Politecnico di Milano - Department of Electronics, Information and Bioengineering

Implementation of a regulator in linux kernel

[Coding Project]

Student Luca Santini
ID 883026

Student Riccardo Remigio
ID 874939

Course Embedded Systems
Academic Year 2017-2018

Advisor Federico Terraneo
Professor William Fornaciari

April 23, 2018

Contents

1 Introduction

The very fast growth of electronic technology has lead to the creation of more and more powerful processors. Over the last few years new processors were developed capable of reaching very high clock rates with, consequently, a very quick increase of the temperature, that must be dealt with. The dynamic power dissipated per unit of time by a chip is described by this formula

$$P = CV^2Af \tag{1}$$

where C is the capacitance being switched per clock cycle, V is the voltage, A is the activity factor indicating the average number of switching events undergone by the transistors in the chip (as a unitless quantity) and f is the switching frequency.
Therefore, voltage is the main determinant of power usage and heating. The voltage required for stable operation is determined by the frequency at which the circuit is clocked, and can be reduced if the frequency is also reduced. Dynamic voltage scaling is a related power conservation technique that is often used in conjunction with frequency scaling (DVFS), as the frequency that a chip may run at is related to the operating voltage. Thus we chose to try to solve the problem of the increase in CPU temperature by properly reducing the clock frequency and so exploiting the DVFS to reduce also the voltage and hence the temperature.

1.1 Problem statement

The aim of our project is to implement a regulator in the linux kernel that keeps the temperature at a set point by adjusting the frequency of the cpu.
Therefore, our regulator is unrelated to power consumption and it does not tune the voltage, that is regulated automatically by the DVFS (Dynamic Voltage and Frequency Scaling) depending on the frequency we select.

1.2 Summary of the work

In order to write a linux kernel module properly we divided our work in five steps.

1. Our first goal was to write a simple kernel module able to print a message in the standard kernel output.

2. The second step was to figure out by looking at Intel documentation which MSR(Model Specific Register) were related to frequency and temperature.

3. Then we analyzed the actual system to find and disable all the other software modules/programs that modify the cpu frequency.

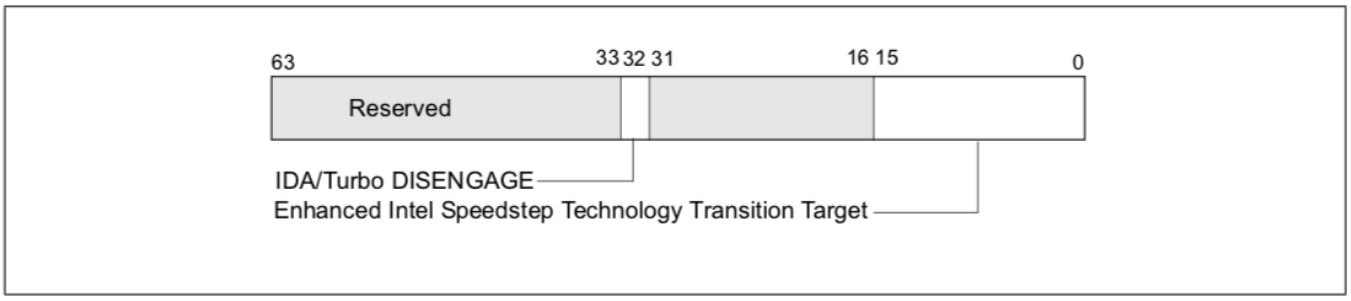
4. After this last step had been completed the new goal was to implement the regulator in our module.

5. In the end we measured the regulator performance monitoring the cpu temperature in a situation of very high workload. We set the regulator parameters according to our measurement.

2 Design and implementation

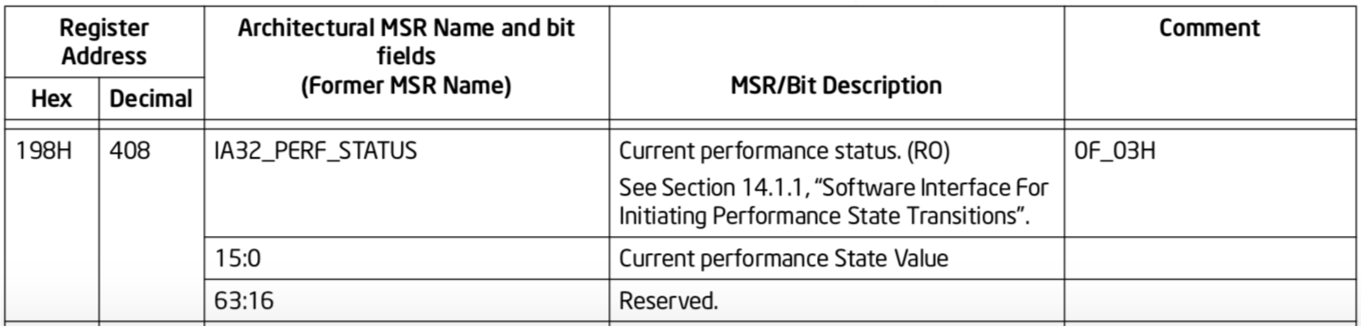
2.1 Intel MSR for Power and Thermal Management

The processor we used for our project is an Intel i7-3630QM of 3rd generation (Ivy Bridge) so we checked the public documentation "Intel 64 and IA-32 Architectures Software Developer's Manual", in particular the most useful in our case were volumes 3b and 4. Intel provides a technology to control the frequency of the processor via software: this technology is called Intel SpeedStep. The technology allows the management of processor power consumption via performance state transitions. These states are defined as discrete operating points associated with different voltages and frequencies. The user can start a performance state transition by writing a 16-bit value to the IA32_PERF_CTL register. If a transition is already in progress, the transition to the new value will subsequently take effect.



