Simon Holmbacka, Fredric Hällis, Wictor Lund, Sébastien Lafond and Johan Lilius

Energy and Power Management, Measurement and Analysis for Multi-core Processors

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 1117, June 2014

TUCS

Energy and Power Management, Measurement and Analysis for Multi-core Processors

Simon Holmbacka, Fredric Hällis, Wictor Lund, Sébastien Lafond and Johan Lilius
Turku Centre for Computer Science, Embedded Systems Laboratory
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland

TUCS Technical Report
No 1117, June 2014

Abstract

This technical report introduces the current evolution of computing system and their power dissipation and energy consumption. It focuses on the power and energy consumption of microprocessor and presents in more details the two mechanisms, DVFS and DPM, available to increase the energy efficiency of computer systems. Different traditional approaches to evaluate the load on a processor are discussed and an extension of the notion of load taking into consideration Quality of Service of applications is proposed. Based on the extended load notion, the construction of power and performance models and a design of a dedicated benchmark, a run-time power manager is presented.

An evaluation of the presented benchmark and run-time power manager provides early results on the achievable power and energy savings and gives a comparison of the proposed approach with by default Completely Fair Linux scheduler and load balancer.

Finally we present two ways of conducting power measurements by using both internal and external power measurement devices. An open-hardware solution to measure power from any kind of chip (provided that the current feed pin are exposed) is presented together with an open-source logger software running on a low-cost Raspberry Pi platform. This provide one concrete example to create a cost efficient power tracing device without industrial scale manufacturing equipment.

Keywords: Power, Energy, Measurements, Multi-Core, Benchmark, Model-based run-time

TUCS Laboratory
Embedded Systems Laboratory

Contents

1 Introduction

Energy efficiency and power dissipation are becoming a key issues in all types of computing systems, from hand-held devices to large scale distributed systems. On the processor level, power dissipation is a major issue due to physical restrictions of the transistors themselves. The power fed into a processor is dissipated as heat and when exceeding a certain level this heat interferes with the transistor operations. This phenomenon is called the power wall, which is the power limit a processor can dissipate safely without heat induced damage or failures.

From the point of view of the average consumer, increasing processor energy efficiency would prolong the time between recharging mobile devices such as laptops and smart-phones and reduce noise levels of active cooling. Regarding data centres this would allow them to decrease the overall electrical bill, enable them smaller cooling infrastructures and provide more affordable cloud services. Lowering the energy consumption of data centres do not only affect their operational costs and ecological footprint, but also have an impact on the possibilities to expand or construct new data centers.

In this report we present a number of processor level approaches to increase the energy efficiency through different software designs. We also include as Annex the schematic of the developed external power meter and a description of the data logger used to measure and log power values of the different studied computing platforms.

1.1 Trends in Computer Evolution

To better understand how the IT-industry expanded to its current form in only a few decades, how the rate of development is being maintained and the challenges of maintaining it, it helps to take a look at some trends and how they have affected the semiconductor revolution.

Since the invent of the transistor, computer evolution, or more precisely chip evolution, has more or less followed Moores law, which states that the complexity of a single chip will double roughly every 18 months [26]. Moore's original prediction is shown in Figure 1. Even though this was an empirical observation at the time, the accuracy of this observation has resulted in it being considered almost as a physical law.

It has been argued that Moores law is somewhat of a self fulfilling prophesy, since the industry has used the law as a guideline on which to base production goals. In this way the law has not been a prediction per say, but more of a goal towards which the industry has strived to and thus far succeeded in reaching. Nevertheless, Moores law has proven to be accurate for the last decades.

The law has also been used to draw parallels between other, closely linked, computational attributes such as computer performance. The growth of computer performance has even become a common abbreviation of mores law, and states

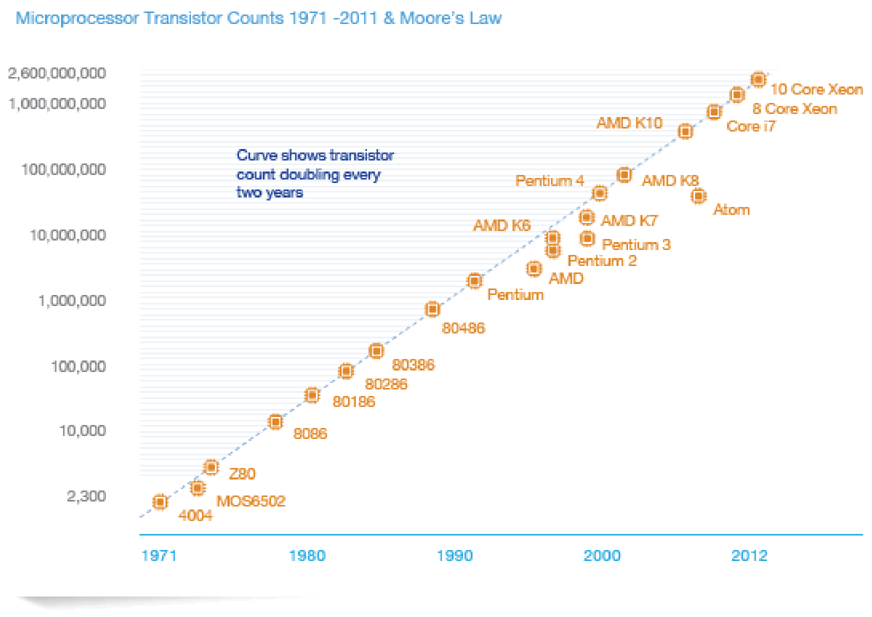


Figure 1: Moore's law: complexity of chip increasing ever 18 months [26].

that computational performance will double roughly every 18 months [20]. Over time this statement has also been shown to hold true.

Another, important observation that correlates with Moores's law, is that the energy efficiency of processors has also increased at a similar rate. This phenomenon, called Koomey's law, is partly a result of Dennard scaling which states that as transistors decrease in size they are able to switch faster and use less power [20] [3]. Koomey's law is shown in Figure 2.

However as the size of individual transistors keep reducing other problems appear. One issue is the manufacturing difficulties and the physical effects of transistors as they get smaller, which has reduced the effects of Dennard scaling [3]. In order to keep performance and efficiency along with the evolutionary rate of chip complexity, more and more effort has gone into creative design of hardware as well as software. One example of this is the introduction of parallel chips.

Over the decades, the means of achieving performance increase has been realized by clock frequency improvements, but since the middle 00s', frequency increase has decreased and even started to revert as illustrated in Figure 3.

The reason is the *power wall*, which is the maximum amount of power a processor can dissipate safely with normal levels of cooling. In short, instead of increasing the frequency for performance, the number of operational entities, cores, are instead increased. The shift from constantly increasing the frequency to introducing more cores has kept Koomey's law intact.

One can wonder why parallel computing has not become common in an earlier stage. The simple answer is that there was really no need for parallel computing in

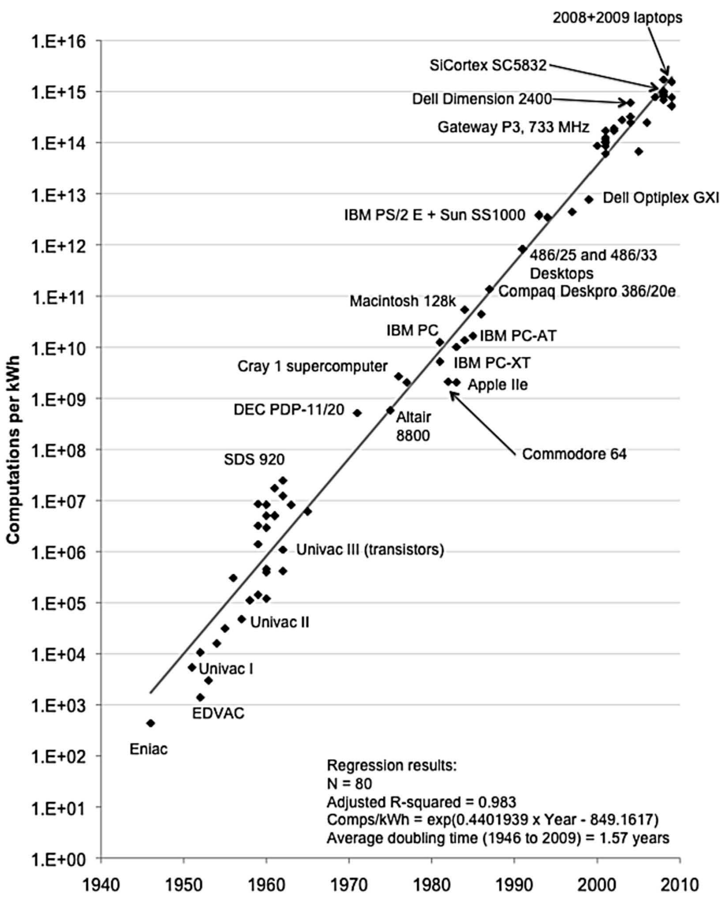


Figure 3. Computations per kilowatt-hour over time. These data include a range of computers, from PCs to mainframe computers and measure computing efficiency at peak performance. Efficiency doubled every 1.57 years from 1946 to 2009.

Figure 2: Koomey's law: computations per KWh over time [20].

the early decades of computer science. Already in 1967 Gene Amdahl presented a paper in which the drawbacks and complexities of parallel systems where presented. This work was later converted into the well known *Amdahl's Law*. Which is used to calculate the relative performance increase based on the inherent parallelism of the software and number of available computational units. Regarding the evolution of computational systems, at that time introducing parallelism did not strictly seem worth the effort. In the same year Daniel Slotnick argued that the need for re-innovation will come, and when it does, new solutions such as ways of exploiting parallelism in an efficient manner will become important. His exact words, later refereed to as Slotnick's law, where:

> *"The parallel approach to computing, it must be said however, does require that some original thinking be done about numerical analyses and data management in order to secure efficient use. In an environ-*

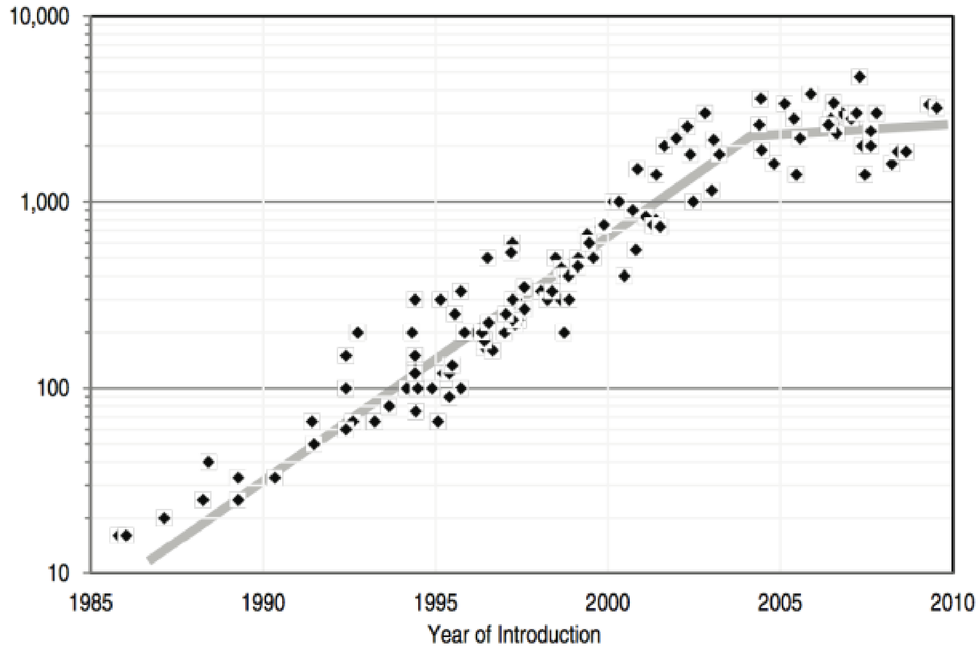


Figure 3: CPU clock frequency change over time [31] (y-axis is the clock frequency in MHz)

ment which has represented the absence of the need to think as its highest virtue this is a decided disadvantage."

2 Power Dissipation and Energy Consumption in Microprocessors

To understand how energy is consumed in multi-core chips, the different components consuming the energy must be defined and analysed. Since this report focuses on CPU/chip level energy consumption, we first address the power dissipation breakdown in semi-conductors.

2.1 Power breakdown

Power is dissipated on a microprocessor as work is being executed and by leakage in the semiconductor material. The total power dissipated by a processing element origins from two distinct sources:

1. The dynamic power dissipation P_d due to the switching activities
2. The static power dissipation P_s mainly due to leakage currents

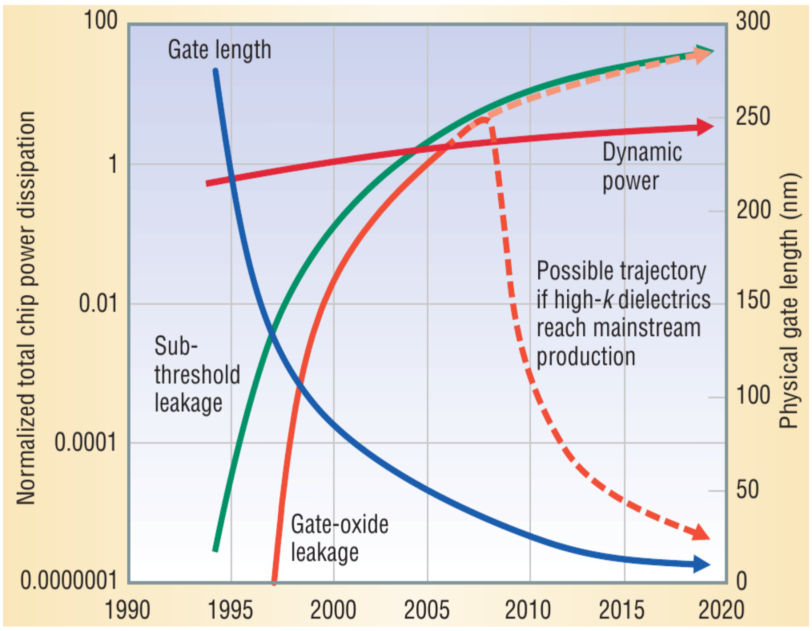


Figure 4: Total chip power breakdown and its trends [18]

The dynamic power is dissipated when the load capacitance of the circuit gates is charged and discharged. Such activities occur when the CPU functional units are active. Because the dynamic power is proportional to the square of the supply voltage $P_d \sim Vdd^2$, much effort was put into the design of integrated circuits being able to operate at a low supply voltage. However decreasing the supply voltage of integrated circuits increases propagation delays which force the clock frequency down accordingly. Therefore by dynamically adjusting the clock frequency along with the supply voltage when using performance states can maximize the power savings. The equation governing the dynamic power is given in Eq. 1.

$$P_d = C \cdot f \cdot Vdd^2 \tag{1}$$

Where C is the circuit capacitance, f is the clock frequency and Vdd is the core voltage.

The static power is dissipated due to leakage current through transistors consisting of subthreshold and gate-oxide leakage [18]. The leakage current is present as long as the chip (or parts of the chip) is powered on. Moreover, when lowering the supply voltage of integrated circuits, the subthreshold leakage current increases which also increases the dissipated static power [32, 4]. In addition to this, scaling down the technology process of integrated circuits increases the gate tunneling current which also leads to an increased static power [32]. Equation 2 shows the subthreshold leakage current

$$I_{sub} = K_1 \cdot W \cdot e^{-V_{th}/n \cdot V_\theta}(1 - e^{-V/V_\theta}) \tag{2}$$

where K_1 and n are architecture specific constants, W is the gate width and V_θ is the thermal voltage (further explained in Section 2.3 covering the thermal impact on leakage current). Gate-oxide leakage is the other leakage component and is further explained in [18].

Until recently, the power dissipated by a processing element was mainly consisting of the switching activities i.e. $P_d \gg P_s$ [5]. However due to technology scaling, the static power dissipation is exponentially increasing and starts to dominate the overall power consumption in microprocessors [19, 32, 1], which leads to increased research efforts in minimizing static power e.g. with the use of sleep states and power gating. Figure 4 shows the power breakdown and its future trends with technology scaling at that time. As seen in the figure, the static power contribution was negligable in the early and mid- 90s', and rapidly grew as the manufacturing technology was scaling down.

2.2 Energy consumption

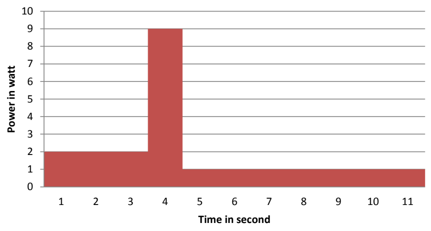
The amount of energy, in joule (J) , consumed by a processor is the product of the average processor power P_{avr} and the corresponding period of time T as shown in Equation 3

$$E = P_{avr} \cdot T \tag{3}$$

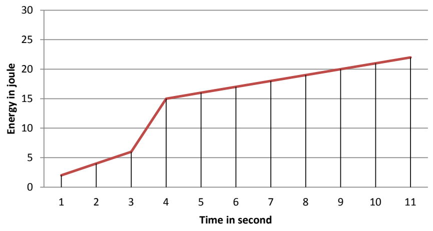
where P_{avr} is the sum of the average dynamic and static power $P_{avr} = P_{d-avr} + P_{s-avr}$ over the corresponding period of time T. Equation 3 is equivalent to Equation 4 , where the period of time T is defined by t_1 and t_2.

$$E = \int_{t_1}^{t_2} P(t) \cdot dt \tag{4}$$

The linear combination of power and time results in a two-variable optimization problem for minimizing the energy consumption. Figures 5 and 6 illustrate the relationship between the instantaneous power dissipation and energy consumption of two systems over a period of 11 seconds. While system 1, on Figure 5, has a relatively high instantaneous power dissipation at 9 watts during one second, its average power dissipation is lower that system 2, on Figure 6, which does not dissipate more than 5 watts at any-time. Even if system 1 might generate more heat during one second, it will consume 22% less energy than system 2.

