Optimising Energy Management of Mobile Computing Devices

M.J. Johnson and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: { m.j.johnson, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

February 2012

ABSTRACT
Mobile computing devices are becoming ubiquitous and the applications they run are demanding greater processing and storage capabilities. Managing the power consumption and battery life of these devices is increasingly difficult but some careful choices made in the software architecture stack can optimise power utilisation while still maintaining needed services on the architecture. We describe the hardware blocks in a modern mobile device and measure their power requirements. We discuss some power management strategies and present results showing how some quite dramatic energy savings are possible on a typical modern mobile device running Android and Linux. We discuss the implications for future mobile computing device architectures.

KEY WORDS
device architecture; power consumption; battery life; mobile devices

1 Introduction

Modern mobile hardware is extremely complex and a mobile device will typically have more peripherals than a standard desktop PC. The hardware supported by a mobile device usually includes the following: Applications CPU, Baseband CPU, LCD Panel and controller, RAM, NAND flash Memory, MMC flash memory, USB Controller, Audio subsystem, GPS, Video Encoder/Decoder, Serial I/O, Bluetooth, Multiple DSPs, GPU, 2D Graphics Controller, Touchscreen, Cameras, Flashlight, LEDs, Battery Monitor, Wifi. All this hardware is typically powered by a 3.7V battery with a capacity of around 1500mAH. It is vital that the power used by the device is managed efficiently to increase the time that the device is usable [3].

Architects of desktop CPUs have long been aware of power consumption and efforts to produce more power efficient machines have led to the use of metrics such as FLOPS/watt and MIPS/watt. However, mobile devices have very different requirements and these metrics are not as useful. In 2 we look at the hardware present in a mobile device and discuss it's power management.

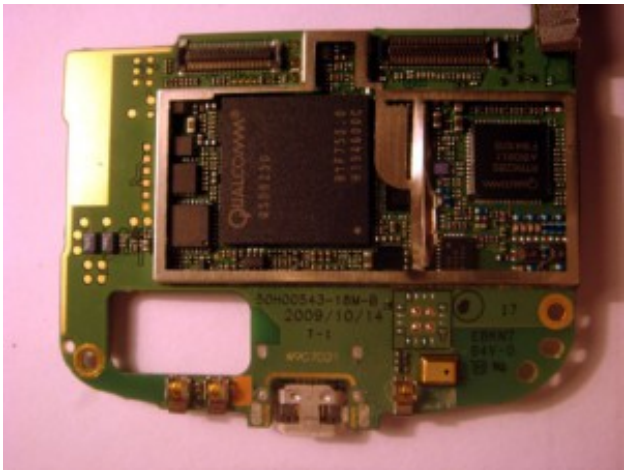

Figure 1: QSD8250 CPU

Currently, mobile devices make up about fifty percent of all personal computing platforms. This share has increased from almost zero in the last 5 years and mobile devices are becoming ever more pervasive. While it is not vital that a mobile device lasts for weeks on a single battery charge, it is important that it can last a complete day even with heavy use because many users charge their phones overnight every day [1].

Power is measured in Watts, however at a fixed voltage this is proportional to the current in Amps, mobile device battery capacity is always specified in mAH (milliamp hours) so throughout this paper we will us mA as a measure of power consumption. While it is possible to save power by careful design of applications and protocols [8], [9] this paper will concentrate on how the Operating System can be used to manage hardware for best power management.

Future mobile devices will need to use even more power than current devices if they are to support higher speed networking, advanced graphics and distributed resource sharing [2]. Figure 1 shows layout of the main logic board on a typical modern mobile device. Most of the hardware discussed in this paper is integrated into the large ASIC on the left.

2 Hardware Architecture

The hardware in a mobile device consists of a System on a Chip (SoC), memory and a number of peripheral devices. The SoC is an ASIC composed of a number of hardware blocks. The hardware in a modern mobile device will usually include the following blocks.

Application processor: This CPU runs all applications and has access to most hardware, it may be mutlicore. Currently most application processor CPUs are ARM Cortex A8 or A9 designs. The ARM CPU was designed as a low power device and it supports various low power modes which will be discussed in Section 3.3.

Baseband Processor: This processor runs the radio software and controls the radio frequency interface, The baseband processor is either an ARM CPU and DSP or a standalone DSP. Some devices have the baseband integrated in the SoC, others use a separate ASIC. The baseband processor usually runs a proprietary real-time operating system and so it's power management techniques are not available for analysis, however we expect them to be very similar to those used by the application processor.

