EACOF: A Framework for Providing Energy Transparency to enable Energy-Aware Software Development

Hayden Field, Glen Anderson, Kerstin Eder
University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol,
BS8 1UB, UK
{hayden-field.2011, glen.anderson.2011, kerstin.eder}@bristol.ac.uk

ABSTRACT

Making energy consumption data accessible to software developers is an essential step towards energy efficient software engineering. The presence of various different, bespoke and incompatible, methods of instrumentation to obtain energy readings is currently limiting the widespread use of energy data in software development. This paper presents EACOF, a modular Energy-Aware Computing Framework that provides a layer of abstraction between sources of energy data and the applications that exploit them. EACOF replaces platform specific instrumentation through two APIs—one accepts input to the framework while the other provides access to application software. This allows developers to profile their code for energy consumption in an easy and portable manner using simple API calls. We outline the design of our framework and provide details of the API functionality. In a use case, where we investigate the impact of data bit width on the energy consumption of various sorting algorithms, we demonstrate that the data obtained using EACOF provides interesting, sometimes counter-intuitive, insights. All the code is available online under an open source license.
`http://github.com/eacof`

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*

General Terms

Design, Management, Measurement, Performance

Keywords

Abstraction, eacof, energy-aware computing, energy profiling, energy transparency

1. INTRODUCTION

Limiting energy use is important in situations such as robotics, portable devices and data centres. Traditionally, reductions in energy usage have been achieved through improved hardware design, however hardware designers must make conservative assumptions about usage patterns to ensure that their devices remain suitable for a wide range of use cases. There is a far greater potential for energy saving when the requirements of software are considered [8]. This is because the control over the computation ultimately lies within the software and algorithms running on the hardware.

Developers can produce more energy efficient software by implementing their data structures and algorithms appropriately, basing decisions on their knowledge of the needs of a particular application with respect to runtime, space utilisation and energy consumption. Knowledge of a user's specific usage patterns and priorities provides further opportunity to reduce energy consumption either by passively monitoring a user's interaction with a device and adapting accordingly [10], or by providing direct feedback about the energy needed to perform different tasks, enabling users to make informed decisions about their behaviour [12].

In order to leverage this domain specific knowledge to produce more energy efficient software, developers first need to know how much energy is required to execute their code. Energy usage profiles must be created separately for each targeted platform as it is not possible to generalise from one to another [6]. In some instances it is possible to create these profiles through the use of static analysis [3]; however, in general, dynamic measurement of energy usage is required due to the limited availability and usability of static techniques. While dynamic measurement techniques are more widely available than static analyses, the overhead of instrumenting code is greatly increased by the low level of abstraction at which energy usage is measured, analogous to requiring machine-specific code to measure execution time.

This paper presents EACOF, our modular **E**nergy **A**ware **CO**mputing **F**ramework, which provides a layer of abstraction between sources of energy data and the applications that exploit them, allowing developers to profile their code for energy consumption in a simple and portable manner. This abstraction replaces platform specific instrumentation through the use of two APIs. The first is used to make energy information available to other software in a portable way while the second is used to access energy information without the need for bespoke instrumentation on each platform. By separating these two key tasks in a modular manner, our framework is designed to both be easily extensible

and encourage the development of maintainable code.

EACOF is designed to be used on a general purpose multi-process Operating System with functional Inter-Process Communication capabilities. It allows sophisticated data collection methods to provide energy consumption data to applications through a simple API. The provided data can be as specific as per-process values for individual hardware components. This energy data may be utilised by a developer during the development process for energy consumption testing. Alternatively, it may provide the end user with indications about energy consumption at run time or allow the development of applications which adapt based on measured energy consumption.

The rest of this paper is structured as follows. Existing work upon which EACOF builds is discussed in Section 2. The characteristics of the different sources of energy information that are available to EACOF are considered in Section 3. The various components that make up EACOF, and their interactions, are described in Section 4. The APIs to allow interaction between components are specified in Section 5. In Section 6 we present a use case showing how EACOF can be used to analyse sorting algorithms to help a developer to select the most appropriate algorithm or data type for a given task. Section 7 concludes and gives an outlook on future work.

2. BACKGROUND

Tools, such as PowerScope [1], provide functionality to monitor the energy usage of code, while requiring additional hardware for this task. On Android devices, AppScope [13] estimates energy consumption of each process based on a static model; however, this data is not available to the application itself. The Intel Power Gadget API [5] allows for the measurement of energy consumption on modern Intel CPUs, although does not provide access to data from other hardware components. JouleUnit [11] is designed to be a generic framework, however has a specific focus on testing during development. The authors of Eprof [7] describe the challenges in attributing the energy usage of hardware to the correct application.

