Understanding and Reducing Smartphone Energy Consumption

Aaron Carroll

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

University of New South Wales

Sydney, Australia

May 2017

THE UNIVERSITY OF NEW SOUTH WALES

Thesis/Dissertation Sheet

Surname or Family name: Carroll	
First name: Aaron	Other name/s:
Abbreviation for degree as given in the University calendar: PhD	
School: Computer Science and Engineering	Faculty: Engineering
Title: Understanding and reducing smartphone energy consumption	

Abstract 350 words maximum:

Modern smartphones are increasingly performant and feature-rich, but because they are battery powered, remain highly power-constrained. Energy management is the art and science of maximising battery lifetime, and effectively doing so requires a solid understanding of how a devices uses energy to inform policy and algorithms backed by accurate data. This work addresses each of these issues.

First, we present a detailed power analysis of two smartphones, the Openmoko Freerunner and the Samsung Galaxy S III. We measure power consumption by direct instrumentation at the circuit level by interposing on the power supplies of the individual components, including CPU, RAM, display, GPU, wireless radios, camera, GPS, storage, audio, and environmental sensors. With this instrumentation in place, we produce breakdowns of how energy is distributed under micro-benchmarks and realistic usage scenarios. We also measure two other devices at the whole-system level to validate our earlier results, and to draw conclusions about how smartphone power consumption is changing over time. Additional to the results presented, we also describe a methodology for instrumenting commercial mass-market off-the-shelf devices.

Based on these measurements we observe that peak CPU energy consumption is increasing due to the advent of multi-core processors in the mobile segment. Thus, effective power management of these will be important for battery life on future mobile devices. Such multi-core processors add a new dimension, the number cores active, to the spectrum of available energy management mechanisms.

In the second part of this work we investigate how this mechanism, which we call core *offlining*, interacts with the well-established technique of dynamic voltage and frequency scaling (DVFS) for minimising power consumption. We find surprising differences in the characteristics of contemporaneous smartphones, specifically in the importance of static power, which we show to be a critical factor in minimising energy consumption. We design and implement *medusa*, a policy that exploits our findings to integrate core offlining with DVFS in the Linux kernel. We show that despite its simplicity, medusa obtains energy savings that are at least as good as, and often better than, the algorithms that ship on the studied phones, and that it approaches the optimal static algorithm.

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed ………………………………………………..........................

Date ………………………………………………..........................

ORIGINALITY STATEMENT

Abstract

Modern smartphones are increasingly performant and feature-rich, but because they are battery powered, remain highly power-constrained. Energy management is the art and science of maximising battery lifetime, and effectively doing so requires a solid understanding of how a devices uses energy to inform policy and algorithms backed by accurate data. This work addresses each of these issues.

First, we present a detailed power analysis of two smartphones, the Openmoko Freerunner and the Samsung Galaxy S III. We measure power consumption by direct instrumentation at the circuit level by interposing on the power supplies of the individual components, including CPU, RAM, display, GPU, wireless radios, camera, GPS, storage, audio, and environmental sensors. With this instrumentation in place, we produce breakdowns of how energy is distributed under micro-benchmarks and realistic usage scenarios. We also measure two other devices at the whole-system level to validate our earlier results, and to draw conclusions about how smartphone power consumption is changing over time. Additional to the results presented, we also describe a methodology for instrumenting commercial mass-market off-the-shelf devices.

Based on these measurements we observe that peak CPU energy consumption is increasing due to the advent of multi-core processors in the mobile segment. Thus, effective power management of these will be important for battery life on future mobile devices. Such multi-core processors add a new dimension, the number cores active, to the spectrum of available energy management mechanisms.

In the second part of this work, we investigate how this mechanism, which we call core *offlining*, interacts with the well-established technique of dynamic voltage and frequency scaling (DVFS) for minimising power consumption. We find surprising differences in the characteristics of contemporaneous smartphones, specifically in the importance of static power, which we show to be a critical factor in minimising energy consumption. We design and implement *medusa*, a policy that exploits our findings to integrate core offlining with DVFS in the Linux kernel. We show that despite its simplicity, medusa obtains energy savings that are at least as good as, and often better than, the algorithms that ship on the studied phones, and that it approaches the optimal static algorithm.

Acknowledgements

I would like to thank my advisor, Gernot Heiser, for his support, guidance, honesty, and heroic patience that made this work possible. Being his student was challenging and rewarding in many ways, both academic and personally, and for that I am indebted to him.

Thanks also to my friends and colleagues at KEG, past and present, who made it such an enjoyable place to be. Special thanks Ali, Bernard, Etienne, Godfrey, Kevin, Leonid, and Peter.

For their motivation, love, and support, thanks to Cecile, Emily, Michael, and my family Pam, Peter, Sheryn, and May.

Publications

Aaron Carroll and Gernot Heiser. **An analysis of power consumption in a smartphone.** In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, Boston, MA, USA, June 2010. *1000+ citations.* [CH10]

Aaron Carroll and Gernot Heiser. **Mobile multicores: Use them or waste them.** In *Proceedings of the 5th Workshop on Power Aware Computing and Systems (HotPower)*, Farmington, PA, USA, November 2013. Reprinted in *ACM SIGOPS Operating Systems Review (OSR), 48(1) pp. 44–48, May 2014.* [CH13a]

Aaron Carroll and Gernot Heiser. **The systems hacker's guide to the Galaxy: Energy usage in a modern smartphone.** In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys)*, Singapore, July 2013. *Awarded best student paper.* [CH13b]

Aaron Carroll and Gernot Heiser. **Unifying DVFS and offlining in mobile multicores.** In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, April 2014. [CH14]

Robert LiKamWa, Zhen Wang, Aaron Carroll, Felix Xiaozhu Lin, and Lin Zhong. **Draining our glass: An energy and heat characterization of Google Glass.** In *Proceedings of the 5th Asia-Pacific Workshop on Systems (APSys)*, Beijing, China, June 2014. [LWC$^+$14]

Contents

Chapter 1

Introduction

Mobile devices derive their energy supply from batteries. For many such devices, particularly mobile phones, energy capacity is severely restricted due to size and mass constraints on the device's form factor, cost of manufacture, and safety and environmental concerns on the battery technology used. Therefore, the careful expenditure of this precious resource is critical to the utility and user experience of mobile devices.

In stark opposition, the functionality, compute throughput, storage capacity, and networking speed of these devices is increasing exponentially. From 1990 to 2003, processor speed, disk capacity, and memory size each increased by more than 200-fold, whereas battery energy density merely tripled [PS05]. In additional to basic telephony, modern high-end mobile phones include such functionality as audio and video streaming, playback, and capture, GPS navigation, 3D and virtual-reality gaming, email and web browsing, and so on. These devices, typically referred to as *smartphones*, are placing increasing pressure on energy consumption and thus battery lifetime. This deepens the need for intelligent and effective energy management.

Energy management is the art and science of using the available energy supply as efficiently as possible, that is, maximising the work done per unit energy consumed. The operating system (OS) plays a crucial role in support of this goal as the implementor of policy around power management mechanisms exposed by the hardware. The OS also manages scheduling of work, such as compute tasks on the processor or packets on the network interface, and thus sets policy for how and when work is presented to the hardware.

In implementing and configuring such policies, it is important to understand how energy is used in a device; specifically, how much energy each component consumes, and under what circumstances. For example, if we can reduce the accuracy of a particular sensor (say, a compass) in order to achieve a 50 % reduction in its power draw, should we make that trade-off? While a halving of power sounds like an attractive optimisation, this in fact would be a poor decision: the vast majority of the energy cost of sampling a sensor is in fact consumed by the processor and not the sensor hardware. Intelligent power management *demands* accurate data.

In this work, we set out to achieve two goals. Firstly, to understand the energy consumption characteristics of modern smartphones with typical workloads and applications, using direct measurement at the finest possible granularity. Secondly, based on observations made in our measurement study, we aim to reduce energy consumption for a particular type of processor typical in current and emerging smartphones: the mobile multi-core.

Measuring smartphone power

Assumptions about smartphone energy consumption are all too often based on intuition or guesswork. For instance, it is often assumed that reducing power is equivalent to reducing energy, when in fact this also depends on factors such as time. For example, executing a compute workload on one processor instead of two significantly reduces the power drawn, but as we show in this work, the net result is often to *increase* the total energy used. Our intuition also tends to fail in approximating the energy consumption of certain tasks. For instance, making a phone call is generally considered an energy intensive workload, but as we will show, on modern devices this ranks among the lowest-energy use cases.

We aim to provide a corpus of ground-truth to this discussion, both in terms of where and how energy is consumed in smartphones. The approach we take is electrical measurement, which involves instrumenting a device with a power meter, then executing certain workloads while collecting power samples in real time. Compared with modeling, the other main approach to estimating energy consumption, instrumentation has several advantages. It is more accurate since no prediction is required, and it is free from observer effects because the software remains unmodified and only passive hardware modifications are required.

Others have undertaken similar endeavours, but these works are limited in several ways. In most cases, the analyses are based on power measurements of the whole system made by instrumenting the power supplied from the battery. This is a coarse-grained approach, in the sense that attributing energy consumed to specific components of the device, such as the processor, memory, wireless interfaces, etc., can only be done indirectly. Smartphones however have many components and their energy consumption is inter-dependent. For example, sending data on a WiFi network consumes energy in the processor to drive the network stack and application, in memory to load and store the associated data, and in the WiFi modem to send and receive data over-the-air. A whole-system approach is unable to differentiate the energy consumed by each of these components.

In contrast to existing works, we perform a fine-grained instrumentation which is able to separately sample power of most of a device's main components. We do this by adding measurement circuitry to the internal power distribution network. Specifically, we insert current-sense resistors between the individual power supplies and the components they power. These resistors are then connected to a data acquisition unit, which is essentially an accurate, multi-channel power meter. This is connected to a desktop computer running our sampling software, which collects the power measurements in real-time and writes them to a file for later processing.

We perform this instrumentation on two devices. On the first, the Openmoko Freerunner, this is a relatively straightforward task due to the availability of documentation, which simplifies identification of power supplies, and the hardware design, which provides convenient instrumentation points. On the second device, the Samsung Galaxy S III, it is a more complicated process. However, we develop a methodology for instrumenting commercial off-the-shelf devices, based on a common design feature of efficient power supplies.

With our instrumented devices, we are able to directly measure the power consumption of the processor, memory, display, graphics unit, wireless interfaces, sensors, and others. We then analyse energy consumption under a range of realistic workloads, including web browsing, sending and receiving email, audio and video playback, still and video photography, gaming, text messaging, and telephony, as well as various idle states. We also perform micro-benchmarks to better understand behaviour of particular components, including sensors, GPS, display, storage, and networking.

In addition to the fine-grained analysis of the Freerunner and Galaxy devices, we also measure whole-system energy consumption for two other devices, the HTC Dream and the Google Nexus One, under similar workloads. This serves two purposes. Firstly, it provides a degree of validation for our earlier results, i.e. that the results apply to a wider class of devices than just the two we measured. Secondly, it allows us to analyse how energy consumption of smartphones has changed over time. The four devices we use represent 5 years of technology, which is sufficient to draw conclusions about trends in smartphone technology.

Reducing multi-core processor power

As a result of our energy consumption analysis, we observe an interesting trend. The compute *capacity* of a smartphone (i.e. the maximum throughput), and in turn the maximum power, have increased significantly, yet the average used in practice has not increased by the same degree. This is a kind of analog to the so-called *dark silicon* problem, which is that compute capacity is exceeding thermal capacity. In other words, the devices are becoming more capable, but less utilised. As a consequence, the efficiency of intermediate power and performance states is becoming increasingly important to optimising a device's overall energy consumption.

Two factors are largely responsible for this trend. Firstly, processors in the embedded space have continued to scale *up*—running at higher clock speeds, and doing more work per clock cycle. However, we have begun to see them also scale *out*, with the introduction of the multi-core processor in the mobile sector. Such processors feature multiple execution units which can run tasks truly in parallel.

The multi-core process also introduces new complexities and opportunities for power management. In a single-core system, one of the main power management mechanisms is *dynamic voltage and frequency scaling* (DVFS). This allows the OS to reduce the clock speed (and thus performance) of the processor, which reduces the power drawn. Because power is super-linear in frequency, if used correctly, DVFS can reduce total energy consumption. The multi-core processor introduces a new dimension that can be controlled: the number of online cores. If more compute throughput is required, two options are available: either a new core can be onlined, or the frequency of existing cores can be increased. The optimal decision depends on the power–performance trade-offs of the two mechanisms.

In the second part of this work, we investigate the trade-off between the frequency and number-of-cores axes of power management. More specifically, we investigate these in the context of *idle* power management, that is, how to most efficiency execute a workload which under-utilises the processor capacity but without affecting performance.

We first run a series of micro-benchmarks to investigate the problem space, measuring power while varying the compute load, number of cores, and core frequency. We also model this response mathematically in order to understand the parameters that affect processor power. We find that per-core static power—the cost of onlining a core, regardless of the work it performs—is an important and differentiating factor among mobile multi-cores.

For processors with low per-core static power, we find that the optimal policy is to always online cores where possible, rather than increasing frequency, since there is no "penalty" to having additional cores running, even if they are under-utilised. However, when static power is non-trivial, a policy must weigh the cost of onlining a core against the cost of increasing frequency. We find that a good model for such a policy is to define a *threshold frequency* which divides the optimisation space into two regions: one where increasing and decreasing frequency is optimal, and one where onlining and offlining cores is optimal. The threshold frequency is device-dependent, and we show how to determine its value both theoretically and experimentally.

Using the insights above, we design and implement *medusa*, a power management policy for Linux. Medusa controls frequency and number of cores to reduce energy consumption without affecting performance, and we implement it on two devices which have different per-core static power characteristics. The first, based on the Samsung Exynos processor, has very low static power, so we configured a threshold frequency of zero. The second device is based on the Qualcomm Snapdragon chip, which has higher per-core static power. We use a model of both the device's processor and our policy to predict the optimal threshold frequency.

We measure the energy consumption of medusa on each device under a number of scenarios, including micro-benchmarks of varying load, video playback, and two mobile benchmarking applications, *AnTuTu 3DRating* and *Vellamo HTML5*. We compare medusa against other policies, such as the defaults that ship on our test devices, standard Linux policies, and, where possible, the optimal policy. In addition to power, we also use the performance metric of the benchmarking applications to quantify the effect our policy has on execution time. Finally, we perform a sensitivity analysis to determine the effect of our chosen threshold frequency on energy consumption.

Summary

We make two main contributions in this work:

- a study of the energy consumption of several smartphones; and
- the design and implementation of a DVFS control algorithm for multi-core processors which incorporates knowledge of offline power states.

Our study of smartphone power leads us to the conclusion that managing intermediate power states is critical to future energy efficiency, due to the growing disparity between average and peak throughput and power. One such driver for this trend is the emergence of the mobile multi-core processor. We perform an analysis of such devices and discover that per-core static power can vary significantly, and that it is a critical factor in implementing efficient power-management policies. We implement such a policy and show that it performs favourably compared with existing policies.

Structure

This remainder of this work is structured as follows. In Chapter 2 we provide the necessary background, including introductions to energy consumption concepts, processor technology, power management mechanisms, and multi-core processors. In Chapter 3 we discuss existing literature, dividing our attention between power measurement and analysis works, and those focused on power management. Then, in Chapter 4, we discuss our work on profiling smartphone energy consumption. Chapter 5 contains a discussion of our power management work, including the design and implementation of the medusa policy. Finally, in Chapter 6, we conclude with a summary of our work, including a discussion of limitations and future work.

Chapter 2

Background

2.1 Power, Energy and Performance

A computer is a device that produces heat and radiation, performing calculations as a side-effect. The input to this process is energy, originating either as chemical energy in a battery, or electrical energy from a mains supply. This energy supply is limited, and must be used wisely: this is the art of *power* or *energy management*. While often used interchangeably to refer to roughly the same ideas, these are different quantities: power, measured in watts (W), is the time rate-of-change in energy, measured in joules (J). Equivalently, energy is the integral of power over time:

$$P = \frac{\mathrm{d}E}{\mathrm{d}t} \quad [\mathrm{W}] \qquad \text{or} \qquad E = \int P \,\mathrm{d}t \quad [\mathrm{J}]\,.$$

While power is an "instantaneous" quantity, average power over some period T is often used:

$$E = P_{\text{average}} T\,.$$

Power/energy management encompasses two distinct but related goals:

- Minimise the energy consumed in performing a computation, perhaps satisfying some performance requirement. This is desirable because batteries are limited in capacity: reducing energy consumption increases the useful life of portable devices. Moreover, the production and storage of energy is environmentally damaging, and lowering consumption reduces the financial costs associated with producing the input energy and removing the output energy.

- Constrain the instantaneous (or short-term average) power draw to comply with temperature limits and/or power supply maxima. This is generally referred to as *thermal* management.

In this work, we will use the term "energy management" to refer specifically to the first sense, and "thermal management" for the second sense. Energy management approaches can be further categorised into two types: *idle* and *active*.

Idle An idle energy management policy aims to reduce energy consumed without affecting performance, by exploiting the difference between the computational resources available and the demand of an application. This difference is called "slack", so the technique is also known as *slack management*.

Active An active policy makes the concession of degraded performance in exchange for reduced energy. Usually the two dimensions are weighted according to some policy: energy-delay product (EDP), energy-delay-squared (ED^{2}P), minimal energy, and bounded-degradation are common.

2.2 CMOS Primer

The basic building block of modern digital logic circuits, such as computer processors, is the *metal-oxide-semiconductor field-effect transistor*, or *MOSFET*: a device which acts logically as a switch. When the input is above a certain voltage the transistor is "on," otherwise it is "off." There are two important types of MOSFET, known as *n-channel* and *p-channel*.

An n-channel MOSFET consists of two pieces of highly-doped n-type semiconductor (denoted "n+") called the *source* and *drain*, separated by a piece of p-type semiconductor, called the *body* or *substrate*. The fourth terminal, called the *gate*, is separated from the body by a dielectric (non-conducting) oxide layer. With no external voltage applied, no current can flow between the source and drain terminals, since the p-type body is depleted of the n-type majority carriers (i.e. electrons) of the source and drain. However, when the gate–body voltage is sufficiently large, above the threshold voltage V_{th}, an *inversion layer* of n-type carriers is produced in the space between the source and drain, called the *channel*, which allows current to flow from drain to source. Figure 2.1 shows the topology of an n-channel MOSFET.

A p-channel device operates similarly; the source and drain terminals are composed of a doped p+ semiconductor and the body an n-doped material which is thus depleted of p-type carriers (i.e. holes). For a p-channel MOSFET, the threshold voltage is negative, and when the gate–body voltage drops below this level, an inversion layer of p-type carriers is produced at the channel allowing current to flow between source and drain. While the MOSFET is actually a 4-terminal device, the source and body terminals are usually shorted internally, so in a circuit schematic appears as a 3-terminal symbol, as shown below in Figure 2.2.

Basic logic gates (AND, OR, NOT, etc.) can be constructed from pairs of MOSFETs, one n-channel and one p-channel, in a technology called *CMOS*: *complimentary metal-oxide-semiconductor*. For example, an inverter, or logical NOT gate, can be created from a single

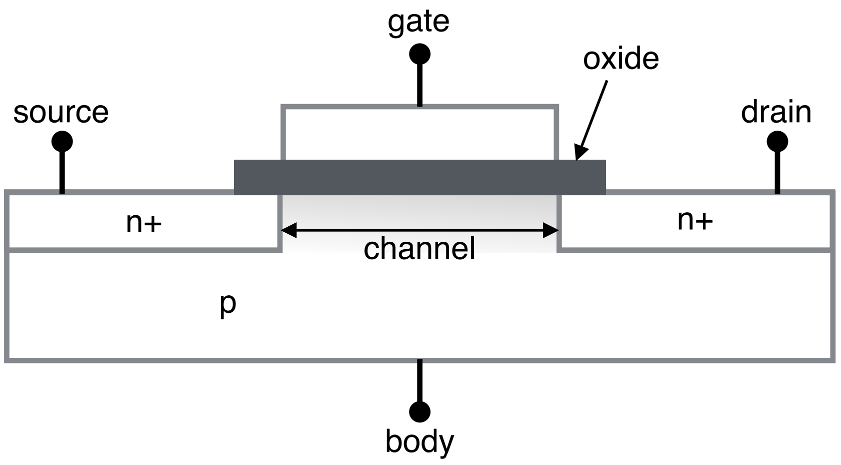


Figure 2.1: Cross-sectional topology of an n-channel MOSFET.

CMOS pair, as shown in Figure 2.2. When the input signal is "high" (i.e. $V_{\text{in}} = V_{\text{dd}}$), the p-channel MOSFET is off so its source–drain is an open circuit, whereas the n-channel is on, pulling its drain, and hence the output (V_{out}), to ground ("low"). When the input is low, the reverse occurs, and the output is "high."

The complimentary arrangement of CMOS pairs means that one MOSFET from each pair is always off, and hence ideally, a CMOS circuit consumes zero energy because there is never a path for current to conduct between the supply voltage V_{dd} and ground. However, the MOSFET is fundamentally an analog device, and so while it acts *logically* as a switch, its behaviour deviates from the digital ideal when considered as an electronic circuit. In particular, there are two classes of energy consumption: *dynamic* power, the cost to switch the state of a gate from "on" to "off" (or vice versa), and *static* power, the constant loss from applying a voltage to the circuit to maintain its current state.

2.2.1 Dynamic power

Dynamic power is the component of CMOS energy consumption due to the switching of transistor states, and is thus a function of frequency. It is caused by two phenomena: *switching* power and *short-circuit* power.

Switching power is due to the parasitic (i.e. unintentional and undesired) capacitance seen between the gate and source/drain terminals of each MOSFET. To change the voltage at the gate, this capacitance must be "charged" which necessitates a certain amount of current flow, and thus, the input of energy. Switching power can be derived as follows. The energy con-

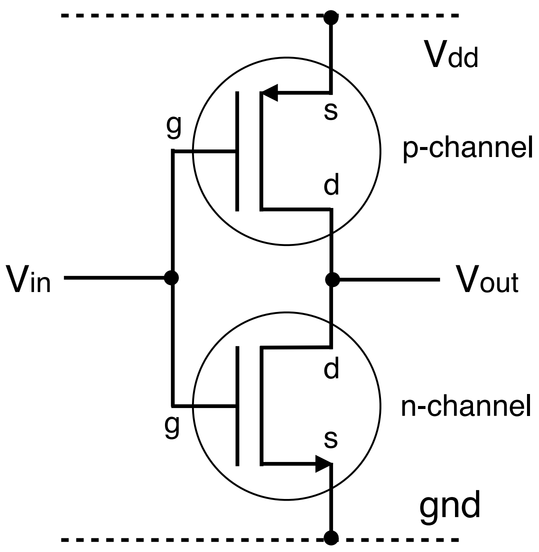


Figure 2.2: CMOS inverter schematic. The inverter is made up of a single MOSFET pair: one p-channel (top) and one n-channel (bottom).

sumed in an arbitrary circuit can be determined by

$$E = \int_0^\infty i(t)v(t)\,\mathrm{d}t\,,$$

where $i(t)$ and $v(t)$ are the instantaneous current and voltage at time t, respectively. (As a matter of notational convention, lower-case v, i are used for the instantaneous, time-varying quantity, whereas upper-case V, I are used for constant or average values.) The fundamental capacitor equation relates the current to the capacitance (C) and voltage as

$$i(t) = C\frac{\mathrm{d}v}{\mathrm{d}t}\,,$$

and so the energy consumed in switching a CMOS circuit with equivalent capacitance of C farads from a supply voltage of V_{dd} is

$$E = \int_0^\infty C\frac{\mathrm{d}v}{\mathrm{d}t}V_{\mathrm{dd}}\,\mathrm{d}t = CV_{\mathrm{dd}}\int_0^{V_{\mathrm{dd}}}\mathrm{d}v = CV_{\mathrm{dd}}^2\,.$$

(As an aside: the energy *stored* in a capacitor is $\frac{1}{2}CV^2$, so charging a capacitor with a *fixed* voltage is at best 50 % efficient.) For a circuit switching at frequency f, this charge/discharge cycle occurs f times per second, so using $E = Pt$, the average power can be expressed as

$$P = CfV_{\mathrm{dd}}^2\,.$$

In typical operation however, only a subset of gates are switched on each clock cycle. This fraction is called the *activity factor*, denoted by α. Multiplying the capacitance C by the activity factor α yields a quantity called the *effective capacitance*, C_{eff}, which is used in the common expression for switching power

$$P_{\text{switching}} = C_{\text{eff}} f V_{\text{dd}}^2 \,.$$

Short-circuit current, the second form of dynamic energy consumption, is due the imperfect relationship between gate voltage and source-drain resistance. Logically, we model this relation as a delta function: the switch is fully open (i.e. infinite resistance) when the gate voltage is above the threshold and fully closed (i.e. zero resistance) when gate voltage is below threshold. However, in practice the MOSFET has three so-called operating regions, depending on whether the gate voltage is: (1) close to zero (the “sub-threshold” region); (2) near but above the threshold voltage (the “linear” region); and (3) near the supply voltage (the “saturation” region).[1] Since switching is not instantaneous, there is a brief period during which both transistors in the CMOS pair are operating in the linear region, temporarily short-circuiting the supply voltage to ground. Current that flows during this time is lost as short-circuit energy. The expression for short-circuit current is reasonably complex, but in the worst-case can be expressed as

$$P_{\text{short-circuit}} \propto f\,(V_{\text{dd}} - 2V_{\text{th}})^3 \ ,$$

where V_{th} is the transistor threshold voltage [Vee84]. In practice, $P_{\text{short-circuit}}$ is significantly less than $P_{\text{switching}}$, no more than 20 % [Vee84, NS00], so short-circuit power is often folded in to the constants for switching power, and total dynamic power simplified to

$$P_{\text{dynamic}} \propto f V^2 \,. \tag{2.1}$$

2.2.2 Static power

Static energy consumption is due to “leakage” current which encompasses several different phenomena whereby current flows between the source, drain, gate and substrate of a MOSFET whenever a supply voltage is applied. In current process nodes ($\approx$ 25 nm), there are three significant forms of leakage current, each of which contributes roughly equally:

Gate leakage The gate terminal is insulated from the source, drain, and substrate by a dielectric (non-conductive) layer. To maximise transistor density, this layer is thin, and this allows electron tunneling whenever a voltage differential is present.

Sub-threshold leakage When gate voltage is below the threshold (i.e. the transistor is op-

[1]Signal amplification, the other main application of MOSFETs, makes use of the linear operating region, but for CMOS digital circuits it is merely an inconvenience.

erating in the sub-threshold region), resistance between source and drain is nominally infinite. However, when a source–drain voltage is present, a small "diffusion" current continues to flow due to a concentration of minority charge carriers which accumulate at the surface of the channel. These are produced by material impurities and input of heat and radiation which causes spontaneous elevation of electrons to the conduction band. Majority carriers, which are introduced intentionally by doping, constitute the main "drift" current when the transistor is switched on and are opposite in polarity to the minority carriers.

Reverse-bias current The drain–substrate and source–substrate interfaces form P-N junctions. When a reverse-bias voltage is applied to a P-N junction with a high doping density (such as in a MOSFET), a particular type of electron tunneling occurs, called band-to-band tunneling (BTBT), whereby valence electrons jump both across the junction and into the conduction band. There is also a small diffusion current between drain/source and substrate due to minority charge carriers, but BTBT dominates the total reverse-bias current.

Note that while reverse-bias and gate leakage flow consistently whenever a supply voltage is applied, sub-threshold leakage only occurs when the transistor is off. When turned on, the diffusion current flows in the same direction as the drift current and thus is not lost energy.

Each of these sources of static energy consumption are increasing over time as transistor size decreases [MRR05, RMMM03, KAB$^+$03]. The exact magnitude of each is a complex function of transistor geometry, material, temperature, and supply voltage. After fabrication, geometry and material composition are immutable. Thus, for a given circuit, the variables available for managing static leakage power are temperature and supply voltage.

Gate leakage is independent of temperature, while reverse-bias leakage is linear (with a shallow gradient) and sub-threshold leakage is exponential. However, in realistic temperature ranges, the overall temperature response is very flat [MRR05], increasing only 30 % from 300–400° K. In terms of supply voltage, reverse-bias current is weakly correlated, whereas gate and sub-threshold leakage grow exponentially, particularly in the 0–1.5 V range where modern CMOS circuits operate [KADB02]. Thus, supply voltage control is the main mechanism for managing static energy consumption.

The relative magnitude of the two sources of energy consumption depends on many quantities, and indeed they are interdependent. For modern processors, both must be addressed as significant contributors, in contrast to early power management work which largely ignored the static power contribution. While perfectly acceptable at the time, this simplification no longer represents reality, despite its prevalence even in the modern literature.

2.2.3 Switching frequency

While a lower source voltage reduces static power, it also limits the switching frequency (i.e. speed) of the circuit. Due to the inherent capacitance and resistance of a CMOS gate, it takes a certain amount of time (the *rise time*) for the transistors to switch between the on and off states. Driving the transistor with a higher source voltage reduces the rise time, so the switching frequency can be increased. The maximum frequency for a CMOS circuit at supply voltage V_{dd} is

$$f \propto \frac{(V_{\mathrm{dd}} - V_{\mathrm{th}})^{\beta}}{C V_{\mathrm{dd}}}, \tag{2.2}$$

where β is a constant in the range 1.4–2.0 [YWV+05].

For a fixed micro-architectural design, the throughput of a processor is primarily determined by the switching frequency. Increases in speed have been largely driven be reducing transistor gate length, which affords increased frequency by several mechanisms. Firstly, the threshold voltage can be reduced, which per Equation 2.2 increases the peak frequency [MRR05]. Secondly it reduces the gate capacitance which improves the rise time of the transistors. Finally, smaller transistors allow a commensurate increase in density, which reduces the effect of clock drift.

In addition to speed increases, smaller transistors also reduce dynamic power (for a given frequency) due to the reduction in gate capacitance. However, they cause an *increase* in static power: thinner gate oxides increase tunneling, higher doping density increase BTBT, and so-called "short-channel" effects increase sub-threshold leakage. Thus, as switching speeds have increased, so has the ratio of static to dynamic power. However, dynamic power grows very quickly with frequency in typical voltage and frequency ranges and continues to dominate at very high frequencies.

2.2.4 Temperature

The energy consumption of a CMOS circuit is dependent on its temperature T, which is both a property of the environment (i.e. ambient temperature) and a side-effect of its operation. As discussed above, static power is exponential in temperature, specifically in sub-threshold leakage, because heat produces minority carriers. Nonetheless, the curve is reasonably flat in realistic operating conditions. Dynamic power and gate leakage, on the other hand, are independent of T [LHL05, MRR05]. While temperature control is therefore only moderately important in *directly* reducing overall energy consumption, there are a number of secondary effects which can impact energy.

CMOS circuits have an upper limit on temperature, the *maximum junction temperature*, above which temporary malfunction or permanent damage can occur, for example, by breakdown of the gate oxide layer. Processor packages usually include a temperature sensor to detect and prevent such situations from occurring. If the measured temperature approaches a predetermined safe upper limit, known as a thermal emergency, the processor is slowed (or stopped completely) to avoid damage. This reduces the performance of the processor, and thus can impact total energy consumption.

Temperature management is also important to other aspects of system design, particularly reliability [SABR04] and safety [LWC+14], and is generally achieved by cooling and power management. Cooling dissipates heat into the environment, either passively via a heat sink, or actively with cooling fans. Power management modulates the circuit frequency and utilisation to reduce the power output and thus temperature.

2.3 Power Control Mechanisms

From Section 2.2 it is clear that many parameters affect CMOS power. Most of these however are properties of the transistor material and geometry and therefore immutable after fabrication: gate width and length, capacitance, doping profile, threshold voltage, oxide thickness, etc. From a systems perspective, the only parameters available to control are supply voltage, V_{dd}, and frequency, f, from which several power control mechanisms can be constructed:

Frequency scaling Per Equation 2.1, the dynamic component of power can be controlled linearly by scaling the operating frequency f. Static (i.e. leakage) power remains unaffected.

Voltage and frequency scaling According to Equation 2.2, reducing frequency also allows the operating voltage V_{dd} to be scaled. This reduces both dynamic and static power, which are super-linear in voltage.

Clock modulation The effective operating frequency of a circuit can be reduced by omitting some clock edges without changing the source clock frequency. This is known as clock *modulation*, and can be seen as equivalent to reducing the activity factor. Compared with frequency scaling, clock modulation has a very simple implementation and can thus switch quickly between effective frequencies. However, it can not be accompanied by a reduction in V_{dd} because the maximum rise time is unchanged.

Clock gating Dynamic power can be reduced to zero by removing (i.e. *gating*) the clock supply completely, effectively reducing frequency to zero and stopping all transistors from switching, but resulting in loss of functionality. Clock gating does not affect static power, which is independent of frequency.

Power gating By completely removing the supply voltage, known as *power gating*, both dynamic and static power are reduced to zero. This also results in complete loss of state, since CMOS is volatile, i.e. transistor "on" or "off" status is lost when no source voltage is applied.

When a mechanism can be enabled and/or configured online, that is, during circuit operation, it is called a *dynamic* mechanism. For example, dynamic frequency scaling allows the circuit frequency to change, and dynamic clock modulation allows controlling the duty cycle, while the circuit is running. A typical processor will utilise some or all of these mechanisms in various forms to control energy consumption. Some of these are automatic and transparent to software, while others are made explicitly available via an API, often at a higher-level abstraction. Figure 2.3 shows how these mechanisms affect the core clock signal and supply voltage over time. Each mechanism is enabled at $t = 0$ and results in a power decrease.

The most important of these from the perspective of operating-system-directed power management are DVFS and clock/power gating, discussed in the following sections.

2.3.1 DVFS

Dynamic voltage and frequency scaling (DVFS), or *P-states* in ACPI [HIM+09] terminology, is a mechanism that allows changing the frequency of a processor core at run time. The supply voltage can then be set to the minimum required for correct operation at the chosen frequency. Per Equation 2.2, this voltage is a non-linear function of several parameters, however it is often approximated as a simple linear function of frequency [KAB+03],

$$V \approx \beta_1 f + \beta_2 \,,$$

for some constants β_1, β_2. Thus, using Equation 2.1, dynamic power can be approximated as

$$P_{\text{dynamic}} \propto fV^2 \propto f^3 \,. \tag{2.3}$$

Performance of a workload is (approximately) linear in frequency, so

$$t \propto \frac{1}{f} \tag{2.4}$$

for execution time t, and thus dynamic energy consumption can be expressed with the first-order approximation

$$E_{\text{dynamic}} = P_{\text{dynamic}} t \approx f^2 \,,$$

showing that DVFS can be highly effective in reducing *dynamic* energy consumption. Here we have assumed that when the workload completes, no more energy is consumed. While it might seem obvious that this would be the case, it is in general *not* true: even moderately complex systems have some cost to keeping them "online", and this must be factored into the calculation. We discuss the solution to this point later in the section.

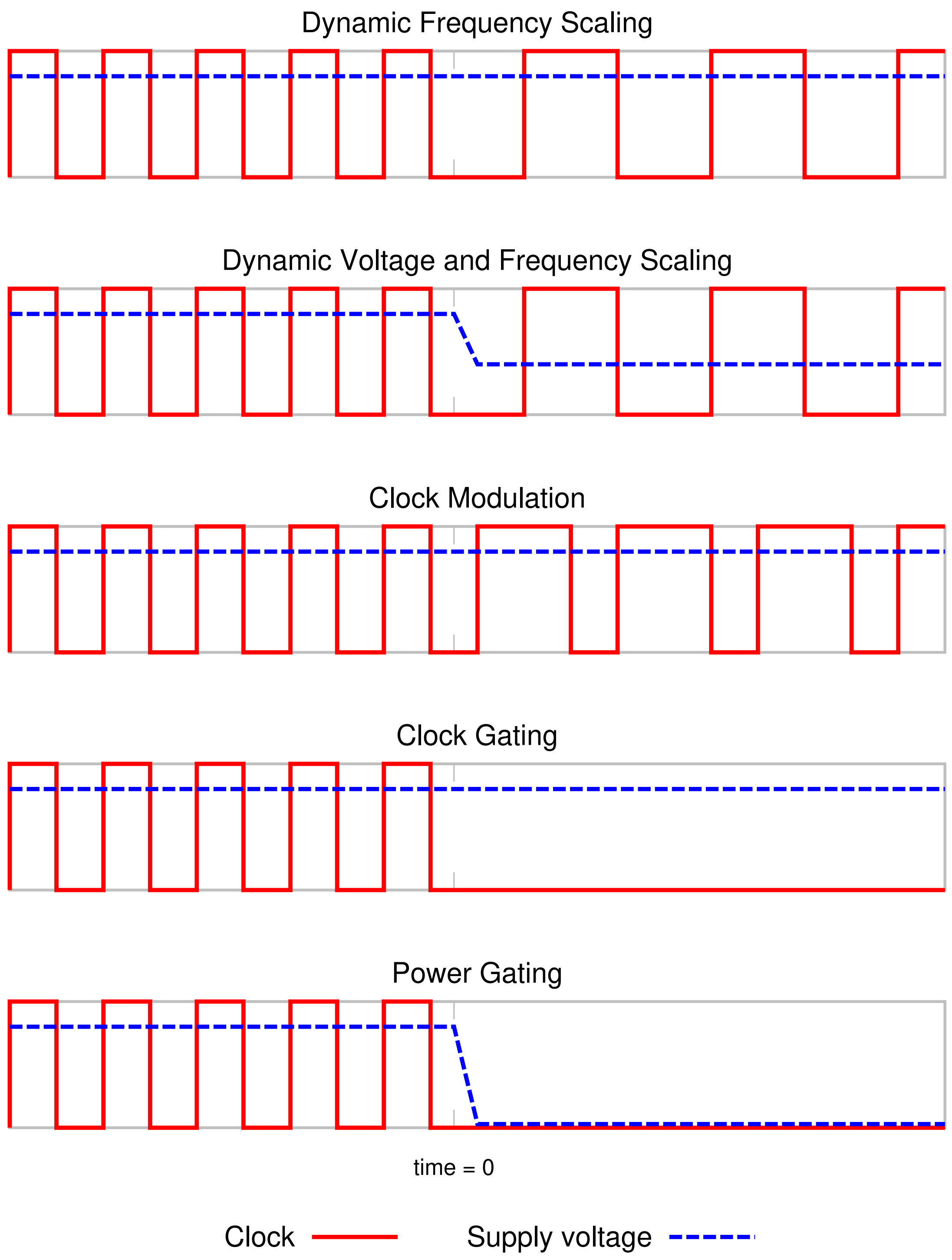


Figure 2.3: Power control mechanisms.

