ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

Cool Linux

SS 2013

Jan Bernegger
and
Gino Brunner

Advisors:	Lars Schor
	Pratyush Kumar
Professor:	Prof. Dr. Lothar Thiele

Zurich
15th July 2013

Abstract

As CPU performance and power consumption of computing systems increase, so does the need for better cooling systems, in order to ensure the reliability and prevent thermal breakdown of the system. However, employing ever more capable cooling systems adds to the bulk and complexity of the system, and worst of all, it is very expensive. A key challenge in research is therefore to find alternative ways of effectively keeping large computer systems cool.

In this thesis, a framework called *Cool Linux* is developed, which allows the execution of a CPU intensive task on any Linux distribution, while keeping the CPU temperature below a certain temperature threshold. This is achieved by using three different *Dynamic Thermal Management* (DTM) techniques (frequency scaling, task migration, scheduled sleep). The framework uses a predictive control algorithm that bases its decisions on the current CPU temperatures and a prediction of the future temperature development, based on the current CPU load. The entire framework is implemented in the*user space*, and therefore independent of special features in the software stack.

Finally, the framework has been evaluated in various case studies. First, the performance is evaluated for different frequencies of the predictive controller, thus finding optimal values for the controller speed. Then, the overall performance of the framework and the accuracy of the temperature predictions are evaluated. Furthermore, the feasibility of the individual *DTM*-methods is determined. Finally, *Cool Linux* is compared to the default *on-demand* frequency governor of Linux. *Cool Linux* manages to significantly decrease the average CPU temperature under full load (up to $20°C$ on a HP laptop), at a minimum performance loss. For some cases (medium CPU load), the framework does not only lower the temperature, but also increases the performance of the evaluated application.

Acknowledgements

We would like to express our gratitude to Prof. Dr. Lothar Thiele for the opportunity to write this semester thesis in his research group.

We also want to thank our supervisors Lars Schor and Pratyush Kumar for the great collaboration that made this thesis possible.

Finally, we would like to thank everyone that assisted us during this semester thesis, either mentally or technically.

Contents

List of Figures

1 Introduction

1.1 Motivation

The demand for more computation power leads to more complex processor designs and integration. One of the consequences is a high power consumption within a small spatial region. This leads to hot processors, which is bad for mainly two reasons. First, it decreases the reliability of the system. Second, at higher temperatures, the power consumption is higher, leading to a vicious cycle that can cause a thermal breakdown. Reducing on-chip temperatures is widely recognised as a key challenge in the design of future computing systems.

Several solutions are being explored to tackle the above challenge. Foremost is the use of cooling solutions ranging from conventional heat sinks to expensive liquid cooling subsystems. Such solutions are expensive and add to the bulk of the system. The second solution is temperature-aware low-level VLSI design. This solution has been adopted for a several decades now. However, it has limitations, especially, as we move towards ever smaller transistor sizes.

A third solution is becoming increasingly crucial; making the software more aware of the temperature of the processor. There have been several works which have explored this solution, such as reducing the frequency of the processor, migration of tasks to cooler parts of the system and delaying the arrival of incoming jobs, among others. Broadly, these techniques are referred to as *Dynamic Thermal Management* (DTM). However, these techniques have not yet been combined into a single framework.

In this thesis a framework called *Cool Linux* (henceforth referred to as *CL*) is developed. *CL* uses three known DTM techniques and integrates them into one framework to be run on an off-the-shelf computer with any Linux distribution. The DTM techniques will be implemented in the *user space*, thus not relying on special features in the software stack, and therefore ensuring maximum portability. The *CL*-framework bases its DTM-decisions on the current temperatures and on predictions of the future temperature development of the CPU-cores. The goal of the *Cool Linux*-framework is to execute a CPU-intensive task on a Linux system while keeping the CPU temperature below a certain threshold. This temperature threshold is freely configurable and it enables the use of less effective and less expensive cooling systems, while still guaranteeing a certain level of performance. Our approach is entirely software based, and therefore cheap to implement.

1.2 Contributions

Temperature Predictions: A program is developed that allows to predict the temperature of the CPU cores based on the current load, without need for manual configuration. The prediction-accuracy is evaluated for different scenarios.

Integration of DTM-Techniques: The feasibility of three different DTM techniques (frequency scaling, task migration, sleeping) is evaluated.

Configurable Framework: Temperature predictions and DTM techniques are combined into a single framework that allows to execute a CPU-intensive task in a cool manner. The effectiveness of the framework is evaluated for different scenarios. The results are compared to the tests done without the framework, or with certain DTM-parts switched off, respectively. In addition, we compare our framework to the default Linux *on demand* frequency governor.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2 shows how the temperatures of the CPU are predicted in the framework. First, it explains how the CPU-core temperatures can be accessed. The second part describes how the temperature development is modelled, which is then analyzed in the third part. The last section of this chapter explains the temperature prediction mechanism.

Chapter 3 introduces the *dynamic thermal management*-techniques that are used in the framework. First, it gives a short overview of power- and thermal management technologies implemented in modern Intel CPUs, and explains how these mechanisms differ from our approach. Then it explains how the CPU frequency can be changed on a Linux system, and what the implications on performance and temperature are. After that, a brief explanation of task migration is given. Finally, it describes how the framework is able to block, i.e. put to sleep, a worker thread.

Chapter 4 presents the final framework. It gives an overview of the specifications/parameters, the overall architecture and features of the framework. It then goes into more detail on the separate parts of the framework and explains how exactly it manages to keep the CPU temperature below a certain temperature threshold.

The evaluation of the *CL* framework is done in Chapter 5. First comes a description of the two different test systems. Then, tests are performed for different controller-speeds and temperature tresholds. Afterwards, different *DTM*-parts of the framework are switched off and the performance of the "remainin" framework is evaluated. Finally, the performance of the framework is compared to the default Linux *ondemand* frequency governor.

The evaluation of the *Cool Linux* framework is presented in Chapter 5. It describes the different experiments that were conducted, and shows their results.

Chapter 6 brings the thesis to a conclusion and presents possible directions for future improvements.

2 CPU Temperature Prediction

This chapter focuses on the requirements and necessary steps to develop a temperature predicting algorithm for CPU temperatures. First, section 2.1 explains the approach used to tackle the problem. Section 2.2 discusses how to obtain current CPU core temperatures. Next, Section 2.3 explains the model used for temperature development and in the last section, the topic of temperature development is discussed.

2.1 Used Approach

The used approach is based on an exponential temperature model (see Section 2.3) for the CPU cores temperatures. The model depends on the current temperatures of all cores as well as on the time difference to the moment for which the prediction should be made. Also, the model is based on constant parameters, explained in Section 2.3.4. In order to consider the *neighbouring effect* (see Section 2.4.2) in the model, the final predictions are composed of the influences of each stressed core to the core for which the predictions are made, using an instance of the model with its own parameters for every stressed-predicted core combination.

The predicting algorithm is based on current temperature values only and takes neither a history of temperatures nor the history of taken measures to reduce the core temperatures into account. It assumes a constant environment, for example steady room temperature and that no other programs are running on the framework's target machines.

Due to limitations of the library used (discussed in Section 2.2.1), the resolution of the temperature readings is 1°C, as the temperature values are returned as Integers.

2.2 Accessing CPU Core Temperatures

Obtaining the values of the temperature sensors of a processor may seem to be a trivial task, but the variety of different CPUs and chipsets is so vast, that it gets complicated heavily, for every chipset needs to be accessed differently. Therefore, it is essential to find a convenient way to access the temperatures regardless of the device's chipset.

Therefore, an existing library was integrated in our solution to solve the problem. The next Sections describe which library was used and what problems were encountered and how they were solved.

2.2.1 lm-sensors Library

`lm-sensors` is an open-source library which provides access to temperatures, voltages, and fans [2]. In this project, it is used to access the individual CPU core's temperature sensors and the general CPU temperature sensor if available.

After installation (described in Appendix B.1.2), an initialisation script has to be run once to detect the device's chipset.

```
sudo yes | sensors-detect
```

The command is also required for checking the compability of the computer with the `coretemp` kernel module, which is required for accessing the CPUs individual core temperatures. After running this script, the library is functional.

2.2.2 Frequency of Temperature Updates

While developing the interface to access the temperatures of the CPU, the temperatures happened to be updating at a very slow rate of approximately once every 1.5 seconds. In terms of degrees this meant that the temperature could jump five to ten degrees in one resolution step. As this uncertainty is too big for this thesis' purpose, the sensor update frequency had to be increased to once every 10ms.

The official `lm-sensors` FAQ[1] states the following on that topic:

> **Q: I read sensor values several times a second, but they are only updated only each second or so. Why?**
> A: If we would read the registers more often, it would not find the time to update them. So we only update our readings once each 1.5 seconds (the actual delay is chip-specific; for some chips, it may not be needed at all).

Analysing the source code of `libsensors` showed that the temperature sensors are in fact accessed through the `sysfs` interface[2], which is a virtual file system provided by Linux to access device and driver information in userspace. Accessing `sysfs` directly by running the following line of `bash` showed the same temperature behaviour as accessing the temperatures through `lm-sensors`.

```
cd /sys/class/hwmon/hwmon1/device/driver/coretemp.0
while [ 1 ]; do cat temp1_input ; done
```

Therefore, the change of frequency had to be limited in the Linux kernel `coretemp` module. A slight modification of the module's source code results in a drastically higher resolution, which is enough for this project's purposes (it has been tested down to a period of 10ms).

The procedure to modify, compile, and install the coretemp module can be found in Appendix C.

2.2.3 CoreTemperatures Class

A wrapper class has been developed to have easy access to the cores individual temperatures. At initialisation of an object of the class, the constructor automatically calls the required `lm-sensors` functions to get the number of cores and to prepare the library for sensor data reading.

A more detailed description of the interface can be found in Appendix D.1.

2.3 Temperature Model

To be able to approximate the temperatures of individual cores, an appropriate temperature model is needed. Due to the physical nature of a modern

[1] `http://www.lm-sensors.org/wiki/FAQ/Chapter3#Ireadsensorvaluesseveraltimesasecondbuttheyareonlyupdatedon`
`Why`
[2] `http://en.wikipedia.org/wiki/Sysfs`

CPU and heat propagation, the temperature of a core rises very fast at the beginning, but takes increasingly more time to get to its maximum. Therefore, the temperature curve is approximated by an exponential function.

2.3.1 Rising Temperature Model

For the approximation of the temperature of a single core, the following equation is used:

$$T(t + t_{diff}) = T(t) + (T_\infty - T(t)) * (1 - \exp(-\frac{t_{diff}}{\tau_i})) \tag{2.1}$$

An example of the model can be found in Figure 2.1. The parameter T_∞ represents the maximum temperature the model can attain for $t \to \infty$ (represented by the green line in Fig. 2.1) and the parameter τ_i is a time constant for the speed with which the exponential curve rises.

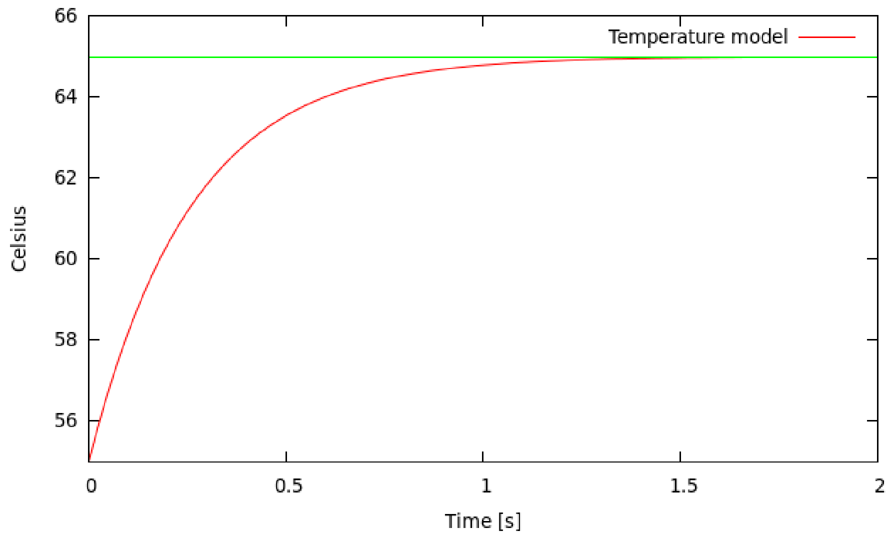


Figure 2.1: Graph of the temperature model.

To further calculate the influence of other stressed cores on the measured one (see Section 2.4.2), the model was extended to the following equation:

$$T_i(t+t_{diff}) = T_i(t) + \sum_{j \in \text{stressed cores}} (T_{\infty_{ij}} - T_i(t)) * (1 - \exp(-\frac{t_{diff}}{\tau_{ij}})) \tag{2.2}$$

Now the model for the predicted temperature does not only depends on the temperature and parameters of the core for which the prediction should be made, but rather on all cores which are stressed during t_{diff}. This approximation is used to include the `Neighbouring effect` into the model and is based on the linearity of the temperatures.

2.3.2 Falling Temperature

Similar to temperature increase, it is possible to calculate the cooling down of a core using the same constants calculated for the model in the previous section and an additional constant T_0. The curve converges to this very constant, which represents the idle temperature of the core:

$$T(t + t_{diff}) = T(t) - (T(t) - T_0) * (1 - \exp(-\frac{t_{diff}}{\tau_i})) \tag{2.3}$$

An example of the cooling down curve can be seen in Figure 2.2, T_0 is represented by the green line.

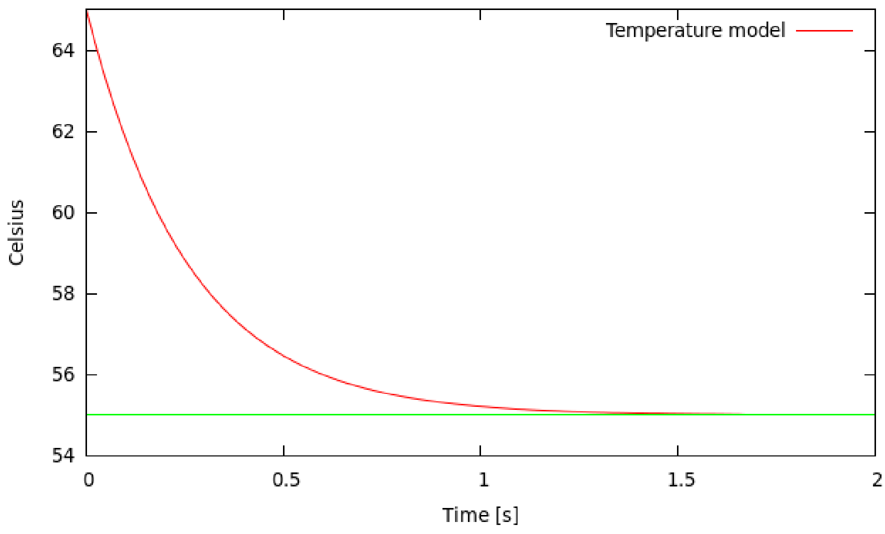


Figure 2.2: Graph of the cooling down temperature model.

2.3.3 Temperature development when coming from a higher frequency

When changing from a hot temperature and a high frequency to a lower frequency, where the actual temperature exceeds T_∞ of the lower frequency and core, the resulting temperature curve for the next t_{diff} looks very similar to the normal cooling down temperature curve. However, instead of T_0, the curve converges to the T_∞ of the lower frequency model.

$$T(t + t_{diff}) = T(t) - (T(t) - T_\infty) * (1 - \exp{(-\frac{t_{diff}}{\tau_i})}) \tag{2.4}$$

Using this equation, the predictor is able to calculate the temperature of a stressed core even if it is currently hotter than T_∞. However, this formula can only be used for one stressed core at a time and would have to be extended for multiple concurrent tasks.

2.3.4 Temperature Model Constants

For all equations discussed in the previous sections, the constants $T_{\infty_{ij}}$ and τ_{ij} are required for each combination of stressed and measured cores. To calculate the temperature of a core which is cooling down, an additional constant T_0 is needed.

For this purpose, the application `TemperatureParameterTester`, which fully automates the process of measuring data and calculating all required parameters, has been developed. Section 2.4 describes in more detail what is done internally while running this tool.

2.4 Temperature Development Analysis

In order for the temperature model to work correctly, the constants T_0, $T_{\infty_{ij}}$ and τ_{ij} have to be obtained. As the values of these constants may differ greatly depending on the machine used and its environment, they have to be analysed at least once for every machine and environment combination.

The developed `TemperatureParameterTester` tool works by setting a static frequency and choosing a single core to stress for 20 seconds while measuring the temperatures of all cores. Using these measurements, all parameters are calculated online for the chosen frequency. Looping through all available frequencies and running the same test multiple times to average the resulting values, all required parameters are calculated and stored in `/etc/CoolLinux`.

Notice that the program has to be run as root in order to be able to change the CPUs frequency while measuring.

2.4.1 Stressing Individual Cores

To stress an individual core, a stressing thread has to be created, moved to the desired core and run.

The following code can be used to stress a core by using an infinite loop of integer additions:

```
int i=0;
while(running!=-1)
    i++;
```

Notice that the stressing algorithm used in this project actually is a program called MJPEG decoder. However, a simple integer addition is described here for convenience.

To set the affinity of a thread, first an instance of `cpu_set_t` has to be created. The resulting object represents a mask to describe on which core the thread is allowed to run. `CPU_ZERO(&cpuset)` sets the permission to run to zero for all cores, `CPU_SET(coreId,&cpuset)` adds the desired core with ID `coreId` to the set. Finally, the thread migration function `pthread_setaffinity_np()` can be called, given the thread and the cpuset instance as parameters.

In the following example, the affinity of the calling thread is set to `CPU0`.

```
int coreId = 0;
cpu_set_t cpuset;
pthread_t thread = pthread_self();

CPU_ZERO(&cpuset);
CPU_SET(coreId, &cpuset);

int s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &
    cpuset);
if (s!=0)
    std::cerr<<"pthread_setaffinity_np "<<s<<std::endl;
```

2.4.2 Heat Propagation

Due to the physical proximity of the individual cores, there is heat propagation which cannot be disregarded. Therefore, the intensity and delay with which the CPU core's temperatures correlate have to be considered in the *CL* framework. This correlating heat propagation between cores is also called the `neighbouring effect`.

Running a task on core 1, one can clearly see the temperature rising on the idle core 0. To account for this fact, the `TemperatureParameterTester` program calculates τ and T_∞ for each measured and stressed core individually, always stressing exactly one core at a time. The temperature of a single core can then be calculated by superposition of all calculated temperature differences for each stressed core. This functionality is already built into the framework, ready to be used in future projects with multiple working threads.

However, the delay aspect of heat propagation was not pursued. Tests showed that the influence of this factor had only a small dimension. Therefore, this aspect was omitted in favour of other parts of the thesis.

2.4.3 Environmental Influence

The environment has a very important role in the development of the temperature. Just a few degrees difference in room temperature can be directly seen in the temperature development of the CPU. Even direct sunlight can heat up the system by a considerable amount. Therefore, it is essential that all parameters are recalculated with big changes of the environment.

2.5 Temperature Prediction

The final temperature predicting algorithm is based on the following arguments:

- Array of all current CPU core temperatures,
- information which core will be stressed during t_{diff},
- targeted core ID for which the prediction should be made, and
- time difference t_{diff}.

Before actually calling the `predictCoreTemp()` function, an instance of `CpuStressMap` has to be set in the predictor using `setStressMap()`. The StressMap contains information about the core, which will be stressed and is called within the `predictCoreTemp()` function.

Using the model and the arguments mentioned before, the influence of the temperatures of each stressed core on the one to be predicted is calculated in degrees and summed up, thus obtaining an approximation of the temperature. If no core is stressed, the function calculates the resulting temperature by using the cooling down curve of the very same core for which the prediction is made. The temperature of the other cores are not taken into account for this calculation.

