voltage regulators, decreases with increasing temperature, so the power used may increase with temperature causing thermal runaway. Increases in voltage or frequency may increase system power demands even faster than the CMOS formula indicates. The key idea behind our approach was to limit performance when temperature crosses the set point. As power is proportional to the square of processor frequency, there is conservation of energy [6], [7]. When the system runs out of cool tasks, there is a rapid temperature increment, then DVFS takes corrective action to maintain temperature below cutoff as the temperature emergencies could happen at a granularity of millisecond.

1.3 Intel thermal daemon

Intel thermal daemon (thermald) is a viable solution for monitoring and controlling core temperatures implemented within Linux. While using techniques like thermal throttling, the system will experience performance loss. In order to prevent such an effect, the thermal daemon proactively cools the CPU by managing performance and thermal states. For controlling these states, P-state drivers and other cooling devices are being used in the Linux kernel developed by Intel. The block diagram for such an implementation is shown in Figure 1.1 which has been reproduced from [5]. Although these P-states perform quite well for the power and performance trade-off, they are not suitable for temperature control at a set point because they are discrete in nature and makes temperatures oscillate below and above the set point. The official information as to how many states are available in a system are not yet published and we know that they are limited in number [8]. This implementation allows user to configure thermal control on the system. A brief illustration of Intel thermal daemon is presented in Figure 1.1.

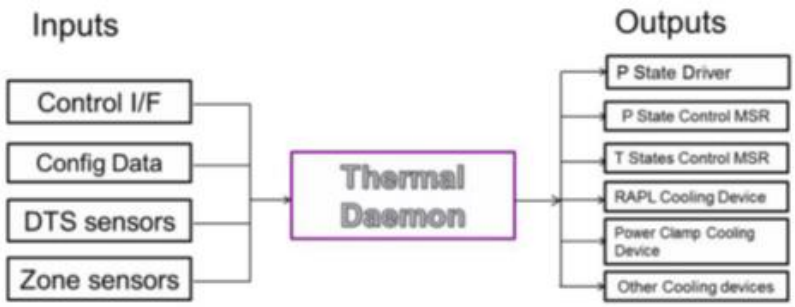


Figure 1.1 Intel Thermal Daemon

This module uses DTS sensors to track temperatures and uses Intel P-state driver, Power clamp driver, Running Average Power Limit control and cpufreq as cooling methods.

1.4 Thermal Headroom Approach

Thermal Head Room analysis helps determine if the temperature may exceed the target temperature in future. The Thermal Headroom approach takes into account the discrete nature of the P-States. Transition to higher P-States is made only when the headroom is available. After looking into several approaches and experiments, we determined that that a modified approach based on thermal headroom works well for temperature and offers energy savings with low schedule length penalty. In this research, a P-State driver was built which distributes performance temporally while governing the target temperature with minimal modifications to the underlying operating system.

Although temperature could be predicted using a temperature prediction model as in [9]– [10] but we used the thermal virtual file system (i.e., /sys/class/hwmon) in Advanced Configuration and Power Interface (ACPI) to directly read the temperature. Temperature reaction time (rise and fall time) was found to be of a tenth of a second similar to that in [11]. Thermal virtual file system reports temperature every 2 seconds which we choose as the sampling interval. Hardware performance counters, thermal virtual file system were made use of in gathering data for temperature, performance and power. Using the above data, the correlation between the CPU maximum performance and temperature was exploited. This approach reduces thermal oscillations and conserves energy by avoiding unnecessary cooling. We implemented this approach in Linux kernel and tested it on HPC applications such as SPEC CPU2006 benchmarks.

1.5 Organization of Thesis

The Chapter 1 of this thesis gives an introduction of the major approaches used for thermal management of multi-core processor systems. It briefly describes main techniques like Dynamic Thermal Management, Dynamic Voltage Frequency Scaling, Linux Thermal Daemon and Thermal Headroom Driver. The Chapter 2 gives an understanding of the different kinds of initial solutions used for temperature control. It also highlights the setbacks and tradeoffs for those solutions. The Chapter 3 introduces the purpose of the experiment followed by the method and calculations and other details of the experiment conducted and finally the results obtained. In the end the Chapter 4 includes the discussion of the results and the conclusion.

Chapter 2

Background

2.1 Thermal Strategies

2.1.1 Heat Balancing

Modern operating systems employ dynamic load balancing to improve response times and prevent starvation when the system is overloaded. In Linux, the load-balancing routine in the scheduler typically runs once in hundreds of milliseconds (200ms by default) [11]. It employs task migration to minimize the differences in task queue length for each core. To enable thermal aware scheduling in Linux, this scheme considers the thermal characteristics of each task and core while making task-migration decisions. When the system is overloaded, the Heat Balancing extension attempts to assign hot and cold tasks to each core in order to create opportunities for leveraging temporal heat slack. When the system has less number of tasks than the number of cores, the original load balancing routine does not perform any task migration but the heat-balancing routine moves a hot task to a colder, idle core to create opportunities for leveraging spatial heat slacks [4].

2.1.2 Deferred Execution of Hot Tasks

In some cases, the Heat-balancing may not be triggered in time to prevent the rising of temperatures. To cover such scenarios, a reactive scheme called the Deferred Execution scheme was implemented in the Linux scheduler [11]. When a core has multiple tasks and one of the tasks consistently heats up a core, the scheduler temporarily suspends the time slice of the current running hot task to allow other colder tasks to run before the hot task further heats up the core.

2.1.3 Cool Loop for SMT and single-thread

When a system is fully-loaded or over-loaded with hot tasks, there is no sufficient heat slack such that either Heat-balancing or Deferred Execution could leverage to reduce on-chip hot spots. In such cases, the system needs to employ fast-responding, workload reduction schemes to control temperature at the expense of performance. To lower temperatures while maintaining reasonable throughput, a new kernel task called Cool Loop was implemented, to create opportunities for temporal heat slack. The Cool Loop can be thought of as the same as the OS idle loop, but with a higher OS priority [11]. It could consist of no-op instructions or power-managing instructions that lower the core temperature through fetch throttling, frequency/voltage scaling or power-gating. When the cool loop is running it does not perform useful computation, but it has an OS priority that is higher than user tasks but lower than interrupt service routines and scheduler ticks. Allowing interrupt servicing routines does not impose additional risks of further heating because interrupt routines including scheduler ticks are typically short.

2.1.4 Feedback Control Scheduling

This technique proposed by Yue [12], uses cache-miss ratio as feedback, where they could set temperature at a reference point with a tight feedback loop controlling frequency via cache-miss ratio. However, in presence of noise due to fan and heat from other cores in case of a multi core general purpose computing CPU, the frequency-miss rate model does not work as intended. Thus, a direct control over frequency such as DVFS is indeed necessary.

2.1.5 Thermal Aware Scheduler in Embedded Systems

This scheme proposed in [13], exploits the variability between soft real-time and best effort applications to maintain the system temperature below a desired level while satisfying requirements such as throughput and fairness in temperature-constrained systems. The thermal model proposed as above would not support rapid temperature change because in a turbo state, the processor frequencies are opportunistically scaled [14] which in-turn leads to abrupt changes in temperature.

2.1.6 Priority Based Scheduling

The hot and cold processes are classified based on instructions per cycle. This technique targeted scenarios common to high-performance computing where processors are fully loaded. However when the system runs out of cool process, there is a rapid temperature increment, then DVFS takes corrective action, suggesting that scheduling cannot replace DVFS as thermal emergencies could happen at a granularity of millisecond [15].

2.1.7 Zig-Zag Scheduling

This scheme executes the workload at maximum speed until the maximum temperature threshold is reached, after that the speed is minimized to allow the processor to cool down. They have formulated an optimal Zig-Zag policy [16] by executing different jobs with different priorities at different discrete processor speeds for reduction in schedule length.

2.2 Intel Processor Technology

2.2.1 Processor States

A CPU can be put to different power states, depending on the current workload. These states are determined by the active parts of the CPU. C0 is active mode running instructions, C1, C1E are auto halt mode scaling frequency and voltage opportunistically. C3 corresponds to L1/L2 caches flush and clock off. G1(Sleeping mode) encompasses S1 corresponding to standby mode, S3 Suspend to Ram (STR) and S4 Hibernate mode. G3 is when the system is mechanically switched off. Processor P-States are defined as frequency/voltage operating states. All P-States are sub-states of C0 and the processor actually executes instructions in all P-States unlike other states, such as C1, C2 and C3. The maximum performance and the most power consuming state is P0. The whole system states and voltage, frequency varying with these states are illustrated in Figures 2.1 and 2.2 which are reproduced from [17].

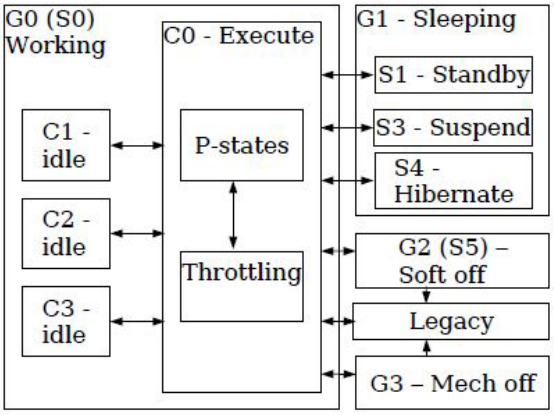


Figure 2.1 States of a CPU

C-State	P-State	MHz	Volts	Watts
C0	P0	1600	1.484	24.5
	P1	1300	1.388	22
	P2	1100	1.180	12
	P3	600	0.956	6
C1, C2	from P0	0	1.484	7.3
	from P3	0	0.956	1.8
C3	from P0	0	1.484	5.1
	from P3	0	0.956	1.1
C4	(any)	0	0.748	0.55

Figure 2.2 C-State and P-State Processor Power

2.2.2 Enhanced Intel Speed Step Technology

Enhanced Intel SpeedStep® Technology is an advanced means of enabling very high performance while also meeting the power-conservation needs. Separation of voltage and frequency changes. By stepping voltage up and down in small increments, the processor is able to reduce periods of system unavailability that occur during frequency change. The system is then able to transition between voltage and frequency states more often, improving balance between power and performance. It reduces the latency associated with changing the voltage/frequency pair, or P-state. Transitions can be undertaken more frequently, enabling more granular demand-based switching, and the optimization of the power and performance balance based on demand.

So while using the above techniques temperature violations may occur while trying to optimize performance under at a particular target temperature due to discrete P-States. To manage increasing constraints on power and thermal budgets, P-State transitions can be dynamically controlled.

Chapter 3

Experiment

3.1 Motivation

A Thermal Headroom based P-State driver is required in Linux Kernel as the P-States are discrete. Sudden P-State transitions might result in thermal violations leading to emergencies. We need real time data while making P-State transitions to prevent the violations. The Thermal Head Room based P-State driver provides a history based learning where thermal head room is measured during the time when the P-State is boosted. This approach learns that there is an overshoot of temperature with an increase in P-State, thus an increase in P-State is performed only when the calculated thermal headroom is available next time.

3.2 Thermal Headroom Based P-State Driver

This research monitors the CPU temperature and its variations by making P-State transitions using an approach that is inspired by Model Predictive Control (MPC). The adaptation of MPC for Thermal Headroom approach will be explained. MPC is a predictive algebraic method used for calculating results of a sequence of control variable changes. After a few initial observations, the controller can use a control sequence to produce desired output. Specific knowledge of the process could be used to optimize the best long-term output by foreseeing the results of an action in future. This approach prevents actions taken by conventional methods as they target short-term goals which are costly in the long-term.

3.2.1 Mathematical Representation

The deviations from the desired output ψ_δ, either specified by mathematical model or reference trajectory, produce an error function $\varepsilon(\tau) = \psi(\tau) - \psi_\delta(\tau)$ for increments of control actions $\Delta\upsilon(\tau) = \upsilon(\tau) - \upsilon(\tau - \Delta\tau)$. Here $\psi(\tau)$ is the output and $\upsilon(\tau)$ is the control action. In case of P-State control $p + 1^{th}$ state and p^{th} state can be considered as control actions and $\psi_\delta(\tau)$ is the desired target-temperature and $\psi(\tau)$ is the current temperature. Thermal Head Room based algorithm is inspired by MPC control. However, it uses a reference trajectory for measurements of $\psi(\tau)$, obtained through a system model which might be inaccurate in case of processors where the temperature varies both with program transformation and input control signal. Thus, we measure $\Delta\psi(\tau)$ as headroom for $\Delta\upsilon(\tau)$. We switch to higher P-State only when $\varepsilon(\tau)$ is less than or equal to this headroom. We switch to a lower P-State whenever the observed temperature is greater than target temperature. Such an approach prevents unnecessary oscillations of temperature around the target temperature because the P-State is increased only if it is predicted that doing so will not increase the temperature above the target temperature.

