FORUM NOKIA

Optimizing Mobile Software with Built-in Power Profiling

Version 1.0; May 19, 2009

Power Management

NOKIA

Contents of this document is a reprint from a book chapter in:

Mobile Phone Programming and its Application to Wireless Networking

Fitzek, Frank H.P.; Reichert, Frank (Eds.)

2007, XXIV, 473 p. With DVD., Hardcover,

ISBN: 978-1-4020-5968-1

http://www.springer.com/engineering/signals/book/978-1-4020-5968-1

This reprint has been published with kind permission from Springer. http://www.springer.com/

Contents

Change history

May 19, 2009	Version 1.0	Initial document release on Forum Nokia. Originally published as a chapter of the book *Mobile Phone Programming and its Application to Wireless Networking* in June 2007.

Optimizing Mobile Software with Built-in Power Profiling

Everybody can make the battery last longer

Gerard Bosch Creus and Mika Kuulusa

1 Summary

Software development for battery-powered mobile devices is tricky business because they are constrained platforms. Mobile applications are typically developed using development kits and emulators and later tested on a real device. Generally, developers are mainly concerned about correct application behavior. Badly designed software can easily impact the standby and active use time of mobile phones, but power consumption profiling needs a costly and cumbersome measurement setup. In this chapter we describe S60 software profiling tools that allow every developer to measure power consumption without any external equipment. Measurement analysis is carried out either on the mobile device or a PC. In addition, we present a set of guidelines and good practices that energy-conscious developers should follow to maximize application use time.

1.1 Motivation

The exploding number of features is rapidly adding to the amount of processing power and related hardware needed for their implementation. Consumers desire more performance, more impressive multimedia, faster data connections, and better usability. As a result, devices are getting more power-hungry to the point where power consumption and thermal issues become seriously limiting factors.

Power management is required because mobile phones are battery-operated devices and run on a limited power supply. Additionally phones are becoming smaller in physical size which can make excessive power consumption heat them up more easily. Battery technology improves at a steady rate, but is not able to keep pace with the continuous upscaling of processing performance and resource usage. Current battery technology cannot offer the energy densities required to make the power consumption problem disappear. Given that battery technology does not seem to provide the necessary improvements regarding energy and power management, the next solution is to try improving the phone platforms so that the desired features can be implemented at a much lower cost in energy consumption.

Possible solutions can be addressed using both hardware and software approaches. The hardware approach is often emphasized since hardware is the part physically draining energy from the battery. From this viewpoint, it makes sense to focus on hardware optimizations. However, since the only mission of hardware is actually to fulfill software needs, one can argue that software is the ultimate consumer of energy and therefore the focus should be on software optimizations [6].

Extensive research has been produced on both hardware and software sides. The approaches should not be considered mutually exclusive, but rather synergistic in nature. Hardware should ideally provide an optimal trade-off between energy and other non-functional attributes such as performance. On the other side, software should strive to use those hardware pathways offering the optimal trade-offs for the application at hand. Most of the software is constructed with the help of supporting software development tools like compilers that may prioritize attributes such as speed or memory footprint at the expense of energy efficiency [9]. However there seems to be a growing understanding that the application code itself holds the solution to the energy consumption problem [2,3,8,10].

While hardware optimizations are the domain of hardware vendors and platform integrators, a significant value offered to customers comes from the software integrated into the devices. The software comes from a variety of providers: smartphone manufacturers, open-source contributors, tool vendors, and other third-parties. Together, these individuals represent a vast community with high impact on the energy expenditure of the software they create.

Knowledge of the starting point is necessary in any optimization process before deciding on the actions necessary to reach the desired objectives. Once this knowledge exists, the optimization process must proceed until some initial requirements are met. However, an average software developer knows very little about the energy consumption of the developed applications. Whereas an electrical engineer smoothly works with current and voltage numbers, these may be hard to grasp for a software engineer more familiar with measures such as CPU load, bandwidth, or memory footprint. Software engineers would prefer to have support for power consumption measurements in their advanced software development tools. Occasionally, neglecting application energy consumption from the software design process can lead to unacceptable situations. For example, imagine a game having the greatest 3D graphics and effects being soon ready to ship to the shops. During testing it is observed that the game play time is less than 1 hour. Consumers may consider this as a fairly unacceptable trade-off for a battery-operated device.

Without the necessary knowledge about the most power-hungry parts of an application, or so-called power hogs, trying to solve the power problem can feel rather difficult. Earlier research recognizes that one of the main areas for action in order to improve energy efficiency are measurement tools and methods to evaluate power consumption [1]. Naturally, before creating any solutions, one needs first to see the problem. This leads to the conclusion that some kind of power consumption profiler is needed for mobile software development.

