Components for Energy-Efficient Operating Systems

Clemens Lang
Friedrich-Alexander University Erlangen-Nuremberg
clemens.lang@fau.de

ABSTRACT

The proliferation of smartphones and tablet computers has raised awareness of an important aspect of modern system design: energy usage can no longer be neglected when designing systems. Advances in battery technology are lagging behind other features like clock speed, memory, storage or bandwidth [3], which makes energy consumption a major issue in today's systems. Not only portable devices, but also data centers profit from energy-aware systems [1, 4].

Operating systems should not ignore energy consumption but use hardware-provided power saving features as efficiently as possible. This document outlines the components used in operating systems to perform efficient power management. It also attempts to give insight into the complexity of power management and highlights problems.

1. INTRODUCTION

Comparing the growth rates of battery capacity to those of clock frequency and memory yields alarming results: Table 1 shows that in the timespan it took to raise clock speeds to a hundredfold, battery life has only increased by a factor of ten. The comparison of RAM and battery life is even more steep.

Year	Clock Speed	RAM	Battery Life
1981	1	1	1
1991	4	512	2
2001	187.5	65535	4
2010	1200[1]	262144	10

Table 1: Technology trends of mobile devices: Features in multiples of their value in 1981, adapted from [3].

In contrast to memory management, which has been a standard component of operating systems for decades, energy consumption was largely unconsidered in software design before 1998 [9]. In general-purpose operating systems, energy efficiency can be increased by adjusting power consumption of a process at runtime. Predicting or measuring and extrapolating the power usage of a running program is a requirement for good power management decisions. For obvious reasons, prediction should add as little overhead on energy consumption as possible. Interpreting or analyzing code is thus a method out of question in online approaches to power management – we need different tools to achieve the necessary precondition. Furthermore, it is required to gain insight into where and how power is used in a computer; on a software level, that means finding out which behavioral patterns are the most energy inefficient ones. This does, however, not mean that other energy consumers in the system can be ignored if we want to minimize the power consumption of the whole system.

The following section will give a brief overview on where and how power is used in CPU and memory. Methods to estimate the power usage at runtime without specialized hardware will be highlighted. Section 3 lists mechanisms available in common hardware to reduce the power usage. The following section discusses operating systems' approaches to minimizing energy consumption by using policies and a number of challenges with software-based power management methods, giving insight into the complexity of power management. Section 5 covers implementation details, before the last section concludes this overview.

[1] in two cores

2. MEASURING POWER CONSUMPTION

Power in semiconductor chips is consumed when current is flowing due to either leakage (i.e., power dissipation) or due to loading or unloading of capacitors caused by a transistor switch [10]. Power dissipation depends on static parameters like voltage and time. The power usage also consists of a dynamic part that is caused by the switching of transistors. Assuming parts of the chip with high switching frequencies account for a significant part of energy consumption defines a region of interest when searching for energy-demanding areas in the system.

Although static power usage might seem irrelevant at first, it can still be beneficial to model it to rule on scheduling and power management decisions. Since static power consumption depends on the voltage, systems using dynamic voltage scaling (DVS) should not neglect it when optimizing energy usage.

To model dynamic power consumption accurately, a precise understanding of where power is used dynamically is crucial. Regions of interest, among others not being discussed in this article, are the *central processing unit* (CPU), caches, a *memory management unit* (MMU), and *dynamic random access memory* (DRAM).

The CPU's high switching frequency suggests a considerable contribution to power usage. The instructions being executed might influence the dynamic power consumption of the processor; this will be discussed in greater detail on the following pages.

Depending on the associativity and the frequency of references, caches also contribute to dynamic power consumption.

One would expect applications heavily using the cache to show higher energy consumption than others keeping their data in CPU registers. Due to the caches inside an MMU like the *translation lookaside buffer* (TLB), significant power usage can also be expected in memory management.

Random access memory is responsible for another considerable share of power consumption. On the one hand, this is caused by the complex mechanisms involved in accessing DRAM. On the other hand, the period refresh needed for dynamic RAM is reflected in an increased static power consumption. While one expects the static part to be independent of the instructions being run, memory-intensive applications could cause higher dynamic energy usage.