3.2.2 Algorithm

The Thermal Headroom algorithm waits for polling interval on line 5 and the temperature is read on line 6, if the observed temperature is greater than target temperature, the P-State is simply reduced to next lower state as in line 8 and a lookup headroom bit is set. When the temperature if found to be less than target temperature on line 10, then based on lookup-headroom the P-State is raised and Thermal Headroom is calculated on line 15. In the consecutive iterations, if the difference between target temperature and current temperature

is greater than the measured headroom, then an increment in P-State is made on line 19. A detailed algorithm is depicted here, reproduced from [18].

Algorithm Thermal Head Room Based P-State Driver

```
1: const P-state-max; P-state-min;                    ► max and min possible states
2: procedure THERMAL HEADROOM (int Target-temp)
3:     int T; lookup-headroom = 0; THR = 0;           ►Temp; flag; Thermal headroom
4:     while (1) do
5:         Wait (δt)                                  ►polling interval
6:         T ← Read Temperature
7:         if (T > Target-temp) and (P-state > P-state-min) then
8:             P-state − −                            ► cool CPU
9:             lookup-headroom ← 1                    ►set flag to compute headroom later
10:        else if (T < Target-temp) then
11:            if (lookup-headroom) and (P-state < P-state-max) then
12:                P-state + +                        ►switch state for computing THR
13:                Wait (δt)
14:                T′ ← Read Temperature
15:                THR ← T′ − T                       ►Compute headroom and turn-off flag
16:                lookup-headroom ← 0
17:            else if (Target-temp − T > THR) and (P-state < P-state-max) then
18:                P-state + +                        ►headroom available, so speed-up
19:            end if
20:        end if
21:    end while
22: end procedure
```

3.3 Implementation

The main purpose of Thermal Headroom Based P-State Driver was to serve thermal management functionality and help in temperature-aware scheduling. In this research we have implemented this logic within the Linux Kernel Scheduler. The code changes were made within /kernel/sched/core.c and a customized Linux kernel was built with long term stable linux4.4.4 version. The implementation can be broadly classified into two parts:

a) *Check threshold*: Computes thermal headroom to monitor the temperature within target using P-State changes.
b) *Measure*: Reads the temperature from the thermal virtual file systems.

In computing, the Advanced Configuration and Power Interface (ACPI) specification provides an open communication that operating systems can use to perform discovery and configuration of computer hardware components, for example, to perform power management by putting unused components to sleep, and to do status monitoring. Internally, ACPI advertises the available components and their functions to the operating system kernel using instruction lists, which the kernel parses and then executes the desired operations.

```
struct cpufreq_policy {

unsigned int          cpu;    /* cpu managing this policy, must be online */
unsigned int          min;    /* in kHz */
unsigned int          max;    /* in kHz */

}
```

The implementation of the current experiment are not a part of default kernel. To activate the thermal headroom features, we have compiled the code changes in linux kernel source to build a custom kernel. We have downloaded a recent stable kernel 4.4.4 to access the source code and made the required changes in Linux scheduler and system calls module. In the next step

we have configured the kernel to enable debugging and logging. We compile all the modules using *make* command and then build the custom kernel code into debian packages. The final step was to install the custom kernel *.deb* package using *make install* command. This updates the new kernel options into the configuration of the grub boot loader. The thermal headroom features are available once we boot into the new linux kernel. The Thermal Headroom module coded within Linux kernel scheduler uses this structure *cpufreq_policy* to make changes to the CPU frequency, where it is increased or decreased based on headroom available. This policy structure is associated with ACPI, described earlier, which will enforce Linux kernel to send commands to hardware to update the changes made.

The flowchart in Figure 3.3.1 demonstrates the flow of events and data within the headroom approach. Initially Thermal Headroom module reads the active/online CPU information from ACPI using *get_cpu( )* function. Then it reads the current frequency information of the CPU by calling *cpufreq_cpu_get(cpu).* Then the headroom module executes the algorithm discussed earlier and makes changes to frequency values within the cpufreq_policy structure. The modified data is then notified to ACPI through *cpufreq_cpu_put(cpu_policy).* ACPI further processes these changes into OS kernel instructions, and finally kernel uses the system commands to trigger hardware changes.

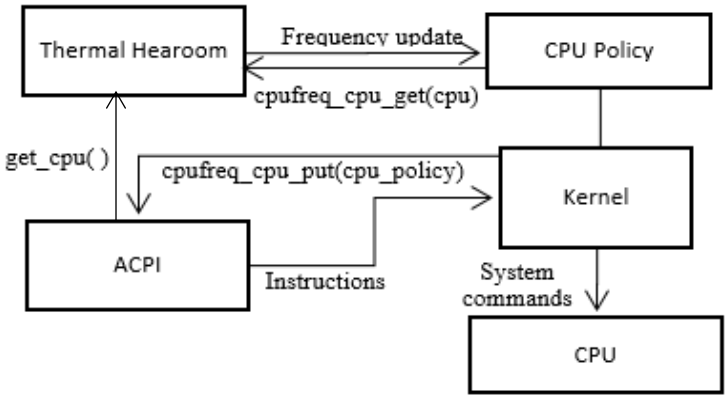


Fig. 3.3.1: Thermal Headroom Driver Flow Chart

We also need user space processes to conduct tests by running benchmarks. A system call is the programmatic way in which a computer program requests a service from the kernel of the

operating system it is executed on. A couple of system calls have been implemented {check_threshold, get_measure} to provide communication with Thermal Headroom module which now implemented as an integral kernel service. This user space application uses the thermal and temporal data obtained from kernel along with Linux Perf tool to evaluate the performance of the headroom approach against other controllers.

3.4 Experimental Setup

The experiments were performed on Intel i5-4200H dual core processor of Lenovo Y50 Gaming laptop running Ubuntu 14.04 LTS at base frequency of 2.8 GHz with 8 GB RAM. The processor chip is equipped with 128KB of L1 cache, 512KB of L2 cache, and 3MB of L3 cache. The PID and Thermal Headroom based approaches were coded in C and the later approach has been implemented within the Linux operating system kernel.

Target Temperature	Benchmarks	Group
56˚C	lbm	Group 1
	sphinx	Group 2
	povray	Group 3
	namd	Group 4
60˚C	calculix + omnetpp	Group 5
	perlbench + soplex	Group 6
	h264ref + astar	Group 7
	games + gromacs	Group 8
	tonto + leslie3d	Group 9
63˚C	bzip2 + gemfdtd	Group 10
	bwaves + gcc	Group 11
	sjeng + cactusadm	Group 12
	lbmquantum + wrf	Group 13
	hammer + gobmk	Group 14
	xalanbmk + zuesmp	Group 15

Table 1: Benchmarks Group Setup for Target Temperatures

The temperature was obtained from the system file */sys/class/hwmon/hwmon2/temp1_input* and the power consumed by the processor was recorded after every interval of 2-3 seconds. This interval was chosen to match the default setting of polling interval of 4 seconds within the thermald software. The perf tool was used to measure the power consumed by the processor. The system performance was broadly classified into 10 P-States each corresponding to range of values from 100 to 0. Each P-State affects the frequency and voltage of the system where the frequency would be altered between 300 MHz to 3GHz.

The processes were pinned to a single core by using task set functionality in Linux, creating full load on the core while the OS runs on the remaining cores. The room temperature was maintained at 21˚C /70˚F. A total of 26 variations of SPEC CPU2006 benchmarks were run. A run of individual benchmarks were able to build enough thermal stress to test target temperature 56˚C, but in order to evaluate higher target temperatures of 60˚C and 63˚C different combinations of benchmarks have been used. The benchmarks were grouped as {G1,G2,G3,G4}, {G5,G6,G7,G8,G9}, {G10,G11,G12,G13,G14,G15,G16} as shown in Table 1. The average temperatures observed when the benchmarks were run without controller are 59.6˚C, 68.2˚C and 74.3˚C. An average temperature of 52˚C is observed when the setup is idle. So target temperatures of 56˚C, 60˚C and 63˚C have been chosen which serve as middle point for idle and without controller scenario.

For a single run on each group at their respective target temperature, data has been recorded in a set of [time, temperature and power consumption] for every instance after a polling interval of 2-3 seconds roughly for a runtime of 700seconds. These data files have been collected for each group under three scenarios: i). *Native*: without any P-State controller ii). *PID Controller*: PID based P-State driver iii). *Thermal Headroom*: Headroom P-State driver

3.5 Data Processing

For a single run on each group at their respective target temperature, data has been recorded in a set of [time, temperature and power consumption] for every instance after a polling interval of 2-3 seconds roughly for a runtime of 700seconds. These data files have been collected for each group under three scenarios: i) Native: without any P-State controller ii) PID Controller: PID based P-State driver iii) Thermal Headroom: Headroom based P-State driver. The graphs are using sampled data with a sample interval of 20 seconds, they include representations of power and temperature for three scenarios described earlier.

