A Multi-Granularity Energy Profiling Approach and a Quantitative Study of a Web Browser[*]

Chen-Ting Chuang[1], Chin-Fu Kuo[1], Tei-Wei Kuo[1+] and Ai-Chun Pang[1+]
[1]Department of Computer Science and Information Engineering
[+]Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106, ROC
Email: {d89005,ktw,acpang}@csie.ntu.edu.tw, Fax:+886-223628167

Abstract

While energy-efficiency considerations become a critical issue in embedded-system designs, little work is done for energy-profiling at different granularity levels. This paper aims at the proposing of a hybrid hardware-software-based profiling solution with different levels of abstraction. Compared to the existing work, the proposed solution could provide richer information on energy profiling with the help of execution-path tracking and exclusion of selected functions/classes/modules. We address technical issues in the implementation of the proposed profiling solution. A quantitative energy-profiling study over a well-known web browser based on an object-oriented design, Konqueror, is then presented to provide a feasibility study of the proposed solution and insights in the design of an energy-efficient web browser.

1 Introduction

The number of transistors on microprocessor chips has been doubling around every 18 months. As the announcements of powerful microprocessor chips keep appearing basically in a non-stopped fashion, the energy dissipation of microprocessor chips also exhibits a trend similar to the performance acceleration of microprocessors predicted by the Moore's Law [6]. While the statement really depends on the advance of the voltage scaling technology, the energy dissipation might increase from 100 watts in 2000 to a number between 2,000 watts and 10,000 watts in 2010 if the trend continues. As a result, techniques in the reduction of energy dissipation become a very important issue, not only for embedded systems but also for desktops and enterprise-class servers.

Software technology for efficient energy management is critical in extending battery lifetime and improving user experiences. There is never any good substitute for intelligent energy management, where energy profiling is a key methodology in software designs for intelligent energy management such that the behaviors of program executions and their users are better understood. Existing energy profiling tools can be roughly classified into simulation-based and monitoring-based profiling. Simulation-based energy profiling [9, 12, 13] is often time-consuming. The accuracy of architecture-level simulation depends on how detailed the system is modelled. While Tiwari, et al. presented instruction-level energy models [13], Brooks, et al. [12] proposed architecture-level power simulators over various platforms with an accuracy level down to cycles. Besides, the energy consumption of an operating system or a wireless LAN was also studied in the literature, e.g., [9]. Monitoring-based energy profiling [8, 11] is usually a preferable way in the energy consumption analysis of real-life applications because it is straightforward and accurate. In particular, Flinn, et al. [8] proposed an energy profiling tool to periodically sample the values of the program counter. Shin, et al. [11] proposed a hardware-based energy profiling tool with onboard profile acquisition circuits to achieve a high sampling rate. For profiling with compiler supports, special (compilation) configurations are usually needed. In particular, a technique called *path profiling* was proposed to maintain the frequency of each execution path [5]. The average profiling overheads could be close to 31%. Profiling could also be done inside the kernel, in terms of some special-purpose hardware, or with a hardware simulator. However, hardware-based profiling solutions are often less flexible, due to the difficulties in the examination of in-core data structures.

While many energy profiling tools usually find code in the system libraries as hot spots, little information on energy consumption was obtained from the view point of the user code and application libraries or even the design of web pages, especially when modifications to system libraries are beyond the capability of many engineers. Many conventional tools for monitoring-based energy profiling merely access the program counter to track the context of a program's execution and cannot provide the information of the execution paths with reasonable overheads. In this paper, we propose a hybrid hardware-software-based energy profiling solution, which could be implemented with little cost but high accuracy. Compared to the existing work, the proposed solution could provide richer information on energy profiling. We propose an execution-path-tracking approach to provide multiple granularity levels in the observation of the energy consumption of a program execution. We address technical issues in the design and implementation of the proposed profiling solution. A quantitative energy-profiling study over a well-known web browser Konqueror is then presented to provide a feasibility study of the proposed solution, where Konqueror is recently adopted by Apple Computer, Inc. as the

[*]This research was supported in part by the National Science Council under grant NSC-92-2213-E-002-091.

base for their web browser [2]. With the help of execution-path tracking and exclusion of selected functions/classes/modules, insights could be obtained based on observations on energy profiling of functions or classes at different levels of abstraction. It helps in the identification of program components that need better improvement, e.g., encoding conversion, and the problems in the designs of web pages, such as those with a heavy usage of JavaScript code.

The rest of this paper is organized as follows. Section 2 underlies the motivation of this research. Section 3 proposes the execution-path-tracking approach and a hybrid hardware-software-based profiling solution. Section 4 presents the energy-profiling tool and a quantitative study over Konqueror. Section 5 is the conclusion.

2 Motivation

Energy profiling provides useful information in software designs for intelligent energy management. Existing energy profiling tools can be roughly classified into two categories: simulation-based and monitoring-based profiling. Simulation-based energy profiling [9, 12, 13] is often time-consuming. The accuracy of architecture-level simulation depends on how detailed the system is modelled, e.g., those done all the way to the circuit and gate level. Although simulation-based energy profiling is, in general, good for hardware developers with complete knowledge of hardware designs, it often has difficulties in the accurate estimation of the energy consumption of operating systems in real workloads. While the time needed for simulation-based energy profiling is usually lengthy and unpredictable, simulation-based energy profiling is better for workloads on batch processing, instead of interactive applications (that involve with the reactions from users and other parties directly or remotely).

