Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Bachelor-Thesis

Estimation of Power Consumption of DVFS-Enabled Processors

Name: Christian Seyda
Student number: 2600280
Supervised by: Prof. Thomas Ludwig
Timo Minartz
Date of submission: Heidelberg, 28th March 2011

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

28. März 2011, ……

Abstract

Saving energy is nowadays a critical factor, especially for data centers or high performance clusters which have a power consumption of several mega watts. The simple use of component energy saving mechanisms is not always possible, because this can lead to performance degradation. For this reason, high performance clusters are mainly not using them, even in low utilization phases. Modelling the power consumption of a component based on specific recordable values can help to find ways of saving energy or predicting the power consumption after replacing the component with a more efficient one.

One of the main power consumer on a recent system is the processor. This thesis presents a model of the power consumption of a processor based on its frequency and voltage. Comparisons with real world power consumption were done to evaluate the model. Furthermore a tracing library was extended to be able to log the processor frequency and idle states if available. Using the presented model and the trace files, a power estimator has been implemented being able to estimate the power consumption of the processor in the given trace file—or a more energy efficient processor—helping to motivate the usage of power saving mechanisms and energy efficient processors, and showing the long term potential for energy saving.

Contents

1. Introduction

> "UNIX is user-friendly, it just chooses its friends."
>
> *(Andreas Bogk)*

An introduction into the field of High Performance Computing and the aspects of "Green IT" are given in this chapter. There is also a description about the goals and the structure of this thesis.

On today's computer systems, the CPU (central processing unit or simply processor) is the component which generally consumes the most energy, besides the GPU (graphics processing unit). The range lies between several watts in idle with activated power management and up to 110 W under load for a CPU [Obo10], and between 20 W and 320 W for a GPU [GS10]. This is reflecting in the thermal design power (TDP) representing the maximum amount of heat the cooling system must be able to dissipate.

This is also reflecting in the amount of power the computer system consumes and therefore the costs it causes. Especially when thousands of nodes with such CPUs respectively GPUs are active in a data center or HPC cluster. HPC stands for *high performance computing* and, as the name suggests, its goals are offering high performance computing systems.

Additionally to CPUs a cluster needs a potent *communication network* with high bandwidths, low latencies and adaptive routing, and storage nodes for backups and user files. Such systems are needed in research and industry, mostly for simulations. Some popular usage examples are

- crash simulations
- climate research
- protein folding
- animation / special effects

Parallelization is crucial to get high performance by an adequate power consumption. Assuming the implementation of the application is not the bottleneck (the limiting factor), and there is the possibility to run the program in parallel, the only two ways for faster processing—except upgrading to newer processor types (or switching to another platform)—are either a higher processor frequency or more processors.

To compare frequency scaling with processing unit (PU) duplication, two more assumptions are needed:

1. processing speed scales linear with the frequency f
2. $P \propto U^2 \cdot f$ and $U \propto f$

Now a small calculation shows what happens to the power consumption P by rising the frequency and voltage U, or adding more PUs:

frequency scaling P_1 is the consumption at normal speed. Doubling f means $P_2 = (a \cdot U)^2 \cdot 2f = 2 \cdot a^2 \cdot P_1$, whereas $a > 1$.

duplicate processing unit By doubling the processing unit, processing speed doubles too, at double power consumption: $P_2 = 2 \cdot P_1$

These are rough assumptions, but are legit for multi-core processors. But it is contra productive to produce processors with many cores at low frequency, because not all programs are (good) multi-threaded and would suffer from poor single-thread performance. Another reason is simply the limited space on a processor die, as more cores will need more space, together with a better inter-core network to unleash their full potential. The right balance between the number of cores and the actual processing speed per core is important.

For multi-node systems (clusters) are many things not considered, for example the power consumption of the network and the other parts of a node (like mainboard, NIC, RAM or HDD). There is another reason against even higher frequency and therefore higher power consumption. The *power density* of the chip rises. There are limits how much heat a chip is able to bear before it dies. This is comparable to a glowing wire, before it melts. Modern processors have a power density of around 100 $\frac{W}{cm^2}$, which is nearly as much as a hot cooking plate.

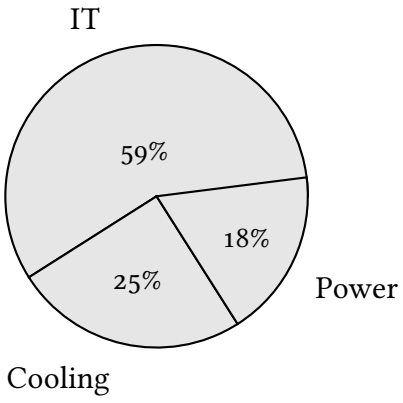


Figure 1.1.: typical energy consumption of a data center [Gro08]

In the end, the power is transformed into heat, which must be dissipated. Some fans and heat sinks on CPU and GPU are enough in a desktop computer. As place is limited in a building / room, cluster configurations are usually build of racks containing the dense packed nodes, offering the most hardware per amount of space. *Cooling* is very important to guarantee the right environment for healthy equipment. As figure 1.1 shows, it is not unusual if cooling consumes more than 25 % of the power of the cluster, and another 15-25 % are

consumed by the power supply (including uninterruptible power supply, general power conversions and efficiency of the power supply unit).

Popular ways of writing parallel programs are *threads* for multi-core processors per single node and *message passing* or *virtual shared memory* for multiple nodes. Parallel programs have new ways of getting things wrong. Every line of code in a single-threaded program executes in order—besides compiler optimization or out-of-order-execution—whereas parallel programmes are hard to get in specific order. Access on shared variables or collective communication are indeterministic. Such *race conditions* can even lead to a total halt, a *deadlock*.

But the hard task is to get the result right and benefit from the multiple PUs—the application has to scale. *Scalability* means the more cores calculate the faster the solution has to show up or the more data can be processed in the same amount of time.

The resulting performance boost by adding N more processing units is called *speedup* S. It is ideally measured with the time T_N using N PUs against the time T_1 of a single-threaded program using the perfect algorithm for this problem—not against the parallel program running with one PU! How well the speedup is in regard to the used processing units is measured by the efficiency E [Lud08].

$$S = \frac{T_1}{T_N} < N \qquad\qquad E = \frac{S}{N} < 1$$

There are two established laws concerning the "prediction" of scalability, whereas P is the part of the program that is actually affected by the parallelization:

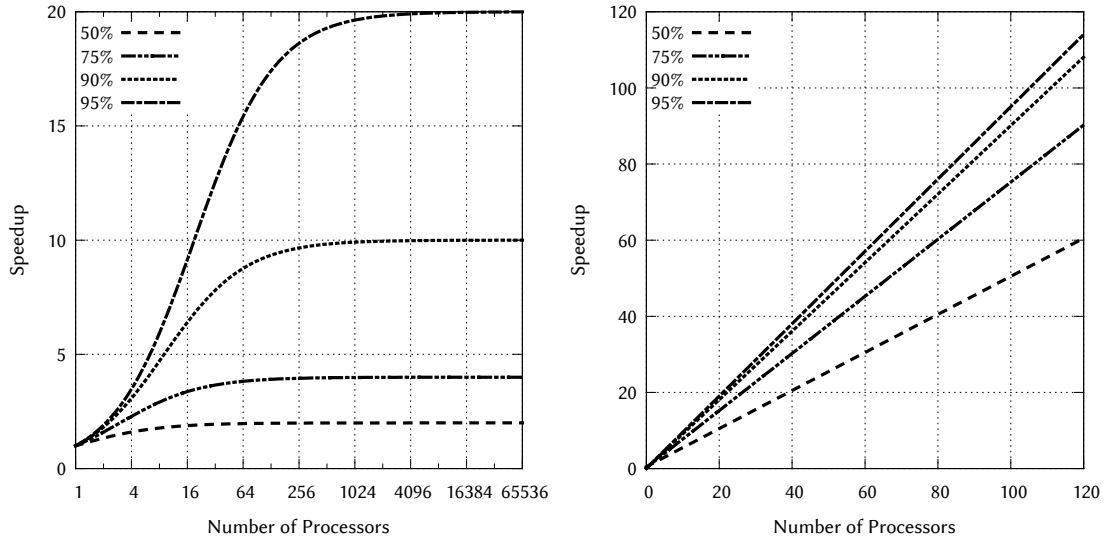
Amdahl's Law $S = \frac{1}{(1-P)+\frac{P}{N}}$

Every program has parts that can not be split onto several PUs or that has to be done by every single PU, for example initialization or communication. This overhead is the reason why E is really smaller than 1. The actual speedup now consists of the overhead $(1 - P)$ and the parts getting faster processed $\frac{P}{N}$.

Gustafson takes another approach, because the part of a program causing the overhead is seldom executed often, mostly at the beginning and the end. The main purpose costing the most time of the program—the calculation—will be parallel, so P is actually 1 while calculating.

Gustafson's Law $S = N - (1 - P) \cdot (N - 1)$

As figure 1.2 implicates, Amdahl's law predicts always a possible upper limit, whereas Gustafson's law does not. The reason is the difference opinion about the used *problem size*. Problem size actually means the amount of data that has to be processed. Amdahl implicates a fixed problem size, so the calculating / communication ratio for every PU decreases when more PUs are used.



(a) Amdahl's law (b) Gustafson's law

Figure 1.2.: Amdahl's law and Gustafson's law

Gustafson predicts that the users will usually process more data (greater problem size) when they have more machines, mainly for getting more accurate results in the same amount of time.

Both laws are valid, as they target slightly different fields under different assumptions.

* * *

The TOP500 project [TP11] aims to deliver statistics about the 500 most powerful general purpose HPC clusters in the world, and the technical development of those, because it helps manufacturers and (potential) users exchanging software and data, as well as establishing collaborations. The list is updated half-yearly (november and june) since 1993 and reflects the actual evolution in computing, as the aggregated performance of the list shows characteristics of Moore's law predicting a growth in performance by factor two every 18 months. Specific data about the computers are among others the number of cores, the power consumption, the theoretical performance (R_{peak}) and measured performance (R_{max}) in TFlops (10^{12} floating point operations per second, double precision). The measurement benchmark is LINPACK, which is basically a distributed solver for dense linear equation systems allowing architecture optimizations and variable problem sizes for achieving the best possible score per cluster.

Table 1.1 shows four systems of the TOP500 list from november 2010. The first three have each more than one PFlops of measured performance, but also a power consumption of more than 2.5 MW. Jaguar drains nearly 7 MW! For comparison, an average personal

Table 1.1.: Abstract TOP500 List, November 2010

Rank	Computer	Cores	R_{max} [TFlops]	R_{peak} [TFlops]	Power [kW]
1	Tianhe-1A	186 368	2566	4701	4040
2	Jaguar	224 162	1759	2331	6951
3	Nebulae	120 640	1271	2984	2580
9	Jugene	294 912	826	1003	2268

computer without monitor needs between 50 and 100 W (at ≈30 GFlops R_{max}), a laptop with screen needs approximately 20 W.

An example breakdown of an actual cluster node can be seen in figure 1.3. Energy saving mechanics were not available—like in almost all clusters, because the main purpose of HPC is high performance, whereas general energy saving mechanics can lead to performance degradation. This picture should be applicable for many other cluster nodes and desktop machines—perhaps with lower CPU idle power consumption and one (or even more) additional GPGPU-cards (general purpose GPU).