Note that this enumeration is not exhaustive. Peripheral devices, storage, I/O, cooling and network cards further contribute to power consumption and may each offer their own power saving mechanisms but are not discussed in this article.

To effectively optimize efficiency by reducing dynamic power consumption, a relation between certain behavioral patterns of software (like a series of memory accesses, or a CPU-bounded computation) and the power consumption at different power saving settings is required. This data can either be supplied by the manufacturer or be measured using a set of benchmarks. Note that current consumption measurement equipment is not integrated into off-the-shelf computers; the data can thus not be acquired online for the system at hand but must be gathered ahead of time. Due to power consumption being very hardware-specific, we have to assume the results will not generalize well enough to be useful for other setups.

Both Weißel et al. and Snowdon et al. measure energy consumption of the complete system they are trying to optimize. The former used a shunt resistor to measure the current flowing into their test system, an Intel IQ 80310 PCI board rated at 3.3 V. The static power consumption was measured while the test board was idle. A commercial AC power meter between the machine's power plug and the wall socket measured power for Snowdon et al. While both test setups did measure the power consumption at the point were it should be minimized, detailed statistics for parts of those systems (e.g., CPU, memory, I/O) have not been generated. The recent development to integrate chips with each other using the package on package approach or by merging the chips completely further increases the difficulty of separate measurements. On-chip mechanisms to gauge power consumption could solve this problem and simplify per-component measurements.

2.1 Event Counters

To tell different software behaviors apart assuming the software is unknown ahead of time, event counters can be used. Commonly used for performance analysis, event counters are hardware-provided counters for events, such as cache misses, cycles, or memory accesses. The number of countable events depends on the processor and is usually in the range of hundreds [5]. However, only a few events can be measured simultaneously due to the relatively low number of event counters. Typical numbers of available counter registers are single-digit to low two-digit numbers, e.g., between 2 and 6 for ARM11 and the ARM Cortex series of processors [6].

Since the number of events exceeds the number of available counters, a subset of events must be chosen. To determine the ideal set of events, a series of measurements can be used: running a number of benchmarks with known behavior at different power saving settings with different subsets of event counters selected generates data that can be used to compute the correlation between counted events and their corresponding power consumption. The events showing a high correlation are good candidates to estimate the energy usage of a process when dedicated energy measurement hardware is not available.

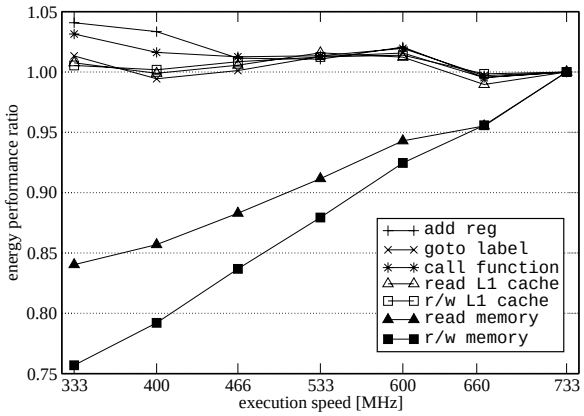

Figure 1: Energy consumption for several benchmarks relative to the energy consumption at full clock speed. From [13].

To target highest efficiency in power usage, the *energy performance ratio* must be minimized. An energy performance ratio of r % at a certain energy saving setting means the task needs r % of the energy it would have needed at full clock speed. Research shows CPU-bound tasks run at high efficiencies regardless of clock speed, while memory-intensive software is up to 25 % more efficient at lower execution speeds in a test setup used by Weißel et al. in 2002. Figure 1 shows this: the five benchmarks "add reg", "goto label", "call function", "read L1 cache" and "r/w L1 cache" are efficient at all tested clock speeds, while the memory-bound tasks "read memory" and "r/w memory" have a lower energy performance ratio (i.e., higher efficiency) at lower clock speeds. Weißel et al. attribute the savings to the slow response time of the memory and the cycles the CPU needs to stall and wait for the results of the memory access. Since the gap between memory speed and CPU clock speed has increased even further since 2002, this effect is more relevant than ever before. Lowering the CPU frequency relative to the memory frequency reduces the number of cycles wasted, thus increasing efficiency. Note that Weißel et al. only discuss dynamic voltage scaling in theory, but have not measured energy usage with DVS in effect due to missing hardware support for voltage scaling.