Monitoring-based energy profiling [8, 11] is usually a preferable way in the energy consumption analysis of real-life applications because it is straightforward and accurate. Monitoring-based energy profiling periodically profiles the program execution context and the energy consumption data with a digital multimeter or on-board measurement circuits. The accuracy and system overheads are closely related to the sampling frequency. Although hardware solutions, such as onboard profile acquisition circuits [11], could achieve a high sampling frequency with limited overheads, they are often not available to many developers. Furthermore, hardware-based profiling solutions could hardly examine in-core data structures. On the other hand, software-oriented solutions, such as interrupt-handler-based sampling plus a multimeter [8], usually has high overheads and large sampling jitter. In addition to that, executions inside critical sections (that are between the entry and exit sections) of the kernel could not be observed such that the profiling accuracy degrades [4].

Many existing tools for monitoring-based energy profiling merely access the program counter to track the context of a program's execution. Although they are useful in the optimization of programs with "fixed" execution flows (such as MPEG players) by locating (energy-consuming) hot spots, they usually fall short in providing enough information for the analysis of more complex applications, that involve interactivities among program components, system/user processes, etc. Figure 1 shows

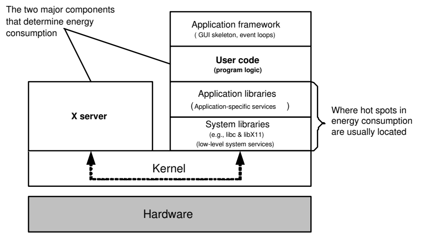

The two major components that determine energy consumption
Application framework
(GUI skeleton, event loops)
User code
(program logic)
X server
Application libraries
(Application-specific services)
System libraries
(e.g., libc & libX11)
(low-level system services)
Where hot spots in energy consumption are usually located
Kernel
Hardware

Figure 1. A typical framework of a GUI application

a typical execution framework of a GUI application, such as a browser. The execution context of a GUI application often consists of the application framework (e.g., GUI skeleton), the user code, related application libraries (e.g., customized application development procedures), and related system libraries (e.g., libc). Existing energy profiling usually finds code in the system libraries as hot spots in energy consumption. While most researchers would consider energy optimization on system libraries, we must point out that the energy optimization from the view point of the user code and application libraries is also highly critical, especially when modifications to system libraries are beyond the capability of application developers.

The above observations underly the motivation of this research: The objective of this research is to propose a hybrid hardware-software-based solution with multiple granularity levels in the observation of the energy consumption of a program execution. We propose to examine stack frames of application executions to track the energy consumption of each individual procedure of user programs and the kernel and their relationship. It is to have a good profiling accuracy but with limited system overheads. The feasibility of this work is demonstrated by an energy profiling study on a popular web browser.

3 Hybrid Monitor-Based Energy Profiling - A Multi-Granularity Approach

The purpose of this section is to propose a hybrid monitor-based energy profiling approach based on the tracking of execution paths. It is to provide better observations of the energy consumption of a program execution in terms of modules, functions, or even classes at different granularity levels. We first address technical problems and issues in the scanning of user-mode and kernel-mode stacks, especially on efficiency considerations and the function invocation relationship. We then present a hybrid monitor-based energy profiling approach and address its design issues. The physical constraints in terms of the rebuilding of the function invocation relationship and the translation of instruction addresses for the corresponding functions are presented. We will then address the clock synchronization issue between the profiled computer system and the measuring computer system. The implementation of the hybrid monitor-based energy profiling tool is presented in Section 4.

3.1 Tracking of Execution Paths

The energy profiling of an application program could be better understood in terms of modules, functions, or even classes at different granularity levels. The achievement of such multi-granularity observations could be done by tracking the execution path of the program. We shall first use a snapshot of the execution of a Konqueror process[1] over Linux 2.4.18 to illustrate the memory layout and stacks of a process:

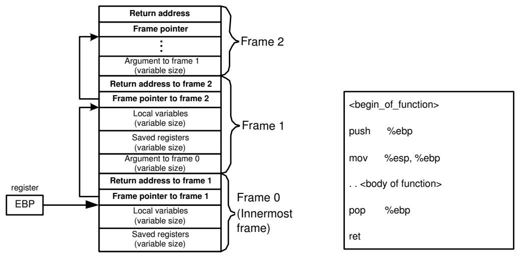
Linux reserves the high 1 GB of the memory address space (beginning at 0xC0000000) for privileged kernel data structures[2]. Program executables and libraries are typically stored in the executable and linking format (ELF) [14]. The user-mode memory of a process, which consists of a stack, heap, text (program code), data (initialized data), and bss (uninitialized data), is located at the low 3 GB of the address space. Shared libraries are mmap()-ed into the user-mode memory address space at the run time. Each process has a user-mode stack and a kernel-mode stack, where the user-mode stack contains the current execution path of a process in terms of stack frames[3] (with caller and callee function addresses). A kernel-mode stack is created for each process to keep the execution context of the process, where the stack space of interrupts stays. Note that the relocatable addresses might have different values for different runs.

We propose to scan process stacks to have an energy profiling of program components through the tracking of the execution path of a program so that multi-granularity observations could be obtained. The technical problems are mainly on (1) how to correctly identify the function invocation relationship in and between (user-mode and kernel-mode) stacks, (2) how to efficiently scan stack frames, and (3) how to reduce the space needed for profiling.

