SEEP: Exploiting Symbolic Execution for Energy-Aware Programming

Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg
{thoenig,ceibel,rrkapitz,wosch}@cs.fau.de

ABSTRACT

In recent years, there has been a rapid evolution of energy-aware computing systems (e.g., mobile devices, wireless sensor nodes), as still rising system complexity and increasing user demands make energy a permanently scarce resource. While static and dynamic optimizations for energy-aware execution have been massively explored, writing energy-efficient programs in the first place has only received limited attention.

This paper proposes SEEP, a framework which exploits symbolic execution and platform-specific energy profiles to provide the basis for *energy-aware programming*. More specifically, the framework provides developers with information about the energy demand of their code at hand, even for the invocation of library functions and in settings with multiple possibly strongly heterogeneous target platforms. This equips developers with the necessary knowledge to take energy demand into account during the task of writing programs.

1. INTRODUCTION

Energy efficiency is an essential goal of today's mobile and wireless systems and demands for a tight interaction between the system software (i.e., the operating system) and the hardware. In order to achieve this goal, various distinct approaches such as dynamic voltage and frequency scaling [1, 2], sleep states [3, 4], and resource accounting [5, 6] have been proposed. In contrast to these runtime-driven approaches, energy-aware compilers assist in optimizing code prior to execution. In particular, loop optimizations [7, 8] and architecture-specific instruction set extensions [9, 10] have proven to be effective to save energy. In addition to these static and dynamic optimizations for energy-aware execution there is the option of *energy-aware programming* that aids developers to implement energy-efficient programs 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 [11]. More generic are recent proposals for programming languages that feature approximation techniques to cut down computational costs, thereby leading to energy savings [12, 13].

In accordance with these recent works we see great potential in energy-aware programming. At present, however, writing energy-efficient programs is still complicated as developers are not aware of the energy demand of their code at hand. To estimate an application's energy costs usually requires running its code on a target platform while measuring the associated power consumption. However, a single iteration is inconclusive as a specific control flow graph of the application is selected which might have either a high or low energy demand compared to the average execution costs. Accordingly, multiple test runs with diverse input sets are required. At the same time, results of such measurement series are prone to have a high amount of jitter as unsettled background activities in user and kernel space have non-deterministic impact on the results.

While it is already difficult and time consuming to determine the energy demand for a single platform, the ever-increasing number of different hardware platforms[1] and the demand to implement applications for a multitude of them renders it almost impossible to carry out measurements for all platforms. Albeit developers might know the measured energy footprint for a given program on one platform, this knowledge often cannot be transferred

[1]The Linux kernel at present counts support for over 110 different platforms, for example.

to another even if the other platform is similar, as the availability of hardware function units (e.g., floating-point unit) can differ among them.

However, knowing the code's impact on energy costs would allow developers to refactor application logic to eliminate energy hotspots in the first place, for example, by disabling features when energy is scarce, simplifying code, or utilizing new energy-saving programming techniques [12, 13].

In sum, we consider high-level decisions in application design to be the crucial factor towards the energy efficiency of software beyond its current level.

In this paper we present SEEP, a three-tier framework, which uses symbolic execution and platform-specific energy profiles to aid energy-aware programming. Our framework introduces a tool-based feedback loop that equips developers with the energy demand of their program while developing. This is achieved by analyzing source code at creation time and providing consumption profiles for arbitrary pieces of code (e.g., libraries) that have been determined offline. To support heterogeneity at the platform level after runtime analysis, energy cost estimates are created using platform-specific energy profiles. The combined results are then provided to the developer enabling energy-aware programming.

Specifically, the key contributions presented in this paper are threefold:

• We present SEEP, a framework to enable energy-aware programming. By exploiting symbolic execution it provides developers with an early and meaningful insight into the expected energy consumption of their program code.

• Energy cost estimates provided by SEEP cover diverse, heterogeneous target platforms, including those unavailable to developers.

• Finally, we designed an easy to use hardware which allows analog energy measurements. It is presented in the evaluation section.

The paper is organized as follows. In Section 2 we present the architecture of the SEEP framework. Section 3 discusses the implementation of our prototype, Section 4 outlines results from our evaluation, and Section 5 concludes and discusses future directions of our research.

2. ARCHITECTURE

The SEEP framework is composed of three main components (see Figure 1) that are consecutively executed: a path explorer, a path-specific complexity explorer and a platform energy profile merger.

The path explorer analyzes program code by applying symbolic execution techniques. It extracts all possible code paths and their corresponding path constraints (i.e., branch conditions). This allows us to further explore the code's behavior at runtime as a next step. The path-specific complexity explorer automatically generates and executes specially crafted binaries originating from the developer's program code, augmented with predefined inputs as received by the path explorer component. To quantify the complexity of the code under investigation, we then create execution traces. During this tracing period SEEP increments a *block counter* for every basic block (branchless sequence of code) each time it is executed. The final result of the block counters unveil how often each basic block has been executed during the tracing period. Finally, the profile merger component combines platform-specific energy profiles with the block counters of the tracing period to produce *energy consumption estimates* for the analyzed code. Thereby, an estimate quantifies the energy demand at function level and provides the same interface as the characterized function. For concrete input parameters it therefore estimates the expected energy demand.

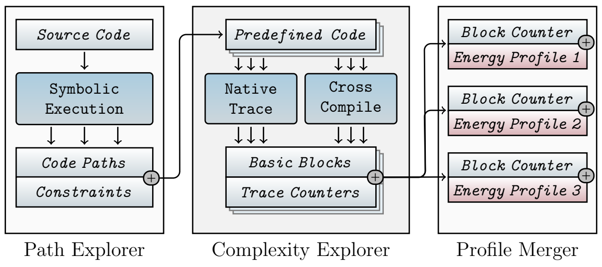

Figure 1: The SEEP architecture

To minimize the analysis time of the code under development, SEEP supports two modes of operation: First, our framework analyzes the developer's program code and explores its complexity and runtime behavior in terms of energy demand. Second, SEEP estimates the expected energy consumption of library calls by means of energy consumption estimates. The latter have previously been determined and are provided by the corresponding libraries.

3. IMPLEMENTATION

In this section, we discuss the implementation of SEEP. Our current prototype utilizes KLEE [14] as a basis for supporting symbolic execution. All tasks of the SEEP framework but the creation of energy profiles are entirely architecture independent and platform independent. Therefore, they can be performed on an arbitrary system different from the target platforms, which is convenient as the LLVM framework required by KLEE might not be available for some of them.

3.1 Code Path Exploration

With SEEP we exploit symbolic execution in two distinct ways. First, executing code symbolically enables us to extract all possible *code paths* of the application under test (see Figure 2a). As symbolic execution is usually brought into action for unattended discovery of defects in program code, testing frameworks based on symbolic execution provide high code coverage by default.

Second, we extract the associated *path constraints* (i.e., branch conditions) which are specific for each code path. This information allows SEEP to create various predefined binaries for the extracted code paths. It is important to note that different binaries of the very same code path can exhibit entirely different energy costs, depending on the actual values of their path constraints. That is why SEEP analyzes multiple different predefined binaries for *each* of the extracted code paths.

3.2 Crafting Path Entities

Each code path has specific path constraints defined by a set of parameter values under which it will be executed. These path constraints in turn are a composition of individual constraints which potentially have an impact on the energy costs of the given program code. First, at conditional branches, a constraint determines which branch of the code is being taken. Second, path constraints have a shaping characteristic on the runtime behavior in terms of energy demand, for example, depending on whether the constraint in question is used as loop variable or not. This information is required by SEEP to further reason about the diverging energy footprint of two or more predefined binaries of the same code path.

SEEP crafts predefined binaries for each code path by exploiting the path constraints. Each predefined binary corresponds to exactly one code path *and* one predefined set of constraints, which are in effect for the specific code path. These distinct entities are called *path entities*. The number of path entities for each code path is determined by the amount of path constraints and their parameter range.

By using predefined input data for the path constraints which are aligned to their corresponding parameter range, for example, by using the upper and lower boundaries, and evenly distributed intermediate values, SEEP crafts and analyzes a subset of significant path entities. Figure 2b shows four distinct path entities for code path B. The differing size of the bottommost node denotes different runtime behavior in terms of energy demand compared to the other entities of the same code path.

```
if $i$ mod 2 then
  if $i$ mod $m$ then
    ... /* Path A */
  else
    ... /* Path B */
  end if
else
  if $i$ mod $n$ then
    ... /* Path C */
  else
    ... /* Path D */
  end if
end if
```

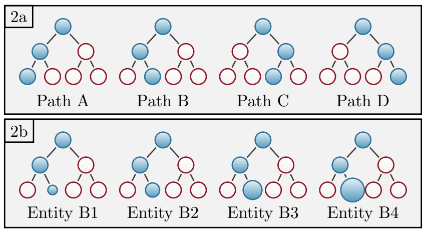

Figure 2: Code paths and code entities

3.3 Tracing Path Entities

The source code of path entities yet does not yield any insight regarding the expected energy footprint of the binary on the target platforms. SEEP addresses this during a twofold tracing phase, respecting the fact that target platforms indeed are likely to be heterogeneous (e.g., different instruction sets).

At the beginning of the tracing phase, SEEP executes several distinct compilation runs. On the one hand, SEEP compiles the code of the path entities for a powerful test system which is used for resource-intensive execution runs. On the other hand, the same code is compiled for each architecture of the target platforms using a cross-compiler. Next, SEEP generates one runtime execution trace for each path entity by executing them on the test system.

By means of execution traces SEEP can infer the runtime behavior on the target platform by examining how often each basic block was executed. For this we have applied a similar strategy as described in [15]. While the structure of the basic blocks is likely to differ, the number of times each basic block is being executed is the same across all platforms. Accordingly, this information can be utilized to determine the energy demand for the target platforms as detailed in the next section.

3.4 Energy Profile Fusion

For this step SEEP uses an energy profile specific to each target platform. An energy profile specifies how much energy is being consumed for each instruction of the instruction set architecture. Such a profile needs to be created only once per target platform and can be distributed independently.

As the structure of the basic blocks for the target architectures are known as a result of the cross-compilation, each basic block's energy consumption can be calculated by adding up the specified energy value of each instruction of the basic block utilizing the platform's energy profile. The expected energy consumption is calculated by multiplying the block counters with the energy costs of the corresponding basic block. A path entity's expected energy consumption then is obtained by adding up these interim results. From the calculated energy costs of the path entities SEEP interpolates the energy cost for arbitrary path entities (i.e., path entities not analyzed during the tracing phase). The final results are either passed to the developer or stored for future reference (i.e., offline usage).

4. EVALUATION

For carrying out the evaluation, we used two similar hardware platforms, which—contrary to their common characteristics—have differing energy requirements. We have verified our approach with a test application consisting of three different code paths, each of them being a unique composition of commonly used code fragments (e.g., loops, if-then-else, and switch statements). To obtain most accurate measurement results, we used a custom circuit board allowing analog measurements.

First, we created energy profiles for each platform; second, we created energy consumption estimates using SEEP for each of the test application's code paths at different work loads. Third, we measured several real execution runs and compared these with the estimates calculated by our prototype.

4.1 Platforms under Test

For evaluation purposes we used two evaluation boards with ARM processors of the OMAP3 microprocessor family (OMAP3530 and DM3730). These widespread processors power a broad range of handsets, including devices built by manufacturers like Motorola and Samsung. On our evaluation platforms, we intercept the main power rails of the power management processor which allows us to exclusively measure the combined power consumption of processor and main memory.

4.2 Energy Measurement Setup

Power consumption is calculated from multiplying the supply voltage by the drained current. Commonly, this current is measured either directly using an ammeter or indirectly by measuring the voltage drop of the current flow at a shunt resistor using a voltmeter. Integrating the power consumption over time then results in the energy usage. However, this integration step causes measurement errors as the measurement device needs to sample. While off-the-shelf multimeters only sample at a couple of hertz, sample rates of the order of gigahertz are not unusual for more expensive oscilloscopes. Yet, the measurement device needs to sample for providing a measured value and the odds are that there is momentary power consumption in between samples, which will *not* be part of the result. Due to this weakness of today's best practices, we chose an alternative approach initially proposed by Kostantakos et al. [16] and designed a custom circuit board based on a current mirror. The unique characteristic of this circuit is that it allows energy measurements without error-prone sampling intervals. We kept the basic concept untouched but dimensioned it for higher power drainage of our target hardware.

The measuring device works as follows. A current equivalent to the one drained by the device under test is being generated and flows through a current mirror. Two capacitors under the control of a flip-flop are being charged and discharged on an alternating basis and the corresponding switching events of the flip-flop are counted. At the end of a measurement series, the energy consumption is calculated under the knowledge of the duration of the measurement series, the number of switching operations, and the final potential of the last capacitor being charged.

In sum, our setup does not sample the measured power consumption, therefore enabling an analog measurement that leads to most accurate results as no power consumption goes unnoticed. Furthermore, it can easily be monitored by an external system and has low acquisition costs.

4.3 Energy Profile Generation

For each of the two evaluation platforms we have created energy profiles stating the average energy cost for each instruction. Both CPUs implement the Cortex-A8 processor design and therefore share the same instruction set (ARMv7). However, the processors have a different power consumption behavior as they are clocked at different speeds (720 MHz vs. 1 GHz) and their die is assembled with different semiconductor technology (65 nm vs. 45 nm).

In line with [17], we have measured the energy consumption of the instructions by executing loops of known length and recording the energy consumption for the total run. The average energy consumption of each instruction then is used for building the platform's energy profile. Although the DM3730 is almost 300 MHz faster compared to the OMAP3530, its energy consumption is lower, which can be attributed to the enhanced semiconductor technology.

4.4 Results

We have executed ten path entities for each code path of our test application. SEEP created two path entities (see Section 3.2) using the upper and lower bounds of the corresponding code path and eight intermediate path entities. Eventually, our framework interpolated the energy costs for all path entities of each code path.

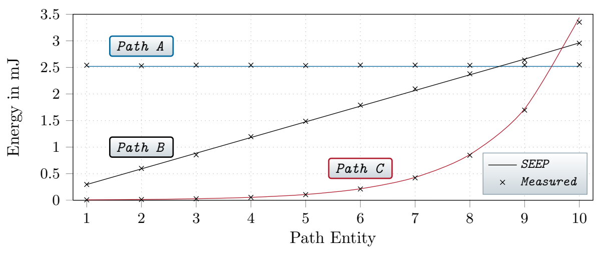

Figure 3: Energy Costs by Path Entity

Code path A exposes a constant energy consumption, as its path constraints are neither used in a loop construct nor does the code path vary its operations depending on the actual path constraints' values. In contrast, code path B and C show a linear and exponential energy consumption, respectively. In code path B, one of the path constraints impacts a loop construct as it is used as a loop test expression for the inner-most loop. For code path C, an exponential amount of computing operations is being executed depending on one of the values of the effective path constraints.

To verify the correctness of SEEP, we measured the energy consumption whilst executing predefined binaries on the DM3730 platform. SEEP's energy estimates and the measurement results are shown in Figure 3. Compared to the measurements, the predicted energy consumption varies by 0.089 mJ at a max with an average deviation of 0.017 mJ.

5. CONCLUSION & FUTURE STEPS

This paper presented SEEP, a framework to aid energy-aware programming. It exploits symbolic execution, selective run-time measurements, and platform-specific energy profiles to provide energy demand estimates during the task of programming. Despite its multi-platform support it is largely platform independent. In sum, we consider SEEP as a key component to make future software energy-efficient prior to deployment.

Today, SEEP provides accurate estimates for the base energy demand of a program. We will extend the framework regarding different aspects in order to make it generally applicable. First, we will integrate further system-specific characteristics into the platform profiles. Especially energy consumption caused by I/O operations and network links need to be considered. Second, memory effects such as cache misses [18], page faults, and varying memory access modes [19] potentially have impact on the energy consumption and thus need to be incorporated. Besides, SEEP uses ongoing research efforts targeted at expanding the scope of symbolic execution [20]. Along with the aforementioned extensions, these efforts assist us to construct real-world scenarios when executing program code symbolically. We envision the integration of SEEP into existing integrated development environments.

6. REFERENCES

[1] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. of the 1st Conf. on Operating Systems Design and Implementation*, 1994.

[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] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12), 2007.

[4] E. Le Sueur and G. Heiser. Slow down or sleep, that is the question. In *Proc. of the USENIX ATC*, 2011.

[5] 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.

[6] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proc. of the 1st Symp. on Cloud Computing*, 2010.

[7] 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.

[8] V. Delaluz, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu. Compiler-directed array interleaving for reducing energy in multi-bank memories. In *Proc. of the ASP Design Automation Conference*, 2002.

[9] 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.

[10] N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, 2000.

[11] 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.

[12] 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.

[13] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the 2010 Conference on Prog. Lang. Design and Impl.*, 2010.

[14] 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.

[15] S. Penolazzi. *A system-level framework for energy and performance estimation of system-on-chip architectures.* PhD thesis, KTH, 2011.

[16] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on Instrumentation and Measurement*, 57(4), 2008.

[17] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2), 1996.

[18] B. F. Ramon, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Intl. Conf. on Parallel Arch. & Compilation Techniques*, 1999.

[19] N. D. Lane and A. T. Campbell. The influence of microprocessor instructions on the energy consumption of wireless sensor networks. In *Proc. of the 3rd Workshop on Embedded Networked Sensors*, 2006.

[20] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proc. of the EuroSys 2011 Conf.*, 2011.