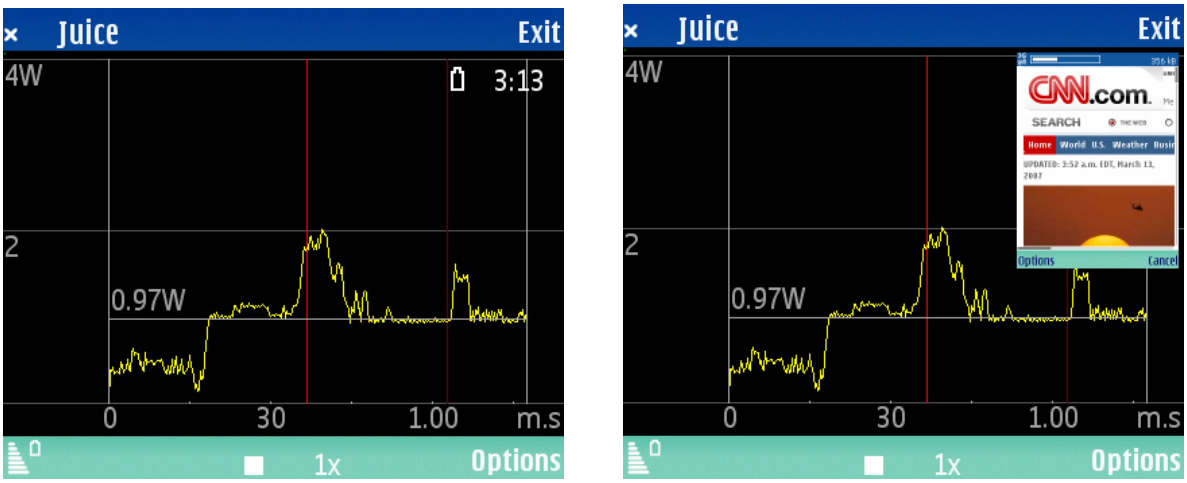
1.2 S60 Power Profiling Application

We have created a power consumption profiling application to assist developers in creating energy-efficient applications and squeeze more use time from the battery. Juice, the power consumption profiler, is an application for S60 3rd Edition smartphones. Its main objective is to graphically show the total power consumption of the device on which it is executing. In this sense, Juice is very similar to a digital oscilloscope displaying voltage, but since it is a computer program it can have much more advanced features. The rationale behind the Juice concept is that if developers can pinpoint the problematic parts of their applications, they may be able to take corrective action. Detecting basic mistakes early in the development process can bring quick benefits without large programming efforts. A key advantage is that it works on the latest Nokia S60 phones and no additional hardware is needed. Juice exploits existing hardware that is responsible for measuring the remaining energy levels for the battery level bars on phone displays. Typically, most battery-operated devices have this kind of hardware although they may have varying degrees of achievable measurement speed and accuracy. Nokia S60 smartphones support power consumption measurements at up to 4Hz frequency.

According to our analysis, the built-in measurements are quite accurate. We used a calibrated digital multimeter as the external reference and found that the Juice built-in measurements are comparable to those obtained with the multimeter. Built-in measurements may miss some short-lived power spikes, but from a long-term energy consumption perspective the measurements remain very accurate with respect to the external reference.

Power consumption measurements are implemented by observing battery current and voltage values. Battery current values are truly average values because the hardware integrates current consumption in the analog domain over the entire measurement period. An important characteristic for any kind of instrumentation is that it should introduce minimal disturbance to the actual measurements. This characteristic must apply particularly for an energy profiling application. In other words, Juice processing and sleep periods have to be heavily optimized for low power operation. This is important because any kind of processing adds to the total energy consumption, so Juice must be as lightweight as possible.

Graphical power measurement analysis on a small phone display must be very easy use. This requires careful user interface (UI) design on a device with restricted display and input capabilities. Especially measurement analysis is something that most software developers would like to perform on a desktop PC. Still, having the measurement analysis available on-device adds to what makes Juice attractive for developers. There is no need to transfer files between phone and PC, making quick analysis and testing very fast. Figure 1.1 presents the user interface for Juice.



(a) Juice main view.

(b) Displaying a captured screen-shot.

Figure 1.1: Juice showing the power consumption of web browsing over a 3G data connection. Average power is 0.97 W during the measurement, lasting 1 minute and 18 seconds. Repeating the use case forever would last 3 hours and 13 minutes for a fully-charged 850 mAh battery, as shown in 1(a). In 1(b), the red vertical lines indicate that a screenshot was captured.

The main mode of operation for Juice is detailed in the list below. Figure 1.2 illustrates the same process in a graphical fashion.

1. Set the desired trace settings through the Options → Settings dialog. Parameters such as the measurement speed and the way battery capacity is calculated can be adjusted. For example, a developer may be interested only in current consumption while voltage is not relevant for the case at hand.
2. Start the measurement. Juice starts receiving power consumption values from hardware.
3. Send Juice to the background to profile the desired use case. The user can then freely experiment with S60 applications.
4. Start the developed application and execute through the use case. This could be previously identified problematic use cases or simply the use cases present in the test plan. Take screenshots manually if needed.
5. Bring Juice to the foreground.
6. Stop the measurement.
7. Analyze the power consumption profile and identify problematic areas. Juice has some features to assist developers in identifying these spots.

(a) Setting the trace parameters

(b) Starting the trace

(c) Executing the use case

(d) Analyzing the power traces

Figure 1.2: Juice measurements on Nokia N95. The Maps application acquires the GPS location and the user navigates to a location using the route search functionality. The average power consumption was 0.97W and this use case could be repeated for 3 hours and 37 minutes with a full battery.