2.2 The Performance Energy Trade-off

It is not sufficient to minimize the energy performance ratio in order to build a power-efficient but fast system. We have seen in Figure 1 that CPU-bound tasks are almost equally efficient at any power saving setting while memory-bound tasks are more efficient at lower clock speeds. Considering only these results, the best strategy to save power would be always running at the lowest possible frequency. However, performance and energy usage are closely linked: Figure 2 shows the performance of a subset of those benchmarks in

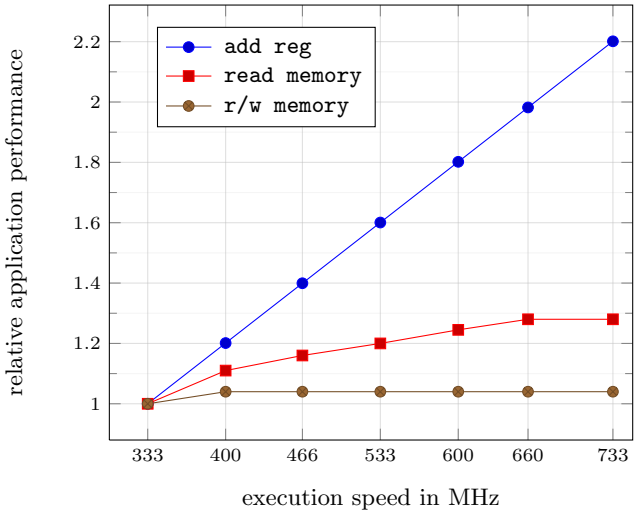

Figure 2: Normalized application performance at different clock speeds. Adapted from [13].

multiples of the execution time at the lowest clock frequency. We can see that the CPU-bound task "add reg" (and the other CPU-bound tasks omitted in this graph) would be slowed down significantly by running at lower clock speeds, leading to a worse user experience in interactive systems.

Since power management decisions also affect the user experience, it makes sense to allow the user to adjust the power management behavior. A simple approach to limit the effects of power saving on user experience is establishing an upper bound on performance loss under power saving. Weißel et al. chose a maximum loss of 10 % in [13], leading to energy savings of up to 15 % compared to execution at optimal performance.

3. POWER SAVING MECHANISMS

Assuming accurate prediction of the behavior of a task, how can power actually be saved? Which mechanisms are offered by today's hardware to reduce power consumption? This section will explain common techniques and discuss power management heuristics proposed in research.

3.1 Dynamic Frequency Scaling

Modern processors' clock frequencies can be adjusted at runtime. Since less switching occurs at lower frequencies, less power is consumed. For example, Intel processors that came to market after the Pentium M processor support *Enhanced Intel Speedstep®Technology* that will scale the core frequency by writing to a machine-specific register [5]. Dynamic voltage scaling is a de-facto standard to save energy [13, 8]. Frequency scaling is often combined with voltage scaling to achieve a higher impact on power consumption.

3.2 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) is a technique where the supply voltage of the CPU is changed during execution. Since the used energy is proportional to the square of the supply voltage, i.e.,

$$E \propto V^2, \tag{1}$$

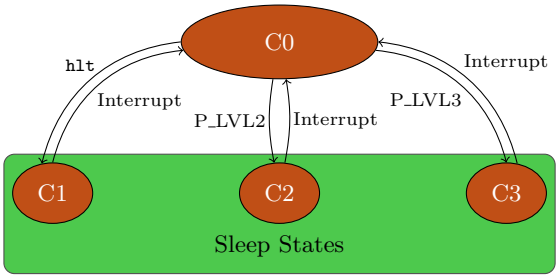

Figure 3: C-states as defined by ACPI. Adapted from [2].

lowering the voltage can reduce the energy consumption significantly [10].