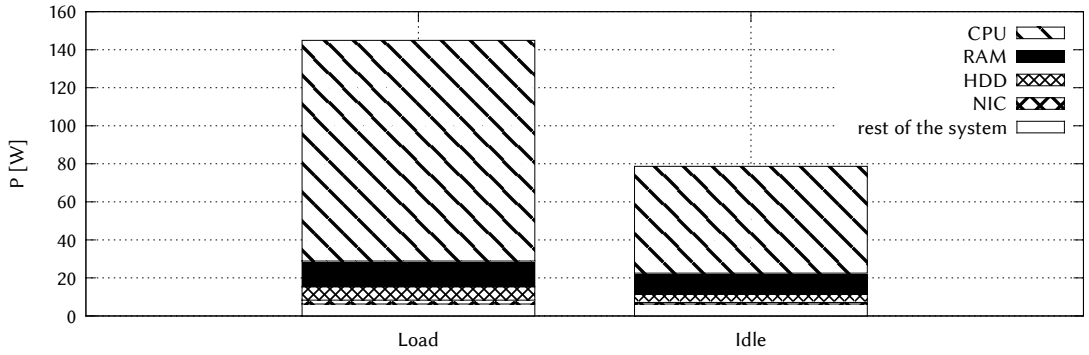


Figure 1.3.: System power consumption [Min09]

1.1. Green IT

Like the name high performance computing implies, the goal in this area is performance. But energy efficiency gets more and more popular, as everyone seems to talk about CO_2 and the rising oil and energy prices. The Green500 project [GP11c] takes the TOP500 list and reorders it considering the efficiency $\frac{MFlops}{W}$ based on R_{max} and the power consumption. It is obvious by referring to table 1.2, that high performance does not automatically imply energy efficiency, as the first ranked in the Green500 list delivers

more than twice the energy efficiency than the first ranked in the TOP500, and more than six times than the second.

Table 1.2.: Abstract Green500 List, November 2010

Rank	Computer	Efficiency [MFlops/W]	Power [kW]	TOP500 Rank
1	NNSA/SC Blue Gene/Q Prototype	1684.20	39	115
11	Tianhe-1A	635.15	4040	1
14	Nebulae	492.64	2580	3
88	Jaguar	253.07	6951	2

Application developers have to think about energy efficiency too, because bad program behaviour (e. g. polling) has impact on sleep time and the needed performance (e. g. algorithms) to finish a task.

But green IT means more than just energy efficiency. It is commercially relevant, as the consumer has direct advantages and a better feeling about "helping the environment". But even more important are the questions concerning the use of toxic materials or recyclability / reusability / refurbishability. Shamefully those topics are not well represented in mass-media, and many consumers do not even know where to dump their electronic waste. Those things are mainly handled and controlled by laws. A well-known example is "RoHS", which can be read on many stickers at electronic articles. RoHS stands for restriction of hazardous substances and describes limits of using toxic materials in products. Another example is the "energy star standard" which gives specifications about how much different devices (for example PCs or gaming consoles) are allowed to consume in active-, idle-, sleep- and standby-mode. [Muro8]

1.2. Goals and Structure of this Thesis

HPC clusters need a large amount of power. Measuring this consumption for the whole cluster is very problematic, especially component measurement is only doable for few nodes, because special equipment is needed to record the consumption of a CPU or GPU. Such measurements are useful to find ways of saving energy or to predict the energy consumption after replacing components. In order to estimate at least the consumption of one of the main consumers—the CPU—a model based on its frequency and voltages will be presented and the needed data sources implemented.

* * *

The rest of this thesis is structured as follows: chapter 2 deals with the calculation of the power consumption of a processor and how this can be approximated. Also some

general strategies to reduce this consumption are listed. Chapter 3 lists some of the interfaces given by a modern Linux needed to get data like frequency and sleep time from a processor. Some details about the projects that will be extended and used, as well as the actual implementation of the new features are described in chapter 4. Last but not least, chapter 5 evaluates the processor model based on real world examples, and shows the functionality of the implementations.

2. Modelling the Processor Power Consumption

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."
>
> *(Gordon E. Moore)*

A formula describing the power consumption of a processor using its voltage and frequency, as well as different approximations about how to calculate this consumption based on reference values will be formulated and compared in this chapter.

As seen in the previous chapter, the cluster energy consumption depends heavily on the used CPUs. The power consumption of a CPU (P_{total}) consists of a dynamic (P_{dyn}), a leakage (P_{leak}) and a short circuit power consumption (P_{short}). Additionally, recent processors include more and more components being formerly part of the mainboard. Those components are separated from the cores and are called "uncore". The main reasons for this are latency reduction (integration of the memory-controller), power efficiency (separate power control unit) and better scalability with many cores. Depending on processor type, uncore includes e. g. the memory-controller, L3 cache or PCIE-links. Of course the power consumption depends on the number of cores, too.

$$P_{total} = \#cores \cdot (P_{dyn} + P_{leak} + P_{short}) + uncore$$

The uncore part of the chip drains between 4 W and 20 W. There is by now no generic formula to calculate the power consumption of the uncore-part, and measurement can only be done with special equipment [Obo10]. For these reasons, uncore is considered to be constant and added to the mainboard power consumption.

Back to the main components of P_{total}:

P_{short} is the power caused by a short-circuit current which flows from the supply to ground during a transition period of input signals when transistors are momentarily on at the same time. Short-circuit current increases with slower signal transition times [HOT96].

"The short circuit power consumption occurs only during signal transitions and is negligible" [MFMB02].

P_{leak} is the gradual loss of energy from charged capacitors or when current leaks out of the intended circuit. P_{leak} is modelled differently depending on the actual research and application [MFMB02, Mud00, LHL05]. It is possible that leakage power drains up to 20 % of P_{total}. So, further model improvements could include P_{leak}, because of its temperature dependency: $P_{leak} \propto T^2 \cdot V$ [BAEP08].

P_{dyn} is the power used to actually charge and discharge the capacitance, composed of gate and interconnect capacitance. The greater this dynamic switching current is, the faster capacitive loads can be charged and discharged, enabling a better performing circuit.

Nowadays P_{dyn} dominates [Mud00] and modelling the processor power consumption can be done with

- U, the supply voltage (often referred as $CPU\ Vcore$ or V_{DD}) [V],
- f, the frequency the processor runs at [Hz]

and

- α, an activity factor (not all gates are always switched),
- C, the capacitance at the gate outputs [F].

So actually, P_{short} and P_{leak} will be ignored and only P_{dyn} considered,

$$O(P_{dyn}) \in O(P_{dyn} + P_{leak} + P_{short}).$$

Which leads to an accurate estimation of the power consumption a processor core is using [Min09]:

$$\begin{aligned} P_{core} &\approx P_{dyn} \\ &= U^2 \cdot f \cdot \alpha \cdot C \end{aligned} \tag{2.1}$$

U is the dominating factor, which means power consumption reduces linear by decreasing the frequency, whereas decreasing the voltage reduces the power consumption quadratically.

By writing about "processor power consumption" in this thesis, it is always meant per core and without uncore.

2.1. Voltage^2-frequency

The factors α and C from the previously introduced power consumption formula (2.1) are depending on many different influences like processor type, workload or temperature. To avoid measuring / guessing C and α, they are considered constant at equal workloads on a given processor:

$$\begin{aligned} const &= \alpha \cdot C \\ \Rightarrow P &= U^2 \cdot f \cdot const \end{aligned}$$

Having measured two (or more) power consumptions P_1 and P_2 at different voltages (U_1, U_2) respectively frequencies (f_1, f_2) of a processor results in an easy transformation equation:

$$\begin{aligned} const &= \frac{P}{U^2 \cdot f} \\ \Rightarrow P_2 &= P_1 \cdot \frac{U_2^2}{U_1^2} \cdot \frac{f_2}{f_1} \end{aligned} \tag{2.2}$$

The estimated power consumption of a processor P_2 can be calculated by having a reference value triple P_1, U_1 and f_1, and the new voltage U_2 and frequency f_2, without having to deal with the actual activity α respectively capacitance C.

2.2. Approximations

Sometimes there is no possibility to measure the voltage. Either the mainboard has no interfaces for software, only `root` can use them or there is no adequate software installed / available.

In the following subsections are approximations described dealing without or only partial / general information about the supply voltages.

2.2.1. Frequency^3 Approach

If no access to any voltages is available, a simplifying approach is to assume, that changing the voltage also requires to change the frequency at the same amount. Generally

speaking $f_{max} \propto U$. This leads to $U = \beta \cdot f$, where β is another constant [EKR03].

$$\begin{aligned} P &= U^2 \cdot f \cdot const \\ U &= \beta \cdot f \end{aligned} \tag{2.3}$$

$$\begin{aligned} \Rightarrow P &= f^3 \cdot const \\ \Rightarrow P_2 &= P_1 \cdot \frac{f_2^3}{f_1^3} \end{aligned} \tag{2.4}$$

In contrast to equation (2.2), the power consumption consists only of the former and the recent frequencies (f_1 and f_2).

2.2.2. Extending Approximation

The connection between the voltage, the frequency and the actual power consumption is unfortunately a bit more complicated, especially at lower voltages. From [Mud00] it is known, that

$$f_{max} \propto \frac{(U_{supply} - U_{threshold})^2}{U_{supply}},$$

where U_{supply} is the supply voltage and $U_{threshold}$ the threshold voltage. The threshold voltage at the gate is needed to prevent the creation of current from the drain to the source of the transistor. If U_{supply} is bigger than $U_{threshold}$ the transistor is on, and vice versa.

In order to lower U_{supply}, there is a constraint to reduce $U_{threshold}$ too. But a reduction of $U_{threshold}$ means, that P_{leak} (and therefore P_{core}) rises, because of $P_{leak} = I_{leak} \cdot U_{supply}$ and the proportionality $I_{leak} \propto \exp(-U_{threshold} \cdot \frac{q}{kT})$ [Lun06].

While this knowledge does not help to improve the approximation directly, it does help focusing on the problems.

In picture 2.1 are some valid f / U-pairs from more or less recent CPUs and a linear regression line for each. It seems that U and f actually are in a linear relationship. But they are not proportional towards each other. The equation should rather be $U = \beta \cdot f + \gamma$, whereas β is the gradient and γ the y-intercept. Replacing U results in

$$\begin{aligned} P &= U^2 \cdot f \cdot const \\ U &= \beta \cdot f + \gamma \end{aligned} \tag{2.5}$$

$$\begin{aligned} \Rightarrow P &= (\beta \cdot f + \gamma)^2 \cdot f \cdot const \\ \Rightarrow P_2 &= P_1 \cdot \frac{f_2}{f_1} \cdot \frac{(\beta \cdot f_2 + \gamma)^2}{(\beta \cdot f_1 + \gamma)^2} \end{aligned} \tag{2.6}$$

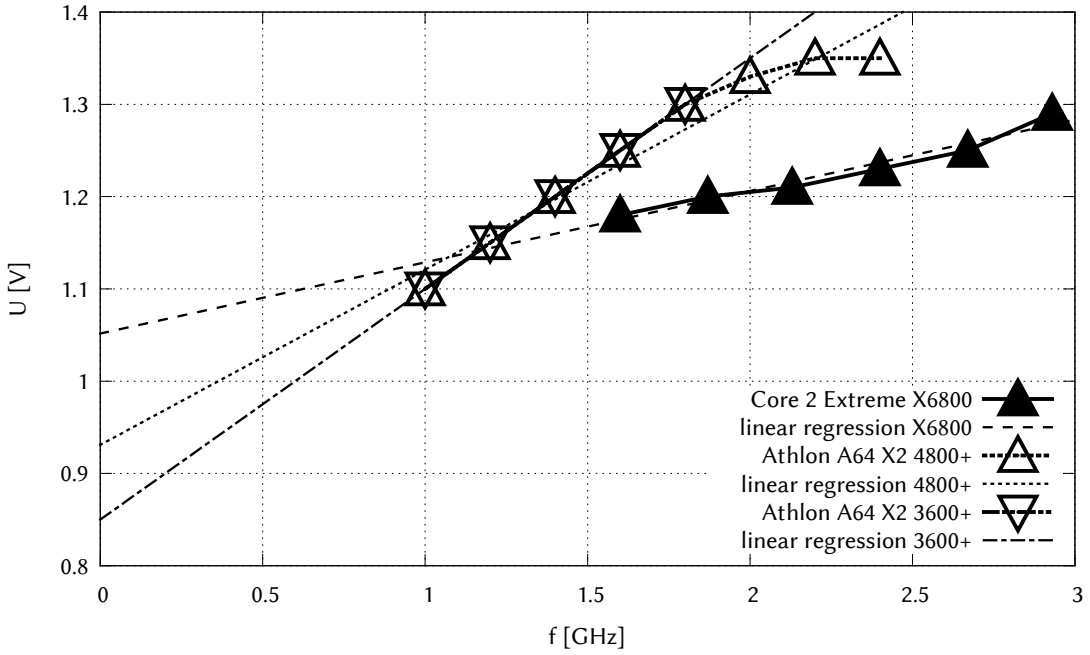


Figure 2.1.: P-state trend lines

It can be seen in figure 2.1 that β and γ are different for every processor type. In order to calculate them, there has to be at least two valid f / U-pairs. Equation (2.5) results directly in

$$\Rightarrow \beta = \frac{\Delta U}{\Delta f} \qquad \text{and} \tag{2.7}$$

$$\Rightarrow \gamma = U_1 - f_1 \cdot \frac{\Delta U}{\Delta f} \tag{2.8}$$

By having introduced two new constants β and γ in equation (2.2.1), there is the option to eliminate one of them.