IA32_PERF_CTL 0x199 is used to switch between cpu's frequencies

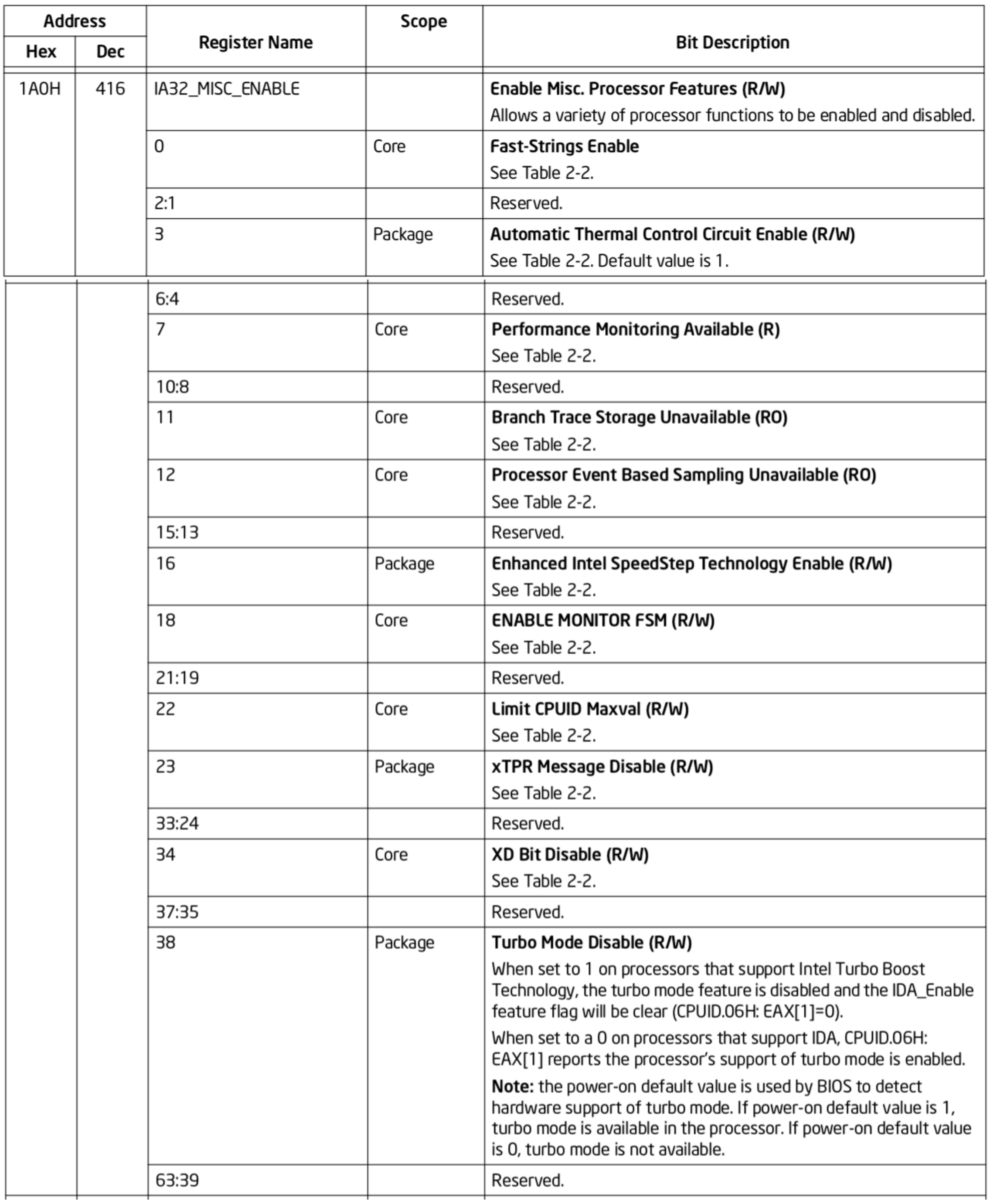
After making some experiments and according to information from unofficial sources we were able to state that, in our processor, the 16-bits related to the performance state were composed of: 8-bits always set to 0 and 8-bits that represent the value of the frequency multiplier (base frequency = 100 mhz). When a value is written in the frequency multiplier the cpu starts a transition that takes effect in a very small interval of time, while the voltage is automatically adapted by the DVFS. It is possible to check the current performance state value by reading the first 16-bit of IA32_PERF_STATUS (0x198).



Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	**Decimal**			
198H	408	IA32_PERF_STATUS	Current performance status. (RO) See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Current performance State Value	
		63:16	Reserved.	

IA32_PERF_STATUS 0x198 is used to read the actual frequency

We used Intel SpeedStep technology and wrote on IA32_PERF_CTL to change the frequency multiplier. All this can be done only if SpeedStep is enabled. There is a register called IA32_MISC_ENABLE (0x1a0) that has a bit used to enable/disable SpeedStep. This is bit 16, that is set to 1 when SpeedStep is enabled and to 0 when it is disabled. Other important features, for our purpose, controlled by this register are the Thermal Control Circuit (TCC) and Intel Turbo Boost Technology. If TCC is set the processor is allowed to automatically reduce power consumption when the junction temperature is almost reached. The Intel Turbo Boost Technology can provide opportunistic performance greater than the performance level corresponding to the processor base frequency of the processor. Since the activation of this technology takes place in a non-deterministic way, we do not want it to activate.

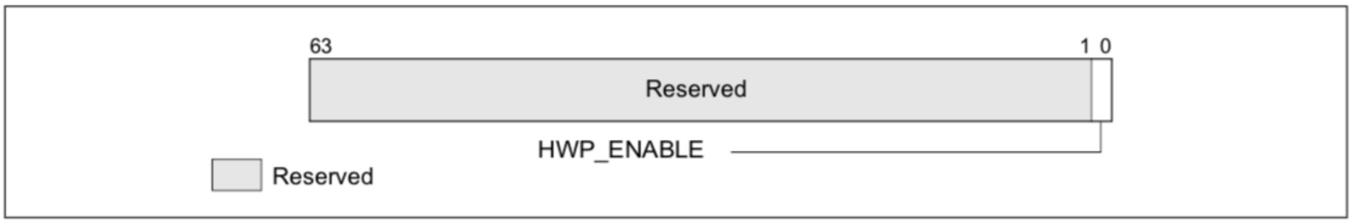


Address		Register Name	Scope	Bit Description
Hex	Dec			
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Core	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Package	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 1.
		6:4		Reserved.
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Package	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.