Each of these systems, many of which are designed for mobile devices, uses a bespoke method of instrumentation tailored to a specific source of data. With EACOF, we provide a standard method of accessing sources of energy data, including these and others, designed primarily for desktop systems. We also address the stated limitations of each existing source.

3. DATA SOURCE CONSIDERATIONS

For EACOF to succeed as an abstraction layer for sources of energy information, it is important that its design does not preclude the use of any particular data source since doing so would force developers to use platform specific instrumentation in their applications. Below we outline the considerations that different data sources place on EACOF. Section 4.1 shows how EACOF has been designed to meet these considerations.

Resolution. The operating system on a laptop might provide a new value for the current charge of the battery once per second, while counters built into a CPU may update thousands or millions of times per second.

Precision. The values regarding the current charge of a battery may only be specified to 3 or 4 significant figures within system files, while hardware counters can be significantly more precise.

Accuracy. While some data sources take measurements directly from hardware, others utilise a more indirect approach. Tools such as PowerTOP [4] are able to provide estimates of energy consumption based on usage statistics provided by the operating system, after a calibration period. Because the data is not gathered directly from hardware, accuracy needs to be taken into account when making use of the provided values.

Probes. A *Probe* is a means of defining one or more hardware sources of energy data (devices). Energy data can cover a range of devices. This can range from individual CPU cores to an entire system, with various steps along the way. Sometimes, a single hardware probe will be attached to multiple devices, causing the energy information provided to be an aggregate of a number of devices.

Units. The units in which data is collected can vary from one source to another—one source might provide energy data in Joules while another provides power data in Watts.

Temporal Continuity. Some sources of data may temporarily be unavailable when attempts are being made to use them. For example, plugging a laptop in to charge will eliminate the ability to use the discharge rate of the battery as a measure of whole system energy usage.

Proliferation. As we place greater demands on systems to be energy-efficient, we will see an increasing number of sources of energy data built directly into the hardware platforms we use. Many of these will have a higher resolution and accuracy than current data sources.

4. SYSTEM ARCHITECTURE

In this section we describe the components of our framework and how they interact with each other.

Providers (§ 4.1) abstract the details of a data source, making it available to higher level components in a portable manner. Consumers (§ 4.2) of energy data are able to access the information made available by Providers. The Central Authority (§ 4.3) marshals data between the Providers and Consumers as well as providing several additional services. Each of these components can be developed and compiled independently and run as separate system processes, providing a simple method of modularly extending the functionality of EACOF. Our device classification system (§ 4.4) provides a common method for the components of EACOF to describe the hardware being monitored. We have utilised various strategies (§ 4.5) to minimise the overhead incurred from use of EACOF.

A library, *libeacof*, is provided to abstract the mechanisms underlying the communication between components from the programmer, providing procedures that can be readily integrated into portable application code. The two APIs provided by *libeacof* are described in Section 5.

4.1 Data Providers

A *Provider* is software that provides energy information to EACOF. The Central Authority acts as a single point of contact between all components in EACOF. Figure 1 shows how a Provider acts as a wrapper for a source of energy information, collecting data in a platform specific way before passing this data to the Central Authority in a portable for-
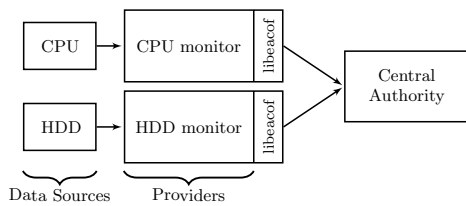
Figure 1: Central Authority with multiple Providers

mat. It is relieved of the need to separate out per-process data by the Central Authority, which undertakes this task on behalf of all Providers. Since Providers act as a wrapper for a data source, they are designed to handle possible variations between existing and future sources of data. The following describes our solutions to considerations about data sources:

Resolution. Providers push their data to the Central Authority rather than waiting for it to be requested. This ensures that sampling is performed in the most appropriate way for a given data source. For example, if a data source updates in a non-linear manner, the Provider can ensure that the Central Authority always has the latest data no matter what the gap between updates.

Precision. Use of *double*s for energy consumption data provides precision for decimal values with up to 15 significant digits, as specified in IEEE 754. This is the greatest level of precision obtainable with portable data types.