Maybe it is easier to think about a y-intercept [U] than a gradient [$\frac{U}{f}$], but the actual choice depends, as always, on the use case.

2.2.3. Intercept γ Approach

Disappearing of β starts again from equation (2.5):

$$\begin{aligned} \frac{U-\gamma}{f} &= \beta \\ \frac{U_1-\gamma}{f_1} &= \frac{U_2-\gamma}{f_2} \\ \Rightarrow U_2 &= U_1 \cdot \frac{f_2}{f_1} + \gamma \cdot (1 - \frac{f_2}{f_1}) \end{aligned} \tag{2.9}$$

Inserting U_2 in the main equation (2.2) results in

$$\begin{aligned} P_2 &= P_1 \cdot \frac{(U_1 \cdot \frac{f_2}{f_1} + \gamma \cdot (1 - \frac{f_2}{f_1}))^2}{U_1^2} \cdot \frac{f_2}{f_1} \\ &= P_1 \cdot \frac{(U_1 + \gamma \cdot (\frac{f_1}{f_2} - 1))^2}{U_1^2} \cdot \frac{f_2^3}{f_1^3} \end{aligned} \tag{2.10}$$

In contrast to equation (2.4) there is now an additional factor involved, which is greater 1 when $f_1 > f_2$ and lesser 1 when $f_1 < f_2$—in contrast to approach 2.2.1 more power drainage by switching to lower speeds, less by higher speeds. It becomes 1 by equality of f_1 and f_2 resulting in $P_1 = P_2$.

2.2.4. Gradient β Approach

Analogue to subsection 2.2.3, γ will vanish and the result becomes part of the main equation (2.2).

$$\begin{aligned} U - \beta \cdot f &= \gamma \\ U_1 - \beta \cdot f_1 &= U_2 - \beta \cdot f_2 \\ \Rightarrow U_2 &= U_1 - \beta \cdot (f_1 - f_2) \end{aligned} \tag{2.11}$$

$$P_2 = P_1 \cdot \frac{(U_1 - \beta \cdot (f_1 - f_2))^2}{U_1^2} \cdot \frac{f_2}{f_1} \tag{2.12}$$

In order to stay consistent with the equations of the other approximations, altering above equation for sake of simplicity produces:

$$P_2 = P_1 \cdot \frac{(U_1 \cdot \frac{f_1}{f_2} - \beta \cdot (\frac{f_1^2}{f_2} - f_1))^2}{U_1^2} \cdot \frac{f_2^3}{f_1^3}$$

but equation (2.11) should rather be used for real-world application.

* * *

As seen, the two extended approximations from section 2.2.2 both need U_1 and the gradient β or the intercept γ, in addition to f_1 and f_2. Nevertheless, there is no reason to use the first approximation from section 2.2.1. U_1 can be seen in the BIOS (basic input output system) while booting, or asking the admin / support for its value. For β or γ some average values could be used. This should deliver acceptable results, as a short comparison shows.

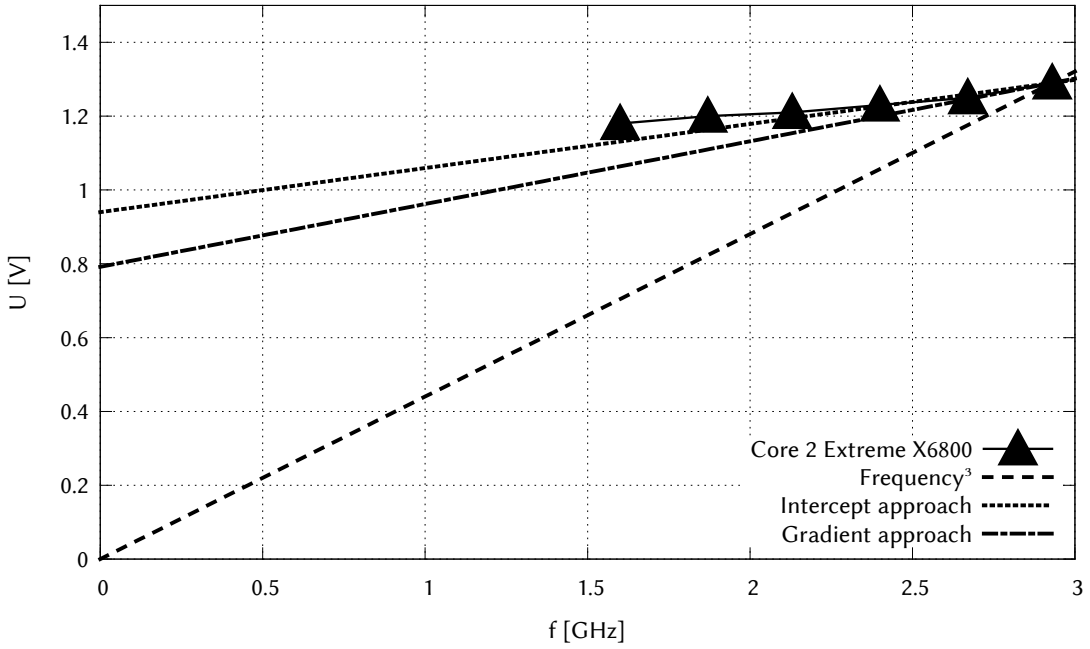


Figure 2.2.: Comparison between the approximations ($\beta = 0.17$, $\gamma = 0.94$)

Figure 2.2 shows a comparison between the three approximations discussed above at an example processor (taken from table A.1), mean values for $\beta = 0.17$ and $\gamma = 0.94$ and the following derived formulas:

$$\text{Frequency}^3 \text{ approach (2.3)} \Rightarrow U_2 = U_1 \cdot \frac{f_2}{f_1}$$

$$\text{Intercept approach (2.9)} \Rightarrow U_2 = U_1 \cdot \frac{f_2}{f_1} + \gamma \cdot (1 - \frac{f_2}{f_1})$$

$$\text{Gradient approach (2.11)} \Rightarrow U_2 = U_1 - \beta \cdot (f_1 - f_2)$$

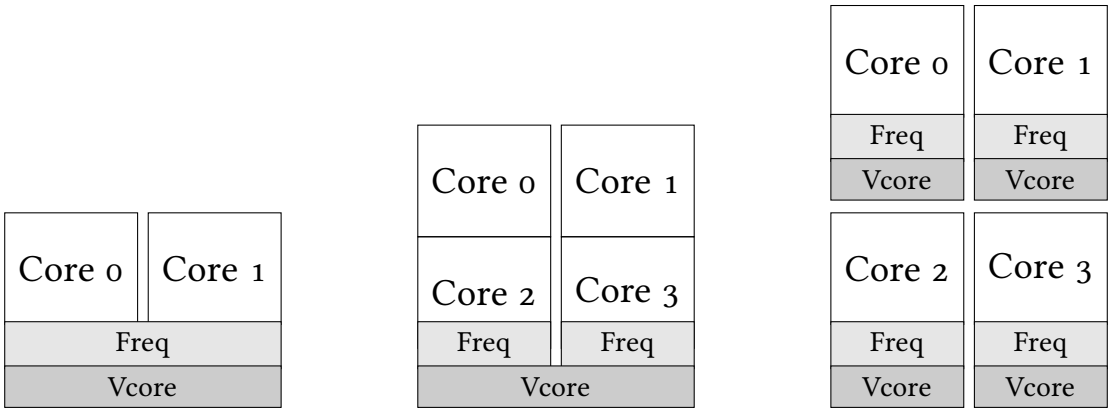
It is obvious that the frequency³ approach is not able to compete against the other two. The assumption $f \propto V$ can not be considered at lower frequencies, resulting in too low estimated power consumptions.

2.3. Power Saving Techniques

As pointed out in this chapter, processors need much energy when calculating, and there is a great potential of reducing the power consumption of them. Power saving is implemented in hardware and in software, as those two have to work closely together in order to save the most energy without performance degradation.

Dynamic Voltage and Frequency Scaling is the direct consequence from this chapter and possibly the most known technique. Based on its utilization the processor decreases its frequency and voltage in order to decrease power consumption too—the processor should only be as fast as needed, and therefore use as low power as possibly.

Mobile devices like notebooks or smart-phones use DVFS already a long time, since they notably rely on low energy consumption. Nowadays, DVFS is widespread among desktop PCs and server, and latterly some HPC cluster are using it too.



(a) DualCore shared Freq & Vcore

(b) QuadCore half-split Freq, shared Vcore

(c) QuadCore split Freq & Vcore

Figure 2.3.: DVFS in multicore-processors

But DVFS has some disadvantages.

- It only reduces P_{dyn}, but P_{leak} will grow strongly in the future [LHL05] due to further miniaturization.
- As shown in figure 2.3, voltage and frequency can not always be free chosen. This is heavily processor and architecture dependent.

- It takes time (and energy) to switch to another frequency- / voltage-level. Wrong decisions in switching can either harm performance respectively power saving, and there is still no "crystal ball" available.

"Sleeping" is used when a core has no work to do. This is done by deactivating parts of the core (ALUs, caches) or the complete core itself via clock- / power-gating, meaning to lower or even cut the frequency or the voltage. There exist different sleep states; the "deeper" the sleep the less power is consumed, but the longer it takes to wake up. A core can not execute instructions while sleeping.

The disadvantages are mainly the same as for DVFS [DG10].

Idle-aware Scheduling describes the idea of scheduling the processes in such a way, that many cores can sleep deeply. This has to be done by the OS in order to gain the most of DVFS and sleeping.

This can of course mean some performance loss as processes have to use the same resources on a core. The opposite idea is to spread the processes on many cores for more performance and faster responses, but higher power consumption [PS07].

Intel Turbo Boost / AMD Turbo Core Both Intel and AMD have implemented automatic overclocking features into their newer CPUs increasing their maximum frequency when the thermal budget is not yet reached. This is mainly the case, when single-threaded applications only stress individual cores. In this case, the other cores can sleep—their thermal budget is unused. It is controversial calling dynamic overclocking a power saving technique, because the power consumption increases disproportionate to the performance gain.

Nevertheless it could help reducing the power consumption, because the faster some work is done, the longer the cores can sleep. The OS is also able to schedule processes longer on less cores when they ran faster, affecting again the sleep time of the remaining cores.

Turbo Boost / Turbo Core is all done hardware-side and transparent to the OS. By now, the only possibility for the OS / user to check if Turbo Boost / Turbo Core is in use is counting the CPU-cycles [Int08].

3. Interfaces

> "One thing I find myself wondering about is whether we shouldn't try and make the 'ACPI' extensions somehow Windows specific. [...] Or maybe we could patent something related to this."
>
> *(Bill Gates)*

In this chapter, there will be a basic introduction to the different used interfaces in order to trace the processor states, beginning with ACPI, following cpufreq and cpuidle.

Modern computer systems have plenty of ways to interact with the hardware. Of course they need to, in order to provide a fully functional and working system. The main communication between the OS and the hardware (or their firmware) is hidden for the normal computer user. But application programmer can access hardware, either direct or with one of the many generic kernel subsystems hiding hardware details and providing the main hardware functions.