While DVFS clearly reduces dynamic energy consumption, this does not necessarily reduce *total* energy, due to the contribution of static power: the longer a workload takes to run, the more static energy accumulates. To illustrate this point, let us make the simplifying assumption that P_{static} is independent of supply voltage. Although not strictly true in general, this assumption is nevertheless ubiquitous in the literature. It then follows from Equation 2.4 that static energy can be expressed as

$$E_{\text{static}} = P_{\text{static}} t \propto \frac{1}{f} \, .$$

So while dynamic energy increases with frequency, static energy decreases and the energy-optimal frequency therefore depends on the relative value of the static and dynamic components. When static dominates, it is optimal to run the workload as quickly as possible (maximum frequency) to reduce the accumulation of static power, an approach known as *race to halt*. When dynamic power dominates, minimum frequency is optimal. In the general case, the optimal frequency is somewhere between these two extremes. Figure 2.4 shows this graphically.

With DVFS, both frequency *and* voltage must be scaled to see any benefit in energy consumption. If V is a constant, then from Equation 2.3 we get

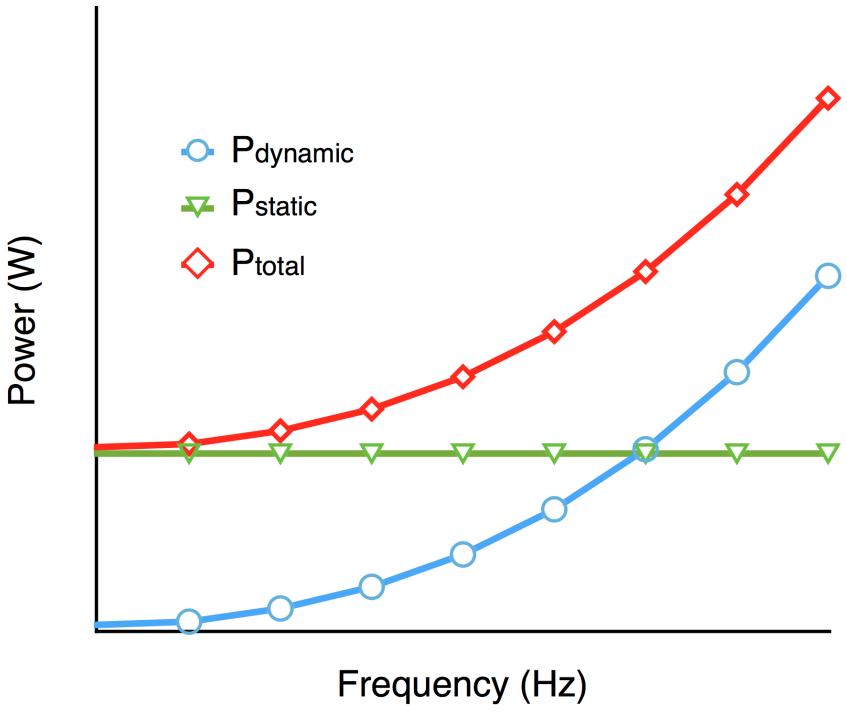
$$P_{\text{dynamic}} \propto fV^2 = kf$$

for some constant k, so

$$E_{\text{dynamic}} = P_{\text{dynamic}} t = \frac{P_{\text{dynamic}}}{f} = k \, .$$

That is, dynamic energy is constant when scaling f without also scaling V. By fixing V, reducing frequency merely increases run-time and accumulation of static energy, with no change to dynamic energy. Fixing f and scaling V is equally nonsensical because there is no advantage to increasing voltage above the minimum required for each frequency. Scaling only frequency can however be useful for thermal management, since it reduces instantaneous *power*, for systems which are so constrained.

Memory In general, it is not true that performance is linear in frequency, because the speed of main memory does not necessarily scale linearly with CPU frequency. A memory-bound workload spends a large portion of its execution time waiting for memory access to complete: if memory speed is constant, increasing the CPU speed does not change the performance of this fraction, so a proportional increase in overall speed is not observed [SAMR03]. For example,



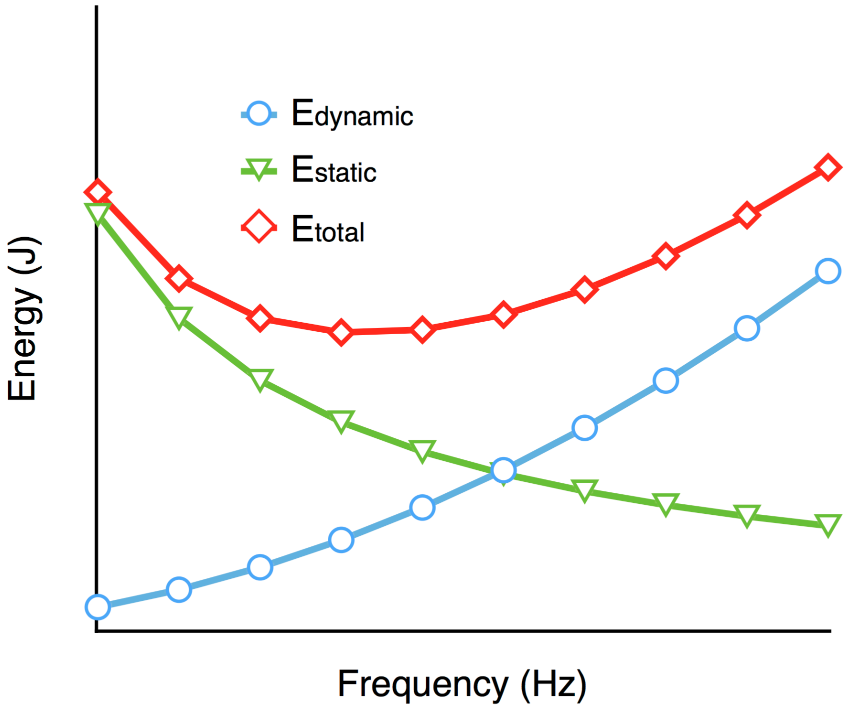


Figure 2.4: Power and energy vs. frequency under DVFS.

if a workload running at frequency f executes in time

$$T_f = T_{\text{mem}} + T_{\text{cpu}} ,$$

where T_{mem} is the time spent waiting on memory, and T_{cpu} is the instruction execution time, then doubling the frequency (with memory speed unchanged) results in an execution time of

$$T_{2f} = T_{\text{mem}} + \frac{1}{2} T_{\text{cpu}} .$$

The overall speedup is therefore

$$\frac{T_f}{T_{2f}} = \frac{T_{\text{mem}} + T_{\text{cpu}}}{T_{\text{mem}} + \frac{1}{2} T_{\text{cpu}}} = 2 - \frac{T_{\text{mem}}}{T_{\text{mem}} + \frac{1}{2} T_{\text{cpu}}} < 2 .$$

In general, performance is a complex function of CPU frequency [SvdLPH07].

Padding The preceding analysis makes the implicit assumption that when a workload completes (i.e. after time T has elapsed), the power drawn is zero. In a real system, this assumption is generally untrue; rather, the system enters a low power state with lower but non-zero power P_{idle}. To make a fair comparison between energy at different frequencies (and hence runtime), energy results must be *padded* by adding idle power to compensate for the difference in run-time:

$$E_{\text{padded}} = E + P_{\text{idle}}(T_{\text{min}} - T) ,$$

where E is the original, uncorrected energy consumption, and T_{min} is the minimum execution time across all frequencies to be compared. Padding always biases the optimal point towards lower frequencies, since higher frequencies have faster run-times and thus a larger $T - T_{\text{min}}$, so they have more padding added. The actual effect on optimal frequency depends on the relative magnitude of P_{idle}.

2.3.2 Power states

Power states (also called *sleep* states, or in ACPI, *C-states*), allow components to be disabled in exchange for reduced energy consumption. In general, multiple states exist, each with different power and latency; *deep* sleep states save significant power but take a long time to exit and enter, whereas *shallow* sleep states are fast, but save less power. Two special cases of sleep states are the *online* state (C0 in ACPI), where the component is fully powered up and available to do work but consuming maximum power, and the *offline* state, the deepest sleep state with power at the minimum level, zero in some cases.

Power states are implemented with various combinations of clock and power gating of the components subsystems, and provide different levels of data retention. In the deeper states, data loss is complete and must be saved in software (if required) before entering the sleep state. The shallower “retention” states save all data by maintaining power to the relevant circuits.

Components supporting sleep states are often arranged in a hierarchical topology of *domains*, which represent the dependencies between components. A domain can enter a particular power state only when each sub-domain has entered that state. For example, a CPU package can sleep only when all cores in the package have entered a sleep state.

2.4 Multi-core Processors

A multi-processor system is one in which multiple threads can execute truly in parallel. Such a system can achieve higher throughput than a uniprocessor clocked at the same frequency because more work is completed on each cycle. The primary unit of instruction execution is called a *core*, each of which can execute one or more threads at any time. A core consists of a register file per thread, and a set of functional, control, and storage units. Cores execute instructions independently of other cores, but share higher-level resources. A key characteristic of a multi-processor system is how the cores are interconnected and their degree of resource sharing. Typically this would be divided into a hierarchy of four levels: system, package, die, and core.

At the system level, several physical CPU packages (or *sockets*) are interconnected by a relatively slow memory bus, and share only high-level resources like the primary power supply, memory, and external I/O buses. A single socket may contains multiple dies (contiguous pieces of silicon) connected by a moderately fast memory bus, and might additionally share resources like a last-level cache (LLC). At the next level, a die can contain one or more cores linked by a very high-speed interconnect and sharing many resources, typically a L2 cache, and perhaps clock synthesisers and voltage regulators. At the lowest level, within a core, hardware threads share all resources with other threads on the same core, except their register file. This includes functional and control units, and the L1 cache.

Of particular interest to us is one point in this design space: the single-socket, single-die, single-threaded multi-core processor. This is often called a *chip-multi-processor* (since the entire device is fabricated on a single piece of silicon) or *mobile multi-core*. This design is typical in current and emerging multi-processor mobile system which integrate the processor onto an SoC (*system-on-chip*): a single-package integrated circuit with a host of peripherals and miscellaneous support modules: ROM, boot RAM, GPU, DSPs, I/O controllers, audio/video codecs, clock synthesizers, etc. The caches in such a system are typically arranged in a 2-level hierarchy, with a private L1 for each core and a shared L2 (last-level) cache.

A smartphone will typically contain several discrete integrated circuits, some of which may contain a processor. However, all but one of these are special purpose: they provide some specific functionality to the rest of the system, such as image or signal processing, and may

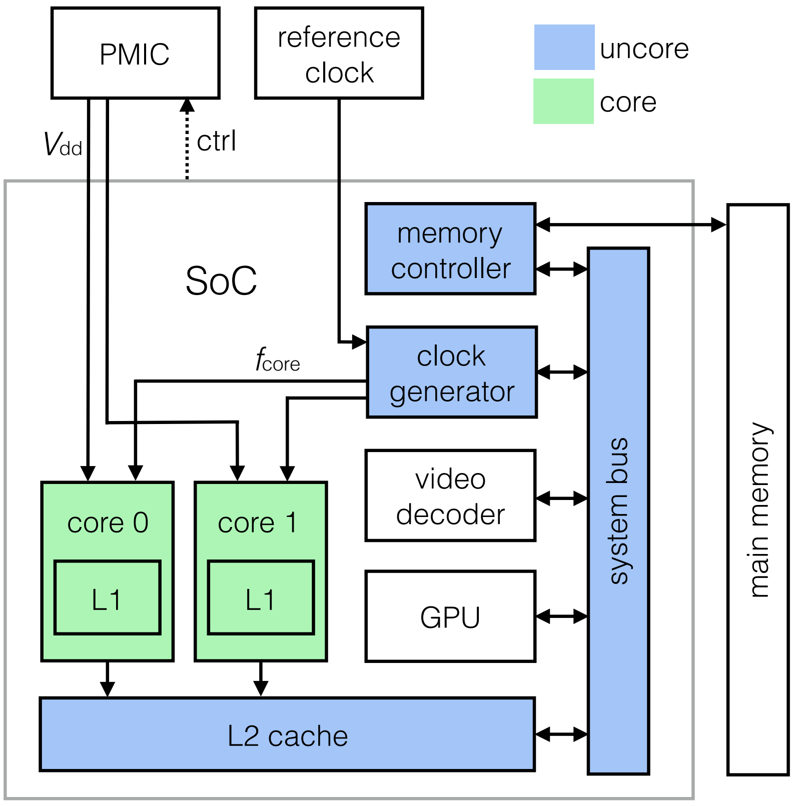


Figure 2.5: Reference mobile multi-core system schematic.

have specialized instruction sets. One processor is designated for general-purpose work: it runs a full operating system, coordinates the other processors and peripherals, and runs the user applications. For this reason it is called the *applications processor*, or AP. For the rest of this work, when we refer to a *multi-core* processor, we specifically mean a mobile SoC applications processor.

Figure 2.5 shows a schematic for an example mobile multi-core system, including the SoC and several external components. The applications processor and its support circuitry can be divided into two categories, called the *core* and *uncore*. The *core* components consist of the circuitry private to a particular processor core, such as the functional and control units, register file, and the L1 cache. These are independent of the other cores, and thus can be powered and clocked independently. The *uncore* consists of those components shared among all cores, including the last-level cache, system bus, and memory controller. These have the property that they must be online if *any* core is online and thus can be powered off only when *all* cores are offline.

2.4.1 Voltage and clock distribution

The distribution network of power and clocks to an AP follow two main designs: either each core receives an independent supply, or all cores are tied to the same voltage and frequency. The latter is simpler, whereas the former allows per-core DVFS. An intermediate design, where cores share one of voltage or frequency, is possible but uninteresting because, as discussed above, DVFS does not save energy unless the voltage is scaled with frequency.

Power is supplied to the cores from an external component called the PMIC—the *power-management integrated circuit*—which converts battery voltage to that needed by the cores. To support DVFS, these are variable-voltage supplies, programmable by the AP, typically over an I^2C bus or similar. Clocks generally follow a multi-stage distribution scheme, with an external component supplying an accurate fixed base frequency to an on-chip clock generator module with a bank of programmable frequency dividers and PLL (phase-locked loop) multipliers for synthesising the requisite core frequencies.

2.4.2 Regulation

Each integrated circuit in a system has some pins used to receive supply power. Typically, a specific voltage is required, which can be either fixed or variable, such as with DVFS-capable devices. The conversion of system input voltage to that required by the component is called *regulation*. These voltage regulators however are not 100 % efficient—some energy is lost in the conversion process. Thus to determine the total energy consumption of a system, the regulation loss must be included.

Two types of voltage regulation are typically employed in power-constrained devices like the smartphone. The first, called the *linear* or *LDO* (low drop-out) regulator, is low cost and low noise, but suffers from poor efficiency proportional to the input/output voltage difference. A switch-mode power supply (SMPS) on the other hand, is highly efficient, typically 80–90 % in the design operating region, but is expensive and noisy.

SMPSs are usually employed for high-current supplies when good efficiency is required, except in low noise situations, such as high-frequency radios. In those cases, linear regulators are used for their superior noise rejection [Vol02]. A multi-stage regulation process may also be used, with the SMPS performing the bulk voltage conversion and a downstream LDO providing noise immunity.

Chapter 3

Related Work

In this chapter we discuss related work. Firstly in Section 3.1, we discuss the literature around energy measurement, and then in Section 3.2 we review existing works on energy management.

3.1 Energy Measurement

There are two types of papers in the literature focused on energy consumption *per se*: direct measurement and analysis of the energy consumption of specific devices, and the design and implementation of systems that provide online power estimation by measurement, modeling, and prediction.

3.1.1 Direct measurement

In this section we will summarise the related work on the direct measurement of energy consumption. First we cover whole-system analyses of mobile computers, then we cover desktop, server, and laptop-class systems. Finally, we look at works that concentrate on specific subsystems which make up a smaller part of some larger system.

Mobile

Sagahyroon [Sag06] performs energy measurements of the iPAQ pocket PC (a precursor to the smartphone but without telephony capabilities). They instrument the whole system at the battery terminals, measuring the power draw for workloads including CPU-intensive, MP3 playback, graphical, and file access. The maximum power observed was 550 mW, which is much lower than in modern devices. The CPU maximum frequency is 500 MHz. They also perform battery discharge experiments by measuring battery voltage, which correlates with charge level. Finally, the author performs some "real-world" experiments involving the opening and closing of word documents. This use-case however is unlike how modern smartphones are used.

In 2010, with my advisor, we published our paper [CH10] performing an energy measurement and analysis of the Openmoko smartphone, as described in Chapter 4. We include this reference here for context, as the paper pre-dates many of the other works summarised in this section.

Perrucci et al. [PFW11] perform a series of experiments on the Nokia N95 phone using the "Nokia Energy Profiler" which appears to provide access to on-board energy measurement functionality. The paper refers to previous work that compares the profiler precision with an external measurement apparatus, showing no significant difference. In these experiments, specific components are exercised with the unrelated components powered down. They focus specifically on wireless interfaces: Bluetooth, WiFi in both infrastructure and ad-hoc modes, and 2G and 3G cellular radios. The various modes of these radios are analysed, including idle, send, receive, and connect, with higher-level uses cases such as SMS messaging, and voice and video calling. The joules per bit metric is used to evaluate the efficiency of each technology at varying bitrates. They also measure the effect of varying the display repaint rate on screen power, but it is unclear whether this affects the display's actual refresh rate, or whether this simply shows the changing cost of software rendering. Finally, Perrucci performs a series of combination measurements with the various radios, showing that their power contributions are additive except for the Bluetooth + WiFi case, since the device features a combined Bluetooth/WiFi chip. Thus the static energy consumption is amortized when using the two radios simultaneously. Across all benchmarks, downloading a file with HSDPA is the most energy-intensive operation.

Li et al. [LZCF14] perform whole-system measurements of three Android smartphones under a number of macro workloads. While no specific conclusions are drawn, a large body of data are presented. As expected, the wireless radios consume a significant amount of power, and the multi-threaded CPU benchmarks show that CPU power can also be significant under certain workloads.

Bai et al. [BMH$^+$13] perform a similar analysis, while also presenting plots of power over time which reveal transitions between various operating states. From these data they show that the power drawn by wireless radios tend to have high variance over short time scales. Similarly, our data show high cellular radio power variance over all time scales. They also show the cost of entering and exiting flight mode, and the suspend mode for 2G/3G modems.

LiKamWa et al. [LWC$^+$14][1] perform energy and heat measurements on the Google Glass, an AR (augmented reality) device with an optical head-mounted display (OHMD). It features the same sort of peripherals as a smartphone, but with different characteristics and use-cases. The heat analysis described in the paper shows that thermal limits may stem from safety con-

[1]Aaron Carroll was co-author of this paper while a visitor at Rice University.

cerns, in addition to power supply and head dissipation restrictions, and that this may place an even smaller power constraint on the device. Further, the paper shows that screen power is important in OHMDs, despite the small size, due to its always-on nature.

Desktop/server/laptop

Flinn and Satyanarayanan [FS99a, FS99b] measure the energy consumption of two laptops: the IBM ThinkPad 560X and the IBM ThinkPad 701C. They use a digital multimeter connected to the machines' power supplies to sample whole-system power. Using benchmarks that exercise specific components, they measured the power draw of the display, network card, and disk, plus the remainder of the system. On the 560X machine, they found that display, particularly at high brightness, consumes the majority of the energy: 35–55 % depending on brightness. However, on the 701C, the brightness of the display surprisingly does not significantly affect power, with only a 20 % increase between the dim and bright settings. It is unclear why this is the case, since both devices feature backlit LCD displays. Further, the authors claim that component power is super-linear when run simultaneously; that is, the power while running two components at the same time is greater than the sum of the power of the components when run individually. Although they do not offer an explanation of this, one possibility is that it is due to power supply non-linearities. When multiple components are powered up, higher current flows, and this may push the power supply into a less efficient region of operation. It could also simply be due to copper loss in the power distribution network, which is the quadratic $P = I^2R$ power loss when a conductor of resistance R ohms carries a current I amps. In any case, this is not true in general, because components often share common circuitry, the energy consumption of which is amortized when running components simultaneously. Uncore is an example of this in CPU power.

Mahesri and Vardhan [MV04] perform power measurements on a laptop, including the CPU, optical drive, hard disk, display, GPU, RAM, wireless radio, speaker, fan, USB, and the whole system. Some of these measurements are direct and some subtractive. However, they were not able to account all power consumption of the platform to a corresponding component. Overall, the main energy consumers are the CPU and display. In particular, they show that reduction of the backlight brightness is one of the most important mechanisms for power saving. Power is measured with a current probe: a device which couples magnetically to a current-carrying wire, but requires a full loop around the conductor. This does not work for PCB slot-connected components, so the authors use connector shims that split out power rails for easy connection to a voltage probe. CPU power is determined by a simple model, $P \propto fV^2$, which ignores static power and the fact that different workloads may have different power consumption at the same frequency. Similar to our results, the paper shows that a white screen is more efficient than a black screen for LCD-type displays. They also measure the effect of varying the colour bit-depth, but show no power variation in doing so.

Hamady et al. [HCK11] measure the power of an Intel-based laptop platform, including the contribution of CPU, storage, RAM, WiFi, and display, using various micro-benchmarks and real-world workloads. They show total energy results as an increase above idle, with a maximum of 140 %, showing this platform has a high static-to-dynamic power ratio. Approximately half of the platform's energy consumption is unaccounted for, as shown in the "rest of platform" numbers. Like our results, Hamady shows that storage and RAM contribute little to overall energy consumption. They also highlight the importance of functional and energy inter-dependence, that is, components that require other components for support. For example, the power of the network card is tied to the CPU power, since the processor is required to process incoming packets.

Ferreira et al. [FHM$^+$13] describe "SEFLab," a system where power measurement is provided as a service. They are able to measure CPU, RAM, the system fans, hard disk, and the motherboard. A server coordinates the execution of the submitted workloads with the collection of power measurements from a connected data acquisition system (DAQ). Like other measurements on server/desktop systems and workloads, they show that idle power is a significant contributor, even for energy-intensive benchmarks. They also claim that the (spinning platter) hard disk is entirely static power. An explanation of this is that keeping the platters spinning, which is required whether or not an access is in flight, is the dominating energy consumer. Further, Ferreira shows that RAM power is also largely static. This mirrors similar studies on desktop-class systems, but differs from our results on mobile platforms which show that RAM power is primarily dynamic. The paper focuses on the energy consumption of different browsers loading different web pages. From benchmarks of these, the authors conclude that both the browser implementation and website content has significant impact on energy consumption, and specifically that browser choice can impact dynamic energy by up to 70 %.

Cui et al. [CZBC11] present a methodology for performing energy measurements on ATX systems—a standard that specifies physical form factor and power supply voltages and connectors for desktop computers. In the work, components are broken down into three classes based on how they are supplied with power: 1) a cable directly from the power supply unit, 2) a slot connector which connects one circuit board to another, or 3) direct connection to conductors on the motherboard (so called *pads* or *lands*). For wired devices such as disks, they simply cut the wire and insert a sense resistor. For slotted components, they use a similar approach to [FHM$^+$13], namely, they design a shim card that passes through signal traces but diverts power rails to external measurement circuitry. This approach works for the RAM, network card, and GPU. Although the CPU could in principle also be measured this way, due to extremely tight timing and noise requirements, it is more practical to instead use the dedicated ATX CPU supply wires. An interesting result presented in this paper is a comparison between sequential and strided reading of RAM. They show that the power draw in both cases is approximately the same, but the strided case takes longer due to an increase number of row-buffer misses. As a result, the total energy consumption is greater by a factor of 6.

Chen, Wang, and Shi [CWS11] present an analysis of energy consumption in two desktop PCs. As in the above studies, the CPU, memory, and disk power are measured by cutting the supply line and inserting instrumentation, in this case a digital multimeter (DMM). Whole-system power is measured with an off-the-shelf power meter. Attributing power supply lines to components is done similarly to us: by instrumenting the unknown lines and exercising different components in software. After this, idle power can be determined trivially.

50–75 % of the system's idle power is unaccounted in this work. The authors claim this power is due to the motherboard and power supply conversion loss. To quantify the latter, they measure the total system power with and without a fan connected, and compare the difference to the power measured directly at the fan. This shows a low efficiency of only 25 %. The CPU, disk, and memory power supply efficiency is also quoted (at 70–90 %, much more reasonable in our experience), though it is unclear how these numbers are determined. We suspect that the low efficiency reported for the fan is because it is variable-speed, so the measurement is including the efficiency of the speed controller as well as the voltage conversion.

Between the old and new systems, idle power is shown to decrease by 45 %, while dynamic power decreased by 35 %. In contrast, our results showed stable idle power with a significant increase in peak dynamic power (and a commensurate increase in compute power). Since performance figures are not provided, it is hard to rationalise this discrepancy. Moreover, Chen claims that cache contributes 20–35 % of the total system power, but the methodology for separating cache power from the rest of the CPU is not explained.

Subsystems

Balan et al. [BLWM14] discuss the energy behaviour of environmental sensors in smartphone platforms. They quote the power drawn for a number of sensors, and highlight the cost in the CPU processing the sensor data, in addition to the physical device. As expected, they show that higher sampling rates lead to higher energy costs, however on some devices this difference is small due to a large static power component, such as GPS. They do not however attempt to decouple the sensor power from the corresponding CPU power. Our work, which does, shows that for the most part power consumed in the sensor unit is insignificant compared with the corresponding CPU cost. Balan et al. go on to measure the costs of sampling several sensors concurrently, showing that some are additive, whereas others are not, due to common circuitry. However, this again conflates the CPU and sensor power costs. Overall they claim that "continuous mobile sensing" is not practical on current platforms, and as such, duty cycling is important in sensing applications.

König et al. [KMD13] also investigate the energy consumption of smartphone sensors with whole system power measurements. They compare the measured cost with that reported by estimation APIs provided by the platform, and show that they are reasonably accurate: 16 % or better. This is in contrast to our experience, which is that they are wildly inaccurate. This may be due to different mechanisms: the platforms we studied use simple models, whereas it is implied (but not directly stated) that their API under test uses some internal sensing. Like us, König et al. assert that the CPU is the main consumer in sensing applications, and in contrast to Balan et al., claim that continuous sensing is in fact feasible. This is at least partly due to a difference in sampling rates each party treats as "continuous," which the authors of this work do not treat explicitly.

Abreu and Villapol [AV12] measure the full-system power of two smartphones under various Bluetooth, WiFi and cellular usage scenarios. They show the surprising result that Bluetooth has the highest scanning cost. However, similarly to us, they show it has the lowest energy per bit cost, following by WiFi and then cellular.

3.1.2 Modeling and prediction

Bellosa [Bel00] looks at the correlation between micro-architectural events and power. These events can be counted automatically in the performance monitoring unit (PMU), including micro-operations retired, floating point operations, and L2 cache and memory accesses. This work established the use of the performance monitoring unit as the basis for online power estimation, a technique that is still widely used today. Power is estimated using a simple linear model where each event corresponds to a particular amount of energy consumed. The performance events are not independent (i.e. an L2 access is the result of a retired load or store, so necessarily include a retired micro-op), so the event coefficients are determined subtractively. Since machines at the time included only two physical counters but more than two possible events, the counters are time multiplexed. As an example use case, the predictions are used for power capping (via clock throttling), and they also implement a power API which allows a thread to read its own power consumption.

Isci and Martonosi [IM03] model a Pentium 4 processor by a linear combination of performance counters with coefficients calibrated offline using measurements of the CPU's 12 V input power supply. They explicitly model all the CPU's functional units (FUs), including: cache, branch predictor, trace cache, ALU, TLB, decode unit, and microcode ROM. They use 15 counters in total to model the energy consumption of 22 functional units. Each FU is modelled as a dynamic plus static component, where the dynamic component is a product of the FU's utilisation and its peak power. Peak power is determined first by an estimate based on die area of that FU, then further tuned by exercising it with specific workloads. Isci proposes program phase detection as a potential use case for this methodology. In particular, they claim that the change in FU power is a good indicator for a phase transition, since total power may not change significantly. With SPEC benchmarks, they show an average prediction error of 4 %, with a worst case of about 10 %.

Snowdon, Heiser, and others [SRH05, SvdLPH07, SPH07] published a series of papers on prediction of power and performance slowdown due to voltage and frequency scaling. Similar to earlier works, they use performance counters sampled online, plus the current CPU frequency, as inputs fed to a linear model which is calibrated offline. Earlier systems either blindly sample all events, or produce the model by manual analysis. In Snowdon's methodology, this is done automatically using best-subset linear regression which is able to determine the most relevant counters for the model. Since it is a multi-variate regression, dependencies among events are also accounted for. Further, Snowdon's formulation is predictive across frequencies. That is, given the current conditions, their model is able to predict the effect on both power and execution time if frequency were to be changed. This avoids the need for searching.

Bircher and John [BJ07] propose extending the use of CPU performance counters for power consumption estimation of a whole desktop system, including CPU, chipset, memory, PCI I/O, and disk. The methodology is based on what they call the "trickle-down effect," which says that devices are connected in a hierarchy rooted at the CPU, and the further away the device is in the hierarchy, the less its activity is correlated to events countable at the CPU. The models are non-linear and derived by hand. Memory, which is close to the CPU, is modeled by last-level cache (LLC) misses and bus transactions, which account for CPU and DMA accesses to memory, respectively. Interestingly, the authors found that features like pre-fetching, write-allocate caches, etc., which break the 1-1 relationship between LLC misses and DRAM accesses, did not significantly impact the accuracy of the model. Further away from the CPU, I/O and disk is modeled with interrupt and DMA rates. However, it is hard to see how this would apply in mobile systems, where components have high and highly-variable power consumption. For example, WiFi and cellular network interfaces can carry the same throughput with vastly different power consumption. On the desktop system in question however, Bircher shows this approach yields 9 % or better prediction accuracy for each subsystem.

In response to significant interest in modeling as a power estimation approach, McCullough et al. [MAK^{+}11] investigate the limitations of event-based linear-regression modeling techniques on modern hardware. They show that, although CPU power prediction is still in single-digit percentage accuracy, multi-core is significantly worse than single core due to contention on shared resources. For peripheral devices, accuracy is much worse, and this is primarily due to hidden states and complex transition semantics. They also highlight the high variability in power consumption for apparently identical devices. Many works assume the power model for a specific device model can be calibrated once (often with significant time, complexity, and/or specialised hardware), then run on any instance of that device. However, McCullough shows that this is not necessarily true, and that calibration for one device is not necessarily good for another. This motivates the inclusion of direct power measurement hardware in devices.

Flinn and Satyanarayanan present PowerScope [FS99b] which is a system for profiling the power consumption of applications. It consists of online power measurements at the supply, which are then correlated with samples of the program counter. A pin on the parallel port is used to synchronise the two samples. After execution, an offline analysis is run on the samples to produce a function-level power profile of the application. Flinn presents a case study into the applicability of this approach with a video playback application.

Yoon et al. [YKJ+12] describe AppScope, a whole-application power estimation methodology for Android applications which does not use performance monitor counters. The models are a linear combination of device state and utilisation, which are produced manually and calibrated offline against test workloads. For example, the CPU model uses utilisation and current core frequency to estimate CPU power. Utilization of other device types is detected though a combination of tracing (kprobes) and IPC (Binder) interposition. AppScope has a higher overhead than PMC-based systems because the counting/sampling happens in software rather than hardware. They measure a 2.1 % CPU overhead, but a power estimation error of only 5.2 %.

Pathak et al. [PHZ+11] argue that such utilisation-based models fail due to complex asynchronous and hidden states, such as tail power, or due to lack of suitable utilisation metrics (for example, the camera). Further, they argue that with PMC- or utilisation-based models it is difficult to account power to functions or even specific applications. Thus, they propose the use of syscall tracing. Devices are modeled as a finite state machine (FSM), where each state corresponds to a certain power level. Transition between states is triggered by syscalls, timeouts, and utilisation thresholds. The FSM for each device is produced by hand.

Pathak et al. [PHZ12] extend their previous work and present "Eprof" which, similar to PowerScope, is a gprof-like tool for analysing the energy consumption of an application in the source code domain. Eprof includes an instrumentation phase which inserts code at function and system-call boundaries to trace program flow at runtime. During the offline analysis phase, these events are replayed to the FSM power model and presented in profile form. Pathak shows that there is significant complexity in accounting power to a responsible agent due to the asynchronous nature of many devices and their corresponding software. An example is so-called "tail power," in which a portion of the energy consumption for an action happens after that action has appeared to complete. Concurrency also presents an accounting complexity, since the energy cost is effectively shared among several, possibly completely unrelated tasks. Pathak devises methods to improve energy accounting under these conditions. Some devices however, like RAM and display panels, do not lend themselves to a syscall-based approach, due to the high rate of access. Handling these is relegated to future work.

3.2 Energy Management

There exists a vast body of work studying mechanisms and policies for controlling power and energy consumption at the OS level. These fall broadly into two categories: reduction of total energy consumption (i.e. energy efficiency), and control of peak power (thermal management).

In this section we discuss the key works in these areas, with a particular focus on dynamic frequency and voltage scaling (DVFS) and multi-processor/multi-core systems. Thermal management is a comparatively immature topic of study, but employs similar tools and techniques. Moreover, as peak CPU power increases as a fraction of average power (the so-called *dark silicon* problem [EBA$^+$11]), thermal management will become increasingly important.

3.2.1 Energy reduction

DVFS

Weiser's seminal paper [WWDS94] kick-started the field of operating-system-directed CPU energy management. They introduced the concept of MIPJ—millions of instructions per joule—as an energy analogue to the MIPS measure of performance, and observed that MIPJ is a constant function of clock speed (when voltage is fixed), because both power and performance are linear in frequency. This is no longer true in general, due to the significance of static power, the effect of memory speed relative to CPU speed, and the prevalence of DVFS in modern machines. While DVFS was not available at the time, Weiser et al. correctly predicted that voltage scaling would be a feasible approach to achieving a super-linear reduction of energy, and performed an analysis under this assumption.

The authors take a trace and simulation approach to investigating slack management algorithms for reducing energy. Events such as entering or exiting the idle loop, performing a scheduling decision, executing a new process, etc. are recorded, and three algorithms simulated against these event logs: one with perfect future knowledge (oracle), one with small look-ahead, and one using only past history. They show a 50–70 % energy reduction is available, depending on the specific assumptions made, and showed the trade-off in selecting the interval for making new frequency decisions. They also observed that running too slow can be counter-productive, due to the additional energy consumed while "catching up."

Miyoshi et al. [MLH$^+$02] laid much of the groundwork for practical dynamic frequency scaling (DFS), particularly in their handling of static power. Most work up to that point had assumed that lower frequencies are more efficient, but this paper shows that the assumption does not hold due to the accumulation of more static power over longer run-times, and that both static and idle power are critical inputs for estimating total energy consumption.

The key insight made by the authors is that a power analysis can not simply assume power goes to zero when the workload completes: systems have non-zero idle power, and this must be accounted for to make fair comparisons. This correction is now known as *padding*. With padding, they show that with clock throttling (as opposed to scaling), higher frequencies are always more efficient. However, with DFS, the optimal frequency depends on the relative magnitude of idle and active power. A particular device can be characterised by its "power slope"—the frequency-power curve. A positive slope implies that lower frequencies are preferable, whereas higher frequencies are more efficient for a negative slope. The *critical* power slope introduced in this paper is the cross-over point between these two regimes.

Miyoshi seems to have missed the physical significance of this result: specifically, that the quantities in opposition are idle power (P_{idle}) and the static component of active power (P_{static}). Since P_{dynamic} is constant under DFS, then the optimal frequency is low if $P_{\text{static}} < P_{\text{idle}}$ and high if $P_{\text{static}} > P_{\text{idle}}$.

The critical power slope is no longer a useful analysis technique, since any modern machine has idle power lower than static power; clock gating is enough to ensure this, but the difference is typically larger due to deep power-gating. Moreover, with the addition of voltage scaling, the power-performance curve is non-linear, thus the power slope is not constant. Miyoshi correctly predicted that under DVFS, the optimal frequency could occur in the middle of the frequency range, rather than its extremes as in DFS. They also recognized that memory performance does not necessarily scale with effective frequency under clock throttling and scaling, because the relative speed of memory can increase as CPU clock speed decreases. They showed this effect to be small, so did not account for it in modeling.

Weissel and Bellosa [WB02] were the first to propose CPU performance counters as a means to estimating both performance *and* power under DFS, a technique which is today applied almost universally. They aimed to control frequency (without voltage scaling) to minimise energy consumption under the constraint of at most 10 % impact on performance, i.e. a *bounded degradation* policy.

Using the "memory accesses per cycle" and "instructions per cycle" (IPC) performance monitoring events, they perform an offline calibration using a series of varying workloads and produce a 2-dimensional lookup table mapping IPC and memory access rate to minimum required frequency. This strategy exploits the fact that some instructions can be blocked behind events which are not tied to the core clock speed, such as memory accesses, so reducing frequency also reduces power but with a lesser effect on performance, thus reducing energy. The results showed up to 22 % reduction in energy consumption for memory-bound tasks (subject to 10 % bounded degradation), and the authors predicted this could be as much as 37 % if voltage scaling was also available.

Extending their previous work on performance and power prediction of DVFS ([SvdLPH07, SPH07, SRH05], discussed above), Snowdon et al. [SLSPH09] develop "Koala," a DVFS control system in Linux. The authors add idle power and frequency switch time to their model which predicts active core power and performance at any frequency, based on a sampling of performance monitoring events at the present frequency. They devise the "generalised energy-delay policy" which can express several policies in terms of a single parameter α, minimising $\nu = P^{1-\alpha}T^{1+\alpha}$ for some $-1 < \alpha < 1$. Energy-delay product (EDP), energy-delay-squared product (ED^{2}P), minimum energy and maximum performance can all be expressed in this formulation with an appropriate selection of α. With the addition of bounded degradation, which they also implement, this covers the most common policies found in the literature. An implementation and benchmarks of Koala on several real systems shows a best case of 26 % energy saving with only 1 % performance degradation.

Le Sueur and Heiser [LSH10, LSH11] revisit the effectiveness of DVFS and idle states on newer platforms. They note that the CPU characteristics on which many assumptions about DVFS rest have changed significantly. In particular: transistor scaling changing the relative static vs. dynamic power, the relative performance of memory and CPU, improving sleep states, and the prevalence of multi-core.