IA32_MISC_ENABLE 0x1a0 is used to check and enable some different features

There is also another feature that is liable to interfere in a very considerable way with our purpose: the Hardware Controlled Performance State (HWP). According to the documentation "Intel processors may contain support for Hardware-Controlled Performance States (HWP), which autonomously selects performance states while utilizing OS supplied performance guidance hints. The Enhanced Intel SpeedStep Technology provides a means for the OS to control and monitor discrete frequency-based

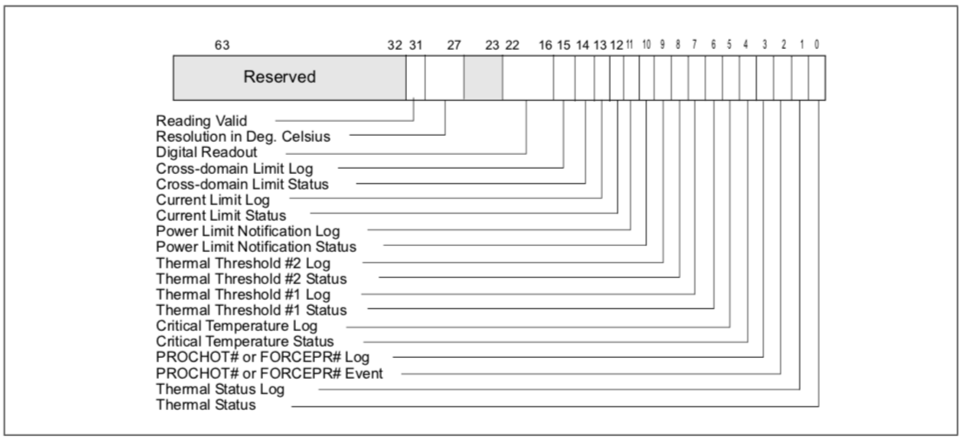
operating points via the IA32_PERF_CTL and IA32_PERF_STATUS MSRs.
When HWP is enabled, the processor autonomously selects performance states as deemed appropriate for the applied workload and with consideration of constraining hints that are programmed by the OS." There is a way to enable HWP and to check whether it is already enabled, but there is not a way to disable HWP if it has already been activated. The register to check is called IA32_PM_ENABLE (0x770), here we report more details taken from the documentation.



IA32_PM_ENABLE 0x770 is used to check and enable HWP

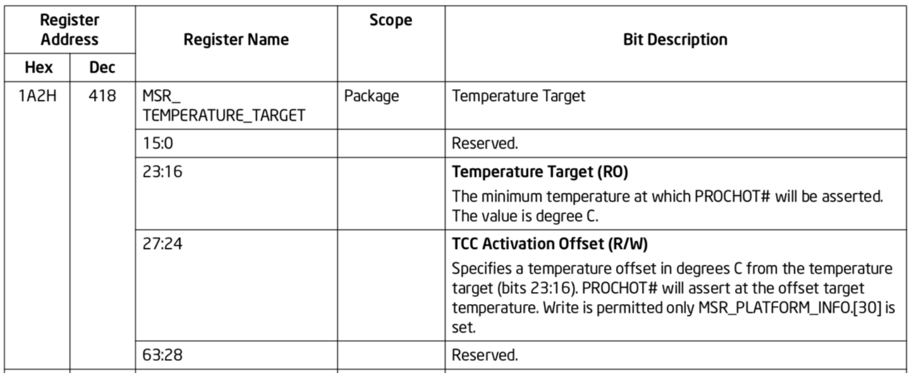
Then we needed to know the current temperature of the processor, that is the variable we had to control with the regulator. Its value is obtained by checking the register called IA32_THERM_STATUS (0x19c) and reading bits 22:16, representing the digital readout. Digital readout is the difference between junction temperature and current temperature in celsius degree, so we could easily calculate the current temperature from the digital readout:

$$T_{current} = T_{Junction} - T_{Readout} \tag{2}$$



IA32_THERM_STATUS 0x19c is used to check the digital readout

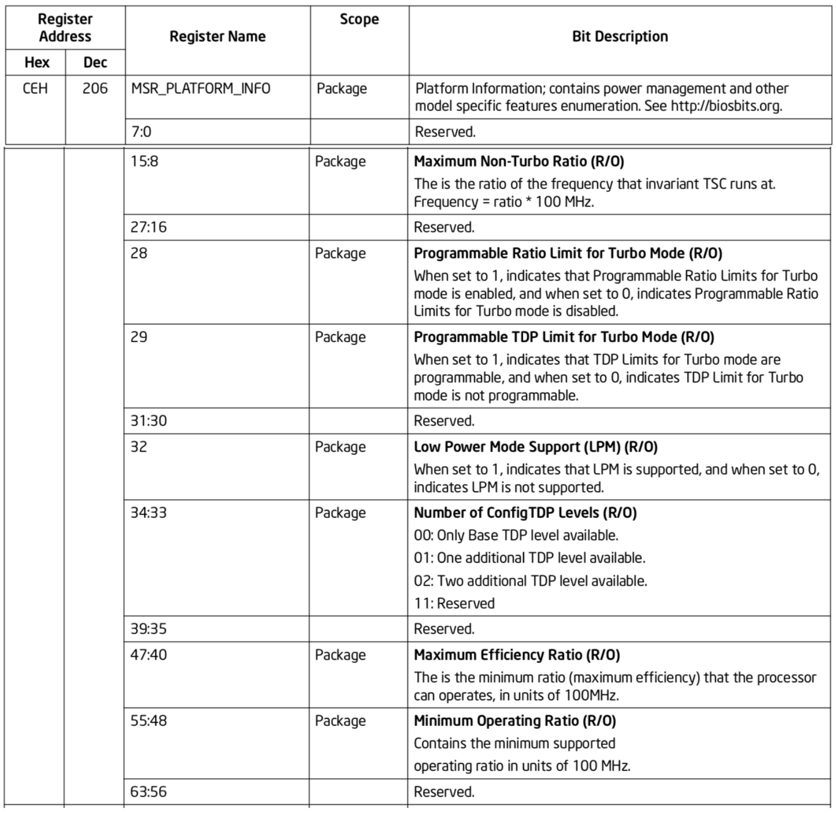
Therefore, we needed to retrieve also the specific junction temperature of our processor. We were able to find this information on the register MSR_TEMPERATURE_TARGET (0x1a2) by reading bits 23:16.



Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		27:24		**TCC Activation Offset (R/W)** Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved.

MSR_TEMPERATURE_TARGET 0x1a2 is used to check the junction temperature

There were more data that we needed to know in order to properly control the system, one of them was the range of frequency that our processor supported without TurboBoost enabled. Fortunately there is a register that collect some platform information including this: this register is called MSR_PLATFORM_INFO (0xce). By reading bits 8:15 we obtained the maximum frequency multiplier without turbo and by reading bits 48:55 we found out the minimum frequency multiplier supported by our processor.



Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.