Depth	Function	Module
0	unix_stream_sendmsg	(kernel)
1	sock_sendmsg	(kernel)
2	sock_write	(kernel)
3	sys_write	(kernel)
4	system_call	(kernel)
5	__libc_write	libc
6	_X11TransSocketWrite	libX11
7	_X11TransWrite	libX11
8	_XFlushInt	libX11
9	_XFlush	libX11
10	XSelectInput	libX11
11	QWidget::setMouseTracking	libqt
12	QWidget::create	libqt
13	QWidget::QWidget	libqt
14	QFrame::QFrame	libqt
...	...	...

Table 1. An example execution path

The tracking of the execution path depends on the correct identification of the function invocation relationship in and between (user-mode and kernel-mode) stacks. Table 1 shows an example execution path in which a C++ class `QWidget` in the libqt library invokes an X11 API `XSelectInput()` to set the mouse tracking. It consequently invokes `write()` in the C library to send data through a Unix domain socket to the X server. Since the code generated by gcc keeps the frame pointer in the `EBP` register[4], as shown in Figure 2.(b), the frame pointer in the innermost stack frame can be accessed through the `EBP` register, as shown in Figure 2.(a). We can trace the stack frames all the way back to the stack frame of `main()`. Three technical


Return address
Frame pointer
Argument to frame 1 (variable size)
Frame 2
Return address to frame 2
Frame pointer to frame 2
Local variables (variable size)
Saved registers (variable size)
Argument to frame 0 (variable size)
Frame 1
Return address to frame 1
Frame pointer to frame 1
Local variables (variable size)
Saved registers (variable size)
Frame 0 (Innermost frame)
register
EBP

```
<begin_of_function>

push    %ebp

mov     %esp, %ebp

. . <body of function>

pop     %ebp

ret
```

(a) The structure of stack frames

(b) The code that maintains the frame pointers

Figure 2. The structure of stack frames and the code that maintains the frame pointers

issues must be addressed so as to correctly identify the function invocation relationship: (1) the validity of the returned address in the stack frame (due to interrupts over frame construction), (2) the validity of the relationship of neighboring stack frames (due to exception handling, e.g., signals), and (3) the potential relationship of stack frames in the user-mode and kernel-mode stacks (due to system calls). Since the text segment of a process might scatter over a wide range of address space, we propose to resolve these issues by maintaining the address ranges of text sections of the profiled programs with an efficient lookup data structure, such as a binary tree.

The efficiency in the scanning of stacks is of paramount importance for the feasibility of execution-path tracking because the number of stack frames is often a big number. For example, the average depth of the execution path of a Konqueror process is between 30 and 40 in our performance evaluation. We propose to maintain a cache of the most recent execution path of each profiled process. When a new sampling starts, the user-mode stack is compared to the cached data first. The scanning of the stack frames first begins with the frame at the bottom of the stack and skip over stack frames which remain unchanged by comparing the stack contents with the cached frame pointers (since the last scanning). When the first changed frame is identified, the stack is scanned from the top of the stack such that new frame pointers are located and cached for later sampling. We must point out that the proposed approach is very effective because most of the contents of a user-mode stack remains unchanged for an extensive period of time. The saving on the profiling space could be achieved at the same time by merely maintaining differences of execution paths between samplings. Note that profiling data could easily go up to several megabytes in tens of seconds without the above caching approach.

3.2 Hybrid Hardware-Software-Based Profiling

The purpose of this section is to present a hybrid hardware-software-based profiling approach to exploit the execution path of an application program. The goal is to have a good sampling frequency but without much overheads and impacts on the program execution.

A profiling system could be partitioned into two major components residing at the *profiled computer system* and the *measuring computer system*, as shown in Figure 3: The objective of the component residing at the profiled computer system is on

[1]When Konqueror loads a web page, some Kioslave processes are created to do network file handling, beside the so-called Konqueror process.

[2]The size of the kernel address space might be different, depending on the system configuration.

[3]The absolute address could be different for different programs under different configurations.

[4]The `-fomit-frame-pointer` option of gcc disables the compilation of frame pointers. The Linux kernel turns on this option by default. If a procedure is not compiled with frame pointers, we should scan the stack to locate returned addresses directly.

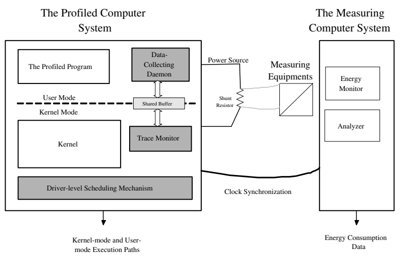


Figure 3. The system architecture of a hybrid hardware-software-based profiling approach

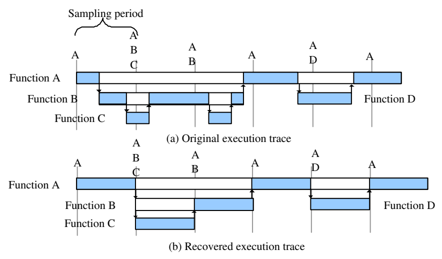


Figure 4. The recovered workload from the samples of execution paths

the trace collection, and the system might consist of three major modules: *data collecting daemon*, *trace monitor*, and the *driver-level scheduling mechanism*. The trace monitor is responsible for the scanning of user-mode and kernel-mode stacks of processes under monitoring and leaves the trace data at the shared buffer for the data collecting daemon. Independent from the trace collection strategy, the data collecting daemon could use the OS services and choose whatever way better for the saving of the trace data, e.g., dumping of trace data into disks or passing them over a wire to its counterpart at the measuring computer system. The driver-level scheduling mechanism should be a lower-level implementation with low overheads and little impacts on application executions, such as RTAI or RTX, to periodically dispatch of the trace monitor.

