國立交通大學

資訊學院　資訊學程

碩士論文

嵌入式裝置的原始碼層級耗電分析工具

Source-Level Energy Profiling Tool for Embedded Devices

研究生：張藝馨

指導教授：曹孝櫟　教授

中華民國一0一年一月

嵌入式裝置的原始碼層級耗電分析工具

Source-Level Energy Profiling Tool for Embedded Devices

研 究 生：張 藝 馨　　Student：Yi-Hsin Chang

指導教授：曹 孝 櫟　　Advisor：Shiao-Li Tsao

國 立 交 通 大 學

資訊學院 資訊學程

碩 士 論 文

A Thesis

Submitted to College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

January 2012

Hsinchu, Taiwan, Republic of China

中華民國 一0一 年 一 月

嵌入式裝置的原始碼層級耗電分析工具

研 究 生：張 藝 馨　　Student：Yi-Hsin Chang

指導教授：曹 孝 櫟　　Advisor：Shiao-Li Tsao

國 立 交 通 大 學 資訊學院 資訊學程碩士班

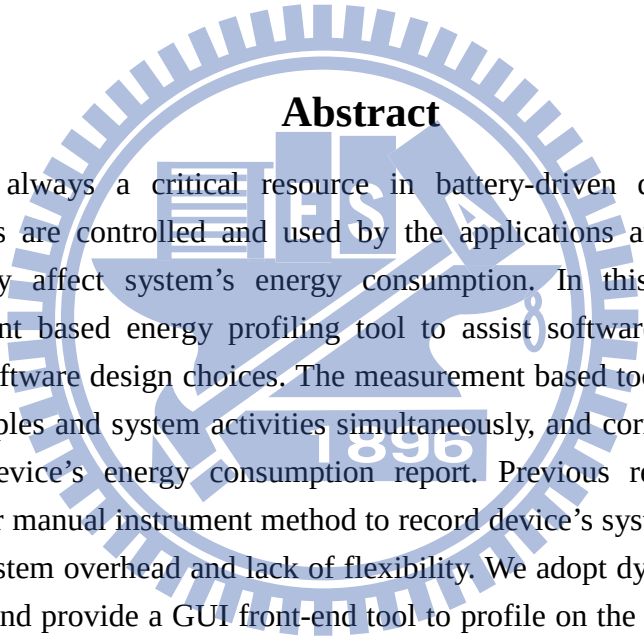
摘要

耗能對於行動裝置來說，一直是個很重要的議題。在有限的電池容量下，如何有效的利用硬體資源以達到省電及效能的要求，並完成工作目標，是軟體開發者重要的議題。在這篇論文中，我們開發及設計了一個耗能量測分析工具。這個工具可以提供軟體開發者在行程層級，函式層級及原始碼層級的耗能分析報告，藉以作為軟體開發者評估其所設計軟體品質的依據，並設計出符合省電及效能的軟體。耗能量測分析工具可以分為，取得行程及函式執行的時間、擷取系統耗能數據、及關聯系統執行時間和耗能數據並提供分析報告等三個部分。我們利用了動態的原始碼插入技術，改善了過往研究在取得行程及函數執行時間時的缺點，提供了一個具有彈性及容易使用的耗能分析工具。我們的工具包括了一個圖型使用者介面，讓程式開發者能動態的新增及刪除想要量測的函數，並且不需要重新編譯原始碼及重開機。實驗結果顯示，利用動態的原始碼插入技術，我們的耗能量測分析工具可以提供正確的行程層級，函式層級及原始碼層級的耗能分析報告。

Source-Level Energy Profiling Tool for Embedded Devices

Student: Yi-Hsin Chang Advisor: Dr. Shiao-Li Tsao

Degree Program of Computer Science
National Chiao Tung University



Abstract

Energy is always a critical resource in battery-driven devices. How hardware components are controlled and used by the applications and system software can significantly affect system's energy consumption. In this thesis, we propose a measurement based energy profiling tool to assist software designer to determine different software design choices. The measurement based tool records target device's power samples and system activities simultaneously, and correlating them together to generate device's energy consumption report. Previous researches use statistical sampling or manual instrument method to record device's system activities which may increase system overhead and lack of flexibility. We adopt dynamic source instrument technique and provide a GUI front-end tool to profile on the native (non-Java) part of Android system. The fine-grained process-level, function-level and code block-level performance and energy profiling reports are provided. The experiment results show the correctness of proposed profiling tool with low instrument overhead (0.02 milliseconds per function calls).

Acknowledgement

感謝指導教授曹孝櫟博士於論文及學業上的指導，使我在研究所生涯中，對於嵌入式系統及資訊科學的專業知識能有正確的觀念及深入的瞭解，遇到問題時能有獨力解決的能力。對於論文的悉心指導及建議，使我得以順利完成論文，在此致上最深的謝意。於口試期間承蒙梁文耀博士及賴槿峰博士對於學生論文的詳加審查，並提供許多寶貴的意見，在此致上由衷之感謝。

感謝建臻、宇宸、文信、小強及實驗室成員們提供許多寶貴的經驗及建議，在遇到問題時的討論及經驗交流，讓我能順利克服難關。最後，我要感謝我的太太美玲，在我離開職場，專心於論文的這段時間，默默的支持我，鼓勵我，讓我無後顧之憂，順利的完成我的心願。僅以本文，感謝所有支持我，關心我的家人及朋友們，祝大家都有個美滿及幸福的人生。

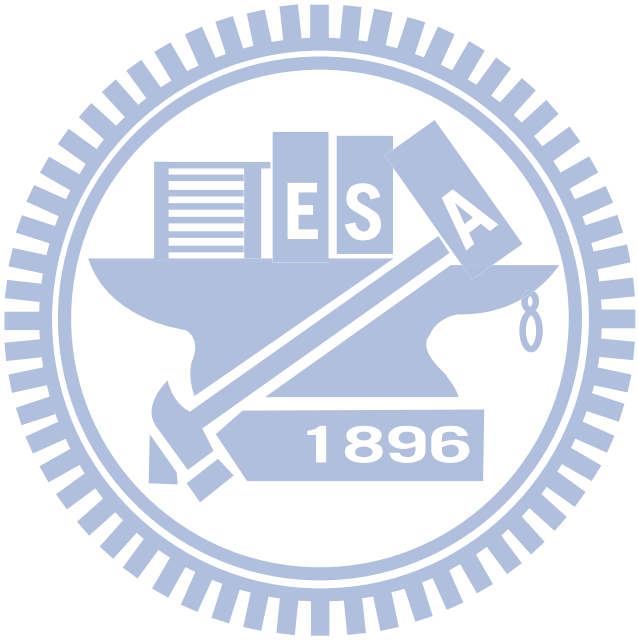
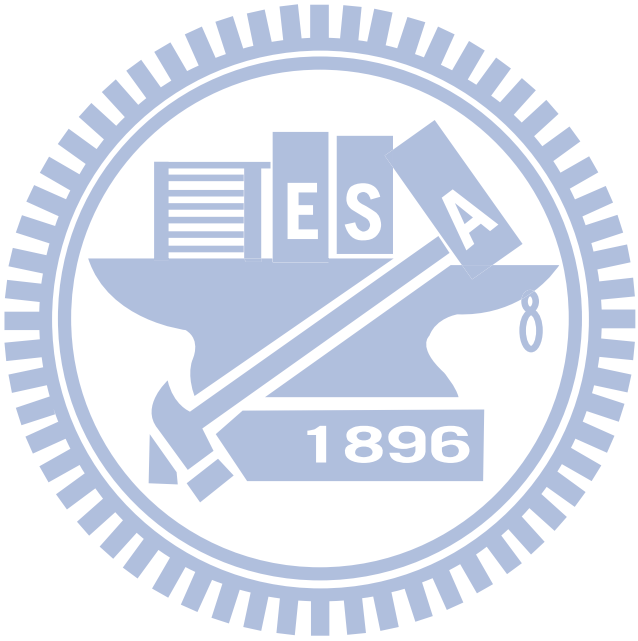


Table of Contents

List of Figures

List of Tables



Chapter 1

Introduction

Hand held devices including smart phones and Tablet PCs have become more and more popular in recent years. These devices provide better user experience to access the Internet. Several complex applications, which are used only on PCs, have been developed and run on hand held devices. Although there have been many improvements in low power hardware design and battery life, energy is still a critical resource for these battery-driven devices [1]. The less energy your device consumes, the longer it can be used without being recharged.

Previous research on energy consumption mainly focuses on hardware. However, energy consumption depends on not only the hardware but also how hardware is controlled and used by the software [5]. There are significant opportunities exist for system energy optimization at application and system software [6]. Operating-system and application developers need to estimate how the different scheduling algorithm and software design choices affects the system's energy consumption [6] [10]. An application designer may reduce mobile device's display quality when battery is low [29] and transmit/receive the E-mail when Wi-Fi transmission with higher data rate [31]. Using energy optimized library, the application can achieve the same functionality but consuming less energy [6]. All these design choices need a tool that can provide fine-grained energy profiling report. Software designer can analysis system's runtime energy consumption to identify energy-critical bottlenecks and determine the best design choice.