One question remaining is how developers can use power traces to optimize their applications. The main problem consists in relating the instantaneous power consumption to the profiled application. This power could very well be consumed by another application running concurrently or by some platform service. This is a reasonable concern and as with any other kind of test, the testing environment should be as free as possible of external influences such as concurrent applications or unnecessary services. However, being able to account for every Joule spent even at the application level is something quite far from the current smartphone platform capabilities. Even with the knowledge of the threads and functions executing on the processor, the load on other hardware components may very well be due to requests by other applications, threads, or functions. Nevertheless, any kind of information provided in relation with the energy expenditure can potentially assist developers in pinpointing problematic threads or functions.

Presenting all the profiled data on a small screen can very easily lead to a case of information overload. Instead, Juice uses a different method to bind the power traces to specific parts of the application: visual screenshots of the phone display. By analyzing the power consumption in real-time, Juice takes screenshots at those time instants in which there is a sudden peak or dip in the power consumption or when something significantly changes on the screen. Changes in the screen content may indicate some event-related processing. Additionally the user can manually take a screenshot through a specific key combination.

The purpose of screenshots is to allow developers to later browse through these screenshots in order to understand what was being done exactly at that time instant. As illustrated in Figure 1.3, the screenshots form a kind of visual history superimposed on the power graph. The capability to map application states to unusual power events can help to focus the developer's attention on a problematic construct or specific design decision. For example, a specific algorithm could be replaced with another with lower computational complexity or some buffering could be introduced to eliminate the problem. The solutions vary from application to application, we present some general guidelines for energy-efficient software later in this chapter.

A key feature in the Juice display is the time estimate (hours:minutes) that we refer to as battery time. This time describes how long a full phone battery would last for the measured use case. The calculation is based on nominal battery capacity (mAh) rating and real measured power consumption. For a developer, this number is very straightforward to understand since it reflects the design objective: the higher the number is, the longer use time your software has. It is interesting to note that battery time may open a whole new differentiation opportunity for S60 application developers. In a world where similar applications compete, better battery time performance can give application vendors a competitive edge over the rest. For example, two different web browser or e-mail client application offerings could have a similar set of features, but one allows for 2 hours more browsing or 5 days more push e-mail standby time. Battery life already plays a differentiating role in some embedded devices such as portable media players. Juice gives a new market opportunity for energy-conscious software developers.

Although the main purpose of Juice is to measure active application use cases, file storage space is the only limitation for the length of a measurement. Figure 1.4 presents an example of an extended measurement spanning 75 hours.

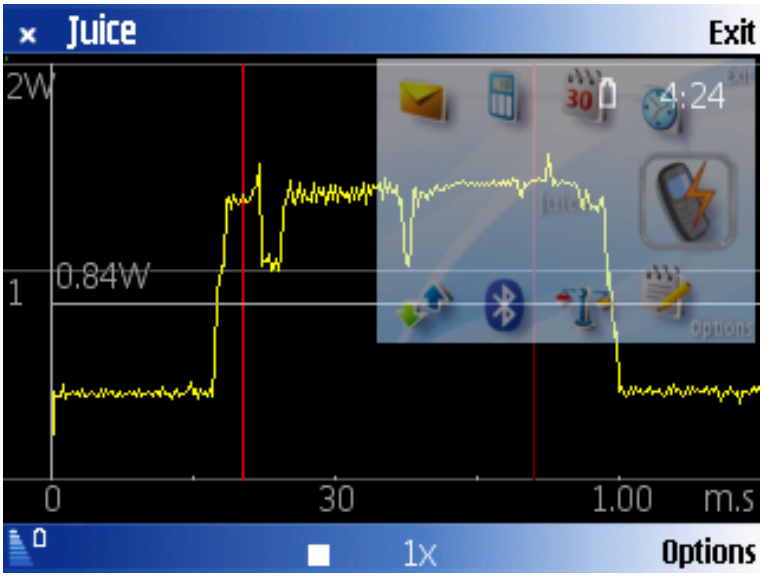
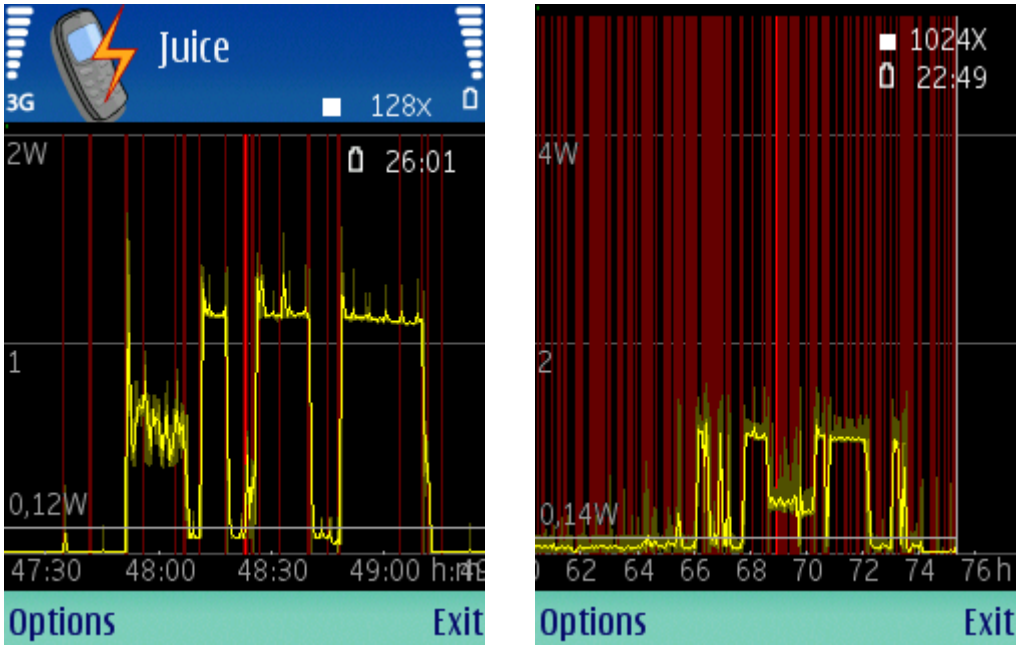