3.1. Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) is a specification describing hard- and software interfaces for the implementation of OS based power management (OSPM), where the OS controls the "power saving features" of the devices. This is necessary considering that only the OS has all the information to efficiently send devices to sleep without harming the performance. ACPI is the successor of APM (Advanced Power Management), where only the devices themselves could control their energy saving features.

ACPI even defines its own source and machine languages: the "ACPI Source Language" (ASL) and the "ACPI Machine Language" (AML).

ACPI defines different levels of operation (operating states), composed of a letter and a number. The letter defines the region of operation and the number is an indication of the saved energy; the higher the number the more power will be saved, but latency returning to a "lower number" increases [Int10].

Here is a short overview of the system states and what they mean:

G-states Global system state definitions

G0 Working

G1 = S-states Sleeping
The computer appears off, but power consumption is not zero. OS context is saved and work can be resumed without rebooting.

The common sleep-states are known under suspend-to-disk ("hibernation") and suspend-to-memory ("standby"), where the OS context is saved to disk or RAM.

G2 = S5 Soft Off / Standby
Power consumption is minimal. No OS context is saved—system must be restarted to return to working state.

G3 Mechanical Off
Power consumption is zero by means of a mechanical switch. The system can be disassembled safely.

Legacy is only used while booting or when the OS is not capable of ACPI. The device then uses its own power policy, similar to APM.

D-states Device power state definitions
device states can be applied to nearly any device on any bus. They range from D0 (Fully-On) to D3 (Off). The meaning of the states in-between is defined by each device class, and may not be defined at all. They can be thought of S-states for devices.

P-states Device and processor performance state definitions
Today, P-states in processors are generally the specification of DVFS. Table 3.1 shows examples for P-states.

P0 Delivers the most performance, but consumes also the most power.

Pn Processors and devices can define up to 16 P-states, all with the same idea of limiting performance in order to save power.

C-states Processor power state definitions
Practically the S-states for the processor

C0 Processor is executing instructions—it is not sleeping.

Cn different processors may implement different C-states. They can also differ in the implementation of the same state. All together is the general characteristic that a higher number implies lower power consumption but a longer wakeup-time. Not all processors report all their C-states to the OS.

Figure 3.1 shows an overview of these states and how they interact which each other.

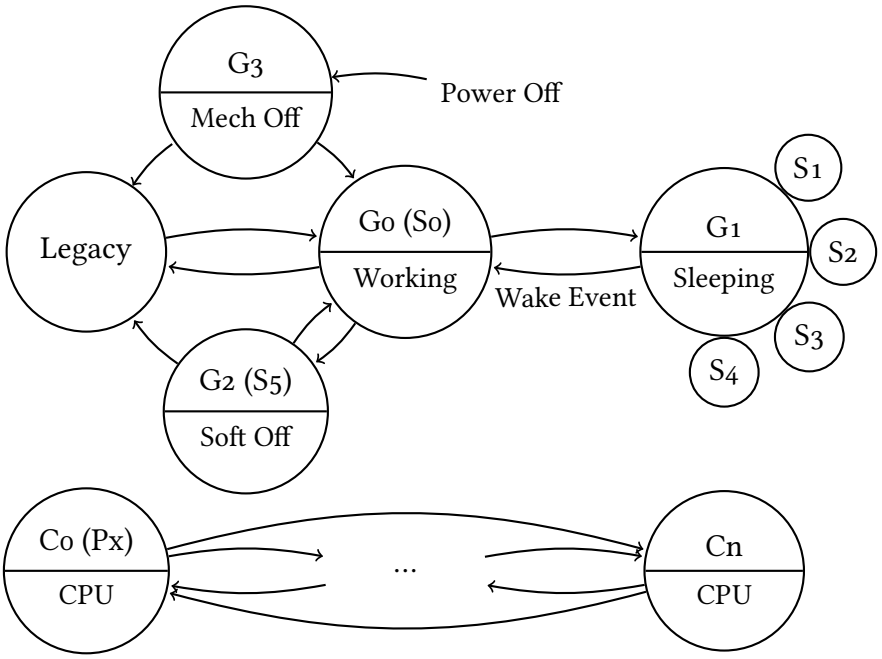


Figure 3.1.: ACPI state transition overview [Int10]

3.2. CPUFreq

CPUFreq is the generic kernel subsystem for managing processor P-states and was merged in kernel 2.6.9. It aims to provide a clean interface, where architecture-specific details (driver) are isolated from the generic power management policies (governor). [Bro11, PS06]

3.2.1. Driver

Drivers are the interface to the processor. They must implement specific functions, like listing the available target frequencies or the frequency-change-mechanism. Providing information about the processor, e. g. the transition latencies or the minimum / maximum frequencies, is a duty of the driver, too. These informations are either hard-coded and general, or retrieved through the MSR of the processor. "Machine Specific Registers" or "Model Specific Registers" are a direct interface to the CPU for configuration and getting information. They can only be accessed with root-privileges and are, as the name suggests, processor dependent. MSR can be used with the assembly instructions `RDMSR` (reads MSR) or `WRMSR` (writes MSR).

Table 3.1.: P-state examples [cha07]

	Core 2 Extreme X6800 [GHz / V]	Athlon A64 X2 4800+ [GHz / V]	Athlon A64 X2 3600+ [GHz / V]
P0	2.93 / 1.2875	2.4 / 1.350	1.8 / 1.30
P1	2.67 / 1.2500	2.2 / 1.350	1.6 / 1.25
P2	2.40 / 1.2250	2.0 / 1.325	1.4 / 1.20
P3	2.13 / 1.2125	1.8 / 1.300	1.2 / 1.15
P4	1.87 / 1.2000	1.6 / 1.250	1.0 / 1.10
P5	1.60 / 1.1750	1.4 / 1.200	–
P6	–	1.2 / 1.150	–
P7	–	1.0 / 1.100	–

As driver are very processor dependent, they also have to check weather they can work with the given CPU or not.

3.2.2. Governor

Governor are fully driver-independent and the power management policies that decide when / if to change the processor frequency.

The following governor are merged into the kernel and deliver quite a variety of policies to chose from.

Performance

The “performance”-governor sets the CPU statically to the highest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.

Powersave

The “powersave”-governor sets the CPU statically to the lowest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.

* * *

The “performance”- and “powersave”-governors are not changing the frequency. They can be useful in some situations—hot summers or phases where latency is crucial—but they essentially disable DVFS by locking the frequency.

Userspace

The “userspace”-governor allows the user, or any userspace program running with UID `root`, to set the CPU to a specific frequency by making a sysfs file `scaling_setspeed` available.

Ondemand

The “ondemand”-governor aggressively sets the CPU depending on the current usage. It mainly switches to the highest frequency in order to complete the task and then switches instantly to the lowest. This tactic is also called “race-to-idle”. To do this the CPU must have the capability to switch the frequency very quickly.

Conservative

The “conservative”-governor, much like the “ondemand”-governor, sets the CPU depending on the current usage. It differs in behaviour in that it gracefully increases and decreases the CPU speed rather than jumping to the maximum speed the moment there is any load on the CPU. This behaviour is more suitable in a battery powered environment.

* * *

The “ondemand”- and “conservative”-governors have some preferences controlling their behaviour. These so called tuneables can be found in the CPUFreq-sysfs interface found in the appendix A.

3.2.3. Programming interface

CPUFreq also offers a C / C++ programming interface by including `<cpufreq.h>` and linking against the `libcpufreq`-library.

The following listing shows an abstract from the interface-specification used later in the implementation.

Listing 3.1: libcpufreq abstract

```
68 /* determine current CPU frequency
69  * - _kernel variant means kernel's opinion of CPU frequency
70  * - _hardware variant means actual hardware CPU frequency,
71  *    which is only available to root.
72  */
73
74 extern unsigned long cpufreq_get_freq_kernel(unsigned int cpu);
75 extern unsigned long cpufreq_get_freq_hardware(unsigned int cpu);
```

3.3. CPUIdle

Like CPUFreq "controls" the P-states, CPUIdle is the generic kernel subsystem managing the C-states reported by ACPI—some BIOS hide deeper C-states. The maximum number of C-states the Linux kernel can handle can be found under `/proc/acpi/processor/*/power`, for example `max_cstate:C8`. This value represents not the number of C-states implemented in the used hardware [Int11]. CPUIdle was merged in kernel 2.6.24. Exactly like CPUFreq it separates architecture-specific details (driver) from the architecture-independent implementation (governor). [cpu11, PLB07, Cor10, Bro10]

The following governor are merged into the kernel.

ladder

The "ladder"-governor works fine with periodic tick-based kernels. It checks every tick if it can go in a deeper idle state or not. This step-wise model works not very well with tickless kernels, because without a periodic timer tick it may not get a chance to use a deeper idle state whenever it goes idle.

menu

The "menu"-governor on the other hand can handle this. It looks at different parameters like what the expected sleep time is (as provided by the tickless kernel), latency requirements, previous C-state residency and `max_cstate` requirement and then picks the deepest possible idle state straight away.

3.3.1. sysfs

Global information can be found under

```
$ ls -l /sys/devices/system/cpu/cpuidle
```

↳ `current_driver`
name of the current driver

↳ `current_governor_ro`
name of the current governor

Specific information per CPU X can be found under

```
$ ls -l /sys/devices/system/cpu/cpuX/cpuidle
```

↳ `state0`

↳ `state1`

↳ `state2`

↳ `state3`

```
$ ls -l /sys/devices/system/cpu/cpuX/cpuidle/state0
```

↳ `desc`
idle state description

↳ `latency`
latency time exiting this state, in microseconds

↳ `name`
idle state name

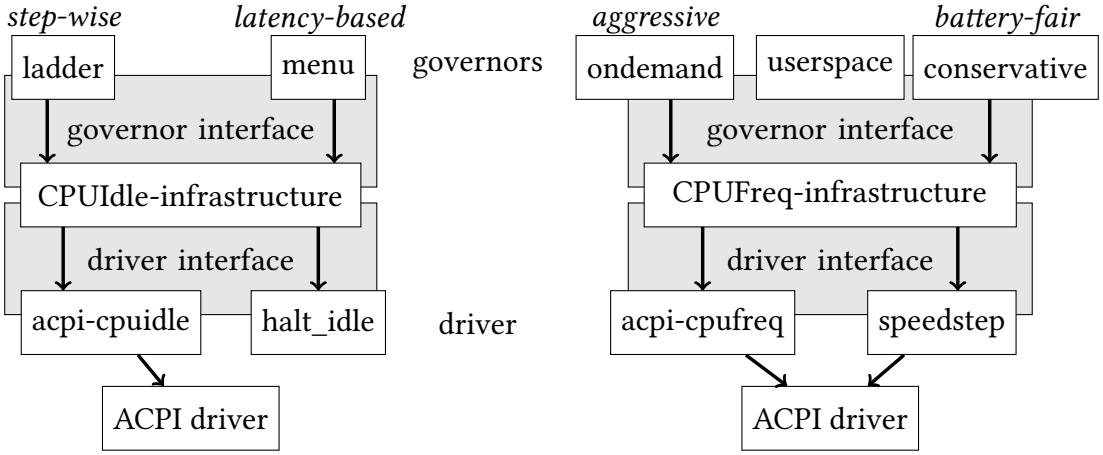
↳ `power`
power consumed in this state

↳ `time`
time spent in this state, in microseconds

↳ `usage`
number of times this state was entered

* * *

Figure 3.2 gives a simple overview over the CPUFreq- and CPUIdle-framework, pointing out the general similarity. Up to now, these two frameworks are working absolutely independent from each other [PS07].



(a) CPUIdle (b) CPUFreq

Figure 3.2.: Design Overview CPUIdle / CPUFreq [Bel10]

3.4. Intelligent Platform Management Interface

IPMI enables system administrators to manage computer systems and monitor their operation through a set of standardized computer system interfaces.

It is possible to communicate with a distant computer over a serial connection or a local network with IPMI. Error messages can be sent with SNMP (simple network management protocol). Direct BIOS access over LAN is also possible.