Another case for the prediction is when the controller is switching to a lower frequency where the actual temperature is higher than the lower frequencies

T_∞. Since the *CL* framework is designed to work with one simultaneous task, it is easily possible to predict the temperature in this case. Here, the cooling down temperature model discussed in Section 2.3.3 is used to predict the temperature.

2.6 Summary

The *CL* framework makes use of multiple variations of an exponential model for the CPU core's temperature development. It is not only able to calculate the temperature when stressing a core, but also the temperature when cooling down or for a core while another core is being stressed.

As the models used depend on constant parameters which are system-specific, they have to be calculated in advance. For this very purpose, the `TemperatureParameterTester` program has been built to fully automatically calculate and calibrate these parameters.

The sensors providing data about the actual CPU core temperatures can be read at any moment, with a resolution of 1°C, using the modified version of the `coretemp` kernel module.

3 Dynamic Thermal Management

This chapter describes the means of the *Cool Linux* framework to reduce the system's temperature, other than using more effective, and therefore more expensive, cooling systems. *Dynamic Thermal Management* is fairly easy to use, since it is software-based and much can be done from the *user space* of an operating system. *DTM* offers a lot of room for innovations through various possibilities of combining different *DTM*-mechanisms.

Section 3.1 gives an overview of the power management mechanisms of modern Intel CPUs. It also gives a short introduction about CPU-states (the CPU's "power saving too"). A basic understanding of these states is required to understand Sections 3.4 and 3.6. Section 3.2 then discusses the "built-in" thermal mamagement mechanisms of Intel CPUs, and how their functionalities and goals differ from the *CL*-framework's approach. Section 3.4 shows how the operating frequency of the CPU can be changed, the resulting effects on performance and how lower frequencies influence the CPU temperature. Section 3.5 shortly explains the idea of migrating POSIX-threads between different cores, and how that mechanism can be used to keep a multi-core CPU cool. The third possibility to reduce the temperature of the CPU is to put the working thread into a sleep state until the CPU has cooled down. This is described in Section 3.6 and is henceforth referred to as *scheduled sleep*.

3.1 Intel Power Management - CPU-States

A CPU can be in different states, mostly depending on the current load and power saving settings. These states basically determine which parts of the CPU are active, i.e. how much power it consumes. *ACPI* (Advanced Configuration and Power Interface) defines a standard interface for the OS to utilize hardware power features and also defines data structures to track the states, and functions to operate on the states. The CPU implements mechanisms to support these states. The BIOS and software drivers hide the difference in CPU-architectures (and therefore the different CPU-states) to support the structures and functions defined by ACPI. In other words, ACPI-states are abstractions of CPU-states. The following discussion is based on a *mobile 3rd generation core i7 processor*. The exact number/function of CPU-states may therefore differ from other CPU-models.

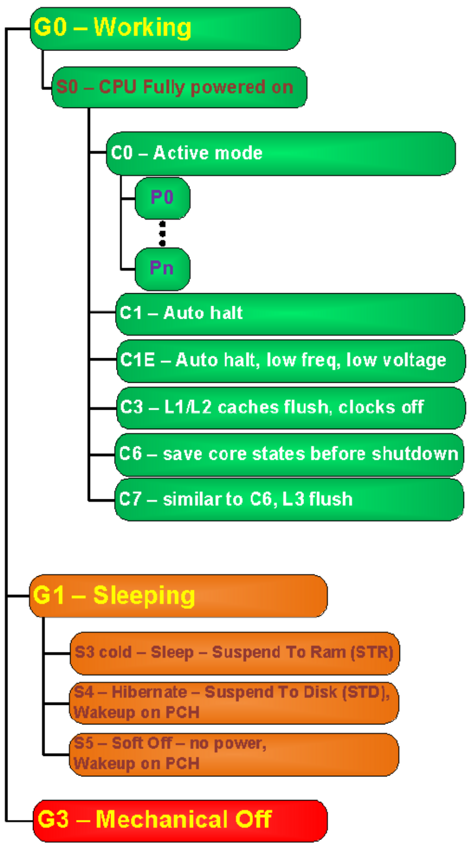

G0 – Working
S0 – CPU Fully powered on
C0 – Active mode
P0
Pn
C1 – Auto halt
C1E – Auto halt, low freq, low voltage
C3 – L1/L2 caches flush, clocks off
C6 – save core states before shutdown
C7 – similar to C6, L3 flush
G1 – Sleeping
S3 cold – Sleep – Suspend To Ram (STR)
S4 – Hibernate – Suspend To Disk (STD), Wakeup on PCH
S5 – Soft Off – no power, Wakeup on PCH
G3 – Mechanical Off

Figure 3.1: Hierarchy of CPU-states for a 3rd-gen core i7 processor [1]

Figure 3.1 shows the CPU-state hierarchy. The important states are the *P*- and *C*-states. The *S*-states are only entered if one manually *suspends*, *hibernates* or powers down the system. How much energy a system consumes while it is in an S-state is generally important, as it determines for how long mobile systems can run on battery power. However, for this thesis, this aspect does not matter since the CL-framework assumes continuous workload, and the CPU will therefore never enter an S-state.

From Figure 3.1 one can see that whenever the CPU is executing code, it must be in the $C0$-state. When the CPU is in $C0$, it resides in one of the P-states, which indicate the current level of performance. Lower numbers mean higher performance and power consumption. While the CPU is idle, it resides in one of the higher C-states ($C1$, $C1E$, $C3$, etc.). The higher the C-state, the lower the power consumption and the higher is the resume delay. The resume delay is the time it takes for the CPU to transition to the active state, i.e. to $C0$.

3.1.1 Low-Power Idle States (C-States)

As mentioned previously, higher C-states (i.e. all C-states except $C0$) are low-power idle states. Further, the C-states are divided into *Thread C-states* (for CPUs with Hyper Threading), *Core C-states* and *Package C-states.* Depending on which thread-/core-/package-state the CPU is in, different power saving actions are taken.

Core C-state

- *Core C-state* is determined by the lowest numerical thread state. For example: Thread 0 is in C1 and thread 1 is in C3, therefore the *Core C-state* is C1.
- Transitions between different *core C-states* always go through $C0$.

Package C-state

- *Package C-state* is determined by the lowest *core C-state* of all cores.
- Transitions between different *package C-states* can be done directly without first entering $C0$.

In essence, this means that different cores (or even threads) can be in different C-states at the same time, therefore consuming different amounts of energy. The CPU can only make the transition into a higher *package C-state* (and therefore save more energy) when all threads/cores are in the same (or higher) *thread-/core C-state.*

The implication of this behavior on frequency scaling is explained in Section 3.4.

3.1.2 Enhanced Intel SpeedStep Technology (P-States)

"Newer" generation Intel processors (most CPUs since *Pentium III*) have a technology called *Enhanced Intel Speedstep Technology* (EIST), which does dynamic frequency and voltage scaling. The P-states represent different operation points of the CPU, i.e. (frequency, voltage)-tuples. When the CPU goes into a higher P-state (lower performance), both voltage and frequency are reduced, which leads to lower power consumption: $P = ACV^2f$, where P is the consumed power, A is the activity factor (the fraction of the transistors that are actually switching) C is the switched capacitance, V is the supply voltage and f is the frequency [3]. *EIST* is software controlled. It is automatically done by the operating system (e.g. by the Linux *on demand* governor) through writing to processor registers. However, the frequency can also be set manually over the ACPI-interface as done in Section 3.4.

As explained in Section 3.1.1, different CPU-cores can be in different C-states at the same time. However, *all cores that are in C0 have to be in the same P-state*. This means that it is not possible to change the operating frequency for each core individually. However, it is possible to have different average frequencies for different cores, because they can be in different C-states. For example: Core 0 is heavily utilized and will be in $C0$ 100% of the time, while core 1 is mostly idle and will reside e.g. in $C6$, and therefore have its voltage/frequency reduced to zero. Tools that measure the average operating frequency (e.g. *i7z*) will then display a lower (average) frequency value for core 1. However, as soon as core 1 enters $C0$ to execute some code, it will be in the same P-state as core 0, therefore running at the same frequency/voltage as core 0.

3.2 Intel Thermal Management

In the official manual for 3rd-generation core i7 processors it is stated:

> To allow for optimal operation and long-term reliability of Intel processor-based systems, the system/processor thermal solution should be designed so that the processor:
>
> - Remains below the maximum junction temperature (T_j, Max) specification at the maximum thermal design power (TDP).
> - Conforms to system constraints, such as system acoustics, system skin-temperatures, and exhaust-temperature requirements.

These two design goals already show that Intel optimizes it's CPU for performance first and foremost. Thermal management is mainly done to prevent system failure, and not to save energy, or keep the system cool. The following subsections introduce different thermal management features of Intel processors.

3.2.1 Intel Turbo Boost and TDP

Newer generation Intel processors have a function called *Turbo Boost*, which allows the processor to run at a higher frequency for a certain time. Turbo Boost mode is entered automatically and opportunistically, as long as the processor stays within current specification limits. While not a thermal management feature in its closest sense, Turbo Boost has several functionalities to enable thermal management.

The processor TDP (Thermal Design Power) represents an expected maximum sustained power for realistic applications when all cores are active and running at the processor's rated frequency. However, in the typical use case, not all cores are active at the same time, therefore the CPU usually consumes less than the TDP at its rated frequency.

When applications are running at Turbo Boost frequency, the system is expected to get closer to the TDP. The TDP may even be exceeded for a short period of time, as long as there is enough “thermal headroom”.

When in Turbo Boost Mode, the processor monitors and controls its power consumption. There are several parameters that influence the behaviour of the Turbo Boost mode, which are adjusted automatically. These parameters basically determine by how much the TDP may be exceeded and for how long. Additionally, they specify limits on the average power consumption over certain periods of time.

Running in Turbo Boost Mode has a large impact on power consumption and therefore heat generation. Changing Turbo Mode parameters, or even disabling it entirely, would lead to a cooler system. However, these parameters are not easily tunable from the *user space*, and one would miss out on the performance gains that Turbo Boost provides. The *CL* framework automatically “disables” (i.e. does not use) Turbo Boost, when using it would cause the CPU-temperature to overstep a specified threshold. But when there is enough thermal headroom, *CL* makes use of the additional performance of Turbo Boost. In other words: There is no need to manually configure the Turbo Boost parameters, *CL* automatically uses Turbo Mode whenever the current CPU temperature allows it. *CL* could even allow to temporarily disregard temperature constraints and run in Turbo Mode nonetheless for a certain time. This could be useful when there is a workload burst and real time constraints have to be met.

For more information on Intel Turbo Boost, see Sections 5.2 and 5.3 of [1].

3.2.2 Configurable TDP (cTDP) and Low Power Mode (LPM)

Other thermal management-functionality of modern Intel CPUs are configurable Thermal Design Point (*cTDP*) and Low Power Mode (*LPM*).

cTDP allows the CPU to adapt its TDP to different operating modes (e.g. power plugged-in, battery power, etc.). Changing the TDP automatically results in a change in performance. *cTDP* has three modes:

Nominal: Processor's rated frequency and TDP.

TDP-Up: When extra cooling capacity is avaiable, e.g. when a laptop is put on a docking station with additional fans, the CPU can operate at higher TDP and ensured frequency. This naturally increases the average power consumption.

TDP-Down: To operate cooler, quieter and longer, the CPU can use this mode to lower TDP and ensured frequency. This of course lowers performance as well as temperature.

Each mode uses different power and frequency ranges for Turbo Boost. However, the maximum Turbo frequency is not changed.

LPM allows the system to operate even below TDP-Down. *LPM* can be configured to use the following methods to reduce power consumption:

- Restrict Turbo Boost Power limits and availability,
- off-Lining core activity, i.e. move processor traffic to a subset of cores,
- place cores in Low/Minimum Frequency Mode, and
- utilize clock modulation.

Configuring *cTDP* and *LPM* could lead to lower power consumption and therefore lower heat generation. However, it is (1) not something one can do from the *user space* and (2) restricts performance even if it is not necessary. Both methods are designed by Intel to enable more versatile use-cases for their CPUs; the same CPU can be put into different devices with different performance requirements, cooling- and docking capabilities.

The *CL* framework on the other hand is capable of dynamically adjust to performance needs while staying within temperature bounds. There is no need to preliminarily restrict the CPUs maximum performance by lowering the TDP. Additionally, it operates entirely in the *user space*, which makes it easy to use and configure.

Note that currently only the absolute high-end Core i7 models (Extreme Edition and Dual Core Ultra) are capable of *cTDP* and *LPM*. For more detail see (Section 5.4 of [1])

3.3 Implications of Intel's Power/Thermal Management Capabilities for Cool Linux

This section gives a short summary of the above discussed Intel technologies' implications on the design of the *CL* framework.

- All active cores (i.e. cores in $C0$) run at the same frequency, i.e. are in the same P-State. Therefore, it is not possible to set different operating frequencies for different cores.
- It is not easily possible to directly manipulate the CPU's C-states from the *user space*. "Putting a CPU-core to sleep" by manually requesting a transition to a higher C-state can therefore not be done.
- The Intel power- and thermal management technologies are mostly there to guarantee safe operation, i.e. to prevent thermal breakdown. Maximum performance is more important than low temperature, as long as the system is able to provide "sufficient" cooling. Sufficient means that the CPU-temperature must not reach the maximum junction temperature $T_{j,max}$, which is usually around $105°C$.
- The Intel power- and thermal management technologies are not designed to be used from the *user space*, they are normally used by the operating system. That means, relying on these Intel technologies for the *CL* framework is not a viable option. Making some of them accessible in the *user space* might be possible, but certainly not easy to do, nor would they then be easy to use.

3.4 Changing CPU Frequency

Naturally, the current operating frequency of the CPU has an influence on its temperature: The faster the transistors switch, the higher the dynamic power dissipation $P = ACV^2f$ is. Therefore, reducing the frequency results in less dynamic power dissipation, which generally leads to lower temperatures.

This section explains how the CPU frequency can be set from the *user space* and how changing the frequency influences CPU temperature and performance, or execution time of code respectively.

3.4.1 Frequency Scaling on Linux

On Linux, one can get control over the *ACPI*[1] CPU frequency kernel functions by using the *cpufreq* library (see Appendix D.3 for more information). *Cpufreq* provides functions to set/get the frequency governors, operating frequencies and other information about the system, like possible frequencies, switching latency and so on. To use the *cpufreq*-functions in a C-program, the *cpufreq* library has to be installed and *cpufreq.h* has to be included. The ACPI *cpufreq* library supports the prevously discussed Intel SpeedStep technology to change the operating frequency.

As explained in Section 3.1.1, it is not possible to set different frequencies for each individual core; all running/active cores are in the same *P*-state and therefore running at the same frequency. This means that the *Cool Linux* framework always changes the frequency of all cores. Since the framework currently only supports one worker-thread, i.e. only one core is utilized at the same time, this fact does not have any major impact. To support several worker-threads all running on different cores, one would have to consider the fact that lowering the frequency based on the overheating of one core, also lowers the frequency for all other cores, even if they are not too hot. However, this is part of future work. See Section 4 for details on the integration of frequency scaling in the framework.

3.4.2 Influence on Temperature

In order to assess the viability of frequency scaling for the purpose of keeping a CPU cool, several tests were done to quantify the effect of frequency scaling on CPU-temperature. The results are shown in Figure 3.2. One can clearly see that the steady state temperature monotonically decreases with lower frequencies. Also note that the turbo-frequency of the i7-processor ($\approx$ 3.4GHz) disproportionately increases the CPU temperature.

The test was run on core 0, which happens to be the hottest core in the test system, supposedly due to the unsymmetrical placing of the cores on the CPU. To stress the CPU, a C-program corresponding to the following pseudo-code was used:

[1]Advanced Configuration and Power Interface, provides standard interface for OSes to utilize hardware power features.

Algorithm 1 Stress CPU

```
for $i \leq iter, i++$ do
    for $j \leq iter, j++$ do
        $pow(i * j, 10)$
    end for
end for
```

These preliminary results show that frequency scaling is a viable option to regulate the temperature of the CPU.

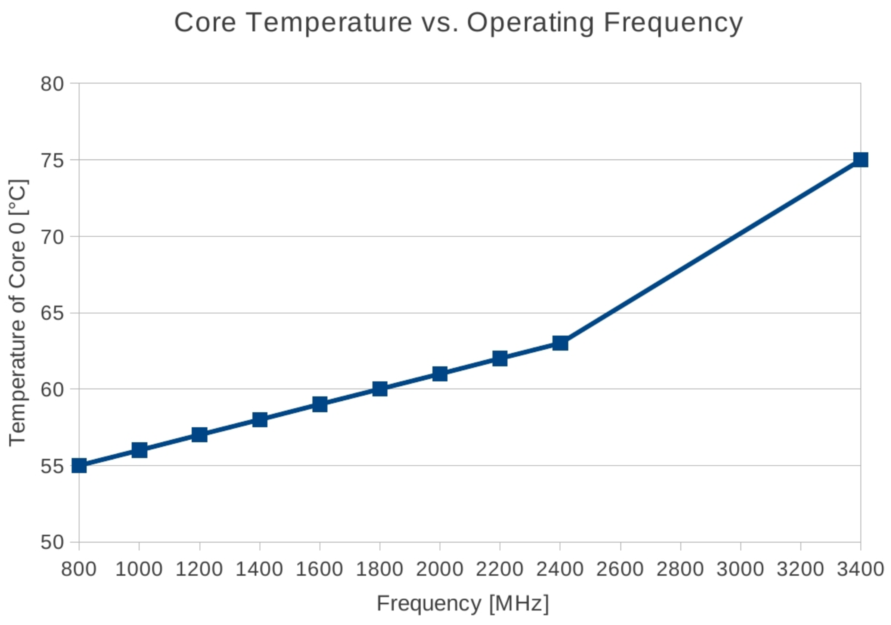


Figure 3.2: Temperature of core 0 in dependence of operating frequency.

3.4.3 Influence on Performance

While the "cooling"-effect shown in the previous subsection is desirable, lowering the frequency also decreases performance, which might be a problem depending on the application. To measure the effects of frequency scaling on performance, the following test was done: The machine executes some piece of code in a while loop for n iterations and measures the execution time for different frequencies. As for the previous test, C-code corresponding to the pseudo code in 1 was used, where $iter$ was fixed to n for all tested frequencies.

The test was repeated for different combinations of mathematical expressions and also for operations on large arrays, in order to “incorporate” the L2 and maybe L3 cache. However, the results were all the same.

Figure 3.3 shows the results. One can see that the performance scales perfectly linearly. In theory, this would mean that the performance loss directly corresponds to the decrease in frequency. In other words: If the *CL* framework reduces the average frequency by 30% in order to keep the CPU cool, the expected performance-loss is also 30%. If this is always true in practice will be discussed in the evaluation (Chapter 5).

The final framework is able to enforce a minimum operating frequency in order to somewhat guarantee a certain level of performance. This is covered in more detail in Chapter 4, where the integration of frequency scaling into the framework is covered. Further evaluations of the feasibility of frequency scaling are done in Chapter 5.