Figure 1.3: Juice graph viewing with screenshots. Visual history with screenshots helps to see what was done at a specific instant during application testing. Here, the user was browsing a 3D menu that uses hardware-accelerated graphics at the instant marked with a bright red vertical marker.



(a) Zoom-in on an active (b) Whole measurement period

Figure. 1.4: Juice graph examples. Figure 4(a) is showing one GSM and three 3G calls in downtown Helsinki. Power consumption is around 0.7W for GSM and 1.2W for 3G. Maximum zoom-out from a 75 hour measurement with over 1000 screenshots is shown on 4(b).

We believe that every platform should provide developers with support for power consumption analysis. Even though Juice is currently available for S60 devices only, we are planning support for other product offerings, such as Maemo. Figure 1.5 shows a concept drawing of MaemoJuice.

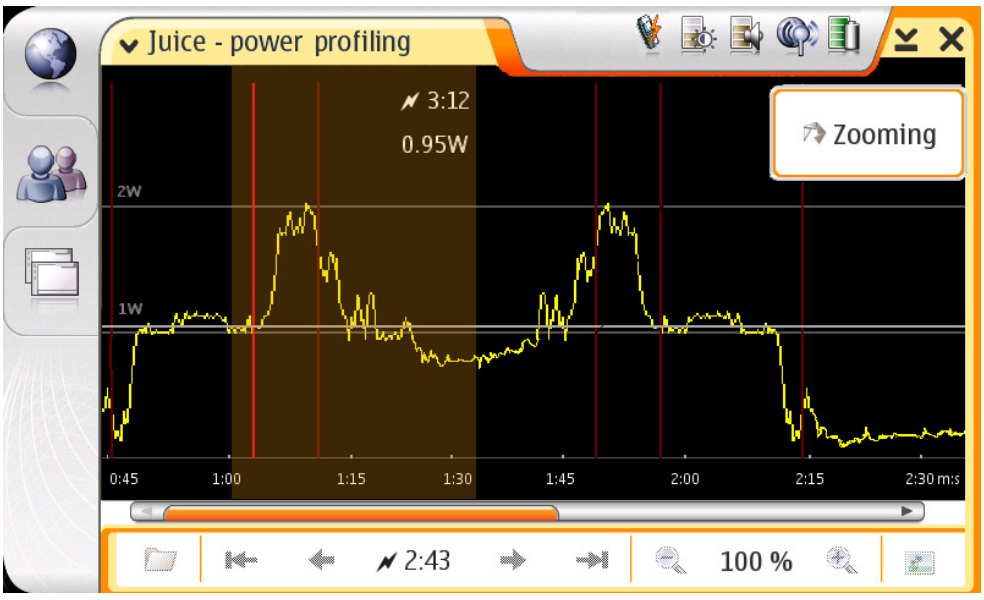


Figure 1.5: Concept drawing of Juice running on the Maemo platform. The bigger display area and pen input available in Internet tablets allow for an improved interface for power consumption analysis.

1.3 Carbide.c++ Power-Performance Profiling

A stand-alone S60 energy profiling application may become limited for more complicated applications or use cases where a deeper analysis is required on the development PC workstation. Many software developers are familiar with Carbide.c++, an integrated development environment (IDE) based on the open-source Eclipse IDE. Carbide.c++ contains a development tool called the Performance Investigator (PI). The PI integrates an on-device profiler together with a PC-based analyzer for S60 software performance optimization. In addition to CPU activity traces, the PI included in Carbide.c++ version 1.2 supports also power consumption measurements. The power measurement functionality in Carbide.c++ uses the same low-level interfaces as Juice and provides exactly the same measurement speeds at the same levels of accuracy. However, while Juice was designed to be a stand-alone application that developers can use on the device, the PI is aimed to be used in the IDE on the PC.