There are two commonly approaches for energy profiling tools, simulation and measurement [2] [30] [31]. Simulation and modeling based approaches can provide energy profiling information

in transistor level, architecture level, and instruction level. The lower level model can produce more accurate energy consumption report but consumes more time. In [12], the executable binary code is fed into cycle-accurate energy simulator to calculate the execute time and energy consumption of all system components including CPU, cache, bus, and memory, and provide function level energy consumption report to the software designer. Simulation and modeling based approaches are useful when hardware device is not available in early design stage, while measurement based approach can provide accurate energy consumption report by connecting external measurement device to data-acquisition card (DAQ) or multi-meter to target device's measurement points. Depending on the type and number of measurement points provided by the target device, power samples for different hardware components such as CPU, memory and I/O can be retrieved simultaneously. The power samples are correlated with system activities together to obtain accurate energy consumption report.

Today's embedded devices consist of several networking I/O components such as 3G, WLAN, and Bluetooth, these I/O components consume significant amount of energy compared with energy consumed by the CPU. The applications and system software running on these devices are getting more and more complicated and it is not a trivial task to analyze energy consumption of these devices. To ease the complexity of analyzing and reducing device's energy consumption, the profiling tool requires the abilities to determine how the energy is used by application and system software on CPU and I/O components. Fine-grained energy consumption report from process-level to code block-level assists system and software developers to understand the energy behavior of the system and optimize them for better performance and lower energy consumption.

In this thesis, we propose a direct measurement based energy profiling tool and profiles energy consumption of native parts of Android system running on a TI OMAP3/Beagle board platform

[35]. The profiling tool provides system's energy consumption report for both CPU and I/O components in process-level, function-level, and code block-level. Using a set of existing techniques, system's energy consumption can be correlated with the source code. The software designer does not require the static instrumentation or modification of source code in traditional way, making it possible for software designer to debug and optimize software repeatedly. A user friendly GUI fronted-end is provided to help software designer control and select interested functions and code blocks for specific application. The system activities and power samples are correlated together to generate fine-grained energy consumption report from process-level to code block-level. Software developer can see inside their application's energy characteristics and determine the effects of program code and flow on energy consumption in real time.

The rest of the thesis is organized as below: Section 2 reviews related work. Section 3 describes the design and implementation of energy profiling tool. Section 4 presents our profiling results. Section 5 presents conclusions and future work.

Chapter 2

Related Work

There are several similar energy profiling tools. As mentioned in the previous section, these tools can be categorized into two main groups: simulation and modeling based [12] [15] [17] [32], and measurement-based tools. Measurement-based tools can be further categorized into measurement-based estimation [13] [14] [16] and direct measurement based [1] [2] [4] [16] [31] tools.

For measurement-based estimation tools, the most widely used concept is to associate instructions running on the processor with their corresponding energy. Tiwari *et al.* [13] [14] propose to measure the base energy consumption of each instruction, and inter-instruction energy, and other energy consumption due to stall and cache misses. An improved measurement method is provided in [16] to measure instruction-level energy consumption in ARM7TDMI. The influences on energy-sensitive factors such as opcode, register number and fetch address are provided. These results are useful guideline for high-level energy reduction mechanism. The memory models are integrated to instruction-level simulator and a cycle-accurate simulation model is provided in [15]. The cycle-accurate simulation engine is then extended in [12] to correlate the source code to instructions, making it easier to evaluate energy efficiency of interested source code.

PowerScope [1] uses the digital multi-meter to trigger the profiling computer to collects program counter and process ID, while data collection computer records power samples simultaneously. The program counter, process ID, and power samples are correlated by energy analyzer together to generate procedure level energy profile. The overhead to obtain the system

activities may vary depending on the interval of external trigger generated by multi-meter. Software designer can't select interested application and focus on the procedures within specific application using PowerScope. Xian et al. [2] use a direct measurement based method and propose a time synchronization method to accurately assign the power samples to the system activities. The power consumption of CPU and several hardware components (I/O) are measured simultaneously and the component-wise energy-assignment method is proposed to raise the accuracy of software's energy consumption. The system activities are only in process level, making it difficult to debug process's energy consumption hotspots. PowerPack[4] is also a direct measurement based profiling tool. It provides a set of APIs for profiled applications to inform control thread that start or stop of interest code regions are hit, how to map power profile with source code. The application designer should insert these APIs to the profiled application manually, making it time consuming and can not apply to complicated applications easily.

Performance monitor techniques can be divided into two main categories: instrumentation and statistical sampling [23]. The Oprofile [34] is a statistical sampling profiling Tool for Linux systems, capable of profiling all running code at low overhead. The program counter and other runtime information are collected when certain interrupt event happened. PowerScope also adopts the same technique to collect system activities. Tracepoints [20], Kernel Probes (Kprobes) [22], Process Trace (Ptrace) and User-space Probes (Uprobes) [25] are instrumentation techniques. Tracepoints is hooking mechanisms providing static instrumentation in Linux kernel's critical locations and can be enabled at runtime (dynamically) with very small footprint when disabled. The Tracepoints is sufficient to record time critical system events such as schedule context switch and receive/transmit network packets. Kprobes is a simple, lightweight kernel instrumentation mechanism and can insert/remove probes into a running kernel dynamically. The performance information can be collected in user-defined

handler when a probe point is hit. While Kprobes is used for kernel instrumentation, Ptrace and Uprobes are provided for user-space applications instrumentation. Ptrace provides a set of system call for debugging user-space applications. The GNU Project Debugger (GDB) is the most widely used application debugger in Linux. GDB uses Ptrace system call to observe and control the execution of traced process, and examine and change its binary image and registers. The drawback of using Ptrace is performance, which is influenced by its high context-switch overheads between tracer and traced application. Uprobes is also a dynamic instrumentation mechanism for user-space application in Linux. The detail implementation is not final and it is not accepted in Linux community until developing this energy profiling tool. But Uprobes significantly improves the performance overhead of Ptrace because of inherent from the Kprobes approach.

In this thesis, we adopt the Kprobes mechanism to record system activities dynamically in function-level and code block-level. The Uprobes patch listed in [27] is used to assist us to develop Uprobes mechanism and support dynamic source instrument in user-space application.

Chapter 3

Design and Implementation of Energy Profiling Tool

The processes share hardware resource, and operation of hardware resource consumes energy. The typical energy consumption behavior of embedded system is shown in Figure 1. In this thesis, we design a measurement-based energy profiling tool which can provide

- GUI front-end tool + profiling on the native (non-Java) part of Android system
- Dynamic source instrument
- Coupling source code to energy consumption
- Fine-grained performance and energy profiling reports
- Mapping energy consumption of I/O devices to corresponding software functions
- Accurate measurement results with low overhead

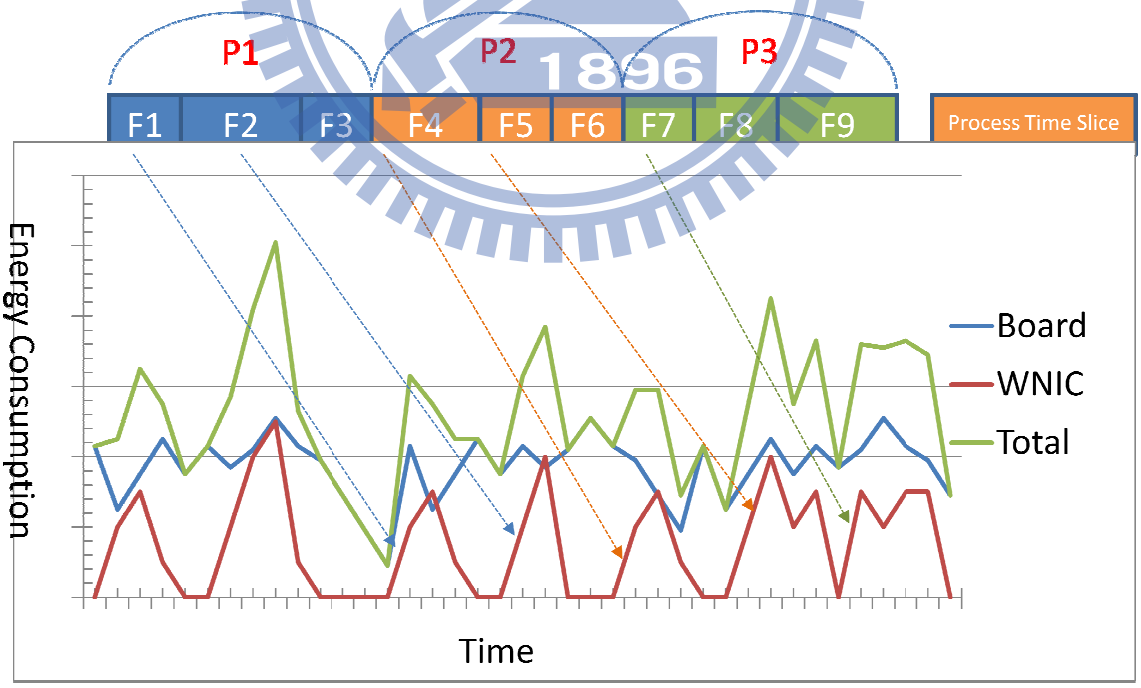


Figure 1 Processes share hardware resources

3.1. Methodology

The main idea of measuring system's energy consumption can be separated into three parts,