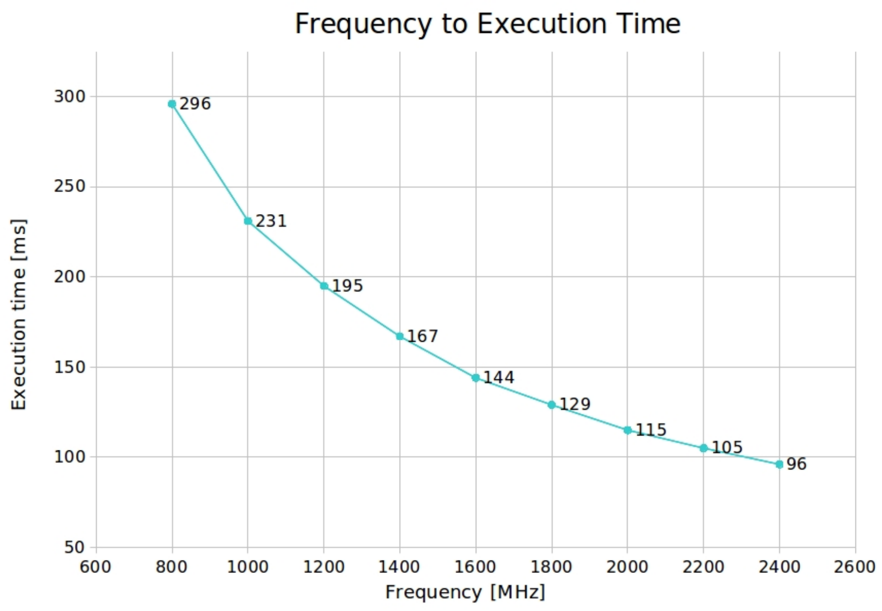


Figure 3.3: Dependence of execution time on operating frequency.

3.5 Task Migration

CPUs in server systems can have up to several dozens of cores and it is very likely that not all cores are symmetrically placed on the chip, nor equally utilized at all times, and therefore, the different cores will have different temperature characteristics. By using the POSIX `pthread_setaffinity_np` method it is possible to tell the OS to continue the execution of a task on another core. By choosing a core with lower temperature than the current one, it is possible to keep the overall CPU temperature lower than if the task is executed on a single core.

When migrating a thread from one core to another there is a certain overhead. Depending on the processor, the average time for a process context switch *with possible migration to another core*, is between 3000 and 4500 ns per context switch [4]. Thread context switches are only marginally faster. However, these numbers indicate the duration of *mere* context switches (and possible migrations to another core), without considering cache pollution. In a real world application, these numbers are expected to be a little higher, depending on the working set size with respect to cache size [5]. Another factor that influences the durations of context switches is the number of concurrently running threads, which generally should not exceed the system's number of hardware threads.

Having said that, the main goal of the Cool Linux framework is not to optimize performance, but to minimize (i.e. set an upper bound on) the temperature of the CPU. Therefore, we make do with ensuring there are not too many ("unnecessary") task migrations. However, the frequency of task migrations is anyway lower bounded by the maximum frequency of the temperature-control-loop, which is somewhere betwen 100 and 200 miliseconds. Additionally, *task migration* allows to do load balancing. If, for example, other tasks than the *CL* framework are running on the machine, *CL* can make sure that its own tasks do not run on the same core as the other tasks whom it has no control over. Furthermore, CPU-cores are not symmetrically placed on the chip, which can lead to very different temperature dynamics for the individual cores. On one test system, the first core got up to $10°C$ hotter than the fourth core. Task migration allows to take these differences into account and preferably run the tasks on these inherently cooler cores.

Chapter 4 explains how task migration is integrated in the framework. Chapter 5 shows how good task migration is for the purpose of dynamic thermal management in our framework.

3.6 Scheduled Sleep

Another way to reduce the temperature of the CPU is to force the working thread to sleep, thereby allowing the entire framework to sleep. Of course, this has a more serious impact on performance than reducing frequency and/or migrating a thread to another core. This is why forcing a thread to sleep is intended to be used as a safeguard to cool down the CPU when it gets unexpectedly hot very fast and the other mechanisms (frequency scaling, task migration) are not able to keep it cool enough.

3.6.1 Using CPU C-States to Sleep

As discussed in Section 3.1, there are different CPU-states, namely C-states (Processor Power States), P-states (Performance States) and S-states (Sleep states). Normally, the OS decides which state the CPU should be in and there are several possible ways to do so [1]:

1. Execute HLT instruction to enter C1 or C1E.

2. Using MONITOR/MWAIT: MONITOR specifies a memory range to be monitored. MWAIT halts the processor until the address specified with MONITOR is accessed.

3. Reading of the P_LVLx I/O registers will cause the CPU to enter a model-specific C-state. Reading the P_LVLx register is internally translated into an MWAIT instruction.

This naturally raises the question whether it is possible to manually set/request different C-states from the *user space* to put entire cores into a low-power state when they get too hot.

Unfortunately, the methods described above only work from *kernel space*. There seems to be a loadable kernel driver available for *Solaris* systems that exposes MONITOR/MWAIT to the *user space* [6]. However, we did not find something similar for Linux that would be ready-to-use. This means that manipulating the CPU-states from the *user space* could actually be done, but to the best of our knowledge, it is not straightforward to do so, neither are there any manuals on the matter. However, we could find some pointers on how it could be done, including (1) integrating ACPICA into the OS as a kernel-resident subsystem, which requires an adaption to the specific OS, or (2) exploring Kernel Mode Linux [6].

Integrating a sleep-functionality that directly utilizes the CPU-states could be topic of future work. The next section describes the idea behind the sleep functionality of the final framework; instead of putting entire CPU-cores to sleep, we temproarily suspend the POSIX thread that is doing the work, and therefore has the highest CPU utilization.

3.6.2 Suspending the Worker Thread

Linux offers an easy-to-use sleep mechanism; in order to suspend the execution of a thread for at least t microseconds, one can just call the function *usleep(t)*. When a thread calls the *usleep(t)*, it loses the CPU to other threads for *at least t* microseconds, before it resumes execution.

In the *CL* framework, there is a *worker*-thread that executes code, e.g. decodes an MJPEG video. In order to cool down the CPU, i.e. to put the worker to sleep, we use a *mutex* shared between the *controller*-thread and the *worker*-thread. When the *contoller*-thread decides that the worker should sleep, it locks the *mutex*, i.e. blocks the *worker*, and then the *main*-thread calls *usleep*. Because the *controller*-thread has locked the *mutex*, the *worker* cannot resume execution, even though the *controller*-thread loses the CPU for the sleep duration. Through that we make sure that (1) the *worker* is indeed doing nothing, and (2) we do not use busy-wait. Using busy-wait would prevent the *main*-thread from losing the CPU, and therefore prevent the *worker* from resuming execution. However, the CPU cannot cool down during a busy-wait, which defeats the purpose of sleeping in the first place.

Section 4 goes into more detail about this sleep mechanism as part of the entire framework. Section 5 shows how well suited this sleeping-mechanism is to keep the CPU cool.

3.7 Summary

Chapter 3 gave necessary background on processor states and introduced the three main actuation mechanisms of the Cool Linux Framework:

Frequency scaling: Reducing the frequency lowers CPU-temperature as well as performance. The performance scales linearly with respect to operating frequency. All active cores ($C0$-state) are in the same P-state and therefore run at the same frequency. To do frequency scaling in Linux, the *cpufreq*-library, which grants access to the ACPI-functions, is used.

Task migration: Migrating threads (or tasks respectively) from a hot core to a cooler one can be used to keep the entire CPU cool. This can be especially useful if there are other tasks running on the system; migration allows the tasks controlled by the Cool Linux framework to "dodge" those processes. Additionally, CPU-cores are not symmetrically placed on the chip and therefore have different temperature behaviours; migration allows to preferably run tasks on cores that stay cooler. The migration delay is negligible for *Cool Linux*. However, the migration overhead might contribute to heating up the CPU.

Scheduled sleep: Scheduling sleep-times for the worker-thread gives the CPU time to cool down. This method has the most negative impact on performance, and should therefore only be used when frequency scaling and task migration fail to keep the CPU cool enough. In order to put the entire program to sleep, a higher priority thread locks a *mutex* that is shared with a lower priority worker-thread, and then executes *usleep*. For the *mutex* is locked by the sleeping thread, the worker thread is in fact also sleeping.

4
Cool Linux Framework

This chapter discusses the implementation of the *CL* framework. Sensing- (Chapter 2) and actuation mechanisms (Chapter 3) are brought together in a control loop to regulate the CPU-temperature.

Section 4.1 introduces some important parameters of the framework. Section 4.2 shows basic the structure of the framework. The most important features of the framework are then discussed in Section 4.3. Section 4.4 explains how the worker and dispatcher threads work, and how they are integrated in the framework. Finally, Section 4.5 introduces the *main control loop thread*, and shows how it manages to control the CPU temperature.

4.1 Specification

In the *CL* framework the following goals can be specified:

prediction threshold T_{pt}**:** The *prediction threshold* is the desired temperature of the system. The system's CPU-temperature should be as close to the *prediction threshold* as possible without exceeding it. Of course, depending on how hot a particular system can get and what it's idle temperature is, this value must be adapted accordingly.

performance threshold f_t**:** The *performance threshold* specifies a lower bound on the average operating frequency of the system. This makes sure that it is not possible to simply run at the minimum frequency, completely disregarding any kind of performance considerations. While the main goal of the framework is temperature control, it also aims at having high performance.

Other parameters of the framework are:

Controller Speed/Period: The controller speed specifies how often the sensors are read and how fast the controller actuates. It is a parameter that must be optimized for a specific system.

Operating Frequency: The CPU's operating frequency is a system-wide parameter and can be set by the actuator.

CPU Affinity: The CPU affinity denotes the core where the application is currently running on. The actuator can set it separately for each POSIX thread.

Duration of Scheduled Sleep: Indicates for how long the actuator should put the *worker* to sleep.

4.2 Framework Architecture Overview

Figure 4.1 shows the most important building blocks of the framework and how they interact.

There are three separate POSIX threads: (1) The main control loop thread, (2) a dispatcher thread, and (3) a *worker* thread. The main control loop thread is where all the intelligence resides and the decisions (i.e. actuation and control) are done. Section 4.5 describes the control loop in more detail. The dispatcher and *worker* form the CPU-intensive part of the framework, and therefore must be controlled by the main control loop, in order to keep the CPU cool. Section 4.4 gives some more insight on the implementation of the dispatcher and *worker*.

4.3 Framework Features

This section gives a short overview of the *CL* framework's most important and distinguishing features:

- Keeping CPU-temperature below specified threshold. The framework automatically decides which DTM-measures to take in order to reach that goal.

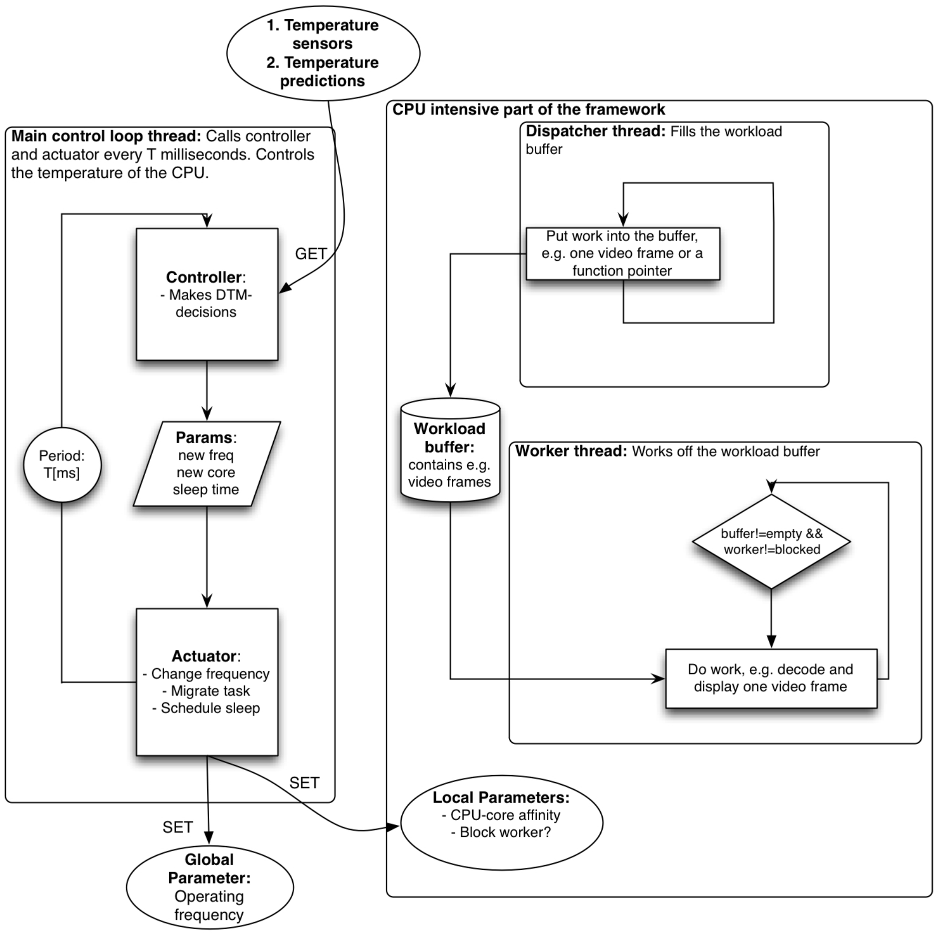

1. Temperature sensors
2. Temperature predictions
Main control loop thread: Calls controller and actuator every T milliseconds. Controls the temperature of the CPU.
GET
Controller: - Makes DTM-decisions
Params: new freq new core sleep time
Period: T[ms]
Actuator: - Change frequency - Migrate task - Schedule sleep
SET
SET
Global Parameter: Operating frequency
CPU intensive part of the framework
Dispatcher thread: Fills the workload buffer
Put work into the buffer, e.g. one video frame or a function pointer
Workload buffer: contains e.g. video frames
Worker thread: Works off the workload buffer
buffer!=empty && worker!=blocked
Do work, e.g. decode and display one video frame
Local Parameters: - CPU-core affinity - Block worker?

Figure 4.1: Overview of the framework and its most important parts.

- Always run on best core, i.e. on the core that has the best temperature dynamics.
- Allow for maximum performance if enough thermal headroom. The framework allows to specify lower bound on operating frequency, which is then used as a lower bound on the operating frequency's moving average.
- Whenever possible, the task keeps running on the same core. Unnecessary task-migrations are avoided due to the associated overhead.

Future Work:

- The framework could allow to deliberately overstep temperature threshold to meet real time deadlines, e.g. allow to run at turbo for a certain time even if there is not enough thermal headroom.

- The framework could allow to monitor performance requirements of the *worker* thread. If, e.g., the *worker* cannot display a video at 24 fps at the current frequency (the video stutters), the framework could choose a higher frequency to play the video smoothly $\Longrightarrow$ Better awareness of the workload. For instance, instead of always running as fast as possible (within the temperature upper-bound), *CL* could always run *as fast as necessary* to meet real-time constraints; if a frequency of 1.2 GHz suffices to run a video at 24 fps, it does not make much sense to run at 2.2 GHz just because the current CPU-temperature would allow it.

4.4 Worker and Dispatcher

The workload executed within the *CL* framework is composed of a *worker* and a dispatcher thread. Since the framework is implemented in the user-space (i.e. it is not a substitute for the OS scheduler), direct control over the workload is necessary in order to perform effective actuation using the three DTM-mechanisms discussed in Chapter 3.

For that purpose, the *main control loop thread* starts two separate threads which it has full control over: (1) The *dispatcher* and the (2) *worker*. The right half of Figure 4.1 shows the *dispatcher*, *worker* and the workload-buffer. In the current framework, the workload consists of a series of MJPEG frames that, when decoded, display a little video. A higher framerate results in a higher workload and generally in higher CPU temperatures. Note that for all tests conducted with the final *CL* framework (see Chapter 5) the video frames were only decoded, but not displayed. The reason is that the video is shown in an external player so that the framework does not have full control over the application anymore, e.g. it cannot migrate the video task to another core.

The *dispatcher* provides work to the *worker* by filling the workload buffer with undecoded video frames. This means the *dispatcher* controls how fast the application runs. If the workload buffer is empty, the *worker* stops and if the *dispatcher* fills the workload buffer with 30 video frames per second, the *worker* runs with a maximum framerate of 30 fps. For the tests done in Chapter 5 the framerate was set to 5000fps to ensure that the CPU is put under maximum load. The value of 5000 was chosen high enough, such that the stressed core is in $C0$ for 100% of the time.

The *worker* has no built-in intelligence and just works as fast as it can. It can however be blocked by the main control loop. This is done using a *mutex*:

Algorithm 2 Worker Thread

```
loop
    if workloadBuffer $\neq$ empty then
        lock(mutex)
        fire(decode_frame)
        unlock(mutex)
        usleep(1)        ▷ Give the main control loop time to lock the mutex
    end if
end loop
```

Therefore, when the controller decides to put the *worker* to sleep, it just has to lock the mutex first and then call *usleep(t)*. As a consequence, the entire program sleeps for t microseconds.

4.5 The Control Loop

The heart of the framework is the control loop inside the *main control loop thread*. As shown in Figure 4.1, it contains two main functions: (1) the *controller* and (2) the *actuator*.

The main thread wakes up every t_{mp} microseconds and performs the following tasks before it sleeps again for t_{mp} microseconds:

Algorithm 3 Main Control Loop Thread

```
loop
    coreTemps=getCoreTemps()
    currentFreq=getCurFreq()
    controlDecisions=CONTROLLER(coreTemps,currentFreq)
    ACTUATOR(controlDecisions)
    usleep($t_{mp}$)
end loop
```

Subsections 4.5.1 and 4.5.2 discuss the *controller* and *actuator* in more detail.

4.5.1 The Controller

Figure 4.2 shows a flow chart of the controller. The chart describes how the controller makes its decisions, i.e. which actions the actuator should take afterwards in order to keep the CPU temperature below the specified temperature threshold for the next iteration.

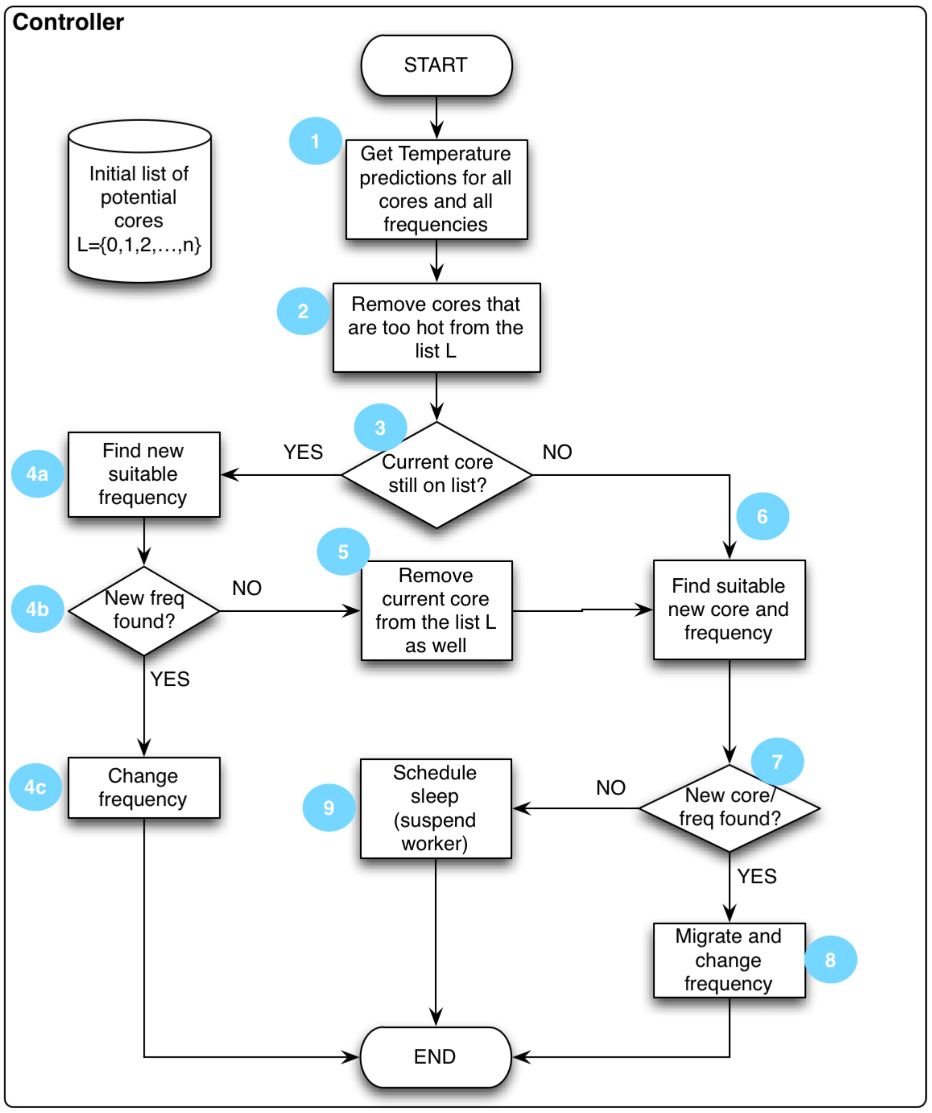

Controller
START
Initial list of potential cores L={0,1,2,…,n}
1
Get Temperature predictions for all cores and all frequencies
2
Remove cores that are too hot from the list L
3
Current core still on list?
YES
NO
4a
Find new suitable frequency
4b
New freq found?
NO
YES
5
Remove current core from the list L as well
6
Find suitable new core and frequency
4c
Change frequency
9
Schedule sleep (suspend worker)
NO
7
New core/ freq found?
YES
8
Migrate and change frequency
END

Figure 4.2: Flow chart of the *controller*

The enumeration below describes Figure 4.2 in more detail. The numbers in the enumeration correspond to the numbers in the flow chart.