The authors show that for pure computational workloads (such as SPEC CPU, which they measure), race-to-sleep is generally the best option. However, for more heterogeneous workloads (for example Apache, video playback, SPEC JBB), DVFS is still effective, particularly for slack management. The measurements are based on both Intel and ARM-based processors covering desktop- and embedded-class systems. They conclude that both DVFS and idle states are still important energy management mechanisms, and that employing the correct one for a given situation is key to maximising energy efficiency.

DVFS controllers are typically reactive, i.e. based only on past and current conditions. Isci et al. [ICM06] introduce a proactive system which sets the DVFS state based on an estimation of future behaviour using a phase analysis and prediction scheme. They use performance monitoring counters (specifically memory accesses per micro-op) to estimate and classify the memory-boundedness of a task into 1 of 6 phases. The current phase feeds a predictor, similar in design to a branch history predictor, which predicts the phase of the thread for the next interval. This prediction indexes a static lookup table which maps phase to optimal frequency.

With this scheme the authors claim an average of 7 % improvement to EDP over a reactive scheme. They observe that most workloads don't see a significant improvement with proactive control (for example because they are non-phasic, or the phases are difficult to predict), while a small number see substantial improvement.

Multi-core

Anderson and Baruah [AB04] consider the trade-off between frequency and core count in real-time systems. They observe that, because power is super-linear in frequency, reducing frequency and adding cores reduces energy consumption, and that energy can be made arbitrarily small by adding cores. However, in their real-time system model, the maximum utilisation of a core must be capped for the schedulability of the system to remain feasible, thus increasing core count reduces the maximum effective frequency. This conflicts with the goal of reducing power by adding arbitrarily-many cores, and the authors devise an algorithm for balancing these two concerns to synthesize a system with minimal energy consumption.

This paper ignores the contribution of static power, which places a (in practice, low) limit on scaling of core count, as does the hardware complexity. The analysis also applies to very specific real-time systems consisting of independent tasks (i.e. no communication) in a fixed-priority scheduling scheme.

Xu et al. [XZR+05] study the relationship between DVFS and offlining in the context of a cluster of single-core embedded systems. Rather than traditional task scheduling, they use a load-balancing front-end that makes node online and offline decisions, and dispatches jobs across nodes. The nodes then select the lowest frequency to sustain the incoming workload. Their power model and analysis is similar to our DVFS work, but applied to a different system architecture. Our model, for instance, includes an uncore component that theirs does not.

The authors show, based on an analysis of their power model, that if static power dominates then the optimal solution is to run fewer nodes at higher frequency. When dynamic power dominates, it is preferable to run more nodes at lower frequencies. This largely mirrors our results, except that for us, the relative uncore and static power is important. They also find that the addition of hysteresis to the implementation of the algorithm to be important to stability. They call this a “threshold scheme,” but this is unrelated to our concept of frequency threshold that we introduce in Chapter 5.

Their systems run at a coarse granularity since it takes 33 seconds to online a node. DVFS decisions are made once per second and driven by incoming jobs from the dispatcher. Compare this with the single-node multi-core case, where decisions are based on processes blocking and therefore have to be made at a much higher rate—in the 10’s of milliseconds range.

Li and Martinez [LM06] developed an approach to selecting the optimal number of cores and frequency specifically for parallel application, that is, a single multi-threaded application running on all cores of the system. Their goal was to minimise energy consumption given a performance target, which they do using a search methodology with online power measurements. The main contribution of this paper is techniques for efficiently exploring this 2D search space. The optimal number of cores is found by hill climbing, and a frequency prediction is made assuming linearity between frequency and performance. The operating point is then adjusted, and measurements of the actual power impact are made before iterating the algorithm.

Simulations of this approach show that the algorithm converges to near optimal in most cases, with algorithmic complexity linear or better (a precise complexity is not derived). However, it requires online power measurements.

Ghasemazar at al. [GPP10] aim to minimize total energy consumption for workloads with a fixed throughput requirement on multi-core DVFS-capable machines. They argue that an analytical solution is NP-hard, and that mathematical optimization is not robust for such problems due to workload variation. Instead they propose a closed-loop control solution using a three-tier hierarchy: a steepest-decent search for the optimal number of cores, a PI controller of frequency, and a task-to-core assignment phase.

This controller is simulated, and compared to a controller with open-loop DVFS and no core-count selection, shows a 17 % improvement in energy consumption while meeting the performance constraints. Compared to an oracle controller, it performs within 9 %.

Bircher and John [BJ08] perform a measurement and analysis of many factors that affect multi-core power management on server-class processors. Firstly they measure the energy cost and latency for transitioning between P- and C-states (frequency and sleep states, respectively). They then look at the effect of the frequency of idle cores in terms of cache probing latency from active cores. They show that more idle cores increase the cost of a cache probe, and show that there exists a "crossover" frequency of the idle cores, above which cache probe latency is no longer a dominant factor. They show the surprising result that core affinity can sometimes *decrease* performance, because the lack of thread migrations cause a decrease in the frequency of idle cores, and thus an increase in cache probe latency.

Further, they analyse the effect of workload "phases," which is affected both by changing application behaviour, and due to process scheduling. Variation in power consumption is used as a metric to classify phases and detect their length. Based on measurements, they show that the operating system is a significant determinant of phase length due to thread scheduling and power management decisions. They show that phases of higher power tend to have shorter lengths, that most phases fall int the 1–10 ms range, and that few phases last above 100 ms.

Analysing the effect of frequency and active core transitions, Bircher shows that running at a sub-optimal operating point is costlier than the overhead of increased transition frequency, which is relatively cheap on modern machines. Finally, they perform CPU and memory power measurements with the SPEC CPU benchmark and show that for highly CPU-bound workloads, core power exceeds memory by a factor of 5, but are approximately equal for memory-bound workloads.

Merkel and Bellosa [MB08a] consider the effect on performance and energy caused by concurrent memory accesses on multi-core systems. Previous work has considered co-scheduling for maximising performance, but has assumed cores run at independent frequencies. Merkel considers the trade-offs when co-scheduling in the case where cores share a single frequency plane, as is the case on many systems. Running at a lower frequency favours memory-bound tasks, but higher frequencies are better for CPU-bound tasks.

The authors find that memory, and not cache, is the constraining resource, since it is statistically unlikely that a single workload's working set fits within the cache, but two workloads do not. Thus, co-scheduled tasks tend to either not conflict at all, or conflict on main memory bandwidth. Further, they show that in selecting frequency, increasing the execution time of compute-bound workloads costs more energy than running a memory-bound task at a higher frequency than would be optimal for it in isolation. In other words, it makes sense from an energy perspective to scale down the frequency plane if only memory-bound tasks are running.

With these observations, Merkel implements a system that uses performance counters to measure memory-boundedness online, and uses this to sort the scheduling run-queues. Tasks are dispatched off opposite ends of these run-queues to achieve a balance of CPU- and memory-bound tasks to reduce contention on shared resources. The maximum frequency is selected unless only memory-bound tasks are running, in which case frequency is reduced. Benchmarks show that this scaling can achieve an energy-delay product of 0.92 compared with running at maximum frequency.

Merkel and Bellosa [MSB10] extend their previous work to use task activity vectors (discussed in the following section), which they use to represent contention at multiple levels of the memory hierarchy. They also introduce new mechanisms, such as cross-node migration as a means to increase workload diversity at each node. They perform an analysis of these techniques on both Intel and AMD server-class machines, but do not consider embedded systems.

Chen et al. [CHK06] consider scheduling of periodic real-time workloads on DVFS-capable multi-core systems from a real-time-theoretical perspective. They use a power model (which accounts for static leakage power, unlike many similar works) combined with a real-time model to minimise energy consumption over a workload's hyper-period, without missing deadlines. They show that this problem is NP-hard, and derive two approximate solutions: a 1.3-approximation (i.e. an algorithm that produces a solution within 1.3 times the optimal) that assumes negligible cost to switch operating points, and a 2-approximation that includes switching energy. The evaluation is entirely analytical (i.e. no measurements on a real system).

Gupta et al. [GBK$^+$12] focus on energy consumption of heterogeneous multi-core systems. In particular, they analyse the increasing importance of the "uncore" component—the part of a CPU shared across all cores. Thus, the uncore cost must be paid if *any* cores in the CPU are online. This means that in a heterogeneous system, running a small, energy-efficient core may not the globally optimal because the uncore power costs may be significant.

The authors model uncore power as a static contribution plus an amount proportional to the last-level cache (LLC) miss rate. They predict that uncore consumes 20–80 % of the total processor power, depending on the specific workload. This is validated against an SMP x86 test system in which some of the cores are "defeatured" to emulate a heterogeneous multi-core, though it remains unclear how this is achieved or exactly what this means at a micro-architectural level.

3.2.2 Thermal management

Isci et al. [IBC+06] consider how to use per-core DVFS to maximise total performance while bounding chip-level power. Their design is to control frequency locally (i.e. at the core level), with a layer of global coordination on top. Three policies are explored: two open-loop control systems, and one with closed-loop power and performance modeling and control. For the open-loop schemes, they consider one which attempts to keep all cores at equal speeds, and another which tries to slow down the slowest core and speed up the fastest core where possible, maximising the performance difference.

With simulation of a POWER4-like system, Isci shows that the predictive model performs within 1 % of an oracle. Moreover, they find that as cores are added, performance improves because the control becomes closer to continuous by amortizing the otherwise discrete operating points.

Rangan et al. [RWB09] present "thread motion," a system for capping the power consumption of a DVFS-capable processor using scheduling, rather than dynamically changing operating point. To motivate this unusual approach, the authors note that DVFS is typically slow to change operating point—too slow to adapt to find-grained application variation. They show this via benchmarking, and note that workloads with low IPC tend to have high variability at short timescales.

Rather than changing core frequency to meet the requirements of the task running on it, thread motion moves the task to a more suitable core. Cores are run at fixed power and performance levels, and intermediate operating points can be "emulated" by rapidly migrating tasks between these cores. This can also reduce system cost, since fewer hardware-supported frequencies are required for a similar outcome. The decision to migrate cores is triggered by one of two means: time-driven migrates at a fixed period, and miss-driven when a cache miss occurs. The thread motion design includes an off-core manager that makes the migration decisions, plus dedicated cache resources for quickly transferring register state between cores.

By simulation, Rangan shows that thread motion can improve performance by up to 20 % compared with a traditional OS-directed policy at a fixed power budget. Further, they show that a thread motion system with only two voltage/frequency planes is nearly as good as continuous, per-core DVFS.

Cochran et al. [CHCR11] present “Pack & Cap”—a system for maximising performance for multi-core HPC applications while capping peak processor power. Their systems uses a multinomial logistic regression (MLR), a mathematical classifier which maps the power cap and performance counter and temperature data to the best predicted operating point (frequency and thread-core assignment).

The paper argues that thread packing, i.e. assigning an arbitrary number of threads created by an application to the cores available, is more practical than reconfiguring the application to use fewer cores, since it is simpler, more general, and achieves similar performance and power results. Moreover, they show via benchmarking that power and energy reduction are at odds: lowering the instantaneous power cap necessarily increases the total energy consumption. This is a consequence of static power and can be considered an analogue of race-to-halt.

The classifier is trained offline with the PARSEC benchmark suite. Using a so-called L_1-regularization, the most relevant performance counter events are automatically selected in a way that reduces sensitivity of the result to any single event by artificially capping the regression coefficient of that event. The resulting model is a linear combination of events and ratios between events. Invoking the classifier online is expensive, costing 10–50 ms for a single iteration of the algorithm. Thus the algorithmic period must be restricted to seconds or longer.

Evaluating the approach, Cochrane shows that it can meet a power budget 82 % of the time. Compared with existing work, this reduces energy consumption by 52 % for the same power range. However, the analysis uses the same benchmark suite for both training and analysis, which can result in model over-fitting and artificially inflated results.

Raghavan et al. [RLC$^+$12] observe that while transistor density is increasing, voltage scaling is no longer progressing since the transistor threshold voltage has been reached. As a result, power is the limiting factor for long-term average throughput. However, workloads often demand responsiveness rather than long-term throughput. “Computational sprinting” is a technique that allows the CPU to exceeds its power budget for short bursts to achieve good responsiveness, then returning to nominal power to cool, without exceeding the instantaneous power budget. The additional short-term power is absorbed by a phase-change material which exhibits high thermal capacitance.

The evaluation system is a 16-core machine which is assumed to have peak power $16\times$ greater than the nominal TDP (thermal design power). The authors perform a thermal heat flow analysis to determine the sprint energy, peak sprint power, and cooling time, in terms of the thermal resistances and capacitances in the system. They also consider the effect on the power distribution network due to the load spike when powering-up cores. To avoid overloading the supply, core power-up is carefully staged to match the decoupling capacitance of the power network. However, this does not address the issue of increased power supply during the main phase of the sprint, which still requires above-TDP power supply.

Merkel and Bellosa [MB06] study the effect on performance of enforced periods of throttling due to power consumption above TDP. They argue that since power consumption is workload-dependent, some CPUs in a multi-processor system may require throttling while others do not, depending on how tasks are assigned to CPUs. Thermally balancing tasks across CPUs may therefore improve performance by reducing the incidence of thermal throttling.

They estimate power consumption using performance counters with a model based on [BWWK03] and note that minimising migrations is a goal due to the high cost of cross-node task migration. At the node level, their policy involves co-scheduling of "hot" and "cold" tasks, as predicted by the power model. If fewer tasks are runnable than CPUs available, hot tasks are migrated regularly to avoid throttling any one CPU. The scheduler is augmented with a processor thermal model: input power plus a thermal RC circuit representing the heat sink, which can be used to predict when a package will reach its thermal limit to trigger a migration.

The system is implemented in Linux and shows an approximately 5 % average increase in throughput for mixed workloads, and 76 % in the best case. The technique is most useful for over-provisioned systems (i.e. where the number of processors exceeds threads). However, for high utilisation systems the effectiveness depends critically on having a diverse mix of hot and cold workloads; homogeneous workloads are inherently balanced so migration offers no benefits.

[MB08b] is an extension of this work (also by Merkel and Bellosa) to the core functional-unit level. They observed by running SPEC CPU workloads on a Xeon machine that temperature varies by up to 30 degrees between functional units on a single core. This is called the "hotspot" problem. However, thermal limiting by throttling activates if *any* part of the chip exceeds the temperature limit. Thus, limiting hotspots can reduce thermal throttling and hence improve performance.

Merkel proposes the concept of the "task activity vector"—an object that represents a task's utilisation of functional units. Performance counters are again used to predict the power consumption at the functional unit level, as proposed by Isci and Martonosi [IM03]. The scheduler run-queues are sorted based on these activity vectors to avoid the formation of hotspots. The sort function is a (higher-dimensional) vector angle, and the next task selected is the one that maximises the vector angle from the current task. They also use a more sophisticated thermal model that represents the physical geometry of the chip: each functional unit is modeled as a thermal capacitor linked to its neighbours by a thermal resistor. The paper also considers the effects of multi-threaded (SMT) processors, and task balancing across multi-core systems to ensure a mix of diverse tasks is available to each core.

With a Linux implementation, the authors measure the system's performance under SPEC. Since it is hard to directly measure the temperature of functional units, they use the (unfortunately named) HotSpot simulator with a model based on the floorplan of a Pentium 4. They are able to achieve a 7–14 % increase in throughput compared to the baseline system, which includes the performance loss due to the run-queue sorting. Finally, they argue that this technique can also increase the lifetime and reliability of a processor, since thermal wear is superlinear in temperature.

Powell et al. [PGV04] make a similar contribution. They observe that SMT exacerbates the hotspot problem since its goal is to increase core utilisation. They also observe that the thermal resistance *between* functional units is much higher than that between the die and the heat sink; that is, heat tends to flow from the hotspots out of the chip, rather than laterally across the chip. Thus, cooling is more efficient if the whole die is allowed to cool at once. To exploit this fact, the authors propose a policy of maximally heating multiple functional units in a core by intelligent SMT scheduling, then cooling the entire core by migrating off all threads. For an over-utilised system (i.e. when there are more software threads than available hardware threads), heat and run increases duty cycle of throttling because the "off" periods are more efficient at cooling, so the overall throughput is increased. Measurements of this approach show a 9 % improvement in throughput compared with a base system, and 6 % higher than if DVFS was used to limit power.

3.3 Other topics

Understanding and optimizing energy consumption is a vast topic which intersects many other subjects within operating systems and computer systems more generally. A full exploration is out of scope for this work, but this section serves to highlight some of these topics and their key works.

Several studies look at how power and energy can be incorporated as a first-class resource in OS design, similar to CPU and memory in contemporary systems. These include the works of Zeng et al., specifically "ECOSystem" and "Currentcy" [ZELV02, ZELV03], and the "Resource Containers" work by Banga et al. [BDM99]. Stoess et al. [SLB07] explore the interaction of virtualization and energy management.

Operating systems design for emerging low-power and heterogeneous architectures has recently seen some attention, such as the "Reflex" and "K2" works of Lin et al. [LWLZ12, LWZ14] and the "Popcorn" architecture by Barbalace et al. [BSA+15]. On the other end of the scale, energy management for data-center-class machines has seen renewed interest due to the explosion in popularity of cloud computing, such as [CAT+01] and [MGW09]. Barroso and

Hölzle's article [BH07] makes the argument that server systems need to be energy efficient at a range of utilisation levels, and name this the "energy-proportionality" problem. Specifically, systems should be designed such that the energy consumption is proportional to the utilisation. This is in contrast to today's server systems, which tend to be under-utilised yet dominated by static power.

A number of works have explored the role of applications in overall system power management. Flinn and Satyanarayanan [FS99a] introduce the idea of trading application "fidelity" for reduced energy consumption. Pathak et al. [PJHM12] look at the downsides of exposing energy decisions to applications via Android's wake-lock API, which introduces a new class of programming error, the "energy bug," which impacts battery life by preventing system sleep.

3.4 Summary

Studies of computer energy consumption, as a topic in itself, has seen small but sustained interest in the academic community in the past two decades. The majority of these works make whole-system measurements by instrumenting the power supply (battery or mains). This approach is limited in specificity, that is, the degree and accuracy to which energy can be attributed to individual sub-components of the system. With whole-system measurements, component power can be estimated by exercising the component and subtracting the measured power from a base-line measurement. However, this approach is fundamentally limited in that it cannot disentangle the power of inter-dependent components.

In desktop, server, and laptop-class systems, a number of works have performed fine-grained power measurements. Power distribution for some components of such systems (e.g. CPU and disk drives) is by dedicated wiring, which makes instrumentation reasonably simple. For components which connect by circuit board slots (e.g. RAM and GPU), a common approach is to use a shim card that sits between the component and motherboard and diverts the power rails for instrumentation. These measurements, while interesting, can not easily be extrapolated to mobile systems, because the systems have different components, form factors, and use cases.

In the mobile space, a number of works have studied whole-system energy consumption. However, what is lacking is a detailed measurement and analysis study at the component level.

Approaches based on modeling fill this gap to some extent. Models for individual component can be calibrated by carefully exercising that component while taking full-system power measurements. These can then be combined for fine-grained energy estimation under arbitrary workloads. This approach is limited by the accuracy of the models, which varies by component and methodology. Single-core CPU models based on performance monitoring counters, for example, can predict power to within a few percent. However, accurate multi-core models are still an open problem.

In general, current approaches struggle to capture functional dependencies among components. The increasing asynchronous nature of component interactions adds further complication, particularly in mobile systems which are rich in such components (such as WiFi, Bluetooth and cellular radios). In response, recent works have moved away from low-level event monitoring toward higher-level, framework- and application-focused models. Moreover, some components, such as OLED displays, remain out of reach for accurate and low-overhead modeling.

Modeling does have some advantages over instrumentation: once the model is built and calibrated, no physical modifications or external hardware are required, so it is easy to deploy. However, for energy consumption analyses, measurement remains the only way to profile power across a wide array of components, with high accuracy and low overhead.

Energy *management*, i.e. the optimization of energy consumption, has seen significant academic attention. DVFS in particular has, and continues to be, the topic of many papers, both theoretical and practical, and in many areas of research, including OS, real-time, distributed systems, and mobile systems. However, DVFS on multi-processor systems has seen surprisingly little attention. This is at least partly due to the elusiveness of accurate modeling of core interactions through shared resources, particularly cache and main memory.

This makes predictive online-model-based approaches (e.g. Koala [SLSPH09]) largely infeasible at present for multi-core systems. Reactive approaches are therefore the most common way to utilise DVFS on multi-processing systems. Idle energy (i.e. slack) management is conducive to a reactive approach, since processor and core utilisation are good parameters to estimate slack, and they are simple to measure online. Existing works, however, are limited in general applicability to mobile multi-cores, for example requiring online power measurement [LM06], or targeting different system architectures [XZR$^+$05]. We attempt to fill this gap.

Thermal management has seen comparatively little study, particularly in mobile, because until recently the ability for devices to supply and dissipate power generally met or exceeded their requirement. This is no longer the case, thus controlling power at relatively short timescales is now an active area of research. Many of these works exploit under-utilised resources to spread heat generation spatially or temporally. Our observation that average utilisation is decreasing implies such techniques will become more relevant in the future.

Chapter 4

Profiling Smartphone Energy Consumption

This section describes work published in a pair of peer-reviewed papers co-authored with my advisor. The first [CH10], published at the 2010 USENIX Annual Technical Conference (ATC), covers the basic methodology and includes the measurements and analysis of the Freerunner device, and the validation on the G1 and N1 devices. The second paper [CH13b] was published at the 2013 Asia-Pacific Workshop on Systems (APSys) and includes our updated methodology for instrumenting commercial off-the-shelf devices, and the measurement and analysis of the Galaxy S III device.

4.1 Introduction

Mobile devices derive the energy required for their operation from batteries. In the case of many consumer-electronics devices, especially mobile phones, battery capacity is severely restricted due to constraints on size, weight, and cost of the device. This implies that energy efficiency of these devices is very important to their usability, so intelligent management of energy consumption is critical. At the same time, device functionality is increasing rapidly. Modern high-end smartphones combine the functionality of a pocket-sized communication device with PC-like capabilities, integrating such diverse functionality as voice communication, audio and video playback, web browsing, short-message and email communication, navigation, gaming, and more. The rich functionality increases the pressure on battery lifetime, and deepens the need for effective energy management.

A core requirement of effective and efficient management of energy is a good understanding of *where* and *how* the energy is used: how much of the system's energy is consumed by which parts of the system and under what circumstances. In this chapter we attempt to answer this question and thus provide a basis for understanding mobile-device energy consumption and for focusing future energy-management research.

Our approach is to profile smartphone power consumption broken down by a device's major subsystems under a range of realistic usage scenarios. We do so by taking physical power measurements at the component level (CPU, RAM, etc.) on real hardware by instrumenting the power supply to these components: Section 4.2 details how to perform such an instrumentation, and a methodology for performing energy benchmarks. In Section 4.3 we describe the two devices under examination: the Openmoko Freerunner, and the Samsung Galaxy S III. The results of benchmarking on these devices is presented in Section 4.4. In Section 4.5 we perform a validation of these data by comparing against two additional smartphones at a less-detailed level: the HTC Dream and Google Nexus One. Along with the Freerunner and Galaxy S III, these four devices represent approximately 5 years of mobile phone technology, 2008–2013. Finally, we analyse the results and trends revealed by the measurements in Section 4.6.

4.2 Methodology

4.2.1 Instrumentation

To calculate the power consumed by a component, two quantities must be known: the current flowing (I), and the source voltage (V_{dd}). From these, the power consumption at that instant can then be calculated by

$$P = IV_{\mathrm{dd}}\,.$$

Source voltage measurements can be taken by attaching a voltmeter in parallel with the component's power rail and electrical ground. To determine current, we measure it indirectly using a *current-sense* or *shunt* resistor: a small-value (typically in the milli-ohm range) resistor which is placed in series with the power rail to be measured. If a voltage drop of V_{sense} is measured across a sense resistor of value R_{sense}, then the current flowing through the resistor and hence into the component can be determined by Ohm's law:

$$I = \frac{V_{\mathrm{sense}}}{R_{\mathrm{sense}}}\,.$$

The use of a sense resistor does decrease the voltage supplied to the component by V_{sense} volts, but by using a low-value sense resistor, the perturbation is kept small enough to be within the tolerance of the component. The trade-off is that a high-sensitivity voltmeter is required to measure the small V_{sense}.

To measure the source and sense voltages, we use a National Instruments PCI-6229 data-acquisition system (DAQ). The key characteristics of this hardware are summarised in Table 4.1. The sense resistors are connected to the DAQ with twisted-pair wiring to reduce common-mode interference. The sense-resistor voltage drops are then sampled differentially

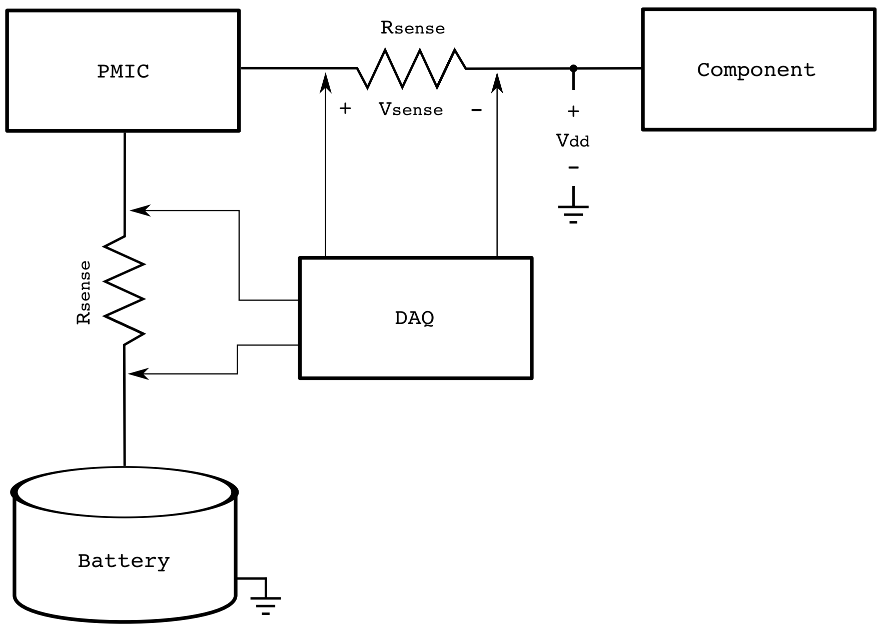


Figure 4.1: Power instrumentation logical schematic. The sense resistors (R_{sense}) are added to the circuit to support power measurements. The PMIC (power management integrated circuit) converts battery voltage to the specific voltage required by each component.

at the ± 0.2 V input range. We used the same physical connections to measure supply voltages, but instead measure the voltage relative to ground from the component side of the resistors, in the ± 5 V range. We apply the same methodology to determine the power supplied by the battery to the PMIC—the power management integrated circuit. The PMIC contains a bank of regulators which convert the battery supply voltage to the voltage required by each component. Instrumenting the battery input yields total energy consumption of the device. Figure 4.1 shows the instrumentation schematic.

This methodology relies on the ability to insert a sense resistor in series with the power rail, which requires the rail to be broken at some point. We achieve this in three different ways. First, we can make use of circuit support in the form of sense-resistor placeholders. These are PCB footprints factory-populated with $0\,\Omega$ resistors, which we simply replace with sense resistors. This is the simplest approach, but generally is not possible on commercial off-the-shelf products, which don't feature sense-resistor placeholders. Second, we can exploit the design of the power supply to instrument the rail. Smartphone designs utilise switch-mode power supplies (SMPS) to achieve good energy efficiency in converting the battery voltage to the various required supply voltages. Inherent to an SMPS is an output inductor, which is typically large and discrete (i.e. not integrated with other packages). This inductor provides a convenient point at which the circuit can be opened and a series current-sense resistor inserted for power measurement. This is shown schematically in Figure 4.2. The inductor is desoldered

Characteristic	Value
Max. sample rate	250 kS/s
Input ranges	± 0.2 V, ± 1 V, ± 5 V and ± 10 V
Resolution	16 b
Accuracy	112 μV @ ± 0.2 V range
	1.62 mV @ ± 5 V range
Sensitivity	5.2 μV @ ± 0.2 V range
	48.8 μV @ ± 5 V range
Input impedance	10 GΩ

Table 4.1: National Instruments PCI-6229 DAQ specifications [Nat16].

on one side and lifted from the board, and the resistor soldered between the DC side of the inductor and the original pad in a "tipi" configuration, shown in Figure 4.3. Finally, we can make use of "choke" components: circuit elements which suppresses high-frequency noise. On power supply lines, chokes are usually implemented as a series inductor which filter out HF signal components by radiating the energy into the environment. We can instrument such supply rails with a tipi configuration, as above. However, we found that by simply replacing the choke inductors with a sense resistors, the residual inductance was sufficient to prevent any noise-related issues, at least under our laboratory conditions.

On the Freerunner, we power the device through a bench power supply connected to the battery terminals. On the other devices this was not possible due to the battery detection feature, which forcibly shuts down the device if the battery is not detected or the charge level becomes too low. For those devices, we use the original battery which we charge as necessary. In either case, we measure total system power at the battery terminals by inserting a sense resistor between the positive terminal and the power supply/battery.

Identifying supplies We approach the problem of mapping components to their power rails (and hence sense resistors) in several ways. The simplest approach is to trace the rails by inspection of the circuit schematic. However, since this documentation is generally not available for off-the-shelf devices, it must be done indirectly. The PMIC device driver often exposes some details, such as the name, voltage, and on/off status of certain power rails, which can be correlated with measurements of the supply voltages. We found however that the most useful strategy is to exercise components in software, and correlate that with physical measurements of the supplies. This can be done in several ways. One is to observe the change in current of the power supplies when exercising a specific component. For example, by changing the display from minimum to maximum brightness, we expect only the supply responsible for powering the backlight to change appreciably. A second method is to enable and disable devices, and correlate this with power supplies turning on and off. For instance, when entering airplane mode, we expect to see the voltage of supplies for components with radios, such as WiFi and

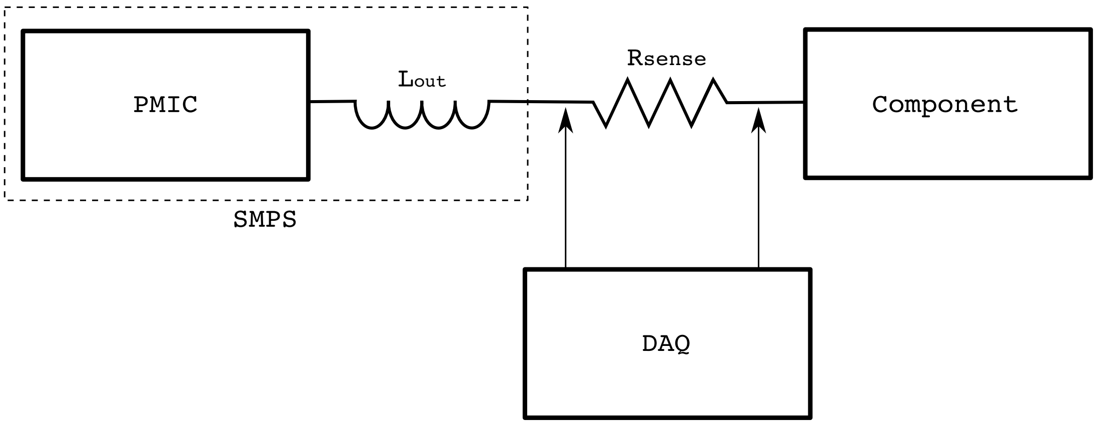


Figure 4.2: SMPS (switch-mode power supply) instrumentation schematic. The PMIC circuitry plus the external output inductor (L_{out}) form the SMPS. This inductor is used as a point to insert the series sense resistor R_{sense}.

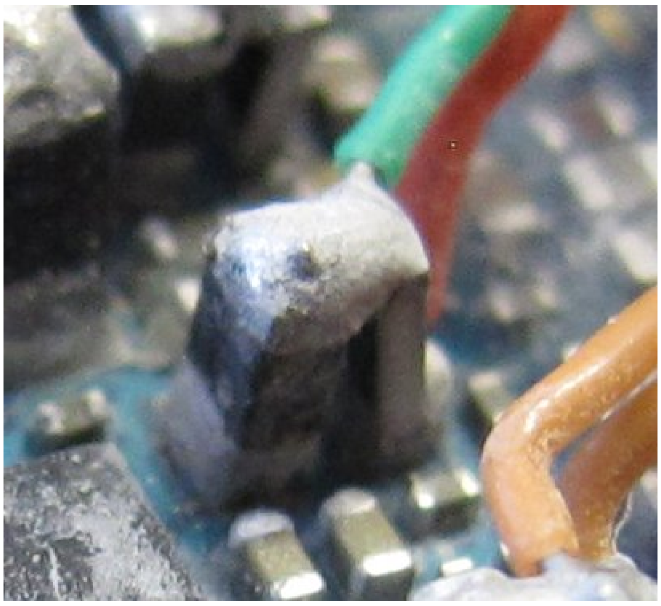


Figure 4.3: SMPS instrumentation photograph of one of the power supplies. The SMPS inductor is on the left, partially lifted off the board and soldered to a sense resistor on the right. Connected to the resistor is a twisted-pair wire leading to the data acquisition system.

Bluetooth, to change to 0 V. Finally, for DVFS-capable devices such as the CPU, we expect supply voltage (as well as current) to change as device utilisation changes.

Software The DAQ is connected to a host PC on which we run power-data collection software: a C program which interfaces with the National Instruments DAQmxBase 3.3 library to collect raw samples, aggregate them, and write the result to a file for post-processing. Each logged sample consists of a timestamp and the power for each power supply, calculated from at least 1000 consecutive raw samples each of source and sense voltage for that supply. A complete snapshot of system power is output every 200–400 ms, depending on the number of measured supplies. The host machine also coordinates the benchmarks on the Freerunner platform via a serial connection. This involves executing the benchmark, synchronising the power measurement and benchmark software, and collecting device data. On the Galaxy S3, there is no serial port available, so we perform these tasks manually.

Regulator efficiency In addition to the power supplied directly to each component, a certain amount of additional energy is lost in converting (regulating) the battery supply voltage to the levels required by the components. This energy is lost in the PMIC (and related circuitry), and thus in our methodology would not be measured as energy consumed by the component, since our instrumentation is inserted *after* the PMIC. However, since this loss is inherent to powering a component, we have decided to account it as energy consumed by the component. Thus, for all our reported results, for devices which are powered by a regulator, we have scaled the raw power measurements to include the regulation loss.

Since the actual regulator efficiency varies from supply to supply and is a function of current, for simplicity we use a fixed scaling factor of 1.2, corresponding to a conversion efficiency of 83 %. We determined this number by a combination of experimentation on the platforms, and from the public data sheet of a typical PMIC (the Philips/NXP PCF50606 [Phi04], similar to the Freerunner PMIC) which quotes efficiencies in the range of 75–85 %, depending on current.

4.2.2 Benchmarks

We run two types of benchmarks: micro-benchmarks designed to independently characterise components of the system, particularly their peak and idle power consumption, and a series of macro-benchmarks based on real usage scenarios. For low-interactivity applications, such as music playback, we simply launch them from the command line. For interactive applications driven by user input, such as web browsing, we take a trace-based approach.

Input tracing and replay Many smartphone applications are highly interactive, driven by touchscreen input from a user. Benchmarking such applications presents a repeatability challenge: we would like to run the benchmark scenarios many times to maximise confidence in the results, but repeating the same sequence of inputs by a human user is both difficult and boring. Thus, we automate this process with a two-stage solution. In the first stage, *tracing*, we execute the workload manually (i.e. with human-generated input) while simultaneously recording the sequence of inputs performed. In the second stage, the live benchmarking situation, we *replay* the same sequence of input events by generating them in software, exactly as they occurred during tracing.

During tracing, we use the benchmark application normally while running another program in the background that records touchscreen touches and physical button presses.[1] For each input event we record a time-stamp, the ID of the device providing the input, and for touchscreen events, the screen coordinates of the touch. These are written to a trace file on disk. The Linux kernel makes this information available by reading from the `/dev/input/event*` device files. To replay a trace file, we run another program that reads the trace file and synthesises input events by writing to the `/dev/input/event*` device files, maintaining the original intra-event timing.

This approach does bypass the hardware and interrupt paths that would usually be followed for an input event. Thus, our methodology does not capture the associated power. However, the vast majority of the energy consumed in delivering an input event to an application is incurred in software [HCH$^+$14]. Our measurements include this cost.

4.3 Devices

4.3.1 Freerunner

The Openmoko Neo Freerunner (revision A6) device is a “2.5G” smartphone released in mid 2008, featuring a touchscreen display and many of the peripherals typical of modern devices. It supports GSM cellular network connectivity with GPRS (2G) data. Table 4.2 lists its key components. The software running on top is the Freerunner port of the Android 1.5 operating system [And09] using the Linux v2.6.29 kernel. The notable differences between this device and contemporary off-the-shelf smartphones are the lack of a camera and 3G/4G modem. We select this device because the design files, particularly the circuit schematics, are freely available [Ope08], and because it features placeholders for sense resistors. This greatly simplifies the instrumentation problem, as described earlier. For the few components where placeholders are not available, we take the approach of replacing choke inductors.

Measuring backlight power requires special attention because its supply voltage (10–15 V, depending on the brightness) far exceeds the maximum range supported by our DAQ. To resolve this, we pre-scale the backlight voltage with some external circuitry, shown in Fig-

[1] The source code for the record and replay tools is available at https://github.com/xaaronc/injectevents. Thanks to Nicholas FitzRoy-Dale for providing the first version of these tools.

