Graphical User Interface Energy Characterization for Handheld Computers*

Lin Zhong and Niraj K. Jha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{lzhong, jha}@ee.princeton.edu

ABSTRACT

A significant fraction of the software and resource usage of a modern handheld computer is devoted to its graphical user interface (GUI). Moreover, GUIs are direct users of the display and also determine how users interact with software. Given that displays consume a significant fraction of system energy, it is very important to optimize GUIs for energy consumption. This work presents the first GUI energy characterization methodology. Energy consumption is characterized for three popular GUI platforms (Windows, X Window system, and Qt) from the hardware, software, and application perspectives. Based on this characterization, insights are offered for improving GUI platforms, and designing GUIs in an energy-efficient and aware fashion. Such a characterization also provides a firm basis for further research on GUI energy optimization.

Categories and Subject Descriptors

C.m [**Computer systems organization**]: Miscellaneous; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*

General Terms

GUI, Low power

Keywords

Energy characterization, Graphical user interface, Handheld computers, Low power design

1. INTRODUCTION

Energy consumption is a critical concern for handheld computers. User interfaces consisted of an average of 48% of the application code even a decade ago [18]. In modern personal computing systems, the user interface is almost always graphical, which only increases its fraction of source code and resource usage. GUIs are direct users of the display, one of the largest power-consumers in mobile computing systems [4, 8]. Moreover, they determine how users interact with software. Much of the software used in mobile computing systems is not CPU-critical, *i.e.*, the time and hence energy required to finish a task is dependent more on user interaction than on CPU speed. It is therefore important to improve system energy efficiency and awareness through GUI optimization. Energy efficiency refers to minimal energy usage to finish a task while energy awareness refers to the capability of trading other aspects of software for energy savings.

GUI energy characterization is the first step towards the above goal. It not only helps choose the appropriate GUI platform and toolkit, but also helps design an energy-efficient and aware GUI. It is also important for incorporating energy scalability and awareness into software and trading different aspects of software for energy efficiency dynamically.

In this work, we study three different GUI platforms on three handheld computers. However, our work should also be helpful in designing GUI software for other mobile computing systems, such as notebook computers. As far as we know, this is the first work on GUI energy characterization. This characterization shows that GUIs are expensive in terms of energy consumption, their different features consume drastically different amounts of energy, and different GUI platforms are quite different in terms of their energy consumption. The characterization suggests ways to make a GUI more energy-efficient and energy-aware, not only to prolong battery lifetime, but to also finish more tasks in a fixed amount of time.

The paper is organized as follows. We first offer background information on GUI platforms and related works in Section 2. Then we analyze how energy is consumed by a GUI in Section 3 and describe the experimental setup and benchmarks in Section 4. Based on the experimental results presented in Section 5, we offer insights for energy-efficient and aware GUI design in Section 6. Finally, we conclude in Section 7.

2. BACKGROUND

In this section, we first discuss the relevant features of handheld computer software, and then provide information

*Acknowledgments: This work was supported by DARPA under contract no. DAAB07-02-C-P302.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.
Copyright 2003 ACM 1-58113-676-5/03/0010 ...$5.00.

on GUI platforms and development toolkits studied in this work. We then discuss related research work.

2.1 Handheld software

Most of the software loaded on handheld computers is interactive (some exceptions are video/audio players, which are CPU-intensive). Such a software has two prominent properties. First, the execution time usually does not depend on the CPU speed, but on user interaction. Second, most system resources are dedicated to user interaction.

A GUI is responsible for interacting with users. First, it presents information to users graphically, usually through GUI windows like buttons, menus, message boxes, text windows, *etc.* Second, it takes inputs from a user in the form of a user responding to GUI windows. Most GUI platforms are extended to include many non-user interface system functionalities, such as file and network operations, typically by wrapping corresponding system calls into a GUI application program interface (API). In this work, we are only concerned with those parts that present and receive user input [16].

2.2 GUI platforms

Almost no GUI application is programmed to directly manipulate display devices. Different APIs are used to accelerate GUI development and improve hardware independence. Such APIs are called GUI platforms. In this work, we study three of the most popular GUI platforms on handheld computers, *i.e.*, Qt, Microsoft Windows (or Windows), and X Window system. Since the nomenclatures used by these GUI platforms are quite different, Table 1 summarizes the different terms.

Table 1: GUI nomenclatures

This work	Qt	Windows	X/GTK
Event	Signal/ Event	Message	Signal/ Event
Window	Widget	Control/ Window	Widget
Event processing routine	Slot	Callback	Callback

2.2.1 *X Window system*

The C-based X Window system [33] is almost ubiquitous on computers running under Unix-like operating systems (OSs). The X Window system has a client-server architecture, in which a single X server serves requests from different GUI applications (called X clients) through interprocess communication. Each application registers the types of events it wishes to handle with the X server, and also registers an event processing routine with each registered event type. When events associated with a window occur, the X server only sends events of registered types to the window, which in turn calls the corresponding processing routine. The X server handles events of unregistered types as the default. Wrapper toolkits are typically used to facilitate GUI development. GTK [10], one of the most popular toolkits, is used in this work. An embedded port of GTK, called GPE [11], is actively under development. In this study, we use the X Window system that comes with the Familiar project [30] and GTK-related libraries from the public Skiff cluster [31]. We use X/GTK to refer to the X Window system and GTK.