MSR_PLATFORM_INFO 0xce is used to check minimum and maximum frequency multiplier supported

2.2 Disabling all the modules that interfere with frequency

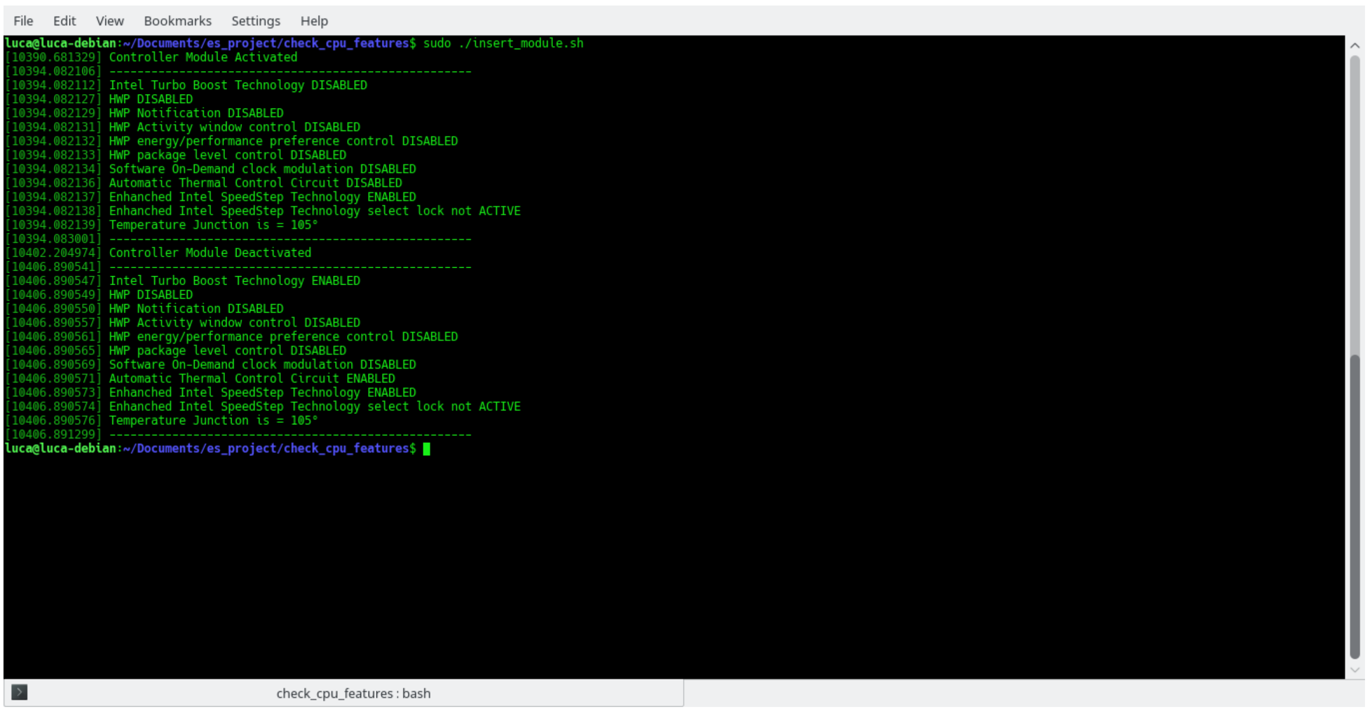
As we previously stated, before starting to implement our regulator we had to analyze the system and look for all the modules that interfered with our purpose, that is all the modules which write to IA32_PERF_CTL. First, we disabled intel_pstate by adding a kernel parameter to the boot "intel_pstate=disable". In our case we added this line to /etc/default/grub. There are also some other modules that are often enabled by default, therefore we wrote a bash script to quickly enable and disable all this modules:

```
#!/bin/bash

#Uncomment the following line if you have thermald enabled
#sudo systemctl stop thermald
#Uncomment the following line if you does not have msr module enabled
#sudo modprobe msr

cpupower frequency-set -g userspace
sudo rmmod -f intel_rapl
sudo rmmod -f intel_rapl_perf
sudo rmmod -f x86_pkg_temp_thermal
sudo rmmod -f intel_powerclamp
sudo rmmod -f acpi_cpufreq
```

At that point, we had disabled some modules but we could not be sure whether other modules that write to the msr register IA32_PERF_CTL existed, and that could change the frequency without our knowledge. To solve this problem we adopted an empirical method: we wrote a script that exploited a linux system function that, if properly set, writes a log of all the msr_write that target the IA32_PERF_CTL. Therefore, our script monitors all the writing that takes place in an interval of time chosen by the user, and then displays the result.

```
#!/bin/bash
# use sudo to execute this script

read -p 'Choose the interval tracing time: ' interv
echo
echo > /sys/kernel/debug/tracing/trace
echo 1 > /sys/kernel/debug/tracing/events/msr/write_msr/enable
sleep $interv
cat /sys/kernel/debug/tracing/trace | grep 'write_msr: 199'
echo 0 > /sys/kernel/debug/tracing/events/msr/write_msr/enable
```

At that point we were almost sure that no other software writing was performed in our system, so we had to ensure that there was no hardware control on the cpu frequency. As explained before we had to ensure that HWP was not enabled in our system. This step was of capital importance because once it is enabled there is no way to disable it until the cpu resets. The only way of disabling HWP is by setting the specific option in the BIOS, if it is present, and by passing the "intel_pstate=no_hwp" to the kernel in the command line before the start. As for TurboBoost, TCC and SpeedStep, the can easily be enabled by writing, as we said before, the appropriate values in IA32_MISC_ENABLE, therefore our controller module took care of them. To simplify and speed the process of checking the active features of the processor up we wrote a kernel module that checked, by using the command "cpuid", all the necessary features. We have added a simple script that inserted and removed the module and then displayed the result by using the command "dmesg". From the following figure we can notice that by executing "check cpu features module" when "controller module" is insert TCC and Turbo are disabled, then when we remove "controller module", Turbo and TCC are re-enabled and HWP stays disabled.