1. Recording System Activities, a process is a running instance of a program stored in the memory, the system activities are the usage instance of a hardware component by a process. Recording the running time slices for each process in the system can help to obtain process level's energy consumption. To have fine-grained energy consumption report, the running time slices for each function within process and running time slices for each block within function are necessary. The penalty is the performance and storage overhead when using static instrument method. Fortunately, dynamic source instrument technique can be used to insert or remove interested probes dynamically and reduce the instrument overhead. Figure 2 represents a piece of application source code. From process-level energy consumption report, we found that this application consumes a lot of energy. We would like to find the energy hot spot of this application. To do this, we separate the main function into three source code blocks, and measure the energy consumes by these three blocks. We found that block2 consumes more energy than block1 and block3 in first run. Therefore, we focus on energy consumption of block2 and found that sort function consumes most of energy. We can concentrate on optimizing performance and energy consumption of sort function.

2. Measuring system's power sample, using external measurement device (DAQ) and connect it to specific measurement points in the device. DAQ can record average loop current and voltage of each hardware component for a given duration. The accuracy of energy consumption will depend on the sampling frequency of DAQ.

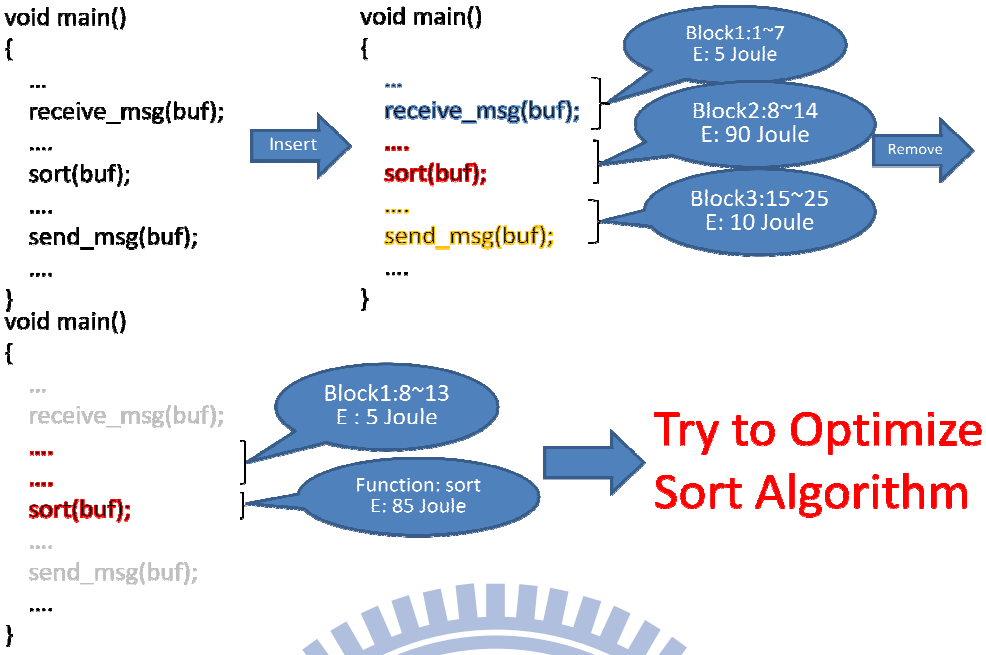


Figure 2 function-level and code block-level energy consumption concepts

3. Correlating system activities and power sample,

The system activities and power samples are measured simultaneously with two hardware platforms. The time synchronization between two hardware platforms should be done correctly to ensure power samples are assigned to running processes correctly. There are several time synchronization methods such as network time protocol (NTP) and external trigger provided by DAQ. We adopt external trigger method to do time synchronization between two platforms. The general purpose input and output (GPIO) signal is used in the device to trigger the external DAQ and start to collect power samples. The timestamp is recorded in the device while triggering the DAQ simultaneously. It is a reference timestamp which mapping to the first power samples. By recording the system activities information such as process ID (PID), function ID (FID), block ID (BID), start timestamp, and end time stamp, the power samples can be correlated with system activities. In figure 3, it represents power samples variation with time. We can correlate each time slice of running processes to power samples using reference timestamp. For the energy consumption that occurs between s_{tart} and t_{end} in Function 1(F1) of

Process 1(P1) can be represented in formula (1), where P (t) = I (t) $_{current}$ x V (t) $_{voltage}$. The same concept can be applied to provide the energy consumption to function-level and block-level.

$$E_{F1}^{P1} = \int_{t_{begin}}^{t_{end}} P(t)dt \tag{1}$$

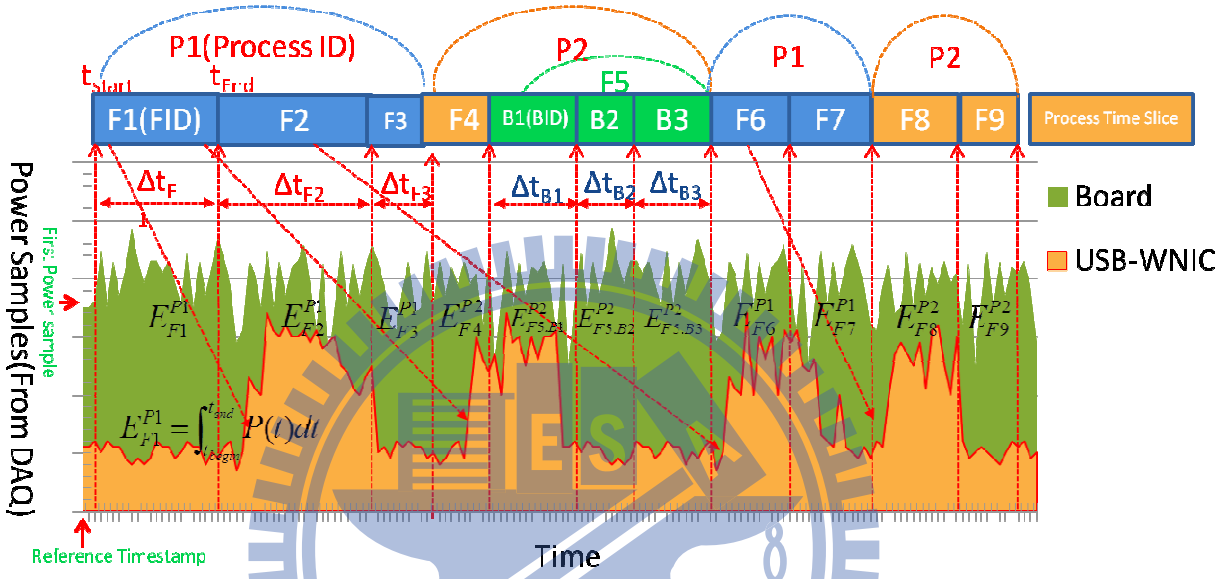


Figure 3 Correlating power samples with system activities

However, the following I/O features cause correlating energy consumption with software to become difficult.

1. Device driver usually removes process ID of I/O events in kernel space for better I/O efficiency.
2. Operating system usually performs scatter-gather for better I/O efficiency.
3. I/O events are asynchronous, and asynchronous issue could be further divided into two phases. The first phase is from application issues the I/O request to the request is processed at driver level. The second phase is from driver notifies device to the device is actually performed the I/O request.

Figure 3 also shows I/O behavior of WNIC. The send or receive I/O requests from Process 1 may be delayed and happened in time slice of process 2 due to its asynchronization I/O nature. We need to modify the kernel to link the applications to related transmitted/received packets.

3.2. System Architecture

The energy profiling tool consists of the monitored target and host PC ad depicted in the Figure 4. The monitored target device is in charge of recording system activities and host PC is in charge of collecting power samples. The power samples and system activities are correlated together in host PC. Putting the time consuming tasks in host PC can reduce significant amount of monitor overhead for target device.

The profiling tool in the host PC consists of the graphical user interface (GUI) front-end, source-level and function-level probes module, symbol to address helper, and energy consumption analyzer. The GUI front-end provides several features, users can select benchmark program, and checking/un-checking preferred functions and code block, and controlling DAQ and target device to start/stop the profiling, and setting the profiling duration. The profiling results are presented with well formatted GUI in the host PC.

The Source/Function Probe module is in charge of preparing the probes information, and symbol to address helper will translate function name and lines of source code to its program address. The probes information will send to target device and register/un-register to dynamic instrument module. The Energy consumption analyzer is in charge of analyzing profiling data and power samples. The process/function/code block time slices will be mapped to corresponding CPU and Wi-Fi power samples to generate fine-grained energy consumption report.