Both dynamic frequency scaling and dynamic voltage scaling are often combined into dynamic frequency and voltage scaling (DVFS), because most processors only support lower voltages at lower clock speeds. Both methods involve a switching time that might be non-negligible and thus has to be taken into account when making power saving decisions. A particularly slow example is the AMD Opteron 246 processor, where voltage scaling takes up to 2 ms when operating within the specifications. This is well outside common switching times in the range of 10 μs to 140 μs Snowdon et al. found for other CPUs [11].

3.3 Sleep States

Modern systems provide several sleep states used for power management. The *Advanced Configuration and Power Interface* (ACPI) standard defines such states for different parts of the hardware. For the short-term power management discussed in this article, the power states of the processor are most relevant. The ACPI specification denotes these states C0, C1, C2, C3, . . . , C*n* [2]. While in state C0, the processor executes instructions. Using the `hlt` instruction, the processor can be put into the *C*1 power state, in which it ceases to execute instructions but maintains execution context and caches. All higher-numbered states are optional. In C-state 2, processors keep their context and caches (which also implies continuation of the cache-coherence protocol) but consume less energy than in C1. C2 is the first state where wakeup delays may be non-negligible. In C3 the processor consumes even less power by handing off cache-coherence to a different entity or flushing its caches completely. Figure 3 illustrates the available C-states and their transitions. Switching delays for these states are available from ACPI to be used by power management policies. Note that switching is only possible between C0 and any other C-state.

Implementations of power management components in operating systems should be aware that hardware may support more than the states defined by the ACPI standard, such as the sub-C-states in Intel CPUs [12, 5], possibly leading to further opportunities for power saving.

4. POWER MANAGEMENT POLICIES

Sections 2 and 3 discussed how to measure or estimate (Section 2) and how to adjust (Section 3) power usage. These components are highly hardware-specific and required in order to increase energy efficiency. However, they just provide the data and actions required to save energy – they do not

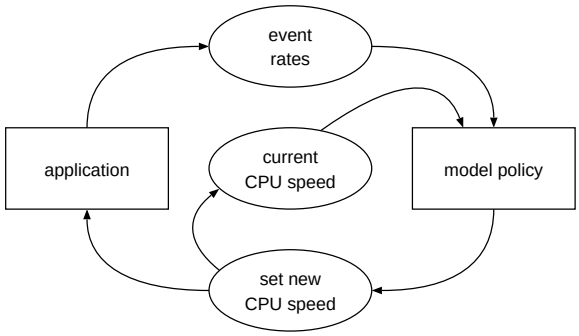

Figure 4: A simple control loop implementing a power management policy. From [13].

save any power on their own. Another component that uses the input data to select the power saving configuration providing the highest energy efficiency is needed – the so-called *power management policy*.

This article previously discussed the use of event counters to classify the workload (see Section 2.1). These counters span a multidimensional space when used as inputs for the power management policy. For each point in this space, the optimal power saving configuration can be pre-computed using a series of benchmarks (see Section 2) using constrained or non-constrained optimization. Different objective functions and constraints lead to different policies, some of which perform better than others.

Figure 4 shows a data flow graph where measurement (event rates, current CPU speed), adjustment (set new CPU speed), and policy (model policy) can be seen.

4.1 The Maximum-Degradation Policy

The so-called *maximum-degradation policy* as suggested by Weißel et al. [13] is represented by the optimization problem

$$\begin{aligned} &\text{minimize} && P \\ &\text{subject to} && T \leq p^{-1} \cdot T_{\text{opt}} \end{aligned} \tag{2}$$

where P is the power consumption, T is the duration of the task[2] with power saving and T_{opt} is the shortest duration the task would take without power saving. p is an adjustable parameter limiting the maximum performance degradation acceptable in order to save power. Values of $p = 0.9$ have been shown to work well [13].

4.2 The Generalized Energy-Delay Policy

A different approach called *generalized energy-delay policy* [11] is given by

$$\text{minimize} \quad P^{1-\alpha} \cdot T^{1+\alpha} \tag{3}$$

with P and T as defined in Equation 2 and $\alpha \in [-1; 1]$ being a variable. Modification of α transforms this policy into a number of policies previously suggested in literature:

- $\alpha = 1$ yields

$$\text{minimize} \quad T, \tag{4}$$

  which minimizes the execution time and thus maximizes the performance. Using this policy will implement a power saving mechanism that will always use the highest available frequencies to finish the task as quickly as possible. In systems that are not completely utilized, the processor can be put to sleep for a longer duration compared to running tasks at lower frequencies. This is called the *race-to-halt* approach.