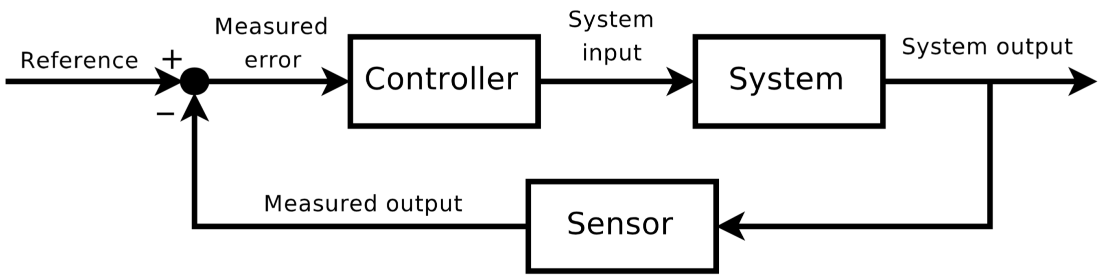


```
luca@luca-debian:~/Documents/es_project/check_cpu_features$ sudo ./insert_module.sh
[10390.681329] Controller Module Activated
[10394.082106] ------------------------------------------------------
[10394.082112] Intel Turbo Boost Technology DISABLED
[10394.082127] HWP DISABLED
[10394.082129] HWP Notification DISABLED
[10394.082131] HWP Activity window control DISABLED
[10394.082132] HWP energy/performance preference control DISABLED
[10394.082133] HWP package level control DISABLED
[10394.082134] Software On-Demand clock modulation DISABLED
[10394.082136] Automatic Thermal Control Circuit DISABLED
[10394.082137] Enhanched Intel SpeedStep Technology ENABLED
[10394.082138] Enhanched Intel SpeedStep Technology select lock not ACTIVE
[10394.082139] Temperature Junction is = 105°
[10394.083001] ------------------------------------------------------
[10402.204974] Controller Module Deactivated
[10406.890541] ------------------------------------------------------
[10406.890547] Intel Turbo Boost Technology ENABLED
[10406.890549] HWP DISABLED
[10406.890550] HWP Notification DISABLED
[10406.890557] HWP Activity window control DISABLED
[10406.890561] HWP energy/performance preference control DISABLED
[10406.890565] HWP package level control DISABLED
[10406.890569] Software On-Demand clock modulation DISABLED
[10406.890571] Automatic Thermal Control Circuit ENABLED
[10406.890573] Enhanched Intel SpeedStep Technology ENABLED
[10406.890574] Enhanched Intel SpeedStep Technology select lock not ACTIVE
[10406.890576] Temperature Junction is = 105°
[10406.891299] ------------------------------------------------------
luca@luca-debian:~/Documents/es_project/check_cpu_features$
```

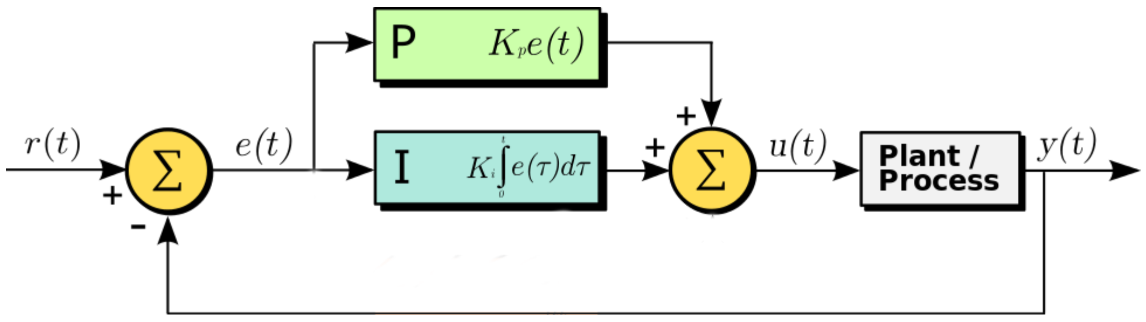
Check CPU Features while controller module is inserted and after it has been removed

2.3 PI regulator

The aim of the project was to implement a feedback controller, in particular a PI regulator that by modifying the CPU frequency is able to keep the temperature as close as possible to the set point, or at least allows to exceed it only by a small amount.



Feedback Controller Scheme



PI Regulator Scheme

General transfer function of a PI regulator

$$u(t) = k_p e(t) + K_i \int_0^t e(t')\,\mathrm{d}t' \tag{3}$$

Our specific transfer function of the process and the regulator

$$P(s) = \frac{10}{1+0.03s} \qquad R(s) = \alpha\frac{1+0.03s}{s} \tag{4}$$

Open-loop transfer function

$$L(s) = P(s)R(s) = \frac{10\alpha}{s} \tag{5}$$

Then we were able to use scilab to plot a step response simulation of the controlled system with an alpha value of 10 and check whether the time response was acceptable.

```
clear;
clc;

P=10/(1+0.03%s);
a=10;
R=a (1+0.03 %s)/%s;
t=0:0.001:1;

u=ones(t);
u(1)=0;

y=csim(u,t,tf2ss(P/(1+R P)));
plot(t,y);
```