3.6 Results (Graphical Representation)

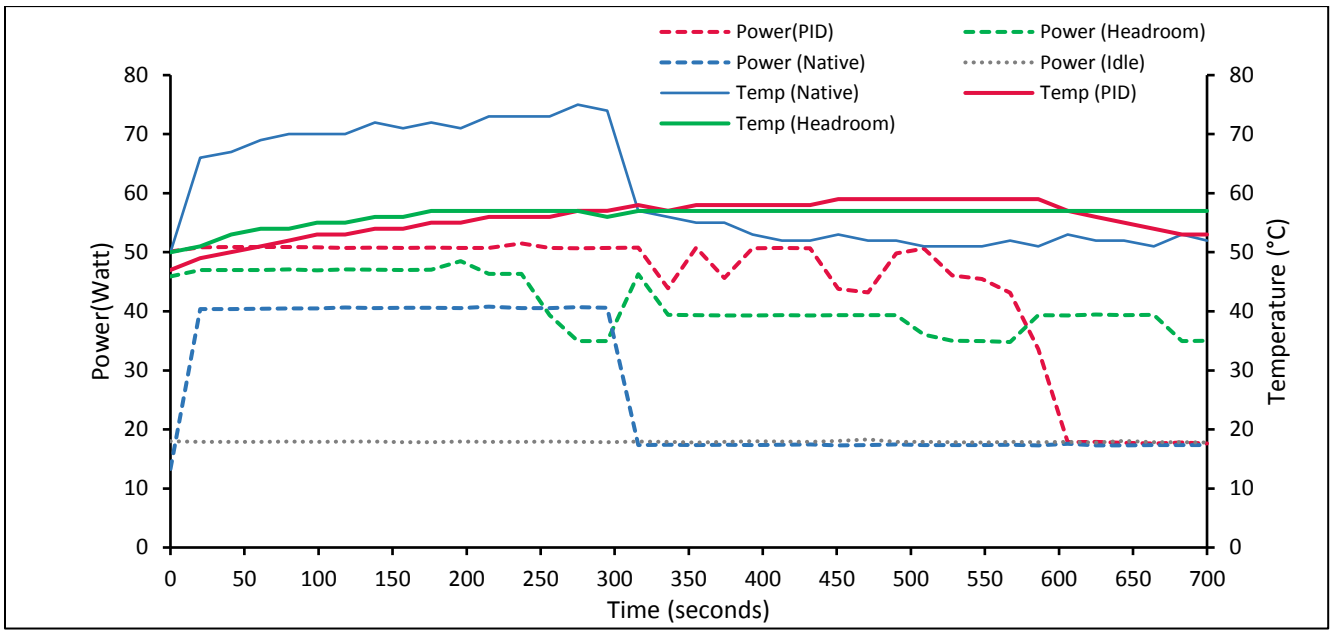


Fig. 3.6.1: Temperature and Power vs Time for Group1

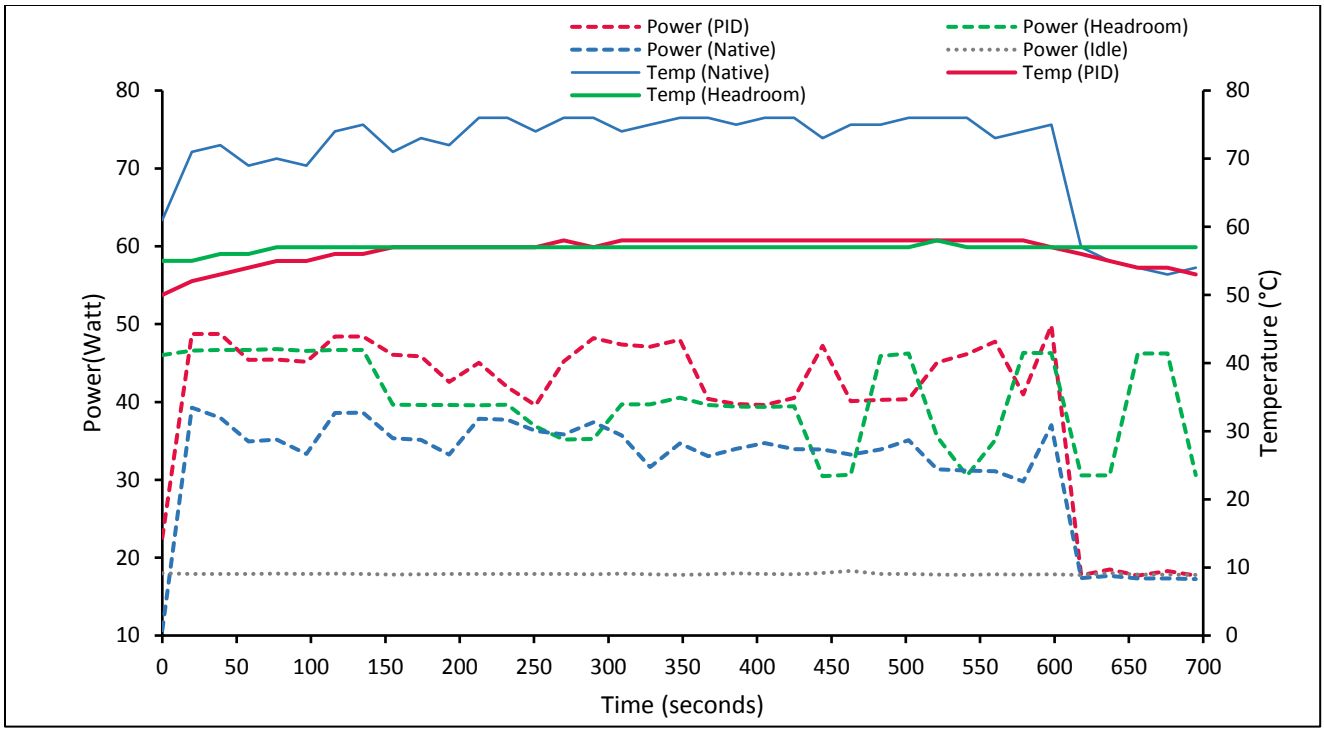


Fig. 3.6.2: Temperature and Power vs Time for Group2

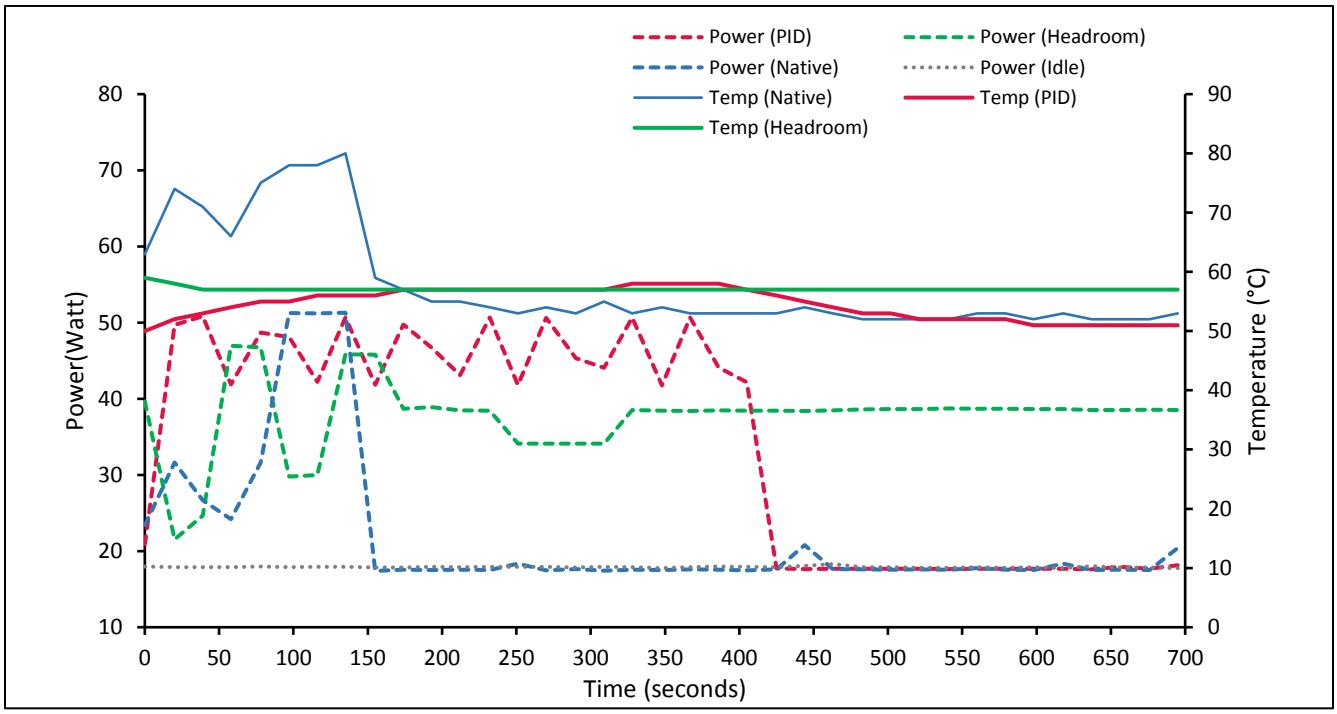


Fig. 3.6.3: Temperature and Power vs Time for Group3

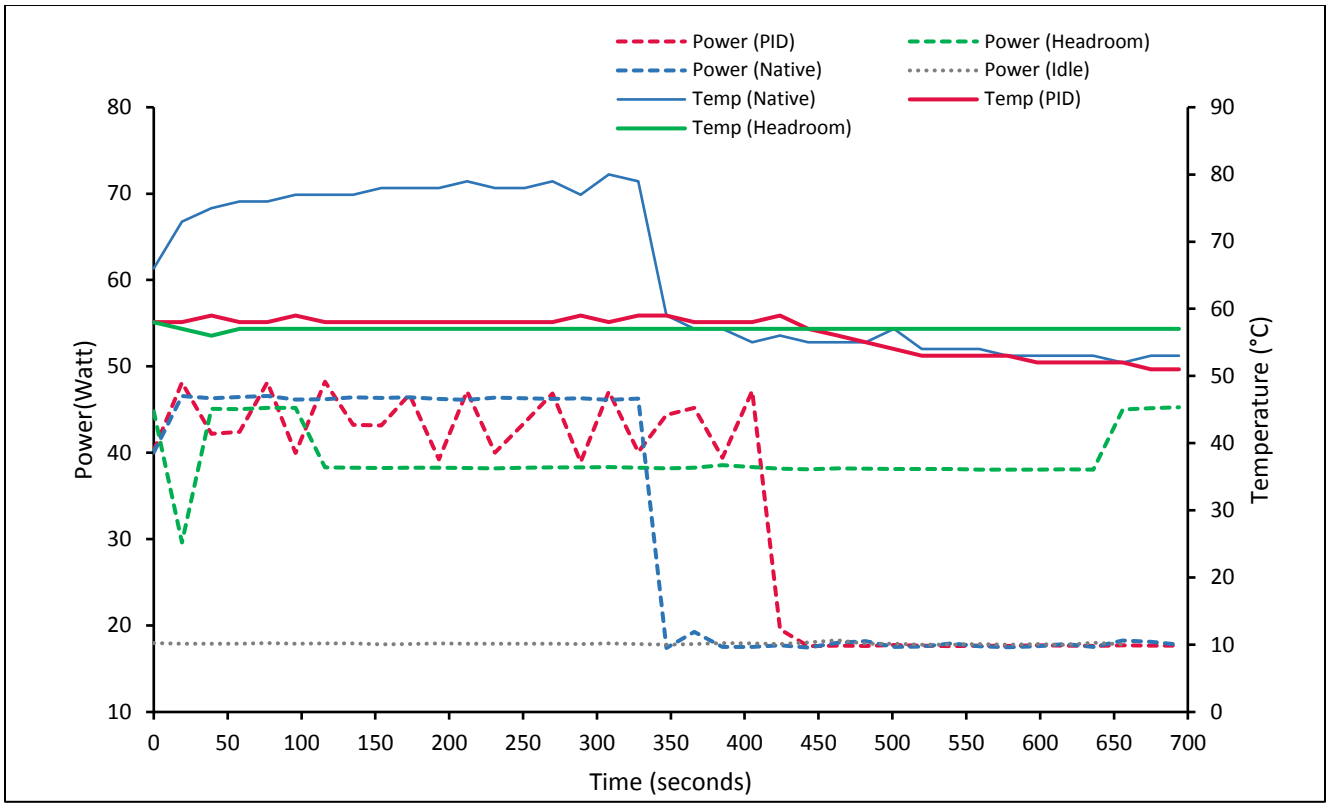


Fig. 3.6.4: Temperature and Power vs Time for Group4

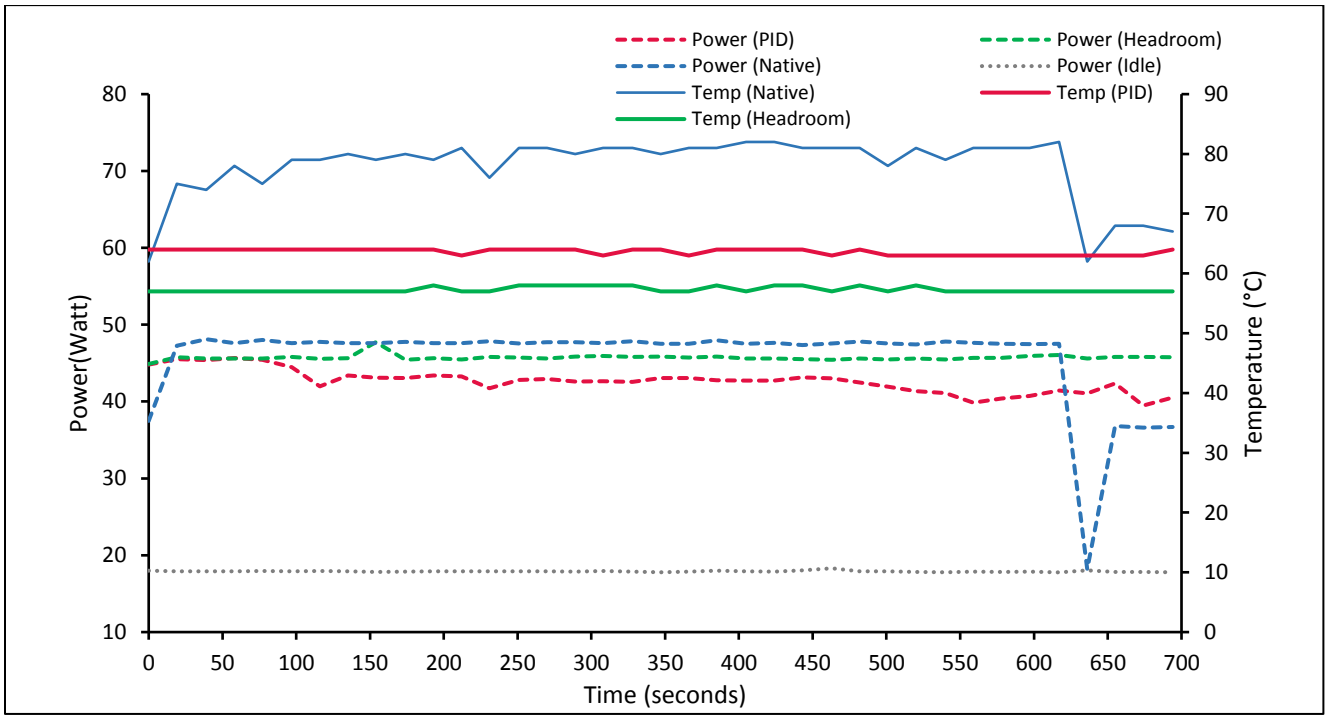


Fig. 3.6.5: Temperature and Power vs Time for Group5

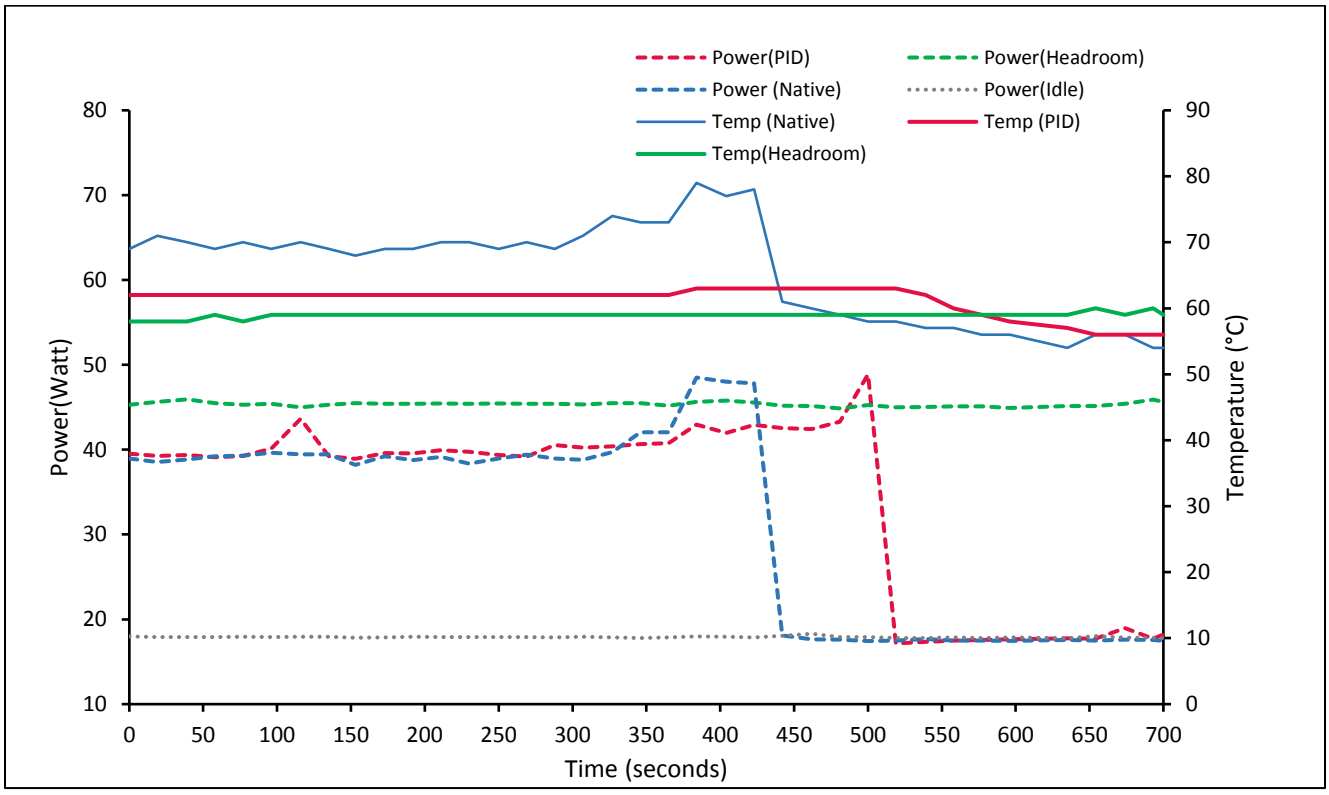


Fig. 3.6.6: Temperature and Power vs Time for Group6

