2012 IEEE International Conference on Green Computing and Communications, Conference on Internet of Things, and Conference on Cyber, Physical and Social Computing

Profiling Software for Energy Consumption

Simon Schubert
EPFL, Lausanne
simon.schubert@epfl.ch

Dejan Kostić
EPFL, Lausanne
dejan.kostic@epfl.ch

Willy Zwaenepoel
EPFL, Lausanne
willy.zwaenepoel@epfl.ch

Kang G. Shin
The University of Michigan
kgshin@eecs.umich.edu

***Abstract*—The amount of energy consumed by computer systems can be lowered through the use of more efficient algorithms and software. Unfortunately, software developers lack the tools to pinpoint energy-hungry sections in their code and therefore have to rely on their intuition when trying to optimize their code for energy consumption. We have developed *eprof*, a profiler that relates energy consumption to code locations; it attributes both the *synchronously* consumed energy in the CPU and the *asynchronously* consumed energy in peripheral devices like hard drives, network cards, etc. *Eprof* requires minimal changes to the kernel (tens of lines of code) and does not require special hardware to energy-profile software. Therefore *eprof* can be widely used to help developers make energy-aware decisions.**

I. Introduction

The growing energy consumption of IT systems is quickly becoming a major concern for users, ranging from corporations trying to keep the total cost of ownership low, to end-users who expect their mobile devices not to run out of battery while on the run.

Choices that developers make in their software architecture and software algorithms have a significant effect on the energy usage of a system. All hardware has a baseline or *idle* power draw which the hardware consumes regardless of the activity of the system; on top of this, current hardware has a large *dynamic* energy component caused by the specific interactions of software with hardware components. Aside from improving hardware to reduce its energy consumption, the system energy consumption can be reduced by optimizing the interaction of software with hardware.

When trying to write energy-efficient software, developers currently have to rely on their intuition, because few tools and methods exist which give insight into the energy consumption of software. To this end, we have developed the *eprof* software energy profiler, a tool which relates the consumed dynamic energy back to the software that caused this consumption. Using *eprof*, software developers can make informed choices about which algorithms use less energy.

For example, a developer might have to choose between using CPU resources to evaluate a function each time it is used versus using memory accesses to look up precomputed values in a table. Or, he might have the choice between storing data in a compressed form on a disk, which requires extra CPU resources versus using an uncompressed format, which requires more disk accesses. In both cases *eprof* allows the developer to understand how much energy is consumed for each option. Moreover, if a software developer must optimize a large code-base to use less energy, he can use *eprof* to identify the code locations that use most energy, in order to rewrite them. Seemingly simple functions with short execution times may, in fact, consume vast amounts of energy because they make asynchronous disk accesses or other device I/O.

While classic performance profiling might help in reducing runtime and therefore energy consumption in the CPU, this CPU-centric approach ignores any energy consumed in peripheral devices. However, especially on smartphones and notebooks, the peripheral devices are significant energy consumers. *Eprof* accounts for the energy consumed in devices and attributes this energy to code locations that are responsible for the device activity.

In contrast to currently available tools, *eprof* does not require instrumentation of the source and generates energy-usage information at a fine granularity; it can identify the energy used by individual functions. After a calibration phase, *eprof* does not require any external devices or circuitry and thus enables the average developer to profile his software for energy consumption.

The contributions of this paper are:

- attribution of energy consumed in asynchronous device interaction to the responsible code;
- profiling for software energy consumption without the need for external instrumentation; and
- design and implementation of *eprof*.

II. Background and Challenges in Energy Accounting and Profiling

Software energy consumption is one of the two types of energy consumption in a computer system: A static component, the *idle* power, is always used, independent of the activities of the system; on top of this exists a *dynamic* component which depends on the activities of the system. Software executing on the system will influence this *software energy* by computation in the CPU, memory accesses and device interaction.

In modern ULV (Ultra-Low Voltage) processors and specialized mobile hardware, the dynamic portion can significantly exceed the idle power draw. In such systems, the reduction of dynamic energy has a large impact on the overall energy consumption. We therefore posit that an *energy profiler* is needed to help the developers gain insight into the distribution of energy consumption among the code locations.

One difficulty in generating energy profiles is that hardware does not track energy-usage information. However, other

978-0-7695-4865-4/12 $26.00 © 2012 IEEE
DOI 10.1109/GreenCom.2012.86

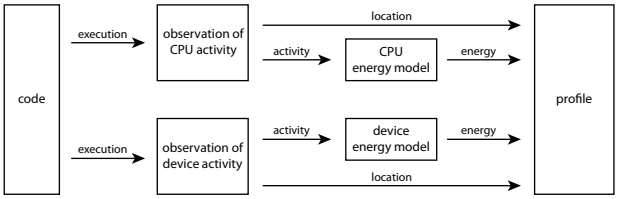

Figure 1. *Eprof* system overview. The profiled code executes, causing activity in CPU and devices. This activity is observed by *eprof*, which records the code location of the activity and estimates the energy consumed via an energy model. Both estimated energy and code location are assembled to form an energy profile of the executed code.

metrics obtained from the devices can be used as proxy for the consumed energy, by means of an *energy model* (Section III-C).