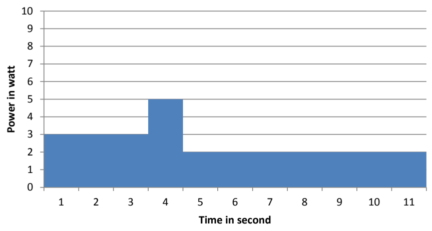


(a) Power dissipation

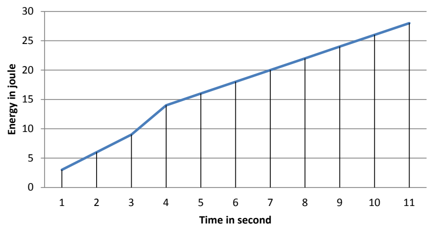


(b) Corresponding energy consumption

Figure 5: System 1 power dissipation and energy consumption



(a) Power dissipation



(b) Corresponding energy consumption

Figure 6: System 2 power dissipation and energy consumption

The notion of energy is more complex than the notion of power. The time factor t represents the execution time of a job and can be related to the notion of system performance, quality of experience or quality of service, which are notions sometimes difficult to define objectively.

We can recognize three different use cases for energy consumption and power dissipation management:

1. *Power-constrained.* The CPU is executing workload (without timing guarantees) with a power cap for limiting the power envelope. Battery time is usually maximized with this strategy while accepting some QoS degradation. For example web browsing in power-saving mode.

2. Time-constrained. The CPU is executing workload with a given deadline to obtain a desired QoS. The hardware can be utilized without restrictions, but the goal is to minimize P_{tot} while keeping QoS guarantees. For example video decoding.

3. *Operation-constrained.* The CPU is executing a given amount of work (specified by nr. operations etc.) without time limit and without limitations on the hardware. The goal is to obtain a combination of P_{tot} and t such that E is minimized. For example video encoding.

Depending on the use-case, different aspects are considered and the objective toward which energy is optimized is not always clear. For example, mobile phones often use a strategy called "race-to-idle"[28] in which the processor is executing the task as fast as possible to minimize time t while sacrificing power for a short time. This strategy might- or might not be an optimal strategy depending on the use-case. Further discussion is seen in Section 5.2.

2.3 Thermal influence

The temperature of a microprocessor directly influences the static power dissipation of the chip since the leakage current increases with increased temperature. The rate at which the static power is increased depends on the architecture and manufacturing techniques, and in this technical report we mainly focus on mobile many-core processors. As recalled from Equation 2 the leakage current I_{sub} is exponentially dependent on the thermal voltage V_θ. The thermal voltage is a measurement of the average energy of individual electrons. It increases linearly when the temperature increases by a coefficient equal to $\frac{kT}{q}$ where k is the Boltzmann's constant, q is the electron charge and T the temperature. Higher temperature modifies the subthreshold slope [29], which degrades the "efficiency" of the transistor operations. With higher temperature the voltage threshold also increases and reduce the voltage difference between an open and a closed transistor. This leads to a higher number of electrons leaking through the transistor even if the transistor is closed.

In order to determine the temperature-to-power ratio, we let a quad-core processor (ARM based Exynos 4) idle with no workload in different ambient temperatures.

Figure 7 shows the increase in static power as a function of the temperature for both board and CPU level measurements. At the left hand side of the curve, the chip was put in a freezer and its internal temperature was measured to be $1\,^\circ\mathrm{C}$, and afterwards it was placed in room temperature and heated up to $80\,^\circ\mathrm{C}$ with an external heat source. As seen from the figure, the power dissipation of the chip increases more than twofold depending on the ambient temperature conditions. The sudden drop in chip power at the $80\,^\circ\mathrm{C}$ point is due to the chip's frequency throttling mechanism, which is automatically activated at this point in order to prevent overheating leading to physical and functional damage.

3 DVFS and DPM

Over the years, a number of different mechanisms targeting the CPUs have been developed in order to increase the energy efficiency of computer systems. One common feature is to exploit variations in the level and type of a systems computational load. This is done in order to tailor the performance and the power dissipation to the workload. These mechanisms monitor and/or predict the system load and place the CPU in a matching operational state. If the load is low, or of

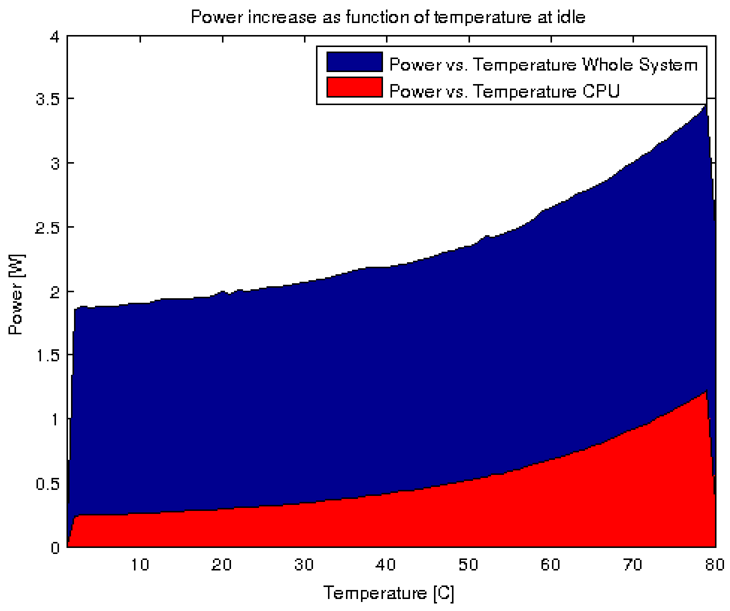


Figure 7: Static power dissipation as function of ambient temperature for an idling ARM based Exynos 4 board

a nature that does not require high instantaneous performance, the mechanisms are used to put the system in a low-power operational state and vice versa. The availability and type of these operational states vary depending on the CPU architecture and manufacturer. The exact way of reaching the operation states and the transition duration depends on the specific mechanisms and the algorithm it employs. Quite often these mechanisms can be configured or influenced depending on the user's preferences and the system's use case. In some cases the exact operation state can be statically defined by the user.

Mechanisms that have been developed for this purpose are DVFS and DPM.

3.1 DVFS

DVFS is a common technique used in from server grade machines, desktops, laptops to tablets, smart-phones to varieties of embedded systems. DVFS focuses on the dynamic power dissipation of a CPU, where the dynamic power means the power used for actual work. The dynamic power is dependent on the total capacitance of the CPU, the switching activity i.e. the frequency and the supply voltage squared. This relationship is shown in equation 5.

$$P_{dynamic} = C \cdot f \cdot V^2 \tag{5}$$

By utilizing DVFS it is possible to reduce the dynamic power dissipation by adjusting the clock frequency and/or supply voltage in order to match the load on

Frequency	Voltage	P-state
3.4 GHz	1.0808 - 1.058 V	P_0
3.3 GHz	1.0608 V	P_1
3.2 GHz	1.0405 - 1.458 V	P_2
3.0 GHz	1.0057 - 1.0107 V	P_3
2.9 GHz	0.9907 - 0.9957 V	P_4
2.8 GHz	0.9757 - 0.9807 V	P_5
2.7 GHz	0.9657 V	P_6
2.6 GHz	0.9557 V	P_7
2.4 GHz	0.9357 V	P_8
2.3 GHz	0.9257 V	P_9
2.2 GHz	0.9202 V	P_{10}
2.1 GHz	0.9156 V	P_{11}
2.0 GHz	0.9056 - 0.9106 V	P_{12}
1.8 GHz	0.9006 - 0.9056 V	P_{13}
1.7 GHz	0.9006 - 0.9056 V	P_{14}
1.6 GHz	0.8956 - 0.9006 V	P_{15}

Table 1: P-states for a Intel Core i7-3770 processor

the system. From the equation it can be seen that the relationship between the dynamic power and the frequency is linear, whilst the relationship between the dynamic power and the supply voltage is quadratic. This makes it more beneficial to reduce the supply voltage rather than the frequency. However, reducing the supply voltage increases the propagation delay in the transistors which means that the frequency must also be sufficiently reduced to accommodate any delays as a results of supply voltage reduction. This also affects the opposite scenario in the same way; by increasing the frequency beyond a certain point, the supply voltage must be increased due to the propagation delays.

In practice DVFS is realized through a set of states, each having a predefined voltage and frequency setting. The states are defined by the Advanced Configuration and Power Interface (ACPI) and are called performance states or P-states. The ACPI provides a standardized interface which the operating system can utilize. Whereas the availability of the different voltage and frequency combinations, the actual P-states, are hardware dependent. The level of the state is indicated by the P-state number, $P_0 ... P_n$, where higher number means a higher power saving. Different CPU manufacturers provide different P-states, both varying in number and frequency/voltage combinations for their processors.

As example, a set of available P-states on an Intel Core i7-3770 processor can be seen in Table 1. The table was derived by manually switching between all available frequencies and by monitoring the core voltage through the i7z monitoring tool.

Each of these states impacts the power dissipation of the CPU in accordance to equation 5. This effect is visualized in Figure 8 which shows the power dissipation of each of the different Intel i7 P-states during full load. Full load was obtained by stressing the CPU with the *stress* CPU test. The CPU power was isolated and monitored utilizing a power measurement and logger device presented in section .

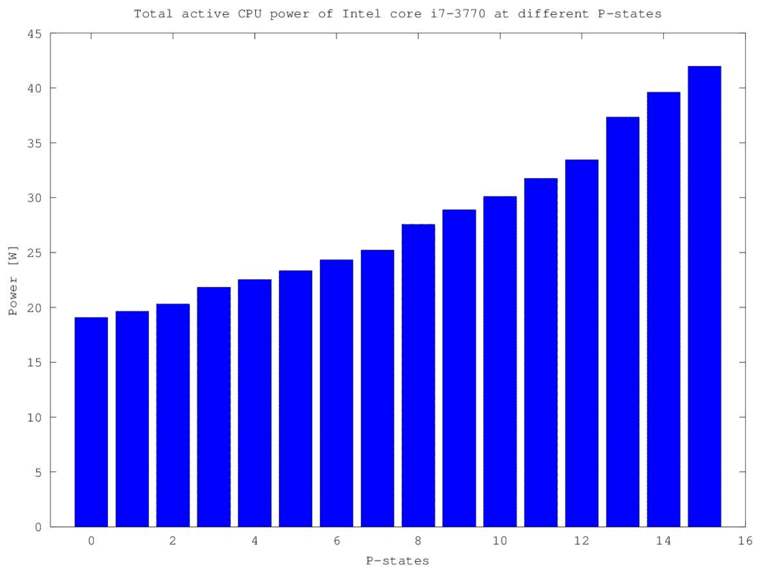


Figure 8: Total CPU power dissipation of Intel core i7-3770 in different P-states during load

The figure shows the exponential relationship between the CPU power dissipation and supply voltage as presented in Eq. 5. The figure also shows the significant difference in power dissipation depending on which P-state is chosen, which shows the potential impact of DVFS.

However, a linear increase in performance will usually result in an exponential increase in power due to the effect of increasing the voltage. This also means that an exponential decrease in power dissipation only requires a linear reduction in performance. In order to provide a good trade-off between performance and power dissipation, a number of different DVFS algorithms and methods used to match different circumstances have been developed. For example, mobile devices, such as laptops, smart-phones and tablets might utilize a relatively strict DVFS method in order to preserve as much power as possible while more stationary systems such as servers and desktops might use a more lenient method in order to trade a lower power consumption against better system responsiveness.

DVFS can also be used for other purposes than power management; some manufactures for example utilize the same technique for processor thermal control. By controlling the processors temperature, thermal related computational errors and long term thermal damage can be kept to a minimum which increases both the systems predictability and lifespan. Other benefits are lowered noise levels due to lower utilization of the CPU fan, which usually is the noisiest component in a computer. One example of DVFS being used and marketed for this purpose is the AMD cool and quiet system.

3.2 DVFS in Linux

In Linux, the entities responsible for governing the DVFS and switching between P-states, for the purpose of power management, are appropriately named frequency *Governers*. In the default Linux kernel, the following governors are common: *Performance*, *Powersave*, *Userspace*, *Conservative* and *Ondemand*. Other DVFS methods, the equivalent of governors, used in Windows and Macintosh systems will not be addressed in this report since they are not open source and difficult to analyse in detail.

Powersave governor The Powersave governor, as the name implies, focuses on minimizing the power dissipation, regardless of the circumstances. Its main, and only functionality, is to statically assign the absolute lowest frequency and voltage the system is capable of achieving. Activating the Power save governor will lock the system to its highest P-state P_n.

Performance governor The Performance governor is the complete opposite of the Powersave governor: it maximizes the performance under all circumstances. Activating the Performance governor will lock the systems in its lowest P-state P_0, assuring maximum frequency at all times.

Userspace The Userspace governor also statically assigns a P-state for the system. However, the Userspace governor allows the user to decide which P-state to use. In practice, the user is able to choose the operation frequency of the CPU and the system will adjust the voltages accordingly.

Conservative The conservative governor monitors the system load and adjusts the P-state gradually until the frequency level is high enough to support the workload of the system. Similarly if the load is decreased it will gradually jump to a higher P-state i.e. lower frequency. The governor measures the total system load as a percentage of active CPU time over a predefined time window. For example if the time window is set to 10ms and the CPU has been active for 5ms during this period, the governor will perceive the CPU load as 50%. The governor is user tunable, which means that the user can tailor its behaviour. The tunable parameters include the time window over which the load is measured, the up and down thresholds for the load levels and the magnitude of the P-state jumps themselves. A figure of the conservative governors behaviour under load can be seen in Figure 9.

The figure was derived by monitoring the frequency setting chosen by the conservative governor, starting at idle, initiating a load spike, by using the *stress* test and finally letting the CPU reach idle state. The CPU used in this visualization was an Intel Core i7-3770 with a maximum frequency of 3.4 GHz and a minimum frequency of 1.6 GHz running Ubuntu 13.10 with a 3.7.0 kernel version. The convervative Governor was reconfigured in order to slow down its response time and simplify the frequency monitoring, while still retaining the basic behaviour pattern of the governor. The OS is capable of changing between 16 different P-states on the CPU by increasing the CPU frequency, from the lowest of 1.6 GHz to its highest setting in increments of a 0.1 GHz. The exact set of P-states, and their corresponding frequency and voltage settings can be seen in Table 1.

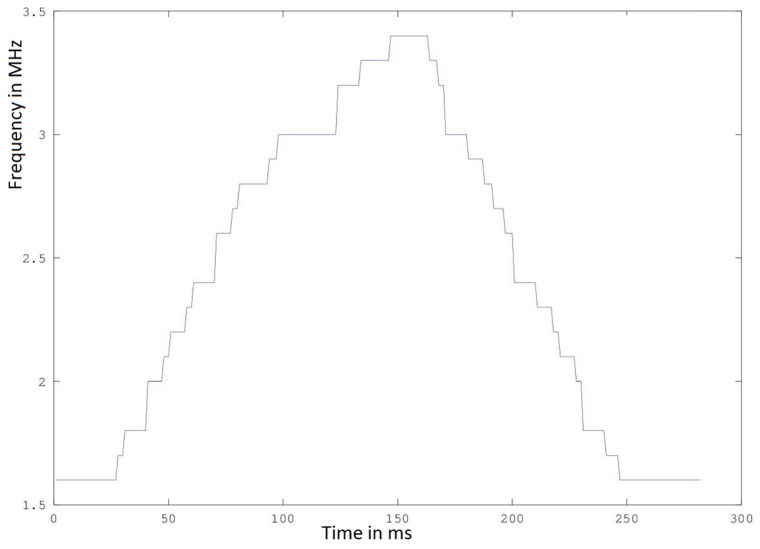


Figure 9: Clock frequency output when using the conservative governor

Ondemand The ondemand governor works in a somewhat similar fashion than the conservative governor, however, it is designed to supply fast performance increase on demand. This governor also monitors the system load, in the same time-windowed fashion as the conservative governor, but when reaching high load the odemand governor will jump straight to the lowest P-state with the highest frequency setting, available. If the load does not require the highest performance point, the governor will calculate the lowest frequency capable of keeping the load beneath the up-threshold limit and select the P-state corresponding to this frequency in order to obtain a suitable performance level. As the conservative governor, ondemand, also includes a set of tunable parameters to enabling the user to tailor its behaviour. A figure of the ondemand governors behaviour during load can be seen in Figure 10. This figure was also derived by monitoring the frequency changes during load on the same OS and hardware platform as in Figure 9.

The load in this particular test was created by utilizing *Spurg-bench*, which enables the creation of load at certain load percentages (further discussed in Section 7). This was done in order to grant the governor some leeway in choosing the appropriate frequency. By utilizing *Spurg-bench* it was possible to induce a somewhat varying load scenario over a convenient time interval. Spurg-bench was set to run three sets of tasks, each set with four threads where each thread was set to contribute 30% to the total load. As in the previous scenario, the ondemand governor was configured to provide for an optimal measuring setting. Hence, it is important to note that the figure does not depict an optimal ondemand performance, rather it visualizes its behaviour.

From the figure, the behaviour of the ondemand governor is relatively easy to disconcern. As the load starts the governor immediately chooses the highest frequency. The governor later deemed this frequency to high and calculated the lowest feasible frequency. Due to the slightly varying nature of the load, the chosen new frequency was predicted too low and the governor immediately changes to the highest frequency once again. This behaviour can bee seen to repeat itself with varying settings for the lower frequency until the governor finds a stable frequency at 3.2 GHz. The behavioral pattern of the ondemand governor shows that it is mainly focused on delivering optimal performance as fast as possible should the need arise as, opposed to the conservative governor.

Even though the principal implementation of a DVFS systems is relatively straight forward, there are a number of important factors that needs to be accounted for when designing such a system. Some of the most important issues include:

- P-state transition latency
- Load monitoring
- Performance versus power trade-off

Switching latencies are an issue which must be taken into account when working with a DVFS system. Even though the switch between P-states can be done in the order of milliseconds, this is a considerable amount of time from a processors point of view.