The interfaces are working independently from any OS and allow system administration without installed OS or even in standby-mode. More possibilities arise while the computer system is running, for example starting / stopping the computer, or disabling the on- / off- or reset-button.

The core of IPMI is the so called baseboard management controller (BMC). Other Controller for monitoring special hardware can be connected to it over the IPMB (intelligent platform management bus). This way monitoring fan speeds or the voltages of CPUs can be done mainly by hardware with low performance impact on the OS or other running software. The BMC is already integrated on recent server mainboards [Int09].

* * *

While IPMI focuses on remote maintenance and large data centers with special hardware, lm-sensors[1] concentrates on providing off-the-shelf hardware monitoring support for Linux. Lm-sensors is widely used in private servers and small test clusters made up of standard PC components.

The minimum update interval of lm-sensors is only 1.5 seconds, whereas IPMI is event-based and can deliver sensor data within 50 milliseconds. For that reason, and because most recent clusters are using IPMI, integration of lm-sensors has low priority.

[1] `http://www.lm-sensors.org/`

4. Implementation

> "The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user."
>
> *(Verity Stob)*

After seeing the needed interfaces in the last chapter, there will be a description about the overall software environment and which components had to be modified in order to implement the processor P- and C-state tracing.

The extended programs are parts of a whole software environment developed at the "Research Group Parallel and Distributed Systems" (PVS). The projects intend to create and visualize trace files. A trace file is the record of the actions a program does respectively how it interacts with its environment.

The *ResourcesUtilizationTracingLibrary* captures the component utilization from the OS, for example the used RAM or the transfer rates of the NIC, whereas the *PowerTracingLibrary* uses an external power meter to record the power consumption of the node. The *HDMPIwrapper* keeps track of MPI calls and utilizes the *Trace Writing Library* to actually write the traces to the hard drive. These projects are running while the observed MPI program runs, meaning they have some performance impact and are therefore implemented in the programming language C. The performance influence differs between 0.60 % (500 ms) and 10.34 % (20 ms) [Kre09] and is mainly based on the interval length the tracing library is waiting until it records a new trace point.

Additionally, there exist projects written in Java in order to evaluate the trace files. *HDTraceFormat* is used to read and modify trace files and *Sunshot* can visualize them. *HDPowerEstimation* can calculate the consumed energy and is able to estimate power savings based on different reschedule strategies. This shows the potential effect of activating power saving mechanisms in a cluster. HDPowerEstimation provides computed savings and pre-after-power curves.

ResourcesUtilzationLibrary will be able to trace P- and C-states, a power estimator—in the future merged into HDPowerEstimation—will be able to read these states and use them, exampled in a new strategy.

Figure 4.1 depicts the software environment and how the described components are interacting with each other.

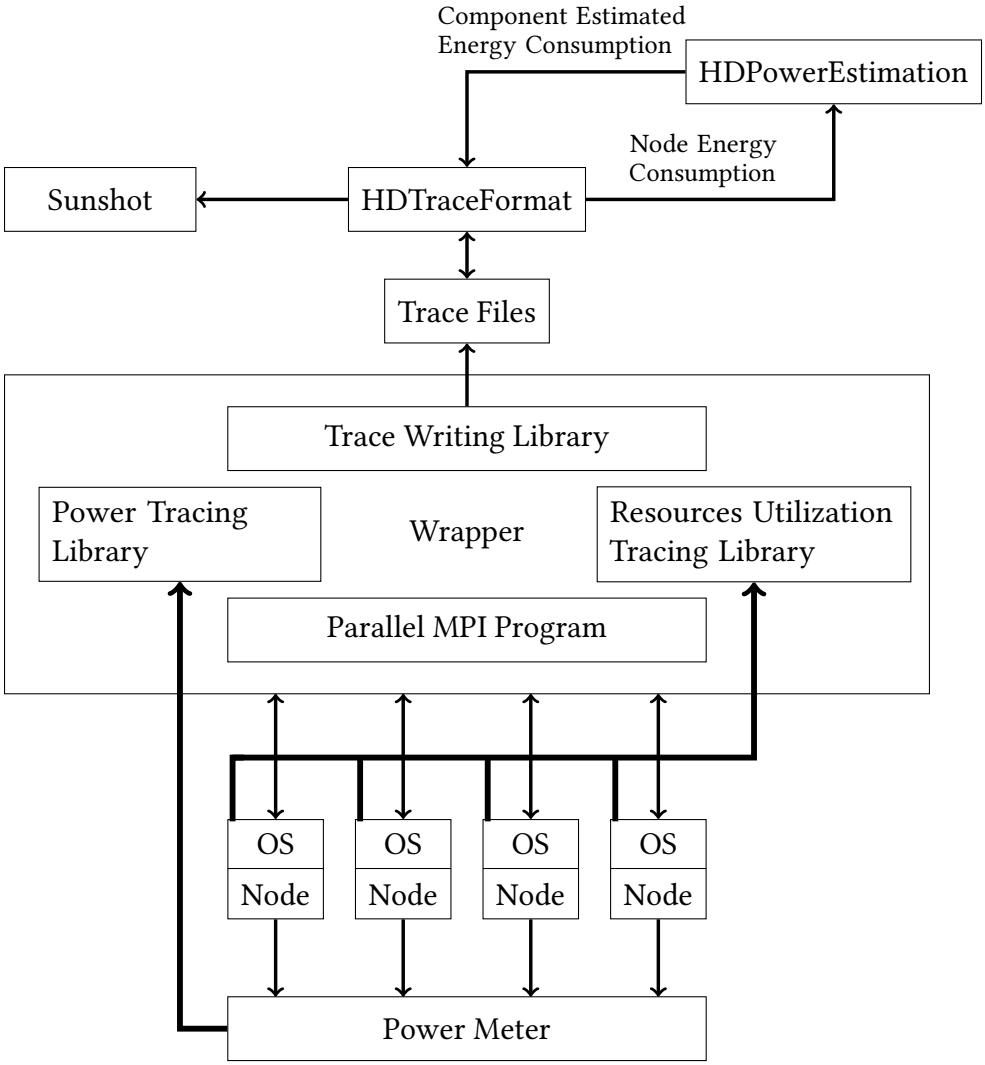


Figure 4.1.: Tracing environment at the PVS cluster [Min09]

4.1. ResourcesUtilizationTracingLibrary

The basic workflow of the library consists of an initialization phase and the tracing phase. In the initialization phase, it checks which values should be traced based on the options marked in the `struct rutSources_s` and then writes these values into the header of the trace file.

Listing 4.1: Frequency tracing activated?

```
152 if(!cpufreq_available()){
153     tracingData->sources.CPU_FREQ_X = 0;
154     INFOMSG("Could␣not␣find␣cpufreq!");}
155 if (tracingData->sources.CPU_FREQ_X){
156     for (int i = 0; i < tracingData->staticData.cpu_num; ++i){
157         ret = snprintf(strbuf, RUT_STRING_BUFFER_LENGTH, "CPU_FREQ_CPU
              %d", i);
158         ADD_VALUE(group, strbuf, INT64, "Hz", "CPU_FREQ");
159 }   }
```

`cpufreq_available()` checks if the sysfs interface for CPUFreq is available. If not, frequency tracing will be deactivated and an info message posted. Otherwise a unique string for every CPU will be composed and added to the traced values, along with some information like the datatype `INT64` and the unit `"Hz"` (in contrast to `"%"` for percentage).

The implementation for CPUIdle is analogue.

The tracing phase is similar, but instead of saying how the value is called and its type, the actual value is written. After every iteration, the library checks if it should stop or continue, and then waits for the length of the interval, until it begins to trace again.

The actual frequency tracing is easy, because CPUFreq offers a shared library implementing the basic functionality. `cpufreq_get_freq_kernel(unsigned int core)` returns the frequency of the given core. Voltage tracing is not yet implemented. This will be done based on IPMI. Because every frequency can be mapped to a corresponding voltage, frequency tracing is sufficient for evaluation. Voltages can be gathered by reading the processor manual or using software like k10ctl or c2ctl[1].

CPUIdle does not (yet?) have a programming interface—there has to be done a little more than at frequency tracing. First step is to save the old C-state usage times and then get the new ones.

Listing 4.2: Idle tracing

```
575 if (tracingData->sources.CPU_IDLE_X){
576     /* save old values */
577         . . .
578     /* get new values */
579     get_c_state_times(
580         tracingData->oldValues.c_states,
581         tracingData->staticData.cpu_num,
582         tracingData->staticData.c_states_num);
```

[1] `http://www.ztex.de/misc/`

`int get_c_state_times(unsigned long int *cstates, int cpu, int states)` is the function to get the C-states. It takes three arguments: the pointer to an array, where the times spend in the C-states are stored, and the number of cores and C-states. Basically, the function traverses the specific folders and files in the sysfs interface of CPUIdle and writes the values to the array [Int11].

Because the times the processors have spend in a state are total values since the last system start, the differences between formal and recent values are needed. Those differences are added together to get the times the cores have spend in idle (in microseconds).

$$\texttt{idle\_time} = \sum_{n=1}^{\text{states}} \text{time in } C_n$$

The active time (C0) is simply the interval length (given in milliseconds) minus the the time spend in idle. There might be a small divergence, because of the precision of the function used to send the tracing thread to sleep while not tracing ("`g_usleep()` may have limited precision, depending on hardware and operating system; don't rely on the exact length of the sleep [GP11a].").

The reason for not just taking the usage time of C0 as the active time is caused in the absence of regular updates for this variable, and these seldom updates are not representing active time. An explanation for this behaviour lays in the mode of operation of CPUIdle, as only the function responsible for sending cores to idle is updating the usage values.

The written values are the percentages the core has spend in these states:

$$\texttt{interval\_length} = \texttt{tracingData->interval} \cdot 1000$$

$$percentage\, Cn = \frac{\text{time in } Cn}{\texttt{interval\_length}} \cdot 100$$

4.2. Power Estimator

The structure of the power estimator resembles CPUFreq / CPUIdle. One part is the representation of a CPU (driver), the other part is a estimation strategy (governor). The driver has to provide values like the available frequencies or the power usage in the different C-states. It also has to implement miscellaneous functions for getting and setting values, and the function `double getPower(int frequency, float[] c_usage)` which is used by the governor to get the recent power consumption. To read the trace files the power estimator uses HDTraceFormat.

In the provided implementation `getPower()` uses the consumption model presented in this thesis, which is extended by the actual usage of the processor in the intervals.

$$P_{recent} = \texttt{max\_power} \cdot \frac{frequency_{recent}}{\texttt{max\_frequency}} \cdot \frac{voltage^2_{recent}}{\texttt{max\_voltage}^2}$$

`c_usage` contains the usage of the different C-states in this interval in percent. The next step is to use this information and add the consumptions of the other C-states, stored in `power_cstates` to finally get the power consumption in this step.

$$
\begin{aligned}
P_{recent} &= P_{recent} \cdot \texttt{c\_usage[0]} \\
P_{idle} &= \sum(\texttt{power\_cstates[n]} \cdot \texttt{c\_usage[n]}) \\
P_{total} &= P_{recent} + P_{idle}
\end{aligned}
$$

The implemented governor now just multiplies each consumption with its interval length and accumulates all these consumptions.

$$
E_{total} = \sum^{steps} (\texttt{getPower()} \cdot \texttt{interval[step]})
$$

The final energy consumption in Wh is $E_{total}/3600$, because interval length is given in seconds.

A governor does not have to simply sum up all the values. It can try to implement a strategy analysing the actual or next steps and alter values to lower the needed power consumption. This scenario is useful on systems not applying any energy saving mechanisms to show a possible effect in reducing the overall power consumption.

Example

A short example shows the sanity of the calculations given above. Table 4.1 shows experienced values for an example CPU.