Workload	read() w/o disk I/O	connect() localhost	pipe()	select() 2 fd	select() 512 fd	page fault of file mapping
Time(μs)	0.7	26.9	5.7	1.8	60.3	12000

Table 2. The execution times of some system calls and system events, where the execution time of read() here does not include the time for disk I/O.

The design issues of the components residing at the profiled computer system include the huge amount of trace data, the efficiency in the stack scanning, the quick identification of the function relationship, the selection of a proper sampling frequency, etc. Beside the discussions of the first three issues in the previous section, the selection of a proper sampling frequency is not an easy problem. While the accuracy of profiling highly depends on the sampling rate, it is not so obvious that profiling has its limitations, due to a small duration time in executing a function or the profiling methodology. The time needed for the processing of a system call or a system event of Linux over an Intel PIII 733 MHz machine, as shown in Table 2, serves as a very good example in revealing some constraints in profiling[5]. Hence, to increase profiling accuracy, a proper sampling period should be less than the execution times of most system calls and system events, such as 50 μs.

Program-counter-based profiling could hardly restore the execution paths of an application program properly, especially when the sampling frequency is not high enough. As shown in Figure 4, the recovered execution times of functions of an application program in terms of program-counter profiling could be a very rough approximation of their counterparts in the original execution time. Although the execution-path tracking approach proposed in this paper could correctly build up the execution paths for an application execution (with the scanning of stack frames), a small duration time in the execution of a function still imposes a physical constraint on the inaccuracy of profiling (because we might hardly hit functions with small execution times during profiling). An excellent performance study in [10] provides a good evidence in the justification of the above argument, and it indicates that the number of machine instructions per function in GUI applications over Windows NT is often less than 100, where an ordinary Intel PIII 733 MHz machine could easily run hundreds of millions of instructions per second. In other words, unless an ultra high sampling frequency is adopted (of course with a formidable overheads), it is always difficult to have an accurate profile even with stack scanning.

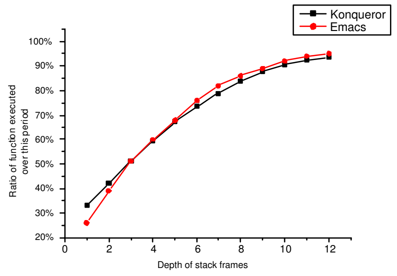


Figure 5. The average depth of a user-mode stack (the sampling period = 50 μs).

Despite the discouraging observations, we must point out that the execution-path tracking approach proposed in this paper could still obtain a good precision in the energy profiling of functions, especially from the program structure's point of view. As shown in Figure 5, even though function invocations corresponding to the innermost stack frames are hard to trace, function invocations corresponding to the outer stack frames could be identified without much difficulty. We could still have a good picture of the energy profiling for an application with a reasonable granularity.

The component residing at the measuring computer system consists of two major modules: an *energy monitor* and an *analyzer*. The energy monitor controls measuring equipments di-

[5]We can trace the processing/invocations of system calls and system events by inserting trace instrumentation into the kernel [15]. However, it is not the focus of the paper.

rectly connected to the profiled computer system. Note that the controlling of measuring equipments should not be at the profiled computer system because the controlling itself still consumes energy. The analyzer analyzes the collected trace data and charges the energy consumption to each individual process, code module, procedure, and kernel service, etc. The instruction addresses of execution paths observed during the data collection time must be translated into the corresponding function names in a high-level language, e.g., C or C++. It could be done by analyzing the relocation addresses of executables and shared libraries to identify the residing module of each given instruction address. The symbol table of related object files is the key to find out the corresponding function of a given instruction address.

3.3 Implementation Remarks

Given instruction addresses of an execution path, the analyzer of the measuring computer system must find out the function names corresponding to the addresses to derive the energy profile of functions. Because every program executable and shared libraries in an operating system, such as Linux, often contain a symbol table for symbol information in the text, data, and bss segments, the mapping of functions and addresses could be found by striping out the symbol table from their corresponding executable or library. The raw data of an executable or a library could also be accessed directly or accessed through utilities, such as `nm(1)` and `objdump(1)` in Unix, to access the corresponding symbol table. Note that shared libraries could be dynamically mmap()-ed into a memory address space, such that their relocation addresses must be taken into account when symbol tables are referred. Furthermore, C++ compilers decorate the symbol names of C++ functions, known as *name mangling*, in order to support function overloading. As a result, de-mangling is needed for the lookup of function names that correspond to the profiled instruction addresses.

Another implementation issue for energy profiling is clock synchronization between the profiled computer system and the measuring computer system, where no system clock is shared. As shown in Figure 3, clock synchronization is simply achieved through a standard RS232 serial interface or a parallel port. Clock synchronization could often be better achieved with a parallel port because the triggering signal port of the measuring equipments (i.e., the general purposed I/O pin) can be directly connected to a parallel port of the profiled computer system. Trigger signals could be sent from the timing circuits of the measuring equipments. However, parallel ports are often not widely available for many embedded systems. An RS232 serial interface could be adopted for clock synchronization instead because of its popularity.