- $\alpha = 0$ minimizes the energy usage:

$$\text{minimize} \quad P \cdot T =: E \tag{5}$$

- $\alpha = -1$ simplifies the optimization problem to

$$\text{minimize} \quad P, \tag{6}$$

  which minimizes the power consumption.

Values for α between zero and one will throttle tasks depending on their behavior: memory-bound processes achieve higher energy savings while sacrificing little power and can thus be run at a lower frequency than CPU-bound tasks. Although setting α to a value in $[-1; 0]$ will still throttle threads depending on the workload, the energy consumption will be higher than at $\alpha = 0$. These values might be useful in environments where power consumption needs to be reduced regardless of the energy used for the complete task (e.g., because the power consumption should not exceed a maximum value).

4.3 Adjustable Policies

Both the maximum-degradation policy and the generalized energy-delay policy are parametrized. Using this parameter, the operating system or the end user can adjust the policy to their needs. Requirements for power management depend on the purpose of the machine and might change over time: servers have different requirements than battery-powered devices, but servers can quickly turn into battery-powered devices over time, e.g., due to a power failure. Keeping the policy parameters adjustable at runtime allows the operating system to react on changed requirements and integrate with other components relevant for power saving.

4.4 Power Management Challenges

Whether energy can be saved by a certain decision depends on a wide range of factors. Modeling and predicting all of them as closely as possible is a complicated task. Some of the problems in generating accurate models of power consumption are outlined in this section.

4.4.1 Workload Prediction

As shown in Figures 1 and 2 the actual power consumption depends on the workload characteristics. Since we can only sample the behavior with a limited number of event counters, the precision of the prediction might suffer. Matching the behavioral patterns to the corresponding ideal power saving setting is error-prone, too: Figure 5 shows two pathological cases that require a good analysis of the task's behavior – the gzip benchmark is most efficient at the highest frequency, whereas the swim graph has its optimal point at a low (but not the lowest) frequency. It is obvious from this graph that energy saving decisions without insight into the type of workload are impossible.

[2] which is equal to the inverse performance

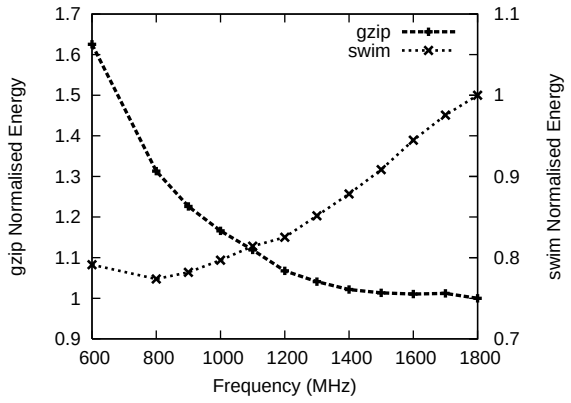

Figure 5: Normalized energy consumption of two benchmarks. From [11].

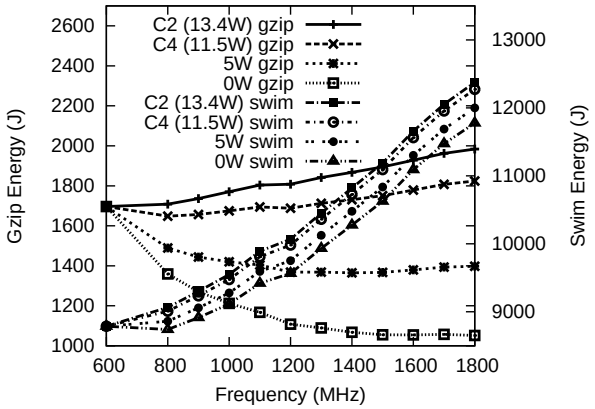

Figure 6: Energy consumption of two benchmarks when using race-to-sleep for the idle states C2, C4 and two hypothetical idle states with 5 and 0 W. From [11].