(1) : The controller calculates the temperature predictions for all cores and all available frequencies. For example: Predictions[2][3] would be the predicted temperature if the *worker* was migrated to core 2 and the frequency was changed to the frequency with index 3, e.g. 2Ghz.

(2) : The controller manages a list of cores where the *worker*/*dispatcher* can potentially run on for the next iteration. At the beginning, all cores are on the list. The controller then removes all cores from the list whose current temperatures are above temperature threshold.

(3) : If possible, the task should not be migrated to avoid unnecessary overhead: Check if current core is still on the list, if yes $\Longrightarrow$ (4a). If not, i.e. the task cannot continue to run on the same core $\Longrightarrow$ (6).

(4a) : Find the new maximum operating frequency f_{max}, such that the predicted temperature T_p for the next iteration is below the *prediction threshold* T_{pt} **and** above the *performance threshold* f_t. For example: $T_{pt} = 65°C$, $f_t = 1.8GHz$, the new frequency (for which the predicted temperature would be below $65°C$) is $f^*_{max} = 1.2GHz < f_t = 1.8GHz$ $\Rightarrow$ 1.2GHz is not a valid frequency!

$\Longrightarrow$ (4b).

Note: Internally, the *performance threshold* is implemented as the average of the frequency over a certain time interval, and therefore temporarily allows for lower frequency values than f_t.

(4b) : If a valid frequency is found $\Longrightarrow$ (4c). If no valid frequency is found $\Longrightarrow$ (5).

(4c) : Decide to change CPU-frequency to f_{max} as found in (4a). $\Longrightarrow$ END

(5) : No valid frequency for current core found, therefore the controller looks for another core to migrate to. Also, it removes the current core from the list of potential cores $\Longrightarrow$ (6).

(6) : Repeat the process of 4a, but this time for all cores that are remaining in the list L of potential core. Try to find a ($newCore$, f_{max})-tuple that satisfies the temperature and performance constraints $\Longrightarrow$ (7).

(7) : If there is a valid (core, frequency)-tuple $\Longrightarrow$ (8). *Note:* If there are several valid (core, frequency)-tuples with the same frequency, the controller chooses the core with the lowest current temperature.

If there are no valid frequencies for any core, i.e. task migration and frequency scaling cannot keep the core cool anymore $\Longrightarrow$ (9).

(8) : Controller decides to migrate to $newCore$ and change frequency to the newly found f_{max} $\Longrightarrow$ END.

(9) : Since there is no way to keep the CPU-temperature below the *prediction threshold* **and** keep the operating frequency above the *performance threshold*, the controller decides to put the *worker/dispatcher* to sleep. The actuator also sets the CPU frequency to the minimum value before sleeping, to get the biggest cooling-effect $\Longrightarrow$ END.

4.5.2 The Actuator

As soon as the controller has made its decisions, the *actuator*-method is called. The *actuator* performs one of the following actions:

Change CPU frequency: The task continues to run on the same core. The CPU frequency is set to f_{max} which was determined by the controller.

Migrate task: The controller decided that the current core is too hot and even lowering the frequency cannot change that quickly enough. Therefore the actuator migrates the task (i.e. *worker* and *dispatcher*) to another core. At the same time, the CPU frequency might be changed as well.

Scheduled sleep: The controller decided that neither frequency scaling nor task migration are sufficient to keep the CPU-temperature below the temperature threshold. The actuator then blocks the task (using mutexes) and goes to sleep. To figure out how long the work should be suspended, the actuator calls the *temperature predictor*. The predictor computes how long it takes for the current core to cool down $x°C$. x can be chosen freely, as long as it is not too large. $T_{current} - x$ must not be lower than the idle temperature of the corresponding CPU core.

4.6 Summary

This chapter described the architecture of the *Cool Linux* framework and explained how it controls the CPU temperature.

The framework consists of two parts: (1) A controlling part and (2) a working part. The working part (dispatcher and *worker*) try to work as fast as possible, or as specified respectively. The goal of the controlling part is to let the working part run as fast as possible, while keeping the CPU temperature below the prediction threshold.

The controlling part itself basically performs two actions:

Controller: The controller makes the decisions to change the frequency, migrate the task or put the *worker* to sleep, based on performance and temperature criteria. In order to do that, it uses temperature sensor readings and temperature predictions.

Actuator: Executes the controllers decisions: (1) Changes operating frequency, (2) migrates the *worker*/dispatcher and/or (3) puts the *worker*/*dispatcher* to sleep.

5 Evaluation

The performance of the *CL* framework depends heavily on the chosen parameters. For example a very low period for the controller and actuator would lead to a high temperature prediction accuracy, but it would also increase the risk of overshoots, which would have a negative influence on both performance and temperature of the framework. On the other hand, a controller period chosen too high would lead to the controller not reacting fast enough when the temperature of the cores exceeds the temperature threshold. Many temperature violations would be the consequence.

Therefore, in order to obtain optimal parameters, the *CL* framework has been evaluated using several tests focusing on different targets, which are explained in the next paragraph. The obtained data and plots can be found in this chapter.

The first experiment in Section 5.2 aims at determining good values for the control loop period. The second one in Section 5.3 measures the performance of each DTM method using the parameters obtained in the first experiment. Finally, the last experiment in Section 5.4 compares the performance and temperature development of the *CL* framework against the standard Linux On Demand governor.

	Lab PC T48	**HP notebook**
Name	Intel Core2Quad Q6600	Intel Core i7-2760QM
Default frequency	2.4 GHz	2.4 Ghz
Supported frequencies	2.4, 2.1, 1.86, 1.6 GHz	3.4(TB), 2.4, 2.2, 2.0, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8 GHz
Hyperthreading	No	Disabled
Turboboost	No	Yes (up to 3.4Ghz)
Operating System	Ubuntu 12.04 x64	Ubuntu 12.04 x64

Table 5.1: Overview of the used target platforms

5.1 Experimental Setup

As can be seen in Table 5.1, the three experiments have been running on two different machines, a desktop computer and notebook by HP. Both machines run an *Intel* quad core CPU at rated frequency of 2.4 GHz and use a 64 bit version of Ubuntu 12.04 as their operating system.

The notebook's CPU uses the *Intel Turboboost* technology, which enables the processor to overclock itself to higher frequencies for a short amount of time if more processing power is required. During analysis of the measured data, this mode is marked with a frequency of 3.4 Ghz.

However, the results for both machines resemble each other very much and therefore, in order to avoid repetition, the experiments are always discussed for one single machine.

5.2 Controller Speed and Temperature Threshold

The first experiment is aimed at controller tuning. The goal is to determine optimal values for the control loop period and the temperature threshold, to enable the *CL* framework to keep the system cool while still achieving reasonable performance.

The parameters obtained will be used later in the following experiments. The task being executed is the MJPEG decoder decoding frames at maximum speed.

5.2.1 Measured and variable parameters

For this experiment, the full framework was run multiple times for 60 seconds, each time varying the values for the controller period and the temperature threshold. The following parameters have been measured:

- CPU frequency
- ID of stressed core
- Temperature of stressed core
- Previously predicted temperature for the stressed core

5.2.2 Results

Analysis

The resulting data of this first experiment using the T48 lab PC can be found in Figure 5.1. It shows a 3D-plot for the **average overshoot per second** over the **controller period** and the **temperature threshold**. The **overshoot per second** is a measure obtained by calculating the overshoot over the temperature threshold each time an overshoot occurs and summing it up. Finally, the value is divided by the total execution time, therefore giving a measure to define by how much the temperature overshoots the threshold for a given parameter configuration each second.

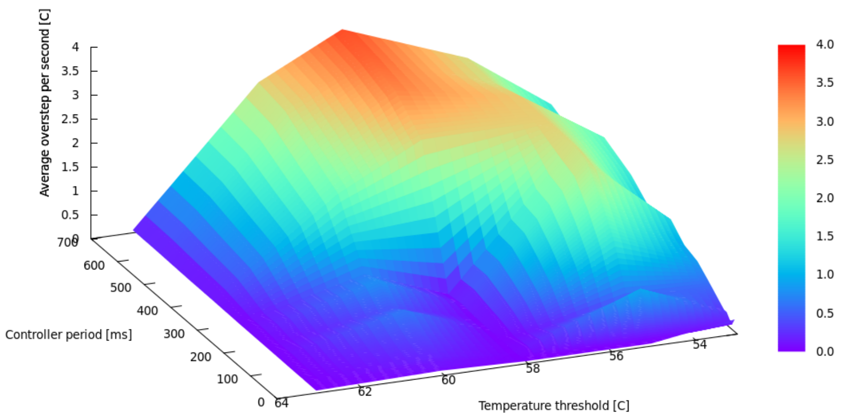


Figure 5.1: First experiment using T48 showing average overshoots per second.

With increasing controller period, the overshoot per second increases dramatically, the framework even reaches an average overshoot per second of 4°Celsius at the highest point. However, as expected, small controller periods result in very few overshoots per second. In this situation, the controller is called so frequently that it is able to react very quickly to the smallest temperature changes and therefore almost no overshoots occur.

For very small as well as for very high temperature thresholds the value for the overshoots per second is also very low. On the one hand, for small thresholds near the CPU's idle temperature, the framework uses the scheduled sleep DTM technique very often, which causes only few overshoots. For very large thresholds on the other hand, the executed task simply does not heat up the CPU enough in order to be able to overshoot the threshold.

Determining Optimal Values

In order to select the optimal values from the graphs, multiple things have to be considered.

Controller period This variable should be as high as possible. The lower the controller period, the more often the controller is called, which leads to the *CL* framework itself requiring more performance, which in turn leads to a higher temperature.

Temperature threshold Using a value near the idle temperature of the CPU causes the framework to lose much of its performance, as it is forced to sleep more frequently. However, a temperature near T_∞ leads the framework to always choose the highest frequency and using the other DTM techniques only very rarely.

Using these criteria, the following values have been selected to be optimal for further experiments on the testing systems:

	Lab PC T48	HP notebook
Controller period	150 ms	200 ms
Temperature threshold	58 Celsius	57 Celsius

5.3 Effect of Different Cooling Methods

The goal of this experiment is to compare the effect of the used DTM techniques on the core's temperatures. Therefore, the *CL* framework was run for every combination of activated DTM methods.

5.3.1 Measured and Variable Parameters

For each combination of the following DTM methods (see Chapter 3), the framework was run for 60 seconds.

- Sleep
- Frequency changing
- Migration

While running the experiment, the following parameters had been measured:

- CPU frequency
- ID of stressed core
- Temperature of stressed core
- Previously predicted temperature

For this evaluation, the parameters for the controller period as well as the temperature threshold have been fixed to the values obtained in the previous experiment (see Section 5.2).

5.3.2 Results

In this section three specific cases are discussed. For the first case, frequency scaling is the only DTM technique which was disabled. The second case uses only frequency scaling and the last case makes use of the full framework.

The data shown in this section originates from the HP notebook.

Sleeping and Migration

For disabled frequency scaling, the result can be seen in Figure 5.2. Focusing on the plot for the stressed cores, very rapid migration can be seen. The pauses between the migration, as can be also seen in the frequency graph, represent the controller having decided to sleep during this time. However, the temperature plot clearly shows that the temperature exceeds the threshold of 57°C. However, the temperatures during the sleeping are lower than what the graphs indicate, as the measurements are taken when the framework's controller is called (which in turn is not called while sleeping). The predictions for this combination were always approximately 5°C below the actual temperature and are not shown in the figure.

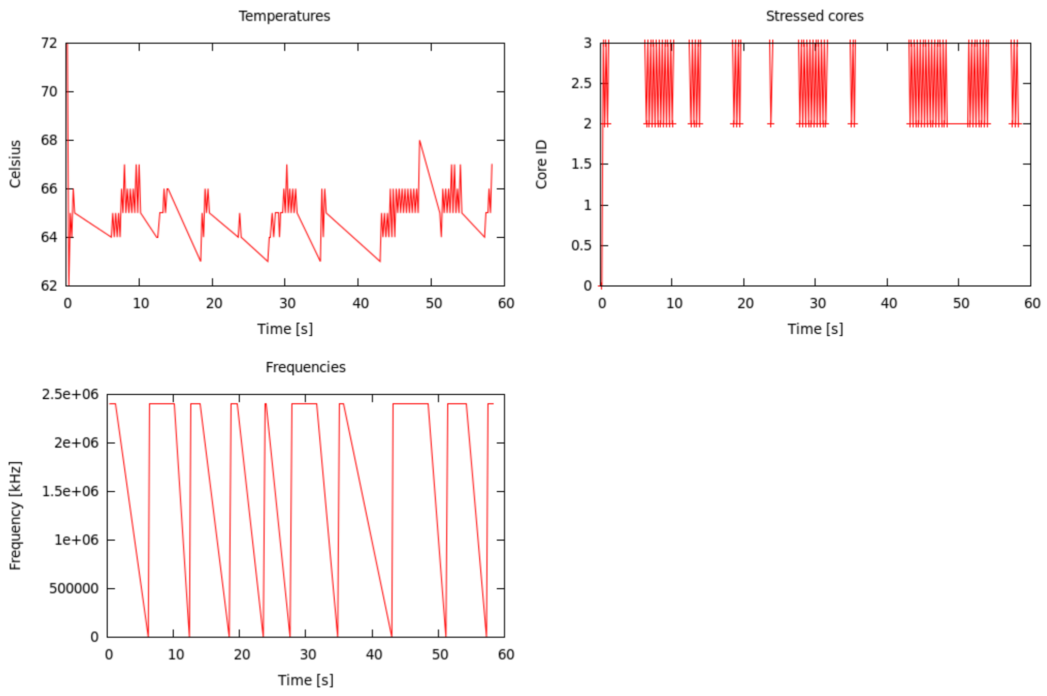


Figure 5.2: Snapshot of the second experiment using the HP notebook, with frequency scaling disabled.

Although sleeping reduces the temperature during the sleep time, the temperature immediately jumps up to nearly the same level as before (± 1.5°C). However, during that time the task cannot be executed at all which results in great performance losses for the framework. Therefore, sleeping as the only DTM method is not enough and might only be used in emergencies.

For migration, one can see that the temperature curve is very bumpy and it is also not able to drastically reduce the temperature. Also, there seems to be a big overhead for the CPU using migration, which has not been considered in the prediction algorithm. Furthermore, due to the exponential temperature increase and the *Neighbouring effect*, it is not feasible for a task to "cycle" through all CPU cores using migration to keep the system cool. The heat correlation between the cores of a quad core CPU is too high and the entire CPU heats up during this process, giving almost no temperature advantage to stay on the current core.

As a conclusion, we can say that sleeping and migration alone are not enough to effectively cool the CPU down.

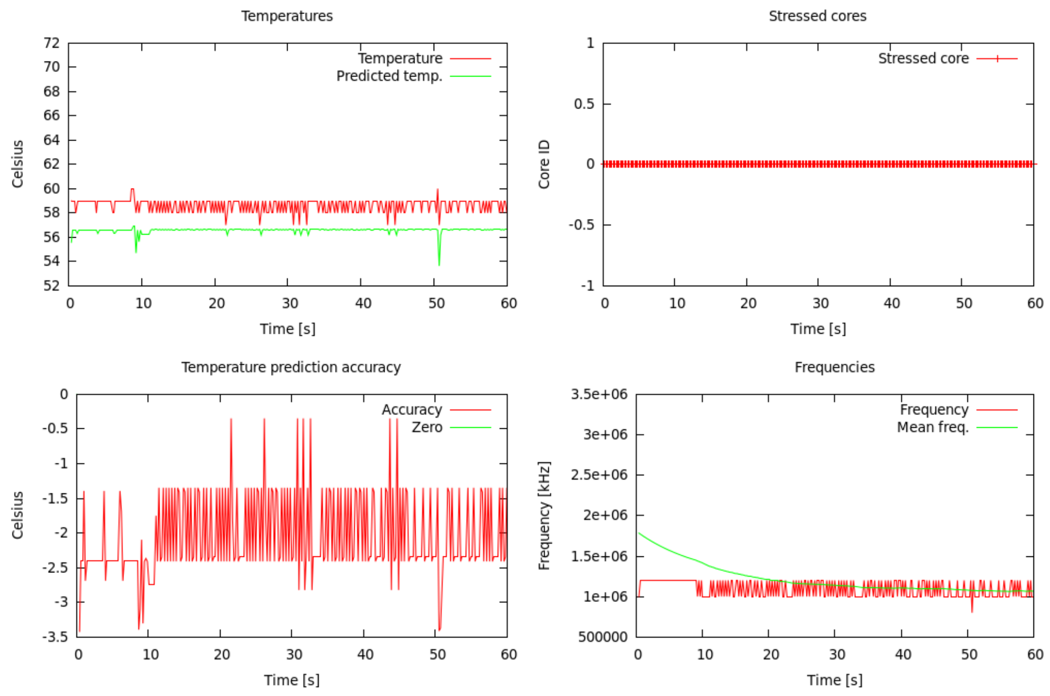


Figure 5.3: Snapshot of the second experiment using the HP notebook, only frequency scaling enabled.

Frequency Scaling

The result for the case when frequency scaling can be used is shown in Figure 5.3. During this instance of the experiment, the task ran at a very low average frequency. This enabled the framework to stay very close to the desired temperature threshold of 57°C. However, the predictions were off by an average of approximately -2°C, which was caused by a too low prediction of T_∞. This deviation is very likely to originate by a slight change of environment compared to when the parameter finding algorithm `TemperatureParameterTester` had been executed (e.g., a higher room temperature).

We can conclude from this experiment that frequency scaling is a good DTM technique which enables the *CL* framework to efficiently reduce the temperature while producing a very steady temperature curve. This experiment also shows the importance of the exactness of the parameters used for the temperature prediction algorithm (see Section 2).

However, compared to the results of the full framework, which can be seen in the next section, the framework runs at a much lower average frequency, thus limiting its performance by a great factor.