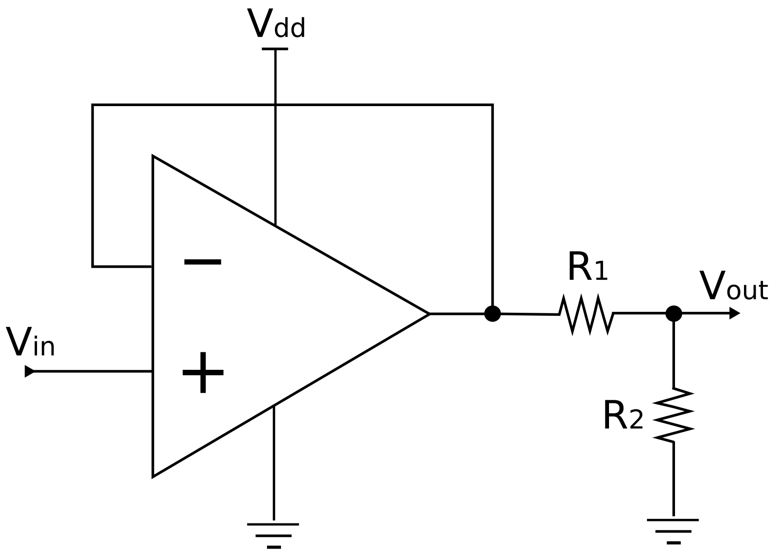


Figure 4.4: Backlight voltage pre-scaling circuit schematic. The FET opamp buffers V_{in} with high input impedance and low output impedance, and drives a voltage divider which scales the output down by a factor of 3.

ure 4.4. A FET opamp is configured as a voltage follower which has the property of high input impedance, meaning that that we draw close to zero current from V_{in}, the backlight power rail, and thus don't affect its operation. The output of the voltage follower feeds a voltage divider network, which scales the voltage down to the $\pm 5\,\text{V}$ range. R_1, R_2 are chosen to achieve a scaling factor of approximately 3, which is constant due to the high input impedance of our DAQ sampling V_{out}. In our case we use $R_1 = 3.3\,\text{k}\Omega$ and $R_2 = 6.8\,\text{k}\Omega$ for a scaling factor of 3.061. Finally we scale the voltage back up to its original value in software.

We are able to directly measure the power consumed by the following components: CPU core, RAM, cellular radio, GPS, Bluetooth, LCD panel and touchscreen, LCD backlight, WiFi, audio (codec and amplifier), internal NAND flash, and SD card. Since the graphics module has too many supply rails to measure directly, we instead use a combination of direct and subtractive measurements.

4.3.2 Galaxy S III

The second device we studied is the Samsung Galaxy S III GT-I9300 (S3) smartphone. This is a high-end mass-market device, released in mid 2012, running Android 4.0.4 on Linux kernel 3.0.15. The main hardware components are summarised in Table 4.3. The main differences with the Freerunner is the OLED display, multi-core CPU, 3G/UMTS modem supporting HSPA data rates, and a camera. Compared to current devices, this model S3 lacks a 4G modem, but is similar in other respects.

Component	Specification
SoC	Samsung S3C2442
CPU	ARM 920T @ 400 MHz
RAM	128 MiB SDRAM
Flash	256 MiB NAND
Cellular radio	TI Calypso GSM+GPRS
GPS	u-blox ANTARIS 4
GPU	Smedia Glamo 3362
LCD	Topploy 480×640
SD Card	SanDisk 2 GiB
Bluetooth	Delta DFBM-CS320
WiFi	Accton 3236AQ
Audio codec	Wolfson WM8753
Audio amplifier	National Semiconductor LM4853
Power controller	NXP PCF50633
Battery	1200 mAh, 3.7 V Li-Ion

Table 4.2: Freerunner hardware specifications.

Component	Specification
SoC	Samsung Exynos 4412
CPU	ARM Cortex-A9 quad-core @ 1.4 GHz
RAM	1 GiB LP-DDR2
3D GPU	ARM Mali-400 MP
Cell radio	Intel PMB9811X 3G UMTS/HSPA
Display	Super AMOLED, 4.8”, 720×1280
PMIC	Maxim MAX77693
Battery	2100 mAh, 3.8 V Li-Ion

Table 4.3: S3 hardware technical specifications.

	Power (mW)						
Image (brightness)	Sum	Total	P_{OLED}	P_{model}	P_{display}	Error	Error %
1 (min)	399.0	673.2	25.7	263.3	274.1	10.9	4.0
1 (max)	408.4	1141.4	322.3	729.0	733.0	4.0	0.6
2 (min)	397.3	649.8	9.3	237.6	252.6	15.0	5.9
2 (max)	402.4	819.0	115.4	404.2	416.6	12.4	3.0
3 (min)	393.4	654.3	19.2	253.2	260.9	7.6	2.9
3 (max)	406.8	1024.7	246.0	609.2	617.9	8.8	1.4
Geometric mean							2.3

Table 4.4: OLED model validation data.

On the S3, instrumentation is not so simple due to the lack of placeholders for sense resistors, and because the schematics are not available. Instead we use the SMPS output inductors as described in Section 4.2.1. By applying this methodology to the S3, we can directly measure: the CPU cores, RAM, 3D GPU and several general system-on-chip (SoC) supplies. For the SoC, we distinguish three supplies: *MIF* (memory interface), *INT* (internal), and remaining miscellany *SoC*. Measuring power supplies to the radios is less straightforward: they are very sensitive to noise in the power supply, and therefore, some of their power is supplied from linear regulators, which are typically less energy efficient but show superior noise performance [Vol02]. As these have no discrete components, we cannot insert a sense resistor. We therefore determine WiFi and cellular power by subtracting the sum of measured components from total power. We also use this approach to determine camera and audio power, which we are unable to measure at all. While this precludes measuring more than one of the WiFi, cellular, camera or audio subsystems simultaneously, we believe this does not rule out any significant benchmarking scenarios. Furthermore, we are only able to measure one of several power supplies for the OLED display. However, we find that

$$P_{\text{model}} = 223\ \text{mW} + P_{\text{OLED}} \times 1.57$$

is a fairly accurate model of the overall display power P_{display}, where P_{OLED} is the part we could measure directly. This model exhibits an average 2.3 % error (maximum 5.9 %) when validated against three images, selected for varying color content, at varying brightness levels. The data for this validation are shown in Table 4.4. Each of the images is measured at minimum and maximum display brightness. "Sum" shows the sum of measured power for all non-OLED supplies, and "Total" is the power measured at the battery. The difference between the two is thus the actual display power (P_{display}). P_{OLED} is the measured OLED power, and P_{model} is the predicted total OLED power produced by our model. "Error" shows the difference between the measured (P_{display}) and predicted (P_{model}) display power.

Similarly, we can only measure part of the power drawn by flash storage, but in this case

	Power (mW)			
Benchmark	Measured	Actual	Error	Error (%)
Read	76	230	154	67
Write	95	191	96	50
Metadata read	51	159	108	68
Metadata write	16	26	10	38

Table 4.5: Flash modeling data.

we are not able to determine an accurate model. Instead we perform a series of benchmarks which stress the storage subsystem performing one of four types of operations: data read, data write, metadata read and metadata write. For each, we determine flash power by subtracting the sum of all measured components from the total power measured at the battery. We then compare the averages of the partial measured value with the actual flash power. The results are shown in Table 4.5.

We see that the error across the benchmarks is high and variable, with a worst-case error of 68 % for the metadata read workload. We use that data point to build a pessimistic model of worst-case flash power: $P_{\text{flash}} = 3.1 \times P_{\text{measured}}$. We then use this model to predict which workloads have potentially significant flash activity. This turns out to be only the camera video recording benchmark. Across all other scenarios, the average flash power is only 6 mW. These data is shown in Section 4.4.4.

Full instrumentation of the S3 smartphone, including circuit analysis, took about 2 person-weeks of work.

4.4 Fine-grained measurements

In this section we describe the details of each benchmark and present the results on both the Freerunner and S3. Sections 4.4.1–4.4.7 deal with the baseline cases and micro-benchmarks, and in Sections 4.4.8–4.4.15 we present the macro-benchmarks. Unless otherwise noted, we use 50 % display brightness power: that is, brightness-dependant power consumption is half of its maximum value for each device. This does not necessarily correspond to a centered brightness slider, which may provide non-linear control. Furthermore, for the Freerunner we report the results with the GSM cellular module enabled and connected to the voice network, but GPRS and WiFi data connection disabled unless required for the benchmark. This represents a common scenario at the time of that device's release. However, with current network heterogeneity and the relative affordability of data access, this simplification is no longer true. Thus, for the S3, we instead perform all measurements in airplane mode (unless networking is required for the benchmark), and separately investigate the impact of network connectivity.

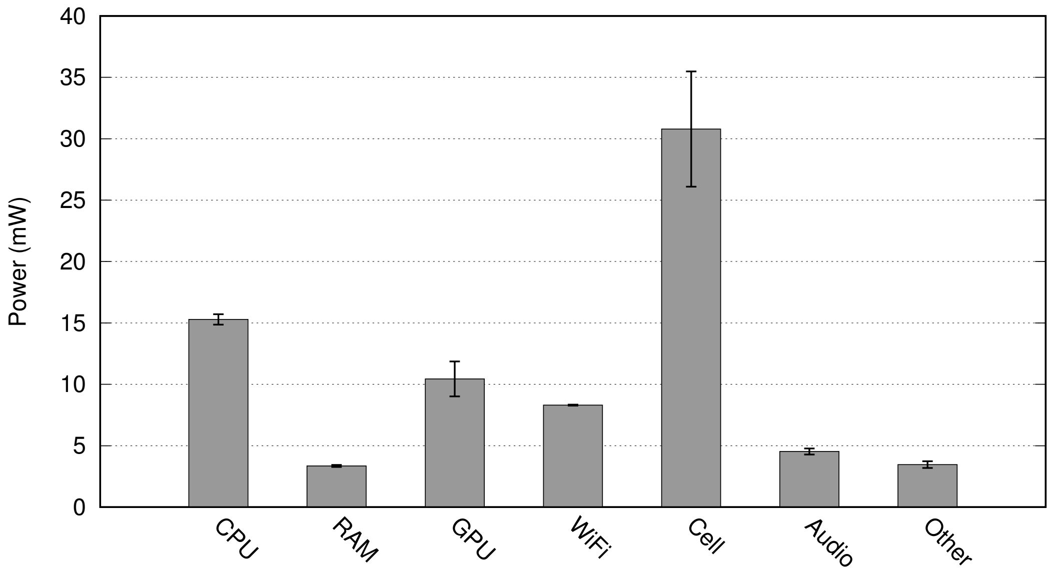


Figure 4.5: Freerunner power breakdown in the suspended state.

Each data point, unless noted otherwise, is an average taken over 3 or more iterations of the respective benchmark. The error bars on graphs show the standard deviation.

4.4.1 Suspend

A mobile phone will typically spend a large amount of time in a state where it is not actively used. This means that the application processor and most peripherals are idle, while the network processors perform a low level of activity to remain connected, in order to be able to receive calls, data, SMS messages, etc. As this state tends to dominate the time during which the phone is switched on, the power consumed in this state is critical to battery lifetime. The Android OS running on the application processor aggressively suspends to RAM during idle periods, whereby all necessary state is written to RAM and the devices are put into low-power sleep modes (where appropriate).

To quantify power use while suspended, we force the device into Android's suspended state and measure the power over at least 60 seconds. Figure 4.5 shows the power breakdown in the suspended state on the Freerunner, with cellular (no GPRS) network connection enabled and WiFi disabled. On the S3 we investigate several suspend scenarios depending on which radio technology was enabled: either none (airplane mode), 2G only, 3G only, or WiFi only. These results are shown in Figure 4.6, broken down by the three highest energy consumers: the radio module (either WiFi or cellular), RAM, and the SoC, plus the sum of all remaining components, "other". The relative standard deviation (RSD) of total power is 6.5 % or less. In all cases we ensure no data packet transfers occur either by explicitly disabling it, or by

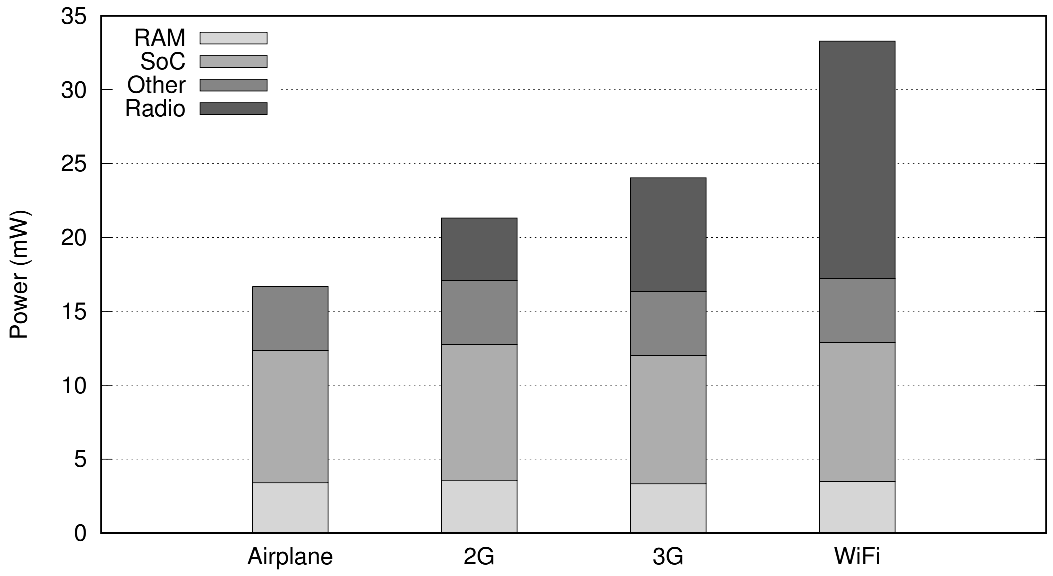


Figure 4.6: S3 power breakdown in the suspended state.

discarding iterations where a transfer occurred. This yields the power required to maintain the network connection, but excludes the cost of data access which is highly application- and user-dependent.

4.4.2 Idle

The device is in the idle state if it is fully awake (i.e. not suspended) but no applications are active, for example with the home screen open. This case constitutes the static contribution to power of an active system. Figure 4.7 shows the Freerunner in the idle state. The cellular module is enabled but with GPRS disabled. The results for the S3 are shown in Figure 4.8, in this case in airplane mode.

The results show that the display-related subsystems consume the largest proportion of power in the idle state: more than 50 % on both devices. On the Freerunner, the cellular subsystem consumes 58 mW, about 10 % of total. On the S3 however, our idle and suspend results show that the cellular consumes less than 10 mW in the worst case: only 1 % of total. While the CPU itself consumes little power on either platform, S3 shows significant consumption in the various SoC supplies (SoC, MIF and INT).

4.4.3 Display

Our two devices feature different display technologies. The Freerunner uses a transmissive liquid crystal display (LCD). Such a display does not generate light itself, but rather forms

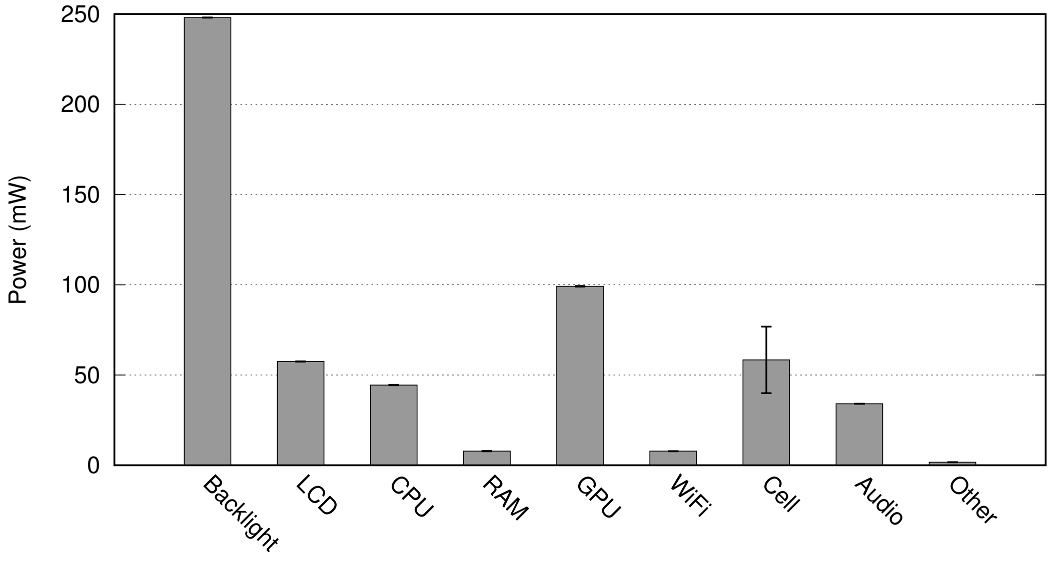


Figure 4.7: Freerunner idle power.

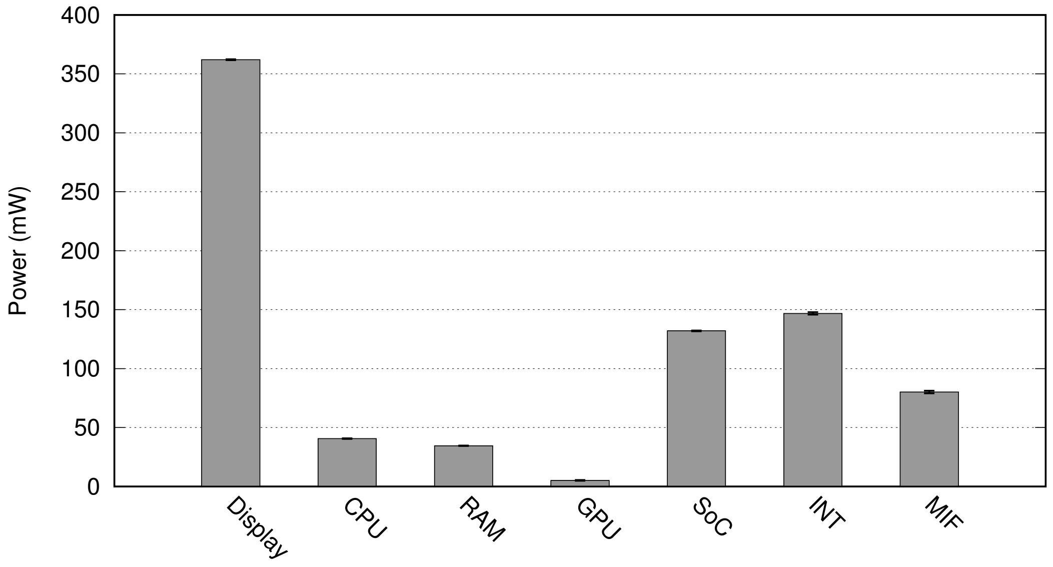


Figure 4.8: S3 power while in the idle state in airplane mode.

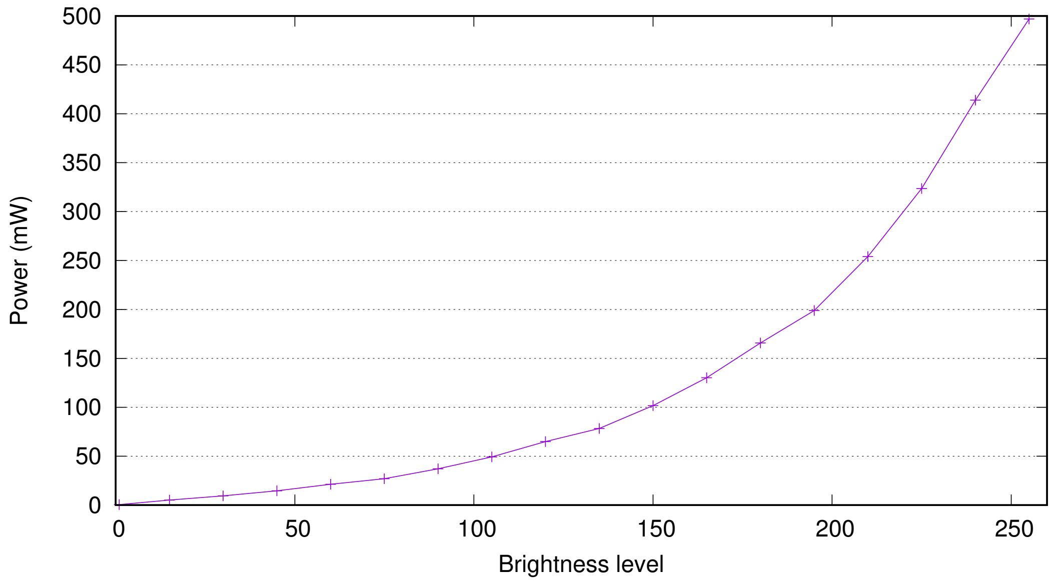


Figure 4.9: Freerunner display backlight power at varying brightness levels.

an image by filtering light generated behind the panel by an LED backlight. Thus on the Freerunner, the energy contribution due to display brightness and content are independent. On the other hand, the S3 display is an emissive OLED (organic light-emitting diode) type. For these displays, each pixel generates light of a particular colour and brightness, so the content and brightness are highly dependent.

Figure 4.9 shows the power consumption of the Freerunner's backlight over the range of available brightness levels. That level is an integer value between 1 and 255, programmed into the power-management module, used to control backlight current. Android's brightness-control user-interface provides linear control of this value between 30 and 255, which corresponds to 9 mW and 497 mW backlight power respectively. At half brightness (centred slider), the backlight consumes 90 mW. For the LCD panel itself, we measure 40 mW for a completely white screen, and 89 mW for a black screen. Display content can therefore affect overall power consumption by up to 49 mW. In addition to the panel and backlight, power is consumed in delivering pixels to the screen by the 2D GPU. Comparing the idle and suspend cases, we see a 90 mW increase in GPU power consumption when the display is enabled.

On the S3, understanding display power is more complicated. Power consumption of an OLED display has a constant component for the display driver, plus a dynamic component dependent on both display content and brightness [DCZ09]. We measure the S3 display to consume a constant 223 mW. For a completely black display (i.e. no pixels actuated) we see an additional 1.9–3.1 mW (for minimum and maximum brightness, respectively). For a completely white screen, the power consumption is an additional 84 mW at minimum brightness,

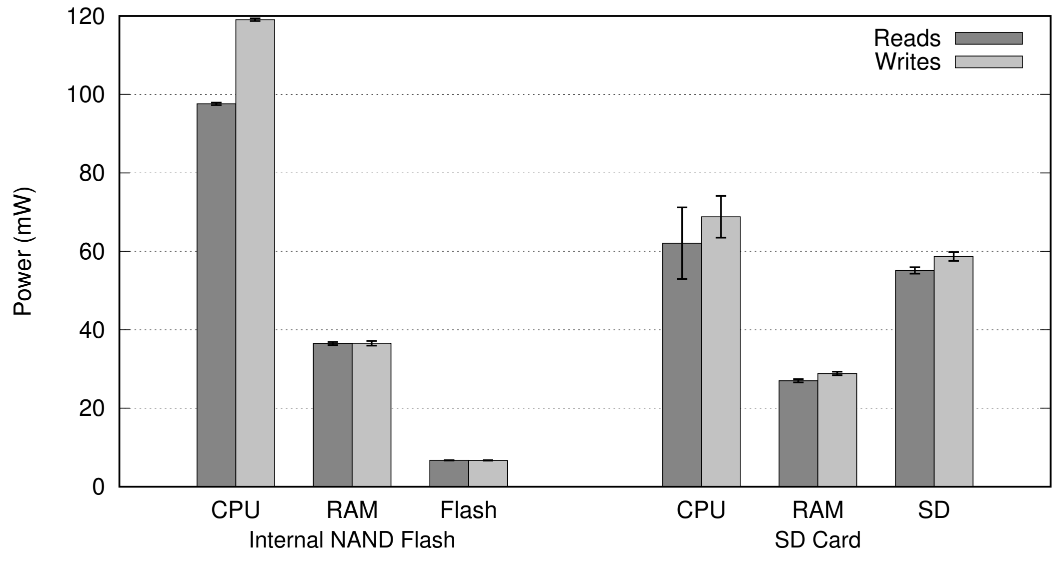


Figure 4.10: SD, flash, CPU and RAM power on Freerunner for flash storage read and write benchmarks.

Metric	Flash	SD
Idle (mW)	0.5	1.7
Read		
throughput (MiB/s)	$4.7 \pm 0.8\%$	$2.3 \pm 2.0\%$
efficiency (MiB/J)	$33.6 \pm 0.9\%$	$16.0 \pm 5.9\%$
Write		
throughput (MiB/s)	$0.91 \pm 0.5\%$	$0.30 \pm 2.0\%$
efficiency (MiB/J)	$5.6 \pm 0.8\%$	$1.9 \pm 2.8\%$

Table 4.6: Freerunner flash storage power and performance.

and 1027 mW at maximum brightness.

4.4.4 Flash storage

Bulk storage on the Freerunner device is provided by 256 MiB of internal NAND flash, and an external micro Secure Digital (SD) card slot. To measure their maximum power consumption, we use the Linux `dd` program to perform streaming reads and writes. For reads we copy a 64 MiB file, filled with random data, to `/dev/null` in 4 KiB blocks. For writes, 8 MiB of random data is written, with an `fsync` between successive 4 KiB blocks to ensure predictability of writes. Between each iteration we force a flush of the page cache.

Figure 4.10 shows the power consumed by the NAND flash and SD card, as well as the CPU and RAM which are required to drive the benchmark. Table 4.6 shows the corresponding

	Power (mW)		
Benchmark	Measured	Predicted	Std. dev. (%)
Camera (video)	16.5	51.1	8.6
Email	4.6	14.3	13.4
Call	2.0	6.2	33.3
Camera (still)	1.6	5.0	44.5
Web	1.4	4.3	9.2
Gaming	1.3	4.0	26.3
SMS	1.3	4.0	24.9
Audio	1.0	3.1	13.7
Video	0.4	1.2	36.8

Table 4.7: S3 predicted worst-case flash power.

data throughput, efficiency (including flash/SD, CPU and RAM), and idle power consumption. For the flash device, equal power is consumed for write and reads (6.7 mW), despite read throughput exceeding writes by more than 5 times. The same is true for the SD card. However, SD power exceeds flash power by an order of magnitude, while achieving less than half the throughput. Including the CPU and RAM power, flash is 2–3 times more efficient than SD.

The S3 features an internal NAND flash, but does not support an external SD card. As previously discussed in Section 4.3.2, we were unable to make direct measurements of the flash storage. Table 4.7 shows the predicted flash power across the benchmarks using the worst-case power model. With the exception of the camera benchmarks, where flash power can be significant, we ignore the contribution of flash on S3, since even the worst-case estimated power is trivial.

4.4.5 Network

In this benchmark we stress the networking components of the devices: WiFi and cellular. On Freerunner, the test consists of downloading a file via HTTP using `wget`. The files contain random data, and are 15 MiB for WiFi, and 50 KiB for GPRS. On S3, we use a common network benchmarking Android application, *speedtest.net*, to benchmark the 3G subsystem. To average out variations due to factors outside our control, network-operator load, environmental variations, signal strength etc., we perform the iterations spread out over a 24 hour period.

Table 4.8 shows the results on both platforms in terms of throughput, power consumption, and network efficiency—that is, the number of bits sent/received per unit energy consumed (higher is better). In terms of download, all three network technologies (2G/GPRS, 3G/UMTS and WiFi) have similar power consumption despite vastly different bandwidths—a factor of 200 between GPRS and WiFi on Freerunner. Unsurprisingly, both power and bandwidth fluctuate significantly, particularly so on the cellular radios.

	Galaxy S3 (3G)		Freerunner	
	Upload	Download	Download (GPRS)	Download (WiFi)
Throughput (kbps)	547 ± 149	1317 ± 814	30 ± 8	5280 ± 296
Radio power (mW)	1137 ± 372	768 ± 64	636 ± 81	874 ± 30
Efficiency (kb/J)	481	1715	47	6041

Table 4.8: Network power and performance (average ± std. dev.)

To test the effect of signal strength on power and throughput, we re-run the Freerunner benchmarks with the device shielded within a metal box of 2 mm thickness. Over GPRS, this results in an increase in cellular power of 30 %, but no effect on throughput. The shielding results in a reported signal strength drop of 10 dBm. Over WiFi, the signal strength drops by only 2 dBm, and no effect on throughput or power consumption is observed.

4.4.6 GPS

To measure power consumption of the GPS subsystem, we enabled the module and ran an Android application (`GPS Status 2` on Freerunner and `Data Monitor` on S3) to collect location information, and compare idle and active power. The Freerunner features an external GPS antenna connector, so on that platform we also measure the power consumption using such an antenna. On S3, we are unable to directly measure GPS power consumption, because our instrumentation requires opening the case, which disconnects the antenna. Instead, we use a different, uninstrumented Galaxy S3 (a slightly different model, the GT-I9305) and compare total system power consumption with and without the GPS enabled.

On Freerunner, we measure 171 mW using the internal antenna, which slightly increased to 199 mW while using the external antenna. These measurements are extremely stable (< 1 % standard deviation), independent of signal and time—neither the number of satellites, the signal strength, nor the tracking time have any appreciable effect. This observation is contrary to the part's data sheet [u-b06], which specifies that power consumption should drop by approximately 30 % after satellite acquisition. It is unclear why we do not see such behaviour, but experience suggests that an error in hardware integration or software is not unlikely.

On S3, we do observe a difference between acquisition and tracking power, but surprisingly, tracking power was *higher*. We measured 386 ± 20 mW in acquisition mode, which jumped to 433 ± 22 mW in tracking: acquisition mode is the time until a position is reported, and tracking is the time thereafter. We also experimented with both hot and cold GPS state, but find no significant difference in average power consumption. However, with cold GPS state, the device takes approximately 400 seconds to acquire a fix, consuming 154 J of energy. For warm state (5–10 minutes old), this reduces to approximately 30 seconds.

4.4.7 Sensors

The Galaxy S3 supports a number of environmental sensors, measuring acceleration, orientation, light intensity, air pressure, and physical proximity. To determine the power draw while using these sensors, we configure the `Data Monitor` application to sample them at 50 Hz (one sensor at a time). We then compare the total power measured against a scenario with the same application sampling a "dummy" sensor, which simply delivers software events at a rate of 50 Hz, but involves no additional hardware. This allows us to isolate the contribution of the sensor itself, without including the idle power, the cost of running the sampling software, etc. In Table 4.9, we display the increase in power consumption for each sensor above the dummy, averaged across 3 runs of 60 seconds each. In all cases, the device is in airplane mode with the screen turned off.

Sensor	Power (mW)
Accelerometer	5 ± 2.3
Gyroscope	30 ± 1.3
Light	3 ± 1.7
Magnetometer	12 ± 0.6
Barometer	1 ± 0.7
Proximity	7 ± 2.2
Dummy	573 ± 0.6

Table 4.9: S3 sensor power consumption (average ± std. dev.) above the "dummy" sensor.

The power consumption of the sensors is clearly small: at most 30 mW, with most of them using under 10 mW. This corresponds to less than 6 % of the total system power of approximately 580 mW. The cost of using the sensors therefore is not the power consumed by the sensor hardware, but rather the cost of having the entire system online.

4.4.8 Audio

This benchmark is designed to measure power in a system being used as a portable media player. We use the in-built media application to play a sample MP3 to a pair of stereo headphones. The measurements are taken with the backlight off, which is representative of the typical case of someone listening to music or podcasts while carrying the phone in their pocket.

Figures 4.11 and 4.12 shows the power breakdown for this benchmark on Freerunner and S3 respectively. The audio subsystems (amplifier and codec) consume 20–40 mW. On Freerunner we can break this down: about 60 % of audio power goes to the codec, and the remaining 40 % to the amplifier. Overall, the audio subsystem accounts for 12–20 % of total system power. We investigate the effect of output volume by comparing the results at both maximum and minimum volume, and notice very little difference. On Freerunner we see amplifier power increase by 5 mW at maximum volume, but due to (perhaps incidental) decreases in other

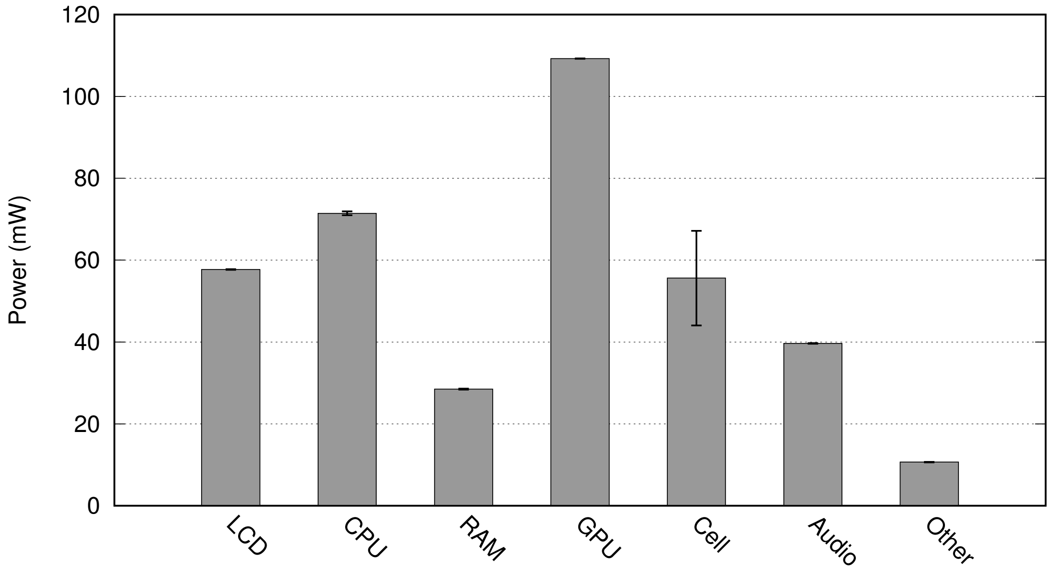


Figure 4.11: Freerunner audio playback power breakdown.

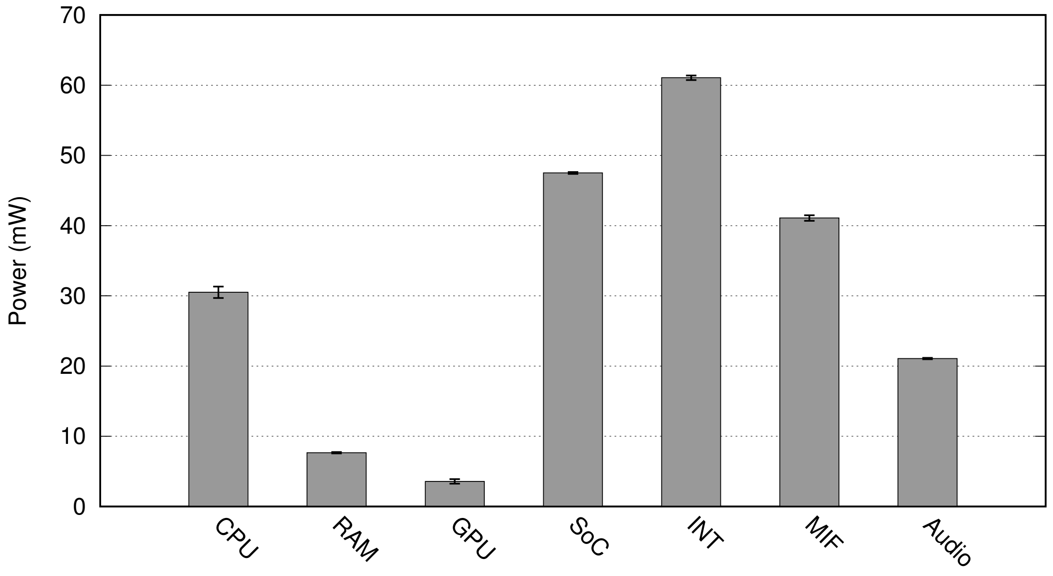


Figure 4.12: S3 audio playback power breakdown.

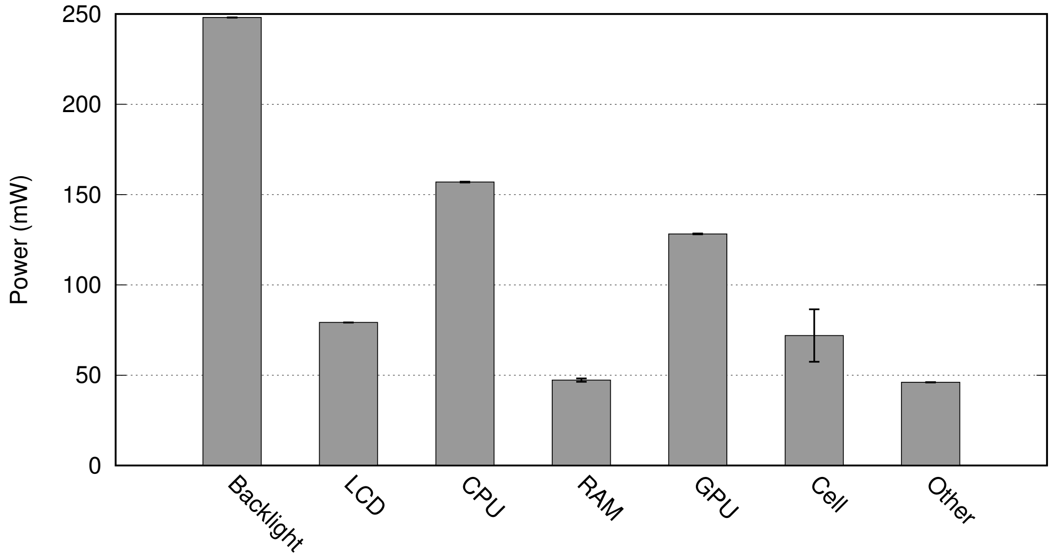


Figure 4.13: Freerunner video playback power breakdown.

supplies, the total system power consumption increases by only 1 mW. On S3, we see a total increase of 5 mW.

On Freerunner, we also experiment with the cost of loading the audio file from storage, in our case, from the SD card. We do this by forcing a flush of the buffer cache between iterations, and the result is a negligible 2 % increase in total power consumption.

4.4.9 Video

In this benchmark we measured the power requirements for playing a video file. For each platform, we used an encoding contemporary with the device: H.263 on Freerunner and H.264 on S3, at native resolution. On S3 we also experimented with video at different bit rates: a high quality one at 9084 kbps, and low quality at 1013 kbps. The S3 features a hardware unit capable of decoding H.264, so we also compared the cases of software and hardware decoding on that platform. On Freerunner we used the in-built camera application to play the video and on S3 we use *MX Player* v1.6j. The power breakdown for the Freerunner benchmark is shown in Figure 4.13, and the S3 results for low- and high-quality respectively are shown in Figures 4.14 and 4.15.

On Freerunner the CPU power is (unsurprisingly) high, since video decoding is a compute-intensive workload. We also see increases in RAM and GPU power. However the display subsystem is still consuming 40 % of total system power.

