



TECHNOLOGY WHITE PAPER

Energy Management

Gernot Heiser, PhD

Founder and Consulting Scientist, Open Kernel Labs
John Lions Chair of Operating Systems, University of New South Wales
Leader, Trustworthy Embedded Systems, NICTA

October 7, 2010

© Copyright 2010 Gernot Heiser. All rights reserved.

This publication is distributed by Open Kernel Labs, Inc. Document Number: OK 40677:2010(0)

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Permission to make digital or hard copies of this work for personal or commercial use, including redistribution, is granted without fee, provided that the copies are distributed intact, without any deletions, alterations, or additions. In particular, this copyright notice and the authorship must be preserved on all copies. To copy otherwise, or to modify, requires prior specific permission.

Authors:

Gernot Heiser, PhD

Founder and Consulting Scientist, Open Kernel Labs

John Lions Chair of Operating Systems, University of New South Wales

Leader, Trustworthy Embedded Systems, NICTA

Contact Details:

Open Kernel Labs, Inc.

200 South Wacker Drive

Floor 15

Chicago, IL 60606

USA

email: info@ok-labs.com

web: <http://www.ok-labs.com/>

1 Introduction

Battery life time is an important factor affecting the competitiveness of a mobile device. Hence, energy is a critical resource in such devices and must be managed effectively.

Energy management is a multi-faceted issue, which affects hardware, systems software and applications. In this white paper we focus on the systems software, and how it can use mechanisms supplied by the hardware in order to manage a device's energy consumption. Also, we concern ourselves only with the management of energy consumed by the processor and main memory (RAM). Device energy is an interesting topic of its own and is beyond the scope of this paper.

The most important hardware mechanisms which allow software to manage the energy consumption of the parts of the system we are interested in are *dynamic voltage and frequency scaling* (DVFS) and processor *sleep states*. Misconceptions about the effect and use of these mechanisms abound, and therefore we will discuss the fundamentals of their use for energy management in [Chapter 2](#).

Virtualization is now firmly entrenched in enterprise computing systems. There are a multitude of reasons for the strong uptake of virtualization in enterprise computing, most having to do with saving cost, and energy. However, virtualization is now also taking hold in the domain of embedded systems. Again, there is a variety of reasons, some similar to the motivations driving uptake in enterprise computing, some different. In [Chapter 3](#) we will give an overview of virtualization, particularly as it relates to embedded systems.

Energy is a global resource, and managing it locally will, in general, not produce optimal results. This means, energy management must be done by, or in cooperation with the software which controls the physical resources of a system. Normally, this is the operating system, but in a virtualized system, it is the *hypervisor*. Moreover, the hypervisor presents a level of indirection which introduces new degrees of freedom into energy management. We discuss this in [Chapter 4](#).

2 Fundamentals of Energy Management

2.1 Dynamic Voltage and Frequency Scaling

2.1.1 CMOS power

The CMOS circuits used in modern microprocessors consist of pairs of complementary (n- and p-channel) transistors connected in series. At a particular logic level, one of these is in the on, the other in the off state, resulting in minimal current flow (leakage current only). However, while switching between logic levels, temporarily both transistors are partially on, leading to a higher current flow.

The amount of charge transferred during a logic switch is approximately constant (a function of the internal capacitances of the circuitry). As switches of logic levels occur during clock ticks, the switching current, and hence the power consumed, is proportional to the clock frequency. The current is also proportional to the voltage. Put together, the power consumption due to switching of CMOS circuits, called the *dynamic power*, is given as

$$P_{\text{dyn}} = CfV^2, \quad (2.1)$$

where f is the clock frequency, V the circuit supply voltage and C is the sum of the capacitances of the clocked circuitry. This means that the power consumption of a CMOS circuit can be reduced by reducing the clock rate. But note that this is only the dynamic power. The total power consumption includes the leakage currents of the circuits, which result in a static (clock-independent) power consumption, P_{stat} , so the total power drawn by the circuit is

$$P = P_{\text{stat}} + P_{\text{dyn}} = P_{\text{stat}} + CfV^2. \quad (2.2)$$

Clocking the circuit at a lower frequency allows reducing the supply voltage. The minimal safe operating voltage of the circuit is roughly proportional to the clock rate (down to some minimal voltage which is well above the threshold voltage of silicon, 0.63 V). Where this proportionality holds, we get a cubic dependence of dynamic power on frequency,

$$P_{\text{dyn}} \propto f^3. \quad (2.3)$$

It is this strong dependence of circuit power on clock frequency which forms the basis of managing energy through DVFS.

