Proactive Energy-Aware Programming with PEEK

Timo Hönig, Heiko Janker, Christopher Eibel, Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Oliver Mihelic, Rüdiger Kapitza
TU Braunschweig

Abstract

Optimization of application and system software for energy efficiency is of ecological, economical, and technical importance—and still challenging. Deficiency in adequate tooling support is a major issue. The few tools available (i.e., measurement instruments, energy profilers) have poorly conceived interfaces and their integration into widely used development processes is missing. This implies time-consuming, tedious measurements and profiling runs and aggravates, if not shoots down, the development of energy-efficient software.

We present PEEK, a systems approach to proactive *energy-aware programming*. PEEK fully automates energy measurement tasks and suggests program-code improvements at development time by providing automatically generated energy optimization hints. Our approach is based on a combined software and hardware infrastructure to automatically determine energy demand of program code and pinpoint energy faults, thereby integrating seamlessly into existing software development environments. As part of PEEK we have designed a lightweight, yet powerful electronic measuring device capable of taking automated, analog energy measurements. Results show an up to 8.4-fold speed-up of energy analysis when using PEEK, while the energy consumption of the analyzed code was improved by 25.3 %.

1 Introduction

A holistic approach to optimize today's computer systems for low energy demand includes, but is not limited to, exploiting hardware energy saving features [1, 2]. In fact, it goes far beyond hardware concerns, which is why energy-aware software design is now in the spotlight and subject of current systems research, in general. To optimize software for energy efficiency, several distinct approaches have been proposed. Semi-automatic compiler optimizations [3, 4, 5, 6] at architecture level are transparent to the development process, but energy saving features at platform level (i.e., dynamic voltage and frequency scaling, sleep states) are exploited only partially. Recent research has focused on optimizing program code at a higher level of abstraction, such as programming languages [7] or runtime environments [8]. However, using specially designed programming languages or runtime environments implies radical changes for existing software projects, which prevents widespread adoption.

Today, developers ultimately have two options for energy analysis of program code. Either they measure the energy consumption using an electronic measuring instrument (i.e., multimeter, oscilloscope) [9, 10, 11] or they analyze their program code using an energy profiler [12, 13, 14, 15]. Either option is extremely time-consuming as both options require manual operations executed by developers. For example, a developer needs to complete many analysis iterations to pin down the culprit of an *energy fault*[1]. These manual tasks are required as electronic measuring instruments and existing energy profilers are controlled by standalone applications which are poorly embedded into development toolchains or integrated development environments, if at all. Hence, compared to established software analysis facilities such as profilers for runtime optimizations (e.g., GNU gprof), the ease of use of existing tooling infrastructure is not direct and straightforward but complex and cumbersome.

Increasingly often, software-based energy profilers cannot even be used since the required energy models for target hardware platforms are unavailable. As technology rapidly advances, it is a direct consequence of the tremendous complexity of today's hardware architectures and applies especially for embedded systems. For example, as of this writing, even the smallest and most energy-efficient [16] ARM processors (i.e., ARM Cortex-M0+) already have a two-staged core pipeline

[1] An *energy fault* is the root cause for unnecessarily high energy consumption that may result in a runtime *error* (deviation from target or actual) or even entails *failure* (breakdown).

and their implementations come with various dynamic energy saving features (e.g., different low-power modes). Establishing trustworthy energy models for systems with a similar or more complex architecture is difficult as, among other aspects, inter-instruction effects, execution modes (e.g., out-of-order execution), and integrated peripherals (e.g., memory controller) need to be considered [17]. Eventually, inaccurate or missing energy models lead to the desperate situation where developers are unable to use software tools to reason about the energy consumption of their program code. Instead, one has to manually run tedious energy measurement series using hardware energy measurement instruments.

Despite recent efforts, a breakthrough for practical tooling support suitable for assisting developers during the task of energy-aware programming is still missing. Developers require solutions that fit into their software development environments and workflows rather than intrusive approaches, which are cumbersome (if at all possible) to integrate. Intrusive approaches are particularly unattractive as they are prone to introduce new functional faults. Preferably, tooling support for energy-aware programming smoothly fits into existing development environments and goes completely unnoticed during software development as long as it is not used actively: just as profilers for runtime optimizations are basically invisible to developers until activation. However, once activated, profound tooling support for energy-aware programming needs to assist developers in various respects. First, developers should receive guidance at identifying energy faults in their program code (e.g., by means of energy consumption data). With such energy consumption data, developers pin down energy-intensive regions of their program code and can further analyze whether the energy consumption is actually justified (e.g., CPU-intensive section) or the code contains an energy fault. This requires a high degree of automation as developers should no longer be required to run through lengthy manual energy analysis. Second, not only data on the energy consumption of program code is of interest to developers. They are especially keen to obtain hints and suggestions on *how* to actually improve their code at hand for energy efficiency. Challenging these two aspects enables developers to proactively design energy-aware applications right at development time—we refer to this as *proactive energy-aware programming*.

In this paper we present PEEK, a systems approach to proactive energy-aware programming. PEEK improves the development process of energy-aware software at various levels. PEEK makes the following contributions: First, a tooling infrastructure for *automatic function-level energy analysis* of source code. Second, a tool that not only provides energy consumption values of program code but also reports *energy optimization hints* directly to developers. Third, for fully automated and accurate energy measurements, an *energy measurement instrument* in the form of a lightweight and yet powerful electronic measuring device. All core components of PEEK will be released under an open-source license.

The paper is organized as follows. In Section 2 we discuss the design and system architecture of PEEK. Section 3 presents our work on energy optimization hints. The implementation of the software and hardware components is presented in Section 4. We evaluate PEEK in Section 5, detail future work in Section 6, and discuss related work in Section 7. Section 8 concludes our work.

2 Design and System Architecture

PEEK is a proactive energy-aware development kit which allows programmers to automatically measure the energy consumption of program code at function level. To assist developers at writing energy-efficient programs, it is inevitable from the very beginning to keep them well informed of the expected energy consumption. Hereby, developers can reason about whether specific changes to their program code have a positive or negative impact on the energy consumption of their application. PEEK's central design objectives are *automatization* and *adaptability*. Thus, we have designed our system to eliminate manual operations commonly required for energy measurements while hooking the required additional components of our system into existing software development infrastructures without interfering with established software development processes.

We designed our system to be modular for three reasons. First, to adapt to existing software development infrastructure we reuse existing tooling components without modifying them (e.g., integrated development environments, energy analysis tools, source code management systems). Second, energy analysis carried out by software energy profilers potentially involves resource-intensive analyses that exploit parallelism by distributing tasks to different computing nodes. Third, as availability of energy measurement instruments may be limited, a modular system structure enables multiple developers to use a single instance of a measurement setup.

Accordingly, we detach tasks of energy-aware programming from the actual energy analysis operations. As a result, our tooling infrastructure is divided into front-end, middle-end, and back-end components. The front end implements an interface for the developer and software development toolchain, it offers a well-defined API for this purpose and controls the processing of data involved during the energy analysis (i.e., source code, meta data, and energy analysis results). The back-end components implement and perform the actual energy analysis. As energy analysis is specific to target hardware

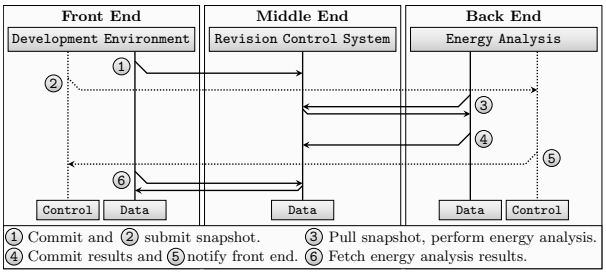

Figure 1: Overview of the PEEK system architecture and workflow: our system detaches energy-aware programming tasks from complex energy analysis operations.

platforms, PEEK does enforce nothing but the requirement that energy analysis must be performed at function level. This permits the implementation and use of different energy-measurement and energy-estimation techniques by different PEEK back ends. A back-end implementation is hardware-platform specific and implements energy-analysis support for source code written in one or more programming languages. PEEK itself is agnostic to the programming language of the source code processed during energy analysis. Conceptually, the middle end of PEEK is a passive component of our system and maintains all data the front end and the back end are working on during the analysis and furthermore archives previous energy analysis data. Figure 1 shows an overview of the modular PEEK system architecture.

The individual components of PEEK embrace and abstract software components of existing development environments to exploit their already available functionality. This eases the adoption of our approach as software developers do not need to modify their existing development infrastructure and established software development processes remain unchanged. A PEEK front end is implemented as part of an integrated development environment, such as Apple Xcode, Eclipse, or an advanced programming editor (e.g., Emacs, vi), and implements the actual user interface for developers. A PEEK back end is a stand-alone component that performs energy analysis tasks and generates energy optimization hints. There are two different types of PEEK back ends: software–hardware back ends and software-only back ends. A software–hardware back end wraps controlling mechanisms of an energy measurement instrument (i.e., multimeter, oscilloscope) [9, 10, 11], whereas a software-only back end abstracts interfaces of existing energy profilers [12, 13, 14, 15]. Both back-end types make energy measurement results available to developers through the unified interface of the front end. As energy measurement tasks potentially comprise complex operations, we use parallelization techniques and optionally run the front-end and back-end components on different nodes. On one hand, this allows us to exploit parallelism features of energy profilers used by software-only back ends. On the other hand, developers do not require local access to energy measurement instruments when working with software–hardware back ends, as the energy measurement instruments can be operated remotely. In general, the PEEK middle end is a passive component used to let both front end and back end exchange data using a shared storage layer. PEEK uses the revision control system Git [18] as middle-end infrastructure and exploits its features for snapshot mechanisms. Our system is designed to exchange components—as currently used by our implementation of PEEK—for different ones. For example, a different middle-end implementation may use another revision control system (e.g., Mercurial [19]).

2.1 Snapshot-based Workflow

The front end coordinates the source code analysis performed by the back end using remote commands. Figure 1 shows the sequence of actions of the workflow. To submit source code for analysis purposes developers create snapshots of their current work. For a submission, they first prepare a version of their source code and commit it to the middle end ① as a new revision. A snapshot includes the source code and meta data which, for example, informs the back end about how the source code is built and which target platforms are applicable. The latter is necessary as energy analysis is platform specific and energy analysis tasks are dispatched to different back-end components according to this information.

After the developer has committed a snapshot, the front end subsequently sends a control message ② to instruct the back end to analyze the corresponding snapshot. After pulling the snapshot ③ from the middle end, the actual energy analysis is performed. Once the analysis completes, the back end submits the energy analysis results ④ to the middle end and notifies ⑤ the front end about the availability of the results. Eventually, the front end pulls the energy analysis results ⑥ from the middle end and passes them to the developer.

Software developers can take influence on the operation of the back-end components by explicitly defining which functions of the submitted source code should be analyzed. This mechanism is used to cut down energy analysis efforts to the extent necessary. It is especially useful when source code of complex applications with a large number of functions is analyzed.

2.2 Multi-Snapshot Analysis

In the simplest case a developer submits a single snapshot s_1 for energy analysis. This yields an energy analysis for the specific version of the source code as registered by s_1. However, developers are keen to compare

the energy consumption of different versions of their program code (i.e., different implementations of one and the same function) as this allows them to judge the impact of their programming decisions on the energy demand of their code. To address this, we use *snapshot bundles*.

With snapshot bundles it is possible to aggregate multiple source code snapshots and examine them jointly in a single batch operation. A snapshot bundle $\mathtt{B}_{id}$ is a collection of multiple snapshots which share a unique identifier $\mathtt{B}_{id} := \langle s_1, s_2, s_3, \ldots, s_n \rangle_{id}$. Developers aggregate an arbitrary number of snapshots (e.g., representing different versions of their program code) and group them as a snapshot bundle. Rather than individually instructing the back end to analyze each snapshot, the front end dispatches a single batch evaluation. This is triggered by a remote command which includes the identifier $\mathtt{B}_{id}$ of the snapshot bundle. The back end then pulls all snapshots belonging to the bundle and analyzes each snapshot. At this level, back-end components can exploit parallelism, for example, by distributing energy analysis tasks of each snapshot to different nodes. When the energy analysis has finished, all results are written back to the meta data of each snapshot of the bundle. Eventually, the front end proceeds and provides the results to the developer. Exemplarily, the results for a single snapshot implementing the Advanced Encryption Standard (AES) on an ARM Cortex-M0+ platform are shown in Table 1.

It is mandatory for all PEEK back ends to provide energy analysis results at function level. Additional results (e.g., invocation count) are optional and specific to each back end. We use reflection techniques to process such optional data in the front end. When passing energy analysis results of a multi-snapshot analysis to the developer, the front end calculates the energy offsets between each pair of snapshots belonging to the same snapshot bundle. This makes it convenient for developers to compare different source code versions regarding their energy consumption footprint and serves as decision-making basis during energy-aware programming.

2.3 Input Data

Beside the source code itself, it is necessary to pass additional data and configuration options from the front-end components to the back-end infrastructure of PEEK. Most importantly, it is required to supply the back-end components with all data necessary to execute the program code (e.g., input parameters, runtime options, data sets). Furthermore, it requires a well-defined path to hand over source-code specific configuration options from the developer to the back end (e.g., build parameters, compile-time options). We pass such input data from the front end to the back end via the middle-end components as part of a snapshot.

Function Name	Energy	Invocation Count
`aes_mixColumns`	1.14 µJ	13
`aes_expandEncKey`	57.69 µJ	14
`aes_subBytes`	84.56 µJ	14

Table 1: Energy analysis of an AES encryption (256-bit key size, 16-bytes data) on an ARM Cortex-M0+ [22].

Functional testing of incremental changes is a common approach to ensure the correctness during the evolution of a software project [20, 21]. Unit testing provides an efficient way for developers to implement test cases which are executed automatically during the development process, for example, when source code changes are applied. However, unit tests commonly only verify functional requirements of program code (i.e., correctness). With PEEK we extend the scope of unit testing mechanisms to also verify non-functional properties of program code (i.e., energy consumption).

In order to efficiently supply input data to the back-end infrastructure, we have designed PEEK to augment existing software development technologies for this purpose. Apart from passing bare call parameters using the meta data of a snapshot, PEEK utilizes unit testing tooling infrastructure (e.g., Google Test, LLVM LIT, JUnit) for this purpose. Unit tests are passed to the back end which will hereinafter use this input data to execute the program code during energy analysis. Existing unit tests previously only used for functional verification can be reused for energy analysis without further modification.

2.4 Scalability Aspects

The energy analysis of an application commonly is a resource-intensive task. After building the source code, the application to be analyzed needs to be executed. In order to gain energy analysis results for different runtime configurations as defined by varying input data (e.g., parameters, unit tests), it is even necessary to run the application several times during the energy analysis process. To address this concern, back-end components need to implement suitable optimizations to reduce the analysis time, for example, by exploiting parallel execution.

The back-end components leverage snapshot bundles to exploit parallelism at snapshot and bundle level. If a back end provides corresponding support, we split up the analysis process by distributing the evaluation for each snapshot to individual nodes. Efficient grouping of snapshots paired with distributing the energy analysis to several nodes allows us to proactively assist developers at the task of energy-aware programming, because lengthy round trips—as they occur when profiling applications manually—are eliminated to the greatest extent. We outline the speed-up of the analysis process in Section 5.3.

3 Energy Optimization Hints

To reduce the energy consumption of software, developers are interested to obtain generated hints and suggestions on *how* to actually improve their code for energy efficiency. With PEEK we present the concept for *energy optimization hints* to address this concern. Energy optimization hints are concrete proposals for program code changes (e.g., source code patch) to achieve energy savings for a provided application. We group energy optimization hints into two categories: (1) source code changes and (2) build-environment modifications.

Source Code Changes. The energy demand of software can be improved by source code changes in two different ways. Either, the source code change improves the program code to use available power management features more efficiently, or, the program-code logic is changed. For the former, we have implemented PEEK to generate corresponding energy optimization hints (Section 3.1), for the latter we work on cross-domain optimization hints (Section 6) as part of our future work.

Build-Environment Modifications. Modifications to the build environment are independent of actual source code changes but also entail a potential to reduce the energy demand of program code (Section 3.2).

3.1 Power Management Features

Today, even smallest microcontrollers implement several different hardware power management features. Despite available documentation and specifications of the hardware, it is often unclear how different power management mechanisms (e.g., sleep states, dynamic voltage and frequency scaling, clock gating) interact with each other when executing a given program: it is extremely difficult to find the right set of power management features. For example, even tasks sharing the same execution semantics (i.e., run-to-completion or blocking) may require different sets of power management features to achieve the best energy savings. We address this challenge at back-end level by automatically generating different implementations of the source code under test (Section 4.1.2) and subsequently running an energy analysis for each of the implementations. The most energy-efficient implementation is eventually proposed as an energy optimization hint. Listing 1 shows an energy optimization hint, where PEEK suggests to run a task in a low-power run mode of the target platform. We quantify the potential for energy savings in Section 5.2.

3.2 Libraries and Compilers

Further energy savings can be achieved by modifying the build environment of the program code. On one hand, linker options can substitute standard libraries with hardware-specific ones in order to exploit special-purpose hardware units [23]. On the other hand, compiler infrastructures and specific compiler flags have also a significant impact on the energy consumption of the application [4, 6]. Changes reflecting such optimizations affect the build infrastructure of a snapshot (e.g., makefiles, GNU Autoconf, or Apache Ant) but do not alter the source code of the program itself. As shown by our evaluation, using a different compiler infrastructure can be a simple, yet effective measure to gain significant energy savings (Section 5.1).

```
--- task.c              2014-08-28 09:26:03 +0200
+++ task.c              2014-08-28 09:26:19 +0200
@@ -1,19 +1,22 @@
 void pm_set_mode(void) {
-  /* Normal-power run mode */
-  SMC->PMCTRL &= ~SMC_PMCTRL_RUNM_MASK;
-  MCG->C2 &= ~MCG_C2_LP_MASK;
+  /* Low-power run mode */
+  SMC->PMCTRL &= ~SMC_PMCTRL_RUNM_MASK;
+  MCG->SC &= ~MCG_SC_FCRDIV_MASK;
+  MCG->SC |= 0x02 << MCG_SC_FCRDIV_SHIFT;
+  MCG->C2 |= MCG_C2_IRCS_MASK;
   MCG->C1 &= ~MCG_C1_CLKS_MASK;
-  MCG->C1 |= MCG_C1_CLKS(0);
-  MCG->C1 |= MCG_C1_IREFS_MASK;
-  while( !(MCG->S & MCG_S_IREFST_MASK) );
-  while( ((MCG->S & 0xc) >> 0x2) != 0x0 );
+  MCG->C1 |= MCG_C1_CLKS(1);
+  while( !(MCG->S & MCG_S_IRCST_MASK) );
+  while( ((MCG->S & 0xc) >> 0x2) != 0x1 );
+  MCG->C2 |= MCG_C2_LP_MASK;
   SystemCoreClockUpdate();
-  run_mode = RUN_FAST;
+  run_mode = RUN_LP;
+  SMC->PMCTRL |= SMC_PMCTRL_RUNM(2);
 }

 void task(void) {
   /* Set power mode */
   pm_set_mode();
   /* Perform work */
   task_work();
 }
```

Listing 1: Implementation of an energy optimization hint which executes a task in a low-power run mode.

3.3 Integration

We have designed PEEK to support energy optimization hints belonging to both categories, source-code changes and build-environment modifications. To propagate energy optimization hints to the developer, PEEK provides the functionality to pass optimization proposals (source code changes representing an energy optimization hint)

from the back end to the front end. The back-end implementations use previously deposited, platform-specific information (e.g., power management features, special-purpose libraries, compiler flags) to transform existing snapshots into new, potentially more energy-efficient snapshots. For this purpose, we allow the back end to create snapshots and add them to a snapshot bundle.

To generate energy optimization hints, the back end initially takes a snapshot s_p proposed by the developer and analyzes the source code of s_p for optimizations. If the back end finds potential improvements, it correspondingly creates a new snapshot $s_{p'}$ which is a copy of s_p including the changes representing the optimization hints. Subsequently, the back end commits $s_{p'}$ and adds $s_{p'}$ to the original bundle of s_p. An energy analysis of the newly added snapshot $s_{p'}$ is performed subsequently. Once finished, the front end is notified about the availability of analysis results; the front end recognizes that a new snapshot has been added. Energy optimization hints are passed to the developer by showing differences of the source code of s_p and $s_{p'}$. Eventually, developers review the source code changes of the energy optimization hints and merge the desired changes into the original main development branch.

4 Implementation

We have implemented PEEK to demonstrate the potential of our approach to energy-aware programming. Our implementation combines a software stack, which we present in Section 4.1, with an energy measurement device we have designed to implement fully automated energy measurements, presented in Section 4.2.

4.1 Software

Our implementation of PEEK uses XML-RPC for interprocess communication of the front-end and back-end components, and YAML [24] to transfer meta data between the front end and the back end. Our middle-end infrastructure uses Git [18].

4.1.1 Core Implementation

The implementation of PEEK is aligned to the system design presented in Section 2. We focused on implementing a system that seamlessly integrates into existing development processes widely used by software developers. The front end is implemented as an extension to the integrated development environment (i.e., Eclipse). We decided to use the revision control system Git as middle end due to its wide adoption as well as its rich feature set and decentralized system structure. We further implemented two different back ends for PEEK. One back end provides energy analysis based on fully automated hardware energy measurements, the second back end implements energy analysis techniques using software components only. We discuss the implementation of the back ends separately in Section 4.1.2.

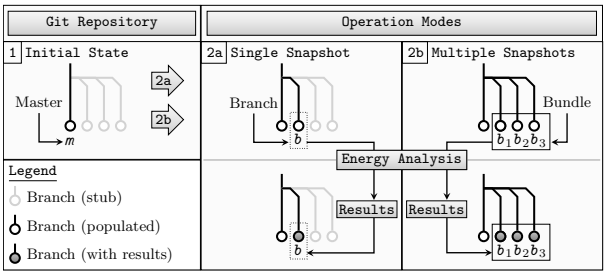

Figure 2: PEEK implements single- and multi-snapshot operation modes for energy analyses of source code.

Rather than altering existing infrastructure or changing programming habits of developers, we add the front-end and back-end components and jointly use Git as middle end without affecting established development workflows. With the chosen set of components, PEEK integrates seamlessly into familiar environments and can easily be adopted by software projects using Git for revision control purposes.

Single-Snapshot Operation. PEEK's snapshot mechanism is implemented using Git branches (see Figure 2). Our implementation of the front-end component creates a new Git branch whenever a developer submits a snapshot for analysis purposes. The newly created branch b is first pushed to the Git repository of the software project. Subsequently, the front end sends a control message $\texttt{analyze}\langle b \rangle$ to the back end in order to trigger the evaluation of the snapshot. The control message contains the name of the branch so that the back end pulls the correct snapshot for the energy analysis. The back end performs the energy analysis at function level and pushes the results to the branch that contains the current snapshot. Once done, the back end calls $\texttt{notify}\langle b \rangle$ to inform the front end of the availability of the results.

Multi-Snapshot Operation. If different revisions of source code are to be compared regarding their energy efficiency, developers submit several snapshots. Our implementation exploits Git's tagging features to implement snapshot bundles (Section 2.2). All Git branches of the snapshots which belong to the same snapshot bundle are tagged with a unique tag t. Once the tagged branches $\langle b_1, b_2, b_3, \ldots, b_n \rangle_t$ are pushed to the Git repository, the front end sends the control message $\texttt{analyze}\langle t \rangle$ to the back end. To fetch the entire snapshot bundle the back end then pulls all branches with the given tag t, analyzes each branch b_i and sends $\texttt{notify}\langle b_i \rangle$ for each finished energy analysis to inform the front end about the progress. Figure 2 shows an overview of the single- and multi-snapshot operations.

Energy Optimization Hints. If a back-end implementation finds potential optimizations for the submitted source code, it generates and applies the corresponding code changes and pushes new branches to the Git repository. The branches $\langle b_{n+1}, b_{n+2}, b_{n+3}, \ldots \rangle_t$ contain the changes proposed to gain energy savings. Before propagating the energy optimization hints to the developer, an energy analysis for the newly created branches is performed. Once the energy analysis results are available, the back end pushes them to the corresponding branches and the front end recognizes the new branches allocated to the snapshot bundle previously submitted for analysis.

4.1.2 Back-End Implementation

We have implemented two different back ends for PEEK: a software–hardware back end that takes fully automated hardware energy measurements and a software-only back end that performs energy analysis techniques using a software energy profiler. The implementations of the two back ends share a similar code structure and provide the same interfaces towards the front-end components of PEEK. All back-end implementations are hardware-platform specific and implement energy-analysis support for source code written in one or more programming languages.

Software–Hardware Back End. To implement fully automated hardware energy measurements, we designed and built an energy measurement board based on the concepts of a *current mirror* (Section 4.2.2). The latter is operated by an ARM Cortex-M4 microcontroller connected to a host running the PEEK back-end components. The device is designed to take energy measurements of arbitrary hardware architectures or platforms.

Our software–hardware back end supports energy analysis for the ARM Cortex-M0+ processor [16], which is the most energy-efficient ARM processor to date. To prepare the energy consumption analysis, the back end first extracts the functions which are requested to be analyzed from the meta data of the snapshot. The measurement is started at the entry of a function and stopped upon its exit. We insert signals for triggering the start and end of a measurement by instrumenting the source code. When a nested function is called, we optionally interrupt the measurement during the execution of the callee. The overhead of our code instrumentation is negligible as we only need a single CPU cycle for each trigger signal.

To implement energy optimization hints, the back end generates program code that executes the relevant functions of the source code with different sets of power management features. Power management features are target-platform specific and the back end leverages the snapshot bundles to pass energy optimization hints to the developer (i.e., modified source code with energy analysis results). For example, our evaluation platform implements an ARM Cortex-M0+ core with 11 different power saving modes. The best energy optimization hint reduced the energy consumption by 25.3 % (Section 5.2).

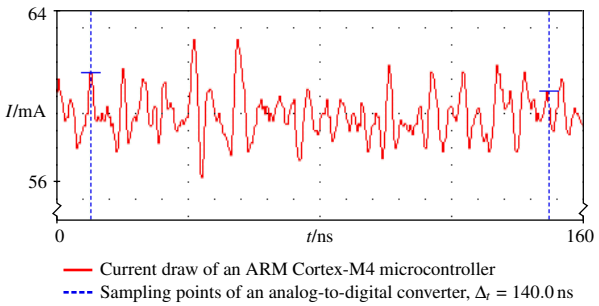

Figure 3: The undersampling of analog-to-digital converters leads to inaccurate energy measurement results.

Software-Only Back End. We have implemented a second back end for PEEK which uses software components only. For energy profiling purposes, this back end uses SEEP [14], which is an energy profiler providing multi-path energy analysis of source code. The energy profiler creates energy consumption estimates using energy profiles (i.e., instruction-based energy models) and is optimized for the TI MSP430 processor [25].

The back end first executes the code symbolically. During this symbolic execution run, the back end extracts valid input parameters for the different code paths of the application under test. Subsequent to this, binaries with concrete input data are created, which are used for further runtime analyses in order to calculate energy consumption estimates at function level. To cut down analysis times we extended the back end to distribute analysis tasks to multiple nodes. This addresses scalability concerns for complex energy analysis scenarios as discussed in Section 2.4. We show numbers on scalability aspects in our evaluation (Section 5.3).

4.2 Hardware

To run fully automated, accurate energy measurements we have designed and implemented a lightweight, yet powerful electronic measurement instrument for PEEK.

4.2.1 Measuring Energy Consumption

Commonly, the voltage drop across a resistor (shunt) is measured to determine the energy consumption of a device under test (DUT). The voltage drop is proportional to the current draw of the DUT as visualized in the graph of Figure 3. Multiplying the electrical current with the operating voltage of the device results in the power consumption. Integrating the power consumption over the measurement time eventually leads to the energy consumption of the DUT. However, the sampling

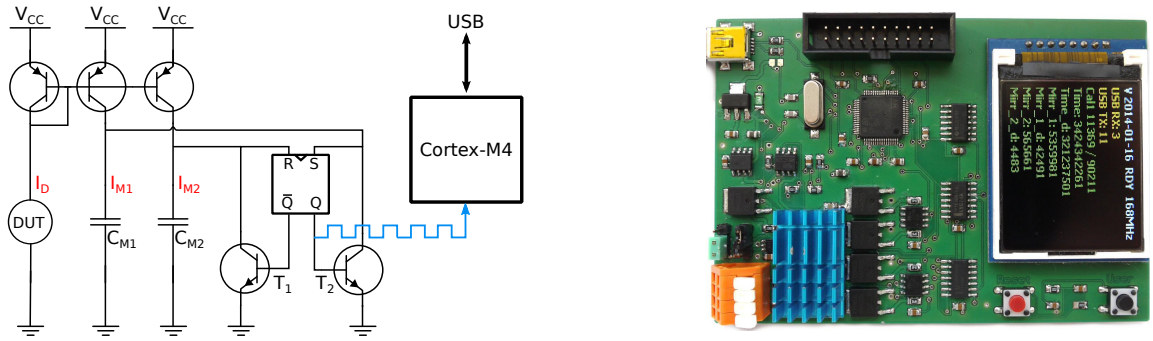

Figure 4: Our fully automated energy measurement device exploits a current mirror for analog energy measurements.

rate of analog-to-digital converters (ADCs) is often too low in order to provide accurate measurement results. Even microcontrollers implementing a more precise approach by chaining several analog-to-digital converters are prone to miss spikes in the current drawn by the DUT during measurement. We show this in Figure 3, where the sampling rate of an ADC employed by an STM32F407VG microcontroller [26] is insufficient in order to accurately capture the waveform of the electrical current. A high-performance digital storage oscilloscope with differential current probes can overcome these sampling constraints. However, setting up measurements with such a device is a tedious and error-prone process, making this approach infeasible for automatic measurements. Widely used outlet meters (e.g., Kill-A-Watt [27]) are unsuitable, as they commonly suffer from a sampling rate of 1 Hz (or worse), only provide a coarse-grained measurement resolution, and do not implement convenient data acquisition interfaces.

4.2.2 Energy Measurement Device

To implement fully automated, accurate energy measurements, PEEK requires a measurement device which does not suffer from sampling constraints, provides a high resolution, and offers a flexible, software-controllable data acquisition interface. As we were unable to find a suitable energy measurement device which satisfies the above requirements, we designed a new energy measurement instrument as part of our work on PEEK.

In our previous work on energy-aware programming [14, 28] we presented the first prototype for a non-discrete energy measurement device (i.e., an energy measurement device which does *not* sample). This early prototype implemented a simple, yet effective analog energy measurement circuit and was based on work by Konstantakos et al. [29]. With PEEK we present a revised energy measurement device with a larger set of features (e.g., microcontroller operation, automatic calibration), extended practicability (e.g., support for a wide range of currents) while keeping its unique characteristics untouched (i.e., analog energy measurements without suffering from sampling rate limitations). The new measurement device fulfills all requirements of PEEK and is used for fully automated energy measurements.

The core of the measurement device implements a transistor circuit exploiting the concepts of a current mirror, paired with a flip-flop to implement a *current-to-frequency* conversion. The transistor circuit consisting of three PNP transistors mirrors the input current I_{DUT} of the device under test (DUT) to the mirrored currents I_{M1} and I_{M2}. As long as the input current I_{DUT} is being drawn the current mirror circuit operates as follows: Under the control of an RS flip-flop, the two capacitors C_{M1} and C_{M2} are being charged and discharged alternately. When the RS flip-flop outputs a logical 1 on the output Q, the path I_{M1} is pulled to ground via the transistor T_2. The path I_{M2}, however, is allowed to charge up the capacitor C_{M2}. Once C_{M2} reaches a voltage level which the flip-flop recognizes as a logical 1, the flip-flop will toggle the output Q. Now, the path I_{M2} is shorted to ground, while path I_{M1} is charging up its capacitor C_{M1}. During each cycle, one capacitor charges, while the other one is being discharged. The switching frequency of the output Q is directly proportional to the current I_{DUT}. This signal is eventually used by our device to calculate the energy consumption of the DUT. The block diagram in Figure 4 shows the core board schematics.

4.2.3 System Integration

To implement fully automated energy measurements we integrated a microcontroller directly onto the circuit board. Figure 4 shows a photo of the PEEK electronic measurement instrument.[2] We use an ARM Cortex-M4 microcontroller (STM32F405RG [26]), which controls

[2]The total cost for the electronic measurement instrument (circuit board including all components) is approximately USD 80.

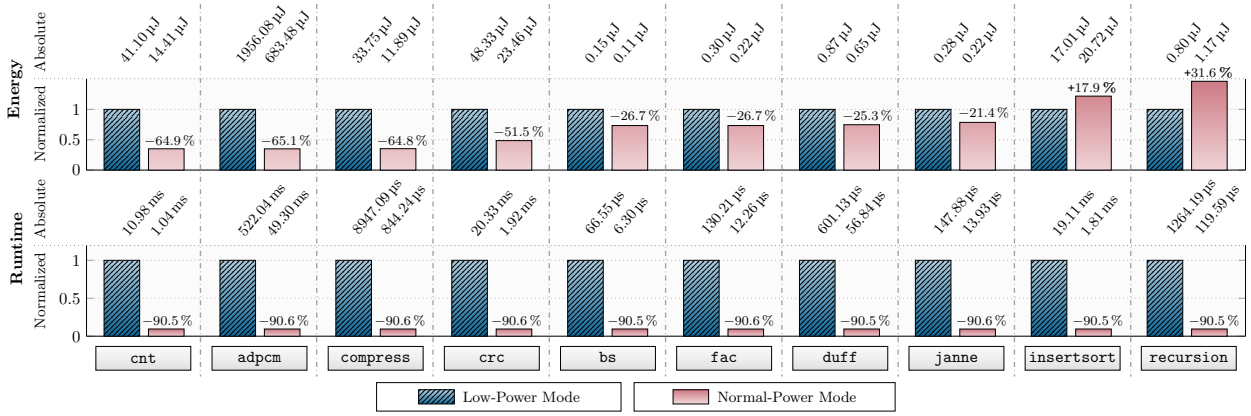

Figure 5: The energy consumption and runtime of benchmark modules running in different power modes.

the energy measurement and provides a host interface that we utilize for the PEEK back end presented in Section 4.1.2. Code instrumentation to start and stop energy measurements is minimally invasive: a single CPU cycle is required to trigger start or stop signals. The measurement device seamlessly fits into the PEEK infrastructure as it is easily adoptable while providing accurate energy measurement results. The device can also be used as a standalone energy measurement device.

5 Evaluation

In this section, we evaluate the software and hardware components of our PEEK implementation using four different scenarios. First, we perform an energy analysis for several benchmark modules running on our evaluation platform, an ARM Cortex-M0+ microcontroller. Second, we use PEEK to generate energy optimization hints for an application and measure the effect on the energy consumption of the code. Third, we measure time savings achieved by using PEEK compared to using a traditional energy profiling approach. Fourth, we present results of a hands-on experiment with students using our system.

5.1 Energy Measurements

We take fully automated energy measurements using PEEK to analyze the energy consumption of the Mälardalen benchmark suite [30], which provides a variety of modules implementing common usage scenarios. To evaluate all aspects of PEEK we choose a microcontroller with a wide array of energy saving features. Our evaluation platform, an ARM Cortex-M0+ microcontroller (Freescale Kinetis KL02 [22]), offers 11 different power modes: two run modes and nine sleep modes. We first execute all modules of the benchmark suite in the two run modes (i.e., low power and normal power) of the evaluation platform to demonstrate their effect (see Figure 5). In a second experiment PEEK compiles the benchmark modules with different compilers (i.e., LLVM Clang 3.4, GNU GCC 4.8) but same optimization level (i.e., `-Os`) and measures the energy consumption of the compiled binaries running in normal-power mode on the evaluation platform (see Figure 6).

The energy E_{run} required to execute a benchmark module is calculated by integrating the time function of the power consumption $p(t)$ over the time $t_{run} = t_1 - t_0$ required to execute the benchmark module:

$$E_{\text{run}} = \int_{t_0}^{t_1} p(t) \cdot \mathrm{d}t$$

Power-Mode Energy Impact. We execute the benchmark modules in low-power mode and normal-power mode to measure the impact of power modes on the energy consumption of the evaluation platform. Figure 5 shows the results of the energy analysis of PEEK.

All benchmark modules have run-to-completion semantics as they are non-blocking applications. Accordingly, it depends on the power consumption $p(t)$ over the execution time t_{run} whether it requires more energy to run a benchmark module in a normal-power mode or in a low-power mode. Applications with run-to-completion semantics commonly require less energy when they are executed in a *normal-power* mode. This is because the reduced performance of a *low-power* mode results in longer execution times, eventually leading to increased energy consumption. Executing the application in a normal-power mode (i.e., higher power consumption but significantly shorter execution time) and entering a sleep state afterwards therefore consumes less energy. This is why *race-to-sleep* strategies [31, 32] are commonly used, in which run-to-completion tasks are executed in a normal-power mode before entering a sleep state.

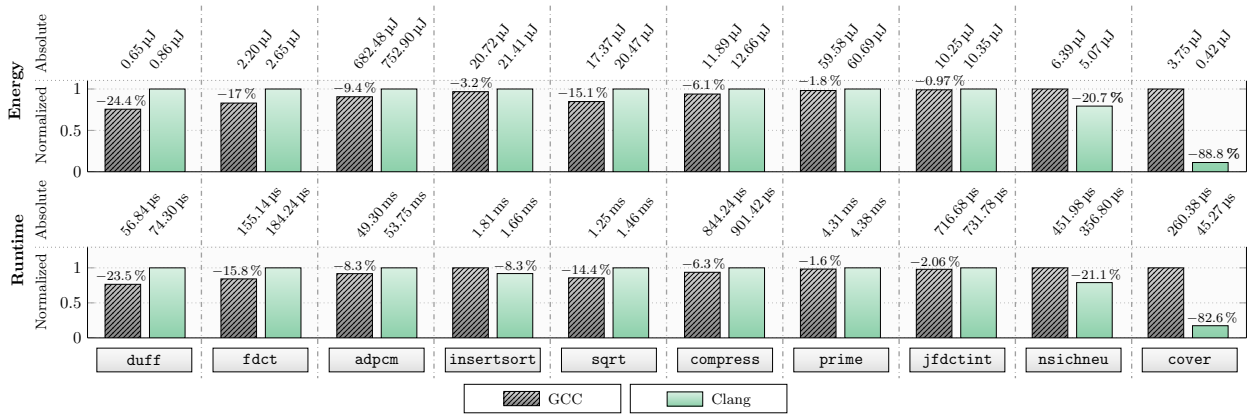

Figure 6: The energy consumption of benchmark modules differs depending on the compiler by up to 88.8 %.

Our evaluation results show that on average, executing the benchmark modules in low-power mode takes about ten times longer compared to normal-power mode. However, the results reveal that the energy consumption is *not* proportional to the runtime required to execute each module. Two of the benchmark modules (i.e., `insertsort` and `recursion`) even consume greater energy in normal-power mode. The module `recursion` requires 31.6 % more energy in normal-power mode ($E_{\text{run}} = 1.17\,\mu\text{J}$, $t_{run} = 119.6\,\mu\text{s}$) compared to low-power mode ($E_{\text{run}} = 0.80\,\mu\text{J}$, $t_{run} = 1264.2\,\mu\text{s}$). We reproduced the results of the experiment on a second evaluation platform to rule out a hardware defect. As the assemblies of the benchmark modules and the evaluation setup are exactly the same for the runs in normal-power mode and low-power mode, we conclude the internal architecture of the microcontroller to be the cause for these results of our experiment.

On the one hand, the results of our experiment reveal that energy consumption most often is *not* proportional to runtime, and race-to-sleep strategies can lead to energy penalties of up to 31.6 %. On the other hand, the experiment highlights the importance of real energy measurements, as instruction-based energy models (as used by software-based energy profilers) would have been unable to lead to these results.

Compiler Energy Impact. In a second experiment, we use PEEK to analyze energy consumption of the benchmark modules using different compilers (LLVM Clang 3.4, GNU GCC 4.8). The experiment executes the benchmark modules in the normal-power mode of the evaluation platform. Clang and GCC generate program code with different energy footprints, however, no general trend can be derived from the results (Figure 6): Some modules consume less energy when they are compiled with Clang, other modules consume less energy when they are compiled with GCC. As both compilers share the same call parameters (i.e., compiler flags, command line options) it is straightforward to choose the better compiler on a case-by-case basis. Accordingly, PEEK automatically generates the corresponding energy optimization hints for each of the benchmark modules. The benchmark module `cover` requires significantly less energy if it is compiled by Clang ($E_{\text{run}} = 0.42\,\mu\text{J}$) compared to GCC ($E_{\text{run}} = 3.75\,\mu\text{J}$). We analyzed the assemblies and found the root cause to be inter-procedural compiler optimizations specific to Clang.

The results of this experiment show that it is important to analyze source code on a case-by-case basis in order to pick the right set of building infrastructure. As shown by our experiment the energy consumption can be reduced by up to 88.8 %, just by choosing a different compiler.

5.2 Energy Optimization Hints

To further evaluate the generation of energy optimization hints (Section 3) we use PEEK to analyze an application consisting of two tasks. The first task T_{sample} reads sensor data from a triple-axis accelerometer and performs pre-calculations, whereas the second task $T_{transfer}$ implements an encrypted data transmission (AES) of the calculated data using a wireless radio. We use the open-source operating system Contiki [33] to execute the application on our evaluation platform.

We first create a snapshot of the source code (i.e., application source code, Contiki) using the front-end components. At back-end level, PEEK uses previously deposited information on the target platform (i.e., power modes) to automatically inject the use of different power modes for both tasks of the application. With this information, PEEK creates four new source code revisions: Two of the revisions keep both tasks of the application running in the same power mode (e.g., normal-power mode, low-power mode). For the third and fourth revision, PEEK alternates the use of power modes by injecting the corresponding code to switch power modes at runtime. For all four revisions, PEEK subsequently runs

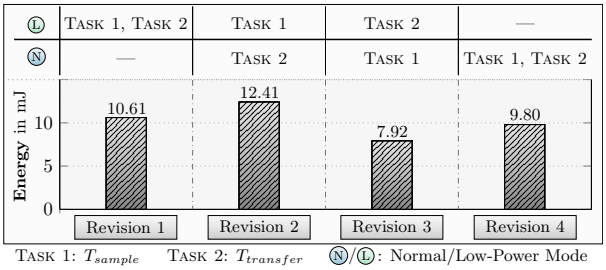

Figure 7: Energy optimization hints generate new revisions of source code with varying energy footprints.

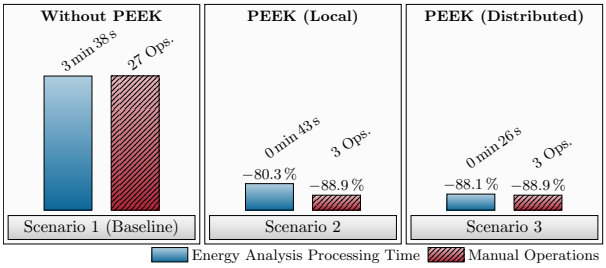

Figure 8: Energy analyses with PEEK reduce the number of manual operations and cut down analysis times.

an energy analysis to determine the energy footprint for each of them and stores the resulting data (i.e., energy consumption results) jointly with the new source code revisions in the middle-end infrastructure and allocates new snapshots to the original snapshot bundle. Eventually, the front end recognizes the new snapshots as energy optimization hints and announces their availability.

The most effective energy optimization hint leads to the third source code revision (see Figure 7). In this revision, task T_{sample} executes in low-power mode and task $T_{transfer}$ executes in normal-power mode. The energy costs for switching power modes are 3.0 µJ (normal to low power) and 0.7 µJ (low to normal power). In the most energy-efficient revision, the entire application requires 7.92 mJ to execute. This is an improvement of 25.3 % compared to the energy consumption of the original source code. In the original source code (represented by the first revision) the application consumes 10.61 mJ.

5.3 Energy Analysis Time Savings

We conduct an experiment to measure the time savings achieved by using PEEK compared to using a traditional energy profiler (i.e., SEEP [14]), which is the baseline.

During our experiment we perform batch energy analyses of five source code snapshots using different sets of tools. In the first scenario (baseline) we use a development environment with a plain energy profiler. For the second scenario, we change the set of tools by using PEEK for energy profiling. In the third scenario, we leave the set of tools unchanged compared to scenario two but we move the back end of PEEK away from the local developing system to a dedicated node on the same network with more processing power. The objective is to use the different sets of tools available in each scenario to identify the snapshot with the smallest energy footprint. This objective is kept the same for all three scenarios.

Figure 8 shows the total time required to identify the snapshot with the smallest energy footprint and the number of manual operations (e.g., task activations, file operations) performed during the analysis. Our experiment takes at least 5.1 times as long without extended tooling support, compared to scenarios where the energy analysis is performed by PEEK. We even achieve a 8.4-fold speed-up in the third scenario. At the same time, PEEK reduces the amount of manual operations to a ninth. The results for the third scenario show that additional hardware resources help PEEK to further decrease the time required for energy analysis considerably (43 s vs. 26 s).

5.4 Hands-On Experiment

To evaluate the experience of working with our tooling infrastructure, we conducted a hands-on experiment. We asked two students of a systems programming class to work on different AES implementations running on a mobile platform. The assignment for the participants was to initially create two different implementations of AES and to optimize the implementations for energy consumption. The students were asked to achieve this by comparing the differences in energy behavior of the two versions, and to improve the implementations for lower energy consumption guided by our tooling support. The participants further took energy measurements using a prototype of our measuring device to determine the energy consumption at hardware level. Subsequent to the hands-on experiment, the participants were asked to report on their experiences during an interview.

The students preferred using our software tooling infrastructure over using a hardware measuring device as it greatly shortened the analysis process. During their work, the participants repeated the analysis process many times over, exceeding 30 times per day, which highlights the relevance of the time saving achieved through PEEK (Section 5.3). To improve the tooling infrastructure, the participants asked for additional indicators to increase the understanding of the observed energy behavior of their code (e.g., further energy optimization hints).

The hands-on experiment confirms that the speed-up of the analysis process through PEEK is critical to developers and that support for energy analysis is a great assistance to developers in order to make meaningful decisions while programming energy-efficient code.

6 Future Work

Developers rely on exhaustive tool support for building software. In contrast to established tools (debuggers, profilers for runtime optimizations), future software development environments will increasingly focus on raising the degree of automatization to cope with complexity of today's software projects—PEEK is one example for such extensions to existing developing infrastructures.

Tooling Integration. An important part of our future work on PEEK is the efficient combination of different software development tools. Today, unit tests are a well-established approach to reveal software defects automatically. Apart from that, our system already exploits existing unit tests and reuses them for input data (Section 2.3). To leverage resources spent on runtime analysis much better, it is important to join different approaches: One and the same run of a unit test should jointly verify functional requirements (i.e., correctness) *and* non-functional requirements (i.e., energy consumption).

Cross-Domain Optimization Hints. Currently, PEEK generates energy optimization hints which are solely focusing on optimizing the energy consumption of program code. However, a lower energy footprint may entail an impact on resources of other system domains (e.g., memory consumption, I/O). We consider it crucial to detect and quantify such side effects as developers may require assistance in their decision making process. We will implement corresponding cross-domain optimization hints to support developers.

Measurement Enhancements. We will further improve our hardware measurement device presented in this work (Section 4.2.2) to implement fine-grained energy measurements at hardware level, which allow the attribution of energy consumption to different hardware components (e.g., memory, network, radio). Further assistance in optimizing energy consumption may be provided by an additional energy tracing functionality. This would assist the developer with automatic energy measurements on a function level during runtime.

We will release the measurement device (schematics, tools) under an open-source license on the project page `https://www4.cs.fau.de/Research/PEEK`.

7 Related Work

Operating system research on optimizing systems (of arbitrary size) for low-energy footprints was initiated by early work on energy-aware runtime aspects (i.e., process scheduling) by Weiser et al. [1] and energy-proportional computing by Barroso et al. [2].

Compiler optimizations [3, 4, 5, 6] are an important factor as they have a significant impact on the energy consumption of program code. However, architecture-level optimizations (i.e., instruction set) do not address runtime power management features at platform level (i.e., sleep states [34], dynamic voltage and frequency scaling [35]). In addition to this, our evaluation of PEEK (Section 5) shows that it pays to analyze case-by-case which compiler generates the most energy-efficient assembly for a given program code.

Energy profilers [12, 13, 15, 36] have a long tradition in the operating systems research community. Commonly, energy profilers use indirect measures (e.g., performance counters [37]) to estimate the energy consumption of program code. This includes our own previous work on exploiting symbolic execution for energy-aware programming [14, 28].

PEEK integrates well with the above techniques from the related domains of energy-aware compilers and energy profilers. Furthermore, the use of energy-saving features at system level can be optimized by PEEK either through measurements or—if specifically supported—in form of optimization hints.

8 Conclusion

This paper presents PEEK, a systems approach to proactive energy-aware programming. PEEK implements fully automated energy measurement techniques which provide developers with energy consumption measurements at function level. Our approach embraces components of existing software development infrastructures and complements established development processes.

Automatically generated energy optimization hints of PEEK assist developers at resolving energy faults early during the software development process. Our software infrastructure is supplemented by a lightweight, yet powerful microcontroller-operated measurement device which takes analog energy measurements.

PEEK achieves an 8.4-fold speed-up of energy analysis while improving the energy consumption of the analyzed source code by up to 25.3 %. The evaluation results show that a combination of mature tooling infrastructure and profound energy measurement setup is inevitable to enable developers to optimize their code proactively, right at development time.

Acknowledgments

We thank Tobias Distler, Thao-Nguyen Do, Peter Wägemann, Laura Lawniczak, Johannes Schilling, Ying Qu, Björn Cassens, the anonymous reviewers, and our shepherd, Doug Terry, for their insightful comments and feedback. This work was supported by the German Research Foundation (DFG), in part by Research Unit FOR 1508 under grant no. KA 3171/3-1 and SCHR 603/11-1 and Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89, Project C1).

References

[1] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st Conference on Operating Systems Design and Implementation*, pp. 449–471, 1994.

[2] L. A. Barroso and U. Hölzle. The case for energy–proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[3] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference*, pp. 304–307, 2000.

[4] M. Valluri and L. K. John. Is compiling for performance == compiling for power? In *Interaction between Compilers and Computer Architectures*, pp. 101–115. Springer, 2001.

[5] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the 2003 Conference on Programming Language Design and Implementation*, pp. 38–48, 2003.

[6] J. Pallister, S. Hollis, and J. Bennett. Identifying compiler options to minimise energy consumption for embedded platforms. *The Computer Journal*, pp. 1–15, 2013.

[7] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, pp. 161–174, 2007.

[8] 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 *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pp. 164–174, 2011.

[9] 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*, 2(4):437–445, 1994.

[10] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th Conference on Operating Systems Design and Implementation*, pp. 323–338, 2008.

[11] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proceedings of the 6th European Conference on Computer Systems*, pp. 139–152, 2011.

[12] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd Workshop on Mobile Computing Systems and Applications*, pp. 2–10, 1999.

[13] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *Performance Evaluation Review*, 36(2):26–31, 2008.

[14] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat. SEEP: Exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, pp. 17–22, 2011.

[15] S