Full Framework

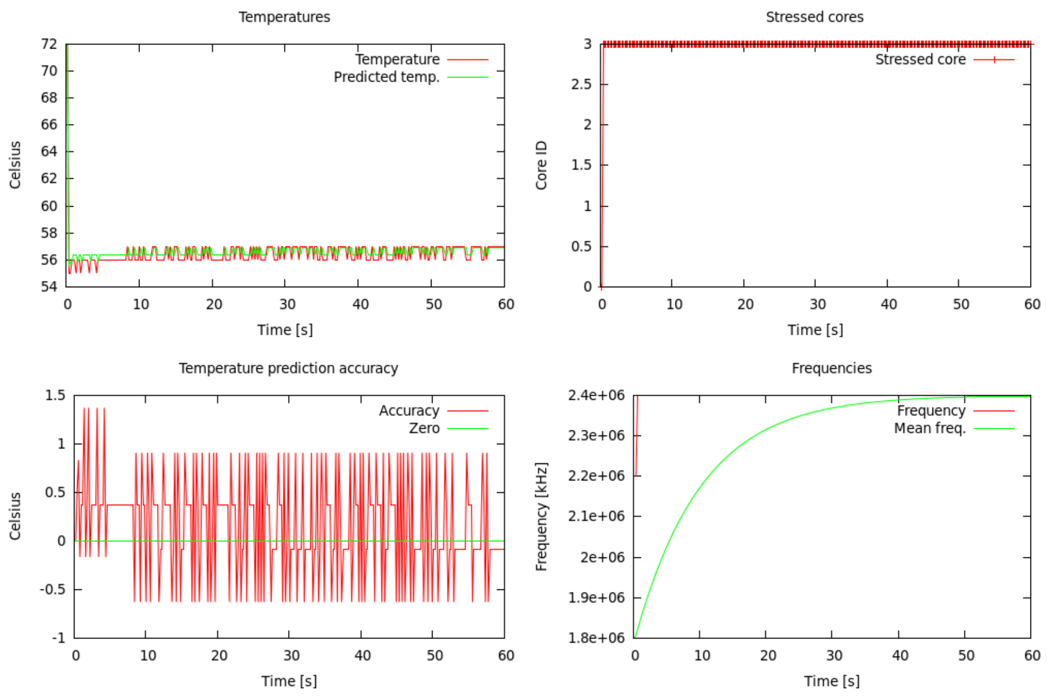


Figure 5.4: Snapshot of the second experiment using the HP notebook, all DTM techniques enabled

The results for the full *CL* framework can be seen in Figure 5.4. In this scenario, the first step that the framework takes is to migrate the task from core 0 to core 3, which essentially is the coolest core on this machine, while also changing the frequency from Turbo-Boost to 2.2 GHz. This enables the framework to cool the CPU down from 72°C to 55°C. In the next call of the controller, the framework decides to switch to 2.4 GHz, since the prediction for 2.4 GHz is 0.25°C still stays under the temperature threshold of 57°C.

Looking at the temperature prediction accuracy, we see that it is below 1°C, 1.4°C at maximum. Considering the resolution of the temperature sensing method being 1°C, the predictions are very close to the actual temperatures.

In this scenario, the *CL* framework was able to effectively cool down the system temperature from 72°C at the beginning to the temperature threshold of 57°C. The average frequency is near 2.4 GHz, which corresponds to the maximum non-turbo frequency of the CPU.

While no DTM method alone would have been able to reduce the temperature that much, the combination of the methods was not only able to achieve this goal, but also to stay at a frequency of 2.4 GHz. The resulting temperature curve is very steady and the predictions are precise.

5.4 Linux On Demand vs. Cool Linux

The purpose of this test is to compare the effect of the developed framework on the core temperatures in comparison to the standard Ubuntu *On Demand Frequency Governor* (from hereby referenced as *ODG*).

Both the framework and the *ODG* were given the same task, to decode 2000 frames using the MJPEG decoder. To measure the performance and temperature for different workloads, the frames per second coming into the dispatcher were limited, this measure is referred to as **dispatcher FPS**.

5.4.1 Measured and Variable Parameters

During the experiment, the average temperature as well as the total execution time are logged. The task was run multiple times, taking the average of all measured values.

To account for varying workload, the experiment was run for differing dispatcher FPS.

5.4.2 Results

The resulting plots for the average temperature as well as the total execution time can be seen in Figure 5.5. The data was conducted using the HP notebook.

The *CL* framework was able to stay at an average CPU temperature below the given threshold of 60°C for every dispatcher FPS tested, in contrast to the Linux ODG, which exceeds the threshold at approximately 250 FPS. While consulting these numbers one has to keep in mind that the temperature seen in the plot is an average, so the maximum temperature of the ODG may be even higher than 60°C before that point.

For our highest workload tested (5000 FPS), which causes the CPU utilization to be 100%, the *CL* framework executed the given task 16.7°C cooler than the *ODG*, which heated the CPU up to an average of 74.6°C. The total execution time of the framework rose from 15.7s to 21.9s, which represents a time increase of 28.4%.

For the the dispatcher FPS being lower than 220, the *CL* framework runs at an increased CPU temperature compared to the the Linux *ODG*, but it also executes the task much faster. This behaviour can be explained

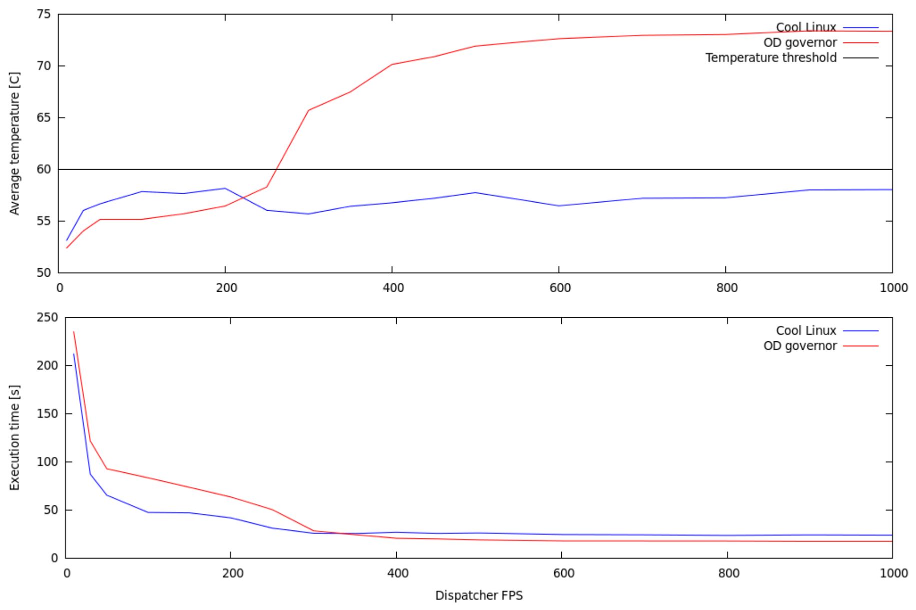


Figure 5.5: Linux On Demand vs. Cool Linux using HP

by the framework choosing the highest frequency if it is possible to stay below the temperature threshold, whereas the Linux *ODG* rapidly changes its frequency, mostly using the highest and the lowest one alternately. For the *ODG*, this is caused by the frames for the worker arriving at much slower rate than the CPU could handle and therefore creating demand bursts. Running at the lowest frequency causes the CPU to stay cooler, but it also decreases the performance of the governor.

However, the two most interesting part of these plots can be seen between the point where the average temperature of both algorithms meet and the point where the performance of both methods is the same. The first point can be found at approximately 220 FPS. While the average temperature is the same for both algorithms, the *CL* framework executes the given task 36% faster than the Linux *ODG*. At a workload of approximately 340 FPS, the *CL* framework executes the given task as fast as the Linux *ODG*, while staying 12°C cooler on average.

Between these two points, the developed *CL* framework does not only cause the CPU to stay cooler than the standard Linux *ODG*, but it also executes the task faster.

5.5 Summary and Conclusion

The three experiments conducted show that the developed *CL* framework behaves as expected and intended. The effect of each DTM technique was analysed and can be clearly seen in the resulting data.

The unique combination of the DTM techniques used as well as the developed temperature prediction algorithm ensure a huge reduction of the CPU's temperature. For certain workloads the *CL* framework even outperforms the default *Linux On Demand Governor*, while still having a clear advantage in terms of temperature.

6 Conclusion and Outlook

6.1 Conclusion

The resulting *CL* framework integrates the three DTM techniques, namely frequency scaling, task migration and scheduled sleep into a unique framework. It makes use of an automatically calibrated temperature predictor and decides dynamically based on those predictions, the current temperatures and other program parameters which combination of DTMs to use next.

A temperature predicting algorithm has been developed and its required machine-dependant parameters are automatically calibrated in a separate offline tool. The algorithm is based on the current CPU core temperatures as well as on information about which core is going to be stressed during the next time period. Furthermore the algorithm is able to predict the temperatures while multiple CPU cores are stressed and a first simple online recalibration algorithm for one parameter is included.

All three DTM techniques used were evaluated, providing useful information about their feasibility. Sleeping can be used to cool down the CPU in a very short amount of time, but the temperature jumps up instantly afterwards and therefore, sleeping should only be used for small time differences in the order of milliseconds. Migration performs very well in terms of cooling the system down, however the predictor would have to be adjusted to take the heat produced during the migration into account. Frequency scaling turns out to be the most efficient way to cool the system down by a small amount of degrees while not overly negatively influencing the system's performance.

The unique combination of DTM methods and temperature prediction of the *Cool Linux* framework enables a Linux system to run tasks with very high workload up to 20°C cooler than the standard Linux On Demand Governor at its peak and 18°C cooler on average. For a certain workload, the *CL* framework even outperforms the default governor in terms of performance while still keeping the system cooler.

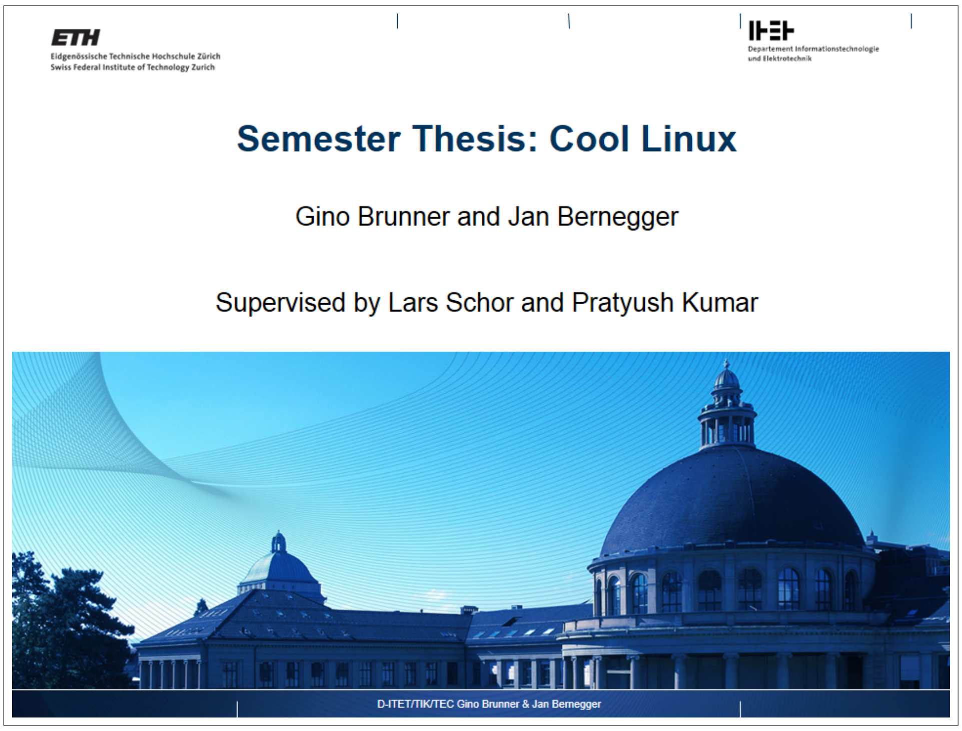
6.2 Outlook

Live Parameter Calculation

Offline calculation of the temperature model's parameters has several drawbacks. For the obtained parameters remaining constant, they cannot react to environmental changes like increased room temperature or direct sunlight. Also, the framework cannot easily handle predictions for varying average CPU loads from different tasks (the MJPEG decoder might use the CPU in another manner than a prime finding program).

Live parameter calculation, however, could solve these problems. An online (re-)calculation of the parameters could react quickly to environmental or task changes. In order for this to work, all required temperature parameters τ, T_0 and T_∞ would have to be (re-)calculated at runtime. A first draft of a self-learning algorithm, in this thesis referred to as *live τ correction*, has already been developed and integrated in the framework as an experimental feature.

A
Presentation Slides



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

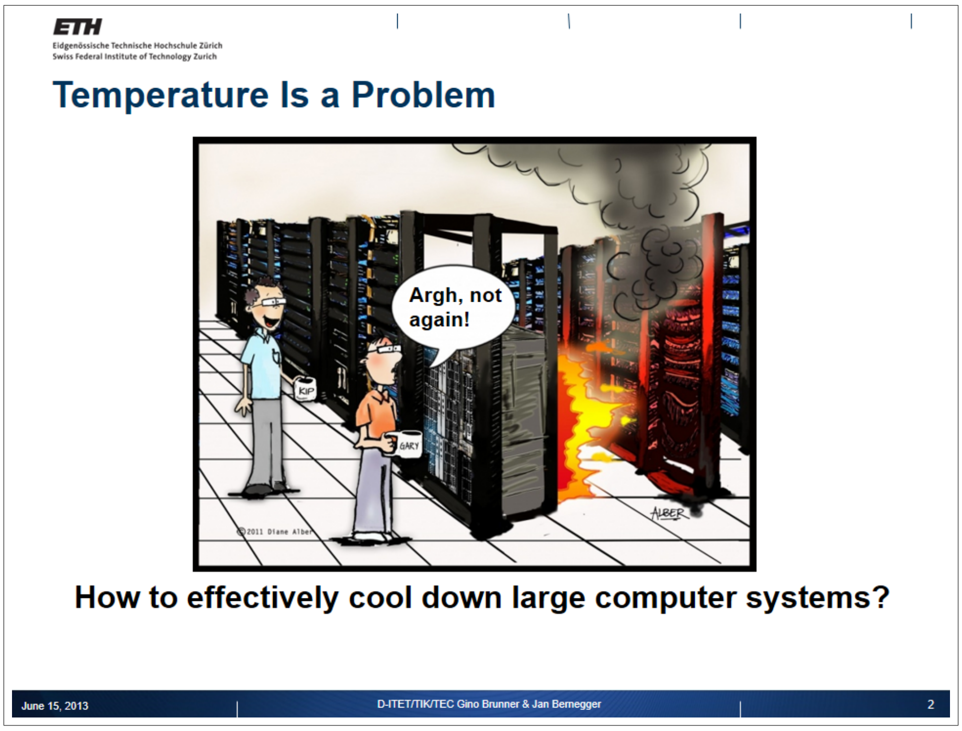
Departement Informationstechnologie und Elektrotechnik

Semester Thesis: Cool Linux

Gino Brunner and Jan Bernegger

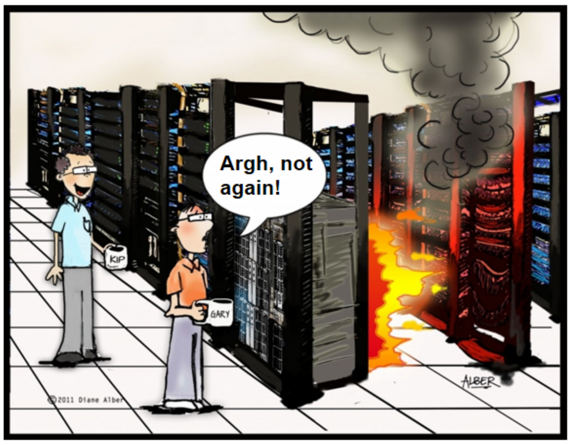
Supervised by Lars Schor and Pratyush Kumar

D-ITET/TIK/TEC Gino Brunner & Jan Bernegger



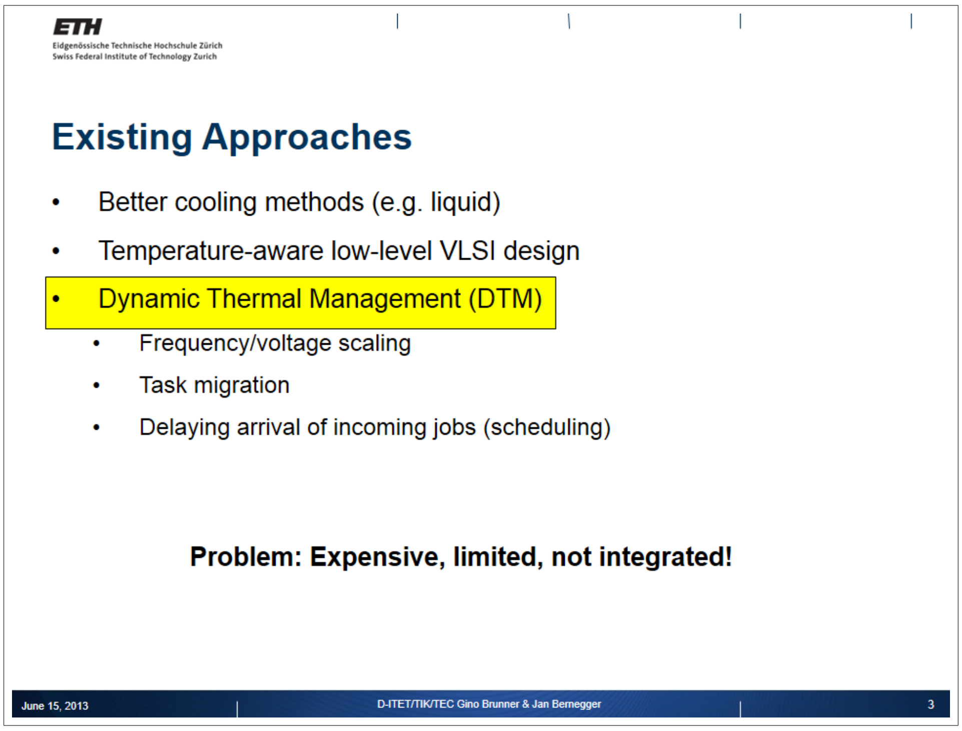
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Temperature Is a Problem



How to effectively cool down large computer systems?

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 2



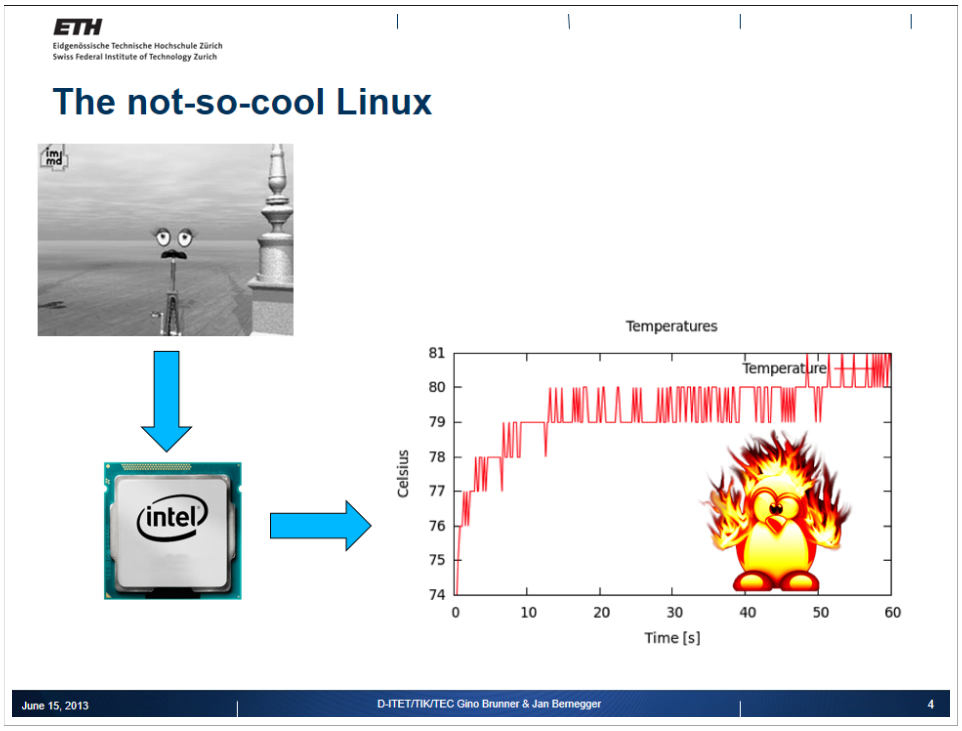
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Existing Approaches

- Better cooling methods (e.g. liquid)
- Temperature-aware low-level VLSI design
- Dynamic Thermal Management (DTM)
  - Frequency/voltage scaling
  - Task migration
  - Delaying arrival of incoming jobs (scheduling)

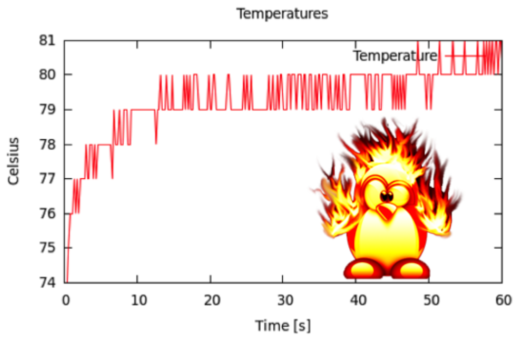
Problem: Expensive, limited, not integrated!