The S3 results show the importance of specialized decoder hardware for modern video codecs and bitrates: it reduces total system power consumption by 30 % and 45 % for low and

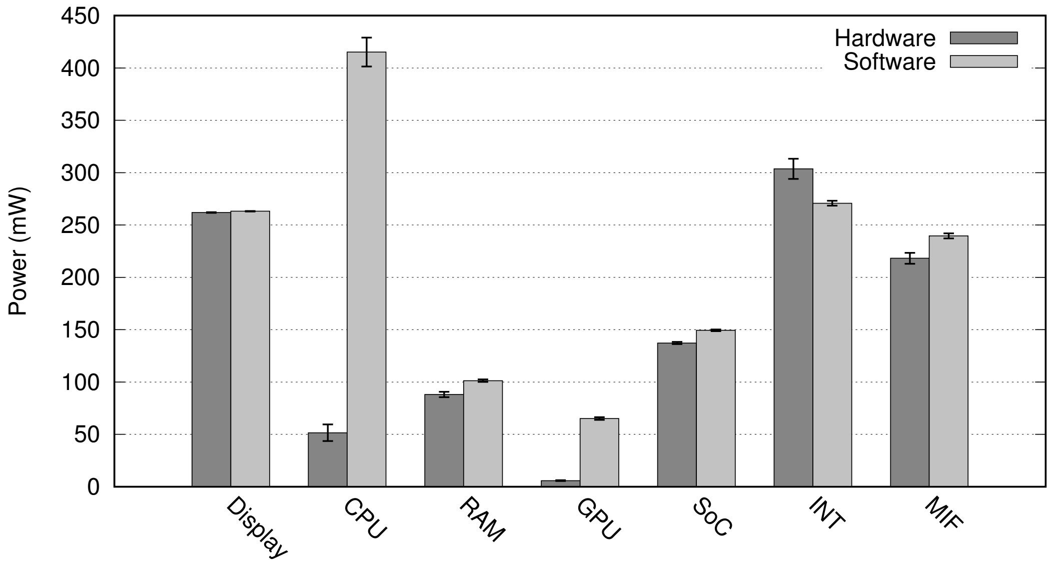


Figure 4.14: S3 low-quality video playback power breakdown.

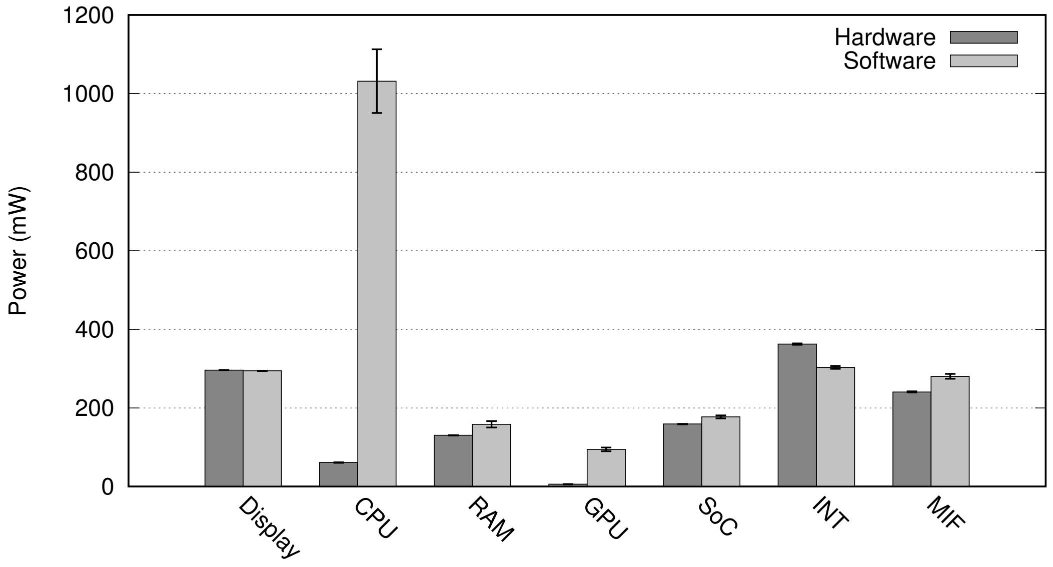


Figure 4.15: S3 high-quality video playback power breakdown.

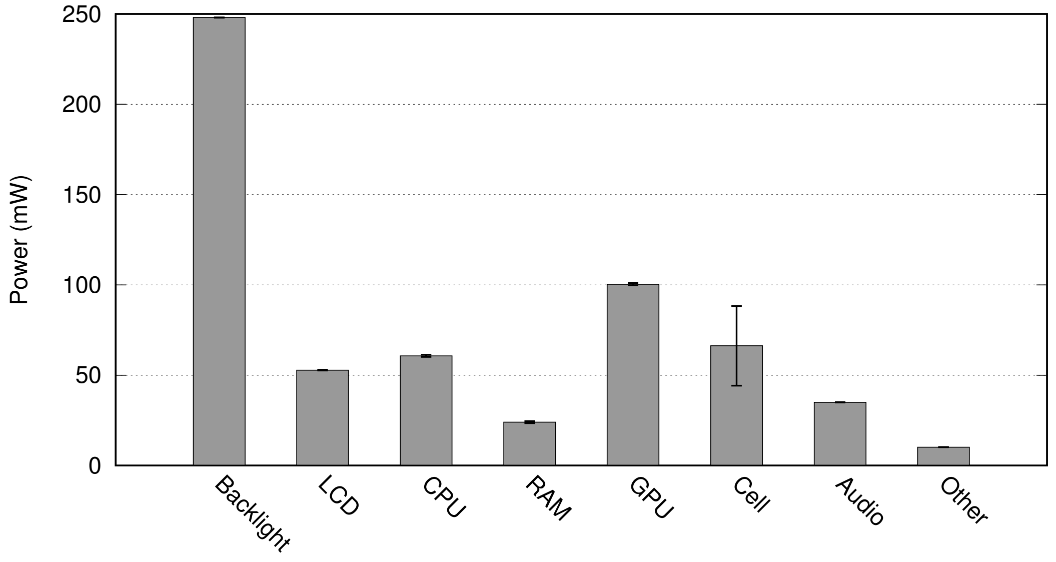


Figure 4.16: Freerunner power breakdown for sending an SMS.

high quality, respectively. Comparing the software- and hardware-decoded cases, we see that at both quality levels, software decoding results in power increases on every supply, with the exception of INT. This suggests that the hardware video decoding unit is powered from this supply.

4.4.10 Text messaging

We benchmarked the cost of sending an SMS by using a trace of real phone usage with the record/replay tools described in Section 4.2.2. The trace consists of loading the contacts application and selecting a contact, typing and sending a message, then returning to the home screen, lasting a total of approximately 1 minute. To ensure the full cost power cost is accounted for, we include the time taken for the cellular transaction after the SMS app is closed, which can take up to an additional 20 seconds.

The results for SMS messaging on Freerunner and S3 are shown in Figures 4.16 and 4.17 respectively. On Freerunner, energy consumed is again dominated by the display components. The cellular radio, which carries the SMS transmission, shows an average power only 8 mW greater than idle over the full length of the benchmark, accounting for 11 % of the aggregate power. On the S3, the cellular subsystem draws similar power, 60 mW average, but due to the increased idle power on S3, this accounts for only 5 % of total power. On the other hand, the total SoC power increase compared with idle is 226 mW; the energy cost of running the SMS app far exceeds the radio cost to send the message.

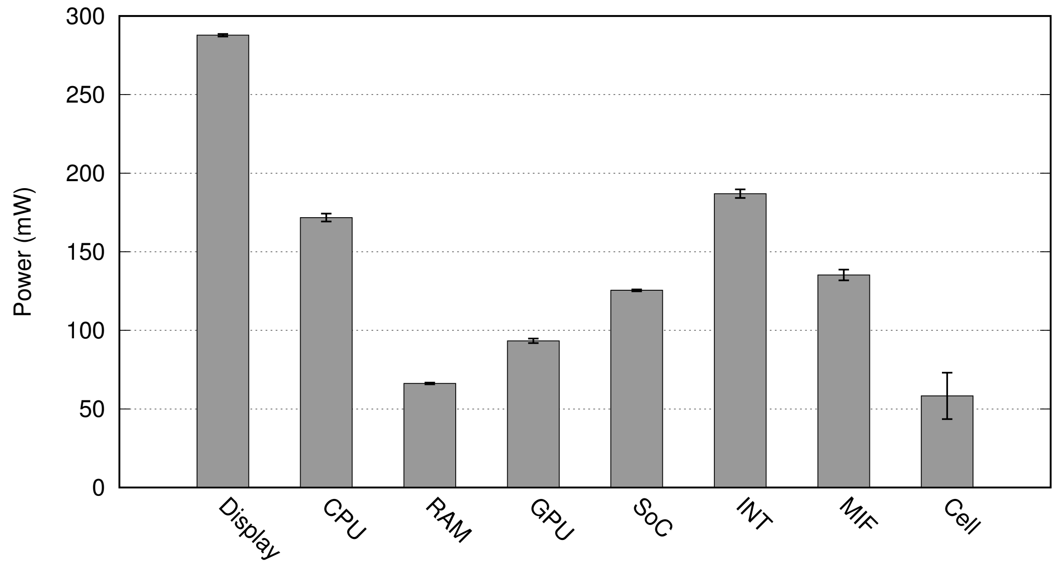


Figure 4.17: S3 power breakdown for sending an SMS.

4.4.11 Phone call

We measured the energy consumption while making a phone call, again using the trace record and replay tools. The trace includes loading the dialer application, dialing a number, then making a 1 minute call. This breaks down as approximately 10 seconds to connect the call, 10 seconds for the receiving device to accept, and 40 seconds of talk time.

The power of this benchmark is graphed in Figures 4.18 and 4.19 for Freerunner and S3 respectively. On both devices, the cellular subsystem dominates significantly at 832 mW and 566 mW, approximately 10 times more than any other subsystem, and accounting for about 70 % of total power on both platforms. Display and backlight contribution is lower than other uses cases, since Android disables the screen during the call. On Freerunner, this is timer-based, and on S3 the proximity sensor is used to detect when the user has put the phone to their ear. Moreover, the S3 supports a pass-through mode, whereby the cellular subsystem directly controls any necessary peripherals (microphone, speaker, etc.) during a call, allowing the applications processor to enter a sleep state. This is why most of the supplies consume less energy during a phone call than even the idle state.

4.4.12 Email

For this benchmark, we used Android's email application to measure the cost of sending and receiving emails. The workload consists of opening the email application, downloading and reading 5 emails (one of which includes a 60 KiB image) and replying to two of them. We use

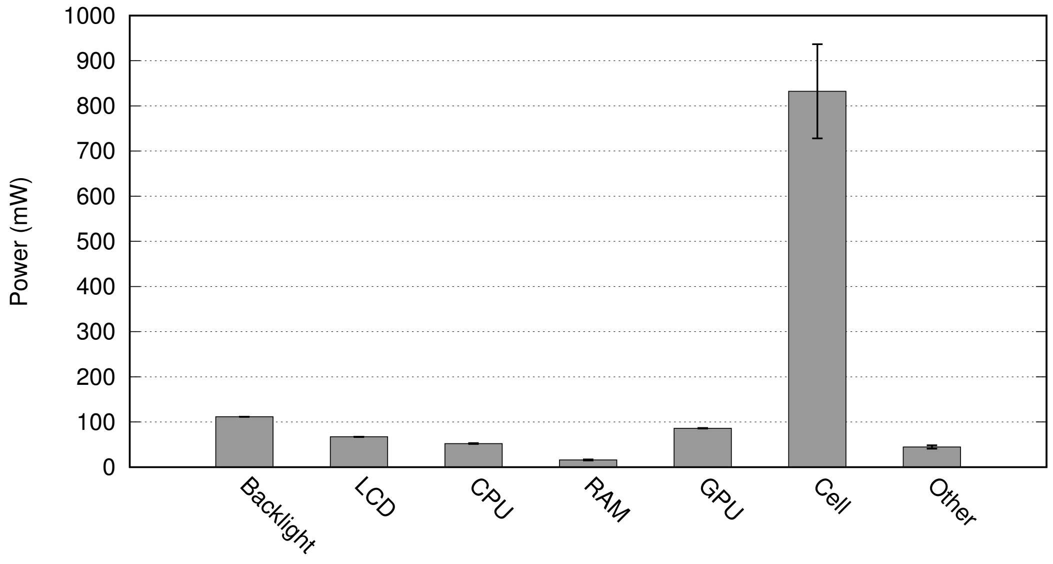


Figure 4.18: Freerunner cellular phone call average power.

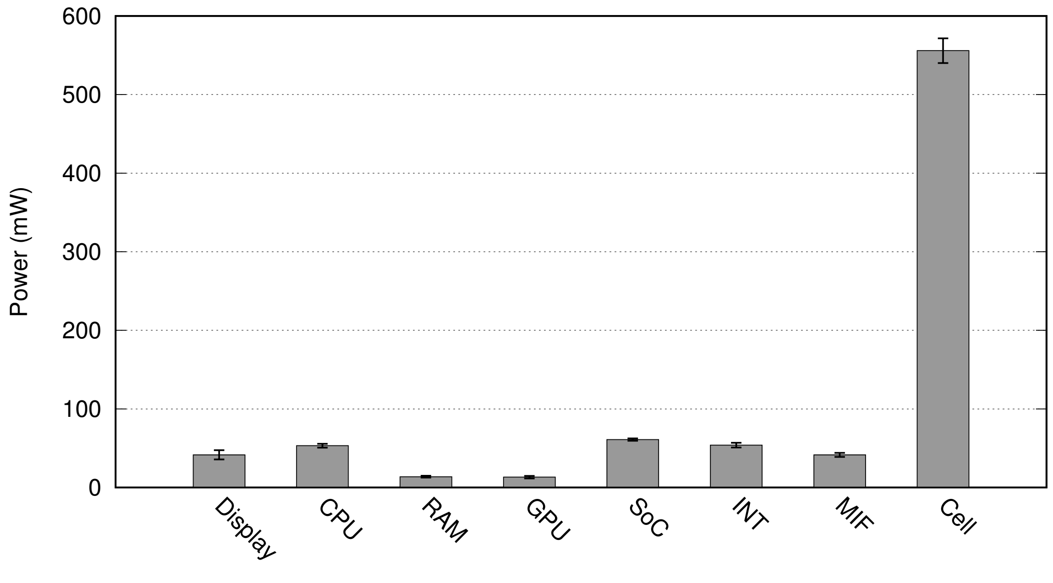


Figure 4.19: S3 phone call average power.

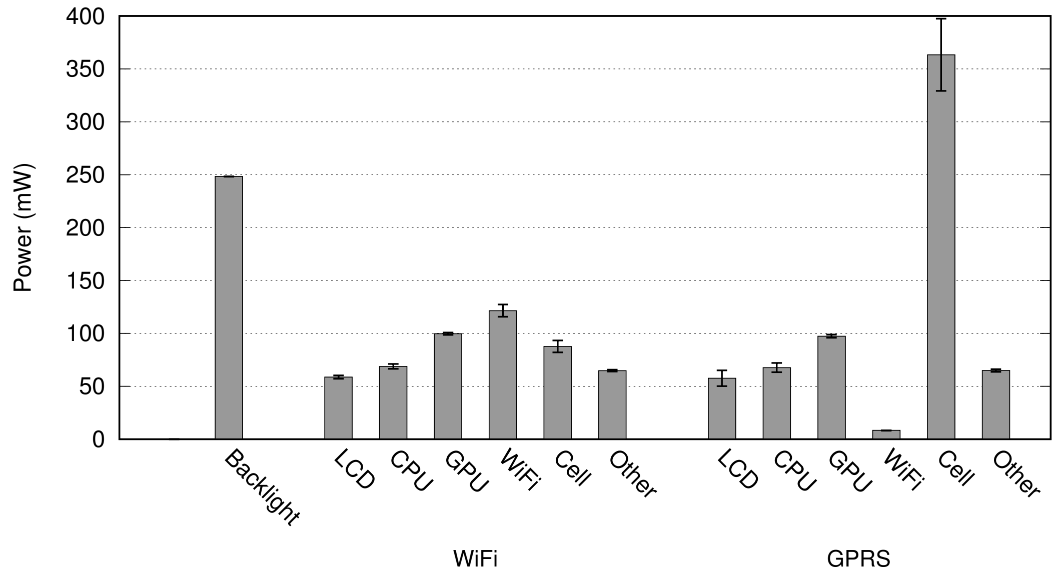


Figure 4.20: Freerunner average power for the email macro-benchmark.

the input tracing tools to drive the benchmark, and after each iteration reset the state of the email server and local email cache. On both platforms we repeat the experiment using both a cellular data connection and WiFi. The results are shown in Figures 4.20 and 4.21.

The power breakdown between the GPRS and WiFi benchmarks is very similar, except for the cellular and WiFi radios themselves. Despite presenting identical workloads to the radios, using a cellular network uses 3–5 times the power of a WiFi connection. Moreover, while the 3G radio on the S3 is a higher-bandwidth interface, it uses lower average power than the slower GPRS radio on the Freerunner. We also see a somewhat elevated level of activity in the SoC supplies, since email includes significant UI interaction.

4.4.13 Web browsing

This benchmark measures a web-browsing workload using both cellular and WiFi connections. It is trace-based, and consists of loading the browser application, selecting a bookmarked web site, and several minutes of browsing pages. We used a snapshot of the BBC News mobile website which we mirrored to improve the reliability of the benchmark. After each run, the browser cache was cleared.

The results are shown in Figures 4.22 and 4.23. Cellular consumes more power than WiFi by about a factor of 3 on both platforms. As in the email scenario, the other components do not display any significant difference in consumption between the cellular and WiFi cases. However, in the case of web browsing, the higher-bandwidth 3G radio interface on the S3 consumes higher power (45 mW average) than the Freerunner's GPRS radio.

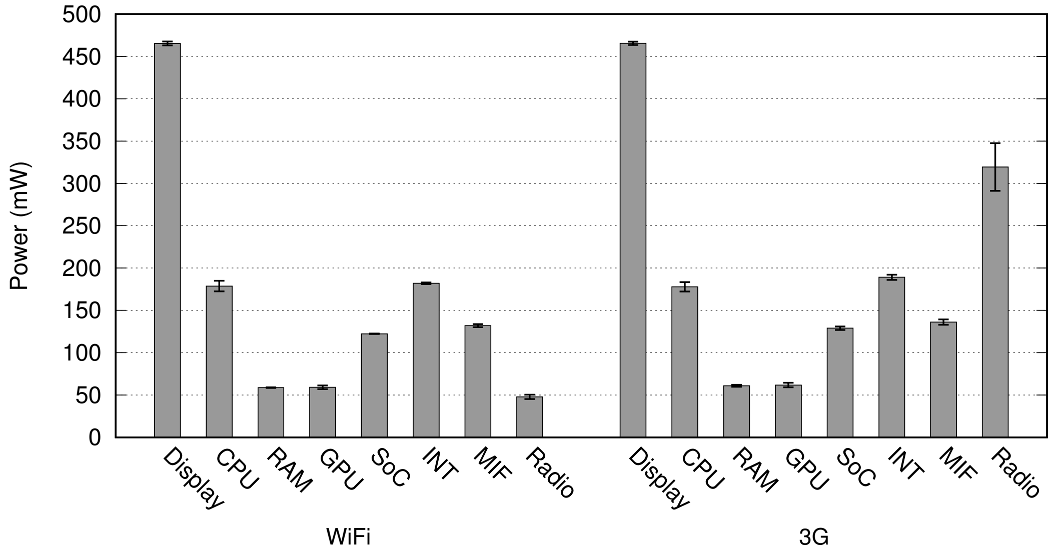


Figure 4.21: S3 average power for the email macro-benchmark.

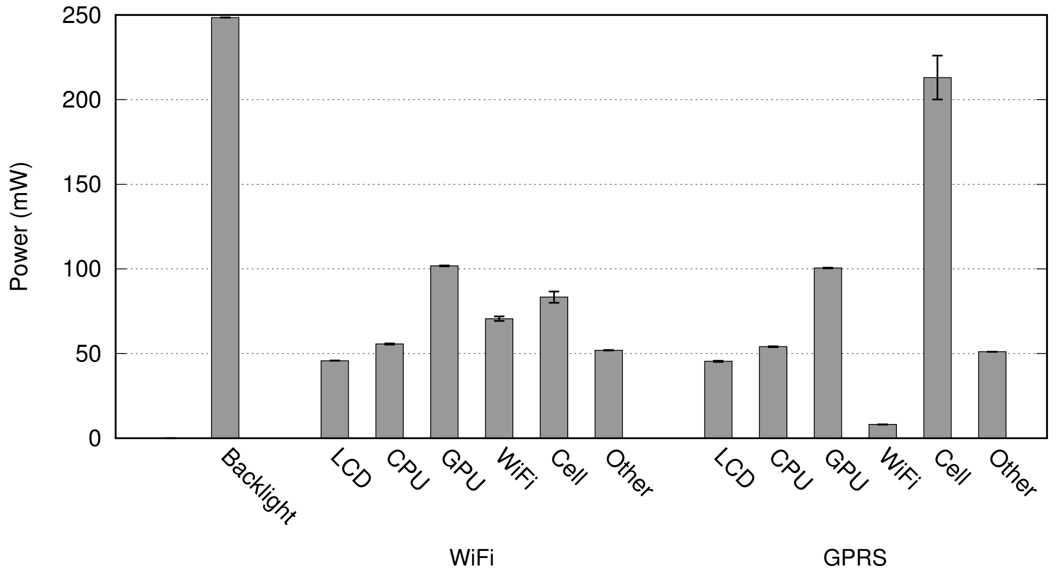


Figure 4.22: Freerunner web browsing average power over WiFi and GPRS.

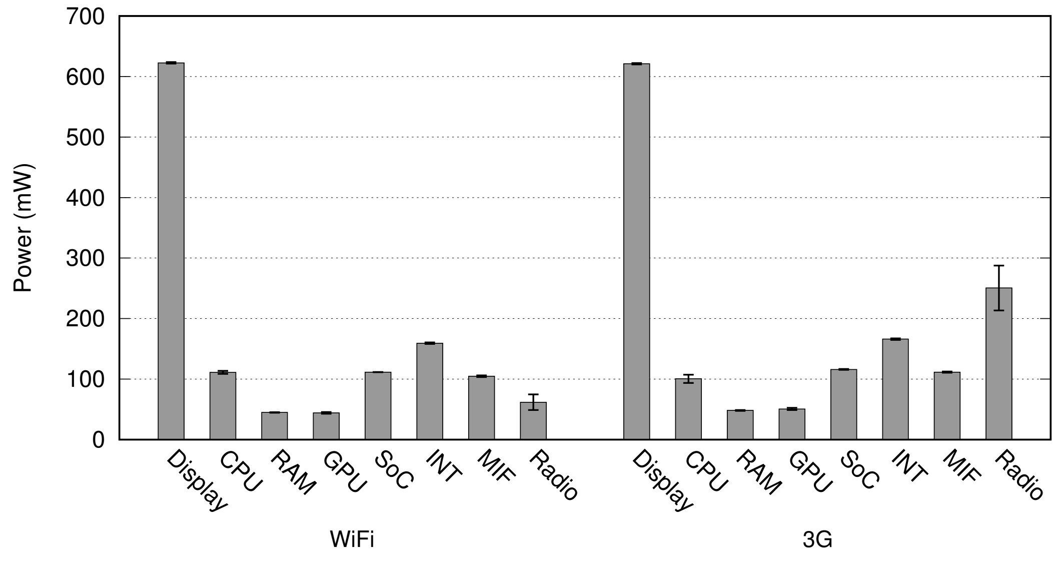


Figure 4.23: S3 web browsing average power over WiFi and 3G.

4.4.14 Gaming

On S3, we measured two gaming scenarios. The first is the 2D game *Angry Birds*. The workload consists of loading the application and playing through two levels using the trace record/replay tools for input, for a total 1.5 minutes of gaming. The second is the graphics-intensive 3D racing game *Need for Speed Most Wanted*. Data is collected while playing the game for a period of around 2 minutes. In the latter case, we provided input manually. We attempted a trace-based benchmark, but found that despite identical input, the results were wildly inconsistent between iterations in terms of game state (i.e. where the car went on the track). We hypothesize that this is due to small variations in execution time of the game engine, causing the game state to diverge with respect to the timing of the input event delivery. This does not occur on any other scenario because they all have periods of idleness: any variation in execution time is absorbed by the idle period, implicitly synchronising the benchmark with the input.

The results of the two gaming scenarios are shown in Figure 4.24. Unsurprisingly we see elevated power consumption across all components, with Angry Birds consuming significantly less power than Need for Speed (1483 and 2400 mW respectively). The GPU in the latter case is particularly power-hungry, using 767 mW (32 % of total).

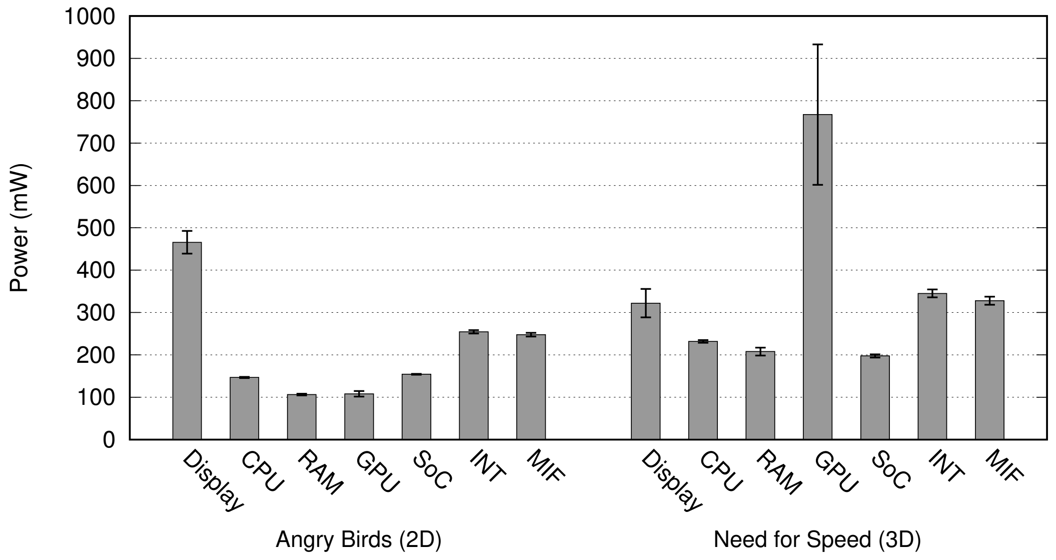


Figure 4.24: S3 gaming power consumption.

4.4.15 Camera

On the S3, we benchmarked two scenarios that make use of the camera: taking a still photograph, and recording a short video. For the still case, the scenario is opening the camera app, focusing on a near-ground object, taking the photograph, and finally viewing the image taken. In the video scenario, we record a 20 s video but do not view it in the benchmark. The results are shown in Figure 4.25.

The total power consumed in the still scenario is 2256 mW, while video consumes 2614 mW. This makes video recording the most power-hungry of all our measured scenarios. As discussed in Section 4.3, the flash[2] storage results are worst-case estimates, but we find that it is still negligible to overall consumption. With the exception of the GPU, we see significant increase in total SoC power consumption across all supplies, with an additional 400–650 mW being consumed in the camera subsystem.

4.5 Validation

We have measured, at a fine-grained per-component level, the energy consumption of two devices. In this section, we attempt to put these measurements into context by comparing the results with measurements taken on two additional smartphones: the HTC Dream (G1), and the Google Nexus One (N1). The aim of this exercise is to evaluate 1) whether our mea-

[2]Throughout this discussion, flash refers to NAND flash storage, not the LED camera flash which is unused in our experiments.

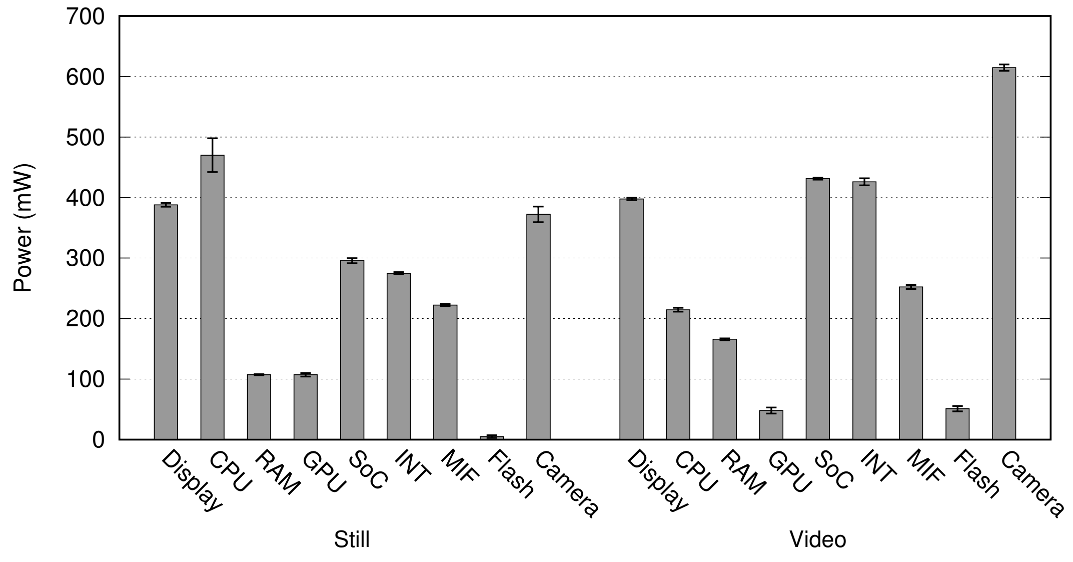


Figure 4.25: S3 camera power consumption.

	G1	N1
SoC	Qualcomm MSM7201	Qualcomm QSD 8250
CPU	ARM 11 @ 528 MHz	Scorpion @ 1 GHz
RAM	192 MiB	512 MiB
Display	3.2” TFT, 320x480	3.7” OLED, 480x800
Radio	3G UMTS+HSPA	3G UMTS+HSPA
OS	Android 1.6	Android 2.1
Kernel	Linux 2.6.29	Linux 2.6.29

Table 4.10: G1 and N1 technical specifications.

surements are representative; and 2) determine power consumption trends across smartphone generations. However, rather than perform a complete instrumentation of these devices, which is very resource-intensive, we instead measure only the total system power consumption, i.e. energy flow at the battery terminals. We use the same methodology to measure this power as in the previous section. Adding the G1 and N1 results to what we already have for the Freerunner and S3 gives coverage of approximately 5 years of smartphone technology.

Table 4.10 lists the key features of the G1 and N1 devices. Both are 3G Android smartphones with single-core CPUs. The N1 is a significantly higher-spec device, with approximately 3 times more RAM, 2.5 times higher clock speed on a newer architecture, and a larger screen with a factor of 2.5 times the screen pixel count at 1.4 times the pixel density. The G1 features an LCD-type display, while the N1 has an OLED-type. Uniquely among these devices, the G1 features a physical slide-out QWERTY keyboard, whereas the others use on-

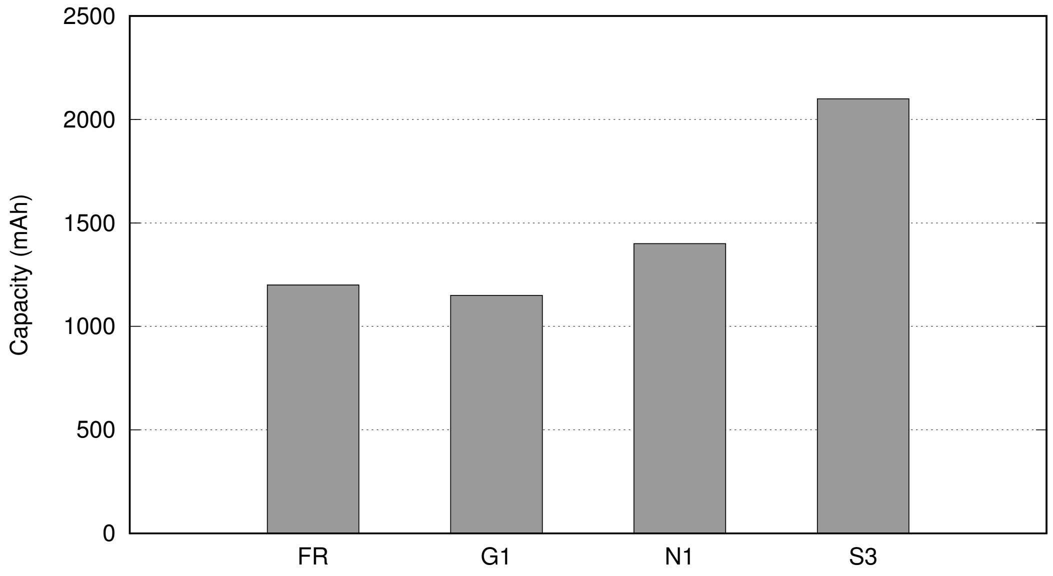


Figure 4.26: Battery capacity for each device.

screen software keyboards. The battery capacity for each device is shown in Figure 4.26.

In this section, we first look at the display/backlight power, and then measure Bluetooth, which we were unable to do on the Freerunner or S3. We then compare the total system power consumption for the usage scenarios across the four devices. We analyse the measurements gathered here in Section 4.6.2.

4.5.1 Display

The G1 device features an LCD-type display, so like the Freerunner it is lit with a backlight. It also features two additional backlights for the physical buttons: one for the keyboard, and one for the control buttons (menu, call, etc.) Figure 4.27 plots the power consumption of the various backlights on the G1 as a function of brightness level. The screen backlight power is controlled linearly with the brightness setting, whereas the keyboard and button backlights have no brightness control, only "on" and "off" settings. When both are enabled, they contribute 189 mW. The screen backlight varies between 20 mW at minimum brightness, and 552 mW at maximum. While the backlight power itself is independent of the display content, it can affect LCD power consumption by up to 17 mW.

As with the S3, the N1 OLED display does not have a separate backlight, but individually lit pixels. Thus the effect of display content and brightness on power consumption are tightly coupled. We measured the power consumption of a completely white screen at minimum brightness at 194 mW, and at maximum brightness it is 1313 mW. For a completely black screen, power is independent of brightness.

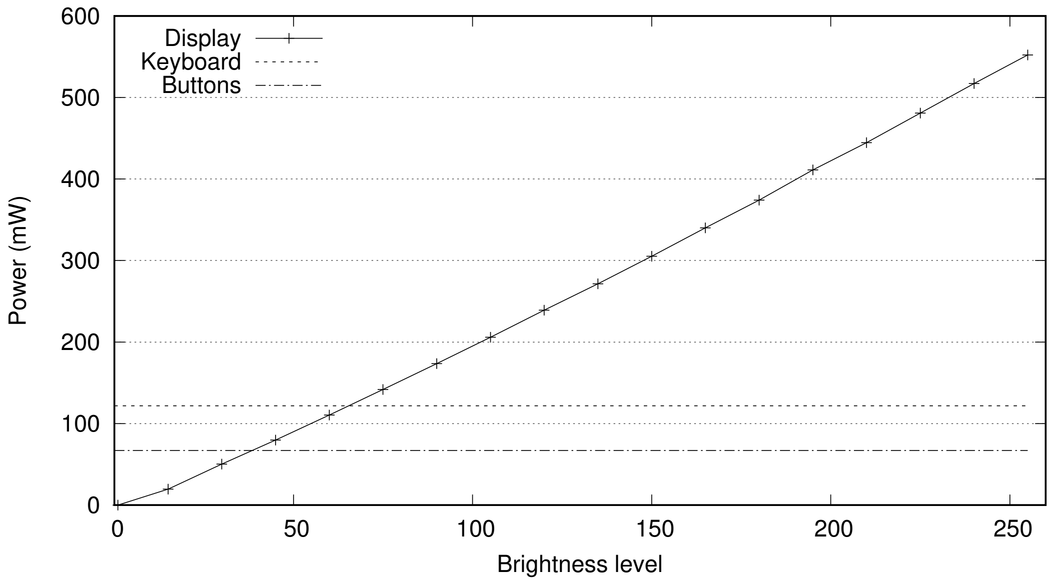


Figure 4.27: Display, button and keyboard backlight power on the G1.

4.5.2 CPU

To investigate the trends in peak CPU power consumption, we perform compute-intensive micro-benchmarking to find a workload that maximally exercises the CPU. On Freerunner, we find that the *equake* benchmark from the SPEC CPU2000 suite elicits worst-case power consumption in the CPU. This benchmark does not significantly stress any other component of the system, including RAM, so we are also able to determine the CPU power for this benchmark on the G1 and N1 devices subtractively. For the S3, which is a multi-core device, we find that the multi-threaded *AnTuTu Benchmark* yielded maximum power. These results are shown in Figure 4.28.

4.5.3 Bluetooth

As noted earlier, we are unable to get Bluetooth working reliably on the Freerunner phone, and we can not get fine-grained measurements on the S3. To get an idea of Bluetooth power consumption, we run the audio benchmark on the G1 with the audio output to 1) a Bluetooth stereo headset; and 2) headphones at minimum volume. We determined from our earlier benchmarks that power consumed in the audio subsystems is largely static, so the difference between the headset and Bluetooth audio results should yield consumption of the Bluetooth module.

Table 4.11 shows the Bluetooth and total system power consumption for the audio benchmarks. In the ”near” benchmark, the headset is placed approximately 30 cm from the phone, and about 10 m in the ”far” benchmark. The Bluetooth module consumes fairly minimal

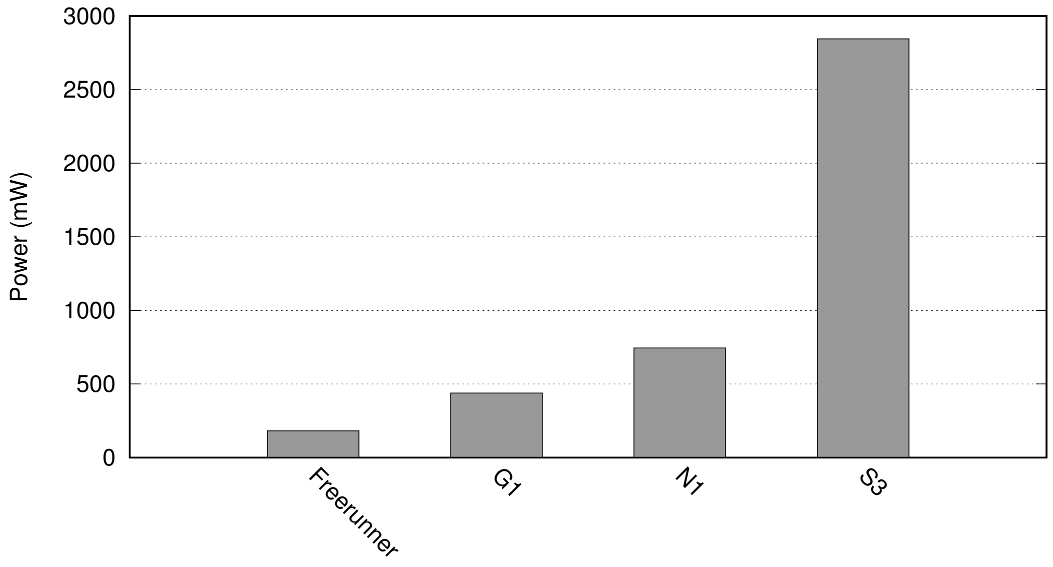


Figure 4.28: Peak CPU power consumption.