The PI is divided into two different components: an on-device profiler application and an analyzer plugin running on the PC. The profiler is responsible for generating the run-time trace files that are subsequently imported to the analyzer. Figure 1.6 shows the user interface for the profiler application. Typically the profiler spends most of the time in the background while the profiled activity takes place on the foreground. The PI offers developers a complete picture of the CPU load and Symbian OS threads together with the power consumption measured from the battery. This kind of side-by-side view can be of significant importance in isolating problematic binary executables, threads, or even function calls. Figure 1.7 shows the PI analyzer with performance and power consumption traces.

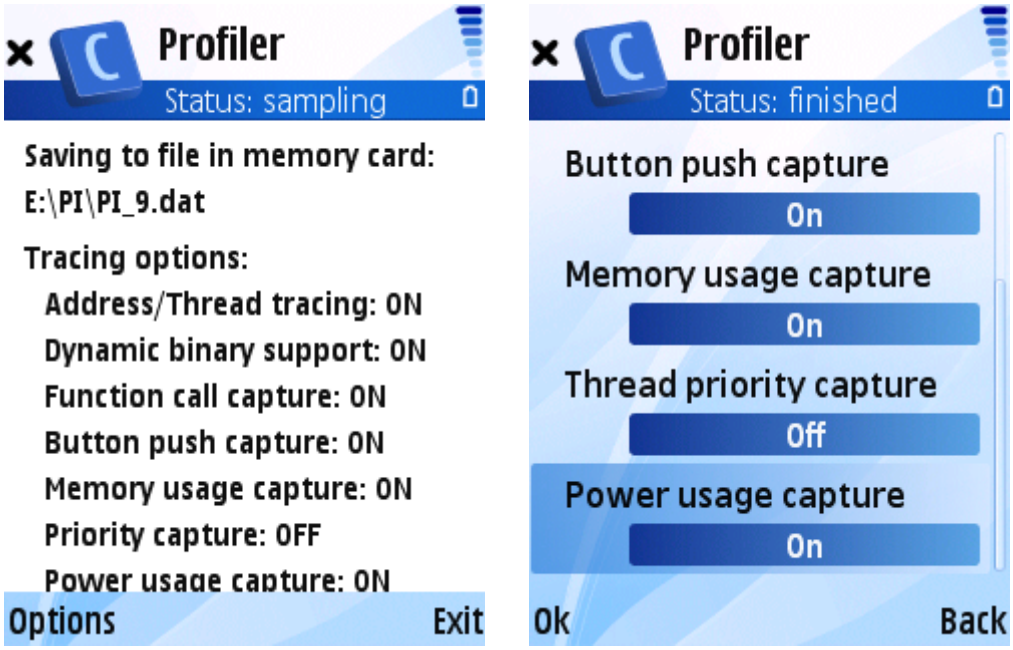


Figure 1.6: User interface for the Performance Investigator on-device profiler. The Options dialog allows for the setup of the parameters to trace, including power consumption.