2.2.2 *Qt*

The Qt platform is a C++ based GUI API. It handles events through class member slots and signals. While Qt works with multiple OSs, its embedded port, Qt/Embedded [22], currently only works under Linux. Unlike Qt for desktops, Qt/Embedded applications directly work on the kernel framebuffer without an X server. The absence of an X server reduces its memory requirement significantly. One can also expect an improvement in its performance and energy efficiency. Qtopia, an application environment, has been developed using Qt/Embedded. There is also an open-source fork of Qtopia called Opie [19]. In this study, we use the Qt/Embedded system shipped with a Sharp Zaurus handheld computer. In the following discussions, Qt is used to refer to Qt/Embedded.

2.2.3 *Windows*

Unlike Unix/Linux systems, Microsoft Windows GUI is integrated with the Windows OS with a well-defined API. Such an integration may offer Windows benefits in terms of GUI energy efficiency. Every Windows window has an event handler. All events generated within a window are passed to its event handler through an event loop. There are multiple ways to develop GUI applications for Windows: Win32 API, MFC, ATL, Visual Basic, *etc.* Unlike the X Window system, a significant number of Windows developers use the Win32 API directly to write Windows applications. Therefore, the Win32 API is adopted in this work for energy characterization.

2.2.4 *Other GUI platforms*

We do not address other GUI platforms in this first study. One of the better known is the Palm GUI widely used with Palm OS [20] powered handheld computers. No Palm handheld computer uses an Intel StrongARM processor yet, which would make a fair comparison difficult. Java and Visual Basic are also popular for GUI development. However, their well-known performance disadvantages compared to C/C++ are also translated into energy disadvantages, although they may have advantages in other aspects. Other GUI platforms/toolkits for embedded Linux are described in a recent article [29].

2.3 Related works

Energy consumption for the Itsy Pocket Computer was characterized in [7, 8]. Power usage of a Palm Pilot was characterized in [5]. System power characterizations for notebook computers can be found in [13, 14]. Recent LCD power management techniques can be found in [3, 9]. All these works are hardware-centric and offer limited insight into how software consumes energy. There are also a number of software system energy characterization works. Energy for OSs, especially those used in embedded systems, has been extensively studied [2, 6, 27]. General software power estimation and optimization techniques have been investigated extensively as well [23, 24, 28, 32]. However, none of these software works address the GUI.

Surveys of user interface software tools can be found in [16,

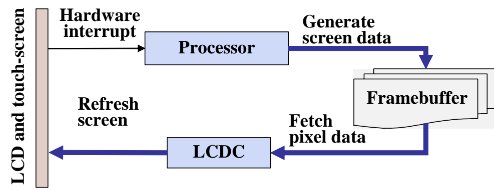

Figure 1: A hardware perspective for GUI energy consumption.

17]. GUI design for handheld computers has also been of research interest. However, most of the work in this area has so far been concerned with limited screen size and input methods. Interested readers may refer to [1, 26] for recent research work.

3. GUI ENERGY CONSUMPTION

Before energy characterization, it is worthwhile to analyze how energy is consumed by a GUI. In this section, we will do so from three perspectives: hardware, software and application.

3.1 Hardware

In the handheld computers we study, the Intel StrongARM SA-1110 system-on-chip (SOC) is used. It has an integrated LCD controller (LCDC). A framebuffer is implemented in off-chip memory (main memory) to store pixel data for a full screen. Whenever there is a screen change, the processor generates new data for the changing screen pixels and stores them into the framebuffer. This implies a higher energy consumption with increased temporal changes in the screen. Meanwhile, to maintain a screen on the LCD, the LCDC must sequentially read screen data from the framebuffer and refresh the LCD pixels even when there is no screen change. This in turn implies a higher energy consumption with increased spatial changes in the screen. The on-chip system bus and off-chip data buses also consume energy for data transfers. For other handheld computers, details may vary but the fundamental organization and operating principles are similar.

The display itself consists of several parts: LCD power circuitry, a front light, and an LCD. The LCDs used in the systems we studied are color active thin film transistor (TFT) LCDs. In such LCDs, each pixel has three components: R, G and B, signifying red, green and blue, respectively. Liquid crystals for each component are independently oriented by two polarizers, which are connected to a storage capacitor. The capacitor is in turn charged and discharged through a TFT to accommodate screen changes. Moreover, the capacitor must be refreshed at a high rate to maintain an appropriate voltage across the polarizers so that the corresponding liquid crystals remain properly oriented.

The hardware perspective is summarized in Figure 1.

3.2 Software

A GUI platform is usually highly OS-dependent. It is impossible to compare GUI platforms without taking the OS into account. The following software processes are involved in GUI usage. First, the OS handles hardware interrupts generated by a user. It then produces events for the GUI platform. The latter delivers events to the GUI application, which catches events through an event loop. Thereafter, the application instructs the platform as to how the GUI should change. The platform coordinates GUIs of different applications, determines how the screen changes, generates new screen pixel data, and then calls OS services to update the screen. Interrupt handling, event processing and screen updating are the three basic steps in the above process.

The software perspective is summarized in Figure 2.

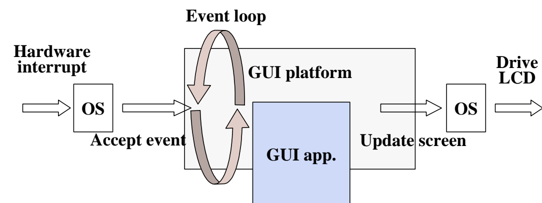

Figure 2: A software perspective for GUI energy consumption.