Challenges

The existing large body of work on building accurate energy models does not permit the developer to associate code locations with the consumed energy. This task requires the equivalent of a profiler that attributes energy, instead of straightforward CPU utilization, to lines of code.

Moreover, the energy models are typically designed to treat the system elements (CPU, memory, cache, etc.) as synchronous entities and do not account for asynchronous requests (e.g., disk). The same applies for the traditional profilers that do not deal with asynchronous devices. This presents an additional challenge in profiling for energy.

It might be tempting to estimate the energy consumed by a given code block or even a single line by creating a testing harness that will exercise the desired code in a tight loop, and measure the consumed energy. While doing so can be accurate on simple systems, this approach fails in complex systems where cache locality, CPU superscalar execution, or operating system buffering effects obscure the results.

III. *Eprof*

An energy profile requires two types of information: the *amount* of energy spent, and the *code location* which caused this energy consumption. For every device, the *eprof* profiler therefore implements two components: *observation* of energy-relevant activity and *estimation* of the amount of energy consumed by this activity (see Fig. 1).

When activity is observed in the CPU or in a device, *eprof* records a stack trace to capture the code location where this activity originated; the process when and where to capture this stack trace depends on whether the activity is happening synchronously or asynchronously. *Eprof* also estimates the energy consumed for a particular CPU or device activity by using a energy model. These two pieces of information, stack trace and energy estimate, form the data required for the energy profile of the tested code.

The CPU and memory subsystem consume energy *synchronously*, i.e. the consumption happens while the code is executing; computation, execution, and access to various stages of the memory hierarchy will consume energy right while they are being processed. Capturing the code location

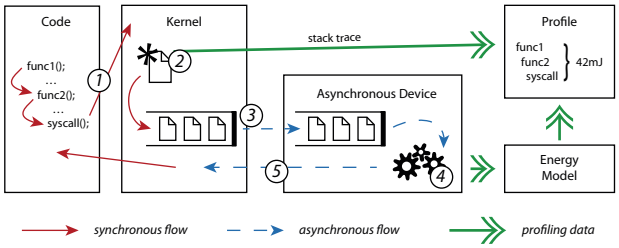

Figure 2. Flow of an asynchronous device request and profiling of its energy consumption. (1) Code executes syscall to perform device I/O. (2) Kernel allocates new request data structure and places it into a queue; *eprof* obtains stack trace. (3) Driver dequeues device request and sends it to device. (4) Device consumes energy while executing request. (5) Device notifies kernel about request completion; *eprof* estimates energy for request. Energy estimate and stack trace update the energy profile.

responsible for this activity is therefore straightforward. On the other hand, devices consume energy *asynchronously*; a process might initiate a read request to the hard drive, but both the operating system and the device might queue the request before processing it. When the request is finally processed by the device, other code will be executing on the CPU. This complicates the association between device activity and originating code.

A. *Profiling CPU Energy Consumption*

For CPU and memory energy profiling, we use statistical profiling via hardware performance event counters (HPCs). *Eprof* observes activity by programming the HPCs to generate an interrupt whenever a counter threshold is reached. When serving this interrupt, *eprof* captures a stack trace of the currently running thread. Using a calibrated CPU and memory energy model, *eprof* programs the HPCs as sampling performance counters with thresholds set so that each sample corresponds to a fixed amount of energy. In our prototype, we use the perf subsystem of the Linux kernel to capture HPC events and to report the profile.

B. *Profiling Device Energy Consumption*

Profiling for device energy consumption is significantly more complicated than profiling for CPU energy consumption due to the *asynchronous* nature of devices and device requests. We will first give an overview of how a device request travels through the system and we will then explain how *eprof* ties into the system to obtain profiling data for code using devices.

Fig. 2 shows how a device request originates and flows through a computer system:

(1) Code executes and calls into the kernel to perform some device activity, such as disk or network I/O.
(2) The respective kernel subsystem eventually allocates a new device request structure and places this request structure into a request queue; this ends the synchronous execution path and the thread blocks until request completion.
(3) Later the request gets dequeued asynchronously and the request is sent to the device, possibly after being reordered. The device may queue and reorder the requests internally as well.

(4) The device processes the request and therefore consumes energy.
(5) Once the device finishes processing, it asynchronously notifies the kernel about the completion. Later the kernel wakes up the waiting thread which then returns synchronously to userspace.