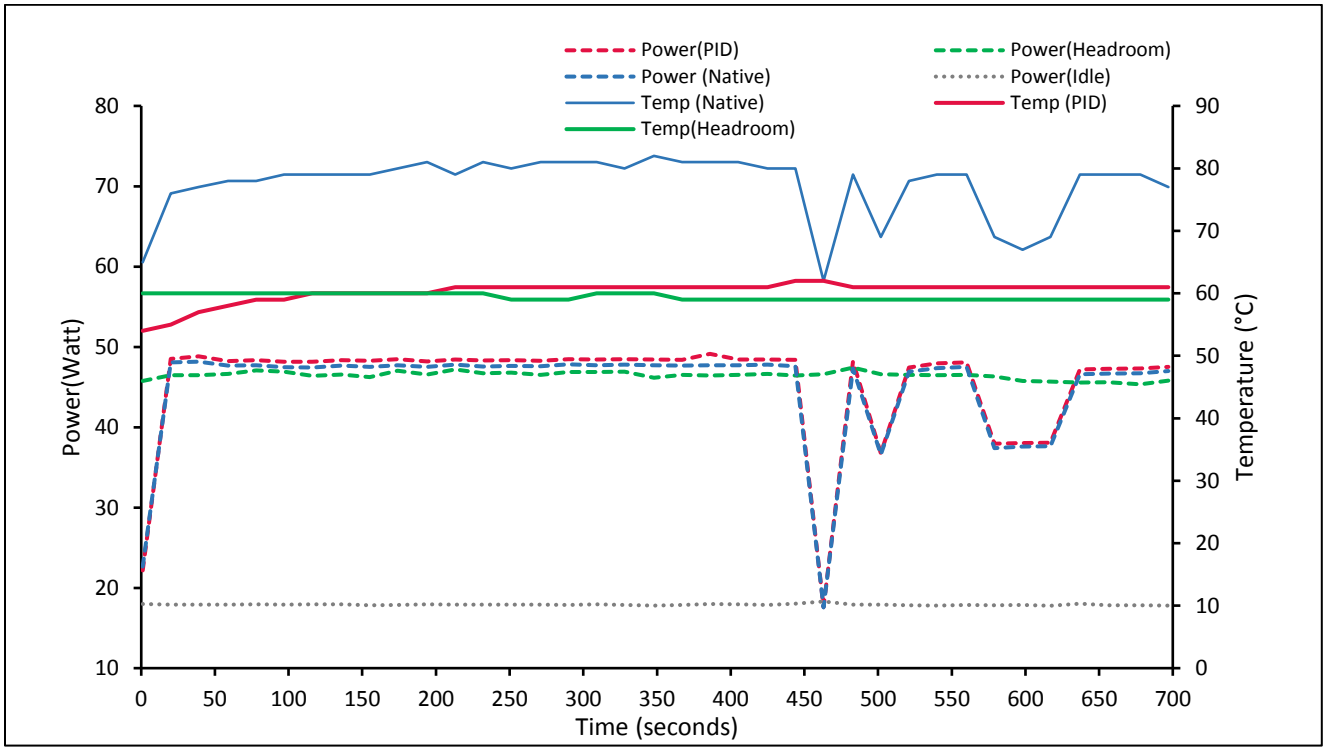


Fig. 3.6.7: Temperature and Power vs Time for Group7

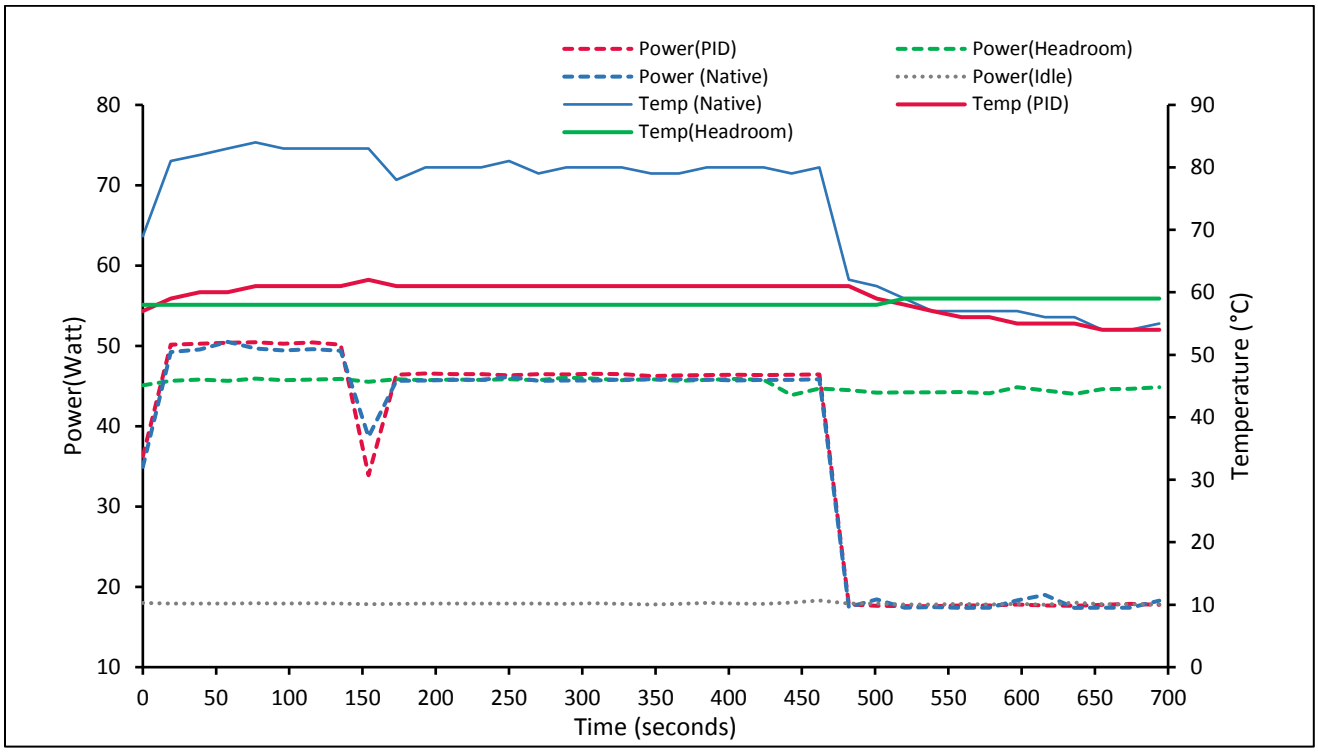


Fig. 3.6.8: Temperature and Power vs Time for Group8

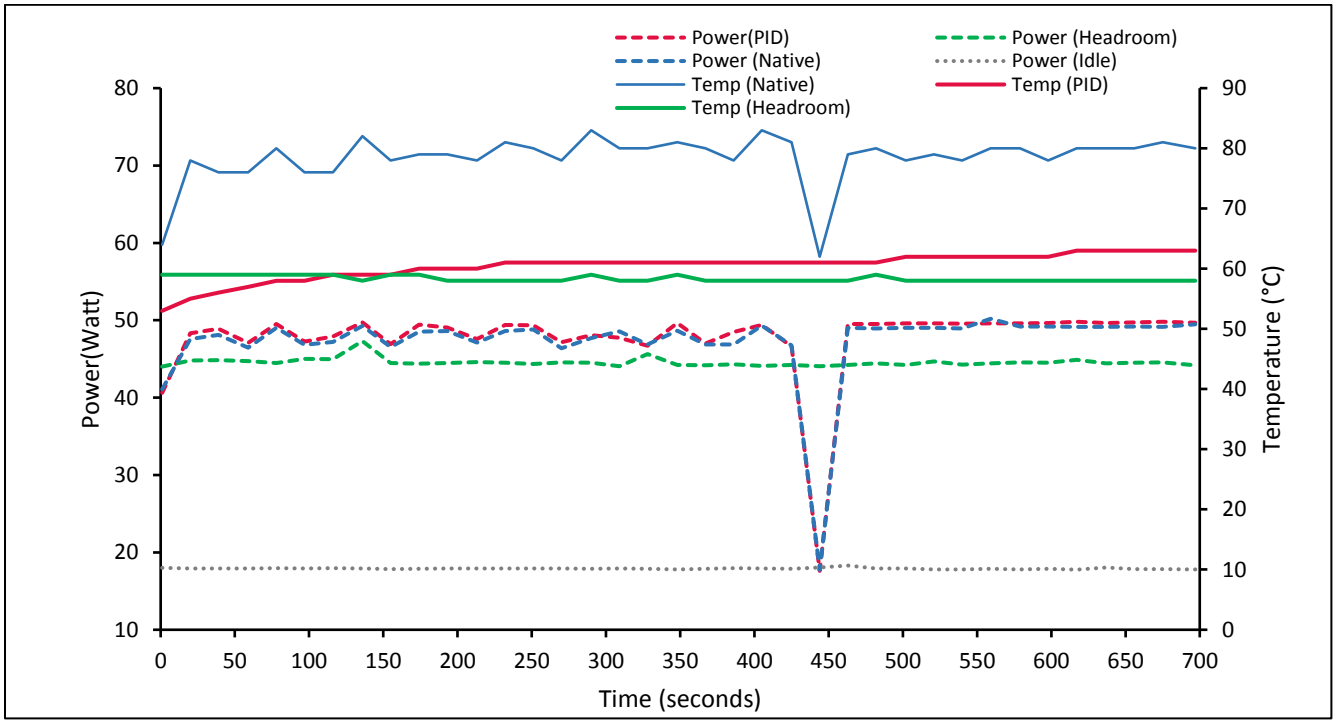


Fig. 3.6.9: Temperature and Power vs Time for Group9

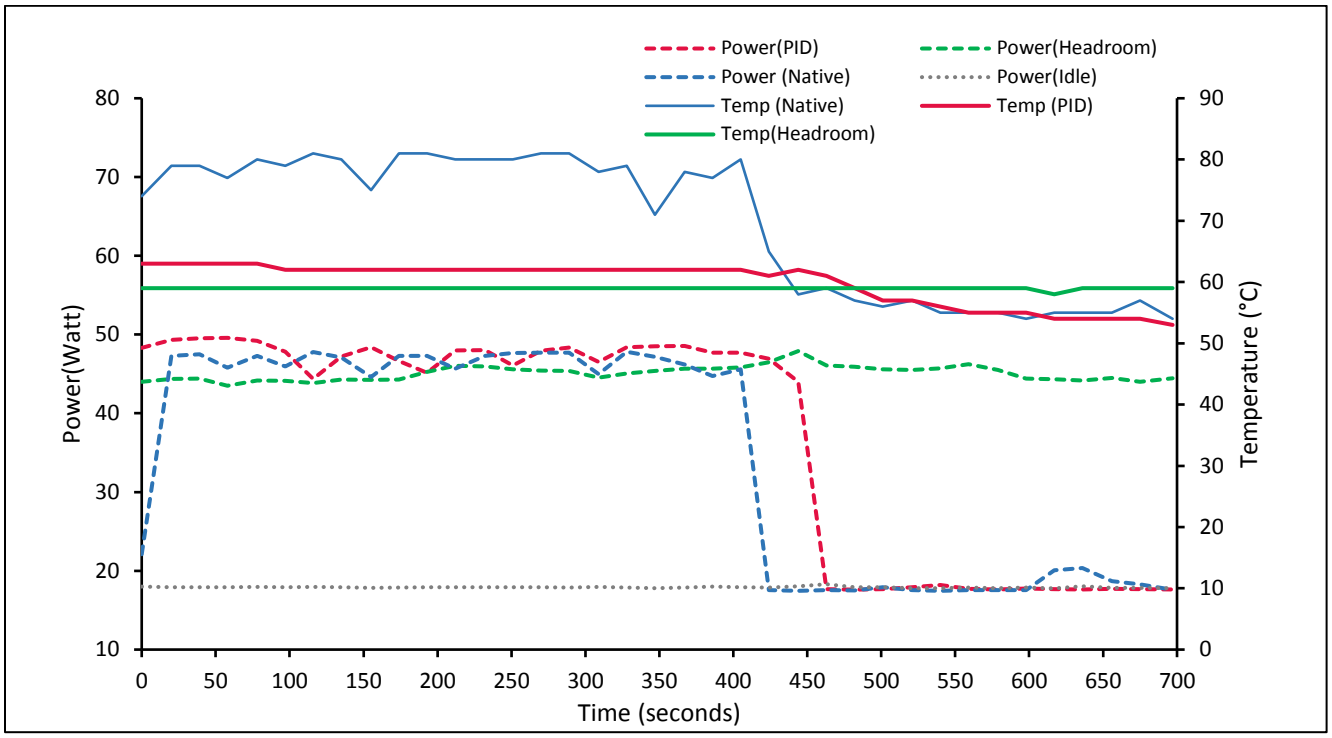


Fig. 3.6.10: Temperature and Power vs Time for Group10

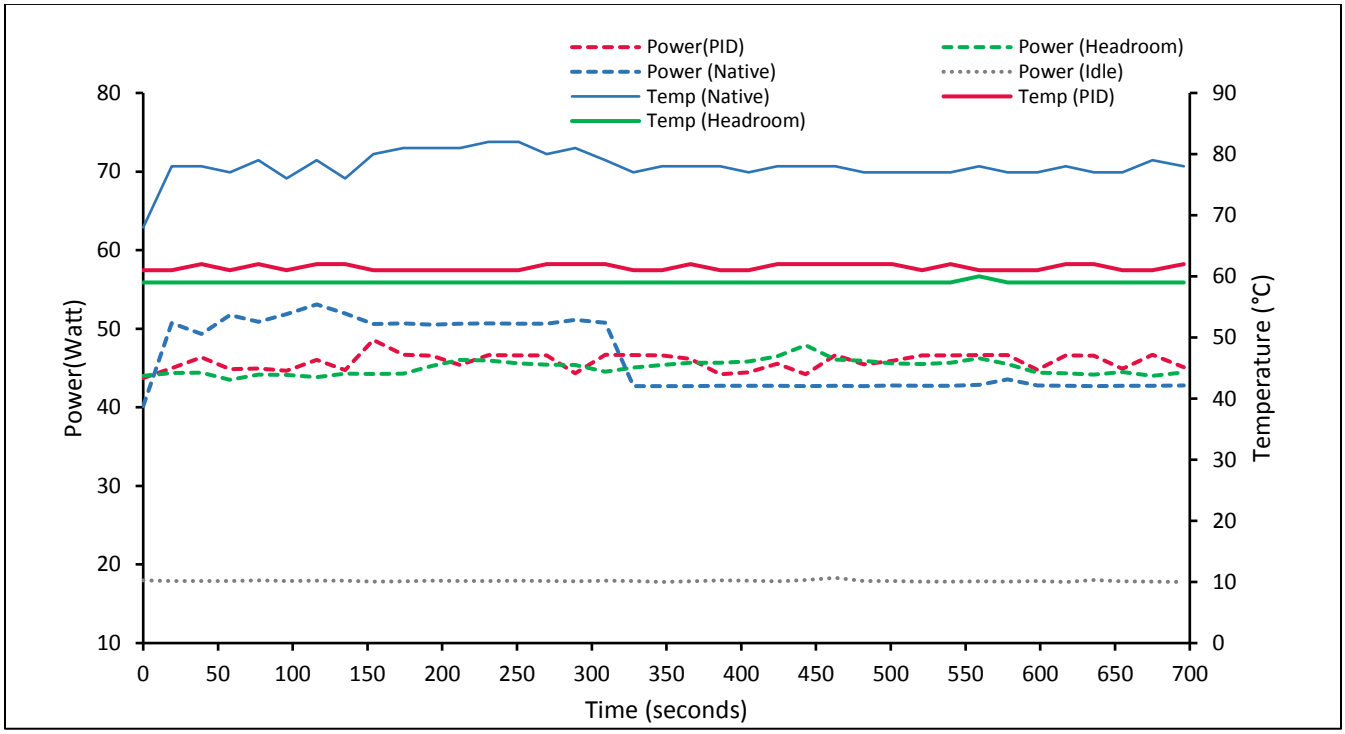


Fig. 3.6.11: Temperature and Power vs Time for Group11

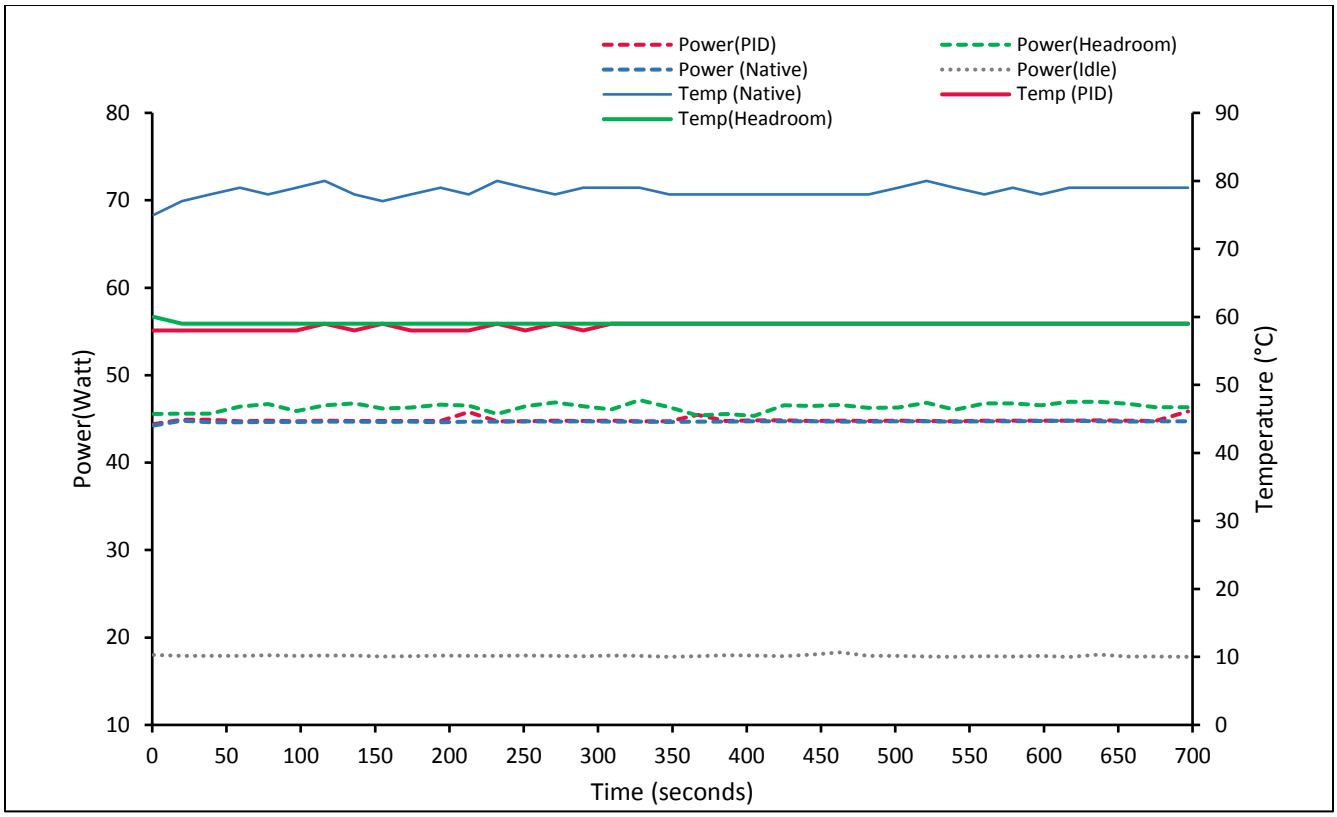


Fig. 3.6.12: Temperature and Power vs Time for Group12

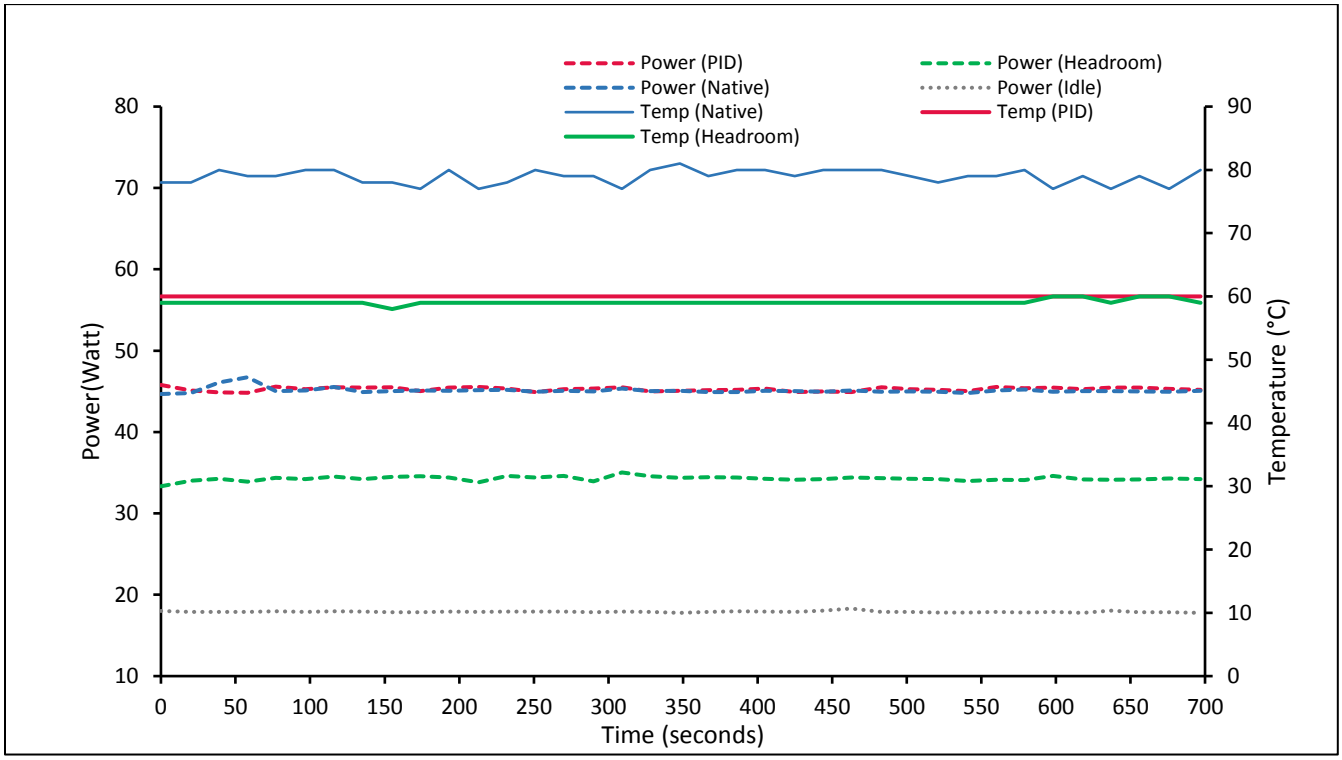