3.3 Application

From the application perspective, a GUI consumes energy through user-GUI interaction sessions, in which a user locates the application, starts it, interacts with it and finally closes it. Such a process usually consists of a series of window operations, such as creation, switching, and manipulation, *etc.*, with intermittent idle intervals, in which the system waits for user input.

Figure 3 gives the second-by-second energy consumption for two GUI sessions executed on an HP/Compaq iPAQ with Pocket PC 2002. The first session is to look for a file using the file manager. The first peak around four seconds is due to activation of the "Start" menu and stepping through its items to locate "Programs". The second peak at 10 seconds is due to activation of "Programs". The third peak at 12 seconds is due to activation of the file manager in the Programs window. The last peak is due to moving of the scroll bar in the file manager to locate the file in the file list and then closing the file manager. The other session, creating an email, is just to open the "New" menu and create a new email message. The two sessions consume 4.7 and 1.4 Joules more energy, respectively, than consumed in

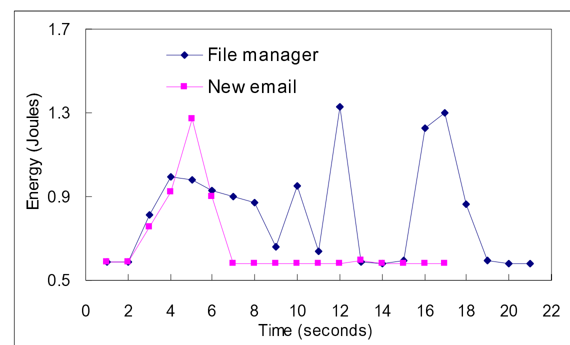

Figure 3: Looking for a file and creating an email.

the corresponding idle period. Being idle for one second consumes 5.9 Joules in these measurements. Creating an email consumes much less extra energy because it requires fewer GUI manipulations. Moreover, since it takes much less time, more email sessions are possible compared to file manager sessions in a fixed amount of battery lifetime. From these two examples, it is obvious that window operations are energy-expensive and an energy-efficient GUI should reduce the number of window operations and usage time.

4. EXPERIMENTAL SETUP

In this section, we first provide information on the handheld computers used and the energy measurement setup, then detail the benchmarks and GUI energy characterization methodology.

4.1 Handheld computer systems studied

The information on the handheld computers used in this work is summarized in Table 2.

Table 2: Hardware and software information on the handheld computers

	iPAQ1	iPAQ2	Zaurus
Vendor	HP/Compaq		Sharp
Model	3870		SL5500
SOC	Intel StrongARM SA-1110 206MHz		
Storage	32MB ROM, 64MB RAM		16MB ROM, 64MB RAM
Display	240×320, 16bit color, reflective with front light		
OS	MS Pocket PC 2002	Familiar Linux 2.4.18	Embedix Linux 2.4.6
GUI	Windows	X/GTK	Qt

The two iPAQs are placed in the same charging cradle when energy is measured. The Zaurus, instead, is directly charged by the same AC/DC converter used for the iPAQs. Energy is measured when the batteries are fully charged. All other detachable peripherals, such as CompactFlash cards, serial and USB ports, *etc.*, are disconnected. Since no current GUI platform has an effect on the front light, it is turned off unless otherwise indicated.

4.2 Energy measurement

The measurement equipment consists of a Windows PC, a hardware data acquisition card, and a wire connector box. The PC is installed with *National Instrument*'s data acquisition software called *Labview*. The voltage is measured across a sense resistor connected in series with the battery to obtain the system power consumption. It is sampled at a rate of $400Hz$. *Labview* is programmed to integrate power every second. By carefully designing the benchmarks, the energy consumption of software events of interest can be captured. The energy measurements for a given benchmark were made on the handheld computers on the same day without interruption (in order to minimize the day-to-day difference due to changes in temperature, slight changes in the resistance of the sense resistor, *etc.*).

4.3 Methodology and benchmarks

While the source code of Qt and X/GTK are freely available, that of Windows is not. However, the GUI APIs of all three platforms are well-documented. The GUI API is an appropriate level of abstraction for modeling GUI platforms. Based on the GUI APIs and platform architectures, we adopt a controlled black-box methodology to characterize different aspects of GUI energy consumption, as explained next.

4.3.1 Additional energy

To account for energy differences due to the OS and hardware, we use the concept of *additional energy*. When a software or hardware event occurs in the system within a certain time interval, the software running in the interval is called a *target*. We design the software to be exactly the same except that it does not trigger that event during the same time interval. The resultant software is called a *context*. The additional energy of the event is defined as the target energy minus the context energy. For every GUI event we characterize, a context is carefully designed. Measurement of a target and corresponding context is repeated in an interleaved way to reduce the impact of random factors.

4.3.2 Normalized energy

We denote the additional energy for performing certain computation-intensive jobs in each system as the *energy unit* (EU). The computation-intensive job we chose was the jpeg_fdct_islow routine in file jfdctint.c from Independent JPEG Group's implementation of JPEG, which comes with the Mibench benchmarks [15]. It performs a forward discrete cosine transform (DCT) on an eight-by-eight block of integers. Three different sets of inputs are randomly chosen from the large image file included in MiBench. The job is memory-intensive as well since input data are read from memory each time before a DCT is performed. To obtain the additional energy for performing one such DCT, we repeat the DCT a total of 3×10^5 times over the set of chosen input data. This is assumed to be the target, which takes our systems about four seconds to complete. The context in this case simply involves making the system idle. The energy of every benchmark is measured with a companion measurement of the EU. In most cases, we report experimental results normalized to the EU thus obtained. This accounts for differences in the hardware and OS. The EU for the three systems we studied is between 8 and 10 μJoules.