The asynchronous nature of steps (3) to (5) means that energy may be consumed before or after the responsible code is executed. For example, energy might be spent in a radio when writing data to a network socket, which will later lead to a network packet being transmitted; the submitting code can be seen as being responsible for the energy consumption of the transmission, even though it executes before the energy will be consumed by the radio. Likewise, a read from a file might lead to energy consumed in the hard drive servicing the read request; the code requesting the file read will be running while it is submitting the read, but because of multiprogramming, another process will be active while the hard drive services this request.

The provenance of a device request must therefore be recorded while the thread is executing synchronously. We identify and discuss three obvious opportunities to do so during the request flow: syscall entry, insertion of the request to the processing queue, or allocation of the request data structure.

Capturing at the syscall entry: The syscall entry of a kernel is a limited API and thus is a single, defined location that needs to be instrumented to capture the stack trace. During syscall entry, there is not enough information to determine whether the syscall will eventually lead to a device request: because operating systems extensively use buffering and request merging, a read syscall might never lead to a disk request being made because the requested data might already be present in the buffer cache; likewise, send syscalls do not always lead to a separate network packet being transmitted; data might be queued until more data allows for more efficient transmission (Nagle's algorithm), or packets might be compressed or fragmented for transmission.

Capturing at the driver level: Capturing a stack trace when the request gets inserted into the driver device queue is the last opportunity in the synchronous processing flow. At this point, it is also definite that a request will be submitted to the device, avoiding the uncertainty problem of the syscall entry approach. However, pinpointing where exactly in the code the stack trace should be captured requires high familiarity with the code of the kernel subsystem and it might even require modification of each driver.

Capturing in the device subsystem: The final option is to record the stack trace when the request data structure is allocated. Within the request flow, this option is located between syscall and request enqueueing. It combines the certainty of request submission of the enqueue location with the limited API of the syscall entry location. Request data structures are commonly allocated by one specialized function for each request type; for example, in the Linux kernel, network packets get allocated by __alloc_skb and disk I/O request are allocated by bio_init. These functions are well-defined interfaces for each kernel subsystem, therefore avoiding tedious study of every driver; instead, they are easy to locate for each request type. They are always called in the synchronous code path, but at the same time they are only called when a request will be submitted to a device; if a syscall does not lead to a device request due to buffering or request merging, no data structure will be allocated by the kernel. In *eprof* we therefore follow the principle of capturing provenance information whenever a device request structure is allocated. Table I shows the amount of kernel changes required for our prototype.

TABLE I. REQUIRED KERNEL CODE CHANGES FOR *eprof*.

Kernel Location	LOC modified/added
Generic *eprof* support	269
Disk energy provenance	15
Paging energy provenance	25
Network energy provenance	33

C. Energy Models

In our prototype of *eprof*, we use previously published energy modeling concepts. If required, more sophisticated models could be used, such as state machine-based device models [1], models including dynamic voltage and frequency scaling [2], or more detailed hard drive energy estimation [3], [4]. We train all models separately for each platform by measuring the system power draw during a set of benchmarks.

CPU and Memory Energy: The CPU and memory energy model is a linear model based on hardware performance counters [5]; it is trained using the SPEC CPU 2006 benchmark suite. We select the set of performance counters that yields the best energy model; this optimization also determines the coefficients for the model [2].

Like [6], we use captured sequences of HPCs and externally measured energy to train a linear model for each valid combination of counters. We select the one combination of counters which provides the lowest residual error. This selection process also determines the coefficients for the linear model.

Hard Drive Energy: Disk energy consumption is modeled using a simple linear relation between request duration and energy [3]. We train the model using a micro-benchmark that varies disk load intensity and disk seek distances.

D. Unifying CPU and Device Energy

Because device energy usage is quantified by a device-specific energy model, while CPU energy usage is captured through statistical profiling, *eprof* must merge these separate sources into a single combined dataset. We do this by transforming the device energy information to be compatible with the statistical profiling samples. As a result, existing statistical profiling tools can be used to analyze the combined dataset.

In our prototype, we use the perf subsystem of the Linux kernel to capture and report the profile. Perf, like most profilers, does not allow for quantitative weights for the recorded samples. This means that every sample corresponds to the same amount measured. For example, in traditional profiling

based on sampling, each sample would represent a fixed unit of time, e.g., 10 ms. With traditional sampling hardware profiling counters, one sample would represent the fixed threshold of the hardware counter, e.g., 10000 L2 cache misses. In *eprof*, every sample represents a fixed amount of energy, e.g., 100 mJ. The asynchronously consumed device energy, E_{dev}, will rarely match exactly this fixed value, E_{sample}. We address this problem by recording the call trace in the profiling data m times rather than merely once, where $m = \lfloor E_{dev}/E_{sample} \rfloor$. That is, we record many samples so that the aggregate recorded equivalent energy $E'_{dev} = m \times E_{sample}$ does not exceed the real consumed energy E_{dev}. The remainder in energy, $\Delta E_{dev} = E_{dev} - E'_{dev}$, is preserved and added to the energy of the next request. This way, no energy consumption is left unreported, while maintaining the properties of statistical profiling. For example, given $E_{sample} = 0.1\,\text{J}$, the real consumed energy $E_{dev} = 0.87\,\text{J}$ will be recorded as $m = 8$ samples in the profiling data, and $\Delta E_{dev} = 0.07\,\text{J}$ will be added to the next request energy.