In the target site, there are two user-space Daemons, PowerMemo and Energy Probes. They are corresponding to Energy Probes kernel module and System Monitor. System Monitor kernel module stores time slice of benchmarking program in process level, function level, and code block level. When the kernel buffer is almost full, kernel module notifies the PowerMemo daemon to receive these kernel level data from kernel-space to user-space as files, and at the end of the profiling it automatically transfers these files to host side for the mapping process. The Energy probes daemon handles register/un-register requests from host PC, these requests will convert to compatible data structure and use IOCTRL provided by Energy probes kernel module to register/un-register the requirement to Kprobes/Uprobes module. The Kprobes/Uprobes provides a set of APIs to service the register/un-register requests for dynamic source instrument in user-space application and Linux kernel. The detail about dynamic source instrument will be described in later section.

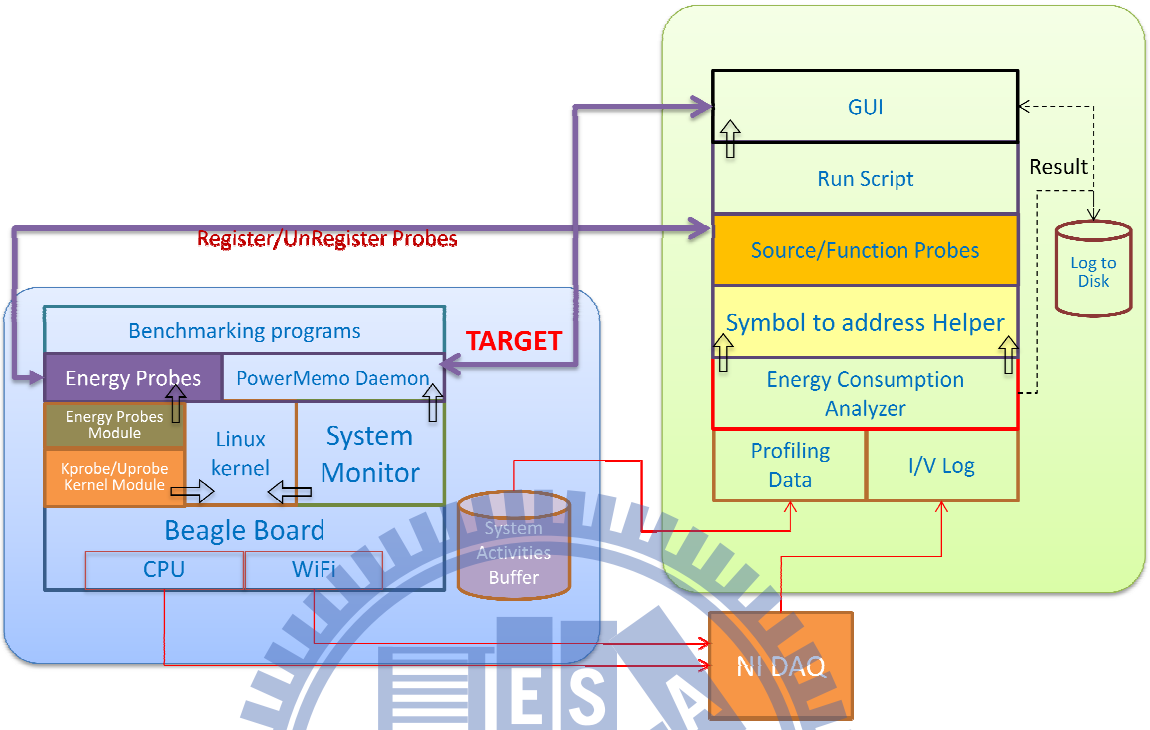


Figure 4 System Architecture

3.3. Recording System Activities

Previous researches on measurement based profiling tool use statistical sampling or manual instrument techniques to record system activities. Statistical sampling can have fine-grained profiling information when sampling interval is short. However, the overhead is high and needs more storage space. Manual instrument can have lower overhead but is not suitable to instrument system activities to function-level or code block-level. The software designers need to spend a lot of efforts to instrument the source code. There are several compiler techniques such as prof, gprof and kernel function trace can help to record system activities. But it is not flexible and need to re-compile the source code. We use dynamic source code instruction technique to improve the drawback of statistical sampling and manual instrument. The rest of this chapter is organized as below: Section 3.3.1 goes through the details about dynamic instrument in user-space application and Linux kernel. Section 3.3.2 describes how to correlate line of codes to energy consumption. Section 3.3.3 and section 3.3.4 present the implementation of recording CPU and WNIC system activities.

3.3.1 Dynamic Source Instrument

As depicted in previous section, there have several performance profiling techniques. We adopt Kernel Probes (Kprobes) to perform dynamic source code instrument. The basic concepts of dynamic instrument are shown in Figure 5. The instruction in preferred program address is replaced to un-defined or break instruction (depending on the architecture) when calling the register probe API. Once the program hits the un-defined instruction, CPU traps an un-defined exception. The user pre-defined handler will be called based on interrupted program address. We can collect timestamp, process ID, function ID, and block number in pre-defined handler. The original instruction and un-defined instruction will be copied to new allocated page. After returning from exception handler, the program counter is set to memory address of original instruction in new allocated page and executing the original instruction there. It is called single step-out-of line. After doing single step out-of-line of original instruction, the un-defined instruction is hit in new page. CPU traps an un-defined exception and user-defined post handler is called. The program counter will be set to memory address next to original instruction address and continue normal program flow.

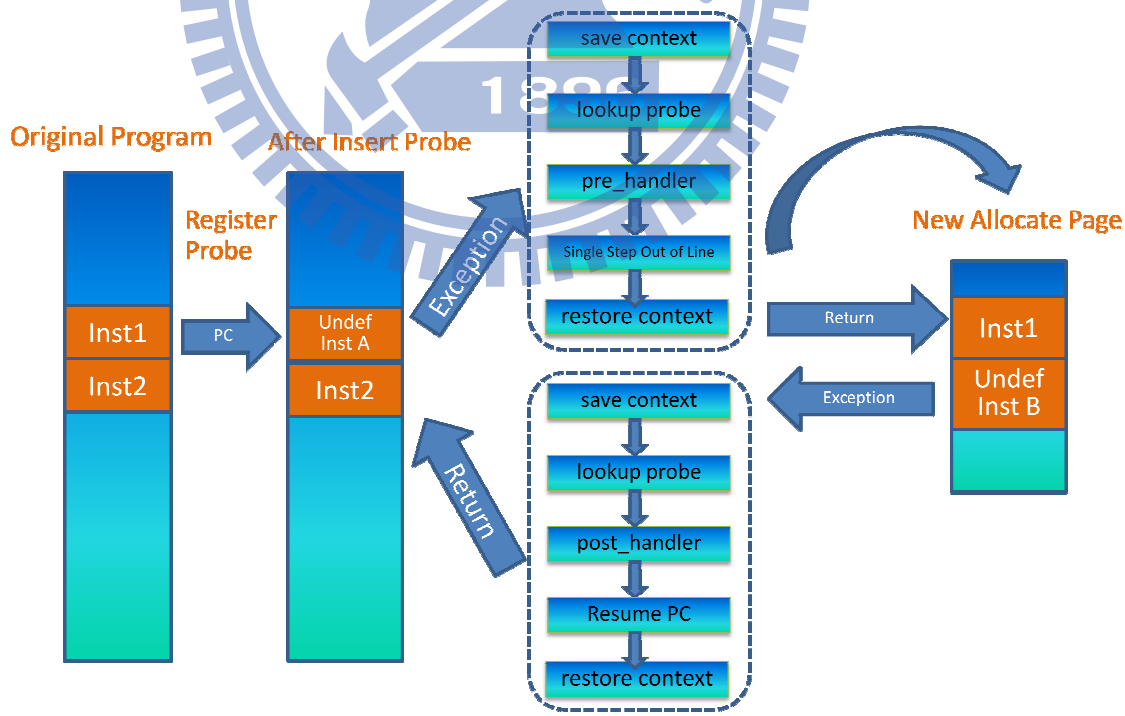


Figure 5 Dynamic instrument concepts

Table 1 lists Kprobes and Uprobes APIs for kernel space and user-space program. The input arguments such as function name or function address and user-defined handler are necessary for function-level register probe function. The input arguments such as instruction address and user-defined handler are necessary for the instruction-level register probe function. Additional information such as application name and binary path are also necessary for user-space register API.

Table 1 Kprobes and Uprobes API

Kernel Space		User Space	
register_kretprobes(…)	Register function level probe	register_uretprobe(…)	Register user-space function-level probe
unregister_kretprobes(…)	Unregister function level probe	unregister_uretprobe(…)	Unregister user-space function-level probe
register_kprobe(…)	Register instruction level probe	register_uprobe(…)	Register user-space instruction probe
unregister_kprobe(…)	Unregister instruction level probe	unregister_uprobe(…)	Unregister user-space instruction probe

To reduce the dynamic instrument overhead, we can single step the original instruction using simulation and emulation method. Simulation is where the instruction's behavior is duplicated in C code. Emulation is where the original instruction is rewritten and executed, often by altering its registers. Using simulation and emulation method can reduce the exception number and improve the instrument overhead. Figure 6 represents the flow for improved dynamic instrument method. We adopt this method in this thesis.