Fig. 3.6.13: Temperature and Power vs Time for Group13

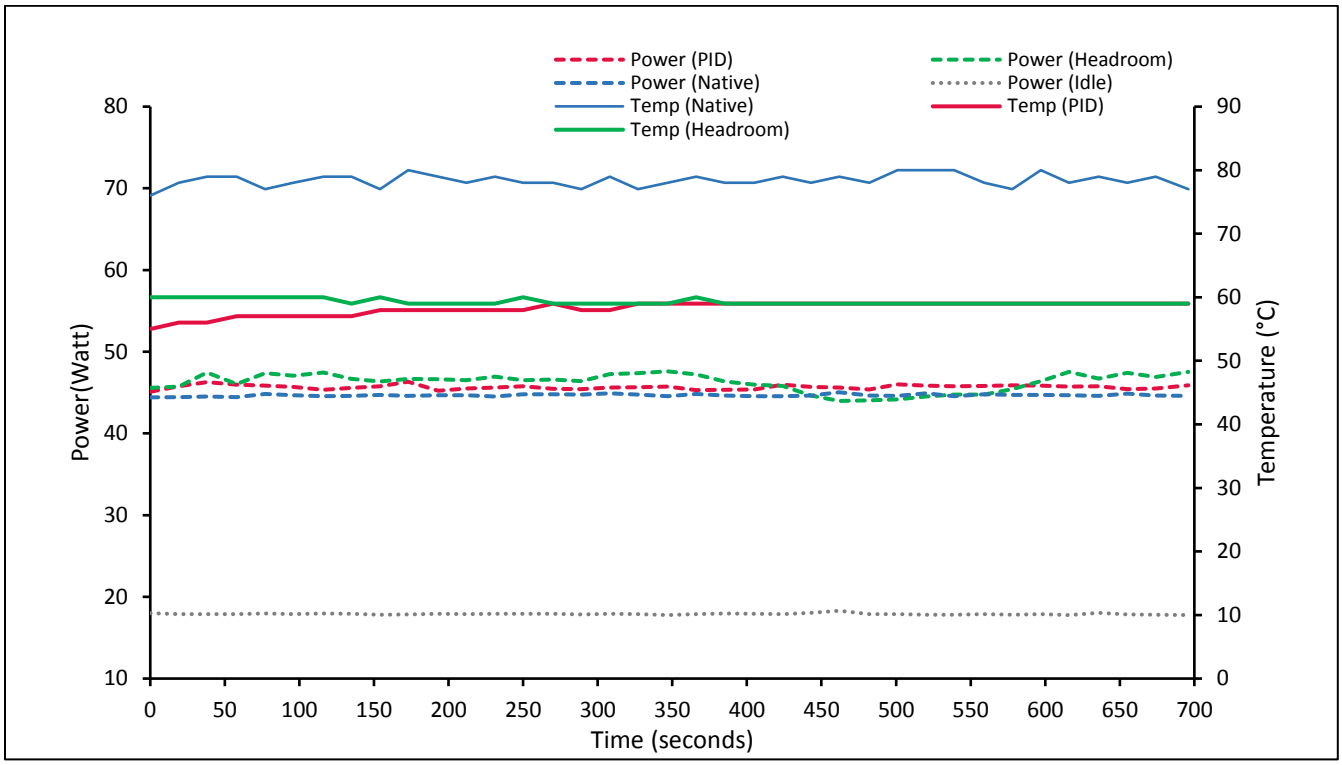


Fig. 3.6.14: Temperature and Power vs Time for Group14

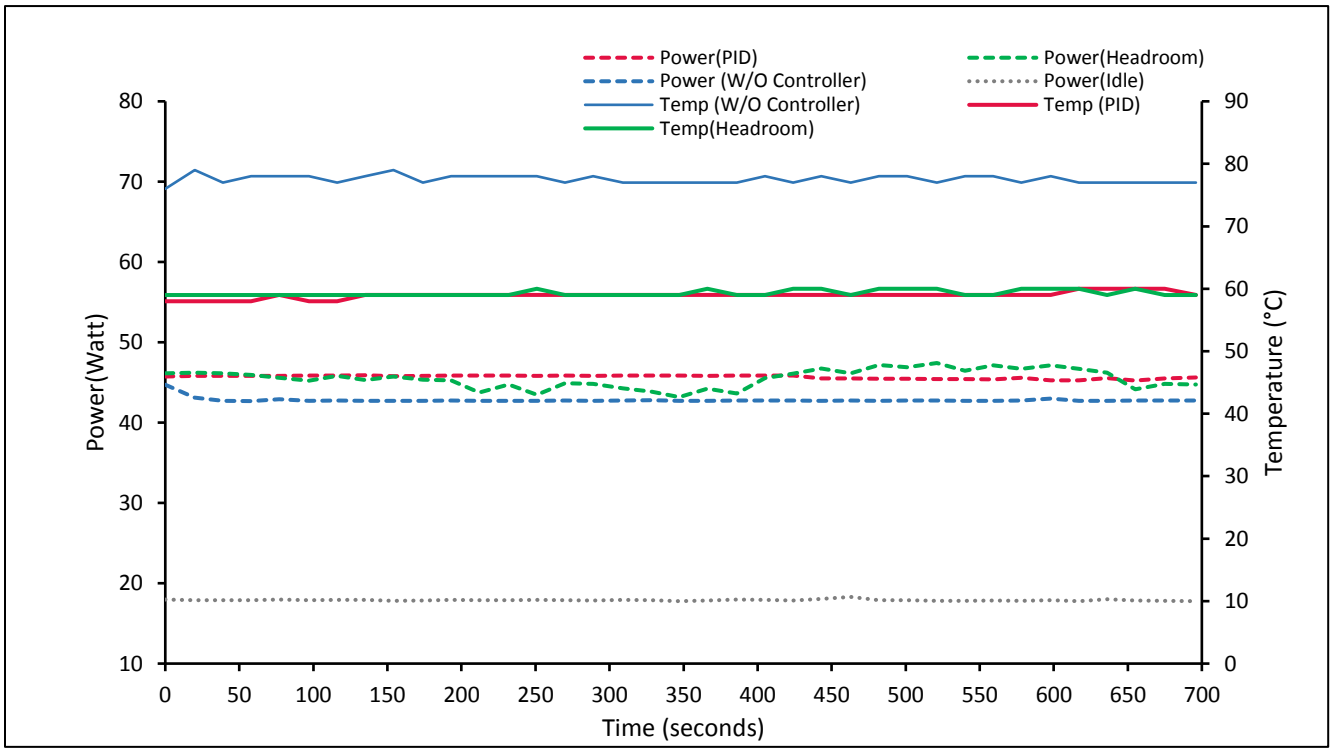


Fig. 3.6.15: Temperature and Power vs Time for Group15

Chapter 4

Discussion

4.1 Peak Temperature Analysis

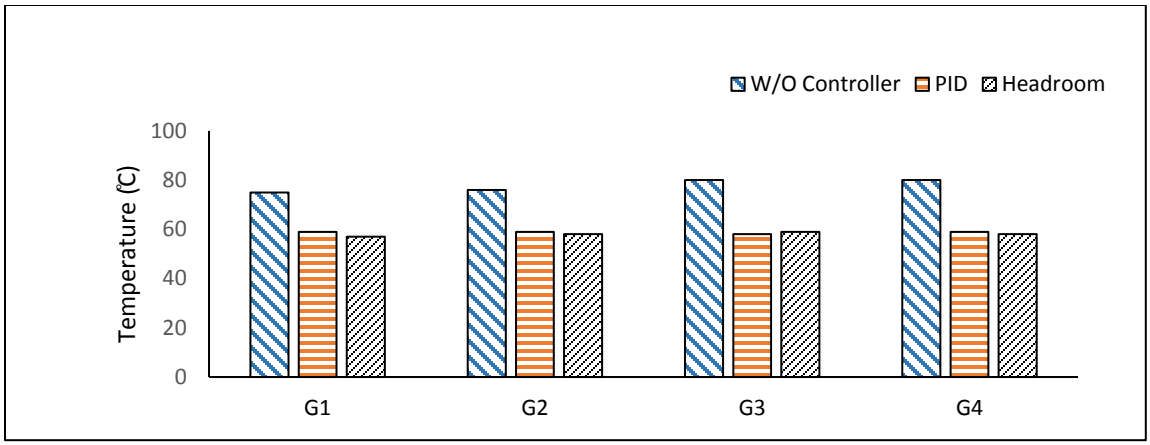
Figures 4.1a, b, and c represent the peak temperatures observed when certain group of benchmarks are run at target temperatures 56˚C, 60˚C and 63˚C respectively in the three different scenarios as described earlier. For each target temperature, the average peak temperature was calculated by taking an average of set of peak temperatures recorded for groups within the scenario. At target temperature 56˚C, the average peak temperature observed for Headroom, PID and Native were 58˚C, 58.75˚C and 77.75˚C. The average peak temperatures observed at target temperature 60˚C were 59.2˚C, 62.8˚C and 82˚C. Lastly, the average peak temperatures observed at target temperature 63˚C were 63˚C, 63.7˚C and 80.5˚C. Headroom approach has performed well against PID based controller, we can see that the reductions in peak temperatures range between 0.75˚C – 3.6˚C. The peak temperatures in Native scenario are relatively pretty higher than PID and Headroom approaches.