IV. Evaluation

We evaluate *eprof* on two hardware platforms: (1) an Asus EeePc 1005P netbook with an Intel Atom N450 CPU (2) a Dell OptiPlex 755 MT with an Intel Core 2 Quad Q6600 CPU We obtain all energy measurements using a WattsUp .Net power meter.

We use the SPEC CPU2006 benchmark suite[2] in reference size to train the profiler and to evaluate its accuracy over a large range of different applications. Leave-one-out cross-validation over the benchmark shows an average CPU model energy estimation error below 10 %; the maximum runtime overhead of *eprof* across this benchmark is 2.7 %.

The tables containing profiling results list the locations energy was spent; userspace function names are formatted plain, kernel functions are in [brackets]. For disk-related energy consumption, we indicate that a function of interest appears as part of a stack trace by setting it in *italics* and prepending an ellipsis (…). For example, *…[sys_read]* indicates that the given amount of energy was spent in the *disk*, and the stack trace contained [sys_read], the read system call function. In particular, [copy_user_generic_string] is used to move data from the buffer cache to userspace memory, and vice versa.

A. Attribution correctness

We show and evaluate the attribution accuracy of *eprof*. Attribution accuracy is independent of any used model; an accurate attribution will assign defined quantities to correct code locations.

We demonstrate that *eprof* correctly attributes energy on the function level by profiling a microbenchmark application. The first part of the benchmark is a CPU-intensive repeated generation of an SHA1 hash, using the OpenSSL library; this computation consists of SHA1_Update, sha1_block_data_order and memcpy. These functions are marked with (C) in the profiling tables. The second part is the memory-intensive repeated sorting of a 50MB array of integers, alternating the sort order each time. This is done with qsort, which in turn calls msort_with_tmp, cmpfn and memcpy. Those functions are marked with (M) in the profile tables.

[2]We omit 401.perlbench, 447.dealII, 481.wrf, and 483.xalancbmk because they do not compile or run due to the compiler version used.

TABLE II. ENERGY PROFILE OF THE CPU INTENSIVE TASK. Real energy measured was 923 J, an error of 4.3 %.

Fraction %	Energy J	Location
60.14	579	(C) sha1_block_data_order
18.75	180	(C) SHA1_Update
13.47	129	(C) memcpy
2.49	23.9	(C) main
Total	963	

TABLE III. ENERGY PROFILE OF THE MEMORY INTENSIVE TASK. Real energy measured was 1396 J, an error of 6.8 %.

Fraction %	Energy J	Location
47.18	613	(M) memcpy
30.85	401	(M) msort_with_tmp
12.45	162	(M) cmpfn
4.98	64.8	(M) main
Total	1301	

TABLE IV. ENERGY PROFILE OF THE MIXED TASK. Real energy measured was 2291 J, an error of 3.6 %. (C) denotes functions of the CPU-intensive task, (M) denotes functions of the memory-intensive task.

Fraction %	Energy J	Location
31.36	692	(M+C) memcpy
27.92	616	(C) sha1_block_data_order
18.63	411	(M) msort_with_tmp
7.24	159	(M) cmpfn
7.18	158	(C) SHA1_Update
3.20	70.6	(M+C) main
Total	2208	

Table II shows the profile for the CPU-intensive computation running alone, Table III shows the profile for qsort running alone. The total energy for both separate computations is estimated with a low error (4.3 % and 6.8 %).

We now combine these two computations into a single benchmark application and compare its profile to the profiles of the separate runs. For the attribution to work correctly, we expect that the attributed amount of energy for each computation stays constant and is not influenced by running both computations together.

The resulting profile is shown in Table IV. Compared to the real energy measured, the total energy consumption is estimated with 3.6 % error. The profile of the combined run matches the sum of the total energy for both separate runs very closely: 1.2 % error for measured energy and 2.5 % error for estimated energy.