4 Case Study - Energy-Profiling of a Browser

In this section, we shall present an energy profiling tool based on the proposed hybrid approach and address related implementation issues. We will then demonstrate the feasibility of the approach by an energy profiling of a well-known web browser Konqueror and provide insights to the design of a more energy-efficient web browser.

4.1 Energy Profiling and Analysis Toolkit

A hybrid monitor-based profiling toolkit called *Energy Profiling and Analysis Toolkit* (EPAT) is built to demonstrate the feasibility of the proposed approach. The profiled computer system is built on an x86 machine with an Intel Pentium III 733 MHz CPU. From Table 2, we observe that the execution time of many system calls is relatively smaller than 50 μs. Hence, the sampling period is set as 50 μs (i.e., the sampling rate is $20,000$ samplings/second). Note that the sampling rate is high enough to get accurate energy profiling. Real-time Application Interface (RTAI), that was developed by Mantegazza, at al. at DIAPM [7], is adopted as the driver-level scheduling mechanism, as shown in Figure 3. It is because RTAI is less intrusive to Linux and has low run-time overheads and short interrupt latency. The trace monitor is implemented as a periodic RTAI task, and the data collecting daemon is a Linux task. While RTAI could virtually interrupt Linux at any time, the stacks examined by the trace monitor might include inconsistent contents because the writing of a stack frame might be interrupted. The trace monitor should skip over any inconsistent stack frame, that is at the top of a stack, as discussed in the previous section. Since RTAI is light-weighted in the implementation, a very small jitter in the sampling is possible. The access of the data structures of an executing Linux task could be done through the global variable `linux_task`. Because a one-shot timer is adopted in the EPAT implementation, there would be a timer-drifting problem for EPAT although it could be a minor issue in the profiling of many user applications.

A digital multimeter, such as an NI 6036E DAQ card [3], could be used to record digitalized current/voltage samples, where an NI 6036E DAQ card could reach 200,000 Hz in the sampling rate. A digital multimeter is installed at the measuring computer system to avoid the consumption of any energy for the profiled computer system. A digital multimeter measures the differential voltage of a shunt resistor between the power supply and the motherboard, where the resistance of a shunt resistor should be chosen carefully to avoid any distortion of the input voltage for the system[6].

The synchronization between the measuring computer system and the profiled computer system is done by an RS232 serial interface. When an energy profile starts, the measuring computer system sends a notification signal over the RS232 cable to the profiled computer system. It should be noted that serial communication over an RS232 cable could suffer from tens of microseconds in communication delay. For example, consider an RS232 serial interface with the communication mode being N51: Let the baud rate be 115200 bps, and the RS232 FIFO-queue length be 1. The delivery delay for a 7-bit notification signal is at least 7 $bits/(115200 bits/sec)$ = 61 μs. The delivery of a serial-port interrupt in RTAI could introduce another 20 μs delay over many i386 platforms. The total delay could be roughly up to 80 μs. Suppose that the sampling period is 50 μs, i.e., 20,000 samplings/second. The first two samplings on the profiled computer system should be ignored in an energy profiling, due to the delivery delay.

4.2 Energy Profiling of a Web Browser

Since a web browser has been a necessity for many mobile systems, a good low-power design of a web browser becomes an

[6]Power supplies that comply with the standard ATX specification provide three output voltages: 3.3V, 5V and 12V. Which input voltage should be used to drive the CPU depends on the design of the motherboards.

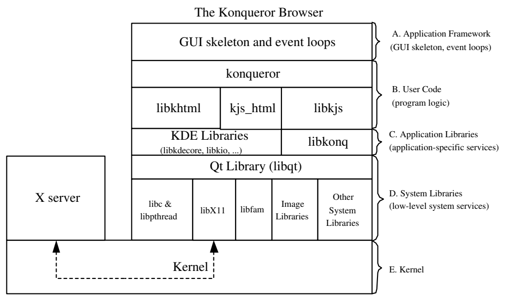

The Konqueror Browser
GUI skeleton and event loops
konqueror
libkhtml
kjs_html
libkjs
KDE Libraries
(libkdecore, libkio, ...)
libkonq
Qt Library (libqt)
X server
libc & libpthread
libX11
libfam
Image Libraries
Other System Libraries
Kernel
A. Application Framework (GUI skeleton, event loops)
B. User Code (program logic)
C. Application Libraries (application-specific services)
D. System Libraries (low-level system services)
E. Kernel

Figure 6. The system architecture of Konqueror

interesting problem. Konqueror [2], an open source browser that is recently adopted by Apple Computer, Inc. as the base for her web browser Safari [1] and is programmed in the C++ language, is profiled to provide references for the energy consumption of a web browser. With EPAT, we could provide an energy consumption analysis for browser functions, components, or even classes at different granularity levels. The results would help in providing more insights to the design of a more energy-efficient web browser.

4.2.1 Konqueror: A Web Browser Example

Konqueror, that is a part of a project for the KDE desktop environment, is an open-source web browser over the Qt application environment for Linux and is programmed in the C++ language. Compared to a well-known browser Mozilla (that has millions of lines of code), the core of Konqueror has less than 200,000 lines of codes [2], and it provides a good set of features for energy profiling, e.g., CSS1, CSS2 (mostly), JavaScript, Java Applet, Flash, SSL, etc. The system architecture of Konqueror is as shown in Figure 6 (similar to that in Figure 1). A summary of her major libraries and code modules is listed in Table 3.