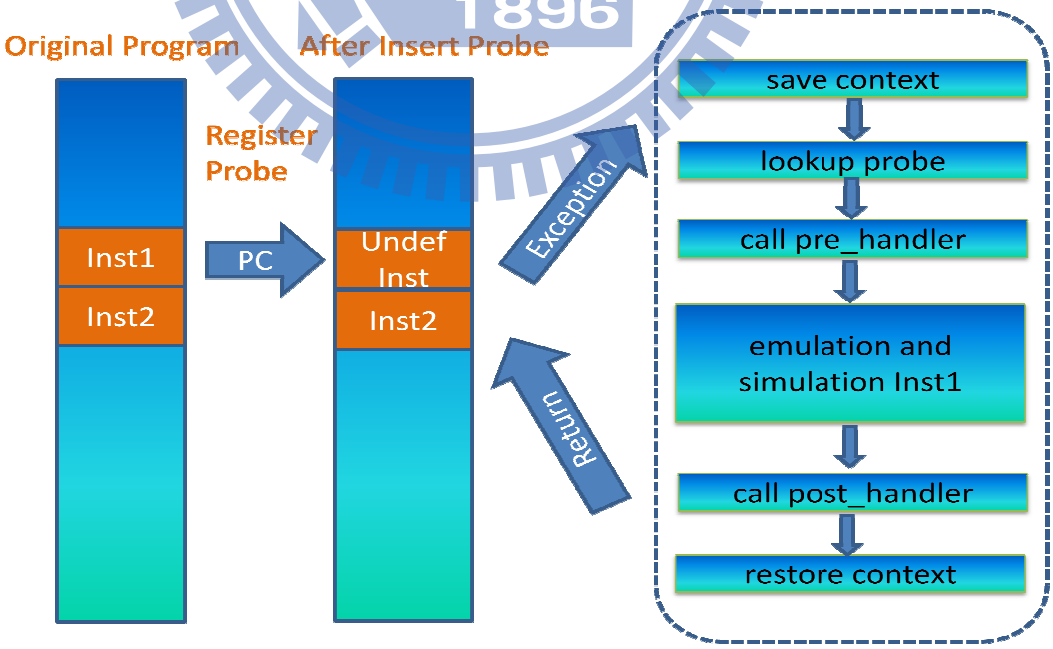


Figure 6 Reducing dynamic instrument overhead

The call flow of function-level instrument is presented in Figure 7. The exception is trapped when function is called and user-defined handler is used to record the enter timestamp. The exception is trapped again when function return and user-defined return handler is used to record the exit timestamp.

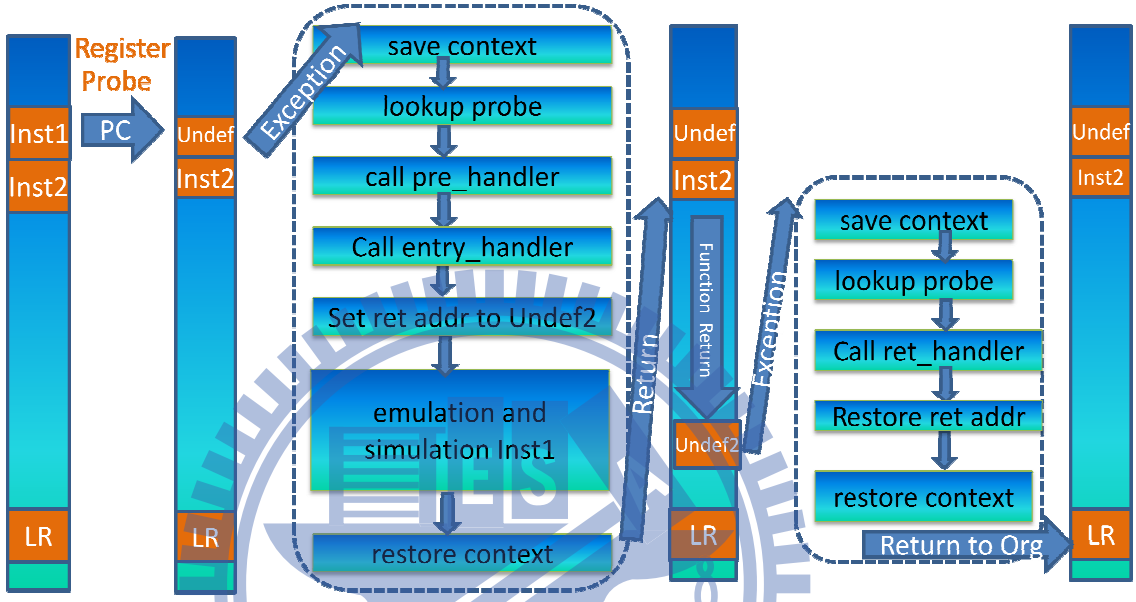


Figure 7 Function-level dynamic instrument concepts

3.3.2 Correlating Energy to Source Code

Correlating the source code with energy consumption makes it possible for software designer to optimize their system's energy consumption. Using a set of tool-chain utilities, we can create lines of code and instruction address mapping table as depicted in Figure 8. We can implement block-level energy consumption feature by using instruction-level register probe functions. For instance, source code can be separated into three blocks, line 5~6, line 7~10, and line 11~12 in Figure 8. Using line to instruction address mapping table and register probe function, we can record the timestamps when program hits these two lines by inserting instruction probes. Correlating timestamps with collected power samples, we can provide energy consumption during the running instant of program between line 7~10.

C Program Source Code	Line	Instruction Address
#include <stdio.h>	1	
#define LAST 10	2	
int main( )	3	
{	4	0x8430
int i = 0;	5	0x843c
int sum = 0;	6	0x8444
for (i = 1; i <= LAST; i++)	7	0x8450
{	8	0x8454
sum += i;	9	0x8478
} /* -for- */	10	0x8482
printf("sum = %d\n", sum);	11	0x848c
return 0;	12	0x8490
}	13	0x8494

Figure 8 lines of code and instruction address mapping

3.3.3 Recording CPU System Activities

The schedule ( ) function is the implementation of scheduler in Linux kernel. Its objective is to find a process in ready queue list and assigns the CPU resource to it. The target device records time slice of the running processes by putting the process_entry( ) and process_exit( ) functions in place where scheduler switch out the old process and switch in the new process in schedule( ) function. For system activities in function-level and block-level, the markerfunc_entry( ) and markerfunc_exit( ) functions are used as user-defined handlers in register probes APIs. The information such as process id, function id, block id, function name, block name, start timestamp, and end timestamp (relative to reference timestamp in Figure 3) are recorded in kernel buffer. Since the kernel buffer size is limited, it sends a signal to notify user-space daemon that kernel buffer is almost full. User-space daemon receives the signal, and reads the system activities from kernel buffer and write to the files. These files are sent to host PC after measurement duration expired or user stops the energy measurement manually. Figure 9 represents that per-process stacks are used when calling process_entry( ) or markerfunc_entry( ) functions. The process_exit( ) or markerfunc_exit( ) functions are called, and the related information will be pop out and stored to kernel buffer.