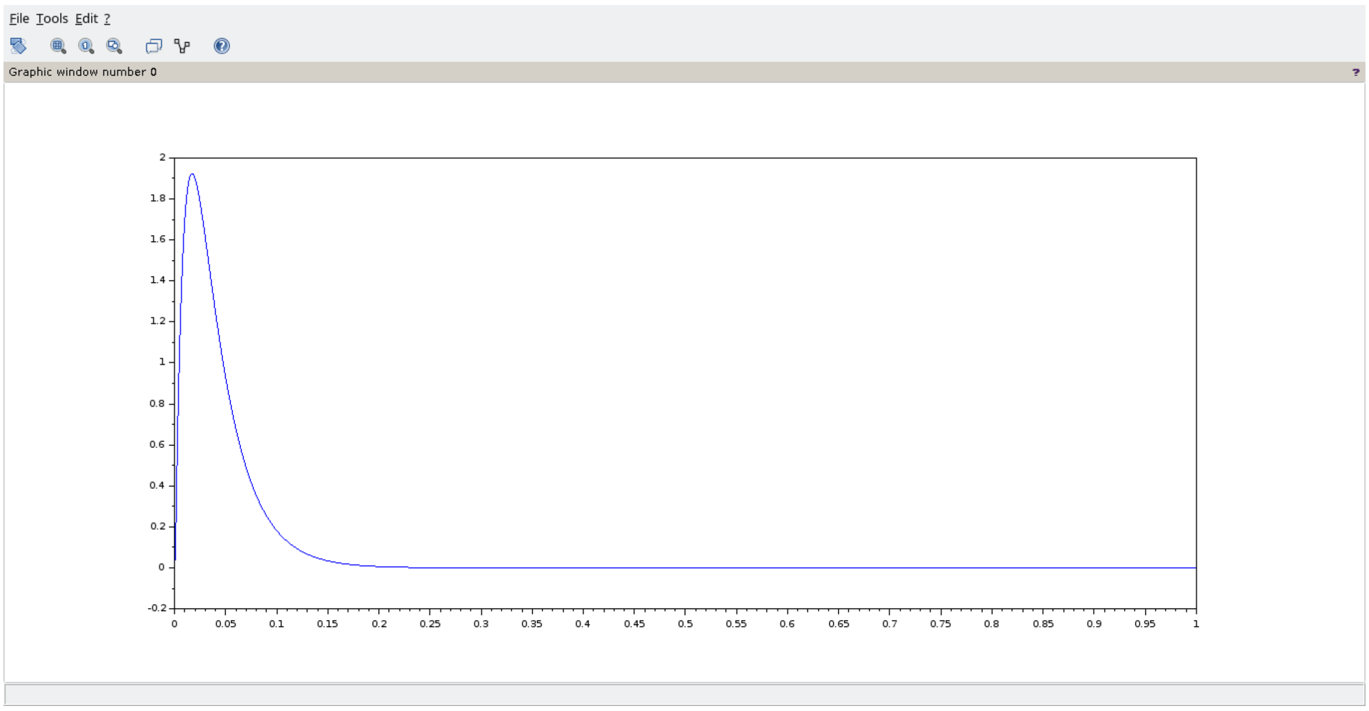


Figure 1: Step Response of the System with alpha = 10

The Figure 1 shows that after 17 milliseconds the system reaches the maximum value of 1.924, and after 100 milliseconds the value is almost come back to 0, precisely is 0.1811. By applying a discretization method with a sampling time of 0.005 we were able to obtain the discretized equation of the controller.

$$T_{sampling} = 0.005 \qquad R(z) = \frac{7\alpha z - 6\alpha}{200(z-1)} \tag{6}$$

2.4 Implementation of the regulator in the kernel module

Once we had the discrete transfer function of the controller, we obtained the state space representation and isolated the output variable of the regulator

$$u(k) = u(k-1) + \frac{7\alpha}{200}e(k) - \frac{6\alpha}{200}e(k-1) \tag{7}$$

Then we began to implement this formula in the our kernel module by following these steps:

Listing 1: pseudocode floating point implementation

```
float e_old=0,u_old=0;
while(thread_stop){
   float e = set_point − get_temp();
   float u = u_old + (7a/200) e−(6a/200)e_old;
   u=max(0,min(1,u));
   u_old=u;
   e_old=e;
   set_frequency(min_freq+u (max_freq−min_freq));
}
```

In order to implement this code in a kernel module we had to convert it in a code with integer values only, because in kernel there are no floating point variables. So this is the conversion.

Listing 2: pseudocode integer implementation

```
int e_old=0,u_old=0;
while(thread_stop){
   float e = 200    set_point − get_temp();
   float u = u_old + 7 a e−6 a e_old;
   u=max(0,min(200 (max_freq−min_freq),u));
   u_old=u;
   e_old=e;
   set_frequency(min_freq+u/200);
}
```

In the real implementation of this code, at the end of the cycle, we placed a function called usleep that keeps in sleep state the thread for a specific range of time. In our case we chose 5 milliseconds, because we wanted the controller to act at that specific rate. We also verified that the delay is precise by looking at the timestamps in the log file. For the implementation of the get_temperature() and set_frequency() we used the already built functions (present in the linux kernel) rdmsr and wrmsr that respectively reads and writes to an MSR. As we mentioned before the register from which we read the temperature was IA32_THERM_STATUS and the register in which we wrote the frequency multiplier was IA32_PERF_CTL.

2.5 Logging

In our situation it was not possible to use the printk function of the linux kernel to log the useful information to understand the controller behavior, because its execution time was greater than the execution rate of the controller so it would have slowed down the controller response time. Our solution to this problem was to use the trace_printk function, that is designed to act in a tenth of microsecond and is also used in interrupt code. In our implementation we can enable the logging by setting a boolean parameter, when trace_printk is enabled it writes in the system linux file /sys/kernel/debug/tracing/trace.

3 Experimental evaluation

After the phases of design and implementation, the last step was to tune the controller parameter in order to obtain the best and the fastest response to temperature changes. The goal was to make so that high frequencies are used when the temperature is lower than the set point and adapt the frequencies (avoiding to keep them to the minimum) when the temperature is near the set point. We will show this trend in the next chapters, using also some graphs.

3.1 Experimental setup

First, we looked for a way of putting the cpu under stress in order to raise its temperature. We achieved this using an Intel benchmark called "Linpack", that stresses the cpu by resolving a large number of linear equations and so giving an high workload to the cpu. In the first try alpha parameter was set to 10. After disabling all the modules that interfere with the cpu frequency and cleaning the trace log, we inserted our module and started the benchmark. To be consistent throughout the different tests we tried to run the benchmark for the same interval of time, so we decided to first run our controller module, after 5 seconds run the benchmark for 45 seconds and finally remove the module after 10 seconds of idle state. This way in all the tests we obtained 60 seconds of log. We repeated the test changing the alpha parameter to 5 and later to 1 trying to understand which value was the best one for our controller. In order to tune correctly the parameter we had to visualize the trend of the temperature and of the frequency of the cores collected in the trace file for each test, so we wrote a python script able to plot the data contained in the trace file. The script collected the data of the temperature and the actual frequency of all the cores and plot aligned by time. This way we were able to observe how the controller changed the frequency based on the changes in the temperature. The script also allowed to sample the data in a range larger than the time of execution of the controller in order to have a cleaner graph. It also printed the mean frequency and the maximum temperature reached to see whether the controller worked properly.

3.2 Results

At the end of our tests we obtained three graphs that show the different trends of our controller with different values of the alpha parameter. In all the three graphs we can see that there were some moments, during the benchmark, in which the temperature went down. These moments occurred more or less at the same point in all the plots, so we can speculate that this trend was caused by the benchmark, probably when it assigned a new problem to solve. Furthermore in the third graph we can see that the points at which the temperature goes down is a bit earlier than in the others. This can be explained by the fact that the average frequency of cpu with the controller with alpha equal to 1 is higher than the others, so in this case the cpu solved the problem earlier.

The first graph in Figure 2 shows the trend of the temperature and frequency of all the cores with the alpha parameter set to 10. As we can notice from the plot, the controller response to temperature changes is very quick, but because of this, frequencies change too much, even when the temperature is far from the set point or temperature changes are very small. With these parameters, temperature is always kept near the set point (under workload) but performance is going to be poor, because the frequency will be set often to the minimum. In the lower graph, that of frequency, we can see that most of the time the level fluctuates between the maximum and the minimum frequency. Through the python script we observed that during the test the mean frequency multiplier was around 18.