The benefits of using the EU are as follows. Experiments were conducted on different days for different benchmarks. The absolute energy figure for an event varied slightly from day to day. However, the energy remained quite constant if normalized to the corresponding EU (within 1%). Moreover, since the EU is only dependent on the SOC and memory, the comparison of non-LCD energy consumption of different systems is fairer after normalization.

4.3.3 Benchmarks

We designed the benchmarks to characterize different aspects of GUI platforms including event handling, typical window operations, and window properties. Also, we characterized common window types and their related usage. Different input methods were also characterized. The benchmarks are described in Table 3. Unless otherwise indicated, they were coded for all three GUI platforms. The coding

Table 3: Benchmarks

Benchmark	Description
Event handling	
Event loop	Send and get an event
System event	Use stylus to tap a window which is programmed to ignore that event
Basic window operations	
Create window	Create and then destroy a window
Show window	Show and then hide a full size window
Using windows of different types	
Menu	Show and then hide a menu window of four items
Message box	Show a dialog box with "OK" and accept user confirmation
Scroll bar	Move a scroll bar of half screen height with various speeds
Tabbed panel	Switch between two full screen tabbed panels (iPAQ1 and Zaurus only)
Window properties (iPAQ1 only)	
Size	Show and then hide menu windows of different numbers of items, normal windows of different sizes (60×80, 90×120, 120×160, 180×240, and 240×320 pixels, respectively), text windows of different text sizes
Colors	Present a full screen window of different colors
Color sequence	Show and then hide full screen windows of different colors on a black background
Color patterns	Present full screen windows of black and white checkerboard patterns with different block sizes
Different user input methods	
Virtual keyboard	Push "x" on the virtual keyboard
Hardware button	Trigger the right cursor key button
Stylus tap	Same as System event
Stylus move	Move stylus vertically along the screen for half screen height on a window which is programmed to ignore corresponding events

and compilation information is given in Table 4. The source code for benchmarks used in this work can be downloaded from [25]. Most benchmarks were coded using a similar scheme in which a timer is set to start some processing. In a target, the processing triggers the event we wish to characterize; in the corresponding context, the processing is the same except that the event is not triggered. A snapshot of the "Event loop" benchmark target for Windows is shown in Figure 4 to illustrate such a scheme. Note that its context uses the same code except that line 13 is commented out.

Table 4: Coding and compilation information for benchmarks

	Qt	Windows	X/GTK
Coding language	C++	C	C
IDE	N/A	eMbedded Visual C++ 3.0	N/A
Compilation setting	gcc -O2	Default	gcc -O2

Routines to turn the LCD off and on were inserted at the beginning and end of the benchmarks to obtain the additional energy due to the LCD.

5. GUI ENERGY CHARACTERIZATION

Experimental results for the energy characterization of the three handheld computers are presented in this section. They are analyzed from the software, hardware, and application perspectives.

5.1 Energy breakdown

Table 5 gives the energy breakdown by hardware for the systems targeted when they are idle and while performing the aforementioned DCT computation. It also gives the one-second energy consumption in EU for different components. The LCD energy is obtained by comparing the system energy before and after the LCD is turned off. Front-light energy is obtained in the same way. "Others" refers to the system energy minus the LCD and front-light energy. It includes the energy consumed by all other hardware. The table shows that the front light and the TFT LCD consume a large fraction of system energy. The percentages are larger than those reported for notebook computers [4], in which a hard-disk, system-on-board and more powerful processor

Table 5: System energy breakdown for the handheld computers

Handheld		LCD %	LCD EU	Front light %	Front light EU	Others %	Others EU	EU μJoule
iPAQ1	idle	9	18,800	73	147,100	18	35,300	8.0
	DCT	7		53		40	112,600	
iPAQ2	idle	14	20,600	53	77,700	33	48,200	9.8
	DCT	9		34		57	129,100	
Zaurus	idle	11	25,200	80	180,000	9	19,000	9.3
	DCT	8		59		33	99,300	

```
1  #define ID_TIMER WM_USER + 300
2  static int count = 0;
3  ......
4  LRESULT CALLBACK WindowFunc(HWND hWnd, UINT message,
5                        WPARAM wParam, LPARAM lParam)
6  {
7     ......
8     switch (message)
9     {
10          case WM_TIMER:
11              if(count<TOTAL_NUM){//Repeat TOTAL_NUM times
12                  //Send a message to myself
13                  SendMessage(hWnd,WM_LBUTTONDOWN,0,0);
14                  count++;
15               } else {
16                  //Request to exit
17                  SendMessage(hWnd,WM_DESTROY,0,0);
18               }
19               break;
20          case WM_LBUTTONDOWN: //Catch the sent message
21               //Do nothing here
22               break;
23          case WM_CREATE:
24               //Set a timer
25               SetTimer(hWnd, ID_TIMER, 2000, NULL);
26               break;
27          case WM_DESTROY:
28               PostQuitMessage(0);
29               break;
30          default:
31               return DefWindowProc(hWnd, message, wParam, lParam);
32     }
33     return 0;
34  }
```

Figure 4: Outline of the target for benchmark "Event loop". The context uses the same code but with line 13 commented out.

are used instead of Flash memory and SOCs. With the front light turned off, the TFT LCD consumes from 14% to 55% of the system energy, depending on how busy the CPU is. However, it should be noted that in interactive application usage, the CPU is idle most of the time.