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 3

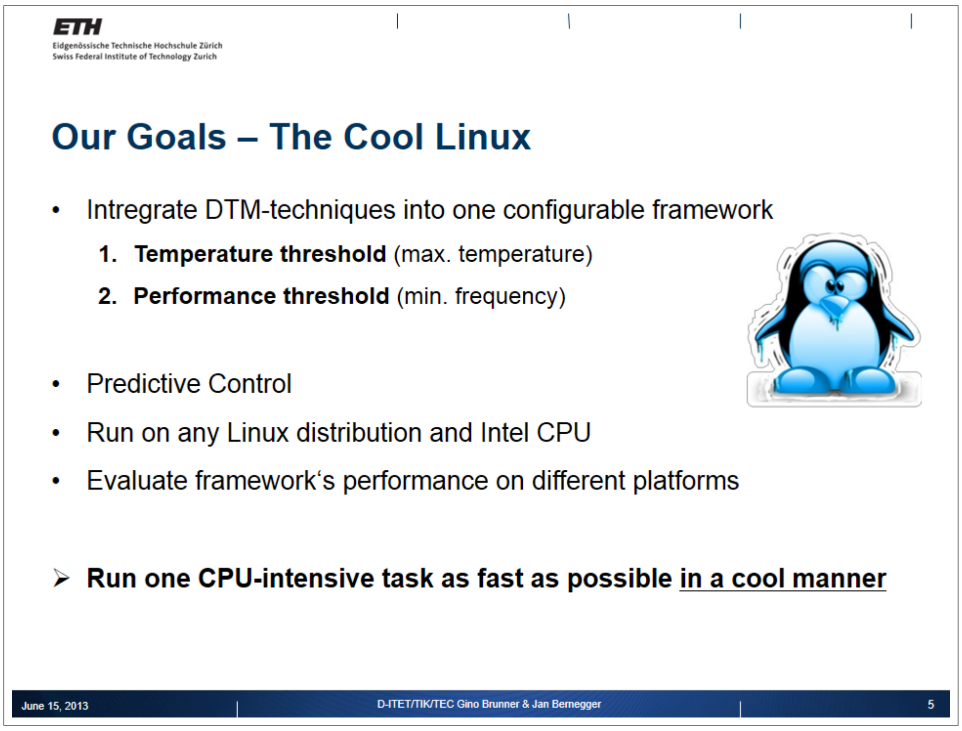


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The not-so-cool Linux



June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 4



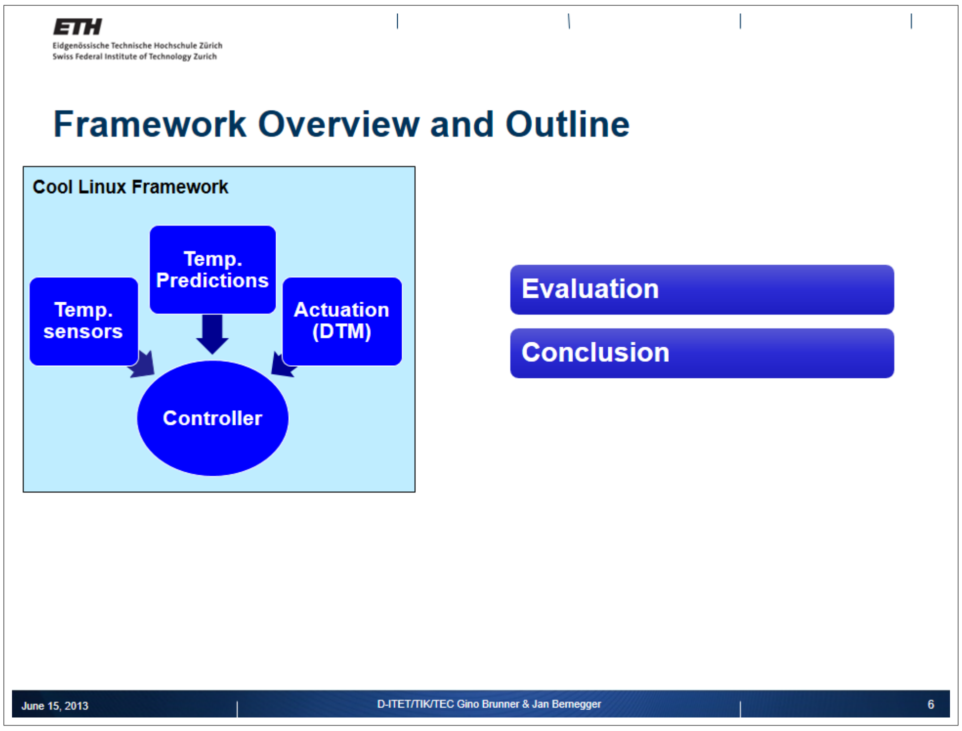
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Our Goals – The Cool Linux

- Intregrate DTM-techniques into one configurable framework
  1. **Temperature threshold** (max. temperature)
  2. **Performance threshold** (min. frequency)
- Predictive Control
- Run on any Linux distribution and Intel CPU
- Evaluate framework‘s performance on different platforms

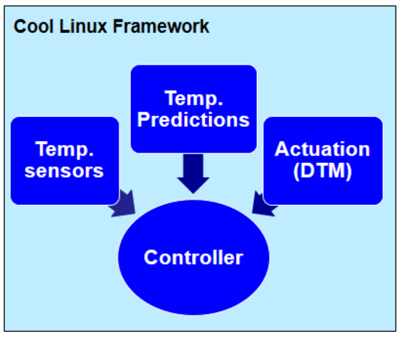
➢ **Run one CPU-intensive task as fast as possible <u>in a cool manner</u>**

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 5



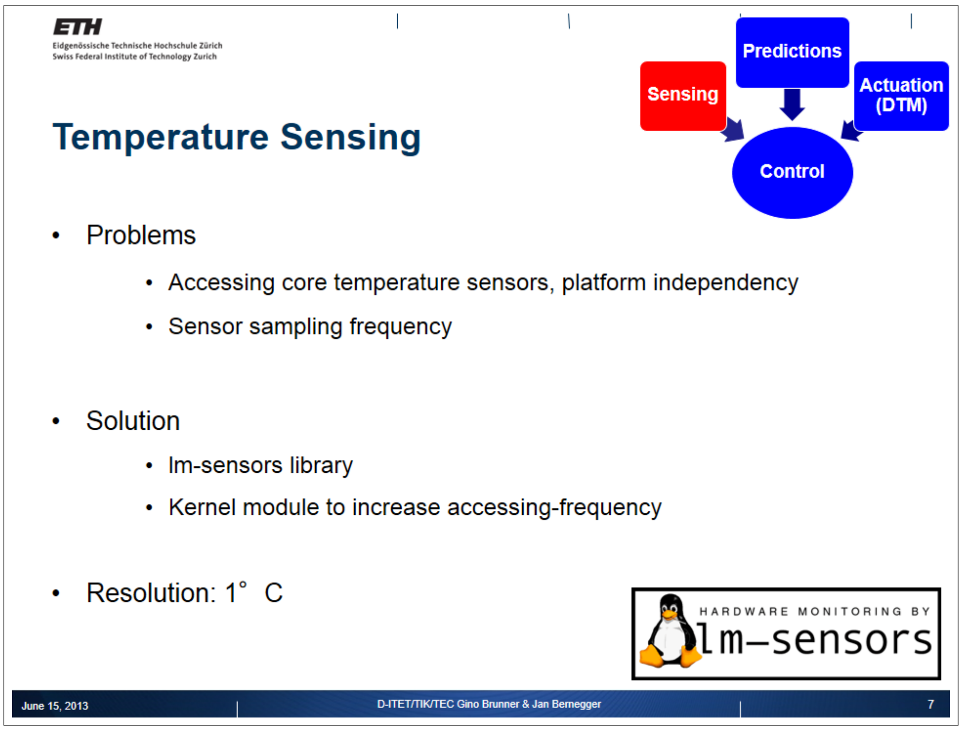
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Framework Overview and Outline



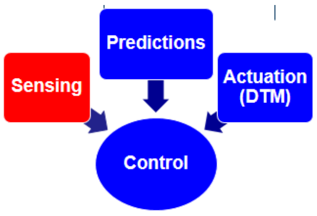
- Evaluation
- Conclusion

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 6



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

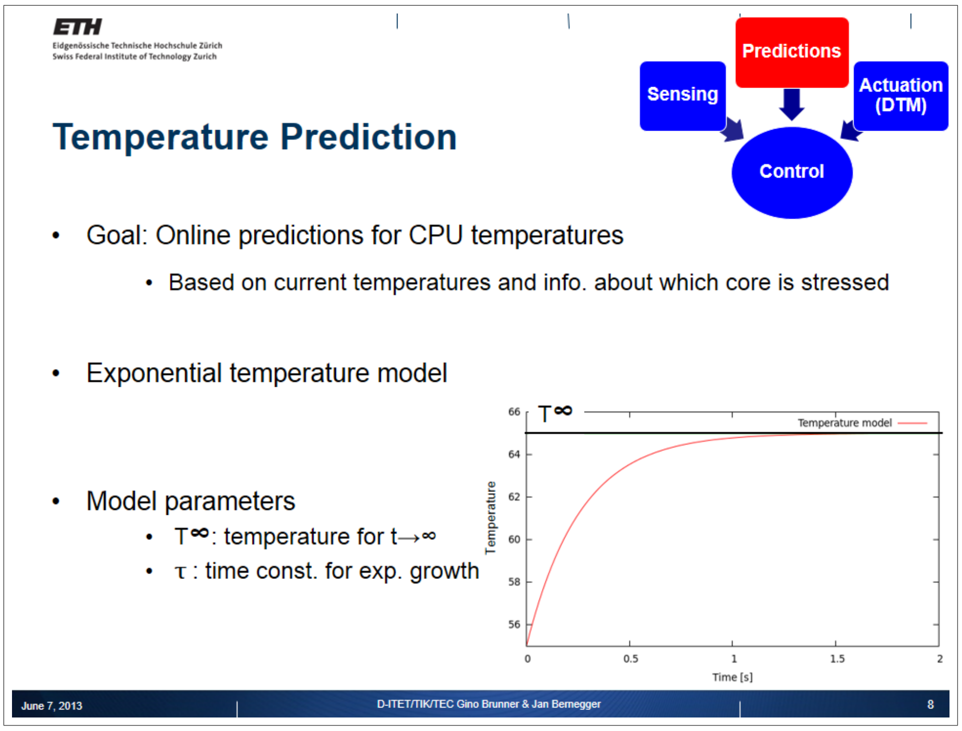
Temperature Sensing



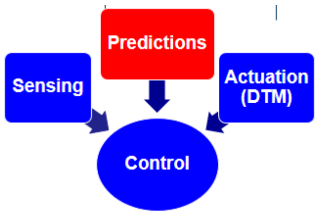
- Problems
  - Accessing core temperature sensors, platform independency
  - Sensor sampling frequency
- Solution
  - lm-sensors library
  - Kernel module to increase accessing-frequency
- Resolution: 1° C

HARDWARE MONITORING BY lm–sensors

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 7

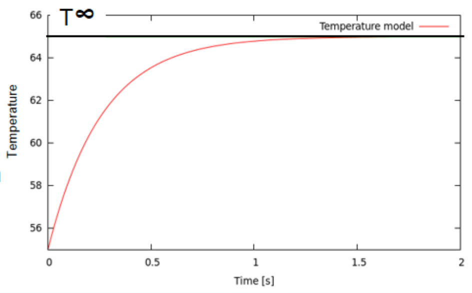


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

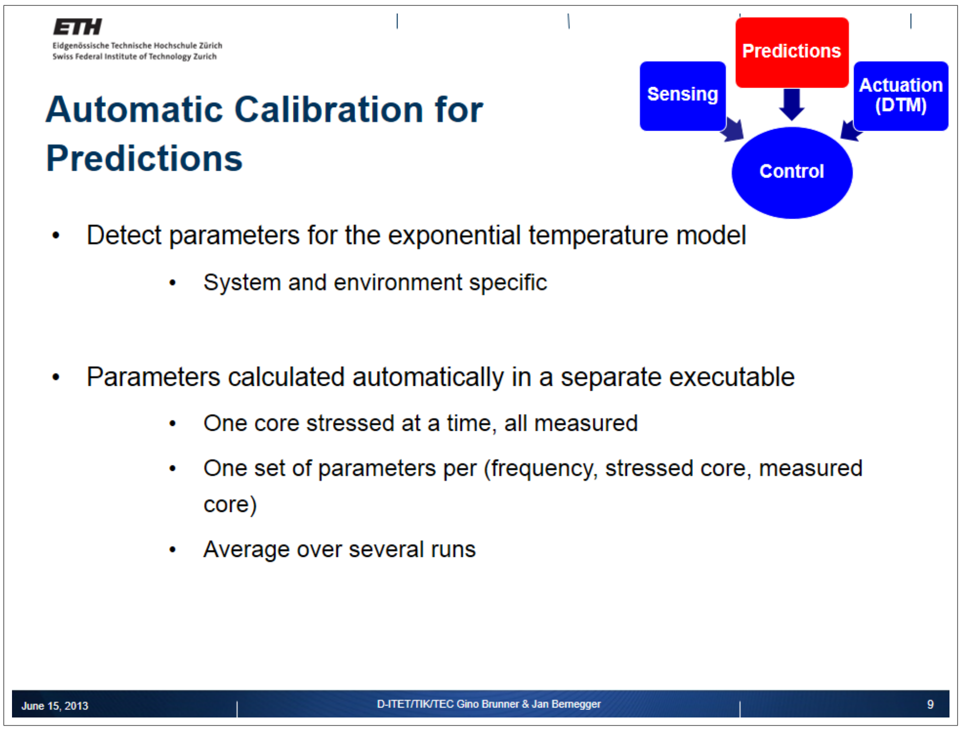


Temperature Prediction

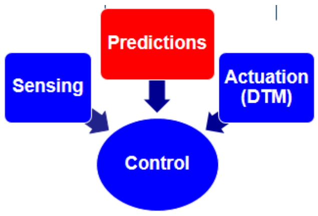
- Goal: Online predictions for CPU temperatures
  - Based on current temperatures and info. about which core is stressed
- Exponential temperature model
- Model parameters
  - T_∞: temperature for $t \rightarrow \infty$
  - τ : time const. for exp. growth



June 7, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 8



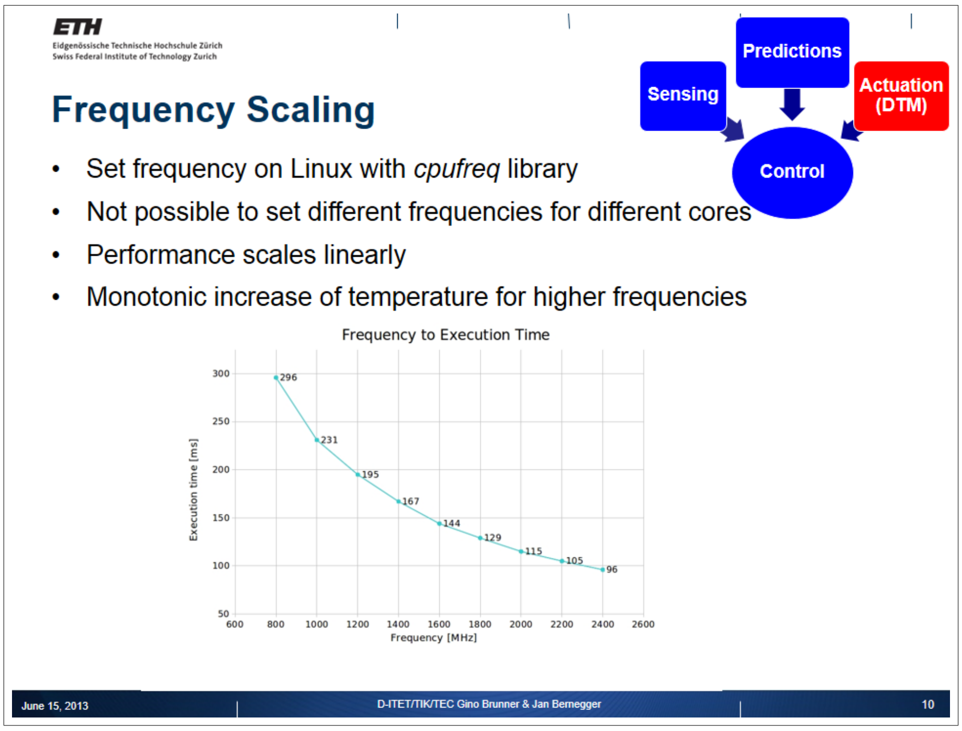
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



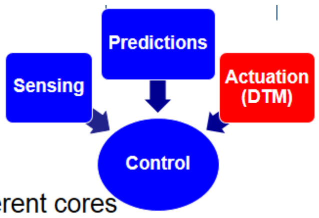
Automatic Calibration for Predictions

- Detect parameters for the exponential temperature model
  - System and environment specific
- Parameters calculated automatically in a separate executable
  - One core stressed at a time, all measured
  - One set of parameters per (frequency, stressed core, measured core)
  - Average over several runs