4.2 Energy Consumption Analysis

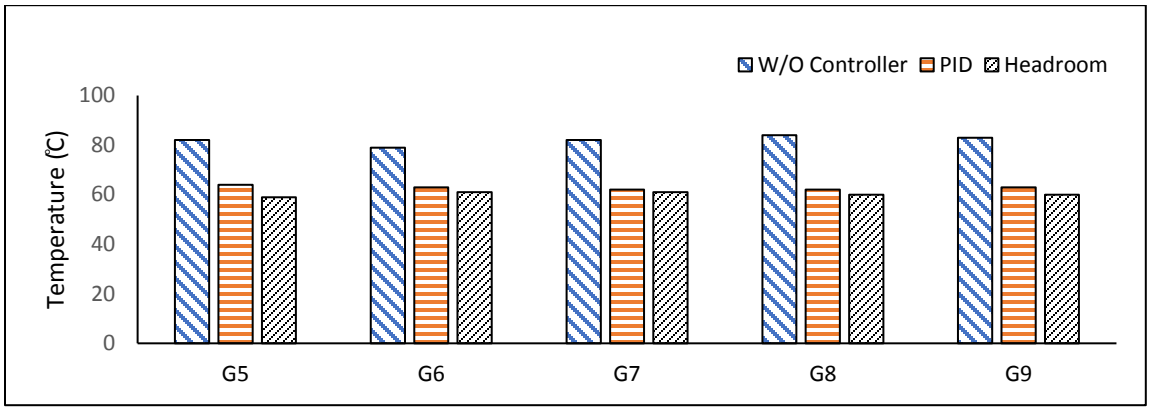
Higher temperatures result in higher leakage current, this is explained by the phenomenon known as Frenkel-Poole Emission. When random thermal fluctuations occur, they give enough energy for the electrons to move out of their localized state into conduction band which results in power leakage. The standard quantitative expression for Frenkel-Poole Emission is:

$$J \propto Eexp\left(\frac{-q(\emptyset_B - \sqrt{qE/(\pi\epsilon)})}{k_B T}\right)$$

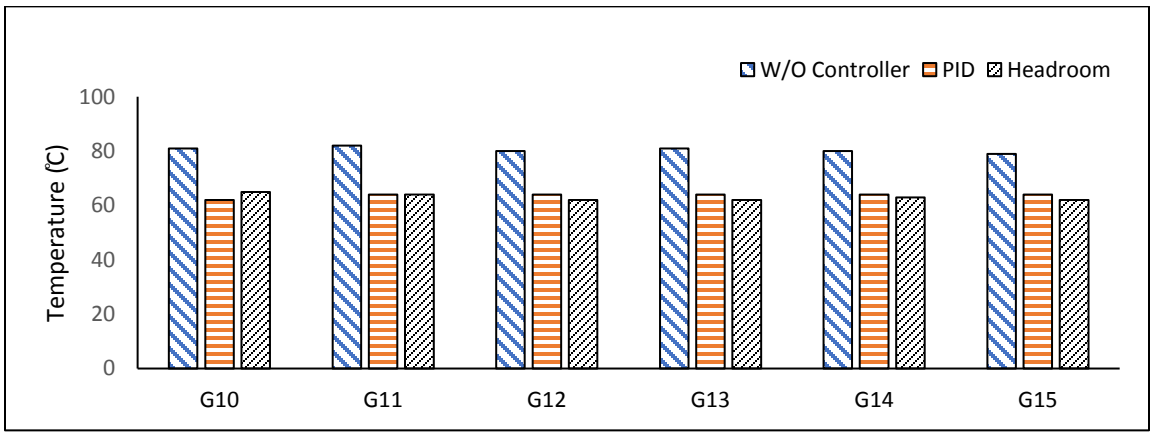
where, J is the current density, E is the applied electric field, q is the elementary charge, $\emptyset_B$ is the voltage barrier, ϵ is the dynamic permittivity, k_B is Boltzmann's constant and T is the temperature. This equation is reproduced from [19].



(a) Peak temperatures at target 56˚C



(b) Peak temperatures at target 60˚C



(c) Peak temperatures at target 63˚C

Fig. 4.1: Peak Temperature Analysis

In the current experiment the applied electric field will be same as we are using the same electronic system. The current density is primarily effected by temperature (*J α 1/T).*

The idle state for the system is defined as the duration when there is no user application or benchmark being run. Power readings have been recorded for the three scenarios in each run. Energy consumption for various groups at different target temperatures, was computed by calculating the integral area under the power graphs for these groups, shown in Figure 4.2.

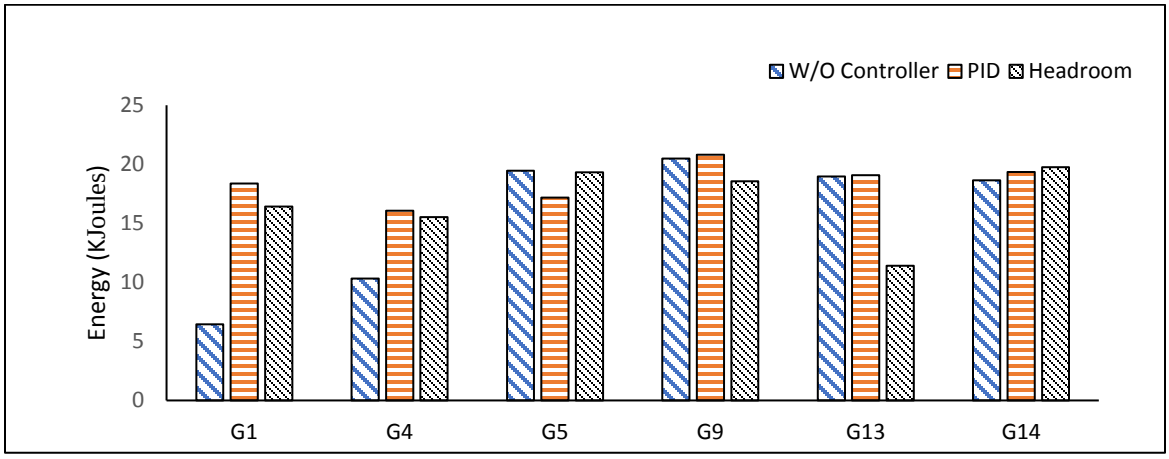


Fig. 4.2: Energy Consumption

The time marked at the point where the system drops into idle state serves as the upper time limit for calculating actual energy consumption. This calculation can be expressed as:

$$E(t) = \int P(t)\partial t$$

$$E_{actual} = E_{total} - E_{idle}$$

As mentioned earlier Figures in section 3.7 show the results at target temperatures 63° C, 60° C and 56° C. We observe that the power readings for PID are close to Native scenario, but the headroom approach as compared to PID consumes 6 KJ less energy in case of G13.

We observe a sudden dip and rise in power for both G5 and G9 benchmarks, as the processor briefly goes into idle power state for a duration of 10-15 seconds. The power readings drop to idle power once the execution of benchmark ends. Energy savings of 1.5 KJ and 1.9KJ were observed for headroom approach against PID for G1 and G9 respectively. In case of G4 the energy consumption of headroom was less than PID by 0.5 KJ, on the other hand no energy savings were made in case of G5 and G14.

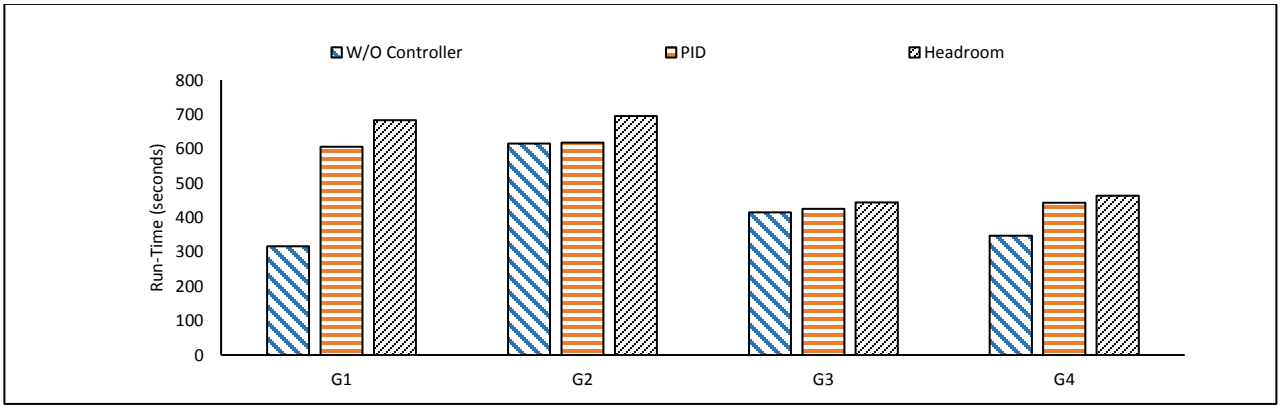
4.3 Run Time Analysis

At target temperature 56° C, Headroom approach when compared to PID controller an average run-time penalty of 9.02% has been observed. At target temperature 60° C, Headroom approach when compared to PID controller an average run-time penalty of 4.3% has been observed. At target temperature 63° C, Headroom approach when compared to PID controller an average run-time penalty of 6% has been observed. Figure 4.3 shows the run times for different groups.

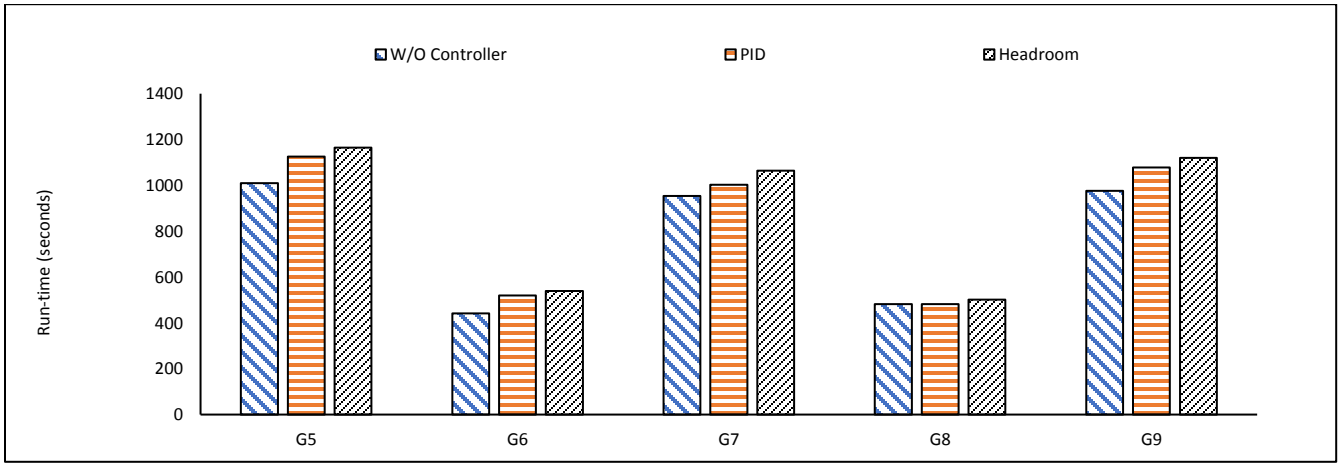
4.4 Reliability Analysis

The Mean Time Between Failure (MTBF) is a widely used metric to express reliability of an approach. In the current scenario we are trying to stabilize the processor temperature around a target temperature. The frequent violations of thermal constraint due to sudden increase in temperature beyond the threshold impose prospective failures on the thermal protection algorithm.

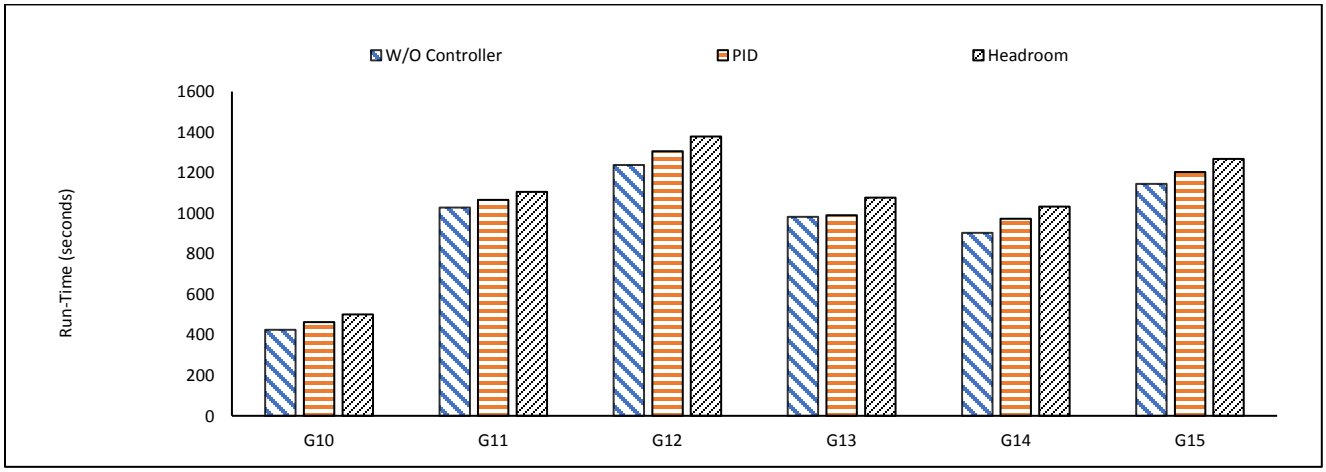
Failure rate (λ) is the frequency with which an engineered approach fails, expressed in failures per unit of time. In special processes called renewal processes, where the time to recover from failure can be neglected and the likelihood of failure remains constant with respect to time, the failure rate is simply the multiplicative inverse of the MTBF ($1/\lambda$). Reliability is defined as a measure of module or an approach to perform its intended function under specified conditions for a specified period of time. Software failures denoted as principal failures in this scenario, can be defined as an incorrect operation of protection caused by a mistake in the planning or design or setting or application of the



(a) Run-Times at target 56˚C



(b) Run-Times at target 60˚C



(c) Run-Times at target 63˚C

Fig. 7: Run-Time Analysis

protection and control mechanism according to International Electro technical vocabulary [20]. Therefore, MTBF can be expressed in terms of failures as:

$$MTBF = 1/\lambda$$

$$\text{MTBF} = \frac{\sum(\text{start of downtime} - \text{start of uptime})}{Number\ of\ failures}$$

The reliability analysis framework applied in the case can be modeled using the Probabilistic Relational Model (PRM) shown in Figure 4.4.1, reproduced from [20]. The benefit of using PRMs for reliability analysis of systems is the ability to both apply the probabilistic reasoning offered by Bayesian networks together with architecture models.