4.4.2 *Multiple Variables and Variable Memory System Performance*

Besides the CPU clock frequency, other frequencies in the system, such as the bus and memory frequency, might be adjustable at runtime. Such systems introduce another variable to be tested when finding the best power saving configuration, which quickly leads to an exploding number of configurations to be benchmarked ahead of time.

Snowdon et al. also found that memory performance is not completely unrelated to CPU clock speed. Some processor features like out of order execution or pre-fetching might become less effective at lower core frequencies. Those effects are hard to predict, because they are highly hardware-specific and few to no performance counters are available to measure them. Memory energy consumption additionally depends on the memory configuration (e.g., in single vs. multi channel configurations).

4.4.3 *Sleep States*

Tasks run at a lower frequency generally take longer to execute. The time delta between running a task at a high and at a low frequency can not be used to put the CPU to a sleep state if the system is otherwise idle. While this does not matter for heavily loaded machines, it is important for general purpose systems, such as laptops or tablets. Whether or not it is advisable to run a CPU-bound task at a high clock frequency, depends on the time delta and the power savings in sleep states the operating system might be able to put the processor into after task completion.

Consider a task starting at $t_0 = 0$ in two different clock frequencies f_{slow} and f_{fast} with a power consumption of P_{slow} and P_{fast}, respectively. We assume the task needs c cycles; at a frequency f_x the task will finish in

$$t_x = \frac{c}{f_x}, \qquad x \in \{\text{slow}, \text{fast}\}. \tag{7}$$

Without loss of generality, we can assume that $f_{\text{slow}} < f_{\text{fast}}$ and thus $t_{\text{slow}} > t_{\text{fast}}$, since $c = \text{const}$. The energy used then is

$$E_x = t_x \cdot P_x, \qquad x \in \{\text{slow}, \text{fast}\}. \tag{8}$$

However, the processor does not drop to zero energy usage after finishing the task – to get accurate results, we need to consider the energy used in the time $t_{\text{slow}} - t_{\text{fast}}$ where the processor is idle. Let us assume P_{idle} is the power consumption when the processor is idle. This leads us to a decision rule to find out whether we should run a task at a high frequency:

$$E_{\text{fast}} + (t_{\text{slow}} - t_{\text{fast}}) \cdot P_{\text{idle}} \leq E_{\text{slow}}. \tag{9}$$

Equation 9 shows that power consumption in idle mode can not be neglected when trying to minimize the energy consumption in interactive systems. We face a trade-off between the race-to-halt approach and running at a lower frequency for a longer time. The gzip benchmark in Figure 6 shows that with the improvement of idle states in processors, race-to-halt might be a better choice compared to running at low frequencies.

4.4.4 *Power-supply Efficiency and Temperature*

When employing dynamic voltage scaling, the power supply might not work equally efficient at different voltage levels. This imposes another difficulty on operating systems trying to save power, because reducing the voltage used to drive the CPU might not reduce the power consumption from battery or wall outlet either. Similarly, the CPU core temperature plays a role in the system's power consumption because of the power needed for fans and the higher leakage current caused by higher core temperatures.

4.4.5 *Switching Overhead*

Both frequency and voltage switching cause the CPU to be unavailable while the respective setting is scaled. This time is overhead, because energy is still being used, but no instructions are executed. Excessive switching of frequencies or voltages might be worse than running at a suboptimal performance setting for a short period of time. Accurate prediction of the duration of the currently running task is required to decide whether the switching overhead will be amortized by subsequent savings. Lu et al. call the decision boundary for this case the *break-even time* [7]. Snowdon et al. propose penalizing a frequency switch compared to staying at the current frequency [11].

5. NOTES ON IMPLEMENTATION

Since the behavior of a single process or thread is easier to predict than the workload of the whole operating system (assuming temporal locality), analysis and prediction is usually implemented per-process. This implies modifications of the dispatching mechanism. Because task switching is implemented in software (either because the hardware does not support task switching or because hardware task switching is not used), event counters will not be reset on task switches. As a consequence, implementations need to read the current values from the configured counters during the task switch. Storing the value enables measuring the event count per task by calculating the difference between the values at dispatch and preemption. Depending on the requirements of the implemented policies and storage space available, different methods to store the event counters per process come to mind: How much history should be kept per task, if any? Are all event counts saved, or only the significant ones? Are the last few bits relevant for the power management decision, or can they be omitted?