3.3 DPM

DPM, is a common method for minimizing the static power dissipation of a CPU. As shown in equation 2, the static power is based on the leakage current times the supply voltage. The main purpose of DPM is to asses the load, and cut the supply voltage to parts of the chip or system not currently needed. DPM can be realized either through CPU sleep-states or more directly through CPU Hot-plugging

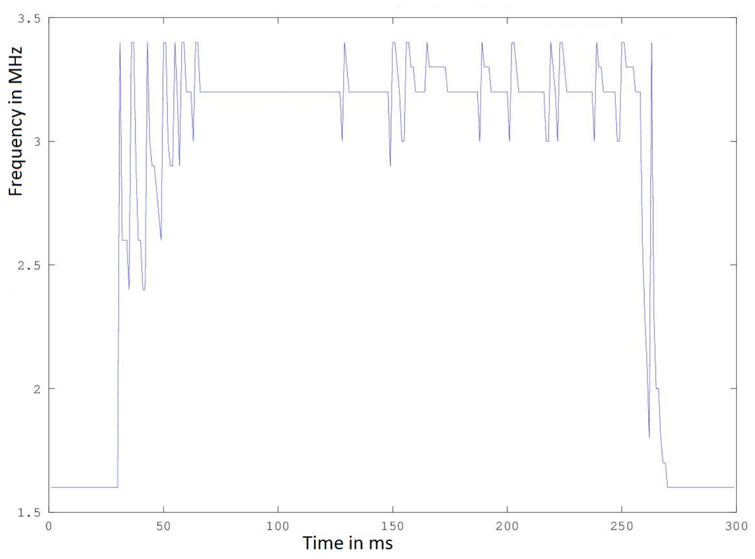


Figure 10: Clock frequency output when using the ondemand governor

3.3.1 Sleep states

The states for different DPM settings are called sleep-states or C-states. The range of available C-states, $C0...Cn$, vary and is highly dependent on the underlying hardware. The higher the C-state value the deeper the sleep and the greater the power savings are. The origin of the C-state model is the ACPI Interface specification, from where a more detailed description can be found [24]. In general, deeper sleep states introduces longer transition latencies. As example, the available sleep states on the Intel i7-3770 and their description are as follow:

- C0: The highest C-state. Essentially the active state of the CPU. The state in which it resides whilst executing instructions.

- C1: is a state that, according to the ACPI standard, must be supported by all hardware architectures. This state is realized through the halt instruction (HLT). When a core enters this state, clock signals to some parts of the chip are gated. The bus interface unit and advanced programmable interrupt controller remain clocked.

- C1E (Enhanced C1): This state is similar to C1, however, this state is also capable of lowering the supply voltage.

- C3: This states offers further improved power savings by completely stopping the internal clock signals, at the cost of transition latencies. In this state the cache is maintained but snoops are ignored.

- C6: The state of the core is saved inside a special static RAM, outside the CPU itself. The voltage to the core can be completely cut off and the core can essentially be shut down.

To illustrate the power saving potential of the different C-states we measured the power consumption during idle when the processor was limited to a certain C-state and frequency. The test was conducted on an Intel i7-3770 with the available sleep-states as described in above and four cores available for all tests. For each level of allowed C-state the frequency was statically set to the lowest: 1.6 GHz, and gradually increased to the highest: 3.4 GHZ, with increments of 200 MHz. The power was measured during each step. The results are depicted in Figure 11.

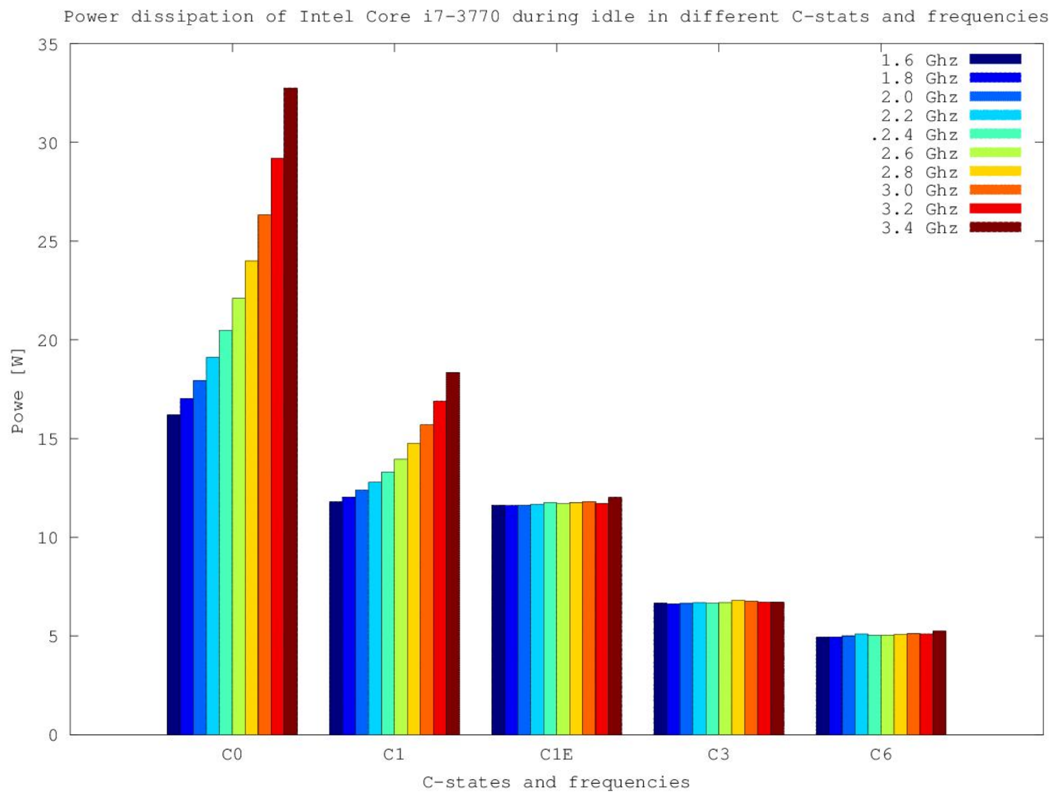


Figure 11: Power dissipation of CPU in different C-states and frequencies

From the figure the different C-state potential becomes clear. If only $C0$ is allowed a significant amount of power is used even during idle, and the impact of utilizing DVFS is quite large. The next set of columns depicts the effect of using $C1$ when the CPU run the *HLT* instruction. From the Figure it can be noticed than from state $C1E$ decreasing the clock frequency does not influence any more the power dissipation of the CPU.

Load and sleep states The time a task will use the processor between system calls is in general non-deterministic. The reasons for this are among others:

1. The time it takes for the hardware to execute a given set of instructions is in general non-deterministic.
2. Non-deterministic input makes impossible to know which set of instructions will be executed between system calls.
3. The Entscheduingsproblem is not generally solvable [35, 7].

These reasons together with the latency associated with entering sleep states makes it very hard to find the optimal moments when the core should put to sleep and be woken up. Intuitively, a low utilization should dissipate very little power but if the moments when the cores are put to sleep are predicted very poorly the system could end up dissipating more power. There is currently a trend in trying to shut off as much of the chip as quickly as possible to minimize these effects.

3.3.2 DPM in Linux

DPM is accessible in Linux via the CPU hotplug interface. CPU hotplugging was originally designed to replace a CPU in a multi-socketed system during runtime and enable components to be serviced and replaced without shutting off the entire system. The same functionality has later been adapted for power saving purposes, since a shut down core does not dissipate static power consumption.

The effectiveness of hotplugging is hardware dependent, however in the ideal case the power to the core is completely cut of and the core is removed from the reach of the systems scheduler to insure that no work is placed on the CPU while it is not active. In some cases the core is only removed from the schedulers reach without actually cutting off its supply power but only placing the core in a state of idle looping. This will result in having unusable core still dissipating power. On some platforms this functionality is not available at all. On our test platform the hotplug functionality will place the core in a *Wait For Interrupt*(wfi) state in which the core clock is shut down, and re-activated as soon as the core receives an interrupt from another core.

Even though this mechanism was not originally intended as means for saving power it has been adapted as a power saving feature in a number of multi-core embedded platforms such as mobile phones and tablets.

Hotplugging a core has different effects depending on the state of the core designated to be turned off. In case the core is idle, turning it off will decrease the total power consumption by the amount of static power the core dissipated. When turning off a loaded core, the load dedicated to this core must be first re-allocate to another core in order to make it idle. This scenario is illustrated in Figure 12. The figure depicts a quad-core system utilizing the default, fair, load distribution policy of the Completely Fair Scheduler (CFS). The figure illustrates

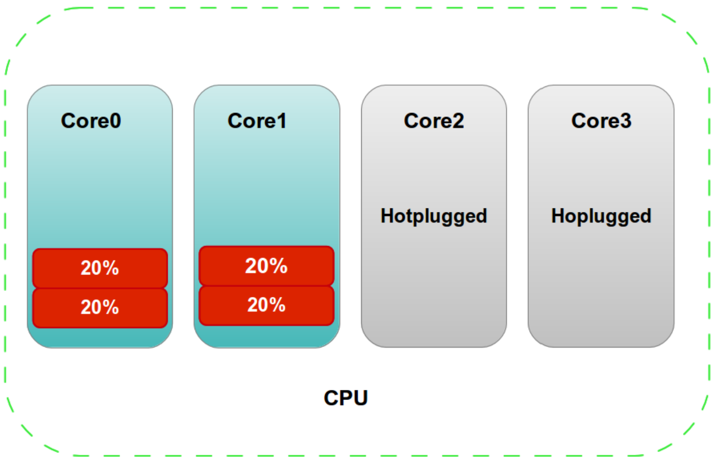


Figure 12: Load distribution during hotplug events

two cores shut down i.e. hotplugged. This forces the load to be redistributed over the remaining online cores.

Since hotplugging loaded cores includes the load transfer to other cores, it is fair to expect a higher latency when hotplugging a higher loaded core than a idle core. Hotplugging a CPU also introduces a larger latency than DVFS because more complex and time consuming functionalities must be executed when using hotplug, such as flushing the cache and transfer of CPU states.

Hotplug governors Experimental frequency governors including hotplug capabilities has been created. The *Hotplug* and *HotplugX* governors utilize similar characteristics as the ondemand governor, with the extended functionality of CPU hotplug. The hotplug governor will disable idle CPU cores as a significantly long period has elapsed and similarly switch a core back online as a significantly long busy period has elapsed. The user can set these periods explicitly in the `sysfs` interface.

While the functionality exists, there are no thorough investigation on "how many" and "how fast" cores should run in order to achieve the most energy efficient execution. The rules for switching on and off a core have simply been selected based on specific use-case applications and the programmer's *good feeling*. A general combined mechanism for DVFS and DPM is thus lacking in current multi-core systems.

We will hence investigate how to measure resource utilization and – if needed – how to extend the current OS view to enable facilities for energy efficient programming and scheduling.

4 Load

When considering load balancing on the Linux kernel, load estimation is an essential part. Ideally the exact contribution of one task to the CPU load would be known as well as the exact amount of resources the task would need and for how long. However, since this is seldom feasible, a best effort approximation is often utilized. In Linux there has through the history been a number of ways to measure the load, with varying complexity. Also, depending on the subsystem utilizing a load metric in order to perform its function, the manner in which load is calculated and perceived varies. The definition of load is also dependent on resource isolation i.e. what resource is being monitored. For example the CPU utilization might be extremely high whilst the load on the disk or main memory is minimal. In our research we have however focused on the behaviour of the CPU unless stated otherwise.

Regardless, deriving a good load metric is extremely important when utilizing load balancing. A good load metric should be able to:

- Correctly show the current load on a system
- Function as starting point for predicting future load behaviour
- Remain relatively stable and not be too sensitive for minor load bursts.

A suitable load metric could give an opportunity to optimize scheduling decisions as well as DVFS and DPM in order to minimize power and energy consumption.

4.1 Notions of load

Load measurements in Linux are used in many parts of the system and come in different forms. We will in the following sections present different ways of currently measuring load.

4.1.1 Utilization

Probably the most basic notion of load is the utilization of a task on a processor, and is defined as:

$$U = \frac{C}{T} \tag{6}$$

where U is the utilization caused by the task, C is the execution-time for the task and T is the period of the task. In practice all tasks do not have a period, or sometimes it is impossible or infeasible to define it. However, there are other methods of measuring load which are built on the notion of utilization.

4.1.2 Load window

The most common metrics can be found in different load monitoring systems such as the *System Monitor* and *htop*, among others. These utilities visualize load as a percentage value derived by tracing the busy vs. idle ratio over a certain time window. The load window method of measuring load is built from the utilization notion, where the period T is set to a fixed interval, and the execution-time C is the execution time of all tasks running on a core during the current interval. This method is illustrated in Figure 13. In multi-core systems, this can be calculated on a per-core basis, i.e. one load level per core.

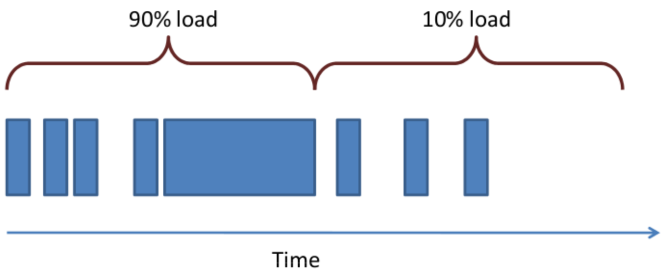
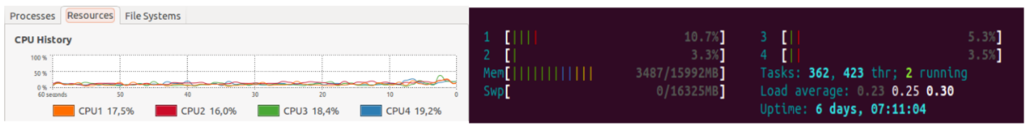


Figure 13: The load is calculated by measuring the busy vs. idle ratio over a time window

An example of how the Linux *system monitor* and *htop* calculates and visualizes load as can be seen in Figure 14 where the per core load is seen in the *system monitor* to the left and *htop* to the right.



```
1  [||||                    10.7%]    3  [||                      5.3%]
2  [|                        3.3%]    4  [||                      3.5%]
Mem[|||||||||||||     3487/15992MB]    Tasks: 362, 423 thr; 2 running
Swp[                    0/16325MB]    Load average: 0.23 0.25 0.30
                                      Uptime: 6 days, 07:11:04
```

Figure 14: Load calculated as a busy vs. idle time ratio presented in *system monitor* and *htop*

This way of perceiving load, as a busy vs. idle ratio expressed in percentages, is not only used to ease the visualization of workload to a human user, it is also used as the main decision metric in various power management mechanisms such as the Linux DVFS governors. A more detailed analysis of the governors and their use of load metrics is presented in Section 3 *Sleep States and DVFS*.

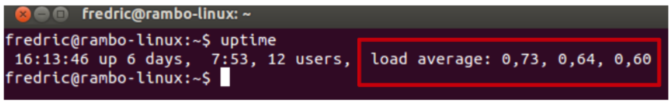
4.1.3 Run-queue length

A more instantaneous version of the load averages values is the run-queue (RQ) length i.e. the number of runnable tasks on the system. In a multi-core systems there exists one RQ per core. This gives an instantaneous numeric value of how many tasks each core has been allotted.

This value is constantly undergoing change due to tasks finishing their execution and due to the influences of the system scheduler, meaning it does not provide the same stability as the load average values. This means that although it is an important metric from the point of view of the system itself, and is used in among other things different scheduling decision mechanisms, it does not provide very much intuitive information to a human user.

4.1.4 Load average

Another load metric in Linux can be found in *top*, *htop*, based on the *uptime* shell command and is derived from the load average [36]. An example output of the *uptime* command can be seen in Figure 15 where the load average is highlighted in red.



```
fredric@rambo-linux: ~
fredric@rambo-linux:~$ uptime
 16:13:46 up 6 days,  7:53, 12 users,  load average: 0,73, 0,64, 0,60
fredric@rambo-linux:~$ 
```

Figure 15: Output of *uptime* showing the *load average* value highlighted in red

These three values show the load average for the past 1, 5 and 15 minutes respectively, whereas the actual load value is not based on CPU utilization but rather on an exponentially weighted moving average of the number of tasks on the system over the three different time periods.

Another perspective of the load average values is in the context of a multi-core machine. For example a quad-core machine with a one minute load average of 2.0 could be seen as the system being loaded to 50% over the one minute time period since the CPU has four cores to spread the load over. During this one minute period, the amount of work done is equivalent to two tasks running continuously. In case the load average would be 5.0 on a four core system, the system is considered as overloaded, since it has the equivalent work of 5 continuously running tasks, which is more than a four core system can handle simultaneously. The other two values of the triplet show the same information over a longer time period in order to provide more stable values.

The behaviour of the load average triplet is shown in Figure 16. The figure was derived by sampling the load average once every 5 seconds over the period of one hour. To induce load, the CPU stress test *stress* was used to execute four threads for 30 minutes after which the stress test was shut down. The machine on which this visualization was performed was the Intel-i7 platform with Ubuntu 13.10 presented in section 3.

From the figure, the behaviour of the load average and the difference in response time of the three values can clearly be seen. The one minute value reacts fairly fast to the *stress* induced load and reaches a value around 4 due to the four stress threads. Due to background tasks, the one minute value fluctuates and

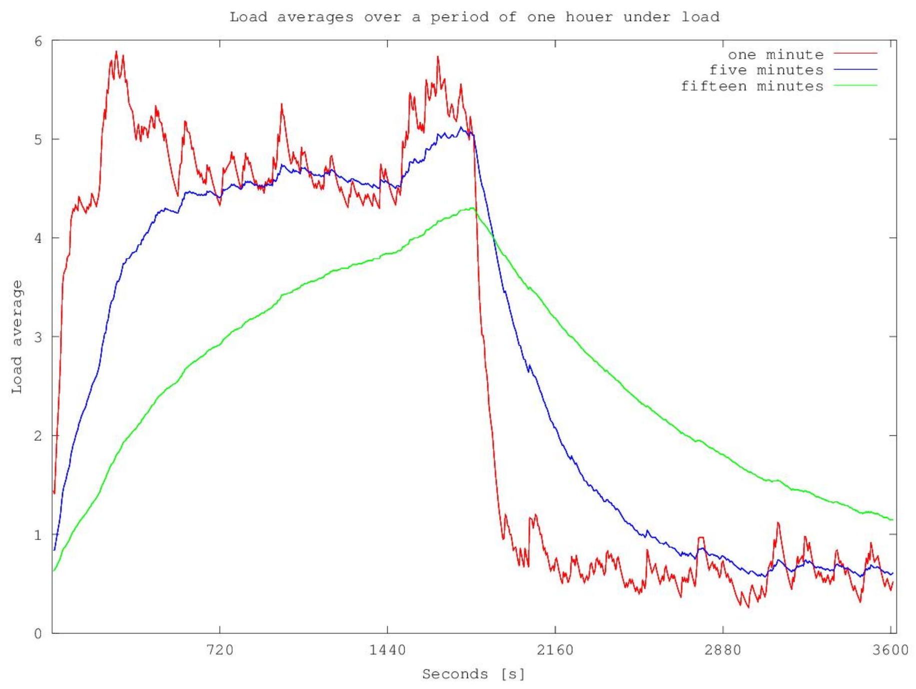
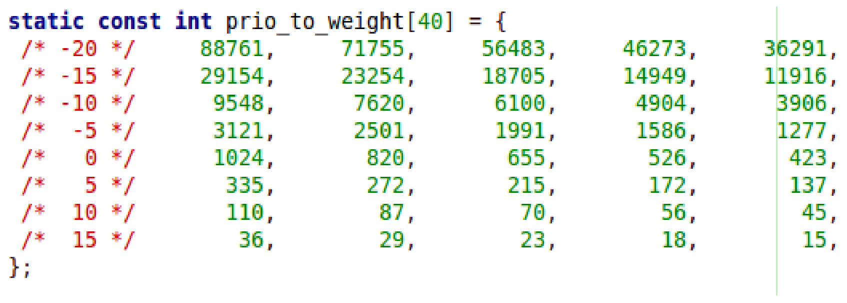


Figure 16: The three load average values over one hour under load

reaches a peak value of roughly 6. When the *stress* test stops, the value decreases relatively fast. The 5 minute value increase more slowly and remains more stable around the 4.5 mark until it to peaks due to background load and finally starts do decrease when *stress* is terminated. The fifteen minute value increases very slowly and is mostly unaffected by any other tasks than the ones induced by *stress*.