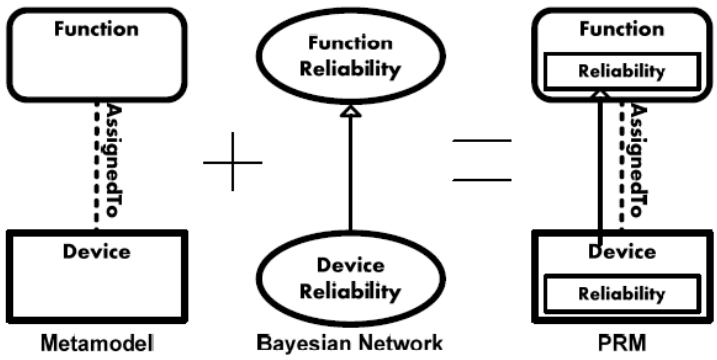


Fig. 4.4.1: Reliability Analysis Framework

Exponential Failure Distribution can be used to depict the reliability of the system. The failure density function in an exponential distribution can be calculated as shown in the following equation, reproduced from [20].

$$F(t) = 1 - e^{-\lambda t}$$

where, λ is the failure rate, t is time, e is natural log base ($\lambda > 0$, $t > 0$).

We have run both the PID and Thermal headroom approach for a duration of 700 seconds each, and every time the temperature was above the target temperature it was considered

as a failure of the algorithm. We have computed failure rates for the thermal protection functions provided through the PID and Thermal Headroom approaches, which reflects the reliability of the algorithms being used. MTBF was computed as the mean of durations of the failures. Failure rate (λ) was then computed as the inverse of the MTBF value calculated. Each graph in Figure 4.4.2 demonstrates the density of failures observed for each algorithm within the duration of 700 seconds in the three scenarios.

The failure rates (λ) for PID vs Thermal Headroom approach at the target temperatures 56˚ C, 60˚C and 63˚C are shown in Table 2. We can observe that PID algorithm has a higher failure rate than Thermal Headroom algorithm in all the cases. Failure rate is a fundamental in determining the safety and reliability of the approach. Lower failure rates in case of Thermal headroom approach as compared to PID help us to conclude that the headroom algorithm is a more reliable thermal management strategy due to fewer number of thermal violations.

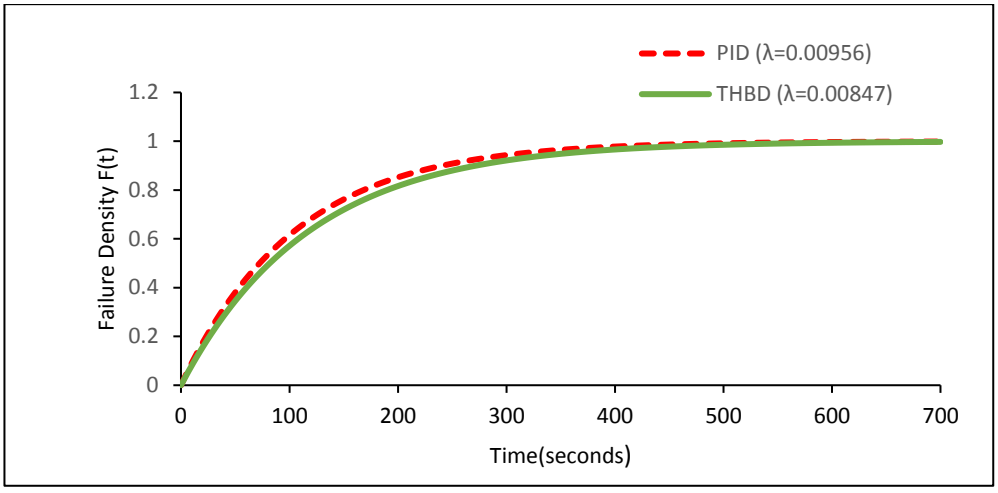
Target Temp	56˚C	60˚C	63˚C
PID	0.00956	0.00652	0.00930
Thermal Headroom	0.00847	0.00526	0.00723

Table 2: Failure rates (λ) for PID and Thermal Headroom

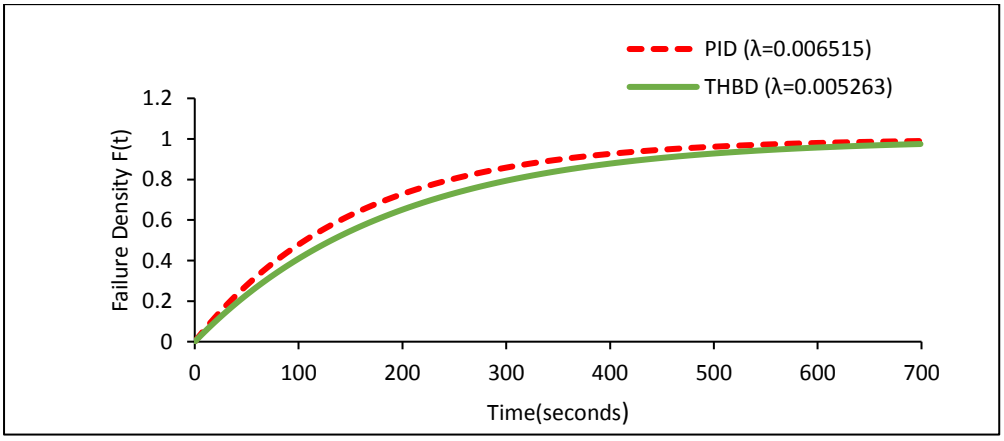
4.5 Conclusion

In this paper, we investigated the trade-off among peak temperatures, power consumption and reliability of a new Thermal Head Room based P-State driver. We could significantly reduce temperature overshoots beyond set-point, as a result we could achieve energy conservation and higher reliability. The prime goal for this research was to analyze Thermald (PID based approach) for temperature violations beyond reference temperature and propose a new mechanism for reducing violations under a thermal constraint in Linux

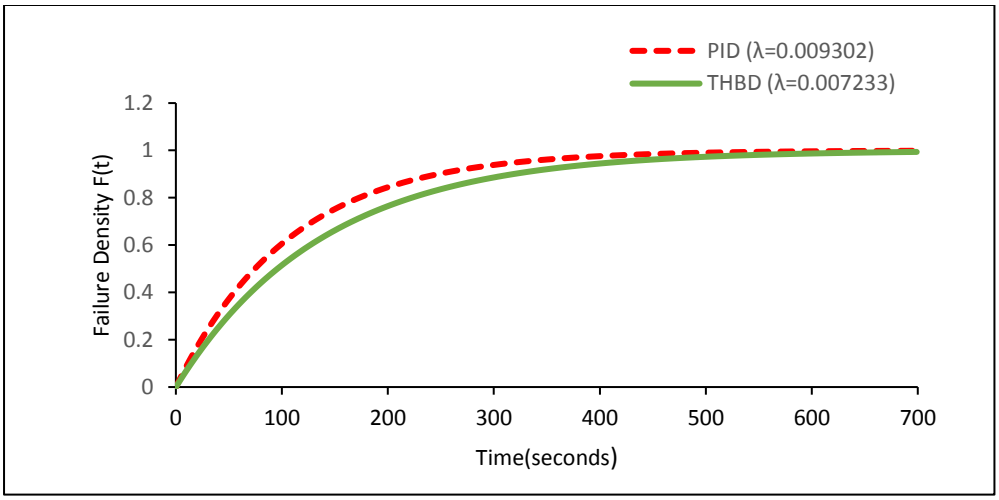
kernel. We have conducted an extensive architectural study of the Head room based thermal driver. This approach was implemented within Linux kernel and thoroughly evaluated. In future this approach can be extended to High Performance Computing perspective in distributed processing systems.



(a) Failure Density at target 56˚C



(b) Failure Density at target 60˚C



(c) Failure Density at target 63˚C

Fig. 4.4.2: Reliability Analysis

References

[1] L. Yeh and R. Chu, "Thermal management of microelectronic equipment: Heat transfer theory," *Analysis Methods, and Design Practices (ASME),* 2002.

[2] S. Gunther, F. Binns, D. M. Carmean and J. C. Hall, "Managing the impact of increasing microprocessor power consumption," *Intel Technology Journal,* vol. 5, no. 1, pp. 1-9, 2001.

[3] H. F. Hamann, A. Weger, J. A. Lacey, P. B. Z. Hu, E. Cohen and W. J., "Hotspot-limited microprocessors: Direct temperature and power distribution measurements," *IEEE Journal of Solid-State Circuits,* vol. 42, no. 1, pp. 56-65, 2007.

[4] C. Y. Lee, S. Yang and R. Chang, "Thermal-aware scheduling collaborating with OS and architecture," in *42nd International Conference on Parallel Processing (ICPP),* Lyon,France, 2013.

[5] S. Pandruvada, "Linux thermal daemon," Intel, [Online]. Available: https://01.org/linux-thermal-daemon/documentation/introductionthermal-daemon.

[6] S. Baskiyar and R. Abdel-Kader, "Energy aware dag scheduling on heterogeneous systems," *Cluster Computing,* vol. 13, no. 4, pp. 373-383, 2010.

[7] S. Baskiyar and K. K. Palli, "Low power scheduling of dags to minimize finish times," *High Performance Computing-HiPC,* vol. 10, no. 1, pp. 353-362, 2006.

[8] "Intel and core i7 (Nehalem) dynamic power management.," Arizona State University, 2011. [Online]. Available: https://impact.asu.edu/cse591sp11/Nahelempm.pdf.

[9] Y. Fu, N. Kottenstette, Y. Chen, C. Lu, X. D. Koutsoukos and H. Wang, "Feedback thermal control for real-time systems," *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* pp. 111-120, 2010.

[10] S. Zhang and K. S. Chatha, "Approximation algorithm for the temperature-aware scheduling problem," in *10th IEEE/ACM Proceedings of the international conference on Computer-aided design*, Beijing, China, 2007.

[11] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger and P. Bose, "Thermal-aware task scheduling at the system software level," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design (ISPLED)*, New York, NY, 2007.

[12] J. Yue, T. Zhang, B. Q. Y. Liu and C. Tianzhou, "Thermal-aware feedback control scheduling for soft real-time systems," in *IEEE 9th International Conference on Embedded Software and Systems (HPCCICESS), IEEE 14th International Conference on High Performance*, Liverpool, United Kingdom, June 2012.

[13] R. Jayaseelan and T. Mitra, "Temperature aware scheduling for embedded processors," in *22nd International Conference on VLSI Design*, New Delhi, India, 2009.

[14] O. Lempel, "2nd generation intel core processor family: Intel core i7, i5," Intel, 2011. [Online]. Available: http://download.intel.com/newsroom/.

[15] D. Li, H. Chang, H. K. Pyla and K. W. Cameron, "System-level, thermal-aware, fully-loaded process scheduling," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.

[16] D. Rajan and P. S. Yu, "“On temperature-aware scheduling for single processor systems," in *14th International Conference on High Performance Computing (HiPC)*, Goa,India, 2007.

[17] White Paper, Intel, "Enhanced Intel speedstep technology for the Intel pentium-m processor," 2004. [Online]. Available: http://download.intel.com/design/network/papers/30117401.pdf.

[18] R. Uppu, "A novel CPU P-State driver for better thermal control with improved power/performance trade-off," Auburn University, 14 December 2015. [Online]. Available: https://etd.auburn.edu/handle/10415/4985.

[19] Wikipedia, "Poole–Frenkel effect," [Online]. Available: https://en.wikipedia.org/wiki/Poole%E2%80%93Frenkel_effect.

[20] J. König and L. Nordström, "Reliability Analysis of Substation Automation System Functions," in *Reliability and Maintainability Symposium (RAMS), 2012 Proceedings - Annual*, 2012.

[21] M. Finkelstein, "Failure Rate Modelling for Reliability and Risk," *Springer Series in Reliability Engineering*, pp. 1-84, 2008.