Activities of the operating system are not exempt from event counting. This leads to the finding that task switch and performance estimation code is attributed to either process currently being switched. Depending on the overhead associated with scheduling decisions, reading the event counters twice might lead to less distorted results. Reading those machine-specific registers might come with a non-negligible overhead that has to be considered, though. Interrupts will also be attributed to the currently running task. Unlike Weißel et al. who seem to ignore this completely, Snowdon et al. later show that the effect does not have a role in the measurements for all systems they considered [11].

When implementing power management policies, the time and complexity involved in solving the optimization problem typically associated with a policy has to be considered. The overhead can be reduced using pre-computed lookup tables or mathematical reformulations, e.g., to avoid floating point operations. For example, Equation 3 can be reformulated as

$$\text{minimize} \quad (1-\alpha)\log P + (1+\alpha)\log T. \tag{10}$$

6. CONCLUSION

To reduce the energy usage and increase energy efficiency, operating systems need to be able to measure or estimate current power consumption, predict a tasks workload and control a series of power saving mechanisms. The component that decides which measures to activate in order to save power is called a *power management policy*. Due to the complexity involved in accurately estimating and predicting power consumption, today's approaches are heuristic.

Due to lots of hardware-specific settings and sensors, power management benefits from adjustment to the hardware at hand. However, this requires significant effort by users or hardware manufacturers. To do this efficiently, vendors need to provide the hardware and have to be able to modify the operating system. We see a raising number of devices, especially in the mobile market, where this is the case.

From a software developer's point of view, a task should be as homogeneous as possible to simplify workload prediction in power management policies. Measured in energy efficiency, it might be better to use a thread for a single type of workload rather than using thread pooling for different tasks.

7. REFERENCES

[1] Frank Bellosa. *The Case for Event-Driven Energy Accounting*. Tech. rep. Erlangen: Friedrich Alexander Universität, 2001.

[2] Hewlett-Packard Corporation et al. *Advanced Configuration and Power Interface Specification, Revision 5.0*. Dec. 6, 2011.

[3] Timo Hönig, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. "Extending Mobile Devices by Exploiting Remote Resources". In: *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Poster Session*. Ed. by ACM SIGOPS. Paris, France, 2010.

[4] Timo Hönig, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. "ProSEEP: A Proactive Approach to Energy-Aware Programming". In: *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012), Poster Session*. Ed. by USENIX Association. Boston, MA, USA, 2012.

[5] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. 253669-046US. 2013.

[6] ARM Ltd. *Performance Monitor Unit example code for ARM11 and Cortex-A/R*. May 2, 2013. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4237.html.

[7] Yung-Hsiang Lu and Giovanni De Micheli. "Comparing system level power management policies". In: *Design & Test of Computers, IEEE* 18.2 (2001), pp. 10–19.

[8] Akihiko Miyoshi et al. "Critical power slope: understanding the runtime effects of frequency scaling". In: *Proceedings of the 16th international conference on Supercomputing*. ACM. 2002, pp. 35–44.

[9] Trevor Pering and Robert Brodersen. "Energy efficient voltage scheduling for real-time operating systems". In: *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98, Work in Progress Session*. 1998.

[10] David C Snowdon, Sergio Ruocco, and Gernot Heiser. "Power management and dynamic voltage scaling: Myths and facts". In: *Proceedings of the 2005 Workshop on Power Aware Real-time Computing, New Jersey, USA*. 2005.

[11] David C. Snowdon et al. "Koala: a platform for OS-level power management". In: *Proceedings of the 4th ACM European conference on Computer systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 289–302.

[12] Intel Corp. Taylor Kidd. *(update) C-states, C-states and even more C-states*. Mar. 27, 2008. URL: http://software.intel.com/en-us/blogs/2008/03/27/update-c-states-c-states-and-even-more-c-states/.

[13] Andreas Weißel and Frank Bellosa. "Process cruise control: event-driven clock scaling for dynamic power management". In: *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2002, pp. 238–246.