When looking at the energy attribution to specific functions used in the computations, a similar picture emerges. For example, msort_with_tmp is attributed 411 J for the combined run and 401 J in the separate run (2.4 % error). The other

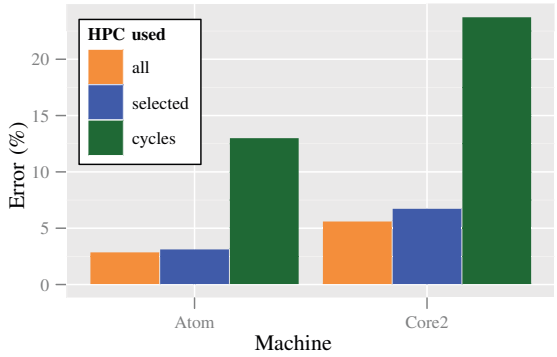

Figure 3. Median error of the CPU/memory energy estimation using different energy models. *All* is an ideal energy model using all captured performance counters and *selected* is the model selected as described in Section III-C. *Cycles* is a model only using CPU cycles, mimicking a conventional execution time profiler.

functions exhibit similar results (cmpfn 1.9 %, SHA1_Update 13.9 %, sha1_block_data_order 6.0 %).

Two functions are shared by both computations, main and memcpy. The sum of memcpy in the separate runs (742 J) also matches closely the attribution of the combined run (692 J, 7.2 % error). The sum for main does not match the combined run well. However, this is expected because main executes the main benchmark loop and therefore duplicates much of the work during the separate runs; in the combined run, this duplication does not occur, therefore reducing the total energy consumption by estimated 25.6 %.

The results of this microbenchmark show that *eprof* can accurately attribute software energy to functions and allows the developer to precisely determine which parts of the code consume how much energy.

B. Energy and run time are not proportional

One might surmise that energy consumed by the program is proportional to the time spent running it. While this tends to be true in general, it does not accurately capture the amount of energy spent. Fig. 3 shows that profiling only for CPU cycles results in an energy model with a significantly higher estimation error. Furthermore, profiling for CPU cycles does not capture asynchronous energy used by devices. In our experiments, we experienced as much as 33 % of the total energy being consumed in asynchronous requests. In Tables V and VI, all energy spent on disk I/O, listed as functions *...[sys_write]*, *...[sys_read]* and *...[do_truncate]* would go unnoticed. On the Atom, this energy amounts to 25 % of the total energy; on the Core2, it amounts to 33 %. This underlines the importance of capturing and attributing asynchronous device energy.

V. Real-world Applications

We now study the energy profile of a real-world application and how energy profiling might guide development. In contrast to performance optimization, energy-conscious development is still in its infancy, and therefore best practices in algorithms and data structures to lower energy consumption have not yet been thoroughly studied. In this paper, we will not venture into proposing how to program more efficiently, but instead we demonstrate how tools might be used to enable energy-aware development.

Table V. Energy profile of processing a large text file on the Atom. Read_and_xlate in tr performs the processing, the other functions do disk I/O; other tables also show processes gunzip and gzip, which are used for decompression and compression of the data. Disk I/O uses 28 % of the total energy, the text processing uses 17 %.

Frac. %	Energy J	Proc.	Location
17.21	44.6	tr	read_and_xlate
13.84	35.9	tr	*...[sys_write]*
9.15	23.7	tr	*...[sys_read]*
2.77	7.19	tr	[copy_user_generic_string]
2.14	5.55	tr	*...[do_truncate]*
Total	260		

Table VI. Energy profile of processing a large text file on the Core2. Disk I/O uses 42 % of the total energy, text processing uses 8 %.

Frac. %	Energy J	Proc.	Location
16.86	80.3	tr	*...[sys_read]*
15.14	72.1	tr	*...[sys_write]*
9.45	45	tr	[copy_user_generic_string]
8.00	38.1	tr	read_and_xlate
1.18	5.62	tr	*...[do_truncate]*
Total	476		

Table VII. Energy profile of processing a compressed text file on the Atom. De- and re-compression use 75 % of the total energy, text processing uses 6.6 %. Disk I/O energy is negligible.

Frac. %	Energy J	Proc.	Location
19.86	116	gzip	longest_match
18.66	109	gzip	deflate
17.46	102	gzip	fill_window
6.65	38.9	tr	read_and_xlate
6.55	38.4	gzip	updcrc
5.99	35.1	gunzip	flush_window
3.61	21.1	gunzip	inflate_codes
2.13	12.4	gzip	ct_tally
Total	586		

Table VIII. Energy profile of processing a compressed text file on the Core2. De- and re-compression use 83 % of the total energy, text processing uses 3.5 %. Disk I/O energy is negligible.

Frac. %	Energy J	Proc.	Location
23.92	253	gzip	longest_match
14.86	157	gzip	deflate
14.79	156	gunzip	flush_window
12.85	136	gunzip	updcrc
10.54	111	gzip	fill_window
3.94	41.7	gunzip	inflate_codes
3.55	37.6	tr	read_and_xlate
2.29	24.2	gzip	compress_block
Total	1060		

A. Use compression to reduce disk I/O?

We now study the energy profile of a real-world application and how energy profiling might guide development.