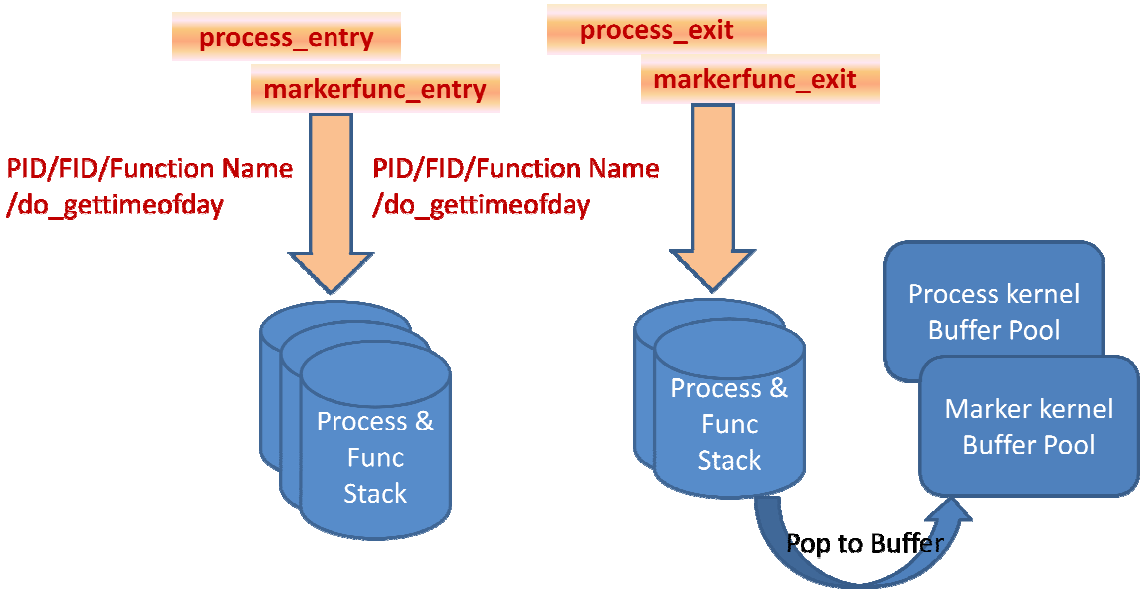


Figure 9 per-process stacks to record system activities to kernel buffer

3.3.4 Recording WNIC System Activities

To obtain begin and end time of process activities that cause energy consumption in the WNIC, we patch the network subsystem and new mac80211 subsystem in Linux kernel. The mac80211 subsystem lies between the kernel's networking stack and WNIC device drivers and provides valuable information that was once only available in hardware logic or device drivers. We can get the data bit rate value for each packet transmitted or received from this subsystem without instrument any driver code. Our implementation calculates duration to transmit or receive a packet using bit rate value, packet size and initial time of the transmit/receive event. We get the begin time of a transmit event before the underlying device driver begins to send out the first bit of the packet to air, and use bit rate value of this transmit event to calculate the end time in offline mapping stage. For a receive packet event, we get the end time when the last bit of the packet reaches to the device driver, and then we use the bit rate value of this newly arrived packet to calculate the begin time. Following formulas summarize this technique:

$$t_{end} = t_{begin} + \frac{packet\ size\ in\ bits}{TX\ bit\ rate} \quad (2)$$

$$t_{begin} = t_{end} - \frac{packet\ size\ in\ bits}{RX\ bit\ rate} \quad (3)$$

$$\Delta t = t_{end} - t_{begin} \quad (4)$$

A duration value (Δt) is the key to energy calculation for a system activity, but is not sufficient to be able to charge processes correctly for the energy they consume. We also need to identify the processes that are responsible for the energy consumed during every set of begin-end time. It is more complicated due to the asynchronous nature of I/O operations. To solve this problem for WNIC devices, we have added process id item to the socket data structure in network subsystem. When we record the time values for a packet, we simply access this item through the packet's socket structure.

3.4. Power Measurement

We adopt the NI-9223 data acquisition (DAQ) card from National Instruments as the measurement device. It supports a common API that can be used under various programming languages like VB, C, C++ and C#. When the "START" button is clicked on the main window of profiling tool GUI, DAQ card is set to run at 50kSample/s/channel and waits for a trigger input from target device. Right after that, a begin command is sent to the user-daemon running in the target. When this command is received, user-daemon uses IOCTRL to triggers GPIO signal which is connected to the trigger input of the DAQ card and records the reference timestamp of target system. After user-daemon loads the benchmarks and issues an IOCTL call to cause the system monitor to set a global flag so that kernel profiling can begin. When the trigger is sensed by DAQ card, it begins to sample and stores the collected samples in files.

3.5. Correlating Energy with System Activities

Energy analyzer module is responsible to correlate power samples with system activities. Energy consumption analyzer module analyzes system activities files to detect the Process ID, Function ID, and Block ID values that were collected during the test. Later on PID values are charged with energy values for both CPU and WNIC activities. As one of profiling tool's limitation, Function ID and Block ID values are charged only with the energy values of CPU activities. The process's total energy consumption can be represented as formula (5) in Beagleboard.

$$E_{PID} = \sum (T_{PID} \times P_{PID}^{Board}) + \sum (T_{PID\_WNIC} \times P_{PID\_WNIC}^{WNICidle}) + E_{PID\_WNIC}^{WNIC\_ACTIVITY}$$
$$E_{PID\_WNIC}^{WNIC\_ACTIVITY} = \sum (T_{PID\_WNIC\_TX} \times P_{PID\_WNIC}^{WNICtx}) + \sum (T_{PID\_WNIC\_RX} \times P_{PID\_WNIC}^{WNICrx}) \quad (5)$$

Chapter 4

Experiment Overview

In this thesis, we propose a measurement base energy profiling tool which can provide process-level, function-level, and code block-level energy consumption report. We design several experiments to demonstrate the capabilities and correctness of our tool. The first experiment measures power consumption of functions with different instruction classes and compares the results with previous researches. The second experiment measures power consumption before and after turning on device's LEDs. The results of these two experiments can show the correctness of our profiling tool in function-level and block-level. The third experiment demonstrates the capability to measure the energy consumption of USB Wi-Fi interface. We also measure the energy consumption of FTP client by sending files with different size to FTP server.

4.1. Experiment Environment

The experiment environment is shown in Figure 10. The measurement equipment contains a current probe and a data acquisition (DAQ) card. The current clamp uses Ampere's law to measure current flow of target I/O device, and the data acquisition card collects measured current value by current clamp. The experimental platform is beagleboard xM using TI Cortex A8 with 1 GHZ (DM3730). The Wi-Fi 802.11 BG USB adaptor is connected to the target board as the I/O component and connected to D-Link DIR-600 AP Router via air. The programs execute on the experimental platform running the Linux kernel 2.6.37 with Android framework version 2.3.4.

Table 2 Specification of experiment equipments

	Name	Model
Measurement Equipment	DAQ	NI cDAQ-9174
	Current clamp	Fluke I30S
Target	Experimental platform	BeagleBoard xM Rev C
	USB WLAN adaptor	D-Link DWL-G122
	Wi-Fi Router	D-Link DIR-600

Figure 10 Measurement Equipments

4.2. Cases study and results

The first experiment measures power consumption of functions with different instruction classes. The functions are implemented using inline assembler as depicted in Figure 11. The add instruction is duplicated 10000 times using .rept directive in addinst() function. The experiment result shows that power consumption of multiply instruction is higher than add, sub, and nop instructions. Previous researches [16][17] also represent the same result for ARM7TDI and ARM926EJ-S- core respectively.

```
void mulplyinst()
{
   asm(
      ".rept 10000"
      "mul r1,r2,r3\n"
      ".endr"
   );
}
void addinst()
{
   asm(
      ".rept 10000"
      "add r1,r2,r3\n"
      ".endr"
   );
}
```

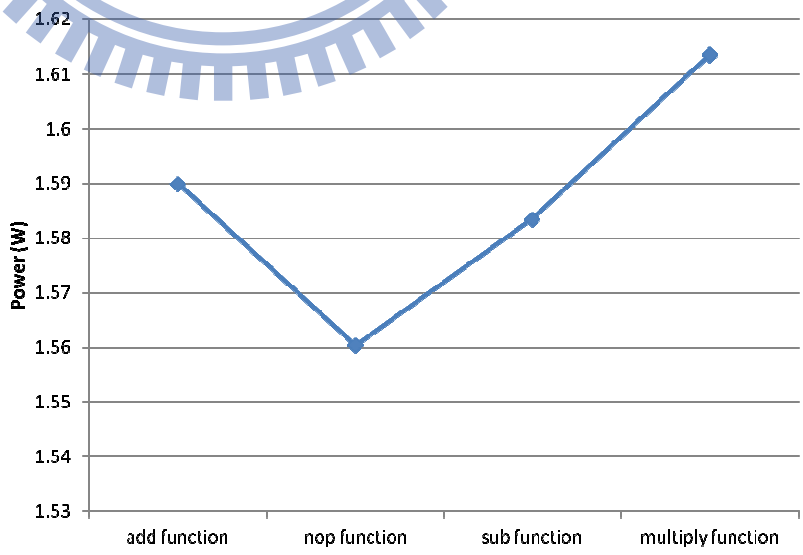


Figure 11 Power consumption of different instruction types

The second experiment measures power consumption before and after turning on LED in target device. The instruction numbers are varied from 10000 to 160000 for different functions. It shows consistent results for different functions before and after turning on LED in Figure 12.

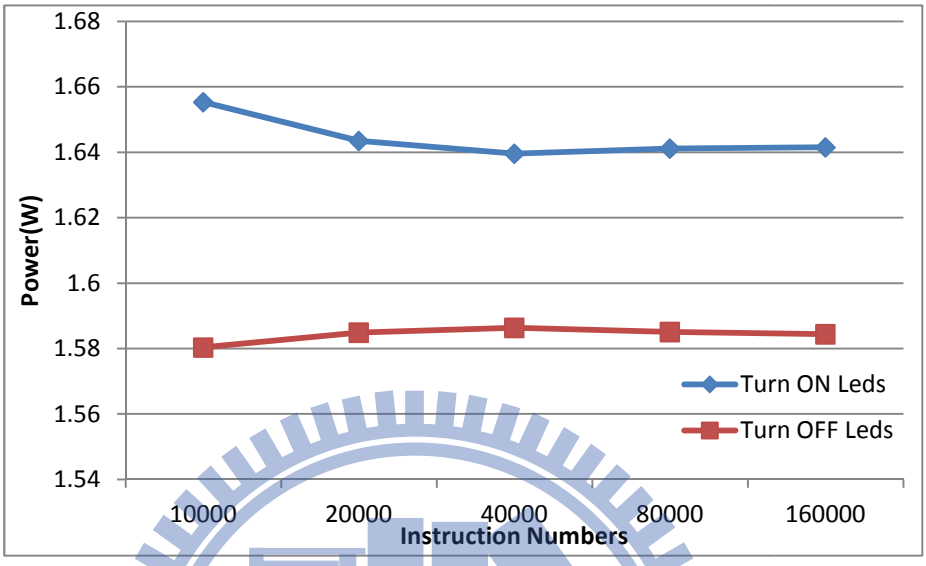


Figure 12 Power consumption of turning on/off LED

The next experiment represents the total energy saving when we turn on the CPU IDLE feature supported in Linux kernel. We observe the energy consumption of the measurement result and find that energy consumes by idle process is high. The measurement result shows that turn on CPU IDLE feature can save 13.5% total system energy and save 250 mW average Power.