4.1.5 Load weight

Another load value used in Linux, and more precisely, in the normal scheduling policy of the system scheduler is the *load weight*. The load weight is a per task priority weighted load value which the system uses in order to schedule tasks i.e. determine which task to run and for how long. In multi-core systems it is used to distribute tasks over the set of available cores. The load weight is based on a task's priority, which in turn is calculated from the tasks `nice` value. The nice value of a task ranges between -20 and 19 where a lower value indicates a higher priority. If a task has a high nice value it can be seen to be "nice" to its fellow tasks an vice versa. The nice value are recalculated into priorities which range from 0 to 139 where the priorities from 100 to 139 are reserved for normal tasks and the priorities ranging from 0 to 99 are reserved for real-time tasks. This done in order to ensure that real-time tasks always receive higher priorities. The Linux kernel provides a set of macros to calculate priorities from nice values. When a task's priority is established the task's load weight can be calculated. This is done through a priority-to-weight conversion table located in the scheduler source files of the kernel. The table is shown in Figure 17.



```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Figure 17: Priority to weight conversion table

On the core level the load weights are used to decide which task to run and for how long. The table in Figure 17 contains one entry for each of the 40 nice levels, and in extension priority, with a multiplier of 1.25 between each entry. These specific values coupled with how they translate into the allotted run time for each task ensures that for each nice level the task will be allotted 10% more, respectively less, CPU time. This allows the load weights to effectively control the amount of time each task is allowed. The load weights are also used to calculate a per task timing variable called virtual runtime. The per task virtual runtime is essentially a timing variable of how long the task has been able to run weighted with the task's load weight. By utilizing the virtual runtime instead of real runtime, the scheduler can increase or decrease the time for an individual task in accordance to their load weight. In practice this means that important tasks, tasks with a high load weight can, from the scheduler point of view, have run for a shorter time period than they actually have. This allows them to run for longer periods of time uninterrupted.

4.2 Load perception in the Linux SMP kernel

On a higher level, looking at a multi-core system, the load weights are also used to decide on which core to place new and newly awakened tasks as well as during load balancing. This is realised by a per RQ core load weight which is the sum of all the task load weights currently on that particular RQ or core. In a default Linux distribution the the Completely Fair Scheduler (CFS) tries to distribute load as evenly as possible over the set of available cores. In order to place an initial task the scheduler scans each RQ, locates the RQ with the lowest total load weight and places the new task on that RQ. Over time this behaviour ensures an even load distribution. However, since tasks are constantly finishing their execution there will become imbalances in the load distribution and this is where the load balancing mechanism comes in.

The load balancing functionality is called periodically on the level of each individual core. The core currently calling the load balancing functionality searches for a load imbalance amongst the set of cores, if found, the imbalance is calculated. The core that invoked load balancing, proceeds to find the busiest core and starts to pull tasks from it onto itself until the imbalance is nullified.

Even though the utilization of load weights has been proven to function as a load metric in Linux load distribution mechanisms, the notion of priority based load values might bring forth some complications. One such is the lack of awareness. Since the load weights are essentially a value of how much a particular tasks wants or needs the system resources, it does not, directly, show how much resources a task is actually using. In extension by looking at the per RQ load weight it is difficult to tell how much actual work the RQs corresponding core is actually doing. Due to the load weight metric being priority based, important tasks will be seen as loading the system more than unimportant tasks regardless of how much resources they actually need. On a multi-core system this means, from the scheduler point of view, few high priority tasks might load a core as much as multiple low priority tasks.

In the default CFS scheduler the load weights work well since the scheduler only needs to quantify the load of one core relative to the other since its main purpose it to keep each core as evenly loaded as possible. However with different scheduling goals, for example maximizing the system energy efficiency instead of fairly distributing the tasks, the notion of load need to be extended as explained in Chapter .

4.3 Load and DVFS

Usually when load is measured, DVFS is not accounted for. Dynamic clock frequency governors in Linux are used to set the appropriate clock frequency of the CPU according to the current workload. This is usually done by setting a load limit, after which the core is considered overloaded and the frequency is increased. The limit differs between systems, but is usually in the range [60% 95%]. Since the workload is determined by measuring the level of CPU utilization over a certain time window, the load percentage will alter according to the current clock frequency. For example, a task utilizing 80% of CPU resources at 400 MHz will (in the theoretical case) utilize 40% when running at 800 MHz.

Consider for example the situation illustrated in Figure 18. When spreading the workload on all four cores, all cores are loaded to 20% and the clock frequency will never increase as no cores are considered overloaded. When executing n-operations per time window, the system will execute $4n$ operations in total over the selected time window. By instead placing all the workload on one CPU (right hand side in Figure 18), the load threshold is more likely reached and the clock frequency is increased (by 4x in this example). This means that each task containing 20% load will execute $4n$-operations in the same time as in the previous case and $16n$-operations in total for the same time window.

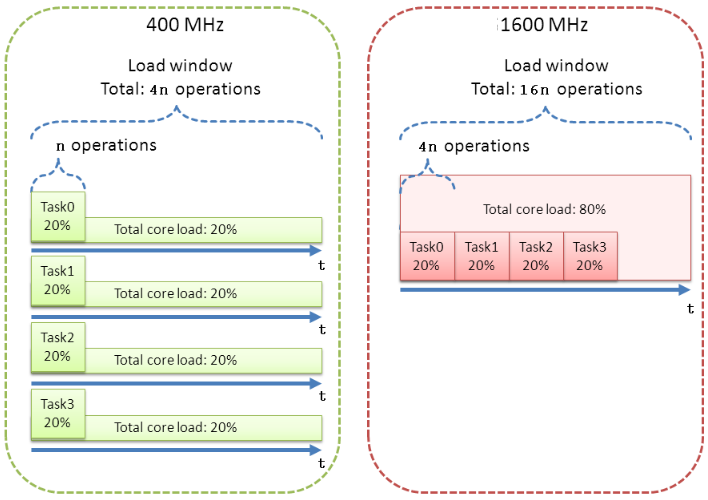


Figure 18: Executing on high clock frequency on as few cores as possible increases performance for a given workload

4.4 Discussion

The concept of CPU load has been used to scale the CPU capabilities more closely to the software requirement to save power. However, the notion of load (given as utilization percentage, run-queue length or an average over time) does not describe the intensions of the software, but only the resource usage. To facilitate more energy efficient software, *performance* of the software should complement the notion of load to more precisely express the software capabilities.

5 Extending the Notion of Load with QoS

By extending the notion of load with application performance, the resource management is able to allocate only the necessary amount of resources to the applications. To enable this, we extend the applications with additional meta-data containing the performance and performance requirements.

5.1 Units of performance

The performance of an application can be measured and compared to the originally stated requirements to give a value on how well the application is performing; also know as Quality-of-Service (QoS). QoS is often introduced in soft real-time systems [25] in which the deadlines for tasks are not hard. This means that deadlines are allowed to be slightly missed as long as the result (quality) keeps a sufficient level. QoS, also a term used in cloud computing [38], is used for *selling* a bundle of processing power to the user with a certain quality. By declaring a QoS limit for an application, it is allowed to relax the performance guarantees with a certain percentage and still be inside the specifications for the application.

In order to create an energy efficient system, the tasks should:

1. Execute on the appropriate execution unit

2. Only allocate the necessary amount of resources, and therefore minimize the energy consumption

For this, the notion of performance is an important measurement of how well a task is able to satisfy the user. Since performance, with this definition, is a subjective matter, the different ways of measuring and controlling performance could be many.

The notion of QoS is used to interpret performance measurements from any application, and by this information strive to satisfy all application requirements. Tasks define – in our model – a performance metric to include in the execution. The value of this metric is periodically measured, and the system ensures that the task is given the necessary resources for upholding sufficient performance. With this approach applications are given only the necessary amount of resources but not more, since it would be considered waste.

The value of QoS for an application is determined by comparing the requirement in the specification with the actual measured performance. The ratio between these two values is the *QoS drop*. If the QoS drop is more than allowed by the specification, the system must control some actors giving the application more resources and thus higher QoS. For example a web server can have a specified requirement of serving 500 `requests/s` with a QoS of 90%, which means that it will consider anything between 450 and 500 as acceptable according to the specification. Similarly a video transcoder can require 25 `fps` with a QoS limit down to 23 `fps`. We suggest the new single entity *QoS* as the measurement of performance. This means that applications can specify performance using any metric, and request the allocation of resources according to this metric.

5.2 QoS aware execution strategy

In our work we focus on applications in which **1)** QoS requirements can be defined and **2)** performance can be measured. An example is a video player illustrated in Figure 19, which processes and displays a video for a set amount of time. From this application we demand a steady playback (e.g. 25 frames per second) for the whole execution, but the execution speed of the internal mechanisms such as the decoder is completely dependent on the hardware resource allocation.

The popular (and easily implementable) execution strategy called *race-to-idle* [28] was implemented to execute a task as fast as possible, after which the processor enters a sleep state (if no other tasks are available). The *ondemand* (OD)

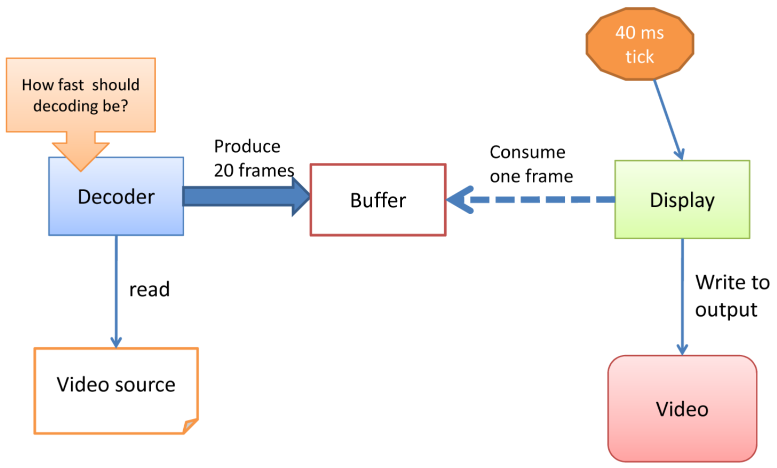


Figure 19: Overall structure of a video player with QoS requirements on the decoder

frequency governor in Linux supports this strategy by increasing the clock frequency of the CPU as long as the workload is above an `upthreshold` limit. Race-to-idle minimizes the execution time t, but on the other hand results in high power dissipation P during the execution. A strategy such as race-to-idle will have a negative impact on energy efficiency if the decrease in time is less than the increase in power i.e. $\Delta^{-}t < \Delta^{+}P$ compared to running on a lower clock frequency. Depending on the CPU architecture and the manufacturing technology this relation varies, but with current clock frequency levels, is it usually very energy inefficient to execute on high clock frequencies [39, 27]. It is also inefficient to execute on very low clock frequencies [10] since the execution time becomes large and the static power is dissipated during the whole execution.

Our strategy is to *execute as slow as possible while still not missing a given deadline*; we call it *QP-Aware*. Figure 20 Illustrates two different execution strategies for a video player: Part A) illustrates the race-to-idle strategy in which the decoder is executed as fast as possible for a short time, after which it idles for the rest of the video frame window. Part B) illustrates the QP-Aware strategy in which the decoder executes as slowly as possible while still keeping the frame deadline of the playback. If the execution time in case A) is twice as fast but the power dissipation is more than twice as high, case B) will be more energy efficient. Moreover, frequently switching the frequency and voltage introduces some additional lag, which also impacts on the energy consumption. We argue for a type B-kind of execution, in which the application executes on more energy efficient frequency

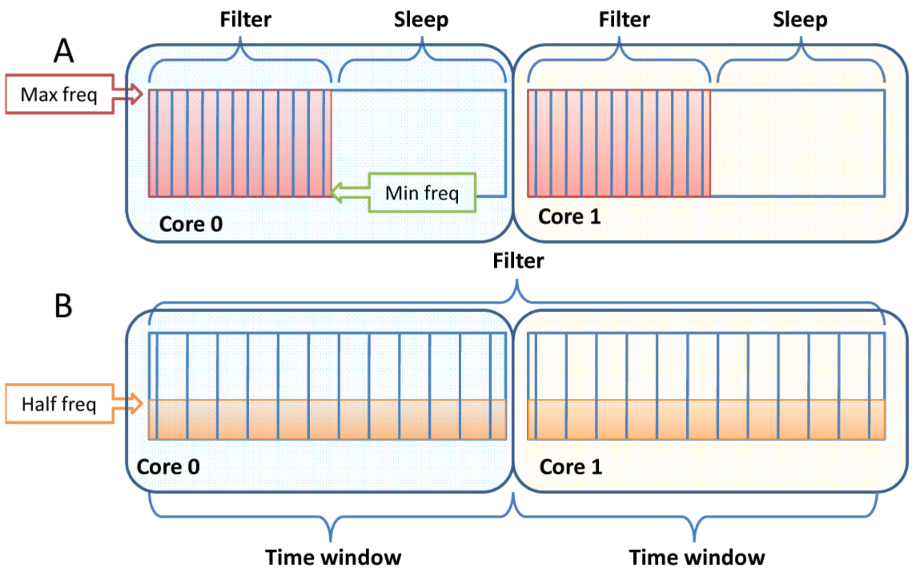


Figure 20: Two execution strategies: A) Race-to-idle B) QP-Aware

5.3 Conclusions

With the extended notion of resource requirements in the applications, we are able to drive the hardware resource allocation more closely to the software requirements. In order to realize this energy efficient resource allocation, we need to create a system able to intercept the added meta-data and accordingly control the hardware. The following chapter introduce a QoS aware runtime power optimizer exploiting DPM and DVFS mechanisms.

6 Power Optimization with DPM and DVFS

Current power managers, such as the frequency governors in Linux, base the resource allocation purely on system workload levels. Resources are allocated/deallocated as the workload reaches a certain *threshold*, which is usually done on system level rather than on core level. This means that the power management has no information of the program behavior such as its parallelism, nor any notion of how the workload should be mapped on the processing elements. With our extended notion of QoS we can set-up the resource allocation according to the performance requirements and which resource allocation technique to use.

We chose an optimization-based approach in which we aim to minimize the power while keeping the performance requirements stated in the applications as follows:

$$\begin{aligned} &\text{Minimize}\{\text{Power}(q,c)\}\text{Subject to:} \\ &\qquad \forall n \in \text{Applications}: E_n - (q+c) < S_n - Q_n \end{aligned} \tag{7}$$

where the variables: q and c are the actuators (DVFS and DPM). S_n is the setpoint, E_n is the error value and Q_n [1] is the lower QoS limit. The optimization rule states to minimize power while eliminating enough errors to reach at least the lower bound QoS limit. This is achieved by setting the actuators (q, c) to a level sufficiently high for each application n.

By optimizing the balance between DVFS and DPM, we are able to find the combination of dynamic and static power needed for a sufficient QoS level in the applications. To solve the optimization problem, a description of the system is needed which models the performance vs. power response to resource allocation with both DVFS and DPM.

6.1 Performance Modeling

We have chosen an approach similar to the work in [34], in which a performance model was used to describe the speed-up of DPM/DVFS regulation i.e. the speed-up for activating cores vs. scaling the frequency. By combining the speed-up with the power cost of increasing either DVFS or DPM in one direction, we can calculate the most energy efficient combination, and with existing optimization methods derive the *energy efficient path* from low-end to high-end performance as illustrated in Figure 21

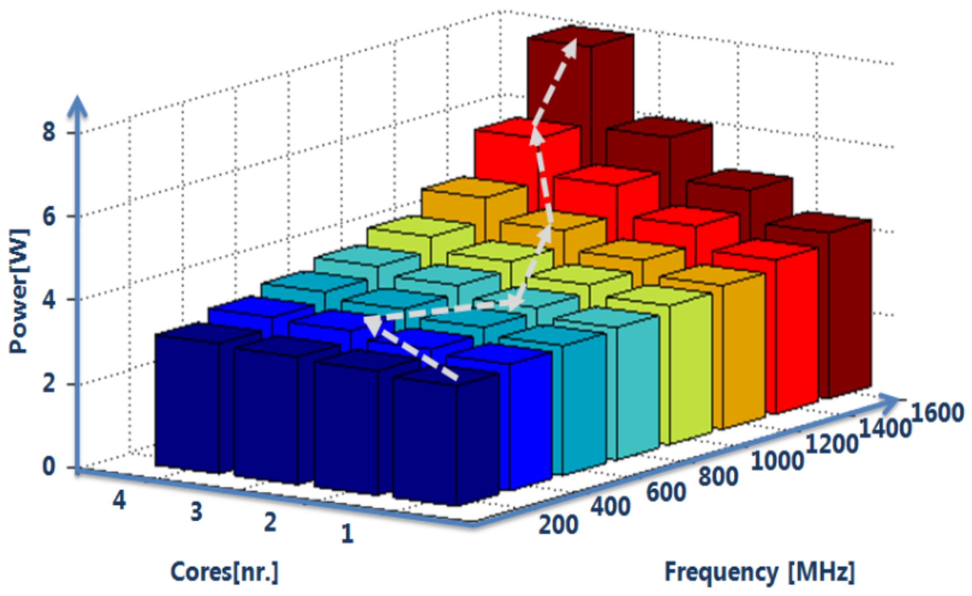


Figure 21: An example of the most energy efficient path from low-end to high-end performance for a selected application

In order to determine whether to use DVFS or DPM, the optimizer requires knowledge on how much it *affects* the applications. For example a sequential pro-

[1] E and Q are normalized to the range in which q and c operate

gram would not gain any performance by increasing the #cores, while a parallel application might save more energy by increasing the #cores instead of increasing the clock frequency. We modeled DVFS performance as a linear combination of clock frequency q as:

$$\text{Perf}(\text{App}_n, q) = K_q \cdot q \tag{8}$$

In contrast to the rather easy relation between performance and clock frequency, modeling the performance as a function of #cores is more difficult since the result depends highly on the inherited parallelism and scalability in the program. To assist the optimizer, we added the notion of expressing parallelism in the applications. The programmer is allowed to enter the parallelism of a program in the range [0, 1] where 0 is a completely sequential program and 1 is an ideal parallel program. This value can either be static or change dynamically according to program phases. In case the exact number is not known, the programmer can approximate a value to assist the optimization algorithm for finding at least a nearly optimal result.