Many data-intensive applications, such as Google's *BigTable*, choose to work with compressed data to reduce the amount of I/O they need to perform. Compression has also been suggested [7] as a way to reduce energy consumption, based on the assumption that the CPU can handle the decompression and compression tasks using less energy than when transferring large amounts of data from and to the hard drive. We use *eprof* to check whether this hypothesis holds.

To compare compressed and uncompressed file processing, we perform text replacement on a large log file, using the UNIX tr utility. In the uncompressed case, the input file (1.7GB) is fed directly to tr, and the output is written to disk. In the compressed case, a compressed version of the input file (101MB) is decompressed on the fly by gunzip and piped to tr; its output is piped to gzip, which re-compresses the data on the fly and writes it back to disk.

Tables V and VI show the energy profile for the uncompressed file processing on Atom and Core2, respectively. Tables VII and VIII show the energy profile for the compressed case. The energy used in the compression/decompression is significantly higher than the amount of energy used by the disk (for both of our platforms). This finding shows that the common wisdom is incorrect, at least for the popular systems that we consider. In addition, this real-world case demonstrates the need for an energy profiler, such as *eprof.*

Next, we discuss the relative energy consumption of Atom vs. Core2. As expected, the Atom (a more energy-efficient platform) consumes significantly less energy for the same task in both the compressed (1.83x) and uncompressed case (1.89x). The increase due to the use of compression is less for the Atom (2.15x) than the Core2 (2.2x). This result agrees with the intuition, as the Core2 system has a higher-performance CPU and disk, leading to a higher joule-consumed-per-amount-of-work-performed.

Surprisingly, the energy profile for the uncompressed case is quite different between the CPUs: On the Atom the largest single consumer is the actual text processing routine, using 17 % of the total energy (44 J). However, on the Core2, the text processing only makes up 8 % of the energy (38 J), which amounts to *less* absolute amount of energy than on the Atom. A significantly larger portion of energy is spent on disk I/O (both disk energy and copying of buffer cache data) on the Core2 (41 %) than on the Atom (25 %).

This profile shows that energy use can differ greatly between architectures and it underlines the necessity to use a profiling tool instead of simply relying on best practices and intuition when it comes to producing energy-efficient software. Although a developer could use an external power meter to compare the energy consumption of both approaches, he would not be able to analyze which parts of the system use how much energy.

TABLE IX. ENERGY PROFILE OF VIDEO DECODING ON THE ATOM.

Fraction %	Energy J	Location
25.38	475	th_decode_packetin
21.40	400	oc_state_frag_copy_list_mmx
7.41	138	oc_huff_token_decode
6.88	128	oc_frag_recon_inter2_mmx
5.55	103	oc_dec_residual_tokens_unpack
5.05	94.5	oc_idct8x8_mmx
3.69	69	oc_dec_mv_unpack_and_frag
2.56	47.9	oc_state_frag_recon_mmx
2.26	42.2	oc_frag_recon_inter_mmx
1.17	21.8	*...[sys_read]*
Total	1872	

TABLE X. ENERGY PROFILE OF VIDEO DECODING ON THE CORE2.

Fraction %	Energy J	Location
34.02	1477	th_decode_packetin
15.52	673	oc_state_frag_copy_list_mmx
9.49	412	oc_huff_token_decode
6.75	293	oc_frag_recon_inter2_mmx
6.24	270	oc_idct8x8_mmx
6.06	263	oc_dec_residual_tokens_unpack
2.80	121	oc_state_frag_recon_mmx
2.57	111	oc_dec_mv_unpack_and_frag
1.98	85.9	oc_frag_recon_inter_mmx
1.85	80.3	0x0000000021e6c0
Total	4342	

B. Video decoding

Video playback is a common and increasingly important use case for mobile devices. Here we show the energy profile of decoding a 720p Xiph Theora video, for both Atom and Core2 systems (Tables IX and X). As expected the Atom has a "flatter" profile among the CPU-intensive functions. A developer might further use these profiles to select functions that allow for a trade-off in video quality and energy spent.

C. Audio encoding choices and adaptation

Voice-over-IP applications can use different codecs to adapt to the current bandwidth constraints while maintaining high voice quality. Developers programming for energy-constrained devices also have to keep track energy consumed in the coding/decoding process.

To demonstrate how a developer might use *eprof* to compare the energy-efficiency of different codecs, we contrast the *eprof*-reported energy-efficiency of two popular VoIP codecs, the GSM full rate speech codec that operates at 13.2kbits/s, and the older G.726 ADPCM codec at 16kbit/s. As can be seen from Tables XI and XII (for the Atom case), the GSM codec uses more energy than G.726, while only providing a slightly lower bitrate. Surprisingly, the G.726 energy efficiency drops slightly for higher bitrates (result not shown). A developer could also program software that can adapt the codec at runtime not only to the bandwidth constraints, but also to the available energy.