5.2 Event handling and basic window operations

Table 6 presents the additional energy for GUI event handling and basic window operations in terms of EUs. "Event loop" shows the additional energy for an event to go through the event loop. It is very energy-efficient compared to "System event," which includes additional energy for hardware interrupt, OS and platform event processing. Windows outperforms Qt and X/GTK significantly, which may be attributed to tighter integration of its GUI platform and OS.

"Create window" shows the additional energy for a GUI platform to claim and relinquish resources for a window according to the request from the applicaion. "Show window" shows the additional energy for the GUI platform to show and hide a full screen window according to the request from the application. The window shown is identical to the background window. Therefore, there is no additional energy incurred due to framebuffer updating, LCDC or LCD. The additional energy can only be attributed to changes in the internal data of the GUI platform as in the cases of "Event loop" and "Create window". While Qt and Windows are relatively close, X/GTK performs significantly worse in "Show window". This hints at the overhead of using an X server.

Table 6: Energy characterization in EUs for different GUI platforms

	Qt	Windows	X/GTK
Event handling			
Event loop	27	11	40
System event	1,100	300	900
Basic window operations			
Create window	1,900	2,600	1,000
Show window	9,900	7,900	18,000
Using windows of different types			
Menu	12,800	15,400	6,100
Message box	14,400	6,000	13,300
Scroll bar	33,400∼ 78,000	20,000∼ 59,000	38,800∼ 84,000
Tabbed panel	10,300	12,000	-
Draw text	-	1,600	-

5.3 Window types

Table 6 also shows the additional energy required for using different types of windows and showing an eighty-letter text. It is obvious that using different window types consumes quite different amounts of energy even when the window sizes and colors are similar, as in a message box and a four-item menu window. Different window types necessitate different user interactions too, which further differentiate their energy consumption. There are several observations worth noting, as discussed next.

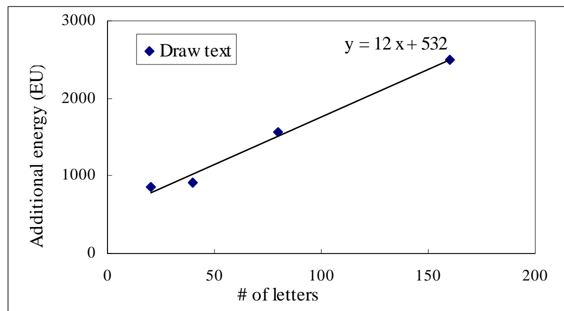

Figure 5: Additional energy for showing texts of different sizes.

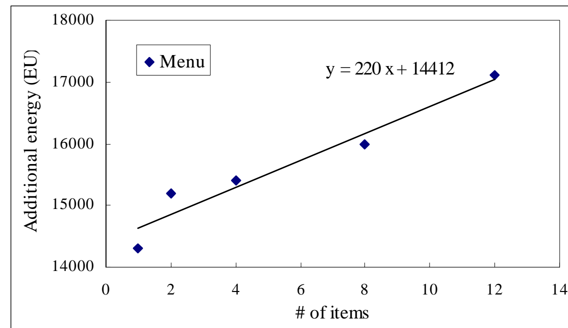

Figure 6: Additional energy for showing menu windows containing different numbers of items.

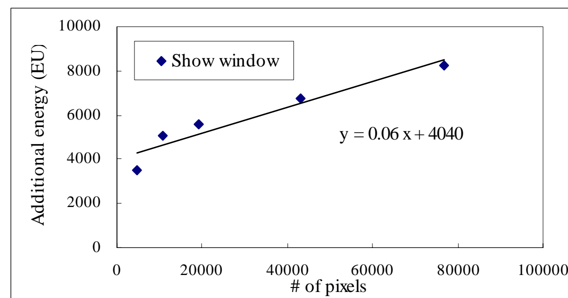

Figure 7: Additional energy for showing windows of different sizes.

First, the same interactive function may be implemented using different window types with different energy efficiencies. For instance, both tabbed panels and a scroll bar can be used to browse a long list. Their energy consumption differs drastically since a scroll bar requires many more screen updates than tabbed panels.

Second, "Message box" of Windows outperforms others simply because it allows a "Message box" to be much simpler than Qt or X/GTK does. For example, "Message box" of Qt has 3D effects and must contain at least one button. This indicates that an inflexibility in the GUI platform can cause extra energy consumption.

Finally, Windows performs much worse for "Menu," although it does well in other window types, because a menu window is animated in Windows. When a menu is tapped, the menu window gradually, but quickly, drops out. Such an animation or continuous screen change requires many more processor cycles, framebuffer updates and screen refreshes, thus leading to more additional energy. Using a scroll bar to browse a window also requires continuous screen changes. For energy efficiency, such continuous screen changes should be avoided, at the expense of a slight sacrifice in GUI aesthetics.

5.4 Window properties

A window usually has properties like size, color and color patterns. We have conducted detailed experiments on iPAQ1 to capture the relationship between these properties and energy consumption.

5.4.1 Size

We have experimented with three different types of sizes. Figures 5-7 show the additional energy for benchmark Size. They also show the equation for the best-fit line, obtained through linear regression. The relationship between the size and additional energy is approximately linear. Each letter consumes about 12 EUs. Each new menu item consumes about 220 EUs. These data suggest that GUI designs should be economical to be energy-efficient. Moreover, the huge constants in the linear regression equations also imply that if a user needs to view a large number of items, putting them in as few windows as possible saves significant energy.