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 9

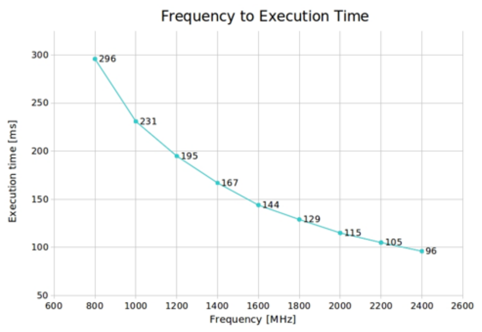


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

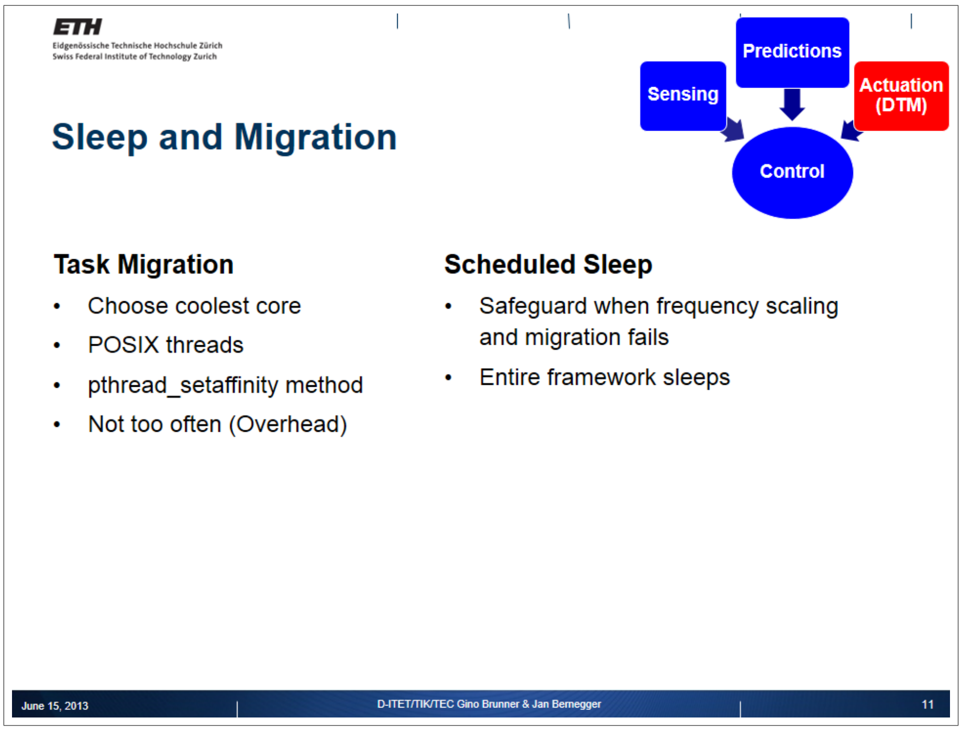


Frequency Scaling

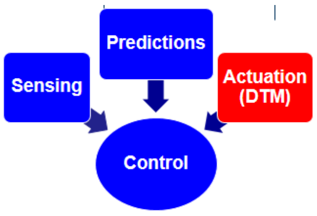
- Set frequency on Linux with *cpufreq* library
- Not possible to set different frequencies for different cores
- Performance scales linearly
- Monotonic increase of temperature for higher frequencies



June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 10



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Sleep and Migration

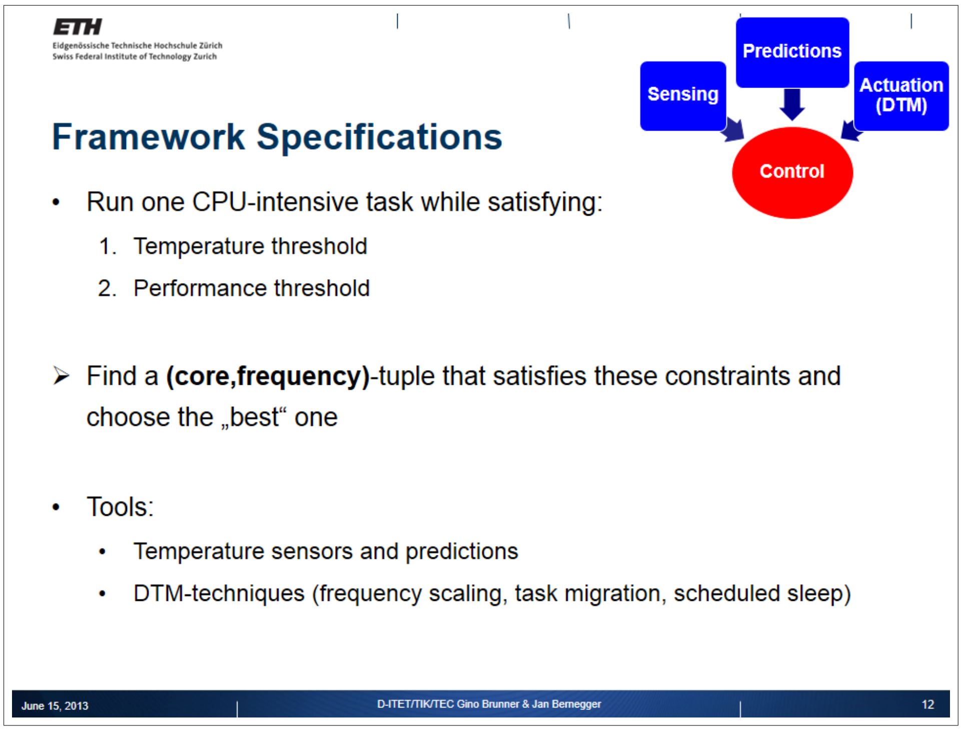
Task Migration

- Choose coolest core
- POSIX threads
- pthread_setaffinity method
- Not too often (Overhead)

Scheduled Sleep

- Safeguard when frequency scaling and migration fails
- Entire framework sleeps

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 11



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

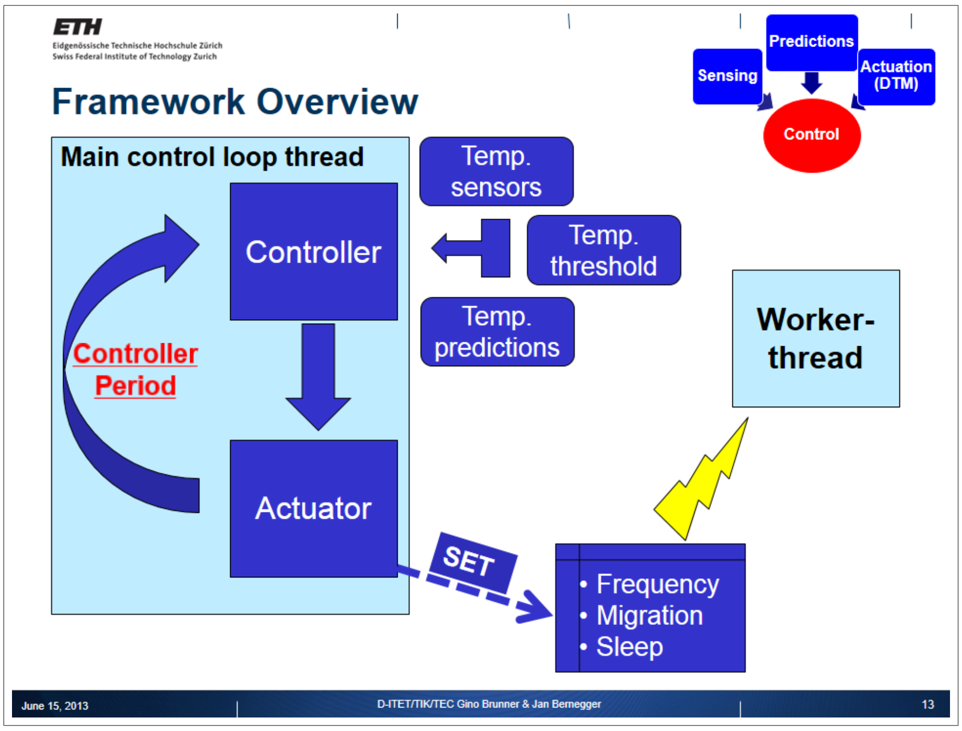
Framework Specifications

- Run one CPU-intensive task while satisfying:
  1. Temperature threshold
  2. Performance threshold

➢ Find a **(core,frequency)**-tuple that satisfies these constraints and choose the „best" one

- Tools:
  - Temperature sensors and predictions
  - DTM-techniques (frequency scaling, task migration, scheduled sleep)

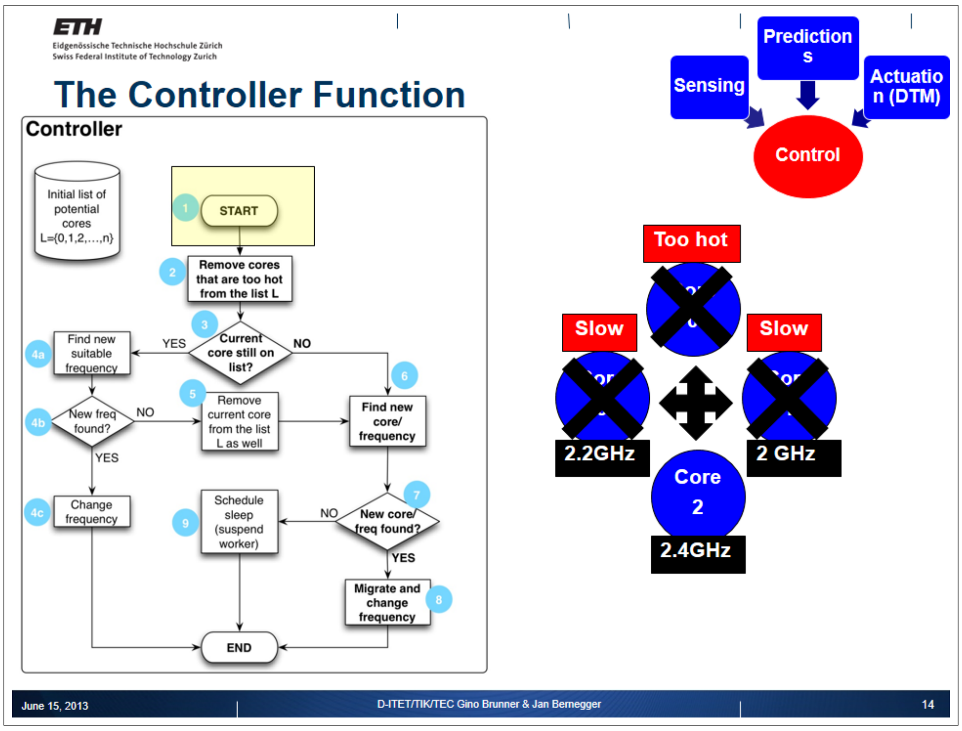
June 15, 2013 D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 12



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Framework Overview

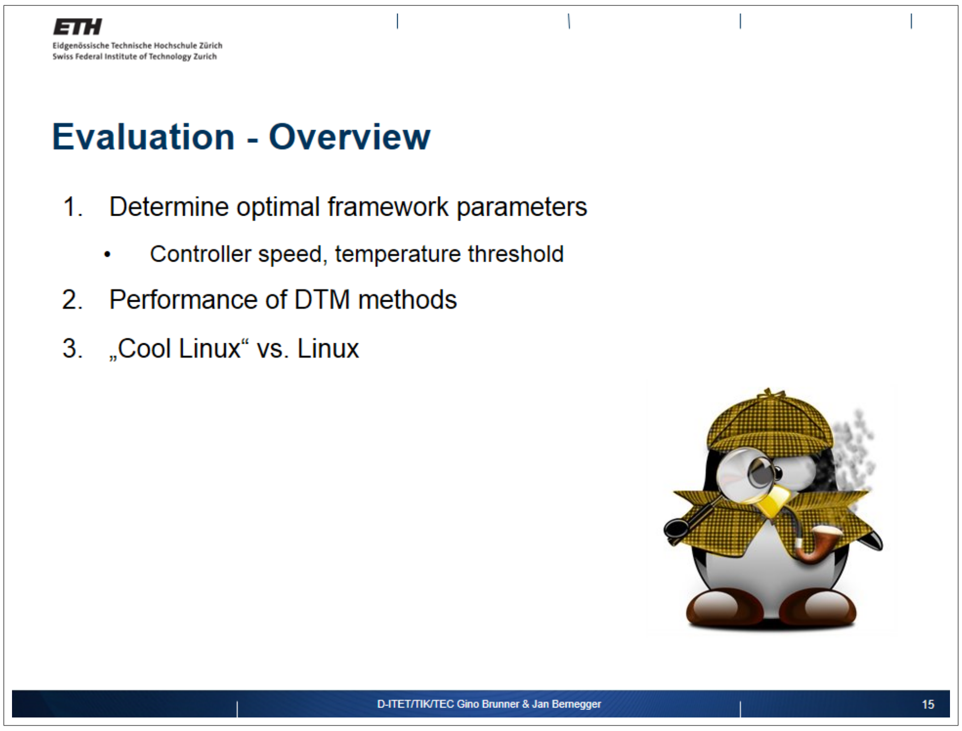
June 15, 2013 D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 13



The Controller Function

Controller
Initial list of potential cores L={0,1,2,...,n}
1 START
2 Remove cores that are too hot from the list L
3 Current core still on list?
YES
NO
4a Find new suitable frequency
4b New freq found?
NO
YES
4c Change frequency
5 Remove current core from the list L as well
6 Find new core/ frequency
7 New core/ freq found?
NO
YES
8 Migrate and change frequency
9 Schedule sleep (suspend worker)
END
Predictions
Sensing
Actuation (DTM)
Control
Too hot
Slow
Slow
2.2GHz
2 GHz
Core 2
2.4GHz

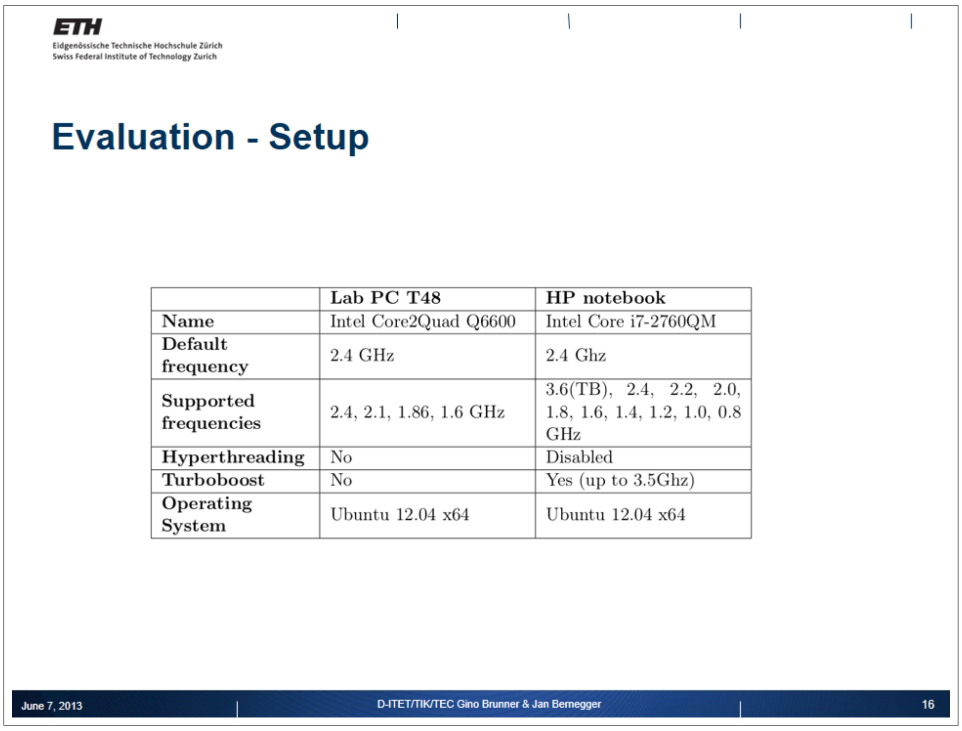
June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 14



Evaluation - Overview

1. Determine optimal framework parameters
   - Controller speed, temperature threshold
2. Performance of DTM methods
3. „Cool Linux“ vs. Linux

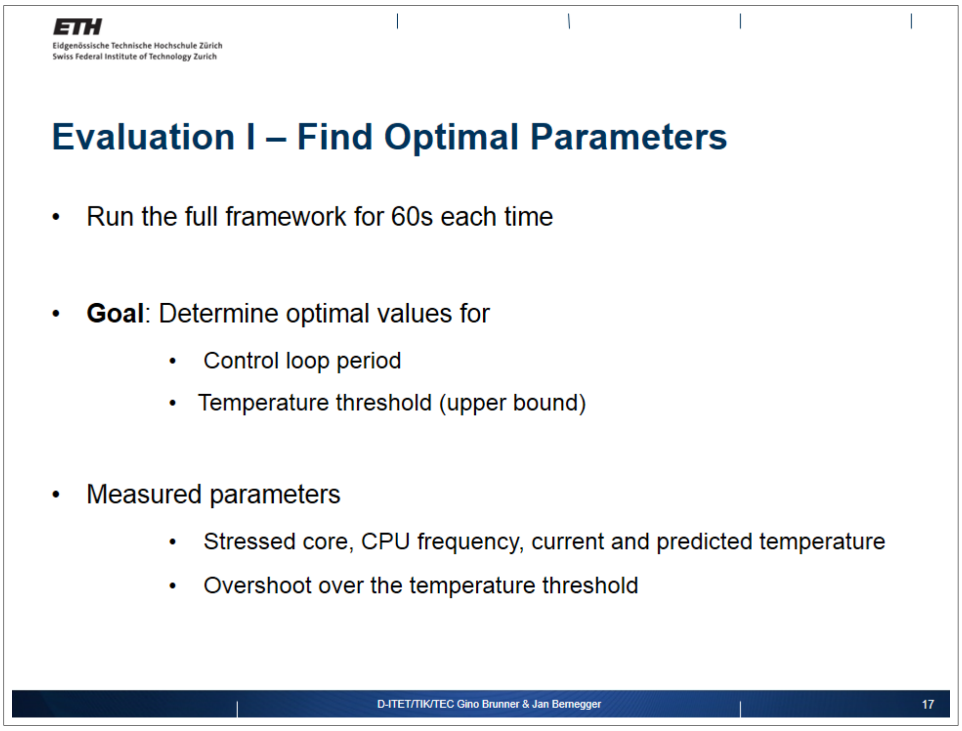
D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 15


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation - Setup

	Lab PC T48	HP notebook
Name	Intel Core2Quad Q6600	Intel Core i7-2760QM
Default frequency	2.4 GHz	2.4 Ghz
Supported frequencies	2.4, 2.1, 1.86, 1.6 GHz	3.6(TB), 2.4, 2.2, 2.0, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8 GHz
Hyperthreading	No	Disabled
Turboboost	No	Yes (up to 3.5Ghz)
Operating System	Ubuntu 12.04 x64	Ubuntu 12.04 x64

June 7, 2013 D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 16


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation I – Find Optimal Parameters

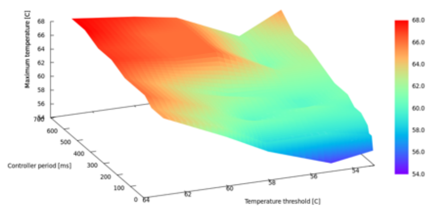
- Run the full framework for 60s each time
- **Goal**: Determine optimal values for
  - Control loop period
  - Temperature threshold (upper bound)
- Measured parameters
  - Stressed core, CPU frequency, current and predicted temperature
  - Overshoot over the temperature threshold

D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 17

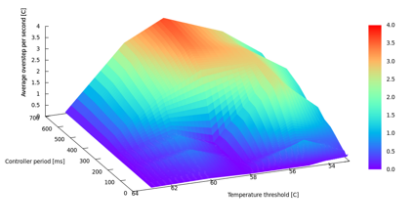
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation I – Results