Table 4.1.: Variables for power estimator calculation example

Variable	Value
`power_cstates`	{0, 5, 2, 1}
`frequencies`	{1833, 1333, 1000}
`voltages`	{1.1, 1.0, 0.9}
`max_power`	100

The consumption will be computed with the following values given for this interval

- `frequency` = 1333
- `c_usage` = {0.3, 0.1, 0.1, 0.5}

Calculation begins with the main formula based on the baseline values, which are just the values for P0.

$$\begin{aligned} P_{recent} &= 100 \cdot \frac{1333}{1833} \cdot \frac{1.0^2}{1.1^2} \\ &= 60.1 \end{aligned}$$

The result in P_{recent} will be further used with the usage information for C0—the percentage the processor was active in this interval

$$\begin{aligned} P_{recent} &= 60.1 \cdot 0.3 \\ &= 18.0 \end{aligned}$$

The power consumption for the active processor in this step is 18.0 W. To get the consumption for the idle processor, the power consumption in the idle states have to be multiplied with their usage

$$\begin{aligned} P_{idle} &= 0.1 \cdot 5 + 0.1 \cdot 2 + 0.5 \cdot 1 \\ &= 1.2 \end{aligned}$$

Resulting in the complete power consumption in this interval

$$\begin{aligned} P_{total} &= P_{recent} + P_{idle} = 18.0 + 1.2 \\ &= 19.2 \end{aligned}$$

which seems reasonable, considering the processor was only 30 % of this interval active.

5. Evaluation

> "12. If all else fails, show pretty pictures and animated videos, and don't talk about performance."
>
> *(David H. Baley)*

The final chapter deals with the evaluation of the processor model described in the second chapter, as well as the evaluation of the extension to ResourcesUtilizationTracingLibrary from the previous chapter.

Measurements of the different frequency / voltage-pairs (P-states) and their corresponding power consumption of the processor (without the other system) are hard to find, because special equipment is needed respectively the mainboard has to be modified [Obo10]. Even measurements with the complete system are mostly not qualified for serious comparisons. Either there are too few measurements or the test procedure is not really described.

One exception are so called "over-clocking"-sites trying to seize the last bit of performance of a processor by using either big tower cooler, water cooling or even liquid nitrogen. This is done by rising the frequency and voltage of the processor until no stable operating is possible expressing in processor miscalculations, memory failures or complete system crashes.

5.1. Processor Model Validation

To check the accuracy of the processor power consumption model, real world measurements were taken from [Gav10]. These measurements are from a variety of different systems with different frequency / voltage-pairs—no P-states because frequency and voltage were increased "by hand"—for the whole system.

Because the essential power consumption equation (2.2, $P = U^2 \cdot f \cdot \alpha \cdot C$) only deals with the processor, not the complete system, the validation is based on the differences between the measured data-points and the calculated data-points to a baseline. This baseline is always the highest reached frequency, and therefore the highest power consumption of a test series.

The power consumption for the whole system consists obviously of the processor consumption and the consumption of the remaining system.

$$W = P + S$$

W stands for the power consumption of the whole system, P for the processor and S for the system without processor. The test systems were stressed with the LINPACK frontend LinX producing load mainly on the processor (on all available cores) and some RAM. By assuming that the rest of the system is idle, or at least always utilized the same amount, it is possible to remove S by writing

$$\begin{aligned} W_1 - W_2 &= (P_1 + S) - (P_2 + S) \\ &= P_1 - P_2 \\ &= U_1^2 \cdot f_1 \cdot \alpha \cdot C - U_2^2 \cdot f_2 \cdot \alpha \cdot C \\ &= \alpha \cdot C \cdot (U_1^2 \cdot f_1 - U_2^2 \cdot f_2) \end{aligned}$$

$$\Rightarrow W_1 - W_2 = \delta \cdot (U_1^2 \cdot f_1 - U_2^2 \cdot f_2)$$

All these values are available, except δ. A little rearrangement of the equation above and building the mean value results in δ:

$$\bar{\delta} = \frac{W_i - W_j}{U_i^2 \cdot f_i - U_j^2 \cdot f_j}$$

Based upon these equation table A.2 shows the complete results of this examination. A mean variance of 4.30 % (6.55 W) and maximum variance of 10.27 % (15.49 W) is quite good, considering the measurements are based on whole systems and quite different processor types (dual- / quad-core, with and without uncore). The calculated results were mostly above the measured data indicating the rising temperature at those high frequencies as an additional influence.

As already discussed in section 2.2.4 the approximations are just as good as the amount of data they work with. Table A.1 based on the P-states of three real world processors shows an average difference of 0.039 V (3.56 %) respectively 0.024 V (2.10 %) for the gradient and intercept approaches, whereas the frequency³ approach is off by 0.28 V (23.38 %). The reasons for this were discussed in section 2.2.2 and lead to the other two approximations. Figure 2.2 depicts this fact.

5.2. P- and C-state Tracing Evaluation

To examine the functionality of the implemented processor state tracing, a script generating a specific load pattern was used on a dual-core notebook enabling a visual check. The script can be found in the appendix A. The notebook has three P-states (frequencies: 1000 MHz, 1333 MHz, 1833 MHz), four C-states and runs the ondemand governor.

The load pattern consists of

00 s sleep for 10 seconds

10 s CPU-load on core 1 for 20 seconds

20 s CPU-load on core 2 for 20 seconds

30 s stop stressing core 1

40 s stop stressing core 2
sleep for 10 seconds

50 s writing 1024 MB to harddisk

90 s terminating trace and script

Figure 5.1 shows the visualization with Sunshot for the resulting trace. It shows well the changes of the P-states under load and the usage of the deep C-states in idle. While idle the trace reports nearly always the use of the C3-state, and never the C1- or even C2-state.

Writing to the harddisk shows an interesting picture. Usage of C3 is still very high, with some activity in C0 and idling in C2 (C1 is again unused), though the utilization of core 1 shows always 100% and core 2 has phases of high utilization, too. P-state usage for core 1 is rather high, whereas core 2 is approximately half the time in the lowest P-state.

After finishing writing, the last 10 seconds show again some idle time with nearly 100 % C3- and lowest P-state-usage.

* * *

As seen in figure 5.1 utilization of one core matches always C0-usage, except while writing to harddisk. But utilization is actually the C0-usage, so they should be exactly the same.

The utilization u_{CPU} in the time interval $[t_{i-1}, t_i]$ is calculated with

$$u_{CPU}(t_{i-1}, t_i) = 1 - \frac{x_{idle}(t_i) - x_{idle}(t_{i-1})}{x_{total}(t_i) - x_{total}(t_{i-1})}$$

whereas $x_{idle}(t_i)$ is the time spent in idle and $x_{total}(t_i)$ the total elapsed time since system start [Kre09]. There is no difference in the formula which was used in section 4.1:

$$percentage\, Cn = \frac{\text{time in } Cn}{\texttt{interval\_length}} \cdot 100$$

$x_{idle}(t_i)$ and $x_{total}(t_i)$ are provided by libgtop [GP11b], which presents the values of `\proc\stat`, which delivers statistics of the kernel and the system. Statistics for the CPU are the times spent in different situations since system start:

user normal processes executing

nice niced processes executing

system processes in kernel mode

idle twiddling thumbs

iowait waiting for I/O to complete

irq servicing interrupts

softirq servicing softirqs

While watching the tracing progress with `vmstat`, which also uses `\proc\stat`, idle counter stalls and `iowait` increases, as expected. After checking the code segment of libgtop which is in charge of providing the values from `\proc\stat`, the different results in tracing the utilization and the Co-usage are becoming clear. Libgtop reads from `\proc\stat` and sums the values up to provide $x_{total}(t_i)$.

In order to get the full idle time, the above equation has to use $x_{iowait}(t_i)$ too.

$$u_{CPU}(t_{i-1}, t_i) = 1 - \frac{x_{idle}(t_i) + x_{iowait}(t_i) - x_{idle}(t_{i-1}) - x_{iowait}(t_{i-1})}{x_{total}(t_i) - x_{total}(t_{i-1})}$$

Figure 5.2 shows the visualization of the same load pattern with the adjusted version of the tracing library. Utilization now equals Co-usage.

Table 5.1.: Tracing Statistics on notebook—average values across the whole trace

	C0 [%]	C1 [%]	C2 [%]	C3 [%]	frequency [MHz]	utilization [%]
core 1	27.50	0.00	5.84	66.12	1392	26.11
core 2	26.41	0.03	6.42	66.37	1375	25.66

Table 5.1 shows that the coverage of the C-states is 99.46 % respectively 99.23 %. The missing 1 percent is the failure given the slightly varying interval times and of course some rounding mistakes. The difference between utilization and Co-usage is now very low, but still there. This will be rechecked after using the exact interval times.