5.4.2 Color

Color affects the additional energy consumption of a GUI in several ways. Our first experiment measures the energy of iPAQ1 when the CPU is idle for one second with screens of different colors. Since each pixel consists of three color components, R, G, and B, we perform measurement for colors containing different combinations of these three components. We also perform measurements with the LCD turned off to obtain the energy consumption of the LCD only. Table 7 summarizes the results under energy consumed by the LCD and the non-LCD energy. It also shows the percentage energy increase compared with pure white. R, G, B, and RG refer to red, green, blue, and orange, respectively. They have R, G, B, and R and G component(s) deactivated, respectively. "Grey" refers to the color obtained when the corresponding originally activated component(s) is (are) half-activated. For example, when all three components are fully activated, the color is "Full" black. When they are half-activated, the color is "Grey" black.

The energy difference disappears after the LCD is turned off, which demonstrates that it is the LCD that makes the difference.

There are two observations one can make. First, the more color components activated, the more the energy consumption. In a reflective TFT LCD, when one color component is deactivated, the corresponding liquid crystals are fully unpolarized, and there is no need to repeatedly charge the polarizers. For instance, "White" has all three color components deactivated and consumes the least energy, while "Black" has all three activated and consumes more energy. This observation is the same as that made in [3] for a transmissive TFT LCD. More interestingly, the second observation is that a half-activated component consumes more energy than an activated component, as the "Grey" ones consume more energy than their "Full" counterparts. As mentioned

Table 7: Energy breakdown for presenting screens of different colors

Color		Non-LCD (EU)	LCD (EU)	% Increase
Black	Full	36,100	18,700	3.3
	Grey	36,100	19,100	5.5
Red	Full	36,100	18,500	2.2
	Grey	36,100	18,700	3.3
Green	Full	36,100	18,500	2.2
	Grey	36,100	18,700	3.3
Blue	Full	36,100	18,600	2.8
	Grey	36,100	18,700	3.3
RG		36,100	18,300	1.1
White		36,100	18,100	0

in Section 3, each color component has a TFT to charge a storage capacitor, which maintains the appropriate voltage between the polarizers. When a component is half-activated, the capacitor puts the TFT into a state that draws a higher amount of current, which contributes to higher energy consumption.

On Zaurus, users can choose different colors for GUI themes. Table 8 gives the system energy for presenting the "Application" window with different colors for the QPE theme for one second. It confirms the observation made in the iPAQ1 experiment. It also shows the percentage system energy increase when compared with that of the "Bright" theme color.

Table 8: Energy of different colors for the QPE theme on Zaurus

Theme color	Energy (Joule)	Over Bright(%)
Bright	0.416	0
Purple	0.417	0.4
Desert	0.422	1.5
Grey	0.423	1.6

5.4.3 Color sequence

As the analysis in Section 3 reveals, temporal changes in the screen cost energy. When the screen changes color, the CPU consumes additional energy to generate data for the framebuffer, the framebuffer has to be then updated, and corresponding liquid crystals have to change orientation. The second experiment was designed to measure the energy of iPAQ1 for changing the screen from black to different colors. To eliminate the additional energy due to the LCD presenting different colors, it is turned off. Therefore, the experiment mainly accounts for the additional energy due to the first two of the three processes mentioned above. The data are presented in Table 9. It demonstrates the more the color changes, in terms of (R,G,B) components, the larger the additional energy consumption. This implies that a GUI with a constant color theme will be more energy-efficient than one that often changes color.

Table 9: Additional energy for showing and hiding windows of different colors on a black background

New color	White	Orange (RG)	Red (R)	Grey	Black
Add. energy (EU)	8,000	7,900	7,500	7,700	7,400

5.4.4 Color patterns

As also evident through the analysis in Section 3, spatial changes within a screen also impact energy consumption. We call how colors are distributed on the screen its *color pattern*. The color pattern determines the spatial changes within the screen. Even if the percentage of pixels of each color remains constant (then the LCD consumes constant energy), different pixel arrangements can introduce different switching activities in the hardware including the LCDC, system bus, and external bus, because data for the screen have to be constantly transferred from the framebuffer sitting in the off-chip memory to the LCDC, and then to the LCD for refreshing the storage capacitors. We measure the energy consumption of the system when the system is idle with full-screen windows of checkerboard patterns on iPAQ1. A checkerboard pattern consists of alternating white and black blocks. For each pattern, the white and black blocks each take up half of the screen pixels so that the LCD energy consumption does not vary. However, as expected, the system energy consumption increases when block size decreases, which leads to more spatial changes within the screen. Table 10 shows the system energy consumption for one second and also gives the percentage energy increase compared with that of presenting a fully white screen. It also shows the system energy for presenting the starting home screen for Pocket PC 2002. The energy difference in Table 10 is due to the showing of the screen only. The screen with smaller blocks takes more CPU time and energy to generate the screen data, which is a separate issue from what we are concerned with here. The above results imply that a plain GUI is more energy-efficient than fancy ones.

Table 10: Different color patterns on iPAQ1

Pattern	Energy (Joule)	Over white (%)
Full white	0.575	0
Full black	0.581	1.0
120×160 block	0.577	0.3
30×40 block	0.584	1.6
12×16 block	0.588	2.3
3×4 block	0.598	4.0
MS home	0.590	2.6

5.5 Input method

From the application perspective, a user interacts with GUIs through different input methods, and uses a certain input method to interact with a certain type of window. Table 11 provides the energy consumption for different input methods on the targeted handheld computers. It also shows the number of stylus taps an input method is equal to (denoted by #) with regard to energy consumption. The stylus tap is the most commonly used input method. Hardware buttons are used to trigger the most-used applications. Virtual keyboard is necessary for text input (Zaurus also comes with a mini hardware keyboard we do not characterize in this work). Stylus move is typically used to move a window like a scroll bar.

