# Energy-Aware Programming Utilizing the SEEP Framework and Symbolic Execution

Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat

Friedrich–Alexander University Erlangen–Nuremberg

## ABSTRACT

Resource-constrained devices, such as wireless sensor nodes, smart phones, tablet computers, and laptops most notably suffer from the limited amount of energy they have available. Yet, emerging battery technologies addressing this issue were unable to improve the situation significantly. This is well documented by rates of growth in the various technology areas. While clock speed, data storage, and data transfer rates have seen growth rates from factor $10^3$ to $10^6$ during the last three decades, battery life could merely be improved by a factor of $10^1$.

At the same time, research efforts led to energy-aware system software exploiting the available energy resources in the most efficient way. Static and dynamic optimizations for energy-aware execution have been widely explored. Though, writing energy-efficient programs in the first place has only received limited attention. To address this, we present SEEP, a framework which exploits symbolic execution and platform-specific energy profiles to provide the basis for *energy-aware programming* [1]. SEEP equips developers with the necessary knowledge to take energy demand into account during the task of writing programs.

## 1. MOTIVATION

Energy efficiency is one of the most important aspects of today's system software running on mobile and wireless systems. To exploit available energy resources, distinct approaches have been proposed over the last years. In general, they are grouped into *run-time driven* approaches and *static* approaches. Dynamic voltage and frequency scaling [2], sleep states, and resource accounting [3] belong to the former. Compiler optimizations, such as loop optimizations [4] and architecture-specific instruction set extensions [5] belong to the latter.

At present, optimizing an application for energy efficiency requires a developer to analyze the application's runtime behavior. This time-consuming task aims at preventing the program to wake the CPU and devices from sleep states and is achieved by adjusting timeouts, batching of periodic system activities, and the avoidance of polling operations.

A different approach compared to dynamic and static optimization methods targeting at energy-aware execution of program code is the option of *energy-aware programming*. It supports developers at implementing energy-efficient applications in the first place. Cooperative I/O, for example, enables developers to specify deadlines for I/O operations in order to permit energy-efficient data access [6]. In contrast to this, recent proposals for programming languages featuring approximation techniques to gain energy savings [7] are more generic.

In accordance with these recent works we see great potential in energy-aware programming. However, as developers are missing appropriate tooling support, writing energy-efficient programs is still a difficult task. With SEEP we present a framework to address this issue. Our approach gives developers an early insight into the base energy demand of their code at hand. To respect today's variety of devices, SEEP targets to provide energy estimates for heterogeneous target platforms quantifying the combined energy demand for CPU, memory access and I/O operations, even for platforms unavailable to the developer. These estimates facilitate developers to resolve energy hotspots and to refactor program code prior to deployment of their applications.

## 2. THE SEEP FRAMEWORK

SEEP is a three-tier framework exploiting symbolic execution and platform-specific energy profiles to aid energy-aware programming by analyzing source code at creation time. SEEP is composed of three main components (see Figure 1) which are

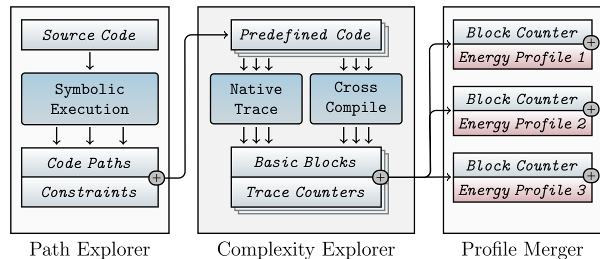


**Figure 1: The SEEP architecture**

consecutively executed: a path explorer, a path-specific complexity explorer, and a profile merger.

**Path Explorer.** SEEP executes the code under test symbolically using KLEE [8]. Applying symbolic execution techniques yields two essential results. First, we extract all possible *code paths*. Second, for each of these paths we obtain the corresponding *path constraints* (i.e., branch conditions).

**Complexity Explorer.** On basis of the results returned by the path explorer, SEEP crafts program code with predefined input data, so-called *path entities*. Each path entity belongs to exactly one code path and has a distinct set of predefined path constraints. Subsequently, SEEP performs a two-fold tracing phase. At the beginning of the tracing phase several distinct compilation runs for each path entity are executed. The code is not only being compiled for a powerful test system but also for each target architecture using a cross compiler. Next, SEEP generates a runtime execution trace for each path entity by executing them on the test system. During this execution SEEP increments a block counter for every basic block (branchless sequence of code) each time it is executed and stores the final result.

**Profile Merger.** By means of platform-specific energy profiles the framework finally calculates the expected energy demand for each path entity. An energy profile specifies how much energy is being consumed for each instruction of the platform's instruction set architecture. As result of the cross-compilation the structures of the basic blocks are known for all target architectures. Hence, each basic block's energy consumption can be calculated by adding up the specified energy value of each instruction of the basic block and multiplying these intermediate results with the block counters previously returned by the complexity explorer.

The final energy estimates are either passed to the developer or stored for future reference (i.e., offline usage). SEEP estimates the energy demand at function level and provides the same interface as the characterized function. For concrete input parameters it therefore estimates the energy demand for a specific target platform. For path entities which were not analyzed earlier, interpolation is used.

# 3. STATUS AND FUTURE WORK

For evaluating our prototype, we have executed distinct path entities for different code paths of our test application on an ARM platform (OMAP3530). The application has three entirely different code paths, each of them being a unique composition of commonly used code fragments (e.g., loops, if-then-else, and switch statements). Excerpts of the evaluation results are shown Table 1. The energy estimates varied by 0.089 mJ at the maximum with an average deviation of 0.017 mJ.

| *Path Entity* | *Energy (SEEP)* | *Energy (Measured)* |
|---|---|---|
| 1 | 2.520 mJ | 2.541 mJ |
| 2 | 0.591 mJ | 0.599 mJ |
| 3 | 2.361 mJ | 2.380 mJ |
| 4 | 0.015 mJ | 0.014 mJ |
| 5 | 1.721 mJ | 1.696 mJ |

**Table 1: Evaluation results**

With the presented prototype of SEEP we prove that it is feasible to exploit symbolic execution for energy-aware programming. Our initial evaluation results are promising as SEEP provides accurate estimates for the base energy demand of a program. However, we are in progress of extending the framework regarding different aspects in order to make it generally applicable. First, system-specific aspects need to be integrated into our energy model. We especially consider energy consumption caused by I/O operations and network links to be crucial. Second, non-deterministic factors such as memory effects (e.g., cache misses, page faults, and varying memory access modes) have substantial impact on the energy consumption and therefore need to be considered. Along with continuous improvements of the existing symbolic execution engines these enhancements of our framework allow us to create real-world scenarios when executing program code symbolically, resulting in reliable energy base cost estimations even for complex systems.

# 4. REFERENCES


[1] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat. SEEP: Exploiting symbolic execution for energy-aware programming. In *Proc. of the 4th Workshop on Power-Aware Computing and Systems*, 2011.

[2] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proc. of the 2002 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[3] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proc. of the 10th Conf. on Architectural Support for Prog. Lang. and Operating Systems*, 2002.

[4] V. Delaluz, M. T. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[5] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. of the 37th Annual Design Automation Conf.*, 2000.

[6] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proc. of the Symp. on Operating Systems Design & Impl.*, 2002.

[7] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Proc. of the 2011 Conference on Prog. Lang. Design and Impl.*, 2011.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th Symp. on Operating Systems Design and Implementation*, 2008.