I2C: A simple 4 wire bus used for controlling slow peripherals such as the touch screen, battery driver and camera CCD. The bus itself has no power management capabilities, power down or sleep commands must be sent to the individual peripherals on the bus.

GPIO: General purpose input/output pins, used for controlling hardware. Each GPIO may have up many functions for controlling internal operation of the SoC and may be multiplexed to SoC I/O pins. There may be many hundreds of GPIOs and each must be configured to draw as little power as possible. A common cause of excess power drain is a misconfigured GPIO.

Display interface: A serial bus used to connect the display panel, common interfaces are MDDI (Mobile Display Digital Interface) and DSI (Display Serial Interface). These buses do include power management capabilities and send messages to the LCD controller about idle states.

Audio: Control of the speaker, handset audio (earcoupled speaker and mic) and headset audio. The audio subsystem can use significant power when playing audio through an amplified speaker.

GPS: Global positioning system, uses a DSP to decode GPS satellite signals for location determination. For fast fix times the mobile network needs to be used to download almanac data. Although the GPS can use significant power it is not always enabled and there are techniques that can be used to minimises it's use [7] [12].

USB: A USB controller in device or host/device mode. The 5V usb power is often used to charge the device so power use is not a issue when the USB port is connected.

SD: Secure digital controller for memory cards, also used for the wifi interface on some devices (SDIO). SD devices use minimal power and may be disconnected and powered down when not is use. Recent Linux kernels have a SD Abstraction Layer (SDAL) which can be used to power down the card when it is idle.

NAND: Flash memory used for storage of the Operating System and data. Recent devices may also have eMMC flash storage. These devices consume very little power.

VFE: Video Front End, used to transfer data from a high resolution camera.

UART: Serial bus connected to the SIM card and Bluetooth controller.

2D graphics engine: Used for block transfers of image data in various formats. May perform scaling and rotation of bitmaps. The 2D engine will often use much less power than the GPU and so unless the 3D engine is needed, graphics composition should use the 2D engine.

DMA: Direct Memory Access, most blocks can transfer data to/from memory without any CPU intervention. This saves power because it allows the CPU to enter a low power state more often.

ADSP: Application Digital Signal Processor, used for processing audio and video data, offload of compute intensive function to the DSP can save power [4].

GPU: Graphics Processing Unit, a 3D graphics engine supporting OpenGL-ES 2.0. The GPU can use significant power, thus the use of user interface features such as complex animated wallpapers should be minimised for best power savings.

MDSP: Modem Digital Signal Processor, used for processing radio frequency signals, this is controlled by the baseband processor.

RAM: Memory external to the SoC, usually Low Power DDR2.

Display: an LCD or LED panel with a resolution of up to 1080x720 pixels. LCD panels use slightly more power than LED panels, modern designs are becoming very power efficient. Until recently, the display was the most power hungry component, this is no longer the case as we show in Section 3.1.

Light sensor: A sensor capable of detecting the ambient light level. This sensor is polled once a second when the display is on and is used to control the backlight for the LCD panel. It uses minimal power and allows the LCD panel to use less power when ambient light is low.

Proximity Sensor: A sensor capable of determinimg if the device is close to another object, this sensor usually uses an Infra-Red emitter and light sensor to detect any reflected light. It uses about 5mA during very short flashes and is only enabled when necessary. It is used to turn off the display when it is placed close to the head during a call. This saves power and avoids the touchscreen being pressed by mistake.

Cameras: A rear facing camera with up to 8Mpixels and capable of streaming HD video. The camera is initialised and focused by sending connads over an I2C bus. There may also be a lower resolution front-facing camera for video calls. Camera use significant power as discussed in Section 3.1.

Accelerometers: Sensors used to detect gravitational or magnetic fields and torque. These sensors do not need to be polled frequently and techniques are available to minimise their effect on power consumption [11].

Real Time Clock: A battery backed clock capable of maintaining the system data and time, it is capable of waking up the device at a set time and must run for long periods of time on minimal power.

Timers: Low and high resolution counters used for system timing. The counters themselves do not use much power but their use should be minimised because they may prevent the device from entering low power modes as discussed in Section 3.3.

Encryption processor: Used for efficient processing of encrypted communication channels. This is controlled by the baseband processor.

Video Processor: Capable of encoding and decoding various video formats including H264 and MPEG4. Software codecs should be avoided because the hardware encoders will use much less power and allow the CPU to idle.