Library/Modules	Description
konqueror	The core of Konqueror
libkhtml	HTML core library
libkjs	JavaScript support
kjs_html	JavaScript language bindings for KHTML
libkonq	The basic services of Konqueror, such as bookmark management, file operations, browser configuration, etc.
libkdecore IPC,	The KDE core library that provides basic functionality, such as internationalization and locale support, system configuration, etc.
libkio	The KDE I/O library for low level access to network files.
libqt	Qt is an object-oriented C++ application development framework that provides standard GUI widgets and APIs for file-handling, network, threading, etc.
libfam	FAM (File Alternation Monitor) that notifies changes in specific files or directories.
libc	The C library
libX11	The X11 library

Table 3. Major libraries and code modules of Konqueror

4.2.2 Energy Profiling of Konqueror and the Overheads of EPAT

The purpose of this section is to explore the energy consumption of Konqueror with various features being activated and to evaluate the overheads of EPAT (with the proposed hybrid monitor-based profiling approach). The evaluation was conducted over well-known web sites, such as *Yahoo!* (`http://www.yahoo.com/`), and a popular Chinese news web site *Chinatimes* (`http://news.chinatimes.com/`).

Konqueror 3.3.1 was profiled over Linux 2.4.18 with gcc 2.95, KDE 3.3.1, and X 4.2.0 (for X servers, libraries, and utilities). EPAT was built over the same platform with RTAI 24.1.9. The filesystem mounted on disk partitions was ext3, and an Intel PIII 733 MHz machine was adopted for performance evaluation. We removed any servers irrelevant to the profiling, such as the `cron` daemon and the `sendmail` server, and shut down `syslogd` to prevent potential factors from interfering with the experimental results. Swapping was disabled, and the `/tmp` partition was mounted as TMPFS (temporary file system), which kept files in the virtual memory, for better performance. The `TSC` register on 80x86 microprocessors was adopted to measure the time spent by EPAT on the profiled computer system at the granularity of a clock cycle.

The energy consumption of a profiled computer system over a time interval T was $E = \int_T P(t)dt = \frac{V_{supply}}{R_{shunt}} \int_T V_{diff}(t)dt$, where R_{shunt}, V_{supply}, $V_{diff}(t)$, and $P(t)$ were the shunt resistor resistance, the supply voltage, the differential voltage (of the shunt resistor between the power supply and the motherboard), and the power usage at time t, respectively. The current through the profiled computer system at time t was $I(t) = V_{diff}(t)/R_{shunt}$ according to the Ohm's Law, and $P(t) = V_{supply} * I(t) = V_{supply} * V_{diff}(t)/R_{shunt}$. Because the energy consumption of the profiled computer system was profiled at a regular period Δt, the energy consumption of the system could be approximated by $E \approx \frac{V_{supply}}{R_{shunt}} \sum_{i=0}^{n} V_{diff}(i)\Delta t$. The power usage was in watt, and the energy consumption was in joule.

The CPU time usage of Konqueror	Overheads (in a percentage)
0.930 seconds	6.45%
1.413 seconds	6.12%
1.810 seconds	5.61%
2.486 seconds	5.32%
2.919 seconds	5.67%
4.008 seconds	4.79%

Table 4. The run-time overheads of EPAT when Konqueror with different workloads was profiled at 20,000 samplings/second in 10 seconds.

For the rest of this section, the energy profiling of Konqueror was reported over Yahoo! and Chinatimes with different browser features being activated. Table 4 shows the run-time overheads of EPAT when Konqueror with different workloads was profiled at 20,000 samplings/second in 10 seconds. Note that the remaining CPU time of the profiled computer system in each 10-second profiling was mostly consumed by the idle task or other system serives, such as those from X. The overheads was derived as (the CPU time usage of EPAT)/(the CPU time usage of Konqueror). It was not surprised that the CPU time usage of EPAT was roughly proportional to the CPU time usage of the profiled program, although some fixed overheads did exist. The profiling overheads were no more than 7% of the profiled system.

- **Loading of a Web Page**

The energy consumption of Konqueror in the loading of a web page over Yahoo! and Chinatimes in each 15 seconds was first measured by EPAT. Compared to Chinatimes, Yahoo!, that is designed being compatible with most web browsers and has a good response time, does not have a heavy use of JavaScript. Loading a web page over Yahoo! consumed only 8.732J, while

Process	Time (sec)	Energy (joule)	Power (watt)
idle	12.965	153.652	11.85
Konqueror	0.626	8.732	13.95
X	0.539	7.765	14.39
Other	0.867	11.908	13.73

Table 5. Energy consumption of different processes in the loading of a web page over Yahoo!

Category	Related Class
Classes related string manipulation	QCString QChar QConstString QString QStringData QTextString QGCache ...
Classes related to basic data structure manipulation	QGArray QGListIterator QMetaObject QObject QValueListPrivate QGList QGDict ...
Other classes	QBig5Codec QBig5Decoder QBig5hkscsCodec QFontMetrics QFontPrivate ...