2.1.2 Classical DVFS Model

The above theory gives us a way for reducing power consumption of a circuit (by throttling the clock, and at the same time reducing the supply voltage). However, reducing power consumption is not the objective, the costly resource is *energy*. Energy is what the data centre buys from the “power utility”, and energy is what is stored in the battery of the mobile device. Energy is what must be managed.

Reducing the power drawn by a circuit does not automatically reduce energy consumption, as a reduced clock frequency will lead to increased execution times. In order to understand how changes of the clock rate affect energy use, we need to understand how it affects the overall execution time of a computation. This is where things become interesting.

The classical assumption underlying DVFS, going back at least to Mark Weiser [1], is that *the execution time is inversely proportional to the clock frequency*, or

$$T \propto 1/f. \quad (2.4)$$

As the energy consumed by a computation is the product of power consumption and execution time, $E = PT$, under the above assumption, the dynamic energy is

$$E_{\text{dyn}} = P_{\text{dyn}}T \propto f^3/f = f^2, \quad (2.5)$$

meaning that energy consumption can be dramatically reduced by running the processor core at the lowest possible frequency.

2.1.3 Reality check: Static power

The simple model of Eq. 2.4, which is frequently assumed in academic research, has a number of problems. The first of them is that the simple model ignores static power. CMOS leakage power was low in Weiser's days, which justified ignoring it. However, in modern CMOS circuits, based on sub 100 nm processes, leakage power is significant, and can exceed dynamic power!

Taking leakage power into account can change the situation significantly. With the above assumption of $T \propto 1/f$, the total energy for a computation becomes

$$E = E_{\text{stat}} + E_{\text{dyn}} = P_{\text{stat}}T + P_{\text{dyn}}T \approx \text{const}_1/f + \text{const}_2f^2. \quad (2.6)$$

Rather than a simple monotonic dependence on f , the energy becomes a tradeoff between *decreasing dynamic energy* and *increasing static energy* when the clock is throttled. This is illustrated in Figure 2.1, which shows a hypothetical (although quite realistic) situation where static power is half of dynamic power at the highest frequency, and the frequency varies by a factor of four. It shows that energy use actually *increases* at low frequencies, and the energy cost of the computation is minimised at some intermediate frequency.

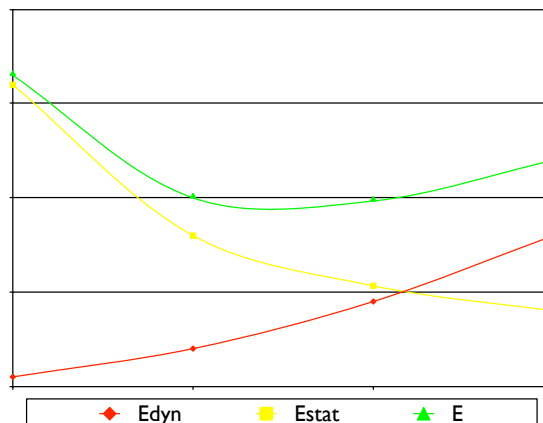


Figure 2.1. Total energy when accounting for leakage energy, as a function of core frequency.

2.2 Effect of memory

Static power is not the only spoiler for the naive run-at-lowest-frequency approach. Another is that the assumption $T \propto 1/f$ is generally not true. The reason is that modern processors, including embedded ones, are pipelined and their CPU is typically clocked much higher than main memory (RAM). This means that many instructions stall on memory transactions (mostly data loads), making their execution time essentially *independent* of the clock rate.

A more realistic model of a program's execution time represents the total execution time as the sum of the time T_{core} taken to execute instructions which only access on-chip resources (mostly the ALU and on-chip caches), and the time T_{mem} taken to access memory (load, stores and instruction fetches). The former scales with core frequency, the latter is independent of the processor clock:

$$T = T_{\text{mem}} + T_{\text{core}} = \text{const}_3 + \text{const}_4/f. \quad (2.7)$$

As the static energy of an execution is proportional to the execution time, the same non-linear relationship applies to the dependency of the execution time on the clock rate. This is shown in Figure 2.2, which indicates that the contribution to static energy from memory effects increases the energy cost of running at the lowest frequency.