Audio Processor: Capable of encoding and decoding audio formats such as MP3 and AMR. Similarly, these should be used instead of software.

Battery Controller: A device capable of determining the charge remaining in the battery and charging the battery when the device is connected to an external power source. The battery controller usually uses coloumb counting to keep a record of how much power has been used.

Bluetooth: Used for short range RF communications, Bluetooth is designed for low power devices and consequently does not significantly affect power consumption, although some applications may make heavy use of it [10].

Mobile Voice Connectivity: A mobile phone must be connected to the voice network when possible, in areas with good signal strength this can be power efficient, however where the signal strength is low or intermittent, techniques must be used to avoid significant power drain [6].

Mobile Data Connectivity: The mobile data network for a modern device is always connected and so application software must be carefully written to avoid frequent use of mobile data. If possible, synchronisation and other background events should be batched and performed together.

Wifi: Wifi was not designed for mobile devices and so it's power management is more problematic, however modern Wifi adapters use techniques such as Beacon and Idle Mode power saving and Traffic Coalescing [13] to limit the time spent transmitting data. Wifi will generally use less power than the mobile data network and so should be used in preference.

3 Software Architecture

We discuss the software architecture with reference to: an example device; clock control; teh Linux CPU clock control mechanism; and deep sleep issues.

3.1 Example Device

The device we worked with contained a Qualcomm QSD8250 SoC running the Android mobile operating system. Android is an Open Source OS running on top of a

Linux Kernel. For this reason it is easy to modify the software to measure power consumption and see how changes to the software stack affect it. We used a device with a ds2482 battery controller, this is connected via the I2C interface to the SoC and via a proprietary serial interface to the battery.

In order to determine power usage of the device we could replace the battery with a power supply and measure the current drawn but in practice it is easier to use the current measurements provided by the battery controller. The standard software only polls the battery controller once every minute so we had to modify the kernel driver to provide a new interface to the hardware that allowed us to record the current draw at any time. We also modified the driver that controlled charging so that it could be disabled. This allowed us to connect to the device through the USB interface to measure power consumption.

Linux device drivers typically use read or write operations on special files in the /dev directory to communicate with userspace utilities. While we could have used this method, we decided to use a simpler interface, sysfs. Sysfs is a virtual filesystem mounted on /sys, it contains properties of device drivers and can be used to set parameters for a driver.

In Linux, device driver parameters were originally used to allow the kernel boot command line to set up a driver, for example to pass the IRQ number that it should use. Recent Linux kernels allow access to the driver parameters though a set of sysfs files:

```
/sys/module/{driver}/parameters/{param}
```

This interface is very easy to implement and can be also used to perform operations. A macro is used to add parameters to a driver:

```
static int enable_charge=1;
module_param(enable_charge,int,0644);
```

This creates a parameter called enable_charge initialised to 1.

```
module_param_call(battery,set_ch,
                  get_batt,NULL,0644);
```

This creates a parameter called battery, writing to it calls set_ch, reading calls the get_batt function. Modifications to the ds2784 battery driver are shown in Figure 3.1. Example usage of the modified battery driver is:

```
# cd /sys/module/ds2784_battery/parameters
# ls
battery          enable_charge
# echo 0 > enable_charge
# cat battery
```

Hardware	Power Usage
Display	50-80mA
Camera	250mA
GPS	80mA
GPU	90mA
CPU	70-210mA
Bluetooth	15mA*
Wifi	90mA*
3G Data	160mA*
Speaker	80mA
Voice Call	150mA

* only while transferring data

Table 1: Current used by various hardware blocks

```
V=4084696 I=-123950 Iav=-131253 C=1262400
```

Using the modified driver we obtained the power usage values shown in Table 3.1. Note that the camera uses the most power, however this is most likely because it is also heavily using the CPU, GPU and DSP. The average power drawn depends on how the device is being used. When reading text, the CPU will be idle and the device will probably use about 100mA, this gives 15 hours of use with a 1500mAH battery.

The CPU is a significant contribution to the power usage, especially when running at full speed.

Note that when the device is suspended only a very minimal set of systems are left powered on (the always-on subsystem) and we measured a power usage of 2-10mA depending on signal strength and mobile network type.