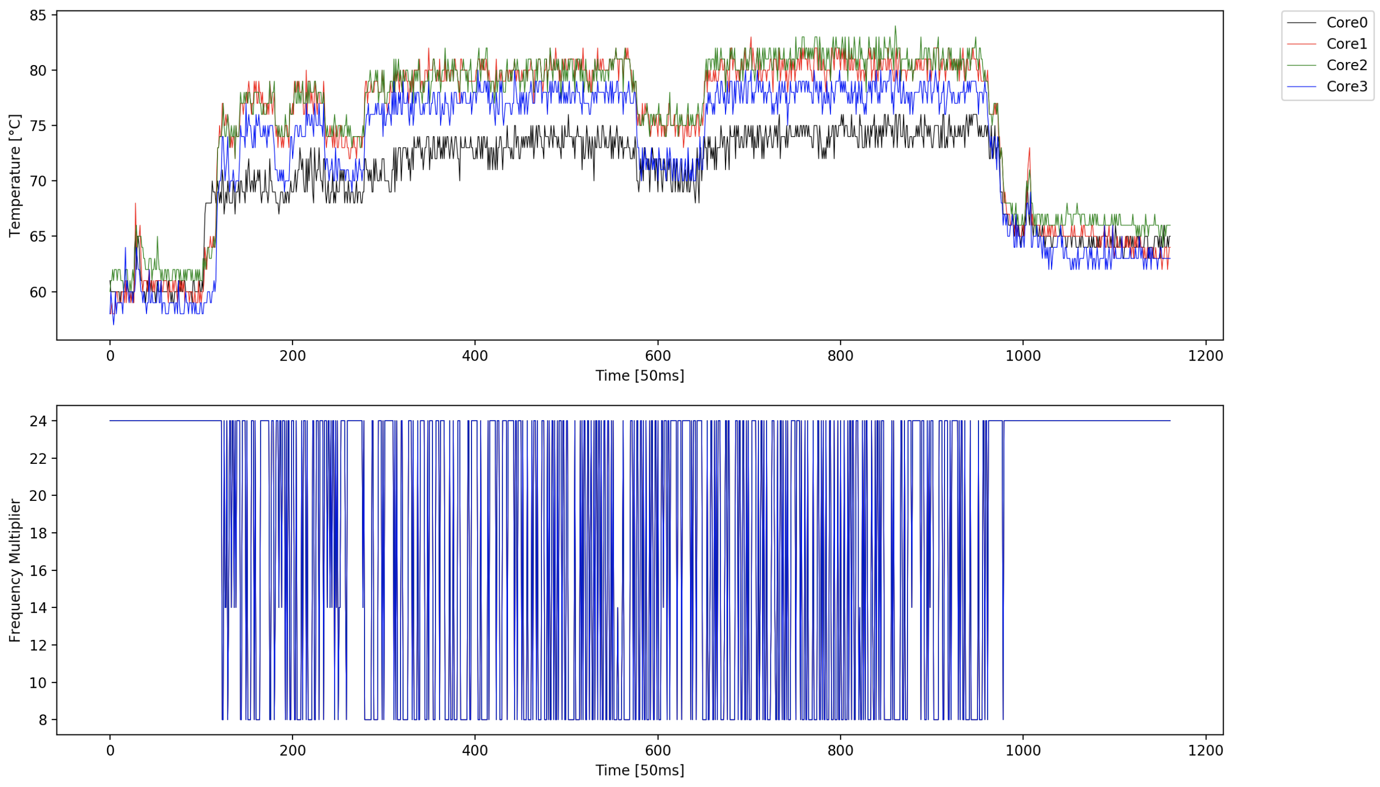


Figure 2: Plotted data of 1 minute of test with alpha = 10

Figure 3 shows the trend with the alpha parameter set to 5. In this case we can notice that there is little difference with the previous graph with the alpha parameter set to 10. Here too we have a fast response but also a lot of fluctuation between the maximum and minimum frequencies and in this test too we obtained a mean frequency of around 18. So we can assert that the performance with these parameters will be similar. We can see that the trend is very similar also in the points of temperature drop described before.