	Power (mW)	
Benchmark	Total	Bluetooth
Audio baseline	460	-
Bluetooth (near)	496	36
Bluetooth (far)	505	45

Table 4.11: G1 Bluetooth power under the audio benchmark.

power: 7 % for the case where the devices are placed near each other, and 9 % when they are placed far away. For use cases where the screen is on, Bluetooth would consume an even smaller fraction of total power.

4.5.4 Usage Scenarios

Table 4.13 shows total system power consumption for the Freerunner, G1, N1 and S3 for our macro-benchmarks. The setup of the G1 and N1 in these scenarios is the same as the Freerunner. The brightness-dependent component of display power consumption has been subtracted out, because it is highly dependent on the user's brightness setting. It is trivial to determine this for backlit LCD displays, but for the OLED it is a more complicated calculation. We show in Table 4.12 the additional power consumption of the OLED display at minimum and maximum brightness levels for several of the benchmarks.

	OLED Power (mW)	
Benchmark	Min.	Max.
Idle	38.0	257.3
Phone call	16.7	112.9
Web	164.2	1111.7
Video	15.1	102.0

Table 4.12: N1 OLED display additional power at maximum and minimum brightness.

	Average System Power (mW)			
Benchmark	Freerunner	G1	N1	S3
Suspend	103	27	25	24
Idle	334	161	334	666
Phone call	1135	822	747	854
Email (cell)	691	599	-	1299
Email (WiFi)	506	349	-	1020
Web (cell)	500	430	538	1080
Web (WiFi)	430	271	412	874
Video	559	568	526	1044†
Audio	419	460	322	226

Table 4.13: Freerunner, G1, N1 and S3 total system power. The brightness-dependent component of power has been subtracted out. †The S3 results use a video of higher resolution to match the significantly larger screen.

4.6 Analysis

4.6.1 Where does the energy go?

Display Display power is a significant contributor in every scenario where the display is active, even at the 50 % brightness level we used in these benchmarks. In all except the most cellular-intensive benchmarks, the brightness of the display is the most critical factor in determining overall power consumption. However, this is a relatively simple component from a power-management perspective, and largely depends on the user's brightness preference. Our results confirm that aggressive backlight dimming can save a significant amount of energy, and further motivates the inclusion of ambient light and proximity sensors in mobile devices to assist with selecting an appropriate brightness. On OLED displays, power is highly-dependent on the image content. For example, on S3, just varying the colour scheme (e.g. to light-on-dark) can save up to 300 mW at medium brightness, or 600 mW at maximum brightness. On LCD-based devices, the screen content has a small but still measurable effect on display power. However this is completely independent of the brightness setting.

Network The networking subsystems, namely the cellular and WiFi radios, are historically considered to be energy-intensive. In scenarios where these are used—web, email, phone call, SMS—we indeed see non-trivial network energy consumption. WiFi is always more energy efficient than the cellular technologies for the usage scenarios, however we found that the cost to maintain an unused data connection is lower for 3G than WiFi: on S3, approximately 10 mW for 3G, compared with 50 mW for WiFi. In the phone call scenario, the cellular radio accounts for a higher fraction of overall power consumption that any other case. However on S3, this scenario is among the lowest in total power consumption. It is in fact more efficient to make a call on this device than to send an SMS. This is because the display can be disabled during the call, which is a particularly significant contributor on S3.

CPU We see the CPU contributing substantially in almost every scenario, with the exception of idle and (most of) phone call, where it is powered off. This is consistent with the fact that the CPU is involved in most operations, such as through updating the user interface or coordinating another component's operation with the device driver. The CPU is the largest contributor in only the most computationally-demanding workloads. However, the peak power results show that in such scenarios, the CPU can consume a very large amount of energy.

Camera The camera workloads show particularly high power, not only in the camera subsystem, but also across the various SoC supplies, the CPU core, and RAM. Surprisingly, these two scenarios are among the highest energy consumers across all our benchmarks: taking a still draws an average of 2149 mW, while video uses 2449 mW, only 300 mW more than still. This result is perhaps surprising, but consistent with our previous results. When taking a still photograph, the image contents are displayed in real-time on the display, similarly to video. The additional work required for video is coding and storage, which we have already shown to be efficient.

Idle and peak power Idle power is the cost off having a device powered up, but not running any workload. Overall, idle power contributes substantially to total system power. On Freerunner, idle power accounts for at least 50 % of the total in the usage scenarios (excepting the phone call case, where cellular power dominates). Including the backlight power raises this figure significantly. On S3, idle power accounts for on average 52 % of total power in scenarios where the screen is on.

A consequence of this observation is that idle power management (shutting down unused components, disabling their power supplies, etc.) is extremely important to achieving energy efficiency. To achieve good end-to-end efficiency, it is critical that devices implement deep low-power states, and that the OS makes good use of these states, in order to keep idle power under control.

Using our data we perform the following thought experiment: for each component, find its maximum power consumption across all benchmarks, and sum these to find the system's peak power under some theoretical workload that maximally stresses all components. On S3, this is almost 9 W: a power draw that could drain a full battery in around 50 minutes, but realistically would produce an unmanageable amount of heat and quickly lead to thermal shutdown. Nonetheless, comparing this result to the average power consumption for realistic workloads (100's of mW range) demonstrates that typical usage is in fact a state of extreme under-utilisation. Thus, as well as deep low-power states needed to control idle power, it is also critical that components support efficient intermediate power states, where they are expected to spend the vast majority of their time.

Dedicated hardware Our S3 video results demonstrate the efficacy of using the dedicated hardware video decode unit, rather than a software decoder running on-processor, for energy-efficient video playback. For the high-quality video, a saving of more than 1 W (45 % reduction) can be achieved, which translates to 2.6 hours of additional playback time. For low-quality video, power is reduced by 30 % (490 mW, 2.2 hours of playback).

Small contributors We have discussed the components that contribute significantly to overall power drain on the system. However some components show consistently low energy consumption, in particular the RAM, sensors, audio, and storage subsystems.

While peak RAM power is high (up to 200 mW), high activity always correlates with high CPU or GPU power consumption. This is because RAM is a slave device: it responds to requests from bus masters and maintains state, but never initiates work. Therefore as a percentage, RAM contributes little to overall power consumption: only 6 % on Freerunner. The exception is suspend mode, where RAM must maintain significant state while all other devices enter their lowest-power mode. The absolute RAM power cost does not increase, but its relative contribution does. On both Freerunner and S3, this costs about 4 mW, which on S3 is 15–25 % of total power, depending on the network technology used.

Environmental sensors contribute little to overall power consumption, even at very high sampling rates. On the S3 device, we measured a maximum of 30 mW consumed in any sensor (the gyroscope) and an average of 10 mW. We measured the cost of running the software that collects the samples to be 10 mW: equal to the hardware cost! While 20 mW is low compared with the full system, this cost assumes that the smartphone is already in (at least) the idle state. If however sensor usage causes the device to wake from suspend mode, the cost is substantial: 583 mW on average, compared with approximately 35 mW of idle power. Thus, restricting non-essential sensor use to times when the device is already online has a significant positive impact on energy efficiency.

The audio subsystem is another largely insignificant energy consumer: 20–40 mW in our experiments, including both the amplifier and codec/DAC, and mostly independent of the output volume. Similarly to the sensor case however, audio playback requires the device to be powered up. Audio playback is therefore very cheap if the smartphone is already on, but, like all workloads, expensive if it requires bringing the device out of suspend. On S3 for example, the audio playback scenario consumes about 220 mW, an order of magnitude higher than suspend, but comparable with idle.

While our micro-benchmarks showed that the peak power of storage could be substantial (50 mW for the SD card on Freerunner, and 230 mW for flash on S3), in practice the utilisation is low enough such that average power is negligible. Even video playback, one of the more data-intensive uses of mobile devices, showed SD and flash power well under 1 % of total. In every scenario, flash memory on the S3 is a negligible contributor to total power. It peaks at 51 mW in the camera video recording benchmark (2 % of total), and for all others is less than 0.5 % of total.

GPS One might also consider GPS to be a sensor, but its power usage profile is significantly different to that of environmental sensors: high and somewhat variable, depending on the "fix" state. In our GPS benchmark, the receiver consumes around 40 % of total. We found the surprising result that on both devices, the tracking power *exceeded* the acquisition power, contrary to popular folklore (and indeed at least one device manual) that suggests power consumption drops when entering tracking mode.

Network-assisted GPS technologies (*A-GPS*) are designed to reduce the time spent in acquisition mode by downloading satellite orbitals from a fast secondary source, such as a cellular connection to a terrestrial server, rather than the slow GPS satellite downlink. Our data demonstrate that the relative magnitude of network and GPS power consumption (approximately 2:1 for 3G) is such that the use of A-GPS is clearly a win in total system energy consumption: not because it reduces instantaneous power, but due to the significant reduction in time to first fix (TTFF), hence reducing the time that the GPS module must be enabled.

4.6.2 Trends

Having collected and analysed the data from four devices of roughly four consecutive smartphone generations, we can now analyse the trends in power consumption over time.

We observe very similar suspend power for the G1, N1 and S3 ($\approx 25\,\mathrm{mW}$), suggesting that this may be an intrinsic lower bound. Given that battery size/capacity has been increasing (Figure 4.26), standby lifetime has increased by around 75 % with no change in functionality.

Idle power, however, appears to increase over time. We believe this to be mainly driven by increasing display size, not only in terms of geometric dimension but also resolution. Physically larger screens consume more energy due to increased light output, while increasing the pixel density causes an increase in power from clocking out more bits per screen update, wider and faster buses, increased cost of rasterization, etc. Across the G1, N1, and S3 devices, the screens increase in both physical dimension *and* pixel density. The Freerunner however has a small screen of particularly high resolution, atypical for devices at the time.[3] Figure 4.29 plots the total number of pixels in each device's screen versus the full-system idle power, a relationship that appears approximately linear. In scenarios where the display is not active, we see either a reduction or no significant change in total power consumption over time, further supporting the conclusion that the screen is the primary driver of energy consumption over time, and suggesting an overall increase in efficiency for the majority of smartphone functionality. However, the screen power increases far outstrip this efficiency improvement, leading to a net increase in energy consumption over time.

Audio playback is the prime example of a workload where the screen is typically disabled. In that scenario, we see that power consumption has dropped steadily across the generations, from 460 mW on the G1 to 226 mW on the S3. Since the display is disabled and audio levels unchanged, this shows a substantial increase in energy efficiency in the newer devices.

The power consumption for a voice call has not appreciably changed across the generations; the G1, N1 and S3 are within 15 % of each other, with the Freerunner a little higher. As we noted earlier, this fact, combined with the overall increase in power consumption, means that making a phone call has gone from the most power-hungry workload (almost 2 times more than the nearest on Freerunner) to one of lowest-power workloads performed on a modern device.

While average energy consumption is the important metric to determine energy efficiency, instantaneous peak power is also relevant due to power supply current constraints and thermal limitations. Our results indicate that maximum power has increased significantly over the generations. As a thought experiment, we determined the maximum "theoretical" power consumption of the S3 to be around 9 W, calculated by summing the peak power observed for each component across all our benchmarks. A similar calculation on the Freerunner yields around 2 W. This factor of 4.5 increase is certainly due to increased functionality of newer devices: they can simply *do more*, requiring more power. However, this also reflects an increase in capacity: CPUs can execute more instructions (and in particular, in parallel with multi-core), GPUs can render more complicated scenes at higher frame rates, network interfaces can push more data, and so on. Our analysis shows that most typical "every-day" workloads do not require this increased capacity. Consequently, the gap between average and peak power is increasing rapidly. This suggests that intelligent management of intermediate states, and thermal management at high power levels, will increasingly become an important aspect of smartphone energy management.

[3]The main interaction with the Freerunner screen is via stylus, partly as a design choice, but also due to its resistive touchscreen, which are generally unsuitable for finger touch input compared with the capacitive touchscreens of the three other devices. However, the stylus provides much finer input control, so a high-resolution display is used to enable finer GUI elements.

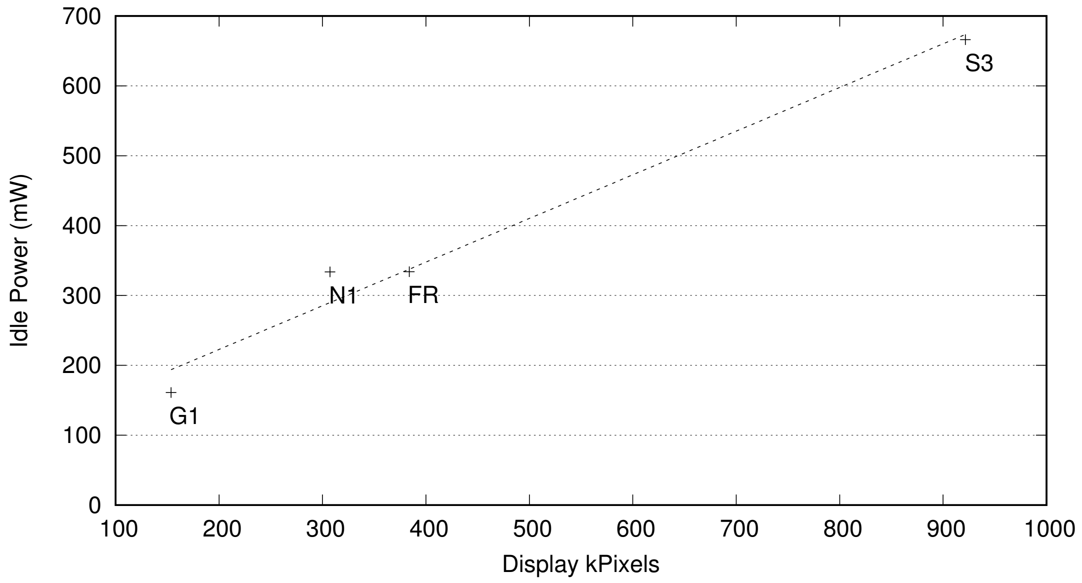


Figure 4.29: Device idle power vs. total number of display pixels.

This is particularly true for CPU in the “light weight” real world usage scenarios, like email, web browsing, and audio playback, where we don’t see any particular upward or downward trend in CPU energy consumption: average power is largely consistent across the devices for many workloads. However, *peak* power has increased significantly. The results in Section 4.5.2 show two phenomena: per-core peak power is increasing, as evidenced by the single-core Freerunner, G1, and N1 results, and total processor peak power is also increasing due to the advent of multi-core. This increase in peak power is a consequence of a significant increase in performance, which enables new, more computationally-demanding applications to become available in the smartphone space. As a result, running the applications processor efficiently at intermediate operating points will be increasingly important for optimising battery lifetime.

Chapter 5

Offline-aware Frequency Scaling

This section describes work published in a pair of peer-reviewed papers co-authored with my advisor. The first [CH13a] was published at the 2013 Workshop on Power-Aware Computing and Systems (HotPower), and in it we present our observation that onlining cores of a multi-core processor is cheaper than expected due to low static power. We also present the first implementation of our *medusa* policy which takes advantage of this fact. In the 2014 Real-Time and Embedded Technology and Applications Symposium (RTAS) we present a second paper [CH14] on this topic which broadens the analysis to include systems with significant per-core static power. We also generalise the medusa algorithm to account for these devices.

5.1 Introduction

In Chapter 4, we observed an interesting trend in smartphone development. Namely, that CPU peak compute capacity and power consumption are increasing substantially over time, yet the average usage has not followed anywhere near as fast. This has enabled computationally-demanding applications, hitherto restricted to more energy-abundant platforms, to become available in the mobile space: image processing, gaming, augmented and virtual reality, etc.

Since battery capacities have not increased substantially, the additional compute resources must be managed very carefully to provide reasonable battery lifetime on a single charge. Clearly the efficiency of the applications processor at intermediate utilisation levels is therefore a key factor to the efficiency of the devices as a whole. In this chapter, we tackle this particular problem.

The capacity increase of applications processors has occurred in three main dimensions: the number of cores per CPU (also called TLP, for *thread-level parallelism*), clock speed, and instruction-level parallelism (ILP), i.e. the number of instructions executed per clock cycle. ILP is largely transparent to software, particularly on architectures common in the smartphone space (ARM, for instance). Except in extreme cases, even low-level system software operates in total ignorance of any ILP implemented by the hardware. That leaves TLP and clock speed as knobs which can be set to control the processor's performance-power trade-off.

The modern mobile multi-core processor provides control of cores and clock frequency with two main mechanisms. The first of these is *offlining*, which allows the operating system to switch off individual cores, reducing the per-core power consumption to zero, but allowing the remaining cores to continue processing. The other is DVFS, or *dynamic voltage and frequency scaling*, which provides for the reduction of CPU operating frequency and voltage at the cost of performance.

These mechanisms can be used simultaneously, and both control performance/power. Using them in tandem therefore presents an interesting trade-off: to respond to increasing system load, either the frequency of online cores can be increased or additional cores can be onlined (and vice versa for decreasing load). The energy-optimal decision depends, among other things, on the particular power/performance response of these two mechanisms.

Past research has observed the increasing importance of static power, and the resulting importance of sleep states in reducing energy consumption [BJ08, LSH11]. When the per-core static power is significant, more active cores means more static power, and thus a higher energy consumption. In particular, deep sleep states mean that the *race-to-halt* approach may be beneficial, as it minimises the accumulation of static energy loss. Consequently, running at minimum frequency may not minimise energy use [MLH$^+$02]. However, since power is super-linear in frequency, simply running at the maximum frequency to minimise the number of cores required is also suboptimal.

In this chapter we revisit these assumptions and observations in the context of modern smartphone applications processors. We investigate the combined efficacy of core idle power states and DVFS for maximising energy efficiency in situations where the CPU is underutilised—a technique more generally known as *slack management*. More precisely, this means minimising energy consumption under the constraint of low impact to application performance. Our goal is to determine a methodology for selecting the energy-optimal setting for CPU frequency and the number of online cores—this pair we call the *operating point*, or OP.

We find a surprising diversity in characteristics of latest-generation ARM processor implementations: from very significant to completely negligible static power. This has a significant impact on energy management, which does not seem to be well understood: phones ship with energy-management policies which produce highly non-optimal results.

We start our investigation by measuring the energy consumption of a range of synthetic workloads while varying the core frequency and number of active cores. Then we analyse these data from both an empirical and theoretical perspective, and show that running with more online cores generally reduces energy consumption, so aggressive offlining of processors cores is an ineffective energy management strategy. We design and implement an energy-management policy in the Linux kernel called *medusa*, which exploits these insights to reduce energy consumption, and we perform a series of benchmarks to evaluate medusa's effectiveness, comparing it against existing policies, and exploring how to adapt its algorithms for particular platforms.

5.2 Motivation

We begin the investigation with a series of simple micro-benchmarks that exercise the CPU in predicable ways. We run these on two different platforms, and measure the system's power consumption while varying the processor's operating point (frequency and core count).

5.2.1 Platforms

We run our benchmarks on two versions of the Samsung Galaxy S4 smartphone. The first, which we denote *S4-E*, is the GT-I9500 model which features the Samsung Exynos 5410 system-on-chip (SoC) with a quad-core Cortex-A15 and a quad-core Cortex-A7 CPU. These run in an ARM big.LITTLE "task migration" configuration [Gre11] whereby one of the quad-core CPUs (called *clusters*) can be active at any time. Switching between the clusters is controlled by the operating system, and on our device this is done in the DVFS subsystem by "virtualising" the frequency control. Like with a traditional OS energy-management policy, the governor periodically selects the clock frequency it thinks best meets the goals of the policy. Normally this would map 1:1 to the actual CPU operating frequency, but when virtualised, it is used as a proxy for desired performance. The selected frequency is used to select a cluster to activate, and then mapped to a physical frequency that the cluster will run at. The particular algorithm implemented on the platform is this: virtual frequencies in the range 250–600 MHz will cause the Cortex-A7 ("little") cluster to be active, running at twice the virtual frequency. For virtual frequencies at or above 800 MHz, the Cortex-A15 ("big") cluster will be active at a physical frequency equal to the virtual frequency. This is the default behaviour that ships with the GT-I9500 device, and for the purposes of our work we retain this behaviour and, unless otherwise noted, we use virtual frequencies in the remainder of the chapter.

Our second platform is the GT-I9505 model featuring a Qualcomm Snapdragon 600 SoC with a quad-core Krait 300 CPU; we call this the *S4-S*. Table 5.1 summarises the relevant parameters for these two platforms. On *S4-E*, all cores run at the same voltage and frequency. The *S4-S* allows cores to run at independent voltages and frequencies, but for the purposes of this work we force all active cores to run at the same setting. Herbert and Marculescu [HM07] show that this is not a significant limitation.

These devices are recent-generation, off-the-shelf commodity smartphones. They represent two SoC vendors (Samsung and Qualcomm) and two independent implementations of the

Characteristic	*S4-E*	*S4-S*
Model	GT-I9500	GT-I9505
SoC	Exynos 5410	Snapdragon 600 (8064T)
CPU	Cortex-A15/A7	Krait 300
cores	4†	4
Frequency (MHz)		
min	250	384
max	1600	1890
Voltage (V)		
min	0.9	0.9
max	1.2125 (A15 "big") 1.15 (A7 "little")	1.2125
Cache (KiB)		
L0 (I/D)	—	4 / 4
L1 (I/D)	32 / 32	16 / 16
L2 (shared)	1024	2048
OS	Android 4.2.2	

Table 5.1: Characteristics of the two Galaxy S4 platforms. †The *S4-E* has 8 CPU cores, but only 4 are active at any time.

ARMv7 architecture (ARM Cortex and Qualcomm Krait), and hence give us good coverage of the high-end mobile CPU space.

5.2.2 Measurement

We measure power consumption of the devices by instrumenting them in much the same way as the devices in Chapter 4. Specifically, we insert a $20\,\mathrm{m}\Omega$ current sense resistor between the battery and the device, and use the same National Instruments NI-6229 data acquisition system to sample the voltage drop across the sense resistor and the battery supply voltage, from which power can be determined. Since we are focused specifically on CPU energy in the present investigation, we configure the devices differently to our previous measurements to minimise the interference from non-CPU components: we run the devices in airplane mode with all unrelated components disabled, including the screen. When the screen is required, as in the medusa evaluation benchmarks, we run it at minimum brightness.

5.2.3 Benchmarks

As discussed earlier, this work targets applications where the CPU is under-utilised. We use a simple periodic task model to represent these workloads, in which the task alternates between fully idle and compute-intensive. The effective CPU utilisation can be controlled by setting the amount of work to perform in the compute phase of each period. We use three variants of this workload, which differ in the level of memory intensity during the compute phase, denoted *loadcpu*, *loadcache* and *loadmem*, which primarily exercise the CPU, L2 cache, and main memory, respectively. We control the number of loops in the compute phase to select a desired level of load, defined as the percentage of total system capacity used. For example, 100 % means running all cores at maximum frequency, 50 % means either all cores at half frequency, or two cores at maximum frequency, and so on. We set up each benchmark at four load levels, 10, 25, 50, and 75 %. 100 % is of course omitted, since there is, by definition, only one operating point which can sustain that load level. We run the benchmarks for a fixed period of time, and measure the power consumption while varying the processor's operating point. However, we select only the operating points which can sustain the desired load without over-running the period. That is, we keep the CPU utilisation less than (or equal to) 100 %. This means that for a particular load level, both the run-time and total work done is constant—what varies is the duty-cycle, i.e. the percentage of each period spent in the compute phase.

In *loadcpu*, the workload is a simple busy loop with no memory accesses. For *loadcache*, each process strides through a memory region performing read-modify-write cycles on successive cache lines. The size of the region is 2× the L1 cache size per process, for a total working set of 128 KiB on *S4-S* and 256 KiB on *S4-E*. This results in significant cache pressure, with few accesses to main memory. For *loadmem* we do the same, but increasing the total working set size to 2× L2 cache size, resulting in many accesses to main memory.

To compare with fully CPU-bound workloads, we also run a micro-benchmark *spin*, which simply executes a busy-loop for a fixed number of iterations, with no periods of idleness. We measure the power consumption of this workload at all operating points, which changes the execution time but not the total work performed.

For all micro-benchmarks, the work is shared over four processes, each doing one quarter of the total, so that it can be distributed across cores. These processes are are scheduled by the default Linux task scheduler. While this does not enforce any particular assignment of processes to cores, we observe that in practice, the work performed by each core involved is approximately equal. In all cases we report the average of three iterations of the workload, and for all data reported, the relative standard deviation across iterations is less than 6 %. We report the energy consumption for the benchmark, normalised to the highest for each load level.

5.2.4 Results

The results of *loadcpu* on each platform are shown in Figures 5.1 and 5.2. Note that as the load level increases, the number of points plotted decreases. As described above in Section 5.2.3, we only measure the *feasible* operating points, which are those that can execute the workload without over-running the fixed execution time. On *S4-E*, the discontinuity from 600 to 800 MHz is due to the switch from the little (A7) to big (A15) CPU cluster.

On both *S4-S* and *S4-E*, energy is an increasing function of frequency when the number of online cores is fixed. That is, the lowest energy consumption occurs at the lowest frequency

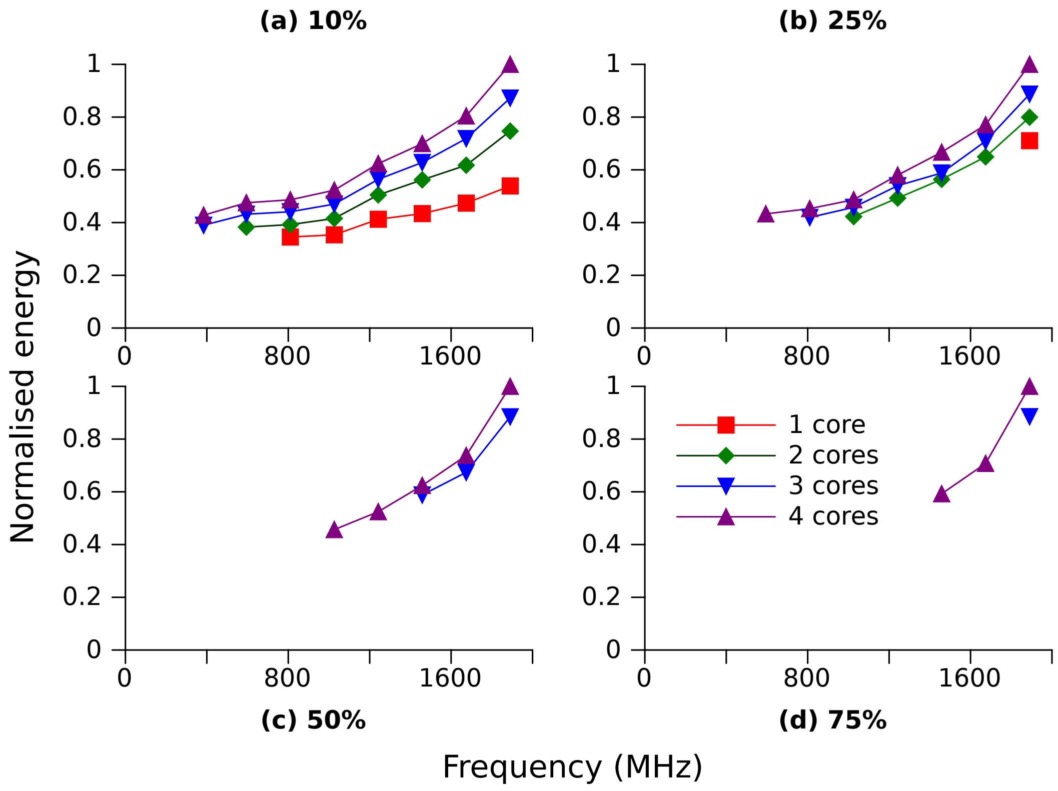


Figure 5.1: *S4-S loadcpu* normalised energy vs. frequency at 10, 25, 50, and 75 % load.

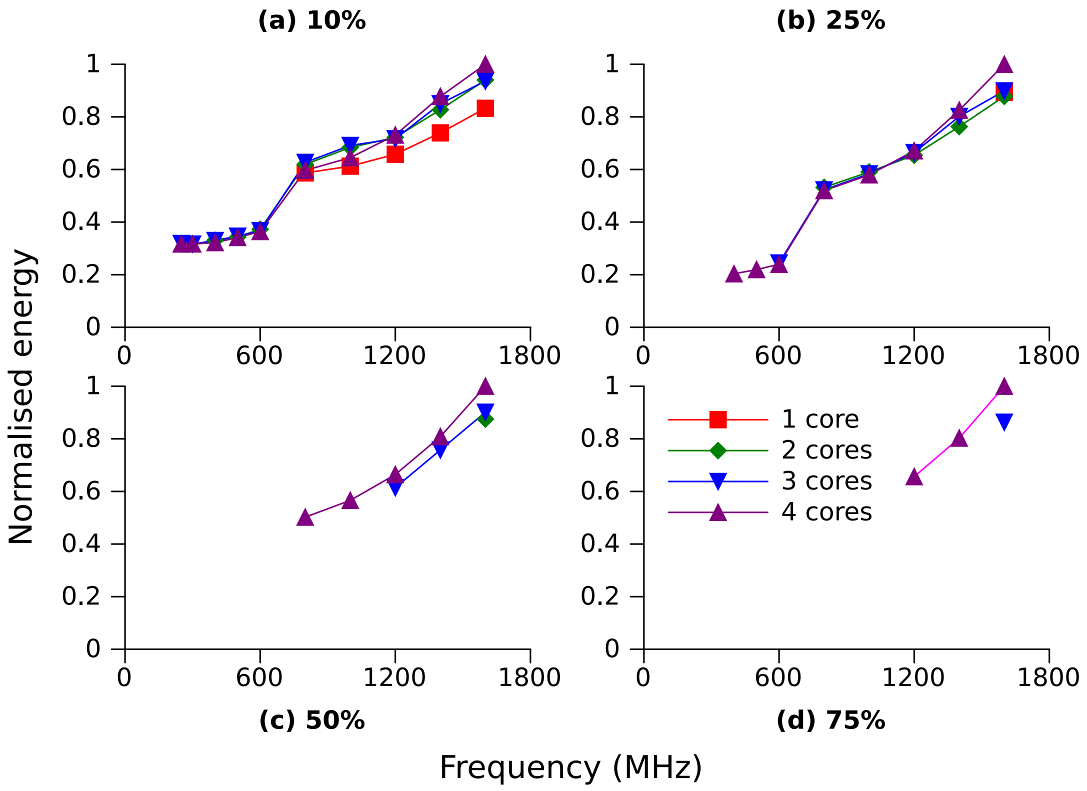


Figure 5.2: *S4-E loadcpu* normalised energy vs. frequency at 10, 25, 50, and 75 % load.

attainable for a given number of cores.

If we fix the frequency, we see different characteristics between the two devices. On *S4-E*, the correlation between online cores and energy consumption is weak: in some cases it is more efficient to run with fewer cores online, but in most cases there is no correlation. However, on the *S4-S*, energy varies significantly with the number of online cores at fixed frequency. At 10 % load, for example, the energy consumption at maximum frequency is more than doubled when running four cores compared with one. As load level increases, the number of cores becomes less significant to overall energy consumption.

On both platforms and considering a fixed frequency, using a single core (where load is low enough to allow this) is more efficient than 2, 3 or 4 cores, and this disparity increases with frequency (although this effect is much more pronounced on the *S4-S* than the *S4-E*). This is due to a CPU-wide sleep state that significantly reduces power, but can be entered only when cores 1–3 are offline, and core 0 becomes idle. We will return to this point in Section 5.3.3.

Across all operating points on both platforms, the minimum energy point occurs with more than the minimal number of online cores in cases where this allows a significant reduction in frequency.

When running with added L2 or main memory pressure, the same observations continue to hold. Figures 5.3–5.5 show the results for *loadcache* and *loadmem*. In particular, adding cache pressure on the *S4-S* platform has not noticeably changed the shape of the curves, but when adding main-memory accesses, we see a pronounced increase in the absolute energy difference while varying the number of cores. The other major difference compared with the CPU-bound case is that, under cache pressure, the discontinuity on *S4-E* from 600–800 MHz caused by the CPU cluster switch has largely disappeared. In other respects, the *loadcache* and *loadmem* results do not differ from *loadcpu* in any significant aspect.

The results of the *spin* benchmark for both platforms are shown in Figure 5.6. While energy is a complex function of frequency, maximising the number of online cores is always optimal for this workload, both globally and at each frequency. The discontinuity in (a) is again due to the CPU cluster switch from 600 to 800 MHz. While the power curve is reasonably smooth across the switch (as shown in Figures 5.2 and 5.4), the performance curve shows a significant discontinuity, resulting in this effect on the energy curve. The use of little cores can significantly improve energy efficiency, more than a factor of two if the correct OPs are selected. However this of course results in an increased workload run-time.

Compared with the previous benchmarks, this nicely demonstrates our earlier claim: energy management for periodic workloads is significantly different than for compute-bound workloads. In the CPU-bound case, selecting the number of cores is simple: online as many as the thread-level parallelism allows. Selecting a frequency, however, is complicated due to the significance of static power. Running faster increases the dynamic energy, but reduces

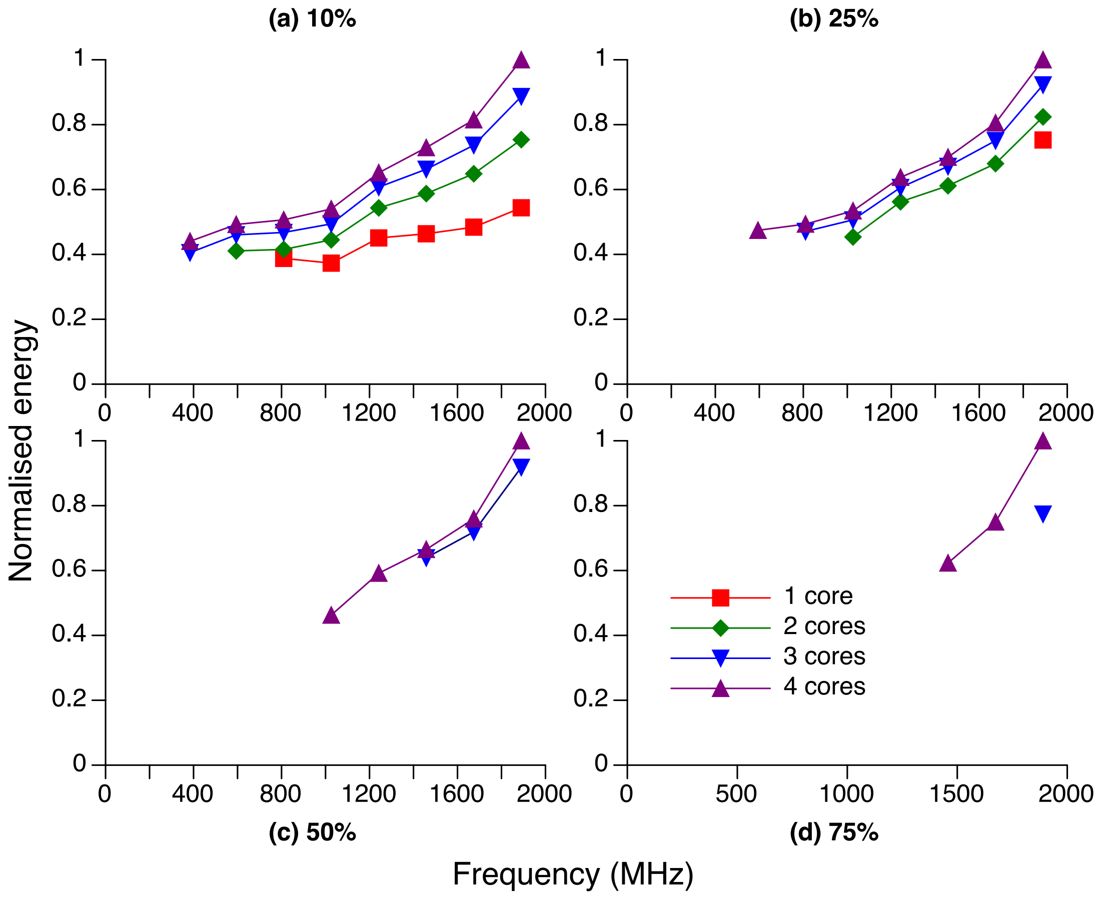


Figure 5.3: *S4-S loadcache* normalised energy vs. frequency at 10, 25, 50, and 75 % load.