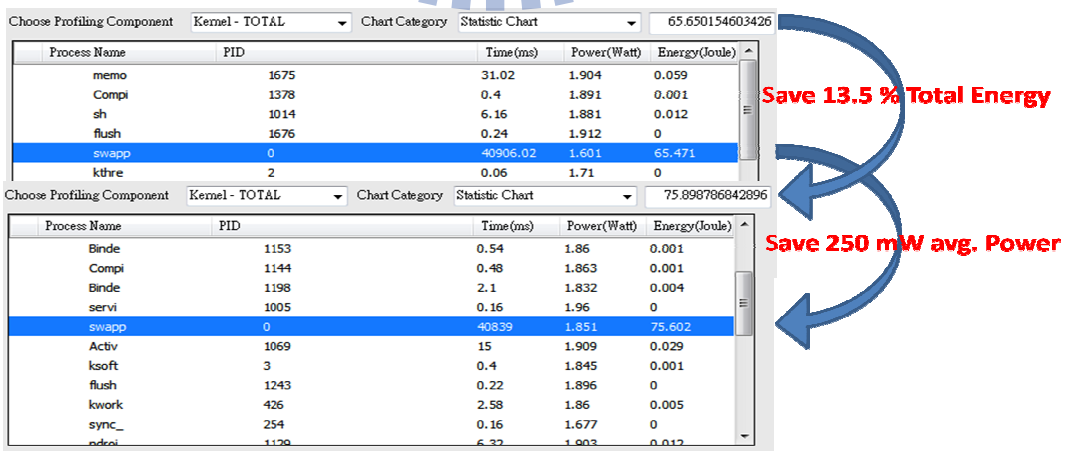


Figure 13 Energy consumption of enabling/disabling CPU IDLE feature

Most mobile devices support Wi-Fi interface. In this experiment, we would like to know how Wi-Fi component contributes total energy consumption of mobile device. Android uses a modified wpa_supplicant daemon for Wi-Fi support and uses dhcpcd daemon to obtain the IP address. We measure the total energy of device before (E1) and after (E2) inserting USB Wi-Fi dongle. The total energy increases 22.93 Joule after inserting USB Wi-Fi dongle (30% total energy increases, (E2-E1)/E1). After inserting USB Wi-Fi dongle, we execute wpa_supplicant daemon to establish Wi-Fi connection and measure total energy (E3). The total energy increases is 18.29 Joule (53.8% total energy increases, (E3-E1)/E1). After establishing Wi-Fi connection, we execute dhcpcd daemon to obtain IP address from dhcp server. Wi-Fi component consumes less energy than CPU because dhcp protocol takes only four packets to complete IP acquirement flow. The total energy increases is unremarkable when executing dhcpcd daemon. Disable Wi-Fi component can improve the energy consumption significantly.

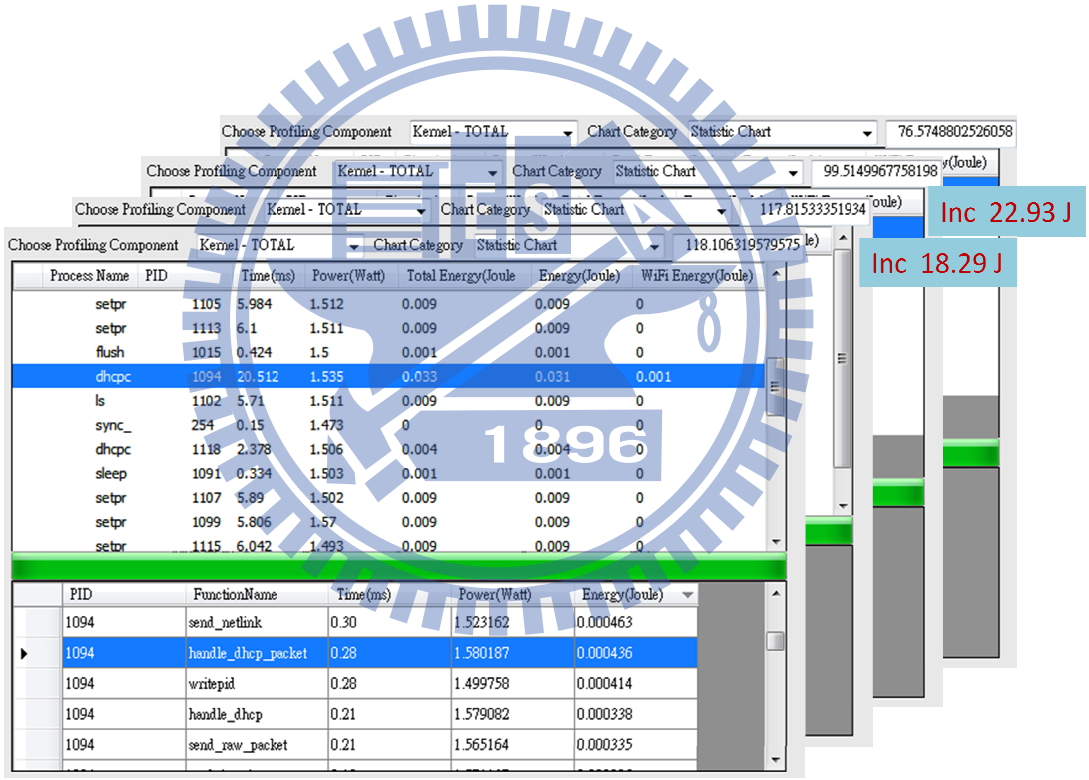


Figure 14 Energy consumption of target device with various scenarios

The next experiment measures energy consumption of FTP client sending files with different size to FTP server. The Wi-Fi component contributes more energy consumption than CPU as depicted in Figure 15 and Table 3. The total energy consumption is proportional to transfer file size.

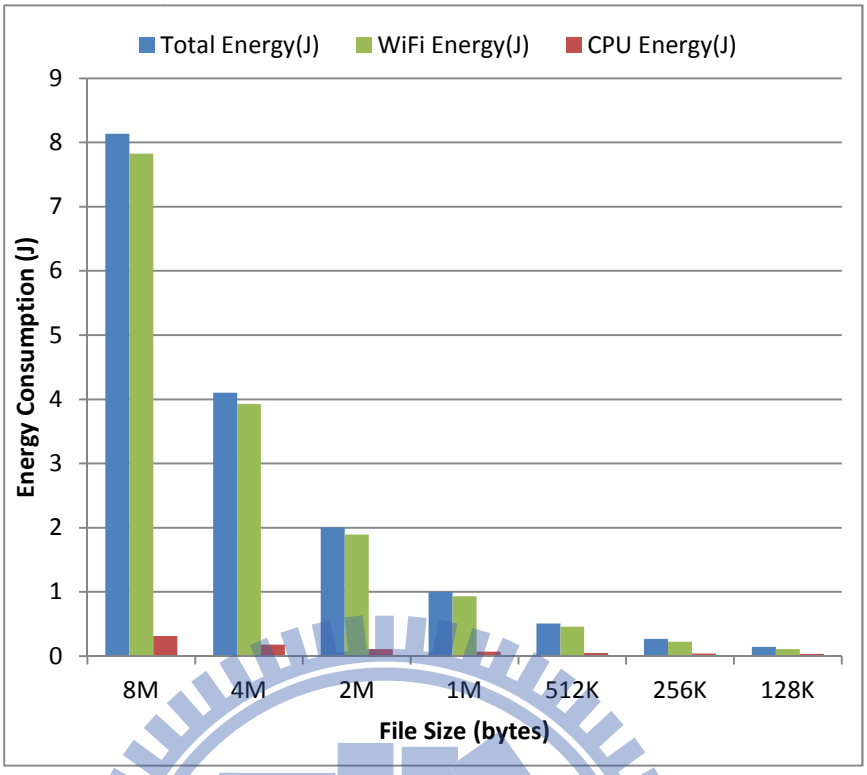


Figure 15 Energy consumption of FTP Client with different file size

Table 3 Energy consumption of FTP Client

File Size(bytes)	Time(ms)	Total Energy(J)	CPU Energy(J)	WiFi Energy(J)
8M	207.938	8.138	0.311	7.828
4M	117.601	4.103	0.176	3.928
2M	72.414	2.003	0.108	1.895
1M	43.498	0.999	0.067	0.932
512K	31.368	0.507	0.048	0.459
256K	25.942	0.266	0.041	0.225
128K	21.672	0.143	0.034	0.111