Our example model for DPM performance uses Amdahl's law

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{9}$$

where P is the parallel proportion of the application and N is the number of processing units. The final performance model for DPM is rewritten as:

$$\text{Perf}(\text{App}_n, c) = K_c \cdot \frac{1}{(1-P) + \frac{P}{c}} \tag{10}$$

where K_c is a constant and c is the number of cores. This models a higher performance increase as long as the #cores is low but decreases as the #cores increase.

6.2 Power Model

By creating a mathematical model of the system as function of resource usage, it is possible to integrate already existing optimization tools for allocating the appropriate amount of resources for the given workload. Especially in multi-core systems, the relation between performance, temperature and power highly depends on the resource allocation both in time and space dimensions.

6.2.1 Bottom-Up Model

The literature shows works such as [6, 21, 14, 17] in which multi-core power models are created bottom-up from an analytical expression combining static and dynamic power based on the clock frequency and the number of active cores. The dynamic power is usually modeled as P_d^α where α is a constant usually in range

[2 3] to model the non-linearity in dynamic power as function of frequency and voltage. In its simplest form, static power is modeled as a constant P_s[6], which gives the total power dissipation: $P = P_d^{\alpha} + P_s$.

As long as the expression is used to model a single-core system, model can be tuned rather accurately to the real-world system. We measured the total power dissipation of a single-core system in each frequency step when loaded to 100% and compared the results to a curve fitted matlab model based on the previous expression. The result is shown in Figure 22, in which we clearly see the close relation between the model and the data. The datapoints were obtained by mea-

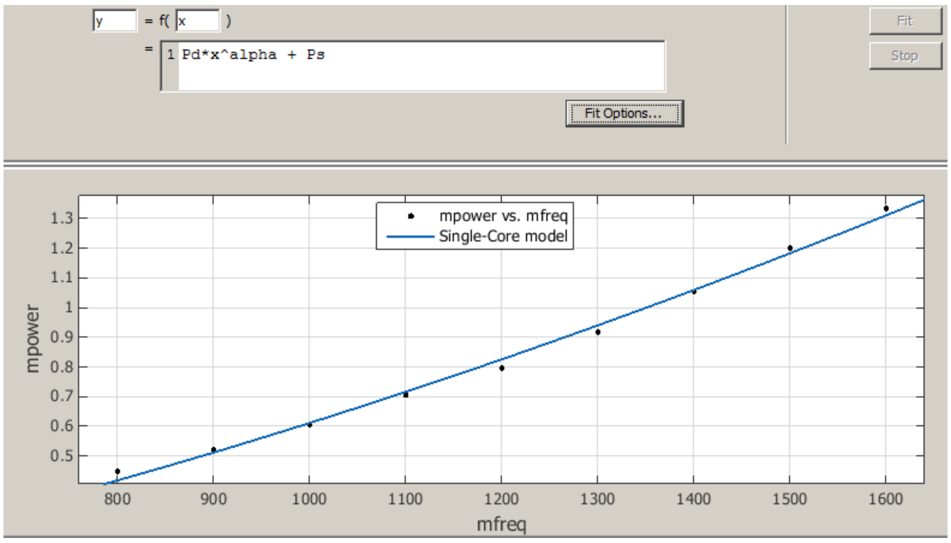


Figure 22: Curve fitted model compared to real data for a single-core system

suring the power of the new Exynos 5 chip with one Cortex-a15 core running on max load from 800 MHz to 1800MHz.

However, when expanding the model to a multi-core system the model must account for the increased static power when activating cores and the dynamic power of scaling the frequency. The authors in [6] suggest to simply add a multiplier N to the static/dynamic power where N is the number of cores. A simplified expression of this would be $P = P_d^{\alpha} \cdot N + P_s \cdot N$. This expression assumes that the power dissipation is linearly related to the number of cores activated. We modeled this expression in matlab and compared it to the real data measured from the same Exynos 5 chip as previously mentioned.

The results are shown in Figure 23. As seen in the figure, the model agrees with the data only for the N-value used as basis for the curve fit. When expanding the model only by multiplying the number of cores the model, the data and model starts to differ and the multi-core model – for this type of chip – is no longer valid. One reason for this is shared resources in the chip which are not replicated when activating more cores, another reason is the thermal influence which influences the static power significantly [10] and increases the non-linearity of the power

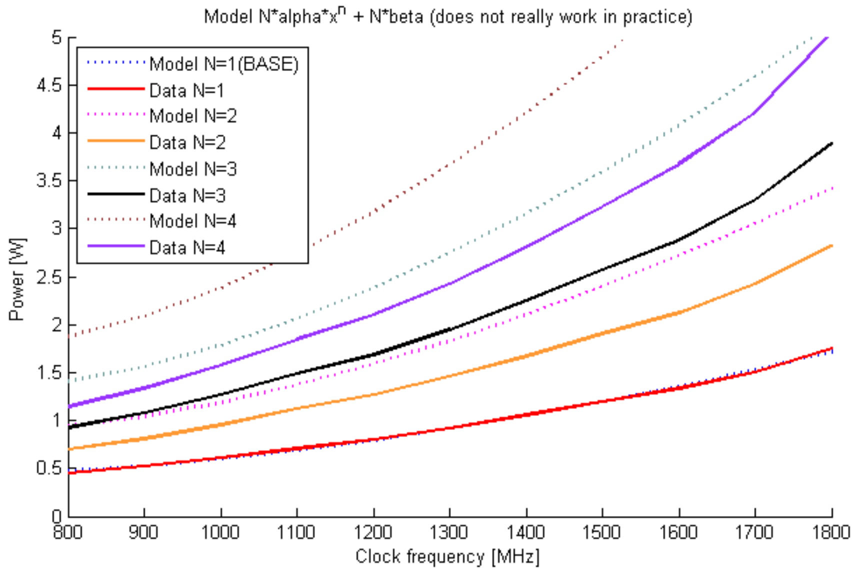


Figure 23: Curve fitted model compared to real data for a multi-core system

dissipation.

A first step to increase the accuracy could be to focus more closely on the static power such as in [9]. The authors model the static power as a temperature dependent Taylor series expansion as $P_l = \alpha_1(t + t_{ref}) + \alpha_2(t + t_{ref})^2$ where α_i are hardware dependent constants, t is the temperature of the chip and t_{ref} is a reference temperature. The problem becomes hence to either measure or model the temperature in order to have a t value in the expression and secondly the ambient temperature will further influence the leakage, which is not taken into account in the model.

6.2.2 Top-Down model

Instead of building the power model from bottom, we have chosen to derive it directly Top-Down from an existing platform real-world conditions, and from real workload running on the CPU. In contrast to the previously explained approach, we measured the power on real hardware after which we find an analytical expression as close as possible to the real data.

We trained both power models (in this example of an Exynos 4 chip) by increasing the frequency and # active cores (nr. of cores) step-wise while fully loading the system. As workload we ran the `stress` benchmark under Linux on four threads during all tests, which stressed all active cores on the CPU to their maximum performance. Moreover, we trained two different models for different ambient temperature conditions: Hot (in room temperature +20 degC) and Cold (in freezer -20 degC) to account for operations in different ambient temperatures. The dissipated power was measured for each point and is shown Figure 24.

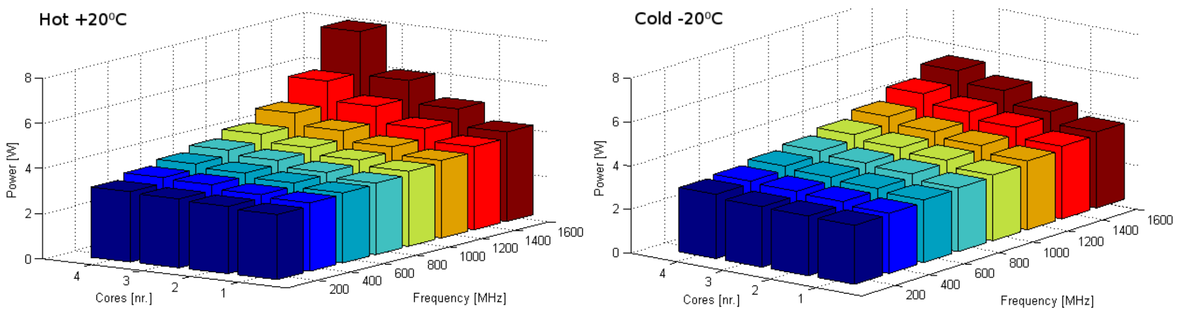


Figure 24: Power as function of #cores and clock frequency (fully loaded). Left: Hot case Right: Cold case

As seen in the figures, the power dissipation of the chip peaked much higher in hot ambient temperature, especially for the high clock frequencies and with many cores. The resulting behavior of the frequency-to-power relation is clearly not linear, especially for the hot case. We clearly see the effect of the static power P_s, which is caused by leakage currents in the transistors and is present as long as the core power source is enabled. The leakage currents increase as the temperature increases because of a higher voltage threshold in the transistors, which leads to higher total power dissipation. With these measurements we will derive a unified equation for approximating the chip power under full load as function of frequency and #cores.

We used a similar approach to [30], in which we fitted a two dimensional plane (q,c) as a function of the power dissipation. The third degree polynomial

$$P(q,c) = p_{00} + p_{10}q + p_{01}c + p_{20}q^2 + p_{11}qc + p_{30}q^3 + p_{21}q^2c \tag{11}$$

where p_{xx} are coefficients was used to define the surface. We used Levenberg-Marquardt's algorithm [15] for multi dimensional curve fitting to find the optimal coefficients, which minimizes the error between the model and the real data.

Table 2: Coefficients for power models

Hot						
p_{00}	p_{01}	p_{10}	p_{11}	p_{20}	p_{21}	p_{30}
2.34	0.0576	0.598	-0.0248	-0.1605	0.0097	0.0120
Cold						
p_{00}	p_{01}	p_{10}	p_{11}	p_{20}	p_{21}	p_{30}
2.29	0.0614	0.302	-0.0193	-0.0569	0.0056	0.0038

Table 2 shows the results for the hot and cold case and Figure 25 illustrates the surface of the hot case with the given parameters where DVFS and DPM utilization is given in the range [1,8] where 1 is minimum and 8 is maximum.

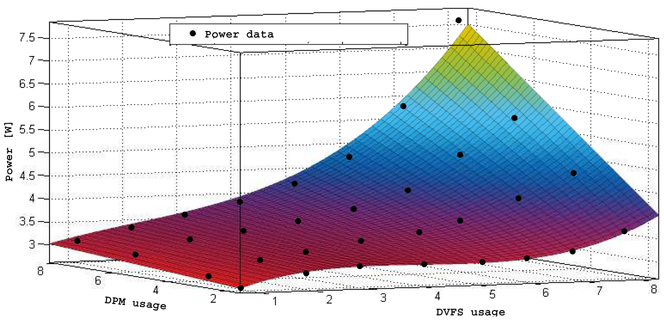


Figure 25: Surface of the hot use case derived from Equation 11. Dots are real data measurements

To verify our model we calculated the error difference between the real data and the derived model. The presented values in Table 3 and Figure 26 show a small average error for both cases. The hot case showed however a higher maximum error than the cold case because of a more difficult surface to fit with a third degree polynomial. With the rather small average difference, we considered these two models feasible for our experiments. The same procedure can be run

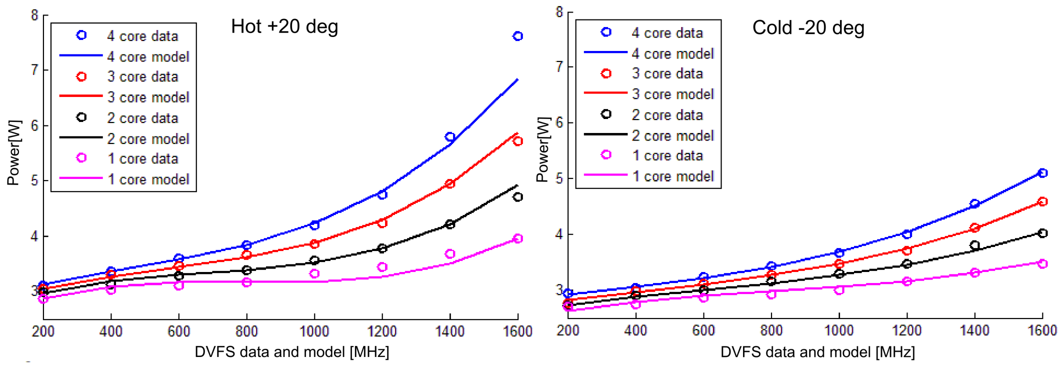


Figure 26: Verification of power model with real data (circles) and model (line). Left: hot case, Right: cold case

Table 3: Differences between real data and model for both hot and cold case

Hot	Max diff	Avg. diff	**Cold**	Max diff	Avg. diff
	10.2%	0.6%		2.4%	0.03%

for any homogeneous multi-core chip by choosing a training benchmark (such as `stress`) and by measuring the real power dissipation of the board.

6.3 Run-time aspects

We evaluated the system on an Exynos 4412 board, which is a quad-core implementation based on the ARM Cortex-A9 MPCore CPU. All cores except *Core0* can independently be placed in a sleep state by the DPM mechanism, which is controlled by software. The DVFS mechanism can scale the frequency of the CPU from 200 MHz to 1.6 GHz by increments of 100 MHz. We chose this chip for the evaluation because of its feature rich hardware and since it is currently one of the leading microprocessors used in mobile phones and tablets.

The power manager was mapped on top of the operating system and available as a middleware to the applications as illustrated in Figure 27. Applications connected to the power manager issue library calls for sending configuration and measurement parameters to the power manager such as performance and P-values. Applications can also freely use the operating system as normally, and the power manager can be bypassed completely if no performance requirements are needed in the application. The power manager controls DVFS and DPM by using kernel calls and the sysfs filesystem.

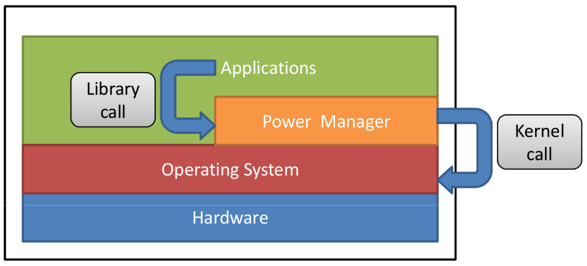


Figure 27: Placement of the power manager as a middleware on top of the OS

We implemented and mapped the power manager and its infrastructure on Linux (kernel version 3.7.0) with the SQP solver and communications backbone implemented in the c-language. The communication was established by using posix queues to push and pull data between the power manager and the applications. The power manager was implemented as a periodic task, which means that the optimization results and the actuator control is obtained with regular intervals. In contrast to the theoretical simulation environment, the real implementation introduces a certain lag time especially for switching on and off cores. The period of the power manager should hence be set long enough to not suffer from the hardware lag time, but also short enough to give an as fast as possible response.

To determine the period, we conducted experiments to measure the lag time of DVFS and DPM. The measurements were simply done by shutting down and waking up a core, and the time between the shut-down/wake-up call and the physical shut-down/wake-up of the core was measured for 100 iterations. Similarly we measured the lag time for switching between two different clock frequencies to determine the lag time for DVFS. The lag time showed to fluctuate depending on the clock frequency of the chip and the current workload of the chip. This means that as the chip was running on a low frequency and with high workload, the lag time was the largest. We chose to select an average scenario with a safety margin in which the chip was running on the lowest frequency and with a workload level of 90% (since the applications running on a chip with higher workload level would require more resources and the workload level would drop). The lag time of such a scenario is seen in Table 4.

Table 4: Expected lag time for DVFS and DPM

Shut-down	wake-up	Change frequency
15 ms	20 ms	5 ms

Based on the lagtime experiments, we chose a period of 40 ms for the power manager, which means that regulating the actuators will be completed before the next period is reached.

6.4 Conclusions

We have presented an approach to identify two model describing system power and performance. The performance model is completely generic and requires only the QoS meta-data added in the applications and the multi-core scalability of the applications. The power model has been tailored to a selected processor type, but can easily be re-engineered for any kind of processor.

We insert the models into a chosen optimization solver, which in turn controls the hardware actuators. After this setup, we evaluate the models by stressing the system with real workload.

7 Spurg-bench: The Multi-Core Energy Benchmark

To evaluate the system under different conditions of workload, a predictable and repeatable method of applying a specific workload on the system is needed. When developing a consolidating load balancer [10], there was an emerging need for a way of testing if the approach was both performance and energy efficient.

Looking at the commercial SPECPower benchmark [12], it is mainly designed to give comparative power/throughput measurements of the computing resources in a server system with low utilization [33].

A sensible approach is to have a load generator generating different kinds of loads which would then be spread across cores using different policies. Spurg-bench is designed with the goal to benchmark: 1) the scheduler and the load balancer in the operating system, and 2) the hardware, i.e. CPU cores, memory buses, memories etc. Most existing benchmark software only test the hardware [11, 37, 2, 8], i.e. they try run a computation as quickly as possible. However in order to benchmark the scheduler, load balancer and hardware, a load which would not run as quickly as possible but run multiple threads with a computation which sleeps some fraction of the time is instead needed. Therefore as existing benchmark software did not suite our purposes, the Spurg-bench load-generator [23] was designed and implemented.

Spurg-bench is open source and available at https://github.com/ESLab/spurg-bench.

7.1 Spurg-bench overview

The load generated by Spurg-bench consists of parallel threads, running and sleeping to achieve a certain utilization. A Spurg-bench run is defined by a number of operations, a number of threads and a load level. In practice this means that the number of threads is spawned when the run starts and the threads tries to maintain a certain sequential CPU-utilization. For example if we set Spurg-bench to generate a 50% load on a quad core system, Spurg-bench can spawn 4 threads which tries to maintain 50% load or 8 threads which tries to maintain 25% load. Spurg-bench tries to generate a load which minimizes the sleep time between system calls, and is thus maximally power inefficient with respect to sleep states, due to the argument in Section 3.3.1.

We emphasize here that we *maintain* a certain level of CPU-utilization, instead of just generate, because we assume that the performance of the threads execution environment is varying in a non-deterministic fashion. The non-determinism is due to complex behavior of the system as a whole. The clock frequency of the core the thread runs on can change or the thread can be migrated to a core with different clock frequency or micro-architecture. These effects could be decreased by locking the threads to cores, however, this is not wanted since the goal is to test the behavior of the scheduler together with the load balancer and threads should be able to migrate across the cores freely.

As there is no sufficient notification API for task migration or frequency changes in Linux these effects were estimated, with the drawback that an inaccurate estimation will yield an incorrect load level. The estimation is done with periodic run-time observations, using an adaptive algorithm. The estimation will change when the performance of the execution environment changes. There is a trade-off between a fast enough adaptation to quickly achieve accuracy on changes in the environment and stability of the estimator.

Spurg-bench has been used for a specific set of tests, but the design is built using an abstraction containing three types of components:

1. The runner script, controls the execution of the of the run.

2. The load generator, directs the control-flow between the operation and sleep.

3. The operation, generates the actual load.

The following sections will describe how these components interact and how they have been implemented for our purposes.

7.2 The runner script

When a run starts, the user gives a set of parameters to Spurg-bench, which are parsed by the runner script. The runner script then starts up threads with some parameters, and starts monitoring the reported amount of operations they are executing. When the total amount of operations are executed the threads are joined and the run is over.

Since the goal is to test schedulers and load balancing polices there is no fairness guarantees and different threads can execute a significantly varying amount of operations in a run. Because of this it is necessary to monitor the amount of operations executed by threads instead of simply divide the total amount of operations among the threads on startup. Moreover to keep the monitoring overhead low the amount of report messages is limited.

The runner script is the only component which has a user-interface. It is a python-script with the following command-line interface:

```
usage:  simple_run.py [-h] [-n N] [-a dont_set,set] [-l L] [-o O]
optional arguments:
-h, --help show this help message and exit
-n N Number of processes to start
-a dont_set,set Affinity setting
-l L The load to set on processes
-o O Total number of operations to perform before exiting
```

The user can initiate a benchmark by running for example `./simple_run.py -n 4 -o 1000000 -l 0.25`

This creates four threads with a total of 1M operations with a load of 25% for each thread.

7.3 The load generator

The load generator has two responsibilities, 1) to direct control flow between the operation and the idle state and 2) to estimate the instantaneous cycle count of the operation. It has three states, *operation*, *sleep* and *estimate*, and the transitions between these states are illustrated in Figure 28.

7.3.1 Pseudo algorithm

For describing how the load generators works, we begin with describing a high-level pseudo algorithm, which describes the general functionality of the load generator. The pseudo algorithm is in subsequent sections refined to match the actual implementation of the load generator.

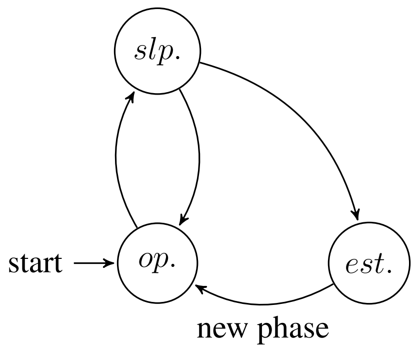


Figure 28: State machine of load generator. States: *operation* ($op.$), *sleep* ($slp.$), *estimate* ($est.$)

The load generator works in phases starting with phase 1 in state *operation*. On transition from *estimate* to *operation* the load-generator enters a new phase $i + 1$. Each phase has a set of parameters calculated in the *estimate* phase. In the beginning of phase i the load generator will transition between the *operation* and the *sleep* states $o(i)$ times, and then enter the *estimation* state. An abstract description of the load generator algorithm, derived from the state-machine, can be found in Algorithm 1. The difference between the state-machine representation in Figure 28 and the representation in Algorithm 1 is that time is added. From this algorithm we can get the utilization of the load generator with

$$U(i) = \frac{t_{pseudo}(i) - o(i) \cdot t_{sleep}(i)}{t_{pseudo}(i)},$$

derived from Equation 6 in the load section.

Algorithm 1 Load generator pseudo algorithm

```
$i \leftarrow 0$, $t_{sleep}(0) \leftarrow 0$, $n(0) \leftarrow 1$
**loop**
  $t_{pseudo}(i) \leftarrow$ time_now()
  **for** $j = 1$ to $o(i)$ **do**
    operation() // *operation* state.
    sleep($t_{sleep}(i)$) // *sleep* state.
  **end for**
  $t_{pseudo}(i) \leftarrow$ time_now()
  << calculate $t_{sleep}(i+1)$ and $o(i+1)$ >> // *estimation* state.
  $i \leftarrow i + 1$ // enter phase $i + 1$.
**end loop**
```

7.3.2 Load generator algorithm

A problem with the pseudo algorithm in Algorithm 1 is that in practice it is not possible to have a $t_{sleep}(i) < t_{sleep_{min}}$, for some $t_{sleep_{min}}$, therefore a further refinement step is is done and is shown in Algorithm 2[2]. Also, algorithmic support is needed for sleeping for as short intervals as possible to minimize the possibilities for the processor entering sleep states. Therefore we have added a double loop construct in the actual algorithm, such that $o(i) = n(i) \cdot m(i)$. For practical reasons both the wall-time $t_{wall}(i)$ and the cpu-time $t_{cpu}(i)$ is measured. This is reflected in Algorithm 2.

1. The inner loop, looping the operation $n(i)$ times has the purpose of increasing the needed sleep time. Since it is possible to set $n(i)$ before every phase it is possible to have a minimum sleep time longer than T_{run}.

2. The outer loop, executing the inner loop and the sleep statement $m(i)$ times, is used to control the time between measurements. For optimal control performance a constant measurement interval is needed. However, in practice there is a minimal $t_{sleep}(i)$ which can be reliably achieved, therefore $m(i)$ is adjusted to achieve $T_{sleep}(i)$ as close to $t_{sleep}(i)$ as possible.

7.3.3 Estimation of utilization

For the algorithm to be of any use it is required that the utilization ca be calculated. In practice, it is possible to calculate the utilization from the measurements performed in Algorithm 2.

For each phase of a Spurg-bench we will have $M(i) = \{1, \ldots, m(i)\}$ and $N(i) = \{1, \ldots, n(i)\}$. The CPU-time consumed during phase i is thus

$$T_{cpu}(i) = \sum_{k,l \in M(i) \times N(i)} T_{run}(i)_{(k,l)}. \tag{12}$$

As can be seen from Algorithm 2, $T_{cpu}(i)$ is measured. This measurement is enough to estimate $E\left[T_{run}(i)_{(k,l)}\right]$. The expected run-time is well defined as long as the execution environment doesn't change, under the assumption that small fluctuations in $T_{run}(i)_{(k,l)}$ is due to random interference and is considered as variance. If we assume

$$\forall k, l \in M(i) \times N(i) : T_{cpu}(i) = m(i) \cdot n(i) \cdot E\left[T_{run}(i)_{(k,l)}\right], \tag{13}$$

we can derive an estimator for $T_{run}(i)$ from Equation 12 as

$$\hat{T}_{run}(i) = \frac{T_{cpu}(i)}{m(i) \cdot n(i)}. \tag{14}$$

[2]For the sake of correctness the refined algorithm is based on the refined state-machine in Figure 28.

1. Note: The assumption in Equation 13 does not hold when the performance of the execution environment changes, such as when the clock frequency changes, any cache is flushed or when the thread is migrated to a core with different performance.

To calculate the utilization for phase i the wall-time for each phase is measured. The wall-time can be modeled with the following equation:

$$T_{wall}(i) = \sum_{k \in M(i)} T_{sleep}(i)_k + \sum_{k,l \in M(i) \times N(i)} T_{run}(i)_{(k,l)} \tag{15}$$

If we assume Equation 13,

$$\forall k \in M(i) : E\left[T_{sleep}(i)_k\right] = t_{sleep}(i) \text{ and}$$
$$\forall k,l \in M(i) \times N(i) : E\left[T_{run}(i)_{(k,l)}\right] = \hat{T}_{run}(i),$$

we can now make another estimator for $\hat{T}_{run}()$:

$$\hat{T}_{run}(i) = \frac{\hat{T}_{wall}(i) - m(i) \cdot t_{sleep}(i)}{m(i) \cdot n(i)} \tag{16}$$

and from the estimators calculate the utilization using Equation 6 as

$$\begin{aligned} \hat{U}(i) &= \frac{\hat{T}_{run}(i)}{\hat{T}_{wall}(i)} \\ &= \frac{\hat{T}_{run}(i)}{m(i) \cdot t_{sleep}(i) + m(i) \cdot n(i) \cdot \hat{T}_{run}(i)} \\ &= \frac{\frac{T_{cpu}(i)}{m(i) \cdot n(i)}}{m(i) \cdot t_{sleep}(i) + T_{cpu}(i)} \\ &= \frac{T_{cpu}(i)}{n(i) \cdot (m(i)^2 \cdot t_{sleep}(i) + m(i) \cdot T_{cpu}(i))} \end{aligned} \tag{17}$$

7.3.4 Computation of parameters for phase $i + 1$

In the previous section the utilization from the measurements in the algorithm was calculated. Parameters for a phase $i + 1$ are obtainable from a state and the measurements done in phase i. The parameters $m(i+1)$, $n(i+1)$ and $t_{sleep}(i+1)$ need to be calculated. There are two separate cases that need to be considered.

1. If $t_{sleep}(i + 1) = 0$, we can set $m(n + 1) = 1$ and set $n(n + 1)$ according to $\hat{T}_{run}(i)$ to achieve some measurement period. In this case the algorithm simply runs the operation without interruption.

2. If $t_{sleep}(i+1) > 0$ we will have the invariant $t_{sleep}(i+1) > t_{sleep_{min}}$, as well as $m(i + 1), n(i + 1) \in \mathbb{Z}_+$. In this case the operation is run a number of time, and then sleep for a given amount of time. It is now easy to calculate a minimal $t_{sleep}(i + 1)$ with these properties.

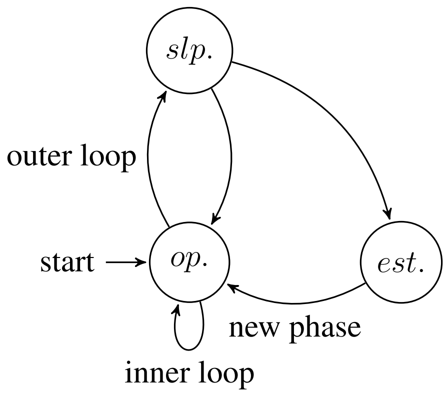


Figure 29: Refined version with double loop of the state machine (from Figure 28) for the load generator, according to Algorithm 2. States: *operation* ($op.$), *sleep* ($slp.$), *estimate* ($est.$)

Algorithm 2 Load generator structure

```
$i \leftarrow 0$, $t_{sleep}(0) \leftarrow 0$, $n(0) \leftarrow 1$
**loop**
  $t_{wall_1}(i) \leftarrow$ walltime_now()
  $t_{cpu_1}(i) \leftarrow$ cputime_now()
  **for** $k = 1$ to $m(i)$ **do**
    **for** $l = 1$ to $n(i)$ **do**
      operation() // *operation* state.
    **end for**// *inner loop* end.
    sleep($t_{sleep}(i)$) // *sleep* state.
  **end for**// *outer loop* end.
  $t_{cpu_2}(i) \leftarrow$ cputime_now()
  $t_{wall_2}(i) \leftarrow$ walltime_now()
  << calculate $t_{sleep}(i+1)$ and $n(i+1)$ >> // *estimation* state.
  $i \leftarrow i + 1$ // enter phase $i + 1$.
**end loop**
```

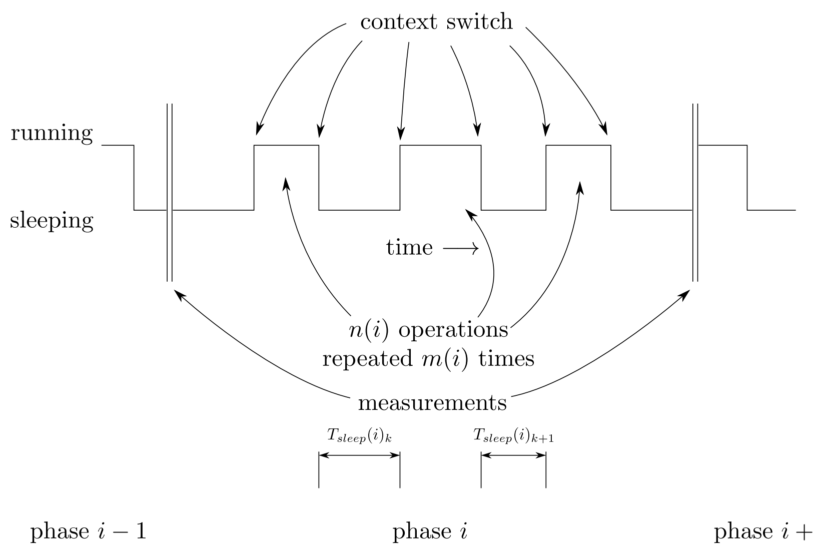


Figure 30: Illustration of a phase in a load generator, with it's context-switches, $m(i)$ sleep intervals and $m(i) \cdot n(i)$ operations.

7.4 The operation

The load from Spurg-bench is mainly generated by the operation[3]. The operation is simply a computation running some deterministic algorithm. Different implementations of the operation create different kinds of load on the CPU. The operation creates a load on the system and can exercise the CPUs, ALUs, floating-point units, vector-units and memory hierarchy with caches and other memory units.

Since the load generated from Spurg-bench should be able to fluctuate to keep a constant utilization while the performance of the execution environment is changing, the operation should have a short enough execution time to give the load generator a small control granularity. Although a small control granularity is required, a very short execution time will introduce overhead. The execution time should thus be preferably in the range of microseconds on a typical system. Listing 1 illustrates an example operation which can be used with Spurg-bench. When run on a fairly new laptop the estimated time of this operation varies between 6 and $11\ \mu s$, depending on the frequency.

For our purpose the operation in Listing 1 is sufficient although trivial since we wanted to compare different scheduling and load balancing policies against each other on the same hardware. For benchmarking hardware platforms against each other an operation which better represents a real computation would be better suited. We chose to use floating-point instruction for the operation to use more energy, thus generating more heat among other things.

[3]There is also some load generated from the load generator and runner script, but they are assumed to negligible.

Listing 1: Example of a operation using the processors floating-point units.