Table 6. Lower-level classes related to the manipulation of strings, basic data structures, etc.

that over Chinatimes consumed 58.321J. It was because Chinatimes (the most popular Chinese news web site in Taiwan) had more complex web page designs and required Chinese encoding conversion. Better conversion algorithms on a browser or good designs of JavaScript code on a web page would be very helpful to the energy-efficient considerations of a browser. Table 5 shows the execution time, energy consumption, and power usage of different processes in loading a web page over Yahoo!. It was surprised to see that the profiled system consumed a huge amount of energy in idling, compared to other parts, where the PIII platform in the experiments did not support dynamic voltage scaling. The idling was mainly due to the lengthy waiting time for data delivery over the networks. Intelligent voltage scaling policies would clearly improve the energy consumption of a browser substantially.

The energy profiling of functions in Konqueror (by excluding basic functions in the handling of data structures, e.g., the classes in Table 6) was reported in Table 7 by an analysis of the collected execution paths. Compared to the energy profiling of all functions, as shown in Table 8, Table 7 provides other information on the behavior of Konqueror (in terms of functions in higher layers of the system architecture, as shown in Figure 6). For example, Table 7 shows that text-drawing, i.e., function `QPainter::drawText`, costed 2.45% of the entire energy consumption. Without any help on the tracking of execution paths, engineers could only observe the energy consumption of each module, such as those shown in Table 9, where the energy consumption of the module `KERNEL` denoted that for Konqueror executing in kernel mode. Observations without ex-

	Function	Module	Layer	% Energy
1	khtml::Font::update	libkhtml	B	8.68%
2	KURL::url	libkdecore	C	2.97%
3	QPainter::drawText	libqt	D	2.45%
4	QEventLoop::enterLoop	libqt	D	1.96%
5	QEventLoop::processEvents	libqt	D	1.86%
6	khtml::CSSStyleSelector::styleForElement	libkhtml	B	1.77%
7	khtml::CSSStyleSelector::checkOneSelector	libkhtml	B	1.54%
8	QPixmap::convertFromImage	libqt	D	1.54%
9	DOM::StyleBaseImpl::parseValue	libkhtml	B	1.43%
10	QTextCodec::fromUnicode	libqt	D	1.28%
11	QGIFFormat::decode	libqt	D	1.22%
12	khtml::HTMLTokenizer::parseTag	libkhtml	B	1.18%
13	DOM::FontFamilyValueImpl::FontFamilyValueImpl	libkhtml	B	1.13%
14	khtml::CSSStyleSelector::applyRule	libkhtml	B	1.08%
15	khtml::Decoder::decode	libkhtml	B	1.06%

Table 7. Energy profiling of functions in Konqueror for the loading of a Yahoo! web page.

	Function	Module	Layer	% Energy
1	**malloc**	libkdecore	C	7.54%
2	**free**	libkdecore	C	3.84%
3	khtml::CSSStyleSelector::styleForElement	libkhtml	B	1.54%
4	**__builtin_new**	libfam	D	1.46%
5	**memcpy**	libc	D	1.27%
6	QGIFFormat::decode	libqt	D	1.21%
7	operator==	libqt	D	1.05%
8	khtml::HTMLTokenizer::parseTag	libkhtml	B	1.01%
9	**QString::setLength**	libqt	D	1.01%
10	QString::find	libqt	D	0.93%
11	**__builtin_delete**	libfam	D	0.89%
12	khtml::CSSStyleSelector::applyRule	libkhtml	B	0.87%
13	QPixmap::convertFromImage	libqt	D	0.86%
14	**QString::QString**	libqt	D	0.85%
15	QFontPrivate::textWidth	libqt	D	0.82%

Table 8. Energy profiling of all functions in Konqueror for the loading of a Yahoo! web page. Functions involved with basic memory management are marked bold.

Module	Energy (joule)	% Energy
libqt	3.601	41.23%
libkhtml	1.819	20.83%
libkdecore	1.228	14.06%
KERNEL	0.485	5.55%
libc	0.364	4.16%
libfam	0.337	3.86%
libX11	0.181	2.07%
libXft	0.131	1.49%
libpthread	0.130	1.49%
libkjs	0.080	0.91%
konqueror	0.072	0.82%
libz	0.060	0.68%
libkio	0.057	0.65%

Table 9. Energy profiling of modules in the loading of a web page over Yahoo!

clusion of selected functions were often on the energy consumption of basic functions. For example, 7 out of 15 most energy-demanding functions, as shown in Table 8, were involved with basic memory management (that costed roughly 17% of the entire energy consumption).

Tracking of execution paths clearly provided observations of energy profiling at different granularity levels. The energy profiling of all Konqueror functions, when a Chinatimes web page was loaded, was similar to that in Table 8, except that the processing of Chinatimes web pages involved with JavaScript parsing and the conversion of Big5 and Unicode in Chinese characters (the table is not included because of the similarity).

The energy consumption amounts for JavaScript parsing and the conversion of Big5 and Unicode in Chinese characters were 0.88% and 2.68% of the entire energy consumption, respectively. Furthermore, the ratios of energy consumption of `khtml::CSStytleSelector::styleForElement` and `khtml::Font::update` were larger than their counterparts in Table 7. We observed that the functions were hot spots in the loading of a web page. Some insights could be drawn from the observations: (1) Because a web browser application could be associated with default fonts, the most frequently used fonts (e.g., the true type) and their font sizes could be pre-rendered and cached to reduce the time for font-rendering-related computations. (2) Since `khtml::Font::update` was a hot spot, any optimization of the related code would be definitely needed. (3) `khtml::CSStytleSelector::styleForElement` was used to sort out most frequently used style rules. Some caching and pre-calculation mechanisms would be helpful to reduce the energy consumption. Although Table 8 showed that functions `malloc` and `free` were hot spots in the loading of a web page, the functions were at a lower level and might be highly optimized already. Tracking of their usages would be more helpful in reducing the energy consumption of a web browser.