A stylus move is very expensive in all three systems. Moreover, it is usually associated with continuous screen changes such as window moving and resizing, for which the energy cost is significant. Moreover, a virtual keyboard is very expensive on iPAQ1 and Zaurus because they have the auto-completion feature in order to accelerate user input. Since reducing usage time saves a significant amount of energy, auto-completion is generally more energy-efficient. However, inputting text is much slower using a virtual keyboard than a real keyboard. Thus, it increases usage time significantly on handheld computers, and leads to more energy consumption from the application perspective. For energy efficiency, stylus move and text input should be minimized.

5.6 GUI platform comparison

In general, all three GUI platforms characterized in this work perform similarly with regard to their energy efficiency. All are descendants of GUI platforms used in non-energy critical computers. None of them is designed with energy efficiency and awareness as a goal. Each contains areas for improvement. Windows performs better in event and interrupt handling due to its tighter integration with the OS, but suffers due to its desktop lineage. X/GTK is better in providing most implementation flexibility for designing an energy-efficient GUI, but suffers from using an X server and lack of enough features. Qt offers more features for a fancy user interface, but suffers from limited flexibility for making a GUI simpler. Moreover, since each of them only works with a unique OS and requires a different license, a platform choice is further complicated. However, the energy characterization of GUI platforms presented here should help GUI designers make a better decision and help GUI platform/toolkit developers improve their work with regard to energy consumption.

6. GUI DESIGN FOR ENERGY EFFICIENCY AND AWARENESS

As the results presented earlier show, the energy consumption of a GUI is significant. In view of the ubiquity of GUIs, it is extremely important to optimize them when energy consumption is of concern.

6.1 Achieving energy-efficient GUI designs

Experiences for designing GUIs have been accumulated over a long time without addressing energy efficiency. Moreover, the importance of these experiences changes from desktop GUIs to handheld GUIs. Based on the GUI energy characterization presented in this work, we offer some insights into energy-efficient GUI design, and compare them with the traditional do's and don'ts for GUI design [12].

6.1.1 Accelerate user interaction

Since even leaving the system idle consumes a lot of energy, reducing the usage time for a task seems to be the most effective way of energy reduction. Therefore, a GUI should be designed for maximum work within a given amount of time. For example, the most frequently used programs or functions should be placed in the GUI so that users can find them quickly instead of after activating a series of windows. Fortunately, fast access to functionalities is also a primary concern for traditional GUI design [12] (pp. 62-78). Note that user interaction acceleration is different from improving GUI responsiveness, another primary concern of traditional GUI design for performing screen changes.

6.1.2 Minimize screen changes

There are several ways for minimizing screen changes, some of which agree with the traditional methods for GUI design to preserve display inertia. For example, GUI changes may be cached. That is, consecutive changes taking place within a short time interval can be presented as one single change without affecting user interaction. This also means that continuous screen changes as animation and window scrolling should be avoided, which disagrees with the traditional method for providing visual continuity [12] (p. 282).

6.1.3 Avoid or minimize text input

Traditional wisdom [12] (pp. 121-128) makes this recommendation to minimize user switching between a mouse and keyboard. This is even more important from the energy point of view for handheld computers since text input is much slower for them. For example, if the range of inputs is known beforehand, a list can be supplied to ask users to choose, instead of type in, text.

6.1.4 Reduce redundancy

This both disagrees and agrees with traditional wisdom. Traditional wisdom asks for an animated progress indicator to improve software responsiveness while a simple busy indicator may be more energy-efficient. Traditional wisdom also asks for consistent menus which keep useless menu items on the menu window, but deactivate them, in order to improve user friendliness [12] (p. 62). However, such items may be removed to save energy once a user has become familiar with the software. To summarize, features that do not accelerate usage should be avoided. This does agree with traditional wisdom on task queue optimization for ignoring outdated user requests [12] (p. 397).

6.1.5 Do something while waiting for user input

Since being idle consumes a lot of energy, a more aggressive way to accelerate user interaction is to speculate on what may be the user input and get the result ready before the next input is provided. For example, auto-completion in the virtual keyboard is one way of speculating. Moreover, tasks can be reordered to first present those windows on the screen that ask for user input, and then go ahead with processing of other tasks. This idea coincides with the traditional wisdom of dynamic time management [12]. However, it is used for a different purpose here.

One can also deduce advice for implementing GUIs. First,

Table 11: Additional energy for different input methods

Input method	Qt		Windows		X/GTK	
	EU	#	EU	#	EU	#
Stylus tap	1,100	1	300	1	900	1
Hardware button	1,400	1.3	560	1.9	1,300	1.4
Virtual keyboard	4,000	3.6	4,700	15.7	1,200	1.3
Stylus move	~13,500	~12.3	~5,700	~19.0	~3,000	~3.3

being economical and terse is important, as we have shown that the window size and text size do matter in terms of energy. Second, choosing colors with fewer color components is beneficial for TFT LCDs as they consume less energy. Colors with partially activated component(s) should be avoided. Color changes from window to window and fine color patterns or highly decorated windows should also be avoided, and plain windows should be preferred.