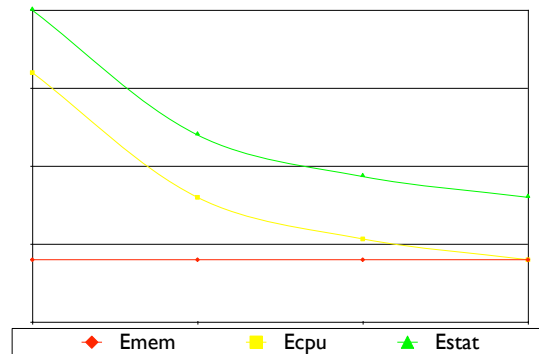


Figure 2.2. Static energy as a function of core frequency.

The energy consumed by RAM shows a very similar behaviour. RAM power also has a static component, which is independent of program execution, and a dynamic component, which is proportional to the number of memory accesses. The static power component of RAM is small, and, for present systems, negligible [2]. The dynamic RAM power, being a function of memory accesses by the core, behaves like the *static* core power, and can therefore be incorporated into P_{dyn} . This strengthens the argument that P_{dyn} is normally too large to be ignored.

2.3 Sleep states

2.3.1 Race to halt

We have shown that the approach, favoured by academic work, of minimising energy by running at lowest frequency is naive and fails when static power is significant. The opposite approach, frequently called *race to halt* and popular in industry, executes at the highest frequency until the computation is complete, and then halts the processor. This approach is based on the assumption that a halted processor consumes negligible energy, and the earlier the processor is halted, the better. Based on the arguments presented in the preceding section, a large static power favours the race-to-halt approach.

2.3.2 Powering down

This requires some explanation. Simply halting the clock would reduce the power consumption to P_{stat} which, as we argued earlier, is significant. How then can race-to-halt work?

The answer lies in *sleep states*: when the processor is halted, some or all of the circuitry can be powered off, significantly reducing static power. This results in what is referred to as a *sleep state*, where power consumption is reduced significantly below the normal static power.

Modern processors typically feature multiple sleep state, distinguished by their (static) power consumption and the cost to enter and exit them. “Shallow” sleep states can be entered and exited very quickly (typically 1–5 clock cycles) but still draw considerable power (although less than the “normal” static power). “Deep” sleep states consume considerably less power, but are expensive to enter or exit (in terms of time and energy): transitioning in or out of them can take milliseconds (millions of cycles!)

Clearly, this presents a new trade-off: Halting the processor may or may not be beneficial, depending on the duration of halting, and the depth of the sleep state chosen.

2.3.3 Counting *all* energy

If the sleep-state power is not negligible (as is typically the case in shallow sleep states which are fast to enter and exit), then it must be accounted for somehow. How?

If we can assume a fixed workload (which is the appropriate scenario in many systems, particularly mobile devices) then the correct figure of merit is normally *the energy used to execute a fixed workload over a fixed period of time*, where the system goes to sleep whenever there is no work to be done. Real-time issues are beyond the scope of this paper, so we assume that the total time period is at least as long as the complete execution time at the lowest frequency. (If this was not the case, the use of the lowest frequencies would have to be restricted, and energy management would be further complicated by the need to maintain deadlines.)

Any left-over time will simply be spent in a sleep state, so without restricting generality, we can assume that the *total time period is equal to the longest execution time*, i.e. at the lowest frequency. (Note that this ignores the fact that when running faster, the total sleep time is longer, and it might pay off to enter a deeper sleep state than otherwise. Hence our assumption might lead to a slight over-estimation of the energy used at the highest frequency.)

What this means is that for a fair comparison, we need to compare total energy for the longest execution time, T_0 . The total energy use for an execution running at a higher frequency and finishing after time $T < T_0$ will be

$$E = E_{\text{dyn}} + E_{\text{stat}} + E_{\text{sleep}} = E_{\text{dyn}} + E_{\text{stat}} + P_{\text{sleep}}(T_0 - T) \quad (2.8)$$

where P_{sleep} is the power consumed by the sleep state. In other words, we need to “pad” the execution time with the sleep time.

2.4 Beyond models: Measuring real systems

With so many assumptions involved, it is important to see how real systems behave. This requires running a range of benchmarks on a real system and measuring the actual execution time and energy usage (with a wattmeter) for a range of different settings. This has been done [3] and we summarise some of the results here.