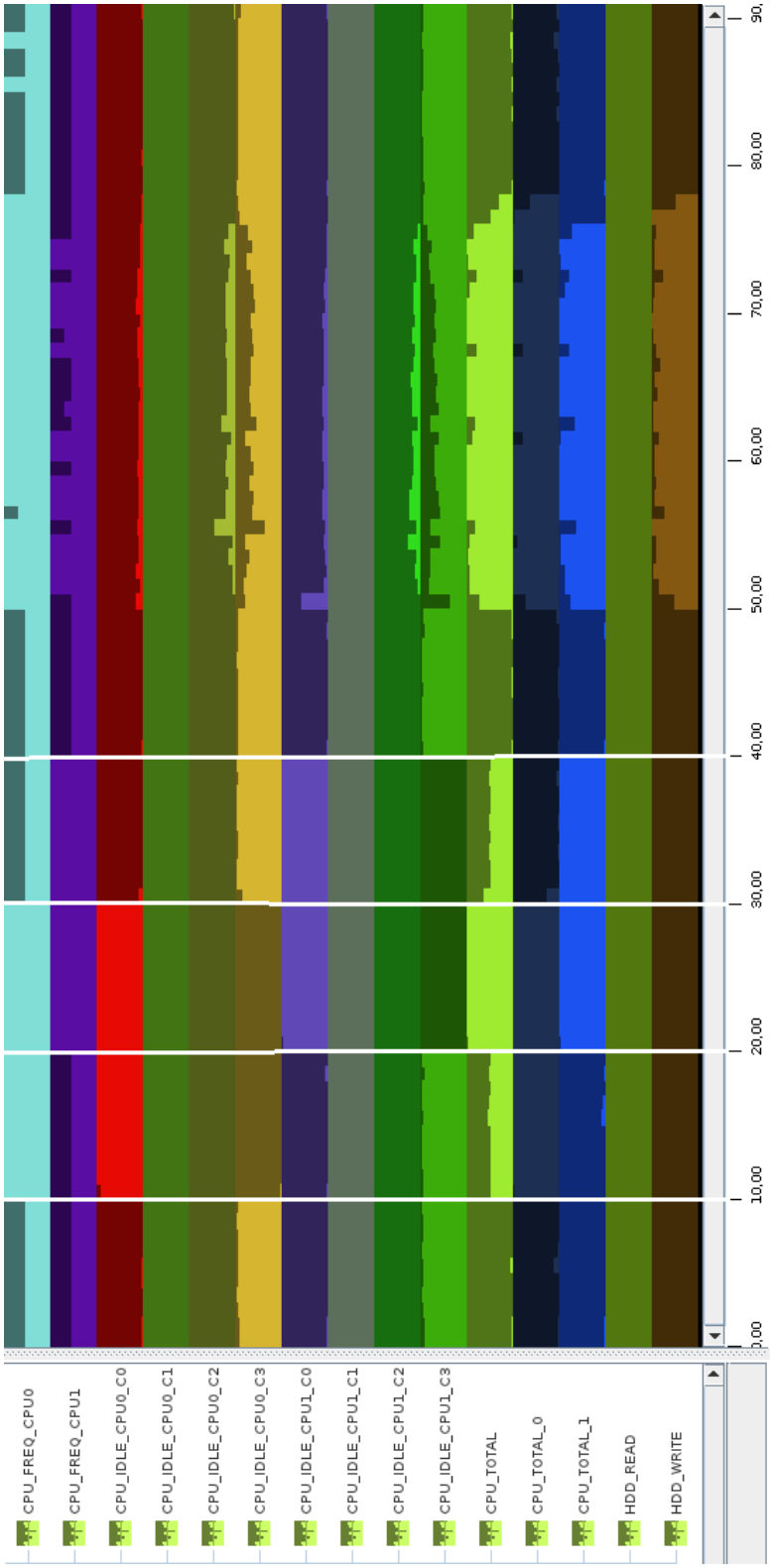


Figure 5.1.: Tracing screenshot showing utilization bug

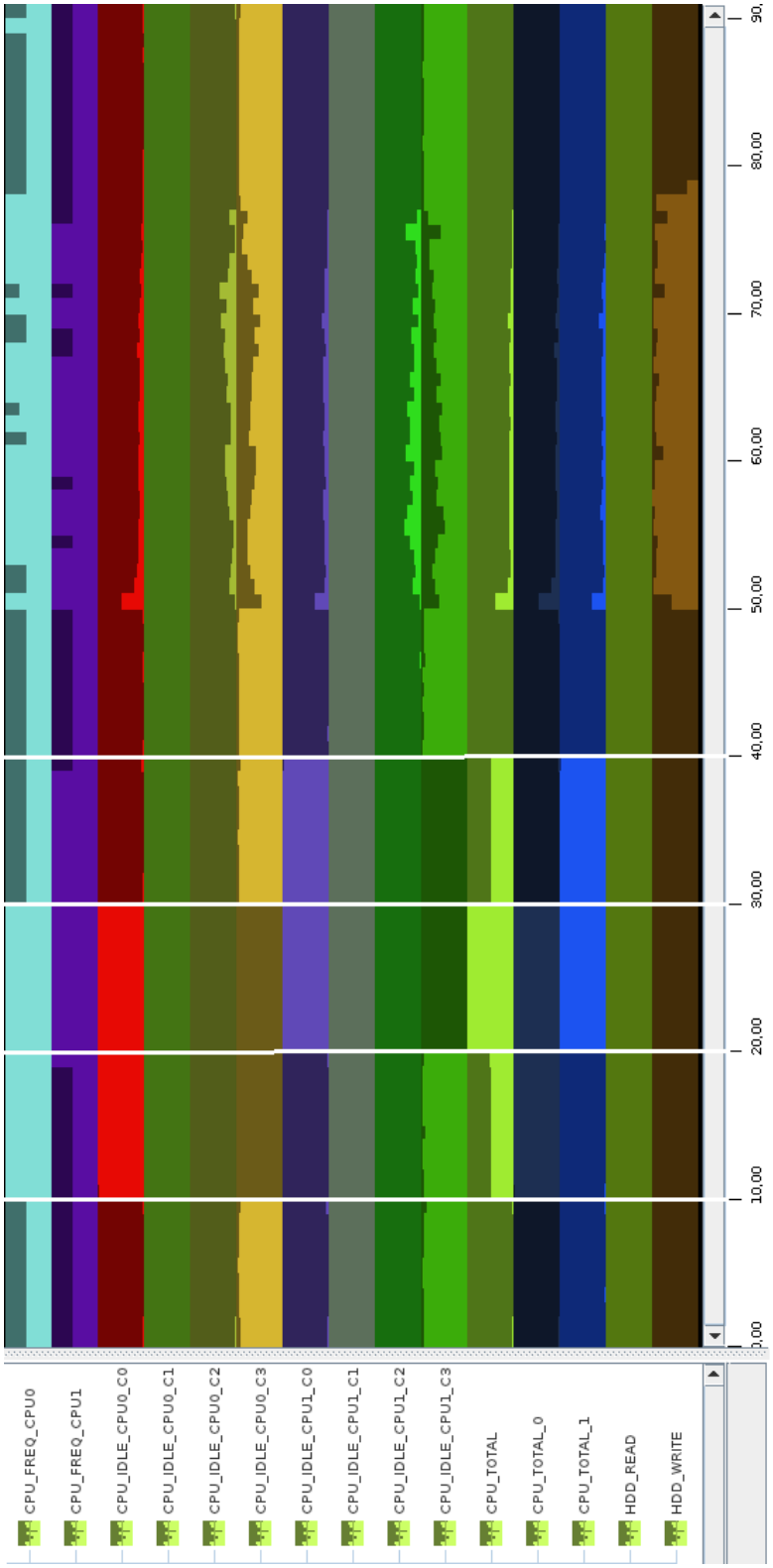


Figure 5.2.: Tracing screenshot without utilization bug

6. Conclusion and Future Work

Reducing power consumption is important. First step is always to measure what consumes when the most. It is usually not possible to measure every component of every node in a cluster. Because of that, a model has been presented calculating the power consumption of a processor based on its frequency and voltage. The resulting equation $P_2 = P_1 \cdot \frac{U_2^2 \cdot f_2}{U_1^2 \cdot f_1}$ needs a baseline measurement for P_1 at frequency f_1 and voltage U_1. The further three approximations can be used when voltage tracing is not (fully) available. Evaluation showed this is a usable way of estimating. Taking many different overclocked systems measured with different frequency- / voltage-pairs (no P-states) resulted in a mean variance of 4.30 % (6.55 W) and maximum variance of 10.27 % (15.49 W) using this model. The approximations were evaluated with the P-states of three processors. The frequency³ approach, based purely on the frequency, showed a mean difference of 23.38 %, whereas the other two (intercept and gradient approaches) based on more information involving the characteristics of the P-states were rather good with mean differences of 3.56 % respectively 2.10 %.

In order to use this model, the appropriate values have to be recorded. To do so, an existing tracing library was extended to log the P- and C-states of a processor using the kernel subsystems CPUFreq and CPUIdle. Visual inspection and the evaluation showed reasonable results.

A power estimator was implemented using the resulting trace files to finally gather the power respectively energy consumption of the processor based on the presented model. The estimator resembles the design of CPUFreq / CPUIdle allowing easy combination of different processor types and analysis strategies.

* * *

Future work consists of improvements to the model by measuring the impact of the temperature change and the consumption of the uncore-part of the CPU.

ResourcesUtilizationTracingLibrary has to be further extended to include the usage of the real elapsed time for calculating the active processor time, and voltage tracing with IPMI has to be implemented.

Later improvements also include merging the power estimator into HDPowerEstimation and considering the amount of state-changes for a more detailed power consumption, as well as developing strategies to predict energy saving potential on hardware not using energy saving mechanisms.

All together, it shows that this approach is a reasonable and working way to estimate the power consumption used by the processor.

A. Appendix

Tables

Table A.1 compares the accuracy of the three processor power consumption approximations (see 2.2.2) for three processors based on their actual P-states, based on table 3.1. All values are simple differences. Last column shows the total variation, based on

$$percentage = \frac{\sum \text{differences}}{\sum \text{target voltages}}$$

Table A.1.: Frequency³ vs. Gradient vs. Intercept approach

	P1 [V]	P2 [V]	P3 [V]	P4 [V]	P5 [V]	P6 [V]	P7 [V]	Δ [%]
Core 2 Extreme X6800								
Frequency³ Δ	-0.077	-0.170	-0.277	-0.378	-0.472	–	–	22.66
Gradient Δ	-0.007	-0.028	-0.061	-0.093	-0.114	–	–	4.97
Intercept Δ	0.007	0.000	-0.020	-0.038	-0.045	–	–	1.82
Athlon A64 X2 4800+								
Frequency³ Δ	-0.113	-0.200	-0.288	-0.350	-0.413	-0.475	-0.538	27.38
Gradient Δ	-0.034	-0.043	-0.052	-0.036	-0.020	-0.004	0.012	2.32
Intercept Δ	-0.034	-0.043	-0.052	-0.037	-0.021	-0.005	0.011	2.34
Athlon A64 X2 3600+								
Frequency³ Δ	-0.094	-0.189	-0.283	-0.378	–	–	–	20.09
Gradient Δ	0.016	0.032	0.048	0.064	–	–	–	3.40
Intercept Δ	0.010	0.020	0.030	0.040	–	–	–	2.13

* * *

Table A.2 compares the measured power consumption with the calculated consumption based on the equations from section 5.1.

Table A.2.: Processor Power Consumption Verification

f [GHz]	U [V]	$P_{measured}$ [W]	$P_{calculated}$ [W]	Δ [W]	Δ [%]
Athlon II X2 255					
3.1	1.40	111.6	115.2	3.59	3.12
3.2	1.40	113.1	117.2	4.13	3.52
3.4	1.40	116.4	121.3	4.90	4.04
3.6	1.40	120.5	125.4	4.88	3.89
3.8	1.50	140.9	140.9	0.00	0.00
Athlon II X4 635					
2.9	1.40	146.3	148.8	2.44	1.64
3.0	1.40	149.7	152.3	2.58	1.69
3.2	1.40	156.4	159.4	2.96	1.86
3.4	1.40	162.7	166.4	3.73	2.24
3.5	1.50	188.3	188.3	0.00	0.00
Athlon II X2 555					
3.2	1.40	122.7	133.9	11.20	8.36
3.4	1.40	125.9	137.9	11.96	8.67
3.6	1.40	128.3	141.8	13.52	9.53
3.8	1.40	130.8	145.8	14.97	10.27
4.0	1.55	167.6	167.6	0.00	0.00
Athlon II X4 965					
3.4	1.40	186.3	195.0	8.65	4.44
3.6	1.40	192.0	203.4	11.35	5.58
3.8	1.40	197.5	211.8	14.26	6.73
3.9	1.50	240.2	240.2	0.00	0.00
Core2Duo E7600					
3.06	1.28	95.5	101.3	5.81	5.73
3.2	1.28	97.3	103.9	6.59	6.34
3.4	1.28	100.6	107.6	6.98	6.49
3.6	1.28	103.9	111.3	7.37	6.63
3.8	1.35	116.7	123.5	6.75	5.47
4.0	1.50	147.0	147.0	0.00	0.00
Core2Quad Q9505					
2.83	1.28	125.9	127.8	1.93	1.51

Processor Power Consumption Verification

f [GHz]	U [V]	$P_{measured}$ [W]	$P_{calculated}$ [W]	Δ [W]	Δ [%]
3.0	1.28	132.6	133.4	0.78	0.59
3.2	1.28	138.8	139.9	1.12	0.80
3.4	1.28	145.6	146.5	0.85	0.58
3.6	1.28	150.3	153.0	2.69	1.76
3.8	1.35	172.3	174.6	2.26	1.29
3.9	1.40	189.0	189.0	0.00	0.00
Core i3-540					
3.07	1.13	85.9	86.8	0.88	1.02
3.2	1.13	88.1	88.8	0.69	0.78
3.4	1.20	106.7	99.1	7.58	7.65
3.6	1.20	109.6	102.6	6.96	6.78
3.8	1.30	118.3	117.8	0.54	0.46
4.0	1.30	120.2	121.9	1.68	1.38
4.2	1.38	136.3	136.3	0.00	0.00
Core i7-860					
2.8	1.13	154.1	160.6	6.45	4.02
3.0	1.13	158.5	167.9	9.40	5.60
3.2	1.13	165.0	175.3	10.25	5.85
3.4	1.13	170.6	182.6	11.99	6.57
3.6	1.20	201.4	208.2	6.76	3.25
3.8	1.20	241.7	244.1	2.40	0.98
4.0	1.38	277.2	277.2	0.00	0.00
Core i7-950					
3.07	1.20	189.8	181.6	8.24	4.54
3.2	1.20	203.7	188.2	15.49	8.23
3.4	1.20	212.3	198.5	13.85	6.98
3.6	1.20	219.9	208.7	11.22	5.38
3.8	1.20	234.4	218.9	15.48	7.07
4.0	1.30	264.9	264.7	0.21	0.08
4.2	1.40	317.0	317.0	0.00	0.00

CPUFreq sysfs

Governor specific information and settings can be found under

```
$ ls -l /sys/devices/system/cpu/cpufreq/GOVERNOR
```

ondemand There are a number of sysfs file accessible parameters:

↳ `sampling_rate`
this is how often the kernel should look at the CPU usage to make decisions on what to do about the frequency, in microseconds.

↳ `show_sampling_rate_min`
The sampling rate is limited by the HW transition latency or by kernel restrictions.