Accuracy. Since a Provider may encapsulate any data source, EACOF provides no guarantees about the accuracy of a Provider's data. It is up to the developer of a Provider to ensure the data output is as accurate as possible and the developers of Consumers to cope with data of varying accuracy.

Probes. Providers define *Probe*s that specify their functionality. Section 4.4 describes in detail the method by which devices may be defined within EACOF.

Units. It is a role of a Provider to convert the value gathered from its data source into Joules before passing it on to the Central Authority. This ensures that Consumers only need to handle a single unit, Joules.

Temporal Continuity. A Provider lets other components know when its capabilities change, either temporarily or permanently. Section 5.1 describes the API we have specified to allow this functionality.

Proliferation. New sources of energy information can be supported through the development of new Providers. These new Providers can be integrated into EACOF without the need to modify the Central Authority or pre-existing Providers due to the modular framework design. Our device classification system (§ 4.4) ensures that Consumers can automatically make use of new Providers without alteration.

4.2 Data Consumers

A *Consumer* is an application that makes use of energy data provided by EACOF. Figure 2 shows how the Central Authority is able to connect to a number of Consumers simultaneously. Each Consumer has no awareness of the Providers that feed energy information into EACOF—as far as it is concerned, all the information accessed is provided by the Central Authority itself.

Consumers determine their energy consumption through the use of Checkpoints. *Checkpoints* are a means of sampling
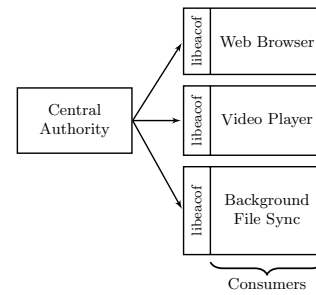

Figure 2: Central Authority with multiple Consumers

the energy used by a combination of one or more devices between multiple points in time. They are added into an application's code through the use of the Consumer API (§ 5.2), allowing the energy consumption of sections of code to be determined. There are three key points in the timeline of a Checkpoint: creating, sampling, and deleting.

To use a Checkpoint, it must first be created, setting a point in time to use as a baseline for the energy being measured. When creating a Checkpoint, the devices and processes that are to be measured must be specified. A Checkpoint can be used to measure the Consumer itself or all running processes. Upon creation of a Checkpoint, the Consumer must communicate with the Central Authority to acquire a unique identifier for the Checkpoint and to ensure that the requested devices can be monitored.

To obtain energy readings, a Checkpoint must be sampled. When a Consumer samples a Checkpoint, the Central Authority will provide the number of Joules used by the specified processes on associated devices since the Checkpoint was last sampled by the same Consumer. Creating a Checkpoint is deemed to be the first sample point.

It is possible for a single Consumer to have multiple active Checkpoints, each monitoring one or more devices. Likewise, it is possible for multiple Consumers to be monitoring the same device.
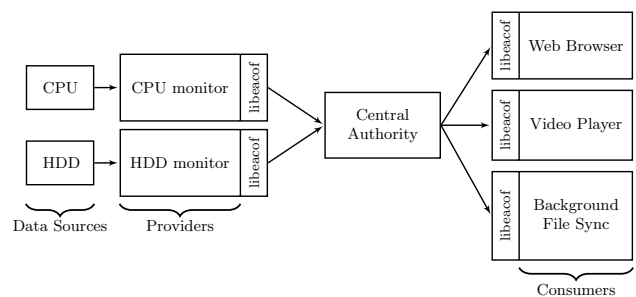
4.3 Central Authority

Figure 3: EACOF System Architecture

The *Central Authority* acts as a single point of contact between all other components in EACOF. Figure 3 shows how these components fit together, with data flowing from a data source all the way through to a Consumer, as directed by the arrows.

By communicating with all other components, the Central Authority is able to perform operations that Providers and Consumers would be unable to complete alone. For example,
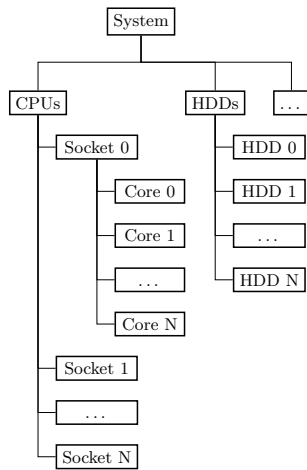
Figure 4: Classification of a subset of devices

let there be two Probes defined by Providers—one measuring both the CPU and Memory, the other measuring only the Memory. If a Consumer then defines a Checkpoint to track energy consumption of the CPU, the Central Authority is able to derive this requested information by reading data from both Probes and calculating the difference.