```
int operation()
{
        int     i;
        double  a  = 2.0;
        for (i = 0; i < 1000; i++) {
                a *= 2.0;
        }
        return 0;
}
```

7.5 Model Validation

We validated our analytically defined models with benchmarks on real hardware to determine their suitability in a real-world system. For these experiments we choose the Exynos 4 quad-core CPU and the derived power model given in Equation 11 with the parameters representing the hot case in Table 2. We hence stressed the CPU with two benchmarks: Spurg-bench for each combination of clock frequency and #cores. After the benchmarks, we multiplied the elapsed execution time with the average power dissipation for each test to obtain the energy profile.

We used the following assumptions for each test:

- As n-Spurg-bench threads are running only when n-CPU cores are active. The others are shut down.
- The number of operations per threads is constant for all test
- The load is 99% for a Spurg-bench thread
- No other workload is running except some light Linux background tasks

We evaluated configurations with 1,2,3 and 4 threads. Each experiment was run with clock frequencies 400 MHz – 1600 MHz on a Exynos 4 platform and the power was measured for each test. We executed 40k Spurg-bench operations for each thread (which means that a low clock frequency will use much time but little power and a fast clock will use much power but short time). Figure 31 shows the total energy consumption for each Spurg-bench run. Very low clock frequencies will use much energy since the execution time is very long even though the power is low. On the opposite side, very high frequencies also use much energy due to the very high dynamic power and high temperatures.

We modeled the same situation with the power model given in Equation 11 multiplied by the execution times obtained in the previous experiment. Figure 32 shows the model based results when multiplying the power model with the real execution times.

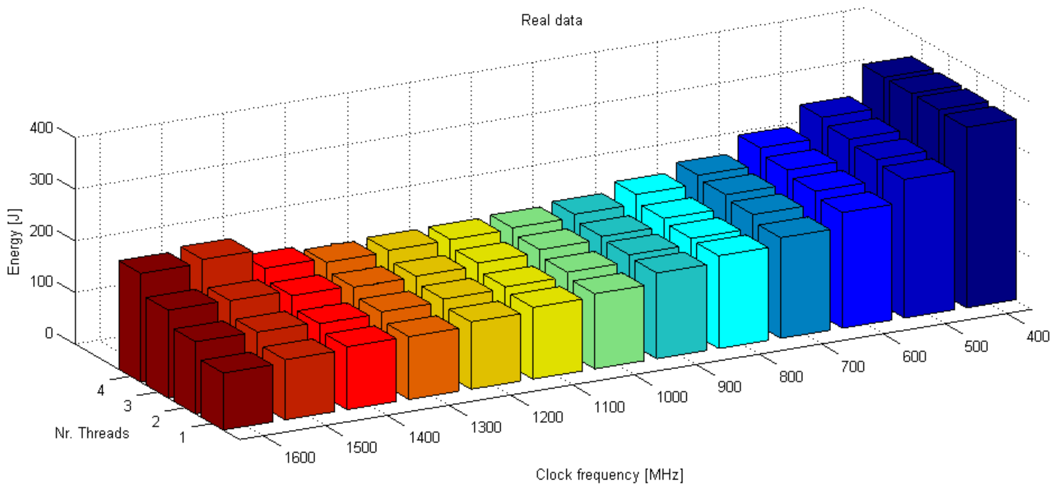


Figure 31: Power as function of #cores and clock frequency running Spurg-bench

In order to compare the model with real data, we calculated the error difference for each set of frequency and #cores. Figure 33 shows the error between the model and real data.

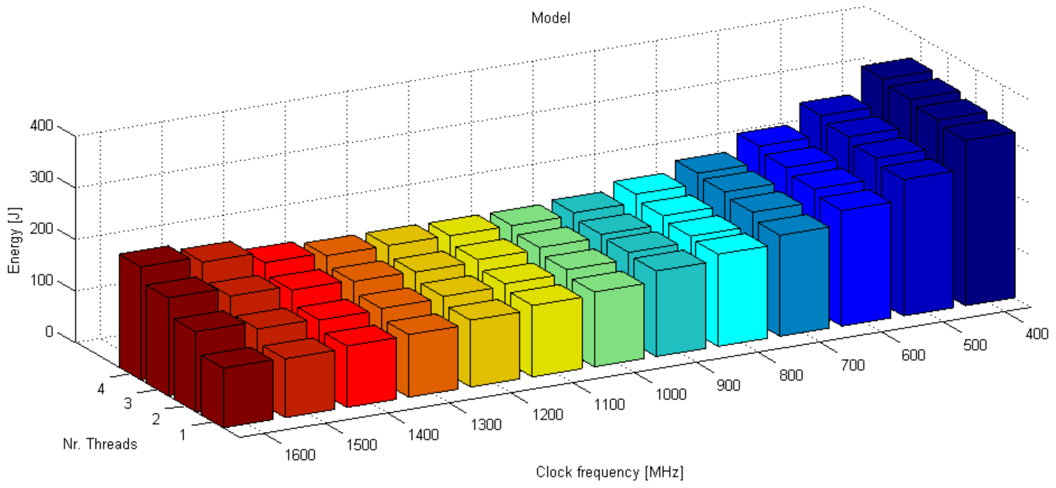


Figure 32: Power as function of #cores and clock frequency using the power model in Equation 11

As we can see in this figure the error is quite small. We can see a difference for 2 reasons:

- The model was trained with the `stress` benchmark while the test are run with Spurg-bench. Since these two tests (most likely) use different kinds of operations, the power dissipation differs somewhat.
- The 3rd degree polynomial is not perfectly representing the real data but it is quite close.

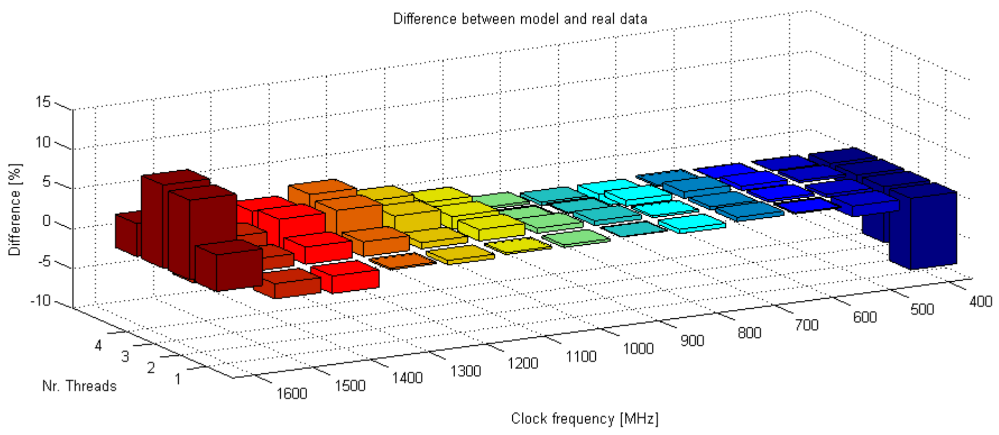


Figure 33: Difference between real data and model for each configuration

The mean error for the whole model was 3.22%.

8 Experimental Results

This section presents early results from the implemented power manager on the Exynos 4412 based (quad-core ARM Cortex-A9) Odroid-x board [22]. To replicate the default behavior and create a fair comparison to the frequency governors, a monitoring application was created to log the load level of each CPU core and to request additional resources in case the load level exceeded a certain threshold. Furthermore, the monitor measured the number of active threads to give the notion of system parallelism to the power manager.

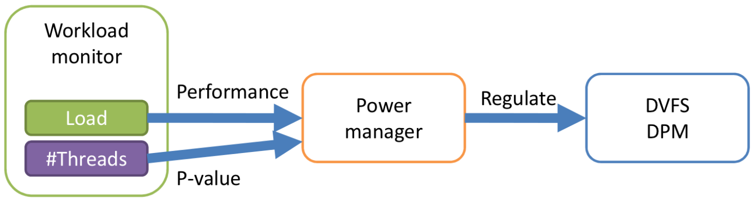


Figure 34: Structure of the workload monitor: The load level is used as measurement of performance and number of threads determines the P-value

Figure 34 illustrates the monitor structure: the total system workload is measured and used as the *performance*. A QoS setpoint is defined as the maximum workload allowed before adding more resources. The monitor measures the amount of active threads during one iteration, which means that the P-value will change between monitor iterations. Our notion of QoS is hence directly the load level. The parallelism of the system was determined by measuring the number of actively running threads i.e. more threads equals a higher level of system parallelism.

8.1 Spurg-Bench

The first energy benchmark was the Spurg-Bench [23] (See previous Sections). As workload, the user can select a certain number of operations to execute, which are evenly divided among the created threads. For our experiments, we chose 100k floating points as the operations executed by the Spurg-Bench threads. Spurg-Bench finally measures the execution time of completing all of the operations.

We ran nine different use cases of Spurg-Bench with four threads and load levels [10 20 30 40 50 60 70 80 90](in %) per thread. All experiments were run both on the default Linux CFS with the ondemand governor and with our optimized power manager directly controlling both DVFS and DPM though the Linux `sysfs` interface. We ran each configuration in both hot (+20°C) and cold (−20°C) ambient temperatures.

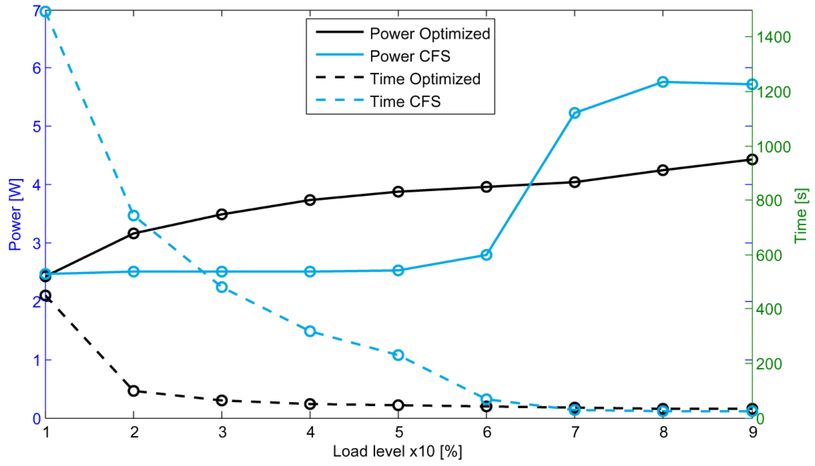


Figure 35: Power and time results from Spurg-Bench compared with the standard Linux CFS+ondemand policy (Rings are data)

The execution time and the power trace for each experiment is shown in Figure 35. Compared to the default CFS+ondemand, the optimized case has an overall higher power dissipation except for when the workload exceeds roughly 65%. On the other hand, the optimized case has a much faster execution time as the load levels go below 60%. These results are due to the combination of using CFS and the ondemand governor: **a)** The workload is always distributed on all cores, which leads to a very low load level on each core **b)** As long as the load levels are very low, the `upthreshold` for increasing the clock frequency is not reached. This means that the CPU retains a low clock frequency, and hence the execution time of the work remains very slow.

While executing workload at slow phases decreases the dynamic power dissipation, the energy consumption increases drastically because of the long execution time. As a product of both power and time, energy is usually wasted when executing workloads at very slow speeds because of the static leakage power ever present. The cold case was removed from Figure 35 because of illustrative reasons, it resulted in similar execution times and an overall lower power dissipation because of lower static leakage power.

The energy trace for Spurg-Bench is shown in Figure 36. Our optimized power manager beats the standard CFS with ondemand governor with a large margin for load levels lower than 60%, and for work loads above this point the results are rather similar.

8.2 Mplayer video

Since Spurg-Bench is a synthetic benchmark, we chose a consumer product as our next evaluation. The chosen experiment was a multi-threaded video decoding benchmark using `mplayer`. We ran each video configuration with different levels of parallelism by fixing the number of decoder threads of the video player with the option `-lavdopts threads=<n>` and `-benchmark` to print evaluation results. The experiments were run with three different video (Mpeg4) resolutions: 1080p, 720p and 480p on both the default Linux CFS with the ondemand governor and with our optimized power manager and each experiment was iterated 10 times.

Figure 37 shows the energy consumption for each number of threads, all video resolutions and the power management schemes for the hot use-case. The effect of low parallelism is mostly noticed in the 1080p case since high video quality forces the maximum clock frequency, which has a very high dynamic power. The high clock frequency also causes high temperatures, which increases the static

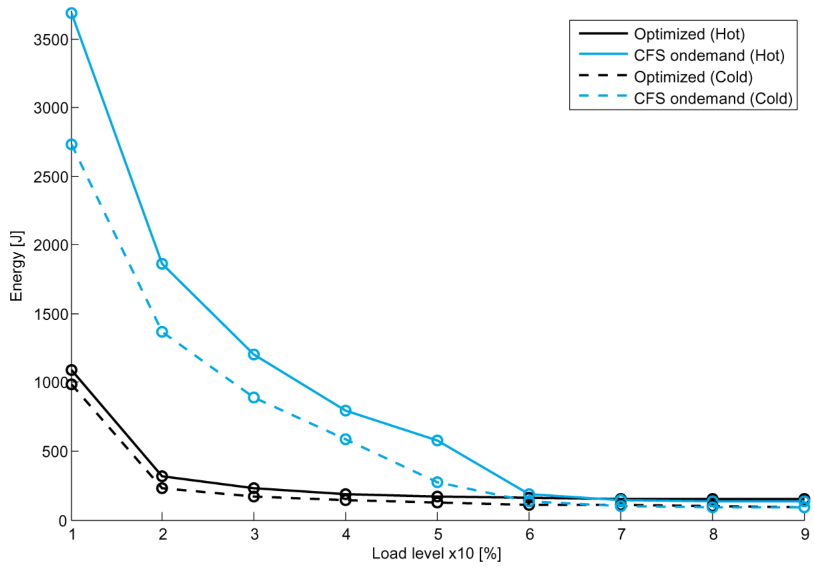


Figure 36: Energy results from Spurg-Bench compared with the standard Linux CFS+ondemand policy (Rings are data)

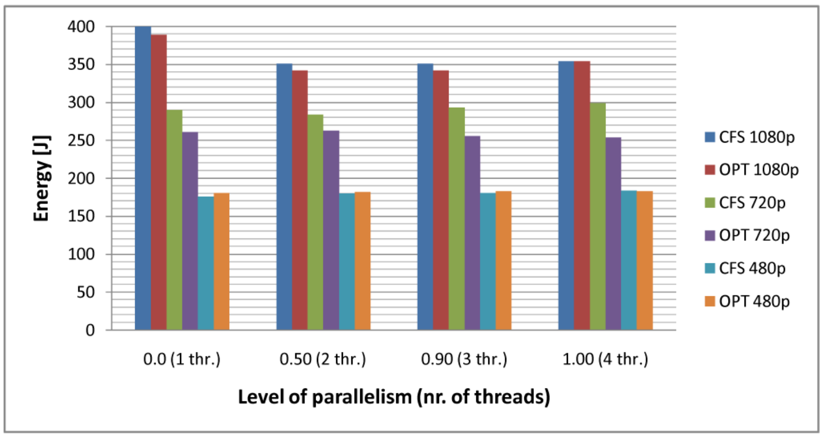


Figure 37: Hot use-case: Mplayer benchmarks with different video resolutions using Linux CFS+ondemand vs. optimized power manager

power further.

In the cold use-case shown in Figure 38, the overall energy consumption is almost half of the energy consumption in the hot use-case (Figure 37). We especially notice the effects of lower leakage power in the 1080p cases, in which the CPU is running on a very high clock frequency but still consumes a conservative amount of energy. Since the consumed energy is lower, the margin for improvements is tighter and the optimized power manager beats the CFS+ondemand only slightly in the 720p case and the other use-cases show roughly the same energy consumption.

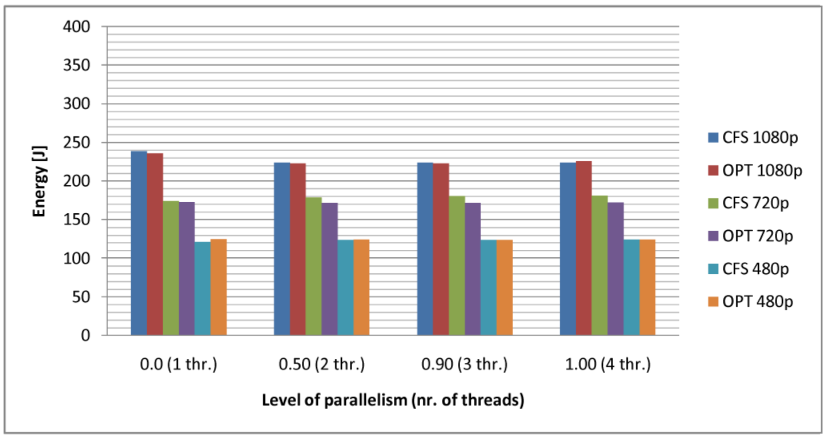


Figure 38: Cold use-case: Mplayer benchmarks with different video resolutions using Linux CFS+ondemand vs. optimized power manager

The choice of parallelism in the 720p and 480p case did not influence the energy consumption for either CFS+ondemand or the optimized power manager much because of: **1)** The lower video quality allows the CPU to run on low/medium clock frequency, which has a significantly lower dynamic power dissipation i.e. changing clock frequency in the low/medium frequency spectrum does not affect the power dissipation significantly. **2)** We benchmarked `mplayer` and noticed that it does not fully scale to four cores, which results in energy waste when running on too many threads. We also noticed increased kernel space activity when running more threads, which may result in energy waste roughly proportional to the energy savings gained by increased parallelism.

In order to fine tune the application, mplayer could be modified to, instead using workload as the performance metric, directly measure the framerate of the decoder. This would result in a less generic, but a more power efficient system designed for mplayer.

8.3 Mixed application

We finally evaluated the energy efficiency in a mixed-load scenario. The primary load was the earlier mentioned `mplayer` video decoding threads, but furthermore we stressed the system with Spurg-Bench threads using different load levels. This scenario was built to represent the presence of background tasks in a real system executing a workload. We fixed the `mplayer` threads to `-lavdopts threads=4`, i.e. four `mplayer` threads, and we generated two Spurg-Bench threads each with the load level in the range [10% - 90%]. The Spurg-Bench threads were run on a lower priority to not degrade the video playback, and the threads executed a fixed number of operations during the video playback.

We decoded a 720p video during external Spurg-Bench loads in the range [10% - 60%] and a 480p video during external Spurg-Bench loads in the range [70% - 90%] to not overload the system. The results are shown in Figure 39.

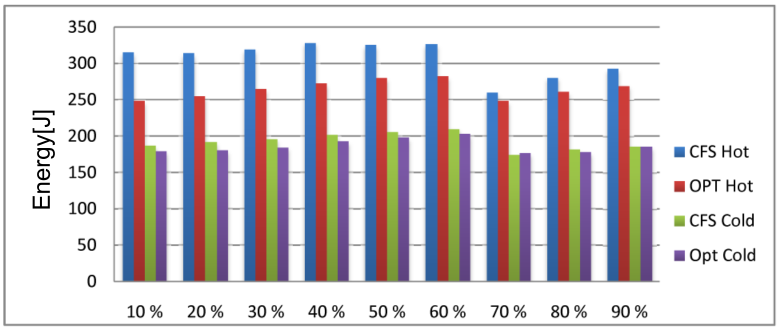


Figure 39: Energy results mplayer from benchmark in combination with background loads using Spurg-Bench threads, x-axis is the load level of the background task

As earlier concluded, the Linux CFS with the ondemand governor handles low workload poorly. This effect is also noticed in the mixed-workload case especially in cases with low external workload. The energy savings of the optimized power manager is obtained from a faster execution of the external workload, and after this only dissipating power for decoding the video. Both methods become more equal as the workload increases, but since our power manager relies on both DVFS and DPM it manages the power usage more efficiently and has an edge over the default configuration. By decreasing the ambient temperature the results of both methods are rather similar with only a slight edge for the optimized power manager in the low load cases, which is because of the lower static power dissipation.

9 Measurement set-ups

To calculate the energy, it is vital to have an efficient way of measuring the power dissipation of the devices. An embedded platform consists of many components (apart from the CPU), and different levels of granularity should be considered for fair power evaluation.

9.1 External power measurements

Our first kind of test bench is based on external power measurements. The device under evaluation is a Exynos 4412 chip implemented on the Odroid-x board [22] and uses a quad-core ARM Cortex-A9 CPU. The board has 1GB DDR2 main memory, six USB 2.0 ports, HDMI, audio and debug ports. The clock frequency of the CPU can be scaled from 200 MHz to 1600 MHz in steps of 100 MHz and all cores (except Core0) can be individually shut down on demand.

The power dissipation of the device is measured with an external power logger implemented on a Raspberry Pi single-chip computer. The Raspberry Pi is connected with probes on both the board level power supply and directly on the CPU power supply, which means that the device is able to measure the power of both the single CPU as well as the board including memory and peripheral devices. The Raspberry Pi logs the current and voltage over a shunt resistor connected to a fast A/D converter, which sends the power readings to the Raspberry Pi over the i2c bus. Figure 40 shows the Odroid-x board connected to the Raspberry Pi with the power probes visible.

A/D converter Figure 41 shows the schematics of the shut resistor and A/D converter used for measure the power. The device supports eight probes which can all be connected to individual devices. Each probe consists of two wires connected in serie to the current feed of the device under measurement. A probe is then connected to a INA226 A/D converter which measures both voltage, current and calculates power. The obtained data is then sent on the i2c bus as serial data in form *voltage*, *current*, *power*, and this sequence is continuously repeated. The left part of Figure 41 shows the Raspberry Pi pin header which connects directly by the Raspberry Pi and is read by the Datalogger software.

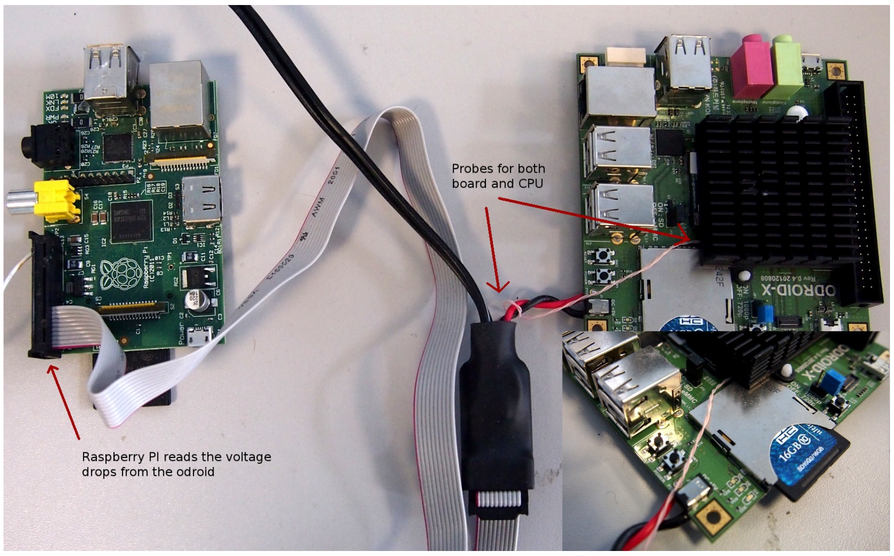

Probes for both board and CPU
Raspberry PI reads the voltage drops from the odroid

Figure 40: Raspberry Pi used as external power meter and connected to the Odroid

Datalogger The power readings are logged by a Raspberry Pi connected to the i2c bus as shown in last paragraph. The Raspberry Pi is accessible by ssh logins and the power readings can be shown in real time or written to a log file. Power readings are obtained by listening to the i2c bus and sorting out the serial sequence of data in form of voltage,current and power. Figure 42 shows a screen shot of the Raspberry Pi power reading program which plots: Time stamp, Board voltage, Board current, Board power, CPU voltage, CPU current and CPU power from the left column to the right. The Datalogger software is opensource and available at https://github.com/ESLab/DataloggerExynos4412

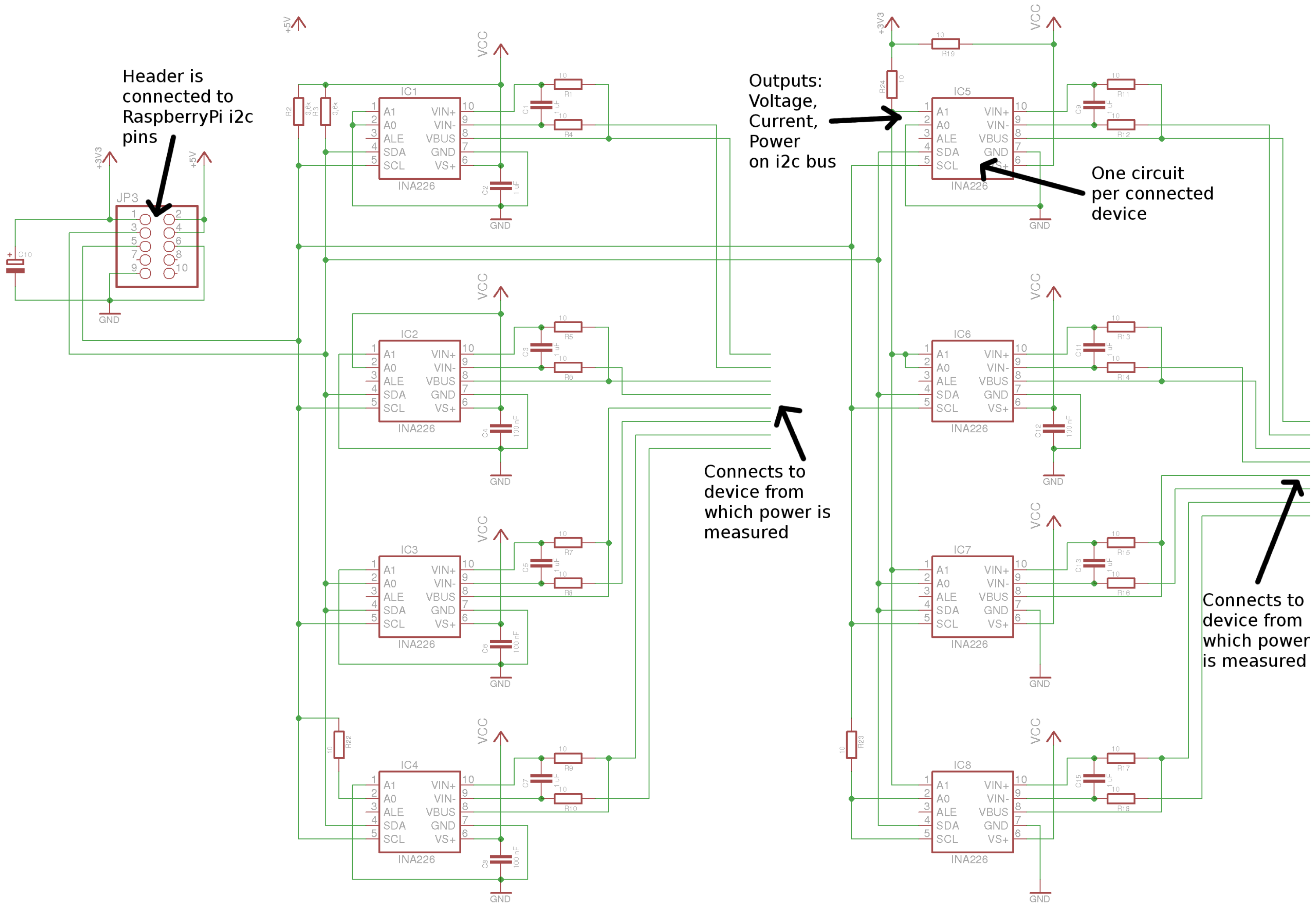
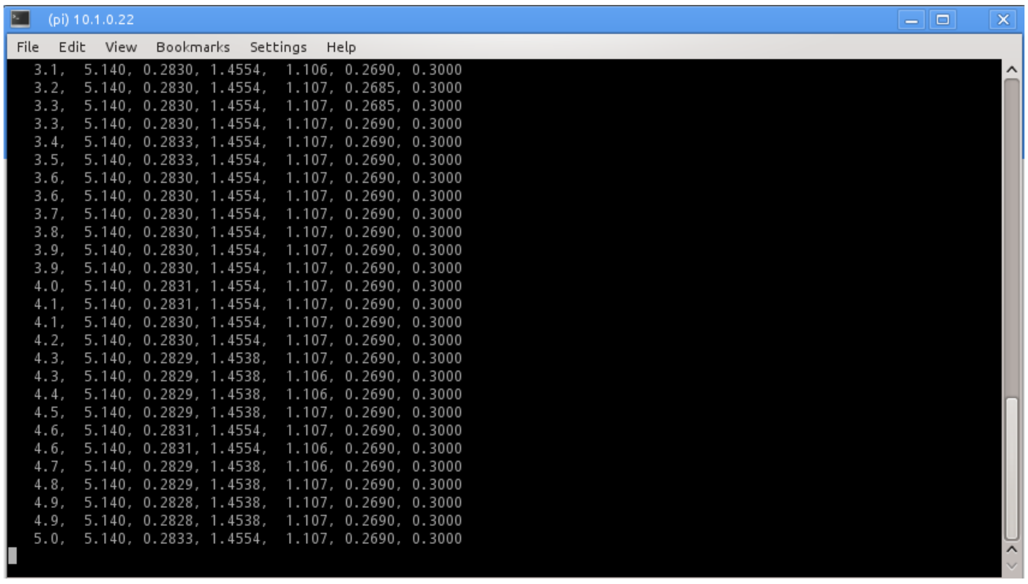

Header is connected to RaspberryPi i2c pins
Outputs: Voltage, Current, Power on i2c bus
One circuit per connected device
Connects to device from which power is measured
Connects to device from which power is measured
JP3
IC1
IC2
IC3
IC4
IC5
IC6
IC7
IC8
INA226
A1
A0
ALE
SDA
SCL
VIN+
VIN-
VBUS
GND
VS+
VCC
GND
+5V
+3V3

Figure 41: Schematics of the external power meter used in the Exynos 4412 board