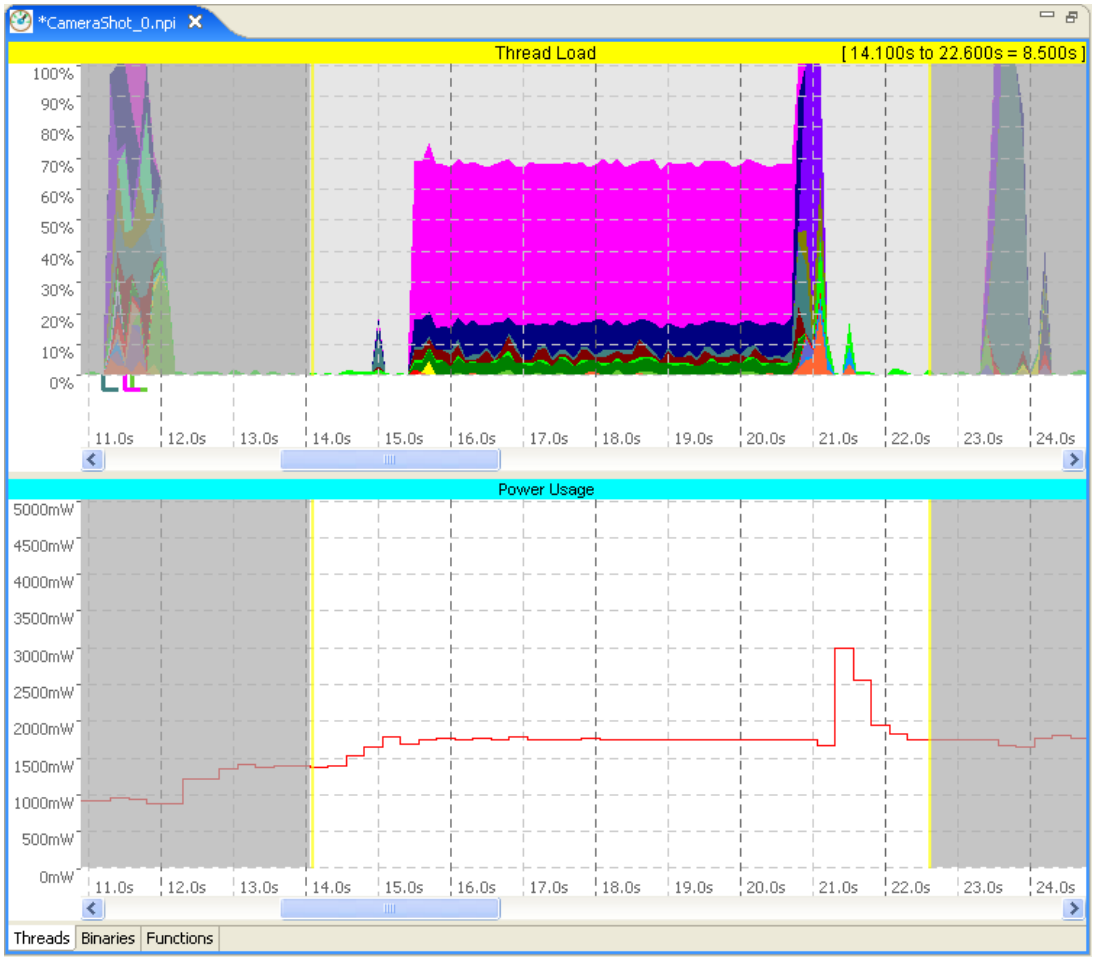


Figure 1.7: Power consumption analysis in the Carbide.c++ PI. The profiled use case involved the camera application, where 5s were spent viewfinding and then a picture was taken. The thread load on the CPU is shown on the top, with the power consumption graph on the bottom.

Setting up the necessary components for the PI power tracing requires a few steps, detailed in the following.

1. Enable the desired traces on the profiler. Power traces can be enabled from the Tracing options settings dialog. The sampling resolution can be set through the Power usage interval setting in the Advanced options dialog. The minimum sampling period of 250 ms corresponds to the maximum sampling speed achievable in current devices.
2. Start the profiler and send it to the background. This can achieved through the Application key, either by pressing it once and navigating to your application or keeping it pressed for about a second and selecting your application from the task list.
3. Execute the use case to profile.
4. Bring the profiler to the foreground (you can access it through the open tasks list) and stop tracing.
5. Transfer the trace files to the PC. The location of the files on the profiled device can be set up through the Output settings dialog in the PI Profiler.
6. Import the PI trace files to Carbide.c++ through the File → Import menu option.
7. Analyze the traces through the PI plugin on the PC. Try to determine if there are any execution constructs that trigger a particularly high power consumption.

The real advantage provided by the PI plugin is that it allows developers to see a concurrent view of the CPU activity and the power consumption at the device level. Still, many hardware components work asynchronously of the main CPU and thus it may be hard to find valid correspondences between the power consumption and specific program threads in cases where the CPU is not the dominant power consumer. In any case, the PI is a very useful tool in identifying recurrent instances of particular function calls or threads that seem to have a significant contribution to the power consumption, for example causing periodic power peaks over time. This may be due to the function activating a certain hardware component with a known power consumption pattern.

One interesting feature of power consumption analysis in conjunction with the CPU activity traces is that it can reveal problems at the system level. For example, the S60 web browser designers were investigating a slowdown when communicating with the 3G modem device driver. Power analysis revealed that the power consumption dropped dramatically after a certain function was executed. A quick look on the power levels confirmed that the 3G modem was erroneously disabled for some reason. A hidden bug in the driver interrupt handling routine was found and fixed. This programming error could have remained undetected for much longer had it not been uncovered through power profiling.

1.4 Energy-Efficient Design Guidelines

There are not many specific studies looking into the energy breakdown of mobile phone use cases. Such knowledge is important to understand which components draw most of the energy from the battery. Developers can then better figure out what are the actions to avoid. Figure 1.8 illustrates a power consumption breakdown in a mobile phone.