3.2 Clock Control

The device has a single clock derived from a temperature compensated crystal oscillator (TCXO). This runs at 19.2 MHz or 32KHz when the device is suspended. Other clocks are made using Phase Locked Loop based multipliers (PLLs) and fractional dividers (M/N:D counters) for example on our test device a CPU clock of 998MHz is generated by multiplying TCXO by 52. Four PLLS are available and dividers are used to generate all the other clocks in the system forming a clock tree. There are roughly 150 clocks. Each hardware block uses at least one clock and to save power these clocks must be disabled when not in use. The CPU clock can be modified to slow it down or speed it up, this gives 20 available frequencies between 245MHz and 1.1GHz.

Fiqure 3.2 shows how clock speed affects CPU power usage, as expected this is linear. CMOS transistors use power to switch on and off and so the faster the clock the more power is used. By extrapolating from the data points we can see that even if the clock is switched off, the CPU still

```
static int enable_charge=1;
module_param(enable_charge, int, 0644);

static int battery_adjust_charge_state(struct ds2784_device_info *di) {
....
if(!enable_charge) charge_mode = CHARGE_OFF;
....
}

static int set_charging(const char *val, struct kernel_param *kp) {
        battery_adjust_charge_state(the_di);
        return 0;
}

int get_battery(char *buffer, struct kernel_param *kp) {
        struct battery_status *s=&(the_di->status);
        ds2784_battery_read_status(the_di);
        return sprintf(buffer,"V=%d I=%d Iav=%d C=%d",s->voltage_uV,
                s->current_uA, s->current_avg_uA, s->charge_uAh);
}

module_param_call(battery, set_charging, get_battery, NULL, 0644);
```

Figure 2: Modifications to the ds82482 Battery Driver

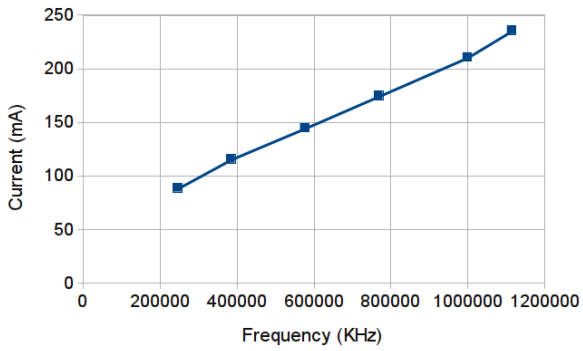

Figure 3: CPU Power usage

draws about 50mA.

3.3 Linux CPU Clock Control

The Linux mechanism for performing CPU clock control is called Cpufreq. This changes the CPU frequency based on load. The CPU clock driver tells the kernel which frequencies are available and provides functions to change frequency. A Cpufreq governor algorithm changes frequency by using various OS load metrics and heuristics. A number of governors are available, the most widely used being the 'ondemand' governor. Cpufreq uses a sysfs interface via file is /sys/devices/system/cpu/cpu0/cpufreq/

Although the CPU clock can not be set to 0 by manipulating the clock hardware, the ARM CPU has a special instruction that can be used for this purpose. The SWFI instruction (Suspend and Wait For Interrupt) disables the CPU clock and enables it again when an interrupt occurs. The Linux idle loop, which runs when no process needs the CPU, executes this instruction. Thus, when the CPU is idle it draws the minimum power in Figure 3.2 (about 50mA). Recent Linux kernels can be 'tickless' this means the kernel doesn't have a periodic timer interrupt so can sleep for long periods of time. Timers can also be 'deferrable' so a non urgent timer interrupt will wait until CPU is not idle. This means that in practice Cpufreq has little effect on power consumption, in fact it can hurt power consumption by preventing it from spending much time idle. The strategy of running as fast as possible and then idling for as long as possible is called 'race to idle'. Whether clock scaling or race to idle is more power efficient depends on a number of factors such as how long it takes to put CPU into an idle state and how memory bound a task is (memory bound tasks are not affected by CPU frequency).

3.4 Deep Sleep

When SWFI is executed, all clocks e.g. timers and bus clocks are still running, it is possible to stop these clocks and save even more power. If this happens then an interrupt can no longer wake the CPU, the only way the CPU can be woken is if the Baseband processor is used to wake it up. This mechanism requires careful cooperation between the application CPU and the baseband CPU.

When we enabled this idle sleep mode the power consump-

tion dropped from 50mA to 30mA.

Even with all the clocks disabled there is still some CPU power usage caused by leakage current. CMOS transistors use power even if they are not clocked and this leakage becomes much worse as the FET channel length is reduced so it is increasingly affecting modern devices.