```
  3.1,  5.140, 0.2830, 1.4554,  1.106, 0.2690, 0.3000
  3.2,  5.140, 0.2830, 1.4554,  1.107, 0.2685, 0.3000
  3.3,  5.140, 0.2830, 1.4554,  1.107, 0.2685, 0.3000
  3.3,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.4,  5.140, 0.2833, 1.4554,  1.107, 0.2690, 0.3000
  3.5,  5.140, 0.2833, 1.4554,  1.107, 0.2690, 0.3000
  3.6,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.6,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.7,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.8,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.9,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  3.9,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  4.0,  5.140, 0.2831, 1.4554,  1.107, 0.2690, 0.3000
  4.1,  5.140, 0.2831, 1.4554,  1.107, 0.2690, 0.3000
  4.1,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  4.2,  5.140, 0.2830, 1.4554,  1.107, 0.2690, 0.3000
  4.3,  5.140, 0.2829, 1.4538,  1.107, 0.2690, 0.3000
  4.3,  5.140, 0.2829, 1.4538,  1.106, 0.2690, 0.3000
  4.4,  5.140, 0.2829, 1.4538,  1.106, 0.2690, 0.3000
  4.5,  5.140, 0.2829, 1.4538,  1.107, 0.2690, 0.3000
  4.6,  5.140, 0.2831, 1.4554,  1.107, 0.2690, 0.3000
  4.6,  5.140, 0.2831, 1.4554,  1.106, 0.2690, 0.3000
  4.7,  5.140, 0.2829, 1.4538,  1.106, 0.2690, 0.3000
  4.8,  5.140, 0.2829, 1.4538,  1.107, 0.2690, 0.3000
  4.9,  5.140, 0.2828, 1.4538,  1.107, 0.2690, 0.3000
  4.9,  5.140, 0.2828, 1.4538,  1.107, 0.2690, 0.3000
  5.0,  5.140, 0.2833, 1.4554,  1.107, 0.2690, 0.3000
```

Figure 42: Screenshot of Raspberry Pi console measuring power

9.2 Internal power measurements

Other devices such as the Odroid-xu-e (Manual not available at this time) contain internal registers for measuring voltage, power, temperature etc. This feature is very useful since no external device is needed. However, to measure the power dissipation internally uses a certain amount of CPU power, which means that measurement scripts or programs should be designed with this fact in mind to not influence the measurement results. The device shown in Figure 43 is the Odroid-xu-e with the Exynos 5410 CPU containing four ARM Cortex-A15 cores and four ARM Cortex-A7 cores. The A7 cores can be clocked to frequencies between 250 and 1200 MHz, but the OS implementation is set to switch from the A7 cores to the A15 as the clock frequency reaches 800 MHz. The A15 cores can be clocked to 1600 MHz and even over clocked up to 1800 MHz. The chip is designed to run low workload on the energy efficient A7 cores and high workload on the faster A15 cores.

The internal registers in the Odroid-xu-e are capable of monitoring the Cortex-A15 power, Cortex-A7 power, GPU power and memory power. These power readings can therefore be sampled with a user defined rate and printed to a log file with a simple script. An example script for reading the power of the Odroid-xu-e is found at https://github.com/ESLab/Exynos5PowerMeter

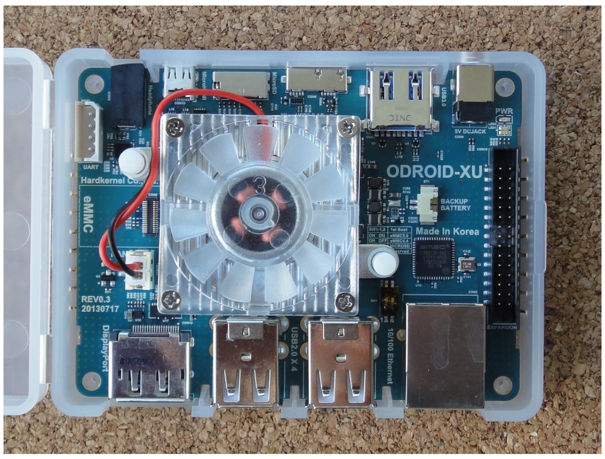


Figure 43: Odroid-xu device with the big.LITTLE configuration containing 4xCortex-A15 and 4xCortex-A7 cores

10 Conclusion

In this report we have presented the most typical considerations and issues when conducting power and energy measurements on multi-core microprocessors. The report firstly explains the power dissipation and its relation to energy consumption. As manufacturing technologies shrink and the cores increase in number, the static leakage power in the transistors increases and starts to dominate the total power dissipation [16]. DPM using sleep states as a technique to reduce the static power dissipation of microprocessors by switching off parts of the chip such as the CPU cores as the core is not in use. A problem with DPM, however, is to provide a notion of parallel workload since this information is not present in power management systems today.

We have described the notion of load in typical Linux based systems and suggested ways of extending this notions with QoS requirements and scalability parameters directly in the applications. These notions could create a more rich application more tightly connected to the power management system, and could be used to create more energy efficient scheduling and power management.

We have presented an energy benchmark for multi-core systems called Spurgbench, which is to our knowledge the only benchmark tool capable of variable load generation suitable for embedded-type of platforms. The commercial tool SpecPOWER [13] was initially evaluated on our system, however, since the tool has been targeted for high-end processors, our embedded test bench was simply too slow for the calibrations to be valid.

An evaluation of the presented benchmark and run-time power manager provides early results on the achievable power and energy savings and provides a comparison of the proposed approach with by default Completely Fair Linux scheduler and load balancer.

Finally we have demonstrated two ways of conducting power measurements by using both internal and external power measurement devices. We have presented an open-hardware solution to read power traces from any kind of chip provided that the current feed pin are exposed. Our open-source software running on a low-cost Raspberry Pi platform is one example of creating a cost efficient power tracing device without industrial scale manufacturing equipment.

References

[1] M. Anis and M.H. Aburahma. Leakage current variability in nanometer technologies. In *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on*, pages 60–63, July 2005.

[2] Christian Bienia. Benchmarking modern multiprocessors.

[3] M. Bohr. A 30 year retrospective on dennard x0027;s mosfet scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, Winter 2007.

[4] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23 –29, jul-aug 1999.

[5] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473 –484, apr 1992.

[6] Sangyeun Cho and R.G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *Parallel and Distributed Systems, IEEE Transactions on*, 21(3):342–353, March 2010.

[7] Alonzo Church. An unsolvable problem of elementary number theory. 58(2):pp. 345–363.

[8] Harold J. Curnow and Brian A. Wichmann. A synthetic benchmark. 19(1):4349.

[9] K. Dev, A.N. Nowroz, and S. Reda. Power mapping and modeling of multi-core processors. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 39–44, Sept 2013.

[10] Fredric Hällis, Simon Holmbacka, Wictor Lund, Robert Slotte, Sébastien Lafond, and Johan Lilius. Thermal influence on the energy efficiency of workload consolidation in many-core architectures. In *Digital Communications - Green ICT (TIWDC), 2013 24th Tyrrhenian International Workshop on*, pages 1–6, 2013.

[11] John L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. 33(7):2835.

[12] Chung-Hsing Hsu and S.W. Poole. Power signature analysis of the specpower_ssj2008 benchmark. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 227–236, 2011.

[13] Chung-Hsing Hsu and S.W. Poole. Power signature analysis of the specpower_ssj2008 benchmark. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 227–236, 2011.

[14] A. Inoue. Comparison between power gating and dvfs from the viewpoint of energy efficiency. In *Quality Electronic Design (ISQED), 2012 13th International Symposium on*, pages 601–608, March 2012.

[15] Kelly Iondry. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, 1999.

[16] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 275–280, New York, NY, USA, 2004. ACM.

[17] J.T. Kao, M. Miyazaki, and A.P. Chandrakasan. A 175-mv multiply-accumulate unit using an adaptive supply voltage and body bias architecture. *Solid-State Circuits, IEEE Journal of*, 37(11):1545–1554, Nov 2002.

[18] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec 2003.

[19] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[20] Jonathan G. Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[21] Yongpan Liu, Huazhong Yang, R.P. Dick, Hui Wang, and Li Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, pages 204–209, March 2007.

[22] Samsung Electronics Co. Ltd. Exynos 4412 risc microprocessor – user's manual, 2012.

[23] Wictor Lund. Spurg-bench: Q&d microbenchmark software, May 2013.

[24] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Configuration and Power Interface: Open Standard, Operating System, Power Management, Cross- Platform, Intel Corporation, Microsoft, Toshiba, ... Sleep Mode, Hibernate (OS Feature), Synonym*. Alpha Press, 2009.

[25] F.J. Monaco, E.L.C. Mamani, M. Nery, and P.N. Nobile. A novel qos modeling approach for soft real-time systems with performance guarantees. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 89 –95, june 2009.

[26] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[27] T. Rauber and G. Runger. Energy-aware execution of fork-join-based task parallelism. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 231–240, 2012.

[28] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[29] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2):305–327, Feb 2003.

[30] M. Sadri, A. Bartolini, and L. Benini. Single-chip cloud computer thermal model. In *Thermal Investigations of ICs and Systems (THERMINIC), 2011 17th International Workshop on*, pages 1–6, 2011.

[31] Stephen Shankland. What would happen if moore's law did fizzle?, October 2012.

[32] H. Singh, K. Agarwal, D. Sylvester, and K.J. Nowka. Enhanced leakage reduction techniques using intermediate strength power gating. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(11):1215 – 1224, nov. 2007.

[33] SPEC. *SPEC benchmark methodology*, 2011.

[34] B.M. Tudor and Yong-Meng Teo. Towards modelling parallelism and energy performance of multicore systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2526–2529, 2012.

[35] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. 58:345363.

[36] Ray Walker. Examining load average. *The Linux Journal*, December 2006.

[37] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. 27(10):10131030.

[38] Peng Zhang and Zheng Yan. A qos-aware system for mobile cloud computing. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 518 –522, sept. 2011.

[39] Du Zhi-bo, Chen Yun, and Chen Ai-dong. The impact of the clock frequency on the power analysis attacks. In *Internet Technology and Applications (iTAP), 2011 International Conference on*, pages 1–4, 2011.

ISBN 978-952-12-3087-5
ISSN 1239-1891