Furthermore, the energy profiling of different C++ classes could be obtained. It provides other information for high-level process executions. For example, `QPainter` (basic painting, such as line drawing) costed 6.18% of the entire energy consumption, and `QGIFFormat` (GIF image decoding) costed 1.29% of the entire energy consumption.

- **Viewing of a Web Page**

The energy profiling of Konqueror in viewing a web page was done by analyzing its energy consumption while a loaded web page was scrolled down at a constant rate in 15 seconds. The energy consumption of Konqueror in page viewing was substantially less than that in loading. For example, the energy consumption of Konqueror in viewing was roughly 1.593J over Yahoo! pages, and that over Chinatimes was roughly 40.334J. Note that Chinatimes had a news-sticker implemented with JavaScript, and that resulted in updating of a web page frequently. Compared to the energy consumption in page loading, the energy consumption in the reviewing of a web page over Yahoo! was about $1.593J/8.732J/ \approx 18\%$ of that in loading. For Chinatimes, the ratio of the energy consumption in viewing and loading was about $40.334J/58.321J/ \approx 69\%$. In the performance evaluation, the energy consumption of the modules that handled JavaScript (e.g., `libkjs` and `kjs_html`), when a web page was loaded (/viewed), costed 9.16% (/7.98%) of the entire energy consumption. It further emphasized the criticality in the energy-efficiency considerations for the handling of JavaScript code.

From the experimental results of the energy consumption of all functions, we observed that many functions were involved with HTML rendering (e.g., `khtml::RenderObject::nodeAtPoint`, `khtml::RenderFlow::paintObject`, etc). The ratio of total energy consumption of such functions was roughly 13%. By excluding basic functions in the handling of basic data structures, the information on the behavior of Konqueror in terms of functions in higher layers of the system architecture could be derived, i.e., energy profiling of C++ classes. We observed that the ratio (19%) of energy consumption of `QPainter` in the viewing of a web page was obviously larger than the counterpart (6.18%) in the loading of a web page. The reason is that the web page was scrolled down at a fixed rate in the experiments such that Konqueror redrew frequently. The energy profiling of all Konqueror functions in the viewing of a Chinatimes web page was similar to that of the viewing of a Yahoo! web page, except that the energy consumption amounts for JavaScript parsing and the conversion of Big5 and Unicode in Chinese characters were 1.2% and 4.65% of the entire energy consumption, respectively. If related data in rendering were cached during consecutive loadings of web pages, the rendering-related computations might be reduced significantly.

When an application is profiled by the proposed profiling approach, the information of energy consumption could be on different levels of abstraction, e.g., functions, modules, or even classes. Application developers could identify the components of an application which need more energy from different-level energy profiling information. Then they could improve the application program. Note that the major objective of this paper is not to provide complete insights about how to improve the profiled application. Our goal is to provide different-level profiling information to let the developers know the possible hot spots in the application program.

5 Conclusion

In this paper, we propose a hybrid hardware-software-based energy profiling solution, which could be implemented with little cost (no more than 7% of the profiled system) but high accuracy. Compared to the existing work, the proposed solution could provide richer information on energy profiling. We propose an execution-path-tracking approach to provide multiple granularity levels in the observation of the energy consumption of a program execution. We address technical issues in the design and implementation of the proposed profiling solution. A quantitative energy-profiling study over a well-known web browser Konqueror is then presented to provide a feasibility study of the proposed solution and insights in the design of an energy-efficient web browser. The experimental results show the energy consumption of functions, modules, or even classes at different layers of a program. It helps in the identification of program components that need better improvement, e.g., encoding conversion, and the problems in the designs of web pages, such as those with a heavy usage of JavaScript code.

For the future work, we shall extend the hybrid hardware-software-based profiling solution for more elaborated energy analysis, especially on interactive GUI-based applications, where the workloads of GUI applications are often driven by external GUI and system events. We should also extend EPAT to the exploiting of energy consumption of an application run on a platform supporting dynamic voltage scaling.

References

[1] Apple^{TM}Safari. `http://www.apple.com/safari/`.

[2] Konqueror. `http://www.konqueror.org/`.

[3] National Instruments DAQ. `http://www.ni.com/daq/`.

[4] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating System Principles*, Oct. 1997.

[5] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[6] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[7] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. Diapm-rtai position paper. *Real Time Operating Systems Workshop*, Nov. 2000.

[8] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.

[9] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. T. Kandemir, T. Li, and L. K. John. Using complete machine simulation for software power estimation: The softwatt approach. In *International Symposium on High-Performance Computer Architecture HPCA*, pages 141–150, 2002.

[10] D. C. Lee, P. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows NT. In *ISCA*, pages 27–38, 1998.

[11] D. Shin, H. Shim, Y. Joo, H.-S. Yun, J. Kim, and N. Chang. Energy-monitoring tool for low-power embedded programs. *IEEE Design and Test of Computers*, 19(4):7–17, 2002.

[12] T. Simunic, L. Benini, and G. D. Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Design Automation Conference*, 1999.

[13] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. In *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[14] Tool Interface Standard (TIS). *Executable and Linking Format (ELF) Specification*.

[15] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.