The solution to minimising this leakage current is to use a global distributed footswitch (or headswitch). This technique involves adding FETs to the power rails of the device (on the V_{dd} rail it is a headswitch, on V_{ss} it is a footswitch) so power can be removed completely. This idle mode is called Deep Sleep or Power Collapse. Once again the baseband processor must wake the application processor but now the application processor must be completely reinitialised.

When we enabled this deep sleep mode the power consumption dropped to 14mA.

The disadvantage of the deeper sleep modes is that they can take a significant time to go to sleep and wake up. Thus, they should only be used when an interrupt is not expected shortly. For asynchronous interrupts, it is not possible to know if one about to happen, however the most common interrupt is a timer interrupt and the OS can determine when this will next happen. Thus, the idle code can look at when the next timer interrupt is due and choose an appropriate sleep mechanism. There are also other constraints on sleep modes, for example on the device we were using, GPU interrupts could not be used to wake from either of the deep sleep modes, this may explain why the measured power consumption of the GPU is relatively high.

4 Voltage Control

The power used by a CMOS device is proportional to the clock frequency, but it is also proportional to the square of the voltage. This means that lowering the CPU voltage should save more power than lowering the frequency. For a given frequency there is a minimum V_{dd} at which the CPU will function, this voltage also depends on temperature because the gate delay of a CMOS transistor increases with temperature.

4.1 Static Voltage Scaling

Static Voltage Scaling (SVS) is the technique of using a set CPU voltage for each possible CPU frequency. This voltage must be chosen such that the CPU will still function at all temperatures.

In order to implement SVS, an extra field is added to the CPU frequency table to specify a voltage for that frequency. It is important that the voltage is changed at the correct time, the voltage must be increased before increasing the frequency but decreased after decreasing the frequency. This ensures that the voltage is always at or above the required value.

4.2 Dynamic Voltage Scaling

The voltage used for SVS must be chosen so that it is safe for all devices and thus it has to be chosen very conservatively. It would be better if the device could provide feedback about the voltage. This is possible if some extra hardware is included, this is called Dynamic Voltage Scaling (DVS) [5].

The SoC we were using has hardware to support DVS, it uses ring oscillators and a number of delay circuits to model the longest possible delay through three different parts of the CPU, the Datapath, the Floating Point Unit and the Level 2 Cache. We can use these delay circuits to tell if V_{dd} is too high or too low. If any oscillator thinks V_{dd} is too low, it must be increased, if all think it is too high, it can be decreased. Since the delay depends on temperature, the die temperature must also be measured and taken into account. Although the device we were using supported DVS it was not implemented in the Linux kernel, we implemented a device driver for DVS.

Our driver uses a 2D table of voltages for frequencies and temperatures, it polls the hardware every 200ms until the voltage for the current temperature has stabilised. When the voltage for a particular temperature has stabilised, the table value is fixed so no more polling is necessary. Module parameters are used for status and to enable/disable DVS. When a voltage is set, the table is updated to clamp to this voltage for all lower frequencies and temperatures.

Output from the status of the driver is shown below.

```
#  cat status
Current TEMPR=9
Current Index=1
Current Vdd=925
Vdd Table:
 0:   19200 925 0
 1: 245760 925 0
 2: 384000 925 0
 3: 576000 975 0
 4: 768000 1150 1
 5: 998400 1225 1
 6:1113600 1300 0
```

This shows that frequencies of 768MHz have stabilised with voltages of 1.15V and 1.225V respectively.

Figure 4 shows the affect of SVS and DVS on power consumption. With a fixed V_{dd}, this voltage must be chosen for the highest possible frequency, thus the device uses significantly more power at lower frequencies. The values for the SVS driver are provided by the Google kernel for this

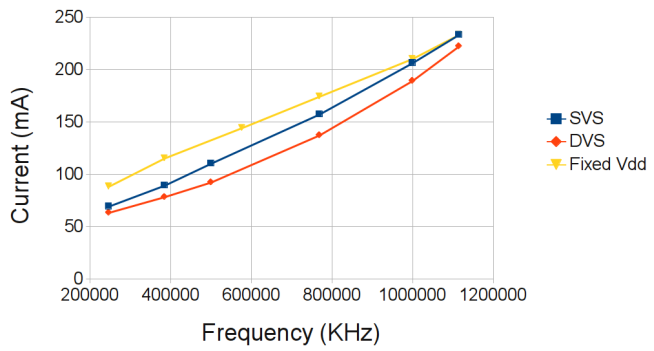

Figure 4: Voltage Scaling Results

device, it looks like the values have been determined by finding the best voltages for the highest and lowest frequencies and drawing a straight line between them. This means that DVS only provides significant power savings for frequencies in the middle of the table.