TABLE XI. ENERGY PROFILE OF GSM ON THE ATOM.

Fraction %	Energy J	Location
18.40	11.7	Calculation_of_the_LTP_paramet
16.92	10.7	Short_term_analysis_filtering
14.77	9.39	av_resample
6.89	4.38	Gsm_LPC_Analysis
4.85	3.08	Gsm_RPE_Encoding
3.84	2.44	*. . . [sys_read]*
3.24	2.06	Gsm_Preprocess
2.32	1.47	audio_resample
1.55	0.986	memcpy
1.41	0.897	av_build_filter
Total	63.6	

TABLE XII. ENERGY PROFILE OF G.726 ON THE ATOM.

Fraction %	Energy J	Location
41.26	20.3	g726_decode
19.05	9.41	av_resample
4.87	2.0	*. . . [sys_read]*
4.38	2.16	g726_encode_frame
2.99	1.47	audio_resample
2.22	1.09	[ext4_readpages]
1.82	0.899	av_build_filter
1.62	0.8	memcpy
1.35	0.667	[copy_user_generic_string]
Total	49.4	

VI. LESSONS LEARNED

In the course of creating a working software energy profiler, we made several observations worth sharing. We first tried profiling for system energy using a full-system emulator, which turned out to be complex and imprecise. Following this, we turned to estimating energy using linear models, which turned out deceivingly easy to mis-train.

Experience with full-system emulation: Instead of hardware performance counters, the first version of *eprof* used a qemu-based full-system emulator to estimate CPU energy consumption, based on the intuition that different opcodes or opcode classes consume different amounts of energy. Using qemu's binary translation system, we added instrumentation that could attribute the estimated per-opcode energy to basic blocks, and a simplified cache model kept track of cache and memory accesses. The energy model was trained in advance using micro-benchmarks that consisted of a single opcode executed repeatedly in succession.

This method of training resulted in a largely imprecise energy model. Further investigation showed that single-opcode benchmarks would allow the CPU to pipeline execution and, due to superscalar execution, retire multiple instructions in parallel. The principal problem here is that one given benchmark would *always* exhibit the effect in its extreme, while another benchmark would inhibit the effect *entirely*.

Because this behavior is significantly more pronounced in micro-benchmarks than in general-purpose code, the energy estimate results in skewed, unreasonable values. By introducing a magic constant, we tried correcting for this problem by measuring the average instructions-per-cycle in general-purpose code, and adjusting our energy model accordingly.

Even then, we could not consistently achieve acceptable estimation errors for general-purpose benchmarks. Apart from the coarse correction for superscalar execution, we believe that the simplified emulated cache architecture could not capture all nuances in processor and memory access behavior, such as prefetcher operation or speculative execution.

Linear models are easy to get wrong: While working on the linear energy models for *eprof*, no matter if using hardware performance counters or software metrics for disk accesses, it became apparent that it is easy to obtain a linear model that, while appearing to be correct, turns out to be quite wrong.

We will illustrate the problem using a performance counter-based model. When training such a model, the naïve approach is to capture the total consumed energy (or average power draw) and the total number of hardware performance events; the linear model is then trained using values from multiple benchmarks. Given a large enough number of benchmarks, this approach will result in a generic model with acceptable error.

Yet, generally, this model will only be accurate for full benchmark runs; when applying this model to short capture periods to obtain a time series of the energy consumption (power draw), large discrepancies between model and measurements will emerge.

We believe that this is most likely due to averaging effects in the power draw and performance counter values which often cancel each other. In the end this leads to a model that appears correct when being checked against the same type of measurements, but breaks down when faced with a higher-resolution estimation task.

Based on other works that deal with energy or power modeling, both submitted for review or already published, we believe that others might also have fallen into this trap without noticing. Once realized, it is easy to address: avoid averaging or aggregating data when training or checking your models.

VII. RELATED WORK

The existing techniques for CPU energy profiling can be split into two distinct categories: In per-opcode modeling [8], [9], [10] an emulator estimates the energy based on the opcodes executed, leading to an accurate yet slow approach. Lower overhead is achieved by probabilistic profiling, which can either use an external power meter [11], [12], [13] or hardware performance counters [14], [15], [6], [16], [17]. *Eprof* uses a probabilistic sampling hardware performance counter model to estimate CPU and memory energy.

Several previous systems perform energy estimation on a full system scale [18], [19], [20], [21], [22]; however, they do not allow for fine-grained attribution of energy. *Eprof* provides a per-function attribution of both synchronous and asynchronous energy.

Pathak *et al.* [1] build a set of finite state machines that model the power behavior of the hardware used in their evaluation platforms. Their profiler for smartphones [23] is similar to

eprof in that they allow fine-grained attribution of energy; however, they require instrumentation of application source code and perform tracing at the system call level. *Eprof* has significantly lower overhead, works on unmodified binaries and obtains attribution information on the subsystem level where kernel requests are no more ambiguous.