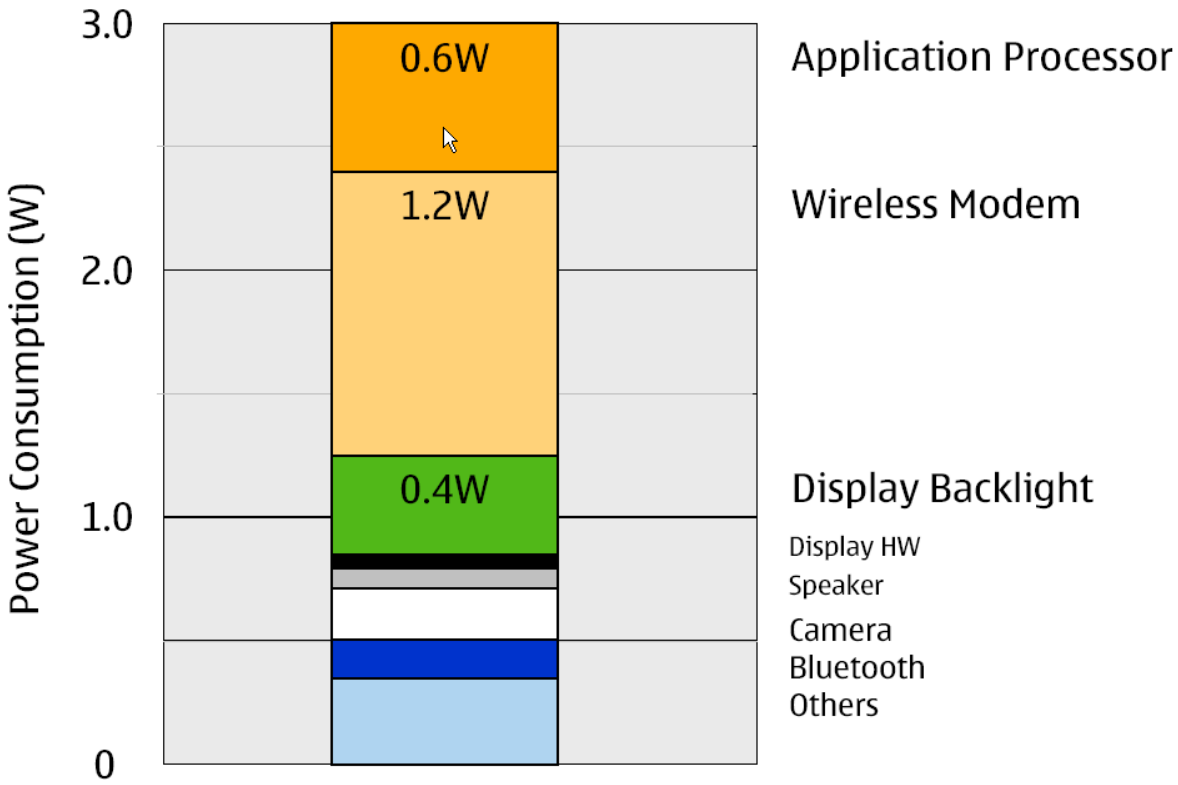


Figure 1.8: Power consumption for video conferencing in the Nokia 6630 [4, 7]. Constant average 3W can be tolerated without uncomfortable heating, but smaller products may have power budgets closer to 2W.

These figures are in agreement with previous findings on the power consumption of mobile computers [3, 5, 11, 12]. The major consumers are (in decreasing order): wireless modems (WLAN, 3G, GSM/EDGE), application processor and display backlight. An important share of the total power is consumed by a mix of components with marginal power consumption by themselves, such as the radio modem control processor and different integrated circuits. Knowing that three components may account for roughly 65% of the total energy consumption, it becomes clear that reducing the load on those components will have a noticeable effect on the overall energy consumption. In practice, taking energy considerations into account will result in a more responsible use of the available resources. For example, since radio interfaces tend to be rather power-hungry, they should be disabled whenever possible.

Most mobile platforms provide some sort of automatic power saving modes and for this reason applications should do their best to support them. Applications may consider implementing some kind of buffering to avoid frequent use of the communications bearer. Similarly, applications should avoid keeping unnecessary networking sockets open if they serve no purpose, allowing the bearer to automatically enter a reduced power mode. In the following, we present a set of guidelines to support the development of energy-efficient software. While these guidelines do not necessarily guarantee energy efficiency, not respecting them will most likely result in wasted energy and lower battery times. Likewise, in some cases there may be other factors preventing their implementation such as hard performance requirements. This is better left to the judgment of the smart software developer.

1. **Make the UI energy-aware**. User interfaces account for a significant share of the energy consumption in mobile computers. This is especially true since the speed gap between users and computers is considerable, and a fair amount of time is spent simply waiting for user input. Previous research looking into this topic has produced a set of recommendations to create more energy-efficient user interfaces [13], which we summarize below.

    - Accelerate user interaction. Make the frequently-used functionality easily accessible.
    - Minimize screen changes. Avoid unnecessary animations or progress indicators for quickly completed tasks. Buffer multiple consecutive changes before redrawing the screen, so that a single update is needed.
    - Avoid or minimize slow user input. Text input is slow on smartphones and should therefore be minimized when possible. Different kind of lists can be used as alternative, faster input mechanisms.
    - Reduce redundancy. Replace progress bars with busy indicators when appropriate and avoid features slowing down the user interaction.
    - Speculate on the user input. Use text auto-completion, last entries and pre-compute data for the most used functionality. Even though some of your predictions are going to be wrong, it's better than wasting energy simply waiting for the user input.

2. **Take advantage of advanced platform features**. Exploit the platform features allowing your application to hint or even directly control some platform resources. For example, S60 offers the Lights API to directly set the state of the display backlight.
3. **Synchronize periodic actions**. Instead of distributing periodic tasks in time, try to perform them in a batch run. This reduces the amount of timers and lengthens the idle periods which enables staying longer in sleep modes.
4. **Consider energy-performance trade-offs**. Application design presents many alternatives with different energy-performance ratios. Compare different implementation alternatives and select the most energy-efficient one that fulfills the desired performance needs. This applies to the whole development process, from the choice of architectural patterns to the selection of a particular algorithm or UI widget.