5 Conclusions

We have investigated various mechanisms for saving power on a mobile device including clock control, deep sleep and voltage control. In summary, we found that deep sleep (power collapse) provides the greatest power savings but it is not always available because there is significant overhead involved in putting the device into this mode. Changing clock frequencies can achieve significant power savings but only for tasks that are not CPU intensive.

Voltage scaling helps save power and has the potential to save more power than frequency scaling because of the quadratic relationship between voltage and power. Dynamic voltage scaling is a useful technique but if the voltages for static voltage scaling are chosen correctly is does not provide significant power savings.

In the future we would like to extend this work by investigating other devices and mechanisms for reducing power usage. Thes e ideas are likely to be of importance for tablet computers as well as for mobile phones.

References

[1] Ferreira, D., Day, A.K., Kostakos, V.: Understanding human-smartphone concerns: a study of battery life. In: Proc. 9th Int. Conf. on Pervasive Computing (Pervasive'11). pp. 19–33. No. 6696 in LNCS (2011)

[2] Furthmuller, J., Waldhorst, O.P.: Energy-aware sharing with mobile devices. In: Proc. Eighth Int. Conf. on Wireless On-Demand Network Systems and Services. pp. 52–59 (2011)

[3] Gordon, D., Sigg, S., Ding, Y., Beigl, M.: Using prediction to conserve energy in recognition on mobile devices. In: Proc. IEEE Int. Conf. on Pervasive Computing and Communications Workshops (PERCOM Workshops). pp. 364–367. Seattle, WA, USA. (21-25 March 2011)

[4] Hameed, R., Qader, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M.: Understanding sources of inefficiency in general-purpose chips. In: Proc. 37th Int. Symp. on Computer Architecture (2010)

[5] Hormann, L.B., Glatz, P.M., Steger, C., Weiss, R.: Evaluation of component-aware dynamic voltage scaling for mobile devices and wireless sensor networks. In: Proc. IEEE Int. Symp on World of Wireless, Mobile and Multimedia Networks (WoWMoM). pp. 1–9. Lucca, Italy (20-24 June 2011)

[6] IEEE: IEEE Draft Standard Methods for Measuring Transmission Performance of Analog and Digital Telephone Sets, Handsets, and Headsets - Amendment 1. IEEE P269a/D5.2, January 2012 pp. 1 –33 (19 2012), 6135531

[7] Kjaergaard, M.B., Bhattacharya, S., Blunck, H., Nurmi, P.: Energy-efficient trajectory tracking for mobile devices. In: Proc. 9th Int. Conf. on Mobile Systems, Applications and Services (MobiSys'11). Bethesda, MD, USA (28 June - 1 July 2011)

[8] Liu, Y., Guo, L., Li, F., Chen, S.: An empirical evaluation of battery power consumption for streaming data transmission to mobile devices. In: Proc. ACM Multimedia Conference (MM'11). Scottsdale, AZ, USA (28 November - 1 December 2011)

[9] Meng, L., Shiu, D., Yeh, P., Chen, K., Lo, H.: Low power consumption solutions for mobile instant messaging. IEEE Trans. Mobile Computing PP99, Online, 1–18 (June 2011)

[10] Park, U., Heidemann, J.: Data muling with mobile phones for sensornets. In: Proc. 9th ACM Conf. on Embedded Networked Sensor Systems (2011)

[11] Priyantha, B., Lymberopoulos, D., Liu, J.: Littlerock: Enabling energy-efficient continuous sensing on mobile phones. Pervasive Computing April-June, 12–15 (2011)

[12] Thiagarajan, A., Ravinranath, L., Balakrishnan, H., Madden, S., Girod, L.: Accurate, low-energy trajectory mapping for mobile devices. In: Proc. 8th USENIX conference on Networked systems design and implementation (NSDI'11). Usenix Association, Berkeley, Boston, MA, USA (30 March - 1 April 2011)

[13] Wang, R., Tsai, J., Maciocco, C., Tai, T.Y.C., Wu, J.: Reducing power consumption for mobile platforms via adaptive traffic coalescing. IEEE Journal on Selected Areas in Communications 29(8), 1618–1629 (September 2011)