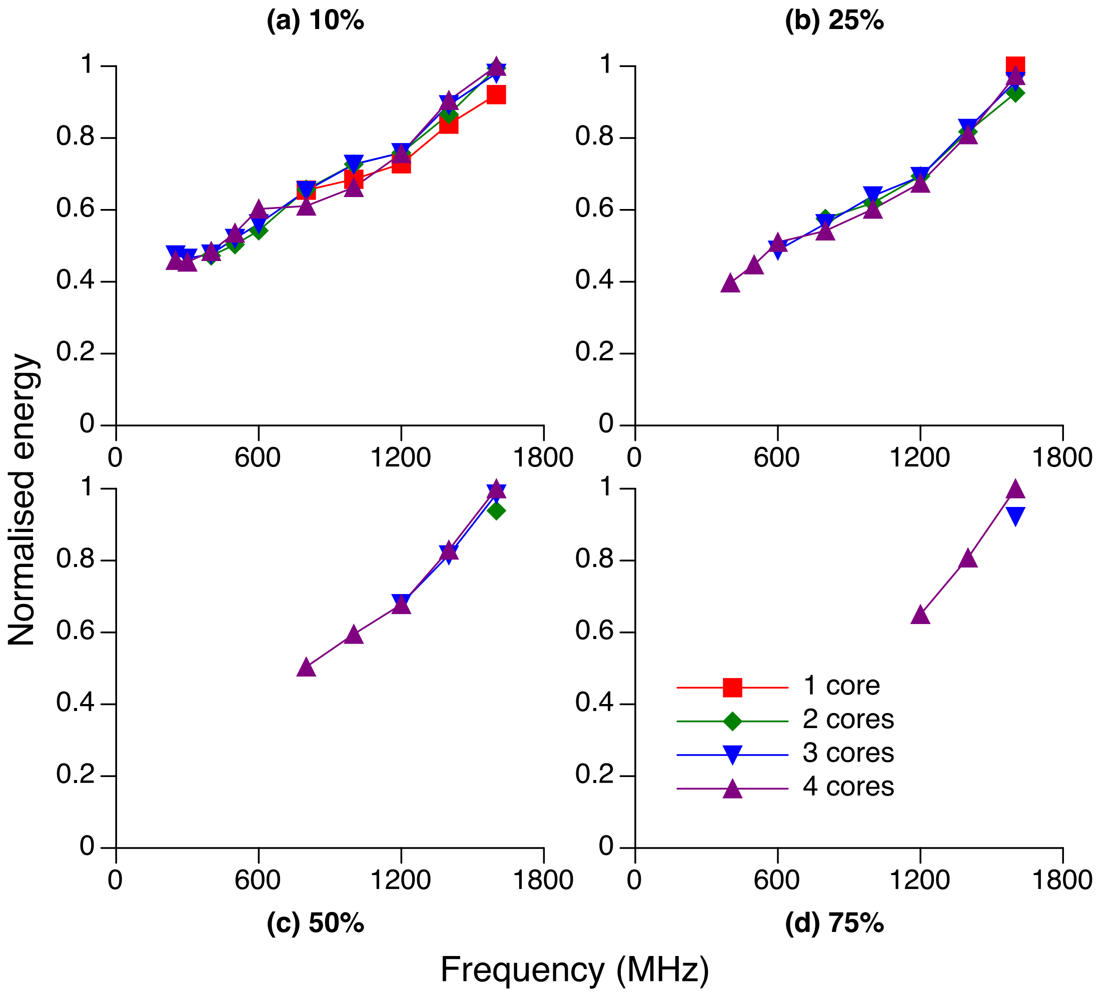


Figure 5.4: *S4-E loadcache* normalised energy vs. frequency at 10, 25, 50, and 75 % load.

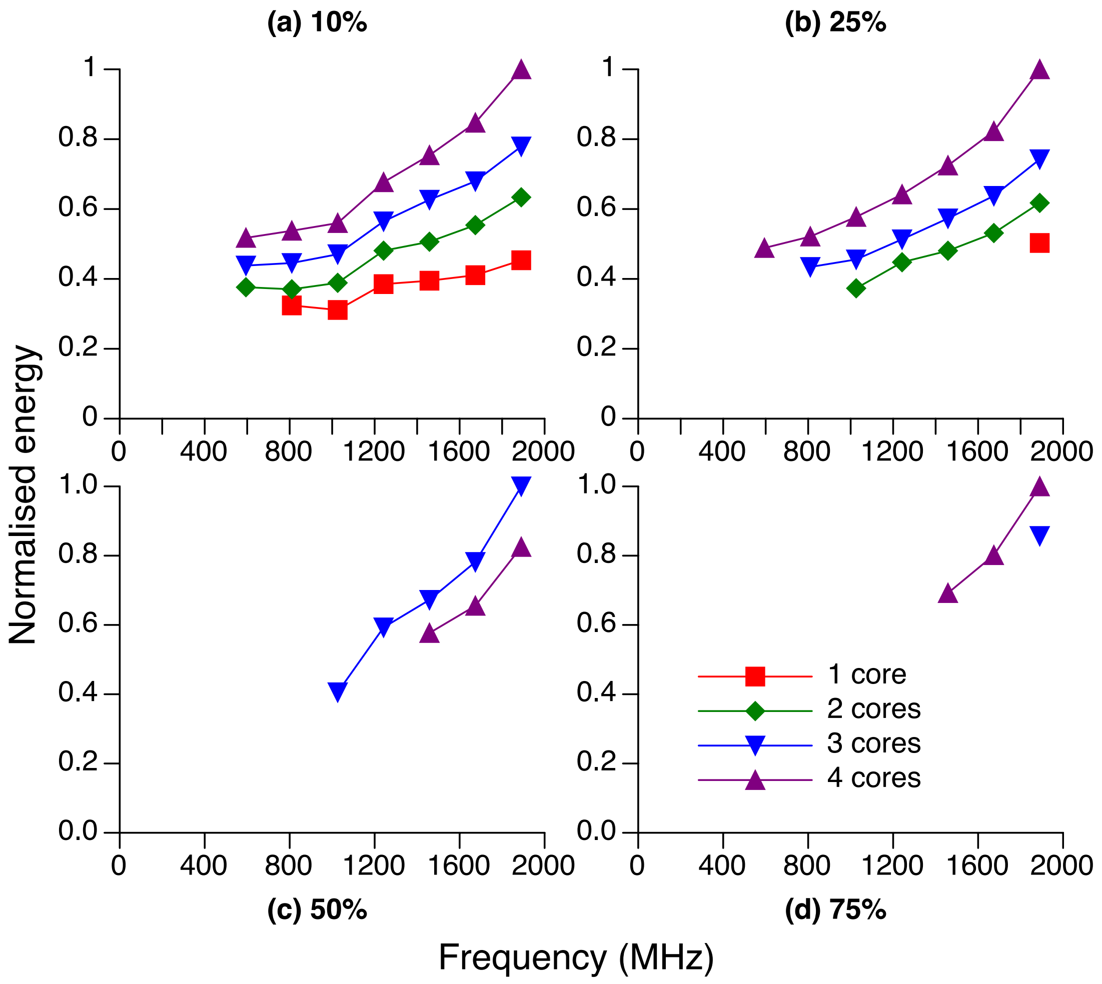


Figure 5.5: *S4-S loadmem* normalised energy vs. frequency at 10, 25, 50, and 75 % load.

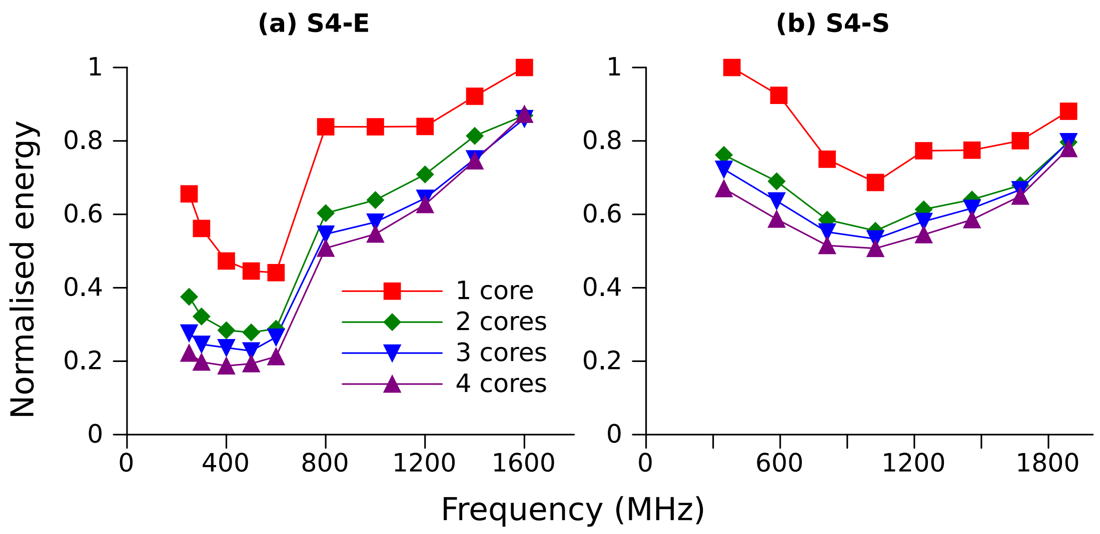


Figure 5.6: *spin* normalised energy vs. frequency (MHz) for (a) *S4-E*, and (b) *S4-S*.

the static energy. Reducing frequency decreases dynamic energy but increases static energy. Performance requirements must also be accounted for. This particular DVFS problem has been well-studied, exemplified by Snowdon et al. [SLSPH09], so we return our attention to the less-well-understood problem of periodic workloads.

5.3 Modeling and Analysis

In this section we analyse the results from Section 5.2, and provide a theoretical framework in which to understand them.

5.3.1 CPU Model

We will use a simple multi-core CPU power model in which each core can be in one of three states, *active*, *idle* and *offline*. Active is the state in which instructions are executing. The offline state is the deepest sleep state in which the core and all supporting circuity is fully powered down. Idle is the shallowest sleep state in which the core is ready to perform computation but not actually executing instructions. These states correspond roughly to ACPI C-states C0, C1–C(x-1), and Cx respectively (if x is the deepest C-state). Each core may be in a different power state, but we assume that the CPU has a single voltage and frequency plane, and hence all cores are clocked at the same frequency and supplied with the same voltage. As mentioned previously, Herbert and Marculescu [HM07] argue that the ability to independently control frequency and voltage for each core in a chip multi-processor yields only marginal improvements in energy consumption. Further, we assume that the idle state can be entered and exited instantaneously with no performance or energy overhead, and that this is done immediately and automatically when no computation is available to run. On the other hand, we assume a deep offline state, which has a high entry and exit cost and thus is used at a coarse granularity under control of the OS-level energy management policy.

We model the power consumption of a CPU with n online cores at frequency f as:

$$P_{\text{CPU}} = P_{\text{uncore}} + n(P_{\text{dynamic}} + P_{\text{static}}) \ . \tag{5.1}$$

P_{static} is the power drawn by a core when it is online but otherwise idle. It is workload independent, but varies with core voltage. P_{dynamic} is the additional power when a core is active, and is dependent on workload and voltage/frequency. The sum of the dynamic and static components forms the per-core contribution to total power. The remaining CPU power is independent of the number of online cores, and hence called the *uncore*, denoted P_{uncore}. The uncore must remain powered as long as any core is online, and typically includes the shared last-level caches, buses, the memory controller, etc. We can express P_{dynamic} by the well-known equation

$$P_{\text{dynamic}} = C_{\text{eff}} f V^2 \ , \tag{5.2}$$

where C_{eff} is the effective switching capacitance, f is the core frequency, and V is the core voltage.

This model is based on Gupta et al. [GBK$^+$12], but others have used similar models [XZR$^+$05, ZMM04]. Importantly, we are using a *functional* power model, and not a physical one: we are interested in how power appears to be drawn from the OS perspective, rather than the location or subsystem to which the corresponding circuit belongs. For example, if the transistor leakage power of a core persists whether or not that core is online, such as if all cores share a power plane, then functionally we consider this uncore power, since it is independent of n. We discuss the validity of this model in Section 5.3.3.

5.3.2 Analysis

The energy cost of choosing an incorrect operating point can be substantial, a result well known from the single-core DVFS literature [SLSPH09]. Our results show that the multi-core processor only exacerbates this problem, both by increasing the penalty of incorrect OP selection, and by increasing the size of the optimisation problem with the additional "number of online cores" dimension (denoted n). Moreover, the results show that the offline and DVFS mechanisms are inherently tied: one cannot be optimised independently of the other. Applying the naive wisdom that lower power implies lower energy, i.e. that fewer cores result in lower energy consumption, can lead to catastrophic results, in some cases (e.g. Figure 5.2(b)) a quadrupling of energy consumption!

As the experiments of Section 5.2.4 show, energy is generally minimised by running at the lowest possible frequency on both platforms. However, to meet performance requirements, cores must be onlined to offset the capacity lost by reducing frequency. Put another way, onlining cores allows us to run workloads at lower, more energy-efficient frequencies for equivalent throughput. This is the primary mechanism by which running more cores can reduce energy consumption.

We can develop an understanding of how this occurs using the above CPU model. Treating the workload as periodic, using Equation 5.1 and substituting T for the period of the workload, and t for the per-period execution time (where $t \leq T$), we get a per-period energy of

$$E_{\text{CPU}} = P_{\text{uncore}}T + n(P_{\text{dynamic}}t + P_{\text{static}}T) \ . \tag{5.3}$$

Using the approximation [SKK11] that

$$P_{\text{dynamic}} \propto \frac{\text{instructions}}{\text{cycles}} \tag{5.4}$$

for fixed frequency f, it follows that E_{dynamic} is proportional to the number of executed instructions, which is constant for a fixed workload. So if we execute i instructions spread across n cores, then

$$E_{\text{dynamic}} \propto i/n \ , \tag{5.5}$$

and substituting into Equation 5.3, we get

$$E_{\text{CPU}} = (P_{\text{uncore}} + nP_{\text{static}})T + ie \ , \tag{5.6}$$

where e is the proportionality constant of Equation 5.4, corresponding to the per-instruction energy, and ie is constant for a given workload. Thus, if P_{static} is small compared to P_{uncore}, then E_{CPU} is independent of n. In the following sections, we show this to be true on *S4-E*, and then deal with the case where it is not. Note that for the purposes of this analysis we can treat P_{uncore} as constant, since adding the dynamic uncore contribution would only increase the uncore/static ratio.

For workloads with no periods of idleness (i.e. compute-bound, such as our *spin*), it is always more energy efficient to run with more cores online, because increasing throughput reduces execution time and thus reduces the accumulation of static CPU energy. This is an example of the race-to-idle effect, which is well-documented in the DVFS literature [MLH$^+$02]. However, with DVFS, dynamic power is super-linear in frequency (since $P \propto fV^2$ and V is monotonic in f) and hence race-to-idle with frequency is not necessarily optimal. On the other hand, core power is linear in the number of online cores.

A workload is considered perfectly scalable in the thread-level parallelism sense if the performance increase seen by increasing the number of CPU cores is proportional to the compute capacity added, or more precisely, if $t \propto 1/n$, where t is the execution time and n is the

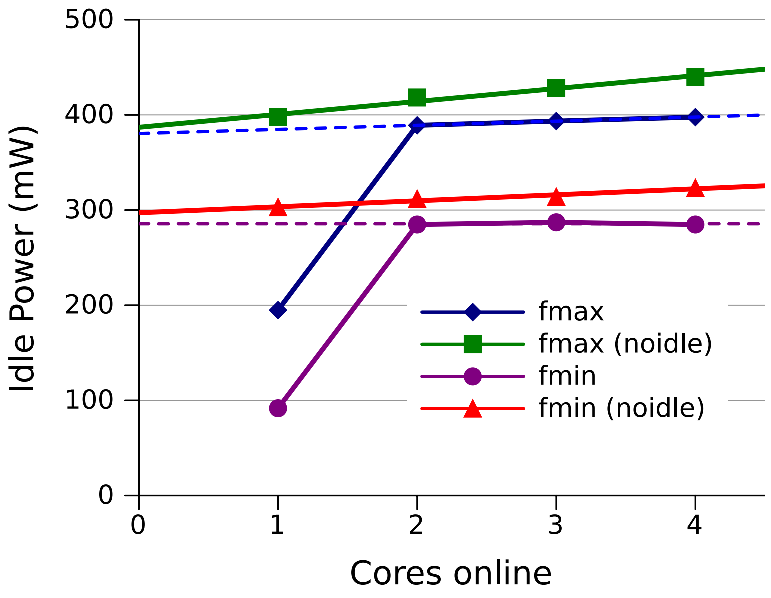


Figure 5.7: *S4-E* idle power for the Cortex-A15 (big) CPU cluster at minimum (800 MHz) and maximum (1600 MHz) frequencies, and with idle states enabled and disabled (noidle).

number of cores used in the computation. Scalability can be broken down into two aspects: algorithmic scalability, that the problem can be divided into components that can be solved in parallel; and hardware scalability, that the processor is able to efficiently execute multiple threads that may be competing for shared resources like L2 cache and the memory bus. If we make the assumption that a workload is scalable, then from Equation 5.1 we get

$$E_{\text{CPU}} \propto \frac{P_{\text{uncore}}}{n} + P_{\text{dynamic}} + P_{\text{static}} \ , \tag{5.7}$$

and hence increasing n will always decrease E_{CPU} (for workloads with ideal scalability). The results of Figure 5.6 (which is scalable by design) show this clearly. Determining the optimal frequency however, depends on the relative values of the power terms and the quality-of-service expectations for the workload. As discussed earlier, this particular problem is outside the scope of the current work.

5.3.3 Model validation

In the previous section, we established that if P_{static} is low, then CPU energy is minimised by onlining as many cores as required to run at the lowest possible frequency. To validate this assumption, we directly measure the static power on both platforms.

Figure 5.7 shows idle power consumption as a function of the number of online cores for the *S4-E* platform running with the big (A15) cores only, at maximum (1600 MHz) and

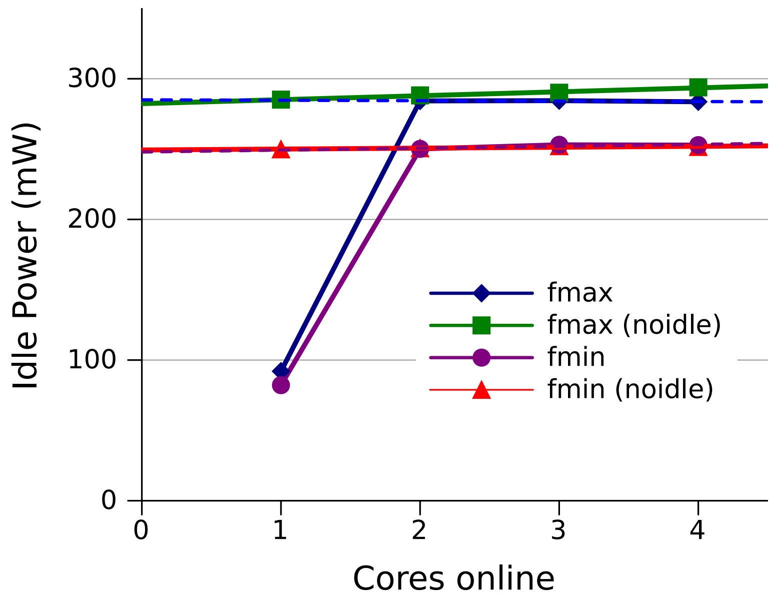


Figure 5.8: *S4-E* idle power for the Cortex-A7 (little) CPU cluster at minimum (500 MHz physical) and maximum (1200 MHz physical) frequencies, and with idle states enabled and disabled (noidle).

minimum (800 MHz) frequencies for that cluster. Shown are two data sets for each; one in the normal configuration, and one with all idle states disabled (noidle). From these graphs we can determine per-core static and uncore power as follows. From Equation 5.1, setting $P_{\text{dynamic}} = 0$ (since we are in the idle state) we get

$$P_{\text{idle}} = P_{\text{uncore}} + nP_{\text{static}} .$$

This is the equation of a line with gradient P_{static} and y-intercept P_{uncore}, which can be read directly off the graphs.

With idle states disabled, we see that P_{static} is approximately 14 mW per core at f_{max}, and 6 mW at f_{min}, determined by dividing the difference in power consumption from 2–4 cores by 2. This is a very small fraction of uncore power, only 3.5 % and 2.1 % respectively, so this platform indeed has low static power. From our previous analysis we thus expect to see that on *S4-E*, onlining cores consumes *less* energy, and Figures 5.2, 5.4 and 5.6 demonstrate exactly that. Furthermore, the small difference in power consumption between the normal and idle-disabled cases (worst-case of 42 mW for 4 cores at f_{max}) shows that our earlier assumption of a 3-state core (offline, idle, active) holds reasonably well on this platform. Moreover, this validates our assumption that idle state entry and exit cost is negligible; if the cost were high, we could simply disable idle states which would remove this cost at negligible power penalty.

We measured the same data for the small (A7) CPU cluster on the *S4-E*, running at max-

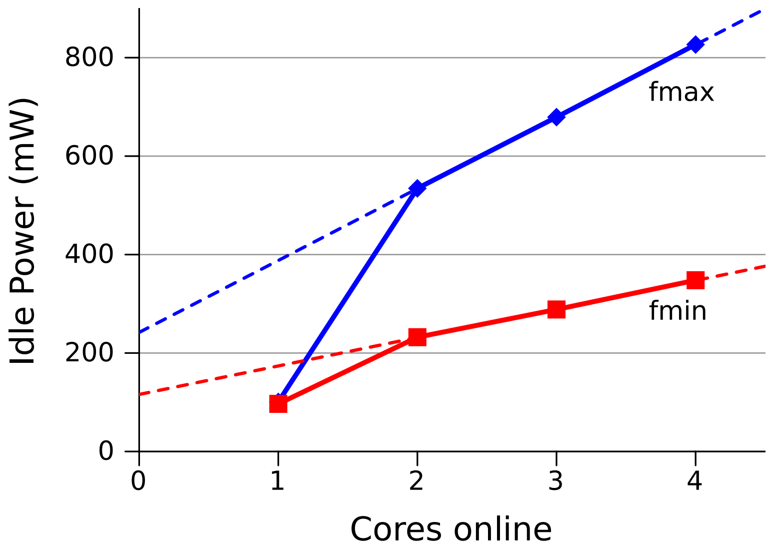


Figure 5.9: *S4-S* idle power at minimum (386 MHz) and maximum (1890 MHz) core frequencies.

imum (1200 MHz) and minimum (500 MHz) physical frequency, plotted in Figure 5.8. This shows negligible P_{static} even with idle states disabled (3 mW per core @ $f_{\max}$), and a worst-case difference between idle and noidle of 10 mW.

Figure 5.9 shows idle power as a function of online cores for the *S4-S* platform, at maximum (1890 MHz) and minimum (386 MHz) frequency. The per-core static power is 146 mW per core at $f_{\max}$, and 58 mW at $f_{\min}$, corresponding to 60 % and 50 % of P_{uncore}, respectively. This is certainly significant, and hence we do not necessarily expect to see that maximising the number of online cores yields minimum energy, and our micro-benchmark results reflect this, particularly Figures 5.1, 5.3 and 5.5 at 10 and 25 % load levels. We deal with this issue in Section 5.3.4. This figure shows the measurements with all core idle states enabled, however disabling them increases the idle power by at most 2 %. This further validates our 3-state CPU model.

As the idle states provide little power saving, race-to-idle is not beneficial. For periodic workloads, this implies that minimising power is equivalent to minimising energy. Moreover, since all power terms are *increasing* functions of f, then for fixed n, energy is minimised by minimising f. The same does not apply however to CPU-bound loads, because the entire CPU can be powered down and hence race-to-halt remains potentially effective.

It is certainly surprising to see such a big difference in static power with two contemporary implementations of the same architecture, by manufacturers who are both major players in the smartphone processor market! One explanation of this is that in the case of *S4-E*, all cores share a power plane (i.e. they are connected to the same power supply), so they are powered regardless of the power state of the core. This means that the leakage power, a significant

	Time (ms) $\pm 1\,\%$		Power (mW) $\pm 4\,\%$		Energy (mJ)	
	S4-E	*S4-S*	*S4-E*	*S4-S*	*S4-E*	*S4-S*
offline at f_{min}	22.6	11.7	357	454	8.1	5.3
offline at f_{max}	16.5	4.1	1305	1926	21.5	7.9
f_{min} to f_{max}	2.1	11.1	1033	420	2.2	4.7
f_{min} to $f_{\mathrm{min}+1}$	0.4	6.1	320	317	0.1	1.9
f_{max} to $f_{\mathrm{max}-1}$	1.2	0.2	1400	838	1.7	0.2

Table 5.2: Average cost of operating point transition pair.

aspect of overall power consumption, can not be reduced by offlining a core. On the other hand, the *S4-S* powers each core from independent supplies and hence leakage can be reduced in offline cores. The latter is consistent with per-core DVFS, which is supported on the *S4-S* platform, where cores require different supply voltages when running at different frequencies.

The significant reduction in power seen on both platforms with a single idle core is due to the previously mentioned "package idle" state, which requires core 0 to be idle, and cores 1–3 to be offline. The existence of low-power idle states such as this means that the energy-optimal OP may in fact not be at minimum frequency, due to race-to-idle reducing static energy loss. Specifically, there is a trade-off between the energy saved by completing the workload faster and entering the low-power state as soon as possible, and the additional dynamic energy consumption caused by running at a higher frequency. We claim that in general, solving such an optimisation requires a multi-core-aware online dynamic power model, which we believe to be an open problem. For the remainder of this work we ignore the effect of package idle states, but note that on our platforms, this appears not to be a significant drawback: the data in Section 5.2 shows that running above minimum frequency does not result in appreciable energy savings.

We asserted earlier that the cost of transitioning between the online and offline states is significant, and thus the mechanism can only be used in a coarse-grained fashion. To validate this, we perform an experiment where we on- and off-line a core 1000 times, measuring the latency and power consumption. Table 5.2 shows the results. Also shown is the cost to transition between core frequencies: from minimum to maximum, between the two lowest frequencies, and between the two highest frequencies. In each case we repeat the experiment 3 times and show the average cost of a transition pair (i.e. online and offline or frequency increase and decrease); our methodology does not allow us to determine the individual cost of each operation. For all power data, the relative standard deviation is $< 4\,\%$, while for time it is $< 1\,\%$.

5.3.4 Adapting for high static power

We have shown both theoretically and empirically how to optimise the OP for a processor with low per-core static power. If static power is significant however, our previous argument does not apply, because we now need to balance the static power cost of onlining cores with the dynamic cost of changing frequency. We now develop a heuristic that can be used to predict an efficient OP for such processors.

Earlier we argued that for periodic workloads, minimising P_{CPU} is equivalent to minimising E_{CPU}. If we treat n and f as continuous variables, then the optimal operating point can be determined by minimising $P_{\text{CPU}}(f, n)$. If we assume scalability (i.e. $n \propto 1/f$), then we can express n in terms of f as

$$n = \frac{n_{\text{max}} f_{\text{max}} u}{f} \equiv \frac{\gamma u}{f} \,, \tag{5.8}$$

where u is the total system utilisation and γ is a constant ($\equiv n_{\text{max}} f_{\text{max}}$). We can then minimise P_{CPU} by finding the f that solves

$$\frac{d}{df} P_{\text{CPU}}(f) = 0 \,. \tag{5.9}$$

Substituting Equations 5.2 and 5.8 into Equation 5.1, we get

$$P_{\text{CPU}} = P_{\text{uncore}} + \gamma u \left(C_{\text{eff}} V^2 + \frac{P_{\text{static}}}{f} \right) \,, \tag{5.10}$$

where P_{uncore}, P_{static} and V are functions of f, γ is a constant, and C_{eff} and u are workload-dependent. Differentiating with respect to f:

$$P'_{\text{CPU}} = P'_{\text{uncore}} + \gamma u \left(2 C_{\text{eff}} V V' + \frac{P'_{\text{static}} f - P_{\text{static}}}{f^2} \right) \,. \tag{5.11}$$

P_{uncore}, P_{static} and V can be measured, and their derivatives determined numerically, so the f that minimises P_{CPU} is a function of the remaining variables, u and C_{eff}, both of which are workload-dependent. Utilisation u can be easily measured online by sampling the task scheduling queues. Instantaneous utilisation $\hat{u}$ for n online cores can be determined by

$$\hat{u} = \frac{1}{n} \sum_{x=1}^{n} \mathrm{R}(x) \,,$$

where

$$\mathrm{R}(x) = \begin{cases} 1 & \text{if core } x \text{ has 1 or more runnable threads} \\ 0 & \text{otherwise} \end{cases}$$

and then average utilisation u can determined by a running average of $\hat{u}$ sampled at some suitable frequency.

The last unknown C_{eff} can be approximated as a function of instructions per cycle (IPC), such as is done by Gupta et al. [GBK$^+$12]. To do this on our platforms, we follow a methodology similar to that of Spiliopoulos et al. [SKK11]. Specifically, we run a series of compute-bound benchmarks, each at minimum and maximum frequency, while concurrently measuring

CPU power with our power meter, and IPC using the processor's performance monitoring unit. From above we have already determined P_{static} and P_{uncore}, so by rearranging Equation 5.1 (with $n = 1$) we get

$$P_{\text{dynamic}} = P_{\text{cpu}} - P_{\text{static}} - P_{\text{uncore}} \,.$$

Then using this value in Equation 5.2, we can determine C_{eff} for each benchmark by

$$C_{\text{eff}} = \frac{P_{\text{dynamic}}}{fV^2} = \frac{P_{\text{cpu}} - P_{\text{static}} - P_{\text{uncore}}}{fV^2} \,.$$

This process yields a set of IPC-C_{eff} pairs, one for each benchmark, to which we can apply linear regression to approximate the function $C_{\text{eff}}(\text{IPC})$, which we can then substitute into Equation 5.11.

For this purpose we use the EEMBC telecommunications benchmark suite [Emb], comprised of 16 compute kernels representative of DSP algorithms common in telecommunications applications, such as convolution and FFT. We observe empirically that these benchmarks show a wide range of IPC behaviour and are reasonably CPU-bound, minimising variation in P_{uncore} from the measurements of Section 5.3.3 which would be caused by L2 cache and memory activity. We verify that these benchmarks are indeed CPU-bound by comparing the measured IPC at minimum and maximum CPU frequency—if performance scales linearly with frequency, the workload is CPU-bound, since memory latency does not scale with frequency. This can also be done directly by measuring the L1 cache miss rate using performance counters. Across all 16 benchmarks, we observe IPC values between 1.0 and 2.3, varying at most by 3 % between minimum and maximum frequency, showing that these workloads are indeed highly CPU-bound. Applying the procedure outlined above, we are able to produce a model of IPC to C_{eff}, with $R^2 = 0.73$.

Finally, we add this model to Equation 5.11, and solve $P'_{\text{CPU}}(u, \text{IPC}) = 0$ for f; in other words, given a workload characterised by its IPC and utilisation u, we can predict the frequency f that minimises power, and hence energy consumption. We use Mathematica to solve this function numerically at the two extremes of IPC, 0 and 3, since the *S4-S* has a triple-issue pipeline. Figure 5.10 plots the predicted optimal frequency as a function of utilisation.

Not all points on the optimal frequency curves correspond to valid operating points because we have been treating n and f as continuous and unbounded, when in fact they are neither. For example, at $u = 0.5$, the energy-optimal frequency is approximately 700 MHz, but achieving the necessary throughput at this frequency would require more than four online cores. Since the platform has only four, we have no option but to run at a sub-optimal frequency. The red and orange dashed lines show the limits of valid operating points, corresponding to one and four online cores.

From Figure 5.10 we observe several interesting characteristics of the optimal frequency

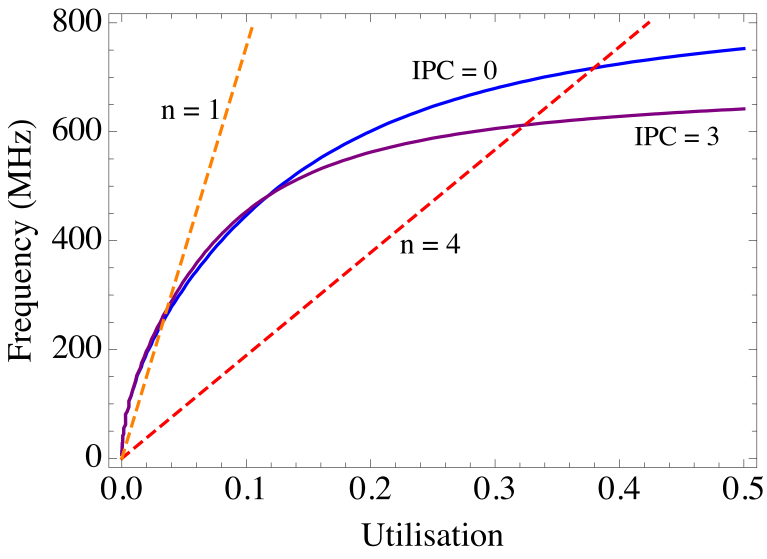


Figure 5.10: Optimal operating frequency predicted by our power model for *S4-S* as a function of utilisation for minimum (0) and maximum (3) IPC. The dashed lines delimit valid operating points.

curves. At low utilisation, the optimal frequency closely tracks the $n = 1$ line, which means that increasing frequency is the energy-efficient response to increasing load. As utilisation increases, the curves deviate and the gradient of the energy-optimal curves rapidly drop. This means we should reduce the rate of frequency increases, requiring the onlining of additional cores to provide the necessary capacity. From utilisations above 40 %, the optimal frequency curves leave the bounds of feasible operating points. Within the valid operating points, the characteristic IPC of the running workload does not significantly affect the optimal OP. Thus, for the remainder of this work, we will assume a fixed IPC at the midpoint 1.5.

5.3.5 Optimal operating point tracking

We have proposed a methodology for determining the optimal frequency (and, by implication, the number of online cores) for a workload with a specific processor utilisation. To put this into practice as an online power-management system, we need to solve several challenges:

- For the purposes of analysis, we made the assumption that n and f are continuous. These parameters need to be discretized to the available operating points;
- The utilisation of a particular workload is not generally known a priori; and
- Workloads do not necessarily have fixed utilisation, but can change over time.

To solve these issues, we propose the algorithm depicted in Figure 5.11 to track the curve of optimal energy consumption. The algorithm is parameterized by a *threshold frequency*,

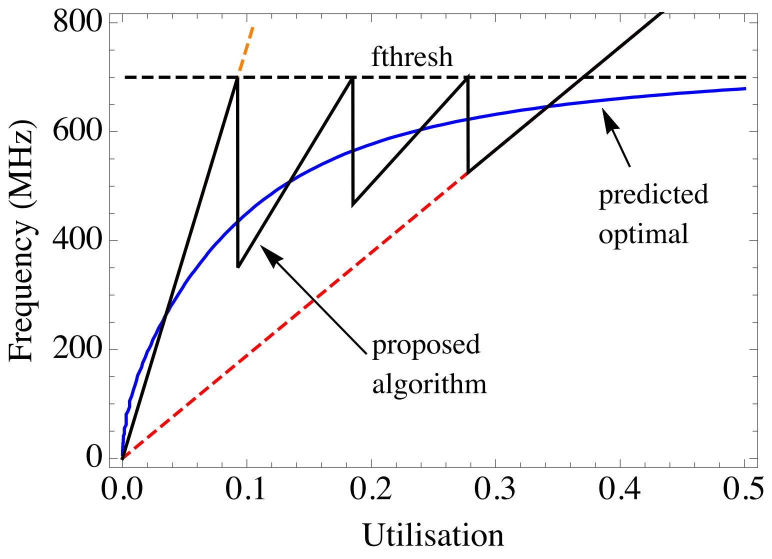


Figure 5.11: Proposed algorithm, shown as the target frequency as a function of utilisation. The spikes in frequency correspond to the onlining/offlining of cores.

f_{thresh}, which intuitively is the platform-specific cross-over point between per-core static and dynamic power. In the figure, we use a hypothetical f_{thresh} value of 700 MHz for illustration purposes. The solid black line represents the (continuous) frequency selected by the algorithm at particular utilisation levels—the actual frequency should be the lowest supported frequency greater than or equal to the this (i.e. above the curve).

As utilisation increases from 0, the frequency is increased on a single core until reaching the threshold frequency. At this point, we online a second core, which means the frequency can be halved while still achieving the same throughput. As utilisation increases from this point, we again scale frequency, now on two cores, until the threshold is reached again, where the third core is onlined. Once all cores have been onlined in this manner, we have no choice but to increase frequency.

This algorithm has several desirable properties:

- A single, well-defined tunable parameter f_{thresh} to adapt it to the characteristics of each platform;
- No workload-specific knowledge;
- It is implementable, as we will soon show.

The effectiveness of this policy clearly depends on selecting an appropriate threshold frequency, for which we propose the following procedure. Using our existing power model, we predict, for each candidate f_{thresh}, the average energy consumption under the policy. We do

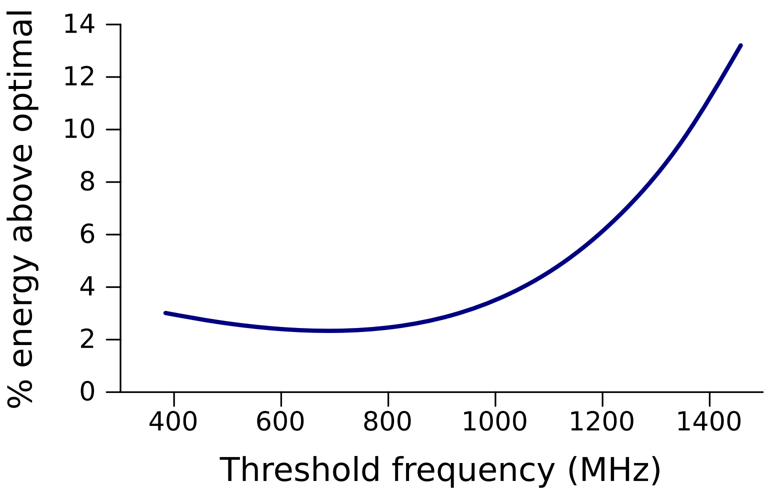


Figure 5.12: Predicted energy consumption of proposed algorithm for varying threshold frequency (f_{thresh}), as a percentage above predicted optimal.

this by integrating the sawtooth curve shown in Figure 5.11 over utilisation, from $u = 0$ to $u = 0.6$. We cap the integration at $u = 0.6$ to improve the fidelity of the analysis—at high utilisations the optimal operating point is not feasible, but the predicted error grows rapidly. We repeat the same procedure on the predicted optimal curve. Finally, we plot the percentage power increase of the proposed policy compared with the predicted optimal, over the range of threshold frequencies. A lower percentage means that the effectiveness of the algorithm is closer to the predicted theoretical optimum. In Figure 5.12, we show the result of this analysis on the *S4-S* platform.

From Figure 5.12 we see that the optimal f_{thresh} selection lies in the 600–800 MHz range. However, threshold frequency selection is not particularly sensitive over quite a wide range: from 400 to 1000 MHz, energy varies by only about 1 % of optimal. At 1000 MHz or above however, the energy loss becomes significant. On the basis of this result, we shall select $f_{\text{thresh}} = 810$ MHz, which is the nearest supported frequency on the *S4-S* at the upper knee of the curve. Selecting a higher threshold frequency, where efficiency is not affected, reduces the density of core online/offline events, which we hope will improve performance and stability of the algorithm.

At the extremes, setting $f_{\text{thresh}} = 0$ corresponds to a policy of onlining all cores before increasing frequency at all—such a policy applies for systems with very low P_{idle}, such as the *S4-E*. On the other hand, setting $f_{\text{thresh}} = \infty$ corresponds to maximising frequency before onlining any additional cores, useful only on (probably fictional) systems where P_{dynamic} is sub-linear in f.

5.3.6 Summary

We have shown that running a workload on more cores can reduce energy consumption by two mechanisms:

1. allowing access to lower, more energy-efficient core frequencies for equivalent throughput; and

2. reducing execution time and thus minimising the energy contribution of the per-core static power and the chip-wide uncore.