Maximum temperature



Average temperature overshoot per second

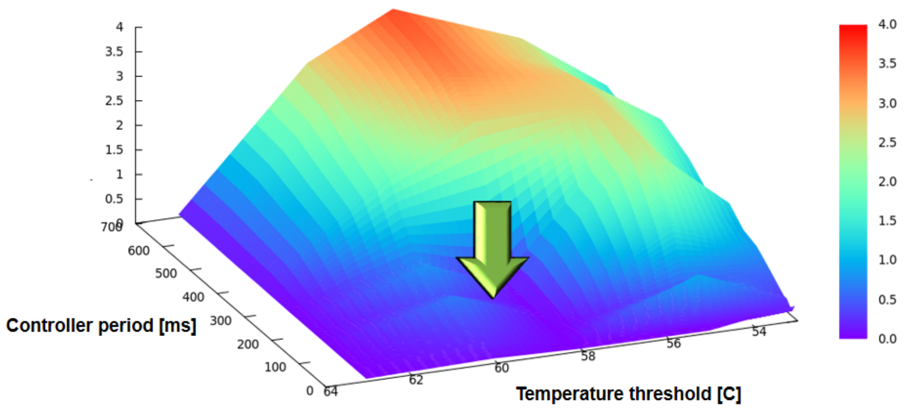


D-ITET/TIK/TEC Gino Brunner & Jan Beregger 18

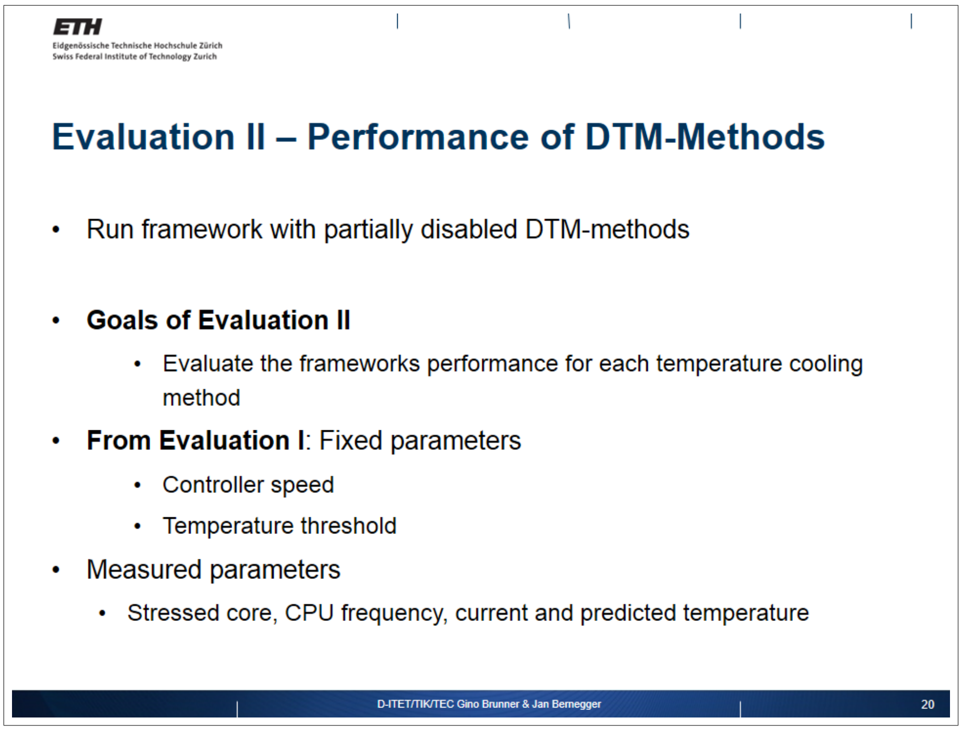
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation I – Results

Average overshoot per second



D-ITET/TIK/TEC Gino Brunner & Jan Beregger 19

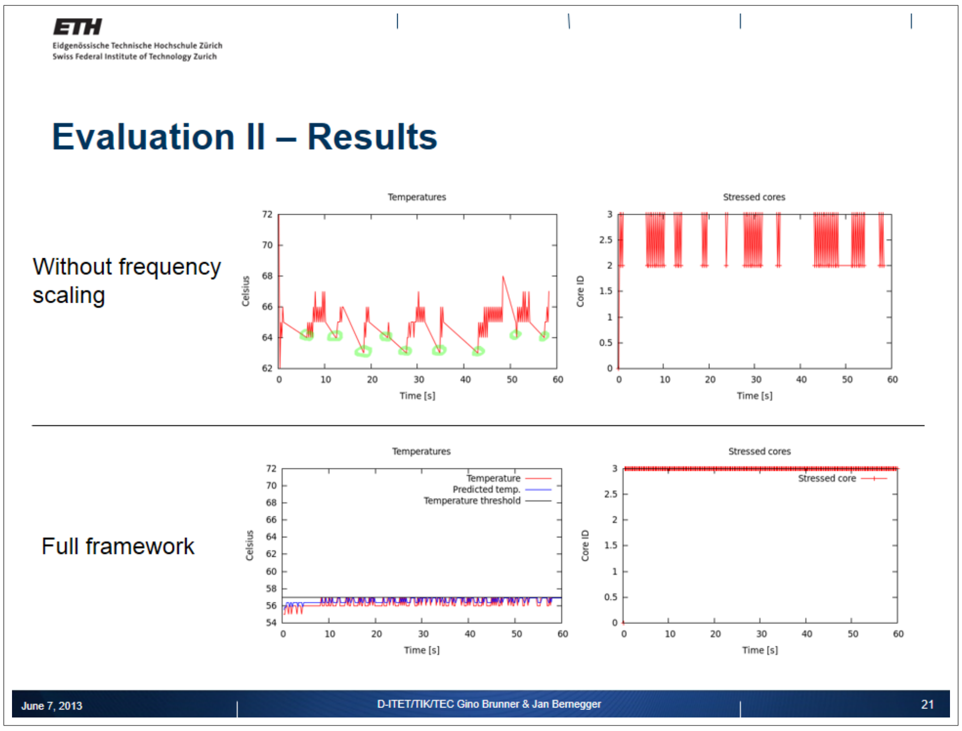


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation II – Performance of DTM-Methods

- Run framework with partially disabled DTM-methods
- **Goals of Evaluation II**
  - Evaluate the frameworks performance for each temperature cooling method
- **From Evaluation I**: Fixed parameters
  - Controller speed
  - Temperature threshold
- Measured parameters
  - Stressed core, CPU frequency, current and predicted temperature

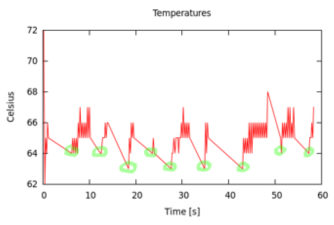
D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 20

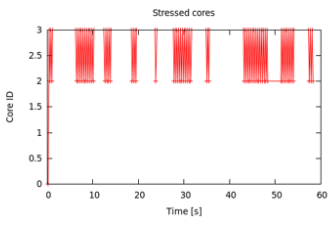


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

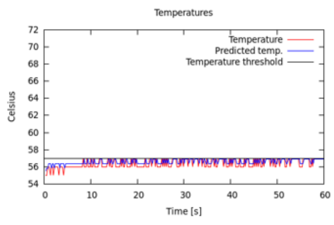
Evaluation II – Results

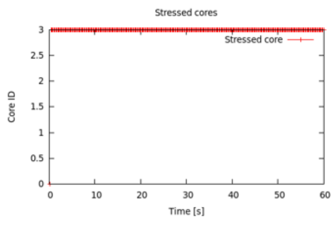
Without frequency scaling



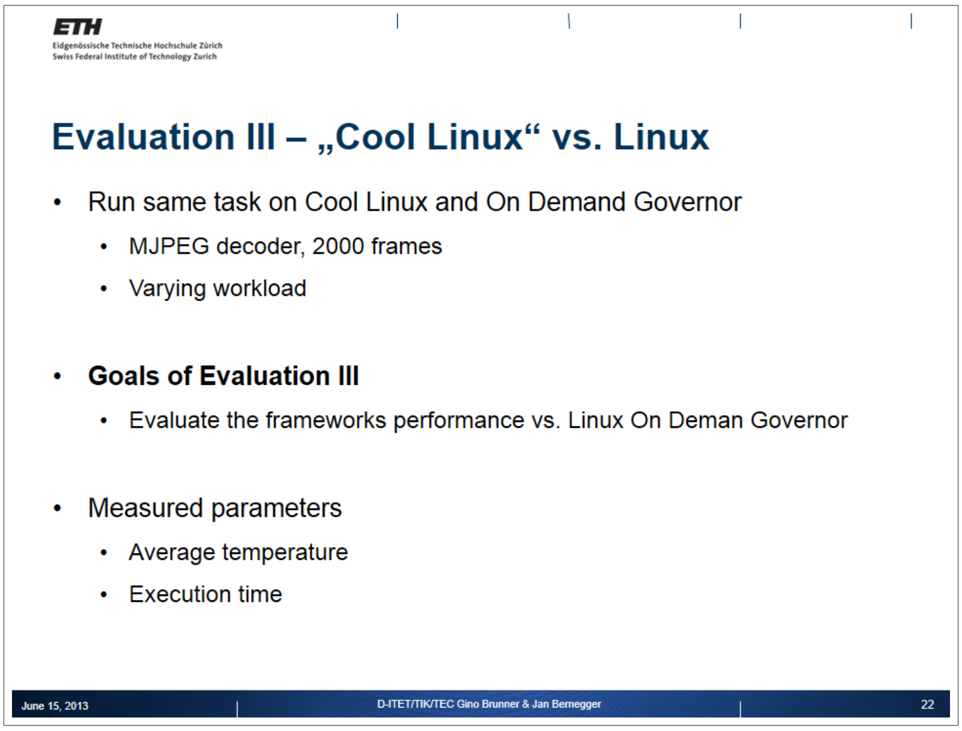


Full framework





June 7, 2013 D-ITET/TIK/TEC Gino Brunner & Jan Bernegger 21



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation III – „Cool Linux“ vs. Linux

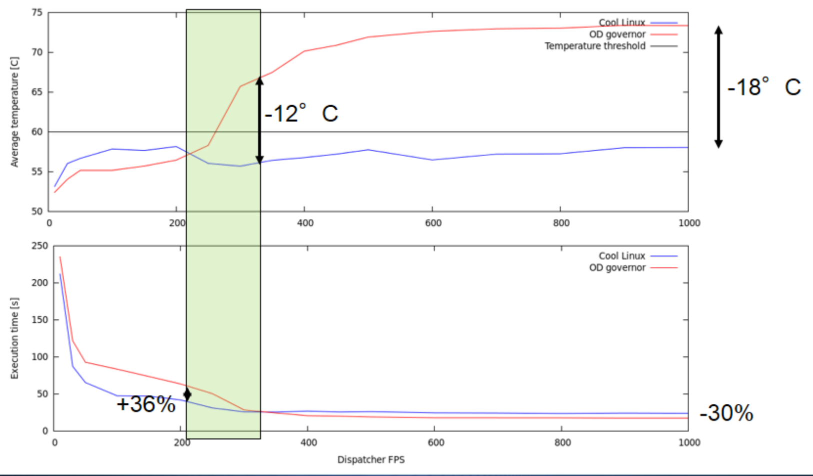
- Run same task on Cool Linux and On Demand Governor
  - MJPEG decoder, 2000 frames
  - Varying workload
- **Goals of Evaluation III**
  - Evaluate the frameworks performance vs. Linux On Deman Governor
- Measured parameters
  - Average temperature
  - Execution time

June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 22

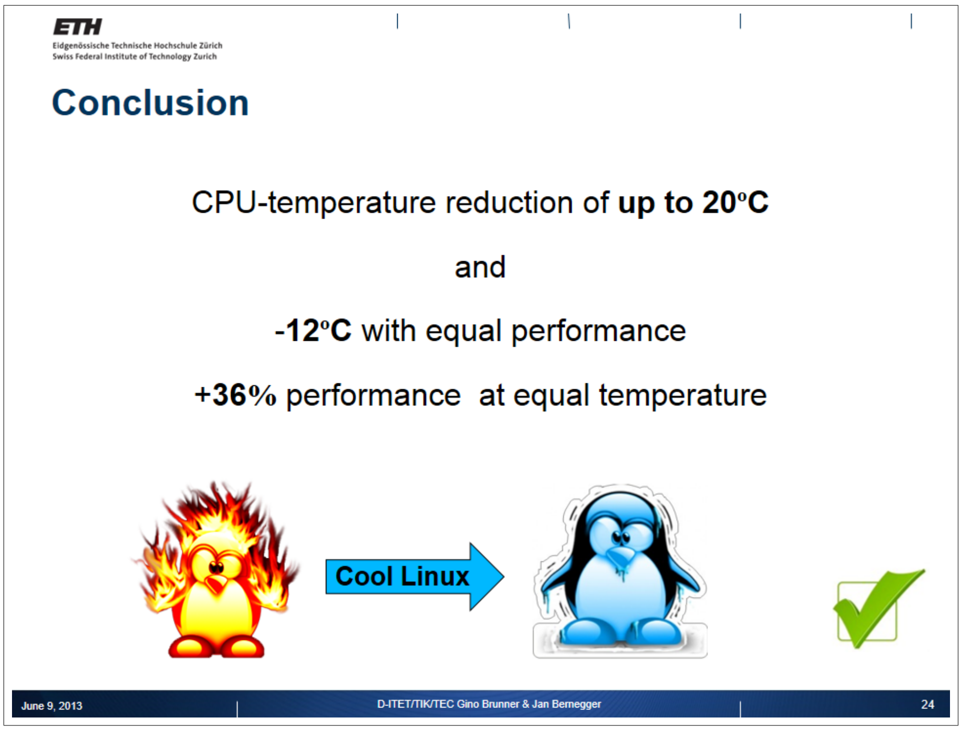
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation III – Results

Average time to decode 2000 frames using MJPEG



June 15, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 23



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Conclusion

CPU-temperature reduction of **up to 20°C**

and

-12°C with equal performance

+36% performance at equal temperature

June 9, 2013 | D-ITET/TIK/TEC Gino Brunner & Jan Bernegger | 24

B

Cool Linux HowTo

B.1 Required Libraries and Configurations

B.1.1 Frequency Scaling

In order to do frequency scaling on Linux, one has to install the `cpufreq` library. The library and corresponding header files can be downloaded from `http://packages.debian.org/stable/libdevel/libcpufreq-dev`, or by using

```
sudo apt-get install libcpufreq-dev
```

If everything worked fine, it should now be possible to include `cpufreq.h` in your C-code. The `CL` framework itself uses some wrapper functions, located in `changeFreq.c`, that make use of the `cpufreq` functions internally.

B.1.2 CPU Temperatures

To be able to access the systems temperatures, one can use `lm-sensors`. The following command line installs the required library and header files on a Ubuntu based machine.

```
sudo apt-get install libsensors libsensors-dev
```

After the installation, an initialisation script has to be run once to detect the device's chipset.

```
sudo yes | sensors-detect
```

However, the library is only able to get a new sample of the temperature sensors each 1.5 seconds. The process of how to increase the accessing frequency is described in Appendix C.

B.2 Compiler and Linker Settings

This section gives an overview over the linker and compiler settings that are needed to sucessfully compile the program:

Compiler:

VIEWER Set this variable to enable X11 video output of the MJPEG decoder

Linker (-l):

libcpufreq Enables the use of the `cpufreq` library

pthread Required for task migration

libsensors Required for accessing CPU temperatures

X11 Enables to display the decoded video

B.3 How to Run the Framework

After successful compilation, the framework can be run from the command line. Use the `-h` option to learn about the configurable parameters.

```
./CLFramework -h
```

C

Increasing update frequency

First get linux sources:

```
apt-get source linux-image-$(uname -r)
cp linux-3.2.0/drivers/hwmon/coretemp.c .
```

Next, edit the code in `coretemp.c` as follows:

```
#define MS 1

[...]

static ssize_t show_temp(...) {
  [...]
  /* Check whether the time interval has elapsed */
  if (!tdata->valid || time_after(jiffies, tdata->last_updated +
    HZ/HZ*MS)) {
  [...]
  }
  [...]
}
```

Listing C.1: Changed code in coretemp.c

Create the following Makefile:

```
obj-m = coretemp.o
KVERSION = $(shell uname -r)

all:
  make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules
clean:
  make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean

install:
  mv /lib/modules/$(KVERSION)/kernel/drivers/hwmon/coretemp.ko /
    lib/modules/$(KVERSION)/kernel/drivers/hwmon/coretemp.ko.old
  cp coretemp.ko /lib/modules/$(KVERSION)/kernel/drivers/hwmon/
  depmod -a

uninstall:
  mv /lib/modules/$(KVERSION)/kernel/drivers/hwmon/coretemp.ko.
    old /lib/modules/$(KVERSION)/kernel/drivers/hwmon/coretemp.ko
  depmod -a
```

Listing C.2: Makefile

Finally, execute:

```
make
sudo make install
```

The module is available after rebooting the system.

D
Interfaces

D.1 Interface for accessing core temperatures

To get easy access to the core temperatures, we developed the class `CoreTemperatures`. When creating an instance of this class, libsensors is automatically initialized and the number of cores is detected. The temperatures and the number of cores are accessible using the following functions:

Function	**Description**
int **getNumberOfCores**()	returns the number of cores
int **getChipTemp**()	returns the temperature of the whole CPU if available and -1 if it is not
int **getCoreTemp**(uint n)	returns the temperature of an individual core n
void **cleanUp**()	needs to be called in order to securely shut down libsensors; is automatically called by the destructor

D.2 Interface for temperature prediction

For easy access to the temperature predictions, we developed the class `TemperatureFrequencyPredictor`. Internally, it stores a map of all Frequencies with their individual `TemperaturePredictor`s. The parameters are automatically loaded from `/etc/CoolLinux` if TemperatureParameterTester had previously been executed.

Function	**Description**
void **setStressMap**(CoreStressMap* map)	gives a pointer of a StressMap to all TemperaturePredictors to be used for all future predictions
float **predictCoreTemp**(int freq, int* actualTemps, int core, float timeDiff)	predicts the temperature for a given frequency `freq`, an array of all current temperatures `actualTemps`, the `core` for which it should be predicted and the time difference `timeDiff` (in seconds)
float* **predictCoreTemp**(int* actualTemps, int core, float timeDiff)	predicts the temperature for all available frequencies `freq` given an array of all current temperatures `actualTemps`, the `core` for which the prediction should be made and the time difference `timeDiff` (in seconds)
float **predictCoolTime**(int freq, int* actualTemps, int core, float newTemp)	predicts the time required for cooling down to `newTemp` for a given frequency `freq`, an array of all current temperatures `actualTemps` and the `core` for which the prediction should be made
float* **predictCoolTime**(int* actualTemps, int core, float newTemp)	predicts the time required for cooling down to `newTemp` for all available frequencies given an array of all current temperatures `actualTemps` and the `core` for which the prediction should be made
void **recalculateTau**(int freq, float startTemp, float endTemp, float timeDiff, int measureCore, int stressedCore)	recalculates the tau for the frequency `freq` for a certain `measureCore` and `stressedCore` combination, given the `startTemp` and `endTemp` after `timeDiff` (in seconds)

D.3 Interface for Frequency Scaling

For easier use of frequency scaling methods, we implemented several functions that internally use the *cpufreq*-library functions, but externally, are taylored to the needs of our framework. To use those functions, include `changeFreq.h` in your C-code. `changeFreq.c` does not implement all `cpufreq` functions. If more are needed in the future, they can easily be implemented. `changeFreq.c` provides the following methods:

Function	**Description**
void **setFrequency**(int freq, int cpuNum)	Sets the operating frequency to `freq`. The function also needs the number of CPU cores in the system as parameter `cpuNum`.
int **getFrequency**(int core, int mode)	Returns the operating frequency of a specified core. However, `core` can always be set to 0, since the frequency is the same for all cores. The parameter `mode` specifies if the kernel frequency ($mode = 0$) or the hardware frequency ($mode = 1$) should be returned.
void **setGovernor**(char * governor, int cpuNum)	Sets the Linux frequency governor to `governor`. The number of CPU cores in the system has to be provided in `cpuNum`.
vector<int> **getAvailableFreq**(unsigned int cpuNum)	Returns a vector of all available frequencies of the system. `setFrequency` can only set the frequency to one of those values. The number of CPU cores in the system has to be provided in `cpuNum`.

Bibliography

[1] *Mobile 3rd Generation Intel Core Processor Familiy (...) Datasheet Volume 1 of 2* , 2nd ed., Intel, Jan. 2013.

[2] "Wikipedia", “lm-sensors,” http://en.wikipedia.org/wiki/Lm_sensors.

[3] B. Mathew, “The Perception Processor,” Ph.D. dissertation, School of Computing, University of Utah, May 2004.

[4] T. Blog", “How long does it take to make a context switch?” http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, Nov. 2010.

[5] C. Li, C. Ding, and K. Shen, “Quantifying the cost of context switch,” in *Proceedings of the 2007 workshop on Experimental computer science*, ser. ExpCS ’07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281702

[6] D. Dice, “Using MWAIT in spin loops,” https://blogs.oracle.com/dave/resource/mwait-blog-final.txt, Nov. 2011.