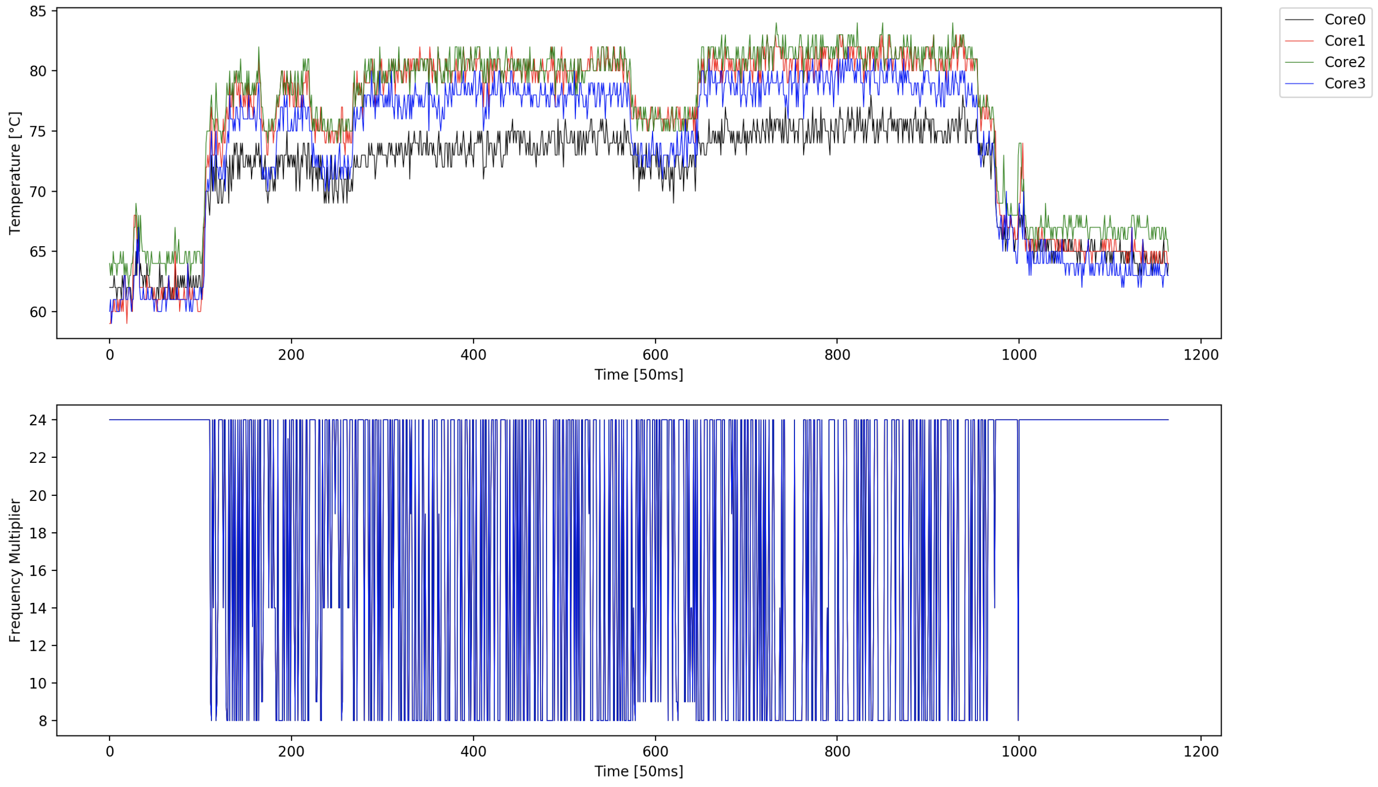


Figure 3: Plotted data of 1 minute of test with alpha = 5

In the last test, the alpha parameter was set to 1 and the graph in Figure 4 shows its trend. In this case we can see that as for the temperature the behavior is similar to the other controllers, so it has a good response and the maximum temperature reached in all the tests is more or less the same. The difference between this controller and the others is mainly how it changes the frequency in response to temperature changes. From the graph we can see that the frequency oscillates in a smaller range compared to the other controllers and this lead to a better performance with a mean frequency of 20. This can be noticed in the graph at the points (explained at the beginning of this chapter) during the benchmark when the temperature goes down. In this controller that trend takes place earlier compared to the other tests, so we can deduce that the benchmark solves the problem earlier.

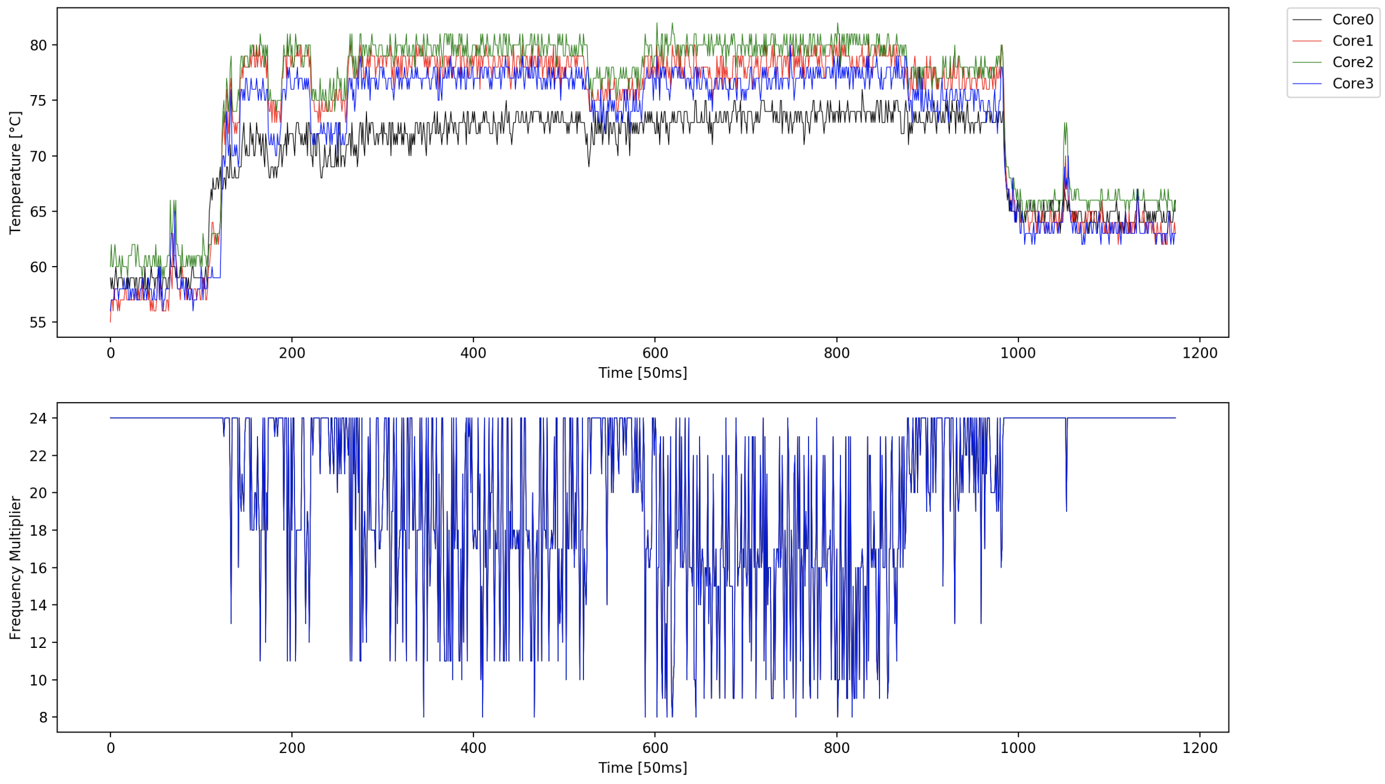


Figure 4: Plotted data of 1 minute of test with alpha = 1

4 Conclusions and Future Works

In conclusion we can state that the best alpha value for our process is 1 because it maintains the temperature around the set point like the other controllers but it is able to keep an higher mean frequency with less peaks towards the minimum frequency. This controller, however, is not intended for achieving low power consumption but it only aims to keep the highest possible frequency while preserving the temperature below a certain threshold. Therefore, a future goal can be to design a controller that takes also care of saving energy when is not under workload.

References

[1] John C Mathes and Dwight W Stevenson. Designing technical reports: Writing for audiences in organizations. 1976.

[2] Carolyn R Miller. A humanistic rationale for technical writing. *College English*, 40(6):610–617, 1979.

[3] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. Volumes 3b, 4. December 2017.