This result is enabled by low per-core static power, which we have shown to be true for the *S4-E*. From this it follows that the optimal policy for such devices is to run all cores at the minimum frequency where utilisation is $\leq 100\,\%$, offlining cores only when they idle. On *S4-S* however, we observed comparatively high per-core static power, so the same policy is not energy-optimal on that platform. However, we can predict the optimal OP from a processor power model. We proposed an algorithm utilising this, which we now implement and evaluate.

5.4 Medusa: an offline-aware frequency governor

Based on our insights, we implemented an energy management policy, *medusa*, in the Linux kernel running on both Galaxy S4 platforms. The goal of this policy is to control the CPU frequency and number of online cores to minimise energy consumption by managing CPU slack time, but without adversely affecting performance. The implementation is a Linux “cpufreq” DVFS governor which controls frequency in the standard way, but also configures the number of online cores with the `cpu_up()` and `cpu_down()` primitives.

The operation of medusa is summarised by the two invariants we attempt to maintain:

1. maximise utilisation $< 100\,\%$; and

2. minimise $|f - f_{\text{thresh}}|$.

The number of cores to online is implied by the these invariants. However, this comes with certain practical constraints. If the number of runnable threads is equal to the number of online cores, there is no benefit to onlining additional cores, even if the existing cores are fully utilised, because each thread occupies at most one core. Utilisation is estimated by sampling and averaging the number of threads on each core’s runnable scheduler queue.

In more detail, medusa runs every 100 ms, and executes the following algorithm:

1. Select a candidate frequency, f_{new}, which is predicted to maximise utilisation without reaching $100\,\%$.

2. If $f_{\text{new}} > f_{\text{thresh}}$, and the number of runnable threads exceeds the number of online cores, online an additional core if available. Otherwise, increase frequency to f_{new}.

3. If $f_{\text{new}} \leq f_{\text{thresh}}$, predict the frequency f'_{new} that would be required to run the current load with $n-1$ online cores. If $f'_{\text{new}} \leq f_{\text{thresh}}$, then offline one core and switch to f'_{new}. Otherwise, switch to f_{new}.

4. If any core is below 5 % utilisation, offline a core.

The candidate frequency f_{new} is predicted with the formula

$$f_{\text{new}} = f_{\text{current}} \times (u_{\text{avg}} + \epsilon) \quad ,$$

and f'_{new} is chosen by

$$f'_{\text{new}} = \frac{n}{n-1} \times f_{\text{new}} \quad ,$$

where u_{avg} is the current average utilisation across all online cores, and $n/(n-1)$ is the decrease in compute capacity when switching from n to $n-1$ cores. We implicitly round up to the next supported frequency. We select 100 ms on that basis that other implementations use similar values: we observed sampling periods of 50–500 ms, depending on the device.

In practice, we apply averaging and hysteresis to most calculations to improve OP stability, particularly to avoid oscillations and thus expensive unnecessary OP switches. For example, we slightly delay offlining a core to account for temporary reductions in system load. On the other hand, we are reasonably aggressive in increasing frequency and onlining cores to improve responsiveness. These increases accelerate—the longer the system has been overloaded, the faster we increase the OP. This is necessary due to a fundamental limitation in using runnable threads as a measure for utilisation, which is that there is no way to discern the degree of overload. We have to effectively search for the real load level, by increasing the OP until average scheduling queue residency drops below 100 %.

The implementation, including extensive configuration and debug support, is 1500 lines of C code.[1]

5.5 Evaluation

We now evaluate the energy consumption under medusa, comparing it against some existing policies. We also investigate the selection of f_{thresh}, both the quality of our theory-based selection process, and a sensitivity analysis to determine how important that selection is. For these benchmarks, we configure medusa on *S4-E* with $f_{\text{thresh}} = 0$, which corresponds to always onlining cores before increasing frequency, where possible. As we showed earlier,

[1]The code is available for download at https://github.com/xaaronc/medusa.

this platform has very low P_{static}, so turning on cores is very energy-efficient compared with increasing frequency. On *S4-S*, we use $f_{\text{thresh}} = 810\,\text{MHz}$, which we selected using the results from Figure 5.12.

An online energy policy has two related but distinct goals. Firstly, selecting the optimal OP for the current workload, and secondly, adapting that selection as the workload changes over time. While our implementation does address both issues out of necessity, our analysis has focused entirely on the first issue. Furthermore, a general-purpose algorithm might deviate from the optimal OP for non-energy-efficiency reasons like quality of service. For example, it may be desirable from a user perspective to increase the OP above the energy-optimal point to provide a certain degree to performance headroom, so that sudden increases in load can be handled immediately with the excess capacity. Such trade-offs are outside the scope of this work, so our benchmarks are focused primarily on OP selection.

5.5.1 Benchmarks

We evaluate medusa with a series of benchmarks of three types. In the first case, we use the *loadcpu* workload at the 10, 25, 50, and 75 % load levels. For each, we compare the performance of medusa with the *static-optimal*. We determine the static-optimal by running the load at every feasible OP and selecting the one that yields lowest energy consumption. These have been determined earlier in Section 5.2. We call it *static* because the OP is fixed for the full run of the benchmark, whereas medusa is a *dynamic* policy because it changes operating point at runtime. However, since this synthetic workload is homogeneous, static-optimal is in fact globally optimal in this case. We also compare with the performance of the policies that ship with the devices, which we call *default*. On *S4-S*, this consists of the *ondemand* frequency governor, plus a (closed source) userspace daemon called *mpdecision* which controls the number of online cores. On *S4-E*, OP is, like medusa, controlled entirely in the kernel with a modified version of *ondemand*.

Our second benchmark is a video playback application configured to use a software decoder to play an H.264-encoded video. We compare medusa with default as above, but also compare with two of the standard Linux DVFS governors, *ondemand* and *conservative*. These governors control only frequency, so we perform a similar procedure as above to select the static optimal number of cores. We repeat the benchmark with the governor under test four times, once for each online core count set statically for the full run. We then report the minimum energy across the four scenarios.

Finally, we use two mobile benchmarking applications, *AnTuTu 3DRating* and *Vellamo HTML5*. We compare only medusa and default because these are dynamic (i.e. time-varying) workloads, so it makes no sense to statically select the number of cores for benchmarking the other governors. In addition to measuring the energy consumption of each benchmark, we also report the "score" which is a performance metric produced by the benchmark application

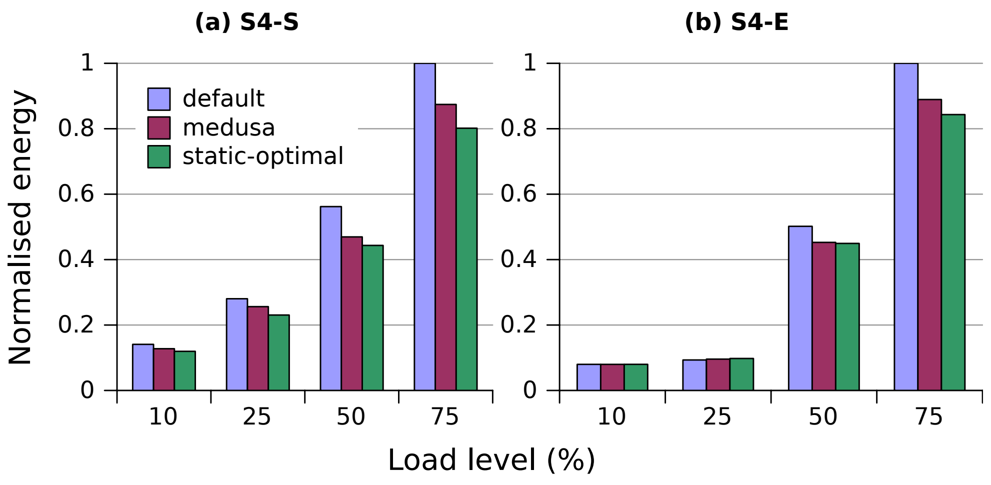


Figure 5.13: Energy consumption of *loadcpu* on both devices under three energy-management policies. For static-optimal the frequency and number of cores is fixed.

which reflects the aggregate performance of the various test workloads it executes. We use this number as an attempt to quantify the impact on performance. As discussed above, performance can vary across policies due to different approaches in responding to workload variation. Both policies, in the steady state, keep utilisation below 100 %, however the default policy has been designed for dynamic OP selection, whereas medusa has not. Nonetheless we include these results as a sanity check, if nothing else. In the case of Vellamo, since we are running in airplane mode, the parts of the benchmark that require network connectivity are not used.

All of these benchmarks are multi-threaded. For *loadcpu*, as earlier, the total work is split over four processes. For video, AnTuTu and Vellamo, the degree of parallelism is controlled by the applications themselves. The related configuration options are left to their default values. For Vellamo, AnTuTu and video, the screen is on at minimum brightness.

5.5.2 Results

Figure 5.13 shows the performance of medusa with the *loadcpu* workload. Comparing geometric mean across all load levels, medusa achieves within 8 % (11 % worst case) of static-optimal on *S4-S*, and within 1 % (5 % worst case) on *S4-E*. The default policy is on average 23 % above optimal on *S4-S* (27 % worst case) and 6 % on *S4-E* (with 19 % worst case). Across all results, the maximum RSD (relative standard deviation) is 7.3 %, and the geometric mean is 1.6 %.

Figure 5.14 shows the results of the video playback benchmark. On both platforms, medusa has the lowest energy consumption: 88 % of default on average. For ondemand, the minimal energy is with one core on *S4-S*, and 4 cores on *S4-E*, and for conservative, 2 cores on *S4-S*. This is consistent with our expectations. The conservative governor on *S4-E* performs extremely poorly, consistently setting frequency vastly above the performance required. Since

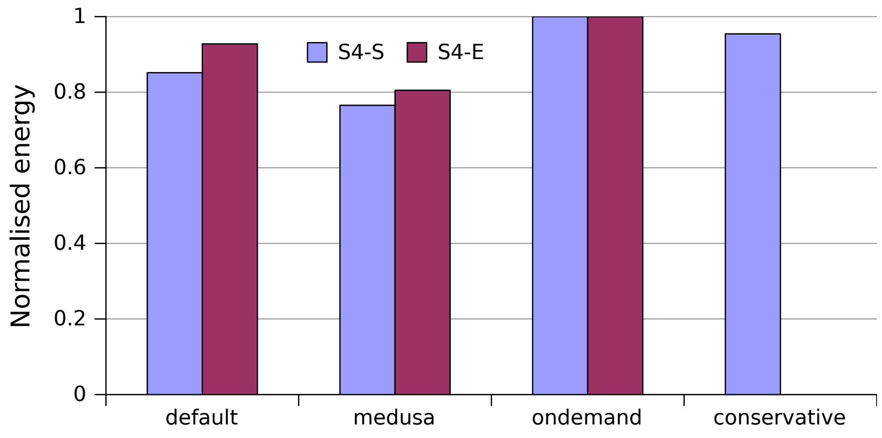


Figure 5.14: Video playback energy consumption on both devices under medusa and default, as well as Linux standard ondemand and conservative governors with the number of cores set statically to that which minimises energy.

this governor performs correctly on *S4-S*, the poor behaviour is probably due to a bug, and thus for fairness we omit the data. The maximum RSD of the reported results is 4.1 % and the geometric mean is 0.9 %.

Figure 5.15 shows the results of AnTuTu 3DMark and Vellamo HTML5 of medusa relative to default, for both energy and benchmark score. In all cases, medusa has both a lower energy consumption and lower score than default. On *S4-S*, medusa reduces energy by an average of 26 %, while the score decreases by only 10 %. For *S4-E*, the energy and score decrease by 6 % and 7 % respectively.

We have already shown that medusa performs well compared with other policies under reasonably static workloads such as *loadgen* and video playback. By contrast, Vellamo and AnTuTu are dynamic workloads and as such, they are sensitive to the algorithm used to react to changing load. As discussed above, we have not focused on this issue. Nonetheless, we can attempt a quantitative evaluation of medusa's performance under such workloads using the benchmark throughput score. Using benchmark score per unit energy as the figure of merit:

$$\mathrm{FoM} = \frac{\mathrm{score}}{E} ,$$

which is roughly analogous to the popular energy-delay product (EDP) metric, default and medusa perform within 1 % of each other on *S4-E*, but on *S4-S*, medusa outperforms default by 22 %. The maximum RSD of the measured energy is 11.7% with a geometric mean of 3.2 %, and for the measured score maximum RSD is 3.7 % with geometric mean 1.0 %.

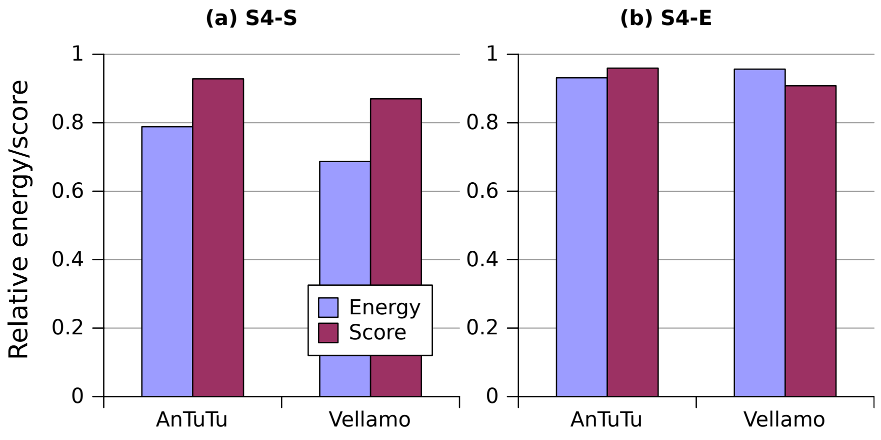


Figure 5.15: AnTuTu 3DMark and Vellamo HTML5 energy consumption and score results of medusa, relative to default.

5.5.3 f_{thresh} sensitivity analysis

In our implementation, we determine f_{thresh} for *S4-S* by combining our model of energy consumption with a model of the algorithm implemented by medusa. To determine the effect of this choice on energy consumption, we perform the following sensitivity analysis. Using the *loadcpu* program on *S4-S*, we run five benchmarks at 10, 15, 20, 30, and 40 % total load level, and measure the energy consumption of each of these benchmarks under the medusa policy, with f_{fthresh} set to 486, 594, 702, 810, 918, 1026, and 1134 MHz. We then calculate the overall efficiency of each threshold frequency by the following method.

Let E_l^f denote the energy consumption of load level l at a particular threshold frequency f. Then we calculate $\hat{E}_l$, which is the minimum energy consumption of load l over each of the p threshold frequencies:

$$\hat{E}_l = \min_{f=f_1}^{f_p} E_l^f \, ,$$

That is, $\hat{E}_l$ is the minimum energy consumption for load l. Then for a particular threshold frequency f, we calculate the overall efficiency of that frequency (denoted by $\nu(f)$) as the geometric mean of the increase in energy consumption of E_l^f compared with $\hat{E}_l$, across all load levels. Mathematically,

$$\nu(f) = \sqrt[q]{\prod_{l=l_1}^{l_q} \frac{E_l^f}{\hat{E}_l}} \, ,$$

where q is the number of load levels. The results are graphed in Figure 5.16.

The results indicate that the optimal selection of f_{thresh} is 810 MHz, which consumes approximately 1.5 % above the minimum energy, on average. The cost of incorrectly setting the threshold frequency is much higher if it is over-estimated (i.e. higher than optimal), compared

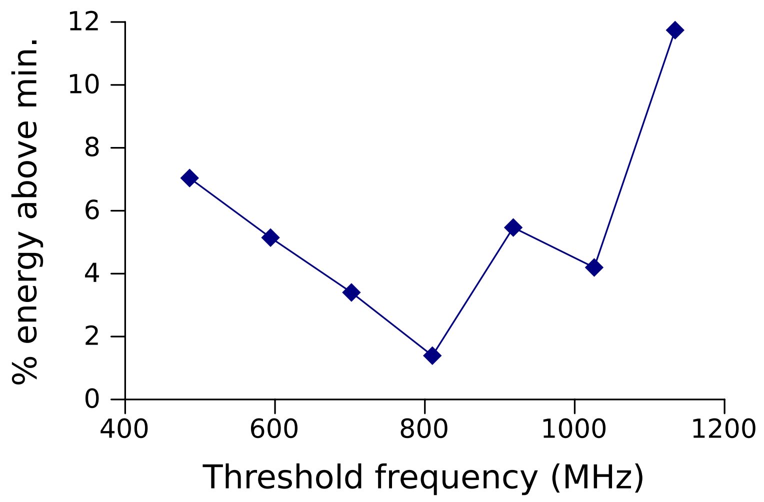


Figure 5.16: Energy consumption vs. f_{thresh} selection, as a percentage above minimum.

with setting it below optimal. This is in line with expectation, because a higher threshold means the core(s) spend more time at higher frequencies, and the power curve grows rapidly in this area due to the quadratic dependence on voltage.

The anomaly at $f = 918\,\text{MHz}$ is statistically significant—all data points have a relative standard deviation of well under 10 %—but the reason for this behaviour is unknown. We see a uniform increase in power consumption at all load levels, compared with both 1026 MHz and 810 MHz, suggesting that the issue is not algorithmic, as that would be unlikely to affect all load levels equally. Rather, this probably points to a deficiency in our modeling due to some characteristic of the platform that is not accounted for. Nonetheless, we see a reasonably good match in the predictions made by the model, compared with the experimental approach. Both predict a range of threshold frequencies in a window of approximately 300 MHz where the energy consumption is not particularly sensitive, performing within a few percentage points of the minimum. Moreover, our somewhat arbitrary choice of $f_{\text{thresh}} = 810\,\text{MHz}$ within that range has turned out to be the best selection in practice. We argued that a higher threshold has the property of reducing switching overhead, and while that appears that it may be the case, the risk of over-estimating the frequency may outweigh this benefit in practice. Furthermore, the theoretical approach to determining f_{thresh} is quite cumbersome, requiring measurements of the device's static power and effective capacitance, and mathematical models of both the power consumption and algorithm. Thus we concede that the empirical approach based on benchmarking and total power measurements is more practical.

Chapter 6

Conclusion

In this work we make two main contributions:

- a detailed analysis of the energy consumption of several smartphones by direct measurement of the individual components under realistic usage scenarios, and;
- the design and implementation of a DVFS control algorithm for multi-core processors which incorporates knowledge of offline power states.

For our energy consumption analysis, we study two smartphone devices in detail: the Openmoko Freerunner Neo, and the Samsung Galaxy S III. We directly measure power consumed by each component of the device by inserting current sense resistors between the power supply unit and the components. Using a data acquisition unit to sample the voltage at the component's power supply and the voltage drop across the sense resistor, we are able to accurately measure instantaneous power consumption. Integrating this over time yields total energy. These measurements show how the different components of the device contribute to overall power consumption.

With this methodology we are able to determine the power consumption of components such as the CPU, RAM, GPU, WiFi and cellular radios, GPS, Bluetooth, display panel and touchscreen, audio, storage, and environmental sensors, as well as total power consumption at the battery terminals. Some of these components we measured directly, while others are indirectly determined by subtracting known power values from the device's total power consumption.

On the Freerunner, this kind of instrumentation is possible due to both the open nature of the design—specifically that the schematics are freely available—and because the device was designed with placeholders for current-sense resistors. Commercial off-the-shelf devices are not generally amenable to this approach. However, we are in fact able to replicate our measurements on a mass-market device, the Galaxy S III, using an observation about the design of smartphone power supplies, namely that efficiency necessitates the use of switch-mode designs which feature large discrete inductors in series with the supply rail, offering an ideal place to break the circuit for current sensing.

In addition to the fine-grained measurements made on the Freerunner and Galaxy devices, we also make full-system measurements of two other devices, the HTC Dream and the Google Nexus One, to validate our results.

We find that the display is the most significant contributor to overall energy consumption in almost every use case, while network and camera also use substantial energy in scenarios where those devices are active. Surprisingly, we find that RAM is not as important as might first be thought: although its peak power is very high (approximately 200 mW), this always correlates with high power in either the CPU or GPU, and thus consumes a small amount of energy relative to the full system. Storage shows a similar trend.

Across device generations, we see sleep power is approximately constant, while idle power is increasing, probably due to larger and higher-resolution screens. CPU efficiency has increased significantly, as has peak performance and power consumption.

The ultimate aim of this work is to enable a data-driven approach to improving power management on mobile devices. We hope that by presenting this ground-truth, we will enable such future research. Furthermore, by publishing a methodology for the instrumentation and analysis of commercial off-the-shelf devices like the Galaxy S III, we hope to reduce the barrier for other researchers to use physical measurements on real devices for their own work, rather than relying purely on models and estimates, as is all too common.

Two interesting observations emerge from the above measurements

- smartphone peak power consumption is increasing significantly faster than average power, and;
- the CPU is largely responsible for this.

With this as motivation, we set out to study how best to manage power on emerging smartphone processors. These can be qualified not just by increasing frequency and complexity, but also increasing core count. Indeed, multi-core application processors are becoming prevalent even on low- and mid-range devices.

Historically, reducing frequency is the main mechanism for reducing energy consumption, due to the super-linear dependence of power on frequency. A multi-core processor however presents an additional dimension along which power can be controlled: the number of cores active at a given time. In this work we consider how the two mechanisms can be used in tandem, and what are the right trade-offs for selecting one over the other based on the workload and devices.

We can divide mobile multi-core processors into classes based on the static power cost of onlining a core. That is, how much does power consumption increase, independent of the selected frequency, if we enable another core. Due to how core power supply networks are designed, devices fall one of two classes based on whether the static core-online cost is significant or negligible. For devices with low static core power, onlining more cores generally leads to *decreased* energy consumption. This is perhaps counter-intuitive, but occurs for two reasons:

- onlining additional cores allows running them at lower, more energy-efficient frequencies, and;
- the additional throughput allows the faster completion of a workload, and thus reduces the accumulation of static power, otherwise known as race-to-idle.

However, when per-core static power is non-negligible, it is not necessarily the case that running with more onlining cores is more energy efficient. Specifically, it depends on the relative contribution of the linear per-core static power to the super-linear dynamic component. For this class of device we have shown that, up to a certain frequency which we call the *threshold frequency*, reducing frequency is preferable to onlining cores. However, once the threshold frequency is reached, it is more efficient to online cores than to further increase frequency. This threshold frequency can be thought of as the cross-over point between the contributions of static and dynamic power. We show how to determine the threshold frequency using both a theoretical model based on various parameters of the processor in question, and a simpler measurement-based empirical approach.

Based on this we design *medusa*, a power management algorithm running in the Linux kernel which controls both frequency and the number of active cores. We implement this on two Galaxy S4 smartphones; one with an Exynos 5410 SoC, and another with the Snapdragon 600, which show low and high static core power, respectively. We show that medusa is capable of achieving close to optimal energy consumption for static workloads by comparing it with an oracle. We also benchmark it against several existing policies, including the default policy shipping on the devices, with favourable results: 12 % energy reduction on average for video playback, and a 22 % improvement in energy-delay product for macro-benchmarks on a device with high per-core static power.

6.1 Limitations

Any empirical analysis is inherently tied to the platform on which it is performed. We try to minimise device specificity by analysing multiple platforms, specifically including a validation phase where we compare total power consumption of additional devices under similar benchmarking conditions as a sanity test of the results. Nonetheless, this does not guarantee that our results are as widely applicable as we would hope.

This problem is exacerbated by the fact that the Freerunner is not a typical commercial off-the-shelf device. Even at the time of that work, it was not a latest-generation mobile phone, and is missing several features typical of contemporaneous devices such as a 3G cellular interface, which supports a much higher data rates than the 2.5G GPRS interface. Moreover, the application processor is based on the dated ARMv4 architecture, though clocked at a rate consistent with 2009-vintage smartphones.

Our medusa work, both the theoretical analysis and the implementation, largely ignores the effect of memory accesses. However, we believe this limitation is not quite as severe as it sounds. Firstly, splitting a workload across multiple cores does not change the set of cache lines actually accessed, just the timings. Secondly, while using multiple cores increases the effective cache size due to the private L1 caches, this causes a significant effect only if the workload's working set size falls between $n\,|L1|$ and $(n+1)\,|L1|$, where n is the number of cores active and $|L1|$ is the size of the L1 cache. Of course this is only true if the cores do not conflict significantly on shared resources, such as last-level cache and memory. Anecdotally we find this to largely be the case, and these findings were explored more thoroughly in a work by Kim et al. [KWC+16], who showed that features like wide memory buses and bank interleaving alleviate this issue significantly.

We have also made a number of assumptions in the processor power model. While we believe these to be reasonable for mobile multi-cores, it is unlikely to apply to a wider range of devices. For example, x86 processors have a wide range of idle "C-states" which are critical to achieving good power efficiency on those platforms. Indeed we experimented with an x86 Atom-based tablet and found that our 3-state model was unable to capture its behaviour with sufficient accuracy.

6.2 Future directions

Due to the constant evolution of mobile devices, not just in performance and size but also features, we believe that ongoing physical power measurements and analyses, like the ones we performed, will be important to direct future power management research by providing accurate ground truths.

The holy grail of power management on modern devices is a multi-core-aware online dynamic power and performance model: essentially Koala [SLSPH09] extended to multi-core. To date this has proved elusive due to the complex interaction between shared resources and the lack of suitable performance counters. With such a model we could feasibly extend medusa to incorporate active energy management, allowing the trading of performance for a reduction in energy consumption.

Our work has assumed a multi-core model where each core is identical and has the same latency to all memory, i.e. SMP. However, heterogeneous multi-cores (HMP), where cores can differ in speed, complexity, and power consumption, are starting to be seen in the mobile world. Handling this type of processor will be crucial for power consumption in emerging devices. An extension of this is processor specialisation, where certain processors are designed for, and dedicated to, certain tasks. The GPU is a well-known and ubiquitous example of this, but trends indicate that specialisation may become more common for other tasks such as sensor management.

List of Figures

List of Tables

Bibliography

[AB04] James H. Anderson and Sanjoy K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, Tokyo, JP, March 2004. 36

[And09] Andriod on Freerunner community. 2009. http://code.google.com/p/android-on-freerunner/. 51

[AV12] David Perez Abreu and Maria E. Villapol. Measuring the energy consumption of communication interfaces on smartphones using a moderately-invasive technique. In *Global Information Infrastructure and Networking Symposium (GIIS)*, Choroni, VZ, December 2012. 30

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–58, New Orleans, LA, US, February 1999. USENIX. 42

[Bel00] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th SIGOPS European Workshop*, Kolding, DK, September 2000. 30

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, December 2007. 43

[BJ07] W. Lloyd Bircher and Lizy K. John. Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, San Jose, CA, US, April 2007. 31

[BJ08] W. Lloyd Bircher and Lizy K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd International Conference on Supercomputing*, Kos, GR, June 2008. 37, 86

[BLWM14] Rajesh Krishna Balan, Youngki Lee, Tan Kiat Wee, and Archan Misra. The challenge of continuous mobile context sensing. In *6th International Conference on Communication Systems and Networks (COMSNETS)*, Bangalore, IN, January 2014. 29

[BMH$^+$13] Ge Bai, Hansi Mou, Yinhong Hou, Yongqiang Lyu, and Weikang Yang. Android power management and analyses of power consumption in an Android smartphone. In *IEEE International Conference on High-Performance Computing and Communications*, Zhangjiajie, CN, November 2013. 26

[BSA$^+$15] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the 10th EuroSys Conference*, Bordeaux, FR, April 2015. 42

[BWWK03] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP)*, New Orleans, LA, US, September 2003. 41

[CAT$^+$01] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Alberta, CA, October 2001. 42

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference (ATC)*, pages 271–284, Boston, MA, US, June 2010. xiii, 26, 45

[CH13a] Aaron Carroll and Gernot Heiser. Mobile multicores: Use them or waste them. In *Proceedings of the 2013 Workshop on Power Aware Computing and Systems (HotPower'13)*, Farmington, PA, US, November 2013. xiii, 85

[CH13b] Aaron Carroll and Gernot Heiser. The systems hackers guide to the Galaxy: Energy usage in a modern smartphone. In *Asia-Pacific Workshop on Systems (APSys)*, Singapore, July 2013. ACM. xiii, 45

[CH14] Aaron Carroll and Gernot Heiser. Unifying DVFS and offlining in mobile multicores. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 287–296, Berlin, DE, April 2014. xiii, 85

[CHCR11] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & cap: adaptive DVFS and thread packing under power caps. In *Proceedings of the*

44th ACM/IEE International Symposium on Microarchitecture, pages 175–185. ACM, 2011. 40

[CHK06] Jian-Jia Chen, Heng-Ruey Hsu, and Tei-Wei Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 408–417. IEEE, 2006. 38

[CWS11] Hui Chen, Shinan Wang, and Weisong Shi. Where does the power go in a computer system: Experimental analysis and implications. In *International Green Computing Conference*, Orlando, FL, US, July 2011. 29

[CZBC11] Zehan Cui, Yan Zhu, Yungang Bao, and Mingyu Chen. A fine-grained component-level power measurement method. In *International Green Computing Conference*, Orlando, FL, US, July 2011. 28

[DCZ09] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power modeling of graphical user interfaces on OLED displays. In *Proceedings of the 46th Design Automation Conference (DAC)*, San Francisco, CA, USA, July 2009. 59

[EBA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, San Jose, CA, US, June 2011. 33

[Emb] Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite. http://www.eembc.org. 104

[FHM$^+$13] Miguel A. Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser, and Joost Visser. Seflab: A lab for measuring software energy footprints. In *2nd International Workshop on Green and Sustainable Software (GREENS)*, San Francisco, CA, US, May 2013. 28

[FS99a] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 48–63, Kiawah Island, NC, US, December 1999. 27, 43

[FS99b] Jason Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE, Workshop on Mobile Computing Systems and Applications*, 1999. 27, 31

[GBK$^+$12] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, and Karsten Schwan. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, US, June 2012. 38, 97, 103

[GPP10] M. Ghasemazar, E. Pakbaznia, and M. Pedram. Minimizing energy consumption of a chip multiprocessor through simultaneous core consolidation and DVFS. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 49–52. IEEE, 2010. 37

[Gre11] Peter Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. http://www.eetimes.com/document.asp?doc_id=1279167, October 2011. [Online; accessed Mar 2017]. 87

[HCH+14] Chun-Ying Huang, Po-Han Chen, Yu-Ling Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Measuring the client performance and energy consumption in mobile cloud gaming. In *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*, Nagoya, Japan, December 2014. 51

[HCK11] Faisal Hamady, Ali Chehab, and Ayman Kayssi. Energy consumption breakdown of a modern mobile platform under various workloads. In *International Conference Energy Aware Computing (ICEAC)*, Istanbul, TR, November 2011. 28

[HIM+09] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification. http://www.acpi.info/spec.htm, June 2009. 15

[HM07] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, Portland, OR, US, August 2007. 87, 96

[IBC+06] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th ACM/IEE International Symposium on Microarchitecture*, pages 347–358, Orlando, FL, US, December 2006. 39

[ICM06] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th ACM/IEE International Symposium on Microarchitecture*, pages 359–370, Orlando, FL, US, December 2006. 35

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings of the 36th ACM/IEE International Symposium on Microarchitecture*, December 2003. 30, 41

[KAB$^+$03] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003. 12, 15

[KADB02] RK Krishnarnurthy, Atila Alvandpour, Vivek De, and Shekhar Borkar. High-performance and low-power challenges for sub-70 nm microprocessor circuits. In *IEEE Custom Integrated Circuits Conference*, pages 125–128, May 2002. 12

[KMD13] Immanuel König, Abdul Qudoos Memon, and Klaus David. Energy consumption of the sensors of smartphones. In *Proceedings of the Tenth International Symposium on Wireless Communication Systems (ISWCS)*, Ilmenau, DE, August 2013. 29

[KWC$^+$16] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, AT, April 2016. 118

[LHL05] Weiping Liao, Lei He, and Kevin M Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICAS)*, 24(7):1042–1053, 2005. 13

[LM06] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 77–87. IEEE, 2006. 36, 44

[LSH10] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Workshop on Power Aware Computing and Systems*, pages 1–5, Vancouver, Canada, October 2010. 35

[LSH11] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX Annual Technical Conference (ATC)*, Portland, Oregon, USA, June 2011. USENIX. 35, 86

[LWC+14] Robert LiKamWa, Zhen Wang, Aaron Carroll, Felix Xiaozhu Lin, and Lin Zhong. Draining our glass: An energy and heat characterization of Google Glass. In *Asia-Pacific Workshop on Systems (APSys)*, Beijing, CN, June 2014. xiii, 14, 26

[LWLZ12] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012. 42

[LWZ14] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, US, March 2014. 42

[LZCF14] Xiangyu Li, Xiao Zhang, Kongyang Chen, and Shengzhong Feng. Measurement and analysis of energy consumption on Android smartphones. In *4th IEEE International Conference on Information Science and Technology (ICIST)*, Shenzhen, CN, April 2014. 26

[MAK+11] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekarand Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, June 2011. 31

[MB06] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st EuroSys Conference*, Leuven, BE, April 2006. 41

[MB08a] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'08)*, San Diego, CA, US, December 2008. 38

[MB08b] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Proceedings of the 3rd EuroSys Conference*, Glasgow, UK, March 2008. 41

[MGW09] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, US, March 2009. 42

[MLH+02] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, pages 35–44, New York, NY, US, June 2002. 33, 86, 98

[MRR05] Saibal Mukhopadhyay, Arijit Raychowdhury, and Kaushik Roy. Accurate estimation of total leakage in nanometer-scale bulk CMOS circuits based on device geometry and doping profile. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CAD-ICAS)*, 24(3):363–381, 2005. 12, 13

[MSB10] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th EuroSys Conference*, Paris, FR, April 2010. 38

[MV04] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Proceedings of the 2004 Workshop on Power-Aware Computer Systems*, Portland, OR, US, December 2004. 27

[Nat16] National Instruments Corporation. NI 6229 device specifications. http://www.ni.com/pdf/manuals/375204c.pdf, June 2016. [Online; accessed Mar 2017]. 48

[NS00] Koichi Nose and Takayasu Sakurai. Analysis and future trend of short-circuit power. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CAD-ICAS)*, 19(9):1023–1030, 2000. 11

[Ope08] OpenMoko, Inc. GTA02 – Neo Freerunner schematics. http://downloads.openmoko.org/developer/schematics/GTA02, August 2008. [Online; accessed Jan 2010]. 51

[PFW11] G.P. Perrucci, F.H.P. Fitzek, and J. Widmer. Survey on energy consumption entities on the smartphone platform. In *73rd IEEE Vehicular Technology Conference*, Budapest, HU, May 2011. 26

[PGV04] Michael D. Powell, Mohamed Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 260–270, Boston, MA, US, September 2004. 42

[Phi04] Philips Electronics N.V. *PCF50606 Controller for Power Supply and Battery Management*, April 2004. Document 12NC Rev. 2.2. 50

[PHZ$^+$11] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th EuroSys Conference*, Salzburg, AT, April 2011. 32

[PHZ12] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th EuroSys Conference*, Bern, Switzerland, April 2012. 32

[PJHM12] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys)*, Low Wood Bay, Lake District, UK, June 2012. 43

[PS05] Joseph A Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005. 1

[RLC$^+$12] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M.K. Martin. Computational sprinting. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, New Orleans, LA, US, February 2012. 40

[RMMM03] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003. 12

[RWB09] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 302–313, Austin, TX, US, June 2009. 39

[SABR04] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 177–187, 2004. 14

[Sag06] Assim Sagahyroon. Power consumption in handheld computers. In *Proceedings of the International Symposium on Circuits and Systems*, pages 1721–1724, December 2006. 25

[SAMR03] Kiran Seth, Aravindh Anantaraman, Frank Mueller, and Eric Rotenberg. FAST: Frequency-aware static timing analysis. In *IEEE Real-Time Systems Symposium*, Cancun, MX, December 2003. 17

[SKK11] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive DVFS. In *International Green Computing Conference*, Orlando, FL, USA, July 2011. IEEE. 98, 103

[SLB07] Jan Stoess, Christian Lang, and Frank Bellosa. Energy management for hypervisor-based virtual machines. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, US, June 2007. 42

[SLSPH09] David Snowdon, Etienne Le Sueur, Stefan Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In *Proceedings of the 4th EuroSys Conference*, pages 289–302, Nuremburg, DE, April 2009. 34, 44, 96, 97, 118

[SPH07] David Snowdon, Stefan Petters, and Gernot Heiser. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *International Conference on Embedded Software*, pages 84–93, Salzburg, AT, December 2007. ACM, Press. 31, 34

[SRH05] 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, September 2005. 31, 34

[SvdLPH07] David Snowdon, Godfrey van der Linden, Stefan Petters, and Gernot Heiser. Accurate run-time prediction of performance degradation under frequency scaling. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, pages 58–64, Pisa, IT, December 2007. 19, 31, 34

[u-b06] u-blox AG. *ATR0630 Data Sheet*, July 2006. GPS.G4-X-06009-P2. 62

[Vee84] Harry J. M. Veendrick. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *IEEE Journal Solid-State Circuits*, 19(4):468–473, 1984. 11

[Vol02] Karl R. Volk. Dealing with noise when powering RF sections in cellular handsets. http://www.eetimes.com/document.asp?doc_id=1277716, July 2002. [Online; accessed Jan 2013]. 23, 54

[WB02] 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 (CASES)*, Grenoble, FR, October 2002. 34

[WWDS94] 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 (OSDI)*, pages 13–23, Monterey, CA, US, November 1994. 33

[XZR$^+$05] Ruibin Xu, Dakai Zhu, Cosmin Rusu, Rami Melhem, and Daniel Mossé. Energy-efficient policies for embedded clusters. In *Conference on Language, Compiler and Tool Support for Embedded Systems (LCTES)*, Chicago, IL, US, June 2005. 36, 44, 97

[YKJ$^+$12] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, USA, June 2012. 32

[YWV$^+$05] Shengqi Yang, Wayne Wolf, Narayanan Vijaykrishnan, Yuan Xie, and Wenping Wang. Accurate stacking effect macro-modeling of leakage power in sub-100nm circuits. In *Proceedings of the 18th IEEE International Conference on VLSI Design*, Kolkata, IN, January 2005. 13

[ZELV02] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, US, October 2002. 42

[ZELV03] Hang Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: Unifying policies for resource management. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC)*, San Antonio, TX, US, June 2003. 42

[ZMM04] Dakai Zhu, Rami Melhem, and Daniel Mossé. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer Aided Design*, pages 35–40, San Jose, CA, US, 2004. 97