6.2 Energy-aware GUI platform and GUI

While energy-efficient GUIs use minimal energy for certain functions, energy-aware GUIs should also be able to trade other aspects of the software for saving energy.

As shown in Section 5, the display is a large source of energy consumption. Thus, GUI platforms can be augmented to capitalize on new display management technologies [3, 9, 21].

As some of the insights offered for improving energy efficiency contradict traditional wisdom, energy efficiency may sometimes be improved only with some sacrifice in another area. Not showing progress bars may be somewhat awkward. Inconsistent menus may be slightly confusing. A plain theme may make users unhappy. Therefore, instead of using an energy-efficient GUI all the time, a GUI should be able to adapt to energy availability to make the best tradeoff. Moreover, users should be able to decide whether and when to use the energy-efficient GUI, thus overriding the automatic adaptation.

7. CONCLUSIONS

GUIs are ubiquitous on handheld computers. In this work, we presented the first study to characterize the energy consumption of GUIs implemented on three different GUI platforms. We analyzed the GUI energy consumption from the hardware, software and application perspectives. Although our characterization is not exhaustive, our methodology can be used for more comprehensive studies. We demonstrated that a GUI consumes a significant amount of energy and showed that there are a number of ways to make it more energy-efficient and energy-aware. This provides a solid foundation for further research on energy-aware and energy-efficient GUI and GUI platform design.

When multiple windows are displayed on the screen simultaneously and used in turn, it is important to update the screen so that the appropriate window is active. This task is called window management. On handheld computers, it is rare to use multiple windows due to the limited screen size. Thus, window management is not important for such computers and, therefore, we did not address it in this study. However, for systems with larger screens, such as notebook computers, window management will be important for a GUI's energy efficiency since it determines how the screen is updated when the user focus changes.

8. REFERENCES

[1] *Proc. Int. Symp. Human Computer Interaction with Mobile Devices and Services*, 1999-2003.

[2] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of three embedded real-time operating systems. In *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, pages 203–210, Nov. 2001.

[3] I. Choi, H. Shim, and N. Chang. Low-power color TFT LCD display for handheld embedded systems. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 112–117, Aug. 2002.

[4] D. Chung. Mobile platform display technology advancements. *Intel Developer Forum*, Sept. 2002.

[5] T. L. Cignetti, K. Komarov, and C. S. Ellis. Energy estimation tools for the Palm. In *Proc. ACM MSWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Aug. 2000.

[6] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Proc. Design Automation Conf.*, pages 312–315, June 2000.

[7] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proc. ACM SIGMETRICS: Int. Conf. Measurement & Modeling of Computer Systems*, pages 252–263, June 2000.

[8] J. Flinn, K. I. Farkas, and J. Anderson. Power and energy characterization of the Itsy pocket computer (version 1.5). Technical Report Technical Note TN-56, Compaq Western Research Laboratory, Feb. 2000.

[9] F. Gatti, A. Acquaviva, L. Benini, and B. Ricco. Low power control techniques for TFT LCD displays. In *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, pages 218–224, Oct. 2002.

[10] GIMP toolkit. *http://www.gtk.org*.

[11] GPE Palmtop environment. *http://gpe.handhelds.org*.

[12] J. Johnson. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers.* Morgan Kaufmann, San Francisco, CA, 2000.

[13] K. Li, R. Kumpf, P. Horton, and T. E. Anderson. A quantitative analysis of disk drive power management

in portable computers. In *Proc. USENIX Winter*, pages 279–291, 1994.

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

[15] MiBench. *http://www.eecs.umich.edu/mibench/*.

[16] B. A. Myers. User interface software tools. *ACM Trans. Computer-Human Interaction*, 2(1):64–103, Mar. 1995.

[17] B. A. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Computer-Human Interaction*, 7(1):3–28, Mar. 2000.

[18] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proc. ACM Conf. Human Factors in Computing Systems*, pages 195–202, May 1992.

[19] Opie. *http://opie.handhelds.org*.

[20] Palm OS development. *http://www.palmsource.com/developers/*.

[21] Philips' display driver solutions for mobile applications. *http://www.semiconductors.philips.com/acrobat/literature/9397/75009652.pdf*.

[22] Qt/Embedded. *http://www.trolltech.com/download/qt/embedded.html*.

[23] T. Simunic, G. De Micheli, L. Benini, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proc. Int. Symp. System Synthesis*, pages 193–199, Sept. 2000.

[24] A. Sinha and A. Chandrakasan. JouleTrack - A web based tool for software energy profiling. In *Proc. Design Automation Conf.*, pages 220–225, June 2001.

[25] Source code for benchmarks. *http://www.ee.princeton.edu/∼cad/benchmarks.html*.

[26] Special issue on human-computer interaction with mobile systems. *ACM Trans. Computer-Human Interaction*, 7(3), Sept. 2000.

[27] T. K. Tan, A. Raghunathan, and N. K. Jha. EMSIM: An energy simulation framework for an embedded operating system. In *Proc. Int. Symp. Circuits & Systems*, pages 464–467, May 2002.

[28] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level software energy macro-modeling. In *Proc. Design Automation Conf.*, pages 605–610, June 2001.

[29] The Embedded Linux GUI/ Windowing Quick Reference Guide. *http://www.linuxdevices.com/articles/AT9202043619.html*.

[30] The Familiar project. *http://familiar.handhelds.org*.

[31] The Skiff cluster. *http://www.handhelds.org/projects/skiffcluster.html*.

[32] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. VLSI Systems*, 2(4):437–445, Dec. 1994.

[33] X Window System. *http://www.x.org*.