↳ `up_threshold`
defines what the average CPU usage between the samplings of `sampling_rate` needs to be for the kernel to make a decision whether it should increase the frequency. For example when it is set to its default value of 95 it means that between the checking intervals the CPU needs to be on average more than 95 % in use to decide that the CPU frequency needs to be increased.

↳ `ignore_nice_load`
this parameter takes a value of 0 or 1. When set to 0 (default), all processes are counted towards the "cpu utilisation" value. When set to 1, the processes that are run with a "nice" value will not count (and thus be ignored) in the overall usage calculation. This is useful if CPU intensive calculations are running on a laptop and it is not important how long it takes to complete.

conservative The governor is tweaked in the same manner as the "ondemand"-governor through sysfs with the addition of:

↳ `freq_step`
this describes what percentage steps the CPU frequency should be increased and decreased smoothly by. By default, the CPU frequency will increase in 5 % chunks of the maximum CPU frequency. It is possible to change this value to anywhere between 0 and 100, where 0 will effectively lock the CPU at a speed regardless of its load whilst 100 will make it behave identically to the "ondemand"-governor.

↳ `down_threshold`
same as the `up_threshold` found for the "ondemand"-governor, but for the opposite direction.

Specific information per CPU X can be found under

```
$ ls -l /sys/devices/system/cpu/cpuX/cpufreq
```

↳ `bios_limit`
frequency limit as told by the BIOS.

↳ `cpuinfo_min_freq`
the minimum operating frequency the processor can run at, in kHz.

↳ `cpuinfo_max_freq`
the maximum operating frequency the processor can run at, in kHz.

↳ `cpuinfo_transition_latency`
the time it takes to switch between two frequencies, in nanoseconds.

↳ `cpuinfo_cur_freq`
current frequency as obtained from the hardware in kHz. This is the frequency the CPU really runs at.

↳ `scaling_available_governors`
available CPUFreq governors in this kernel.

↳ `scaling_available_frequencies`
available frequencies in kHz.

↳ `scaling_governor`
currently activated used CPUFreq governor—"echoing" the name of another governor changes it.

↳ `scaling_cur_freq`
current frequency as determined by the governor and CPUFreq core in KHz. This is the frequency the kernel thinks the CPU runs at.

↳ `scaling_driver`
used CPUFreq driver.

↳ `scaling_setspeed`
if "userspace"-governor is selected, current speed can be read here. Changing speed can be done by "echoing" a new frequency, but only within the limits of `scaling_min_freq` and `scaling_max_freq`.

↳ `scaling_min_freq`
the current minimum frequency limit in kHz—"echoing" a new value changes the limit.

↳ `scaling_max_freq`
the current maximum frequency limit in kHz—"echoing" a new value changes the limit.

↳ `affected_cpus`
CPUs that require software coordination of frequency.

↳ `related_cpus`
CPUs that need some sort of frequency coordination, whether software or hardware.

Tracing Test Script

The following script is used to start / stop tracing, stressing the CPU using `stress`[1] and writing to HDD with `dd`. It is based on the script used in [Kreo9].

Listing A.1: Tracing Test Script

```
#!/bin/bash

set -b

echo [00s] start tracing CPU
./traceLoop "CPU" &
PID=$!

sleep 10
echo [10s] start stressing first CPU with 100%
taskset 0x1 stress --cpu 1 --timeout 20s &

sleep 10
echo [20s] start stressing second CPU with 100\%
taskset 0x2 stress --cpu 1 --timeout 20s &

sleep 10
echo [30s] stop stressing first CPU

sleep 10
echo [40s] stop stressing second CPU

sleep 10
echo [50s] start writing to local HDD
dd if=/dev/zero of=/tmp/verifyRUT.tmp bs=16M count=64 2> /dev/null &

sleep 40
echo [90s] terminate program
kill -INT $PID
```

[1] `http://weather.ou.edu/\protect\kern+.1667em\relaxapw/projects/stress/`

Tools and Software

A bunch of great free and open tools has been used to produce this thesis:

- *TEXworks*[2] for writing and *luaLATEX*[3] for producing text,
- *LibreOffice Calc*[4] for data analysis,
- *Gnuplot*[5] for plotting data and *PGF/TikZ*[6] for generating graphics, as well as *GIMP*[7] for screenshot modifications,
- *Eclipse*[8] for writing and exploring code,
- revision control systems *git*[9] for code and *subversion*[10] for the thesis.

Fonts used in this thesis are *Linux Libertine*[11] as serif font, *Linux Biolinum*[12]as sans-serif font, *DejaVu Sans Mono*[13] as monospace font and the standard math font from *Computer Modern*.

[2] http://code.google.com/p/texworks/
[3] http://www.luatex.org/
[4] http://www.libreoffice.org/
[5] http://www.gnuplot.info/
[6] http://sourceforge.net/projects/pgf/
[7] http://www.gimp.org/
[8] http://www.eclipse.org/
[9] http://git-scm.com/
[10] http://subversion.apache.org/
[11] http://www.linuxlibertine.org/
[13] http://dejavu-fonts.org/

Citations

These nice citations were used at the headings of each chapter:

Andreas Bogk "UNIX is user-friendly, it just chooses its friends."[14]

Gordon E. Moore "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."[15]

Bill Gates "One thing I find myself wondering about is whether we shouldn't try and make the 'ACPI' extensions somehow Windows specific. [...] Or maybe we could patent something related to this."[16]

Verity Stob "The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user."[17]

David H. Baley "12. If all else fails, show pretty pictures and animated videos, and don't talk about performance."[18]

[14] http://chaosradio.ccc.de/cr40.html

[15] ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

[16] http://antitrust.slated.org/www.iowaconsumercase.org/011607/3000/PX03020.pdf

[17] http://www.regdeveloper.co.uk/2006/01/11/exception_handling/page2.html

[18] http://crd.lbl.gov/\protect\kern+.1667em\relaxdhbailey/dhbpapers/twelve-ways.pdf

List of Figures

List of Tables

Listings

Bibliography

[BAEP08] M. Bao, A. Andrei, P. Eles, and Z. Peng. Temperature-aware voltage selection for energy optimization. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, number 978-3-9810801-3-1, pages 1083–1086, New York, NY, USA, 2008. ACM.

[Bel10] Patrick Bellasi. Linux Power Management Architecture—A review on Linux PM frameworks. `http://home.dei.polimi.it/bellasi/lib/exe/fetch.php?media=teaching:2009:linuxpowermanagement.pdf`, December 2010. last checked: 21st March 2011.

[Bro10] Len Brown. Saving energy with intel_idle cpuidle driver. `http://video.linux.com/video/1788` and `http://events.linuxfoundation.org/slides/2010/linuxcon2010_brown.pdf`, 2010. last checked: 21st March 2011.

[Bro11] Dominik Brodowski. *CPU frequency and voltage scaling code in the Linux™ kernel*. /Documentation/cpu-freq/, 2011. `http://www.kernel.org/doc/Documentation/cpu-freq/`.

[cha07] charge-n-go. PC Power Management Guide Rev. 2.0. `http://www.techarp.com/showarticle.aspx?artno=420&pgno=7`, July 2007. last checked: 21st March 2011.

[Cor10] Jonathan Corbet. The cpuidle subsystem. `http://lwn.net/Articles/384146/`, April 2010. last checked: 21st March 2011.

[cpu11] *Supporting multiple CPU idle levels in kernel*. /Documentation/cpuidle/, 2011. `http://www.kernel.org/doc/Documentation/cpuidle`.

[DG10] Johan De Gelas. Dynamic power management: A quantitative approach. `http://www.anandtech.com/show/2919`, January 2010. last checked: 21st March 2011.

[EKR03] E. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In Babak Falsafi and T. Vijaykumar, editors, *Power-Aware*

Computer Systems, volume 2325 of *Lecture Notes in Computer Science*, pages 179–197. IBM Research Low-Power Computing Research Center Austin TX 78758 USA, Springer Berlin / Heidelberg, 2003.

[Gav10] Ilya Gavrichenkov. CPU Overclocking vs. Power Consumption. http://www.xbitlabs.com/articles/cpu/display/power-consumption-overclocking.html, December 2010. last checked: 21st March 2011.

[GP11a] The GNOME Project. GLib Reference Manual. http://library.gnome.org/devel/glib/2.28/, 2011. last checked: 21st March 2011.

[GP11b] The GNOME Project. Libgtop Reference Manual. http://library.gnome.org/devel/libgtop/2.28/, 2011. last checked: 27th March 2011.

[GP11c] The Green500 Project. Green500. http://www.green500.org, 2011. last checked: 21st March 2011.

[Gro08] Silicon Valley Leadership Group. Data center energy forecast—final report. https://microsite.accenture.com/svlgreport/Documents/pdf/SVLG_Report.pdf, July 2008.

[GS10] Peter Gräber and Leander Sturm. NVIDIA GeForce GTX 580: Runderneuerung ermöglicht Fermi im Vollausbau. http://ht4u.net/reviews/2010/nvidia_geforce_gtx_580_test/index13.php, November 2010. last checked: 21st March 2011.

[HOT96] Akio Hirata, Hidetoshi Onodera, and Keikichi Tamaru. Estimation of short-circuit power dissipation and its influence on propagation delay for static cmos gates. In *Proc. of ISCAS 96*, pages 751–754, 1996.

[Int08] Intel Corporation. *Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors*, November 2008. http://download.intel.com/design/processor/applnots/320354.pdf.

[Int09] Intel Corporation, Hewlett-Packard Company, NEC Corporation and Dell Inc. *- IPMI - Intelligent Platform Management Interface Specification Second Generation v2.0*, 2009. http://download.intel.com/design/servers/ipmi/IPMI2_0E4_Markup_061209.pdf.

[Int10] Intel Corporation, Hewlett-Packard, Microsoft, Toshiba and Phoenix Technologies. *Advanced Configuration and Power Interface Specification, Revision 4.0a*, 2010. http://www.acpi.info/.

[Int11] Intel Corporation. Powertop. http://www.lesswatts.org/projects/powertop/, 2011. last checked: 21st March 2011.

[Kre09] Stephan Krempel. Design and implementation of a profiling environment for trace based analysis of energy effciency benchmarks in high performance computing. Master's thesis, Ruprecht-Karls-Universität Heidelberg, August 2009.

[LHL05] Weiping Liao, Lei He, and Kevin M. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24 Issue:7:1042 – 1053, July 2005.

[Lud08] Thomas Ludwig. Cluster computing. Lecture script, April 2008.

[Lun06] Mark Lundstrom. Ece 612 lecture 13: Threshold voltage and mosfet capacitances, Oct 2006.

[MFMB02] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, number 0-7803-7607-2, pages 721–725, New York, NY, USA, 2002. ACM.

[Min09] Timo Minartz. Model and simulation of power consumption and powersaving potential of energy effcient cluster hardware. Master's thesis, Ruprecht-Karls-Universität Heidelberg, August 2009.

[Mud00] Trevor Mudge. Power: A first class design constraint for future architectures. *Computer*, 34:52–57, 2000.

[Mur08] San Murugesan. Harnessing green it: Principles and practices. *IT Professional*, 10:24–33, January 2008.

[Obo10] Fabian Oboril. Realer Energieverbrauch von Intels Core-i-CPUs, deren iGPU und der Chipsätze. http://ht4u.net/reviews/2010/leistungsaufnahme_intel_core_i_cpus/index8.php, November 2010. last checked: 21st March 2011.

[PLB07] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle—do nothing, efficiently. . . In *Proceedings of the Linux Symposium*, volume 2, pages 119–126, June 2007. http://ols.108.redhat.com/2007/Reprints/pallipadi-Reprint.pdf.

[PS06] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium*, volume 2, pages 223–238, 2006. http://ftp.kernel.org/pub/linux/kernel/people/lenb/acpi/doc/OLS2006-ondemand-paper.pdf.

[PS07] Venkatesh Pallipadi and Suresh B Siddha. Processor power management features and process scheduler: Do we need to tie them together? In *LinuxConf*, 2007.

[TP11] The TOP500 Project. Top500. http://www.top500.org, 2011. last checked: 21st March 2011.