VIII. Conclusion

Hardware is becoming increasingly energy-proportional and mainstream computing starts to encounter power delivery limits. It thus becomes crucial for developers to be able to obtain energy profiles of their code because dynamic energy accounts for the majority of the total system energy.

Unfortunately, traditional tools can only characterize energy behavior on a per-application basis, without being able to attribute energy to specific code locations or accurately attributing energy consumed by asynchronous devices. External instrumentation using a power meter cannot obtain a correct software energy profile; it only works when profiling exclusively for synchronous energy, but fails for asynchronous energy. Without recording of request provenance, the externally determined amount of asynchronously consumed energy cannot be mapped back to the originating software location.

To address these issues, we have designed and implemented *eprof*, a software energy profiler. *Eprof* makes it possible to calibrate a hardware platform once (using a power meter), and then use the calibration data to obtain energy profiles of the software running on that platform, without requiring the use of a power meter. *Eprof* accounts for the asynchronously consumed energy in device requests and attributes used dynamic energy to code locations. While we use this feature to study real-world scenarios involving energy used by a hard drive, we think that the techniques described in this paper can be applied to any asynchronous device; we identify generally applicable rules on how energy can be tracked back to code locations.

Eprof currently runs on the x86-64 platform and supports attribution of energy consumed by CPU, memory, disk, and wireless radios. The source code to *eprof* is available at `labos.epfl.ch/eprof`.

Acknowledgments

Simon Schubert is supported in part by a Microsoft Research Cambridge PhD Fellowship. This research was sponsored in part by a grant from Microsoft Corporation under the Microsoft Switzerland ICES program.

References

[1] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-Grained Power Modeling for Smartphones Using System Call Tracing," in *European Conference on Computer Systems (EuroSys 2011)*. ACM, 2011.

[2] D. C. Snowdon, S. M. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software - EMSOFT '07*. New York, New York, USA: ACM Press, 2007, p. 84.

[3] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang, "Modeling Hard-Disk Power Consumption," in *2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX, 2003.

[4] A. Hylick and R. Sohan, "A Methodology for Generating Disk Drive Energy Models Using Performance Data," in *HotPower'09*. ACM, 2009.

[5] A. Merkel and F. Bellosa, "Balancing Power Consumption in Multiprocessor Systems," in *European Conference on Computer Systems (EuroSys'06)*, 2006.

[6] D. C. Snowdon, E. L. Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for OS-Level Power Management," in *European Conference on Computer Systems (EuroSys 2009)*, 2009, pp. 289–302.

[7] A. Kansal and F. Zhao, "Fine-Grained Energy Profiling for Power-Aware Application Design," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, p. 26, Aug. 2008.

[8] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 437–445, 1994.

[9] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 13, no. 2-3, pp. 223–238, 1996.

[10] A. Sinha, N. Ickes, and A. P. Chandrakasan, "Instruction level and operating system profiling for energy exposed software," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1044–1057, Dec. 2003.

[11] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, 1999, pp. 2–10.

[12] F. Chang, K. Farkas, and P. Ranganathan, "Energy-Driven Statistical Profiling: Detecting Software Hotspots," in *Workshop on Power-Aware Computer Systems (PACS'02)*, 2002.

[13] C. Hu, D. A. Jiménez, and U. Kremer, "Combining Edge Vector and Event Counter for Time-Dependent Power Behavior Characterization," *Transactions on High Performance Embedded Architectures and Compilers (HiPEAC)*, vol. 2, no. 1, pp. 88–101, 2007.

[14] F. Bellosa, "The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems," in *ACM SIGOPS European Workshop*, 2000.

[15] A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2002.

[16] G. Contreras and M. Martonosi, "Power prediction for Intel XScale® Processors Using Performance Monitoring Unit Events," in *International Symposium on Low Power Electronics and Design*, 2005.

[17] C. Isci and M. Martonosi, "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data," in *36th International Symposium on Microarchitecture*. IEEE Comput. Soc, 2003, pp. 93–104.

[18] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," *HPCA*, 2002.

[19] W. L. Bircher and L. K. John, "Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events," in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2007, pp. 158–168.

[20] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A Comparison of High-Level Full-System Power Models," in *HotPower'08*. ACM, 2008.

[21] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-System Power Analysis and Modeling for Server Environments," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2006.

[22] A. Kansal, F. Zhao, N. Kothari, and A. A. Bhattacharya, "Virtual Machine Power Metering and Provisioning," in *ACM Symposium on Cloud Computing (SOCC)*. ACM, 2010.

[23] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?" in *European Conference on Computer Systems - EuroSys '12*. New York, New York, USA: ACM Press, 2012, p. 29.