The Central Authority is also able to keep track of recent readings from multiple Providers and extrapolate from them to estimate future energy consumption even if the resolution of available Providers is low. A Consumer could perform this task, however, performing it in a central location makes it easier to integrate the full capabilities of EACOF into an application.

4.4 Device Classification

The definition of a particular device is key to a clear common understanding between components within EACOF. A device classification system must be precise enough that data received by Consumers meets their expectations while also being flexible enough to allow switching of Probes with equivalent functionality in a manner transparent to Consumers.

Figure 4 shows a subset of the classification system used by EACOF. While starting at a very high System level, our classification system provides scope to zoom in on specific components. Energy can be provided and consumed for devices at any level of this classification tree, for example at the whole System level, for all CPUs, or for a single core of a particular CPU.

The classification system sets expectations of *what* is being monitored rather than *how* it is being monitored. For example, readings from energy counters in the CPU in Socket 0 and the power supply to Socket 0 itself would both be classified as Socket 0.

It is the responsibility of a Provider to specify which devices the *Probe*s it contains represents. Similarly, it is up to each Consumer to make it known to other components which devices it is interested in obtaining data for.

4.5 Dealing with Overheads

When running with a single Provider sampling CPU energy consumption at a rate of 50Hz, EACOF increases system power consumption by less than 1 Watt on an 2012 MacbookPro 13" with a 2.5GHz Intel Core i5. This is lower than the idle power consumption of many common applications [9].

When EACOF is used to profile code during development, these overheads are of little concern as they will not be incurred by the final application. However, when EACOF is an integral part of the application it is vital that developers are able to mitigate these overheads when necessary. This can be achieved by sampling fewer devices less frequently or by instrumenting only the most critical sections of code.

The overheads incurred when using EACOF must always be weighed against the benefits. While static analysis can deliver results without runtime overhead, there are limits to what can be determined in this manner. In such situations, the only alternative is to use a method of dynamic analysis such as EACOF, where the inherent overheads are necessary to achieve the desired outcome.

The Central Authority plays a key role in minimising the overheads incurred when using EACOF by reducing the number of communication channels that need to be established. More formally, let the number of Consumers be C and the number of Providers be P. In a naive configuration, where Consumers communicate directly to Providers, $O(C * P)$ communication channels are required. This is reduced to $O(C + P)$ by introducing the Central Authority. In adding an extra step between the Provider and Consumer, latency is increased, however this design prevents a poorly designed Provider blocking a Consumer request for a long period—it is easier to ensure predictable latency within a single Central Authority than within many Providers.

5. INTERFACE

To allow components of EACOF to interact with each other we have developed two APIs—one for Providers, the other for Consumers. The Provider API (§ 5.1) is used to extend the framework with additional sources of energy information independently of the development and compilation of the framework itself. The Consumer API (§ 5.2) is used by software developers wishing to exploit the gathered information, either for the purposes of profiling or as an integral part of the design of their application—perhaps to provide energy awareness to end users. These APIs have been designed to work together and may be used simultaneously by a single application.

Our implementation of the API is available in an online repository at: `http://github.com/eacof`

5.1 Provider API

A *Probe* is an abstract data type representing a capability of a Provider and can be thought of as a virtual description of a physical Probe used to monitor one or more devices. The API functions shown in Figure 5 are used to create, manipulate and delete *Probe*s.

createProbe() is a variadic function used to specify the configuration for a *Probe*. *libeacof* will allocate a new *Probe* and populate it with the devices provided by the caller. A unique identifier for the new *Probe* is acquired from the Central Authority and the address of the new *Probe* is stored in the location pointed to by *newprobe*.

While a *Probe* is inactive to begin with, *activateProbe()* and *deactivateProbe()* may be used to toggle whether it is active. Once finished with, *deleteProbe()* will delete a *Probe* and stop further attempts at trying to access the data it provides.

```
1  int createProbe(Probe **newprobe, DeviceCount dcount, ...);
2  int deleteProbe(Probe **probe);
3  int activateProbe(Probe *probe);
4  int deactivateProbe(Probe *probe);
5  int addSample(Probe *probe, double joules);
```

Figure 5: Provider API Function Prototypes

```
1  int setCheckpoint(Checkpoint **newcheckpoint, ProcessSpecifier pspec, DeviceCount dcount, ...);
2  int sampleCheckpoint(Checkpoint *checkpoint, double *joules);
3  int deleteCheckpoint(Checkpoint *checkpoint);
```

Figure 6: Consumer API Function Prototypes

addSample() provides the amount of energy used, in Joules, by the devices associated with the supplied *Probe*. The sample is the amount of energy used since the latter of *a)* the last point at which the *Probe* was activated; or *b)* the last point at which a sample was added.

The return value of all API functions in this section and the following is an error code. This allows users of the API to know when a requested operation failed, making it clear when alternative action, such as retrying the function call or entering a non-energy-aware state, should be taken.

5.2 Consumer API

A *Checkpoint* is an abstract data type representing a Checkpoint as described in Section 4.2. The API functions shown in Figure 6 are used to create, sample and delete *Checkpoint*s.

setCheckpoint() is a variadic function used to specify a set of devices to be monitored along with the processes they should be monitored for, creating a Checkpoint with the given configuration. *pspec* should either be *ALL* or *SELF* depending on whether the caller wishes to monitor all running processes or itself respectively.

sampleCheckpoint() is used to query the amount of energy in Joules used by the process-device combination specified by *checkpoint* since the latter of *a)* the last call to *sampleCheckpoint()*; or *b)* the point at which the Checkpoint was created.

deleteCheckpoint() is used to signal that the *Checkpoint checkpoint* is no longer required, deleting it so it may not be used again until recreated.

6. USE CASE EXAMPLE

As an example, this section demonstrates how EACOF can be applied to gain insight into the energy usage of code. While energy-efficient software design is beyond the primary goals of EACOF, the framework makes energy consumption during computation transparent, so as to enable developers to gain an insight into the energy usage of code. This use case demonstrates that our framework provides data for developers to make more informed decisions about software energy consumption. It is not, however, designed to provide an exhaustive demonstration of all the capabilities of EACOF— a demonstration of multi-provider and multi-platform functionality is available in the online repository.

We use a Provider based on the Intel Power Gadget API, with a resolution of 20ms, to measure the energy usage of the CPU in Joules.

Our application is a Consumer designed to sort an array of integers in the range [0, 255]. The same array of numbers is represented using data types containing varying numbers of bits and sorted using a number of standard deterministic sorting algorithms. Checkpoints are set and sampled directly before and after the sorting occurs. This setup allows us to examine the difference in energy consumption caused by using different data types to perform a task, a typical problem that a developer may wish to use EACOF to solve. Since the focus is that of differences between data types, comparisons between rows should not be made for Time or Energy in Table 1.

We ran the code on a 2012 MacbookPro 13" with a 2.5 GHz Intel Core i5 and 8GB RAM running OS X 10.8.4 to gather results. All code was compiled on the target machine using the vendor's standard compiler with an optimisation level of *-O3*. The instrumented code used to gather the results is available in an online repository at: http://github.com/eacof

Table 1 shows the results gained from running our program. Time is measured in Seconds (s) and Energy is measured in Joules (J). The average power consumption, energy over time, is measured in Watts (W). Each algorithm was run with the specified number of elements 200 times for each data type. The displayed figures are the calculated means over all 200 runs. We keep run time similar by providing each algorithm a different length array as input, so as to allow measurement of algorithms of differing algorithmic complexities. Counting Sort takes significantly less time per run than the other algorithms because it is memory limited and would require an array containing around 4 billion elements to take a similar amount of time as the others. The standard deviation of all result values is low, apart from for Insertion Sort where the standard deviation for both Time and Energy is around 10% of the mean values.

The data collected with EACOF demonstrates that, against common intuition, time and energy consumption are not necessarily directly correlated. It can be seen in the cells marked with a ● in Table 1 that with Merge Sort and *qsort()* from *stdlib* [2], sorting 64 bit values takes less time than sorting values with fewer bits, however more energy is consumed in the process.

It can also be seen that the average power consumption when using a single data type varies between algorithms (★). In a situation where there is a limited power supply, it may be desired to choose an algorithm with lower power consumption even if it means the time taken or energy used is higher.

Another insight that can be gained is that the amount of energy used to perform a sorting task will generally increase as the number of bits in the data type increases. In our data, marked with a ♦, Counting Sort uses 75% more energy to sort the same 200 arrays of numbers when they are repre-

		Data Type											
		uint8_t			uint16_t			uint32_t			uint64_t		
		Total Time (s)	Total Energy (J)	Average Power (W)	Total Time (s)	Total Energy (J)	Average Power (W)	Total Time (s)	Total Energy (J)	Average Power (W)	Total Time (s)	Total Energy (J)	Average Power (W)
Algorithm	Num Elements												
Bubble Sort	50,000	5.53	66.66	12.03	5.39	65.29	12.09	5.66	69.05	12.19	5.78	71.83	12.41
Insertion Sort	200,000	7.98	■102.18	12.75	7.98	■103.00	12.85	7.46	■98.81	13.21	7.54	■105.03	13.89
Quicksort	2,000,000	5.51	61.73	11.20	5.53	61.90	11.19	5.52	61.60	11.15	5.51	62.90	★11.42
Merge Sort	60,000,000	●6.06	●72.33	11.93	6.07	72.46	11.93	6.12	75.65	12.36	●5.93	●76.98	★12.98
qsort	100,000,000	●5.84	●72.39	12.37	6.15	76.90	12.48	6.79	86.29	12.69	●5.69	●73.25	12.86
Counting Sort	200,000,000	0.23	◆2.92	12.75	0.24	◆3.16	13.23	0.25	◆3.58	14.15	0.35	◆5.12	14.44

Table 1: Comparison of the energy required to sort integers of different bit widths

sented as 64 bit rather than 8 bit values. Similar increases are not consistent for all algorithms—Insertion Sort broke an otherwise increasing trend when values were represented using a 32 bit data type (■).

Each of these examples highlight insights that can be gained by having access to the energy consumption information provided by EACOF. To take a step further, one would want to investigate why Insertion Sort uses less energy when using 32 bit values. This investigation is beyond the scope of this project, however the issue would not have come up without access to the data.

7. CONCLUSIONS AND FUTURE WORK

We have created EACOF, a framework which allows access to information about the energy consumption of software through the use of simple API calls. This allows the development of code that may portably provide energy transparency to both the end user and developer. In turn, it enables informed decisions to be made with respect to trade-offs regarding energy consumption.

EACOF is designed to separate the two tasks of collecting and utilising dynamic energy consumption data through the use of Providers and Consumers. This separation allows for a clear distinction of the two tasks, reducing the development and maintenance overhead required for the successful completion of each task. In addition, our modular design allows for functionality to improve over time without re-compilation of all existing system components.

As demonstrated in our use case (Section 6), our framework provides the functionality required to obtain values regarding software energy consumption.

We intend to continue developing EACOF and use it to study the improvements in energy-efficiency that can be gained in real-world applications. We also intend to perform analysis similar to our use case with a range of Providers on a variety of hardware set-ups with respect to other algorithms, data structures and software constructs. Alongside, we will perform further validation of the figures provided by EACOF to determine how best to obtain accurate data.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[2] B. Kernighan and D. Ritchie. *C Programming Language*, pages 251–253. Pearson Education, 1988.

[3] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy consumption analysis of programs based on xmos isa-level models. In *23rd International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, September 2013.

[4] Intel. PowerTOP. *https://01.org/powertop/*.

[5] Intel. Intel power gadget API. *http://software.intel.com/en-us/blogs/2012/12/13/using-the-intel-power-gadget-api-on-mac-os-x*, 2012.

[6] J. Pallister, S. Hollis, and J. Bennett. Identifying compiler options to minimise energy consumption for embedded platforms. *arXiv preprint arXiv:1303.6485*, 2013.

[7] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[8] K. Roy and M. C. Johnson. *Low power design in deep submicron electronics*, chapter Software design for low power., pages 433–460. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[9] B. Steigerwald, C. Lucero, C. Akella, and A. Agrawal. *Energy Aware Computing: Powerful Approaches for Green System Design*, chapter Impact of Software on Energy Consumption, pages 51–57. Intel Press, 2012.

[10] S. te Brinke, S. Malakuti, C. Bockisch, L. Bergmans, and M. Akşit. A design method for modular energy-aware software. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1180–1182. ACM, 2013.

[11] C. Wilke, S. Götz, and S. Richly. Jouleunit: a generic framework for software energy profiling and testing. In *Proceedings of the 2013 workshop on Green in/by software engineering*, pages 9–14. ACM, 2013.

[12] C. Wilke, C. Piechnick, S. Richly, G. Püschel, S. Götz, and U. Aßmann. Comparing mobile applications' energy consumption. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1177–1179. ACM, 2013.

[13] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, 2012.