We have demonstrated the capabilities of our profiling tool in previous experiments. The next experiment wants to evaluate the performance overhead of dynamic source instrument. We chose Lmbench [38] performance analysis tool as our benchmark program. The average running time is measured in three cases, instrument all functions, instrument all instructions within main() function, and without instrument in benchmark program. The measurement results are represented in Figure 16. Per-function instrument overhead is 0.021 milliseconds and is less than per-instruction instrument overhead which is 0.149 milliseconds. The instruction numbers within main( ) function is too small and may lead to higher overhead.

	bw_pipe	bw_tcp
AVG RUN Time(s) (Without Instrument)	27.75	0.56
AVG RUN Time(s) (Instrument All functions)	27.85	0.61
Overhead(s)(1)	0.1	0.05
Number of Function Calls(Function Numbers)(2)	4791(91)	3814(99)
Overhead(ms) of Per Functions Instrument = (1)/(2)	**0.020872**	**0.013109**
AVG RUN TIME(s) (Instrument All Instructions within main() function)	27.76	0.57
Overhead(s)(3)	0.01	0.01
Number of Instruction Calls(Instruction Numbers)(4)	67(142)	76(216)
Overhead(ms) of Per Instructions Instrument = (3)/(4)	**0.149253**	**0.131578**

Figure 16 Dynamic source instrument overhead

Chapter 5

Conclusion

In this thesis, we have developed an energy profiling tool which can provide GUI front-end and profiles native part of Android system. Using the dynamic source instrument technique, we have improved our profiling tool and provide more flexible way to record system activities. Using line of codes and instruction address mapping makes it possible to coupling source code to energy consumption. We verify the correctness of profiling tool by several experiments. The result shows that the profiling tool can measure power consumption in process-level, function-level and code block-level with low overhead. Furthermore, the asynchronize I/O issue is addressed. We instrument the kernel code to map energy consumes by Wi-Fi component with correct process and demonstrate the ability of the profiling tool on asynchronize I/O issue using ftp client application. The well formatted performance and energy profiling reports in process-level, function-level, and code block-level make software designers to focus on the energy consumption hot spot and optimize their software in a better way.

Chapter 6

References

[1] J. Flinn and M. Satyanarayanan, “Powerscope: A Tool for Profiling the Energy Usage of Mobile Applications”, in Proceedings of Second IEEE Workshop Mobile Computer Systems and Applications, 1999.

[2] Changjiu Xian, Le Cai, and Yung-Hsiang Lu, “Power Measurement of Software Programs on Computers With Multiple I/O Components”, IEEE Transactions on Instrumentation and Measurement, Vol. 56, pp. 2079-2086, 2007.

[3] Kutty S Banerjee, Emmanuel Agu., “PowerSpy: Fine-Grained Software Power Profiling for Mobile Devices”, in Proceedings of IEEE WirelessCom, 2005.

[4] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li , and K.W. Cameron., “Powerpack: Energy profiling and analysis of high-performance systems and applications”, IEEE Transactions on Parallel and Distributed Systems, 2010.

[5] IAR Systems, http://www.iar.com/en/Products/IAR-Embedded-Workbench/Power-debuggi ng.

[6] A. Kansal and F. Zhao, “Fine-grained energy profiling for power-aware application design”, in Proceedings of the Workshop on Measurement and Modeling of Computer Systems, 2008.

[7] T. Do, S. Rawshdeh, and W. Shi, “pTop: A Process-level Power Profiling Tool”, in Proceedings of the Workshop on Power Aware Computing and Systems, October 2009.

[8] Changjiu Xian, Yung-Hsiang Lu, Zhiyuan Li, “A Programming Environment with Runtime Energy Characterization for Energy-Aware Applications”, ISLPED'07, August 27-29, 2007.

[9] Kanishka Lahiri, Anand Raghunathan, Sujit Dey, “Efficient Power Profiling for Battery-Driven Embedded System Design”, IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol. 23, No. 6, June. 2004.

[10] Y.-H. Lu, L. Benini and G. Michelli, “Power-Aware Operating Systems for Interactive Systems”, IEEE Transactions on VLSI Systems, Vol. 10, No. 2, Apr. 2002.

[11] Yunsi Fei , Srivaths Ravi , Anand Raghunathan , Niraj K. Jha, “Energy-optimizing source code transformations for operating system-driven embedded software”, ACM Transactions on Embedded Computing Systems (TECS), Vol. 7, No 1, December 2007.

[12] Tajana Šimunić , Luca Benini , Giovanni De Micheli , Mat Hans, “Source code optimization and profiling of energy consumption in embedded systems”, in Proceedings of the 13th international symposium on System synthesis, September 20-22, 2000.

[13] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: A first step toward software power minimization”, IEEE Transactions on VLSI System, Vol. 2, 1994.

[14] V. Tiwari, S. Malik, A. Wolfe, M. Lee, “Instruction Level Power Analysis”, Journal of

VLSI Signal Processing Systems, No 1, pp.223–2383, 1996.
[15] T. Simunic, L. Benini, G. De Micheli,“Cycle-Accurate Simulation of Energy Consumption in Embedded Systems”, DAC, 1999.
[16] N. Chang, K. Kim, and H. G. Lee, “Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI”, in Proceedings of Int. Symp. Low Power Electron. Design, pp.185 - 190, 2000.
[17] Blume, H., Becker, D., Rotenberg, L., Botteck, M., Brakensiek, J., Noll, T.G. ”Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures”, Journal of Systems Architecture, 53 (10), pp. 689-702, 2007.
[18] SystemTap, http://sourceware.org/ systemtap/
[19] J. Levon, “Oprofile - a system profiler for linux”, http://oprofile.sourceforge.net/doc/ index.html.
[20] Tracing Wiki, http://lttng.org/tracingwiki/index.php/Tracepoints_and_Markers.
[21] Ptrace, http://en.wikipedia.org/wiki/Ptrace.
[22] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, “probing the guts of kprobes”, in Ottawa Linux Symposium, pp. 101–115, 2006.
[23] Yao Guo, Ziwen Chen, Xiangqun Chen, “A Lightweight Dynamic Performance Monitoring Framework for Embedded Systems”, Embedded Software and Systems, ICESS '09. 25-27 May 2009.
[24] Alexey G., Sergey G., Jaehoon J., “Dynamic Binary Instrumentation Framework for CE Devices”, in Proceedings of the Linux Symposium, July 13th–16th, 2010.
[25] Jim K., Ananth M., Prasanna P., Vara P., “Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps”, in Proceedings of the Linux Symposium, Volume One, June 27th–30th, 2007.
[26] LessWatts. http://www.lesswatts.org
[27] uprobes patch, http://thread.gmane.org/gmane.linux.kernel/390558
[28] Kprobes Support for MIPS, Lubna, Vikas, Madhvesh, Sony India Software Centre. http://elinux.org/images/4/44/Kprobes-MIPS-overview.pdf
[29] RANGANATHAN, P., GEELHOED, E., MANAHAN, M., AND NICHOLAS, K. “Energy-Aware User Interfaces and Energy- Adaptive Displays”, Computer 39, 3, 31-38, 2006.
[30] Cheng-kun Yu, Wen-Chih Peng,“Profiling Energy Consumption of I/O Events for Embedded Systems”, A Thesis for master degree of Dept. of Comput. Sci. & Inf. Eng., Nat. Chiao Tung Univ., Hsinchu, Taiwan, 2010.
[31] Ilter Suat and Shiao-Li Tsao, “Energy Consumption Profiling Tool for Mobile Devices in an Emulated Wireless Environment”, A Thesis for master degree of Dept. of Comput. Sci. & Inf. Eng., Nat. Chiao Tung Univ., Hsinchu, Taiwan, 2010.
[32] Xiang Zhou, Bing Guo, Yan Shen and Qi Li, “Design and implementation of an improved C source-code level program energy model”, Embedded Software and Systems,

2009.
[33] Pollari, M. and Kanstren, T., “A Probe Framework for Monitoring Embedded Real-time Systems”, Internet Monitoring and Protection, 2009.
[34] J. Levon, “Oprofile - a system profiler for linux”, http://oprofile.sourceforge.net/doc/index.html.
[35] Beagle board xM Platform, http://beagleboard.org/
[36] NI PCI-6115 DAQ Card, http://sine.ni.com/nips/cds/view/p/lang/en/nid/11886
[37] D-Link DWL-G122 USB Wireless NIC - FW Version 3.0, http://www.dlink.com/products/?pid=334
[38] Lmbench – Tools for Performance Analysis, http://www.bitmover.com/lmbench/