5. **Optimize the server side**. Applications using wireless data transfers are typically the most energy consuming. Applications typically connect to an Internet server using specific protocols and exchange application-specific data to and from the mobile phone. Consider optimizing the traffic pattern or compressing the data to a format that is lightweight to process on the device. Try to minimize the amount of data that needs to be transmitted.
6. **Experiment with different devices**. Mobile phones differ in features and hardware components. For example, the main CPU may have a different performance or even be a different model, such as ARM9, ARM11, multi-core ARM11 (MPCore), or Cortex-A8. Some devices may have higher speed data connectivity and WLAN to complement the cellular modems. Also display sizes are different. An optimization that works on a high-end phone may not work that well on low-end one.

1.5 Conclusions

This chapter has covered the field of software optimization from an energy perspective. We have presented two different approaches for getting a global view on the power consumption of software requiring no additional tracing hardware than that already available on smartphones. In addition, we have given a set of guidelines and recommendations to develop energy-efficient software.

There seems to be no silver bullet to eliminate or even alleviate the energy limitations that mobile devices face nowadays. Even device manufacturers, with a relatively high degree of control over the hardware and software components integrated into their product offerings, cannot alone solve the energy problem. Smartphone platforms are designed to be open to third-party software that adds to the value provided by the product. In fact, a fair share of the active use time on a mobile phone may be spent on this externally provided add-on software. If such software is not developed taking energy considerations into account, the problem is only going to get worse as the energy demands of the available features increase.

Mobile phone application development already exhibits a considerable degree of complexity. In order to create energy-efficient software, developers need tools to support them in such a challenge. The Juice power consumption profiler and the power tracing functionality included in the Performance Investigator of Carbide.c++ aim to fulfill this need. These tools do not require a basic set of electrical engineering skills to operate. Instead, they provide embedded power consumption measurements so that developers can easily access the required data. Juice provides an easy-to use tool for basic power consumption analysis directly on the profiled device. For more detailed analysis, the Carbide.c++ PI plugin offers the possibility to link power consumption to specific software constructs running on the device.

The solution to the energy problem lies in the hands of every software developer contributing to enrich the mobile software ecosystem. Carefully designing software to be more energy-efficient can greatly enhance the battery life of mobile devices. Everybody can make the battery last longer.

2 List of Abbreviations and Symbols

Term or abbreviation	Meaning
3D	Three-dimensional
3G	Third generation
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
EDGE	Enhanced Data rates for GSM Evolution GSM Global System for Mobile communications
OS	Operating System
PC	Personal Computer PI Performance Investigator
UI	User Interface
WLAN	Wireless Local Area Network

3 References

[1] C.Ellis, A. Lebeck, and A. Vahdat. System support for energy management in mobile and embedded workloads: A white paper. Technical report, Duke University, Department of Computer Science, October 1999.

[2] C. Ellis. The case for higher level power management. In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS'99), March 1999.

[3] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In Proceedings of the Seventeenth Symposium on Operating System Principles (SOSP'99), December 1999.

[4] Mika Kuulusa. Multiprocessors in wireless multimedia terminals. In-depth presentations, Sixth International Forum on Application-Specific Multi-Processor SoC (MPSoC'06), August 2006. [Online] Available: http://www.mpsoc-forum.org/2006/slides/Kuulusa.pdf .

[5] J. Lorch. A complete picture of the energy consumption of a portable computer. Master's thesis, University of California at Berkeley, 1995.

[6] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-aware power reduction. In International Symposium on System Synthesis, pages 18–23. Stanford University, September 2000.

[7] Yrjö Neuvo. Cellular phones as embedded systems. In IEEE Solid-State Circuits Conference (ISSCC'04). Digest of Technical Papers, volume 1, pages 32–37, February 2004.

[8] B. Noble. System support for mobile, adaptive applications. IEEE Personal Communications, 7(1):44–49, February 2000.

[9] T. Simunic, L. Benini, and G. De Micheli. Energy-efficient design of battery-powered embedded systems. In Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'98), June 1998.

[10] T. K. Tan, A. Raghunathan, and N. Jha. Software architectural transformations: A new approach to low energy embedded software. In Proceedings of the Conference on Design Automation and Test in Europe (DATE'03), 2003.

[11] Sanjay Udani and Jonathan Smith. The power broker: Intelligent power management for mobile computers. Technical Report MS-CIS-96-12, Department of Computer Science, University of Pennsylvania, May 1996.

[12] Sanjay Udani and Jonathan Smith. Power management in mobile computing (a survey). Technical Report MS-CIS-98-26, Department of Computer Science, University of Pennsylvania, August 1996.

[13] 13. Lin Zhong and Niraj K. Jha. Graphical user interface energy characterization for handheld computers. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'03), October 2003.

Appendix A Useful Links

[1] Nokia Energy Profiler tool (SISX), available for download at http://www.forum.nokia.com/energyprofiler

[2] Nokia Energy Profiler QuickStart guide available at Forum Nokia

[3] Discuss and ask questions using the Nokia Energy Profiler sticky thread in the Forum Nokia developer discussion board http://discussion.forum.nokia.com

[4] Nokia Energy Profiler webinar recording available at http://forumnokia.emea.acrobat.com/p91819182/

[5] Nokia Energy Profiler webinar presentation slide set available at Forum Nokia

[6] S60 Platform: Effective Power and Resource Management available at Forum Nokia