Figure 2.3 clearly shows that for a *CPU-bound* task, the number of cycles is independent of clock frequency, which is exactly the assumption behind Eq. 2.4.

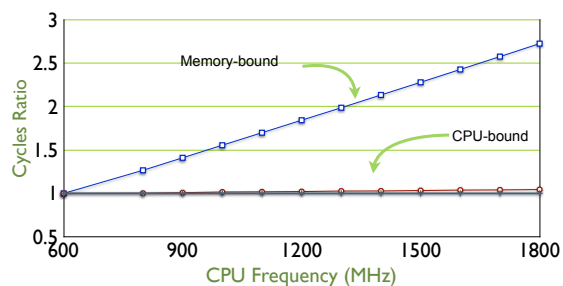


Figure 2.3. CPU cycles as a function of clock frequency for memory-bound and CPU-bound tasks.

However, we also see that a *memory-bound* task behaves completely differently: the number of clock cycles is exactly proportional to the clock frequency, which means that its execution time is *independent of the clock rate*. This is totally at odds with Eq. 2.4.

The actual energy use of the two tasks is shown in Figure 2.4. We can see that the two processes behave nowhere near as predicted by the naive model. The memory-bound process at least follows the prediction in that it consumes least energy at the lowest frequency, and most energy at the highest frequency.

It is the other way round for the CPU-bound task. Its minimal energy consumption is at the highest frequency, while at the lowest frequency its energy consumption is maximised.

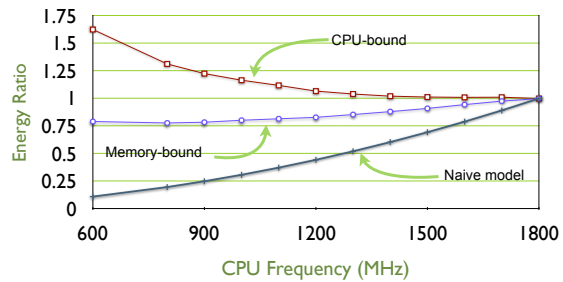


Figure 2.4. Total energy used for execution of CPU-bound and memory-bound task.

Clearly, the naive DVFS approach fails catastrophically in this case, while race-to-halt looks attractive! This is somewhat ironic, as the CPU-bound process is the one for which Eq. 2.4 holds. The reason the naive model breaks down anyway is that it ignores static energy, which is significant on this platform.

However, this is not all yet. As we argued above, for a fair comparison, we really need to compare energy use for a fixed execution time (that at the lowest frequency) and add the idle energy consumed by the faster runs. This is shown (for the CPU-bound process only) in Figure 2.5, for a number of different sleep states

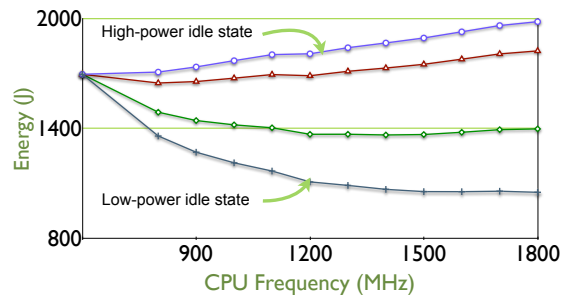


Figure 2.5. Total (padded) energy used for a fixed period for execution of CPU-bound and memory-bound task.

We can see that the depth of the sleep state matters a lot. A really deep sleep state draws almost no power, and padding makes no difference (so the lowest curve in this graph is the same as the CPU-bound curve in Figure 2.4). Race-to-halt is then the right approach for the CPU-bound task. In contrast, a shallow (high-power) sleep state consumes significant energy, and the minimal energy is achieved at the lowest frequency, as predicted by the naive model (although for the wrong reasons).

Even more interesting, intermediate sleep states result in the energy consumption being minimised at some *intermediate* frequency, which is at odds with both the naive DVFS as well as the race-to-halt approach!

Note also that the deepest sleep state shown in the graph is purely hypothetical, the actual hardware platform did not offer such a deep sleep state, and therefore run-to-halt never wins on the real hardware.

2.5 Energy management is difficult!

The above data, obtained from measurements on real systems, show that optimal energy management is far more difficult than is widely believed. In particular, we can learn the following lessons:

Naive models don't work. As we have seen, both the naive DVFS approach as well as the simple race-to-halt approach can fail catastrophically, leading to a maximisation rather than a minimisation of the energy consumed by a computation.

The policy needs to take application behaviour into account We have clearly seen that memory-bound and CPU-bound tasks behave radically differently under frequency scaling. What is good for one can be bad for the other. Tasks which are neither

completely CPU-bound nor completely memory-bound exhibit a behaviour which is somewhere between these extremes. The system must “know” what kind of application is running in order to manage energy. How?

The system must monitor workload behaviour to adjust its energy-management policy.

In an open system, such as a phone, the workloads and their behaviour cannot be determined *a priori*, and must therefore be determined dynamically, at run time. This is possible with the use of performance counters [3, 4].

Energy management must be done by, or with support of, privileged software.

This goes without saying: only the privileged software in control of hardware resources, normally the operating system (OS), can control the energy-management settings of the hardware, and collect the data needed to characterise application behaviour.

We will return to some of these points in the context of virtualization.

3 Virtualization

3.1 Basic concepts

Virtualization provides a *virtual machine*, which is an *efficient, isolated duplicate of a real machine* [5]. This allows multiple virtual machines to co-exist on a single physical machine.

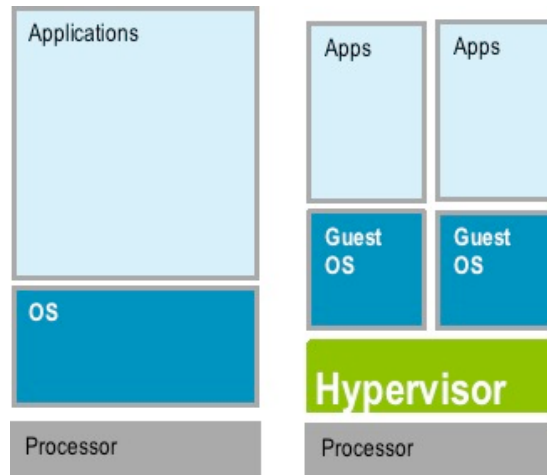


Figure 3.1. Physical machine (left) and virtual machines (right).

This is shown in Figure 3.1. Each virtual machine (VM) runs its own software stack, including an OS (called the “guest OS”) just like a physical machine. The virtual machine abstraction is provided by a software layer called the *hypervisor*.

The hypervisor is in control of all the physical resources in the system. The VM sees virtual resources, which are mapped by the hypervisor on (part of) the physical resources (CPU, memory, interrupts, devices). By controlling this mapping, the hypervisor manages the VMs and securely multiplexes the hardware resources between them. This requires that the hypervisor is more privileged than the guest OS. The latter is either completely de-privileged (running in user mode) or, if the hardware provides more than two privilege levels, may run at intermediate privilege.

The main use of virtualization is running multiple OS environments. In server virtualization, this is typically multiple copies of the same OS, each providing a different service. VMs there provide QoS isolation between these services similar to separate hardware but without the expense. This use case is called *server consolidation*.

In embedded systems, the motivation is often different. Subsystems of an embedded system need to cooperate, so QoS isolation is not normally a driver (although protecting the rest of a system from a compromised outward-facing subsystem frequently is). Virtualization in embedded systems is more frequently characterised by the co-existence of *heterogenous* OS environments, providing different aspects of the overall system mission. A typical example is the co-existence of a high-level (and frequently open) “application OS” running user-facing applications, with a real-time OS (RTOS) running time-critical code. A representative of such a scenario is a mobile phone, with a smartphone OS providing the user interface, and an RTOS supporting the modem function.

3.2 Energy management in a virtualized system

We observed in [Section 2.5](#) that the global resource energy must be managed by the most privileged software. In a virtualized system, this means the hypervisor. Individual guests do not have a global view of energy and can therefore not manage it optimally.

While the hypervisor must be involved, energy management must be all implemented inside the hypervisor. In fact, guest OS and hypervisor should cooperate in managing energy (although the guest may not realise that it is not in control). This is achieved by the hypervisor virtualising the hardware's energy management mechanisms. Each guest uses these to signal its intentions to the hypervisor, and the hypervisor uses these as inputs to its global management policies.

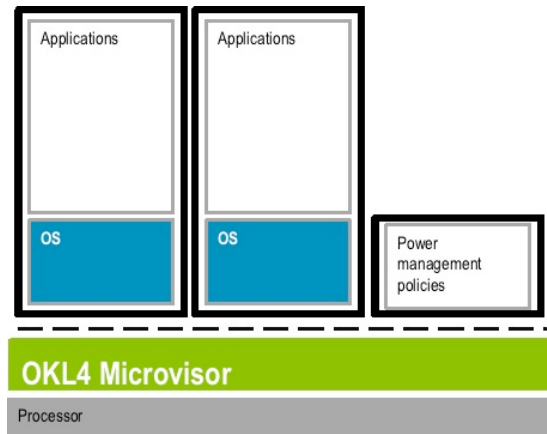


Figure 3.2. Policy-mechanism separation for managing energy of a virtualized system.

Furthermore, the policies themselves need not be implemented inside the hypervisor. In fact, it is preferable to keep policies completely out of the hypervisor. Instead, management should be left to an unprivileged *policy module*, which is given access to the guests' intentions as well as the actual energy-management mechanisms of the hardware. This is shown in Figure 3.2, and is the approach used by the OKL4 Microvisor, the virtualization solution from Open Kernel Labs.

4 Energy Management in Multicores

Multicore processors are now appearing in embedded systems. They provide new challenges as well as opportunities for energy management, and virtualization provides an excellent way of dealing with them.

For one, virtualization allows transparent migration of activities off lowly-loaded cores, reducing the number of active cores and transitioning unneeded ones into (deep) sleep states. This is particularly attractive where subsystems are mapped to their “own” cores. The hypervisor can detect when the total load of a subsystem is low enough to consolidate it onto a shared core (and reversing this when processor demand increases). This is shown in Figure 4.1.

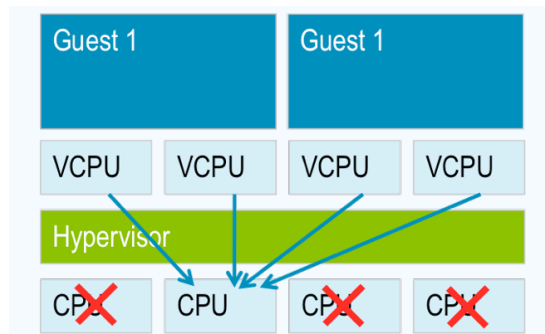


Figure 4.1. Managing energy by temporary core consolidation.

Of course, the same could be achieved in the OS (assuming that all cores are under control of the same OS). However, there are several reasons why it is preferable to run a hypervisor on multicore processors.

Firstly, existing OSes are large, monolithic pieces of software, which are difficult to adapt to new challenges, such as shutting down individual cores for energy management. Furthermore, manufacturers often have to deal with multiple OSes for different segments of their product portfolio. For example, many manufacturers of smartphones sell devices running Android as well as Windows- or Symbian-based handsets. This means that multiple OSes must be adapted to new scenarios. Alternatively, virtualising all systems allows adding support for multi-core energy management to the (single) hypervisor, which is simpler and less expensive.

Secondly, with a growing number of cores we can expect that designers will tie individual cores to particular subsystems. These need to run their own OS environment. In such a case, the hypervisor not only protects subsystems from faults in other subsystems. It also becomes the only software with the global knowledge required for effective energy management.

Finally, we can expect that in the medium term, systems will not support per-core DVFS. The reason is that the implementation of cache coherence across multiple cores becomes complex and expensive if those cores run at different clock frequencies. While a number of recent ARM multicores do support per-core DVFS, we consider it highly likely that with increasing number of cores, at least groups of cores will run at the same frequency, a trend which is already visible in server platforms. This again requires an entity with a global view of energy use in order to determine an optimal setting.

5 Conclusions

We have discussed the challenges of managing energy use in modern open systems. We have found that simple models, such as always running at the lowest possible frequency (popular in academia) or race-to-halt (frequently deployed in industry), generally do not work. Effective energy management requires real-time characterisation of workloads, and an understanding of hardware-specific tradeoffs.

We have also found that energy management must be done at (or in collaboration with) the most privileged software layer. We found that there are several reasons why it is best done in a global policy module, with support of an energy-aware hypervisor. There are good reasons why in the future of multicores, this is the *only* effective way to manage energy.

Bibliography

- [1] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, CA, USA, November 1994.
- [2] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010.
- [3] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, April 2009.
- [4] Andreas Weissel and Frank Bellosa. Process cruise control—event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Grenoble, France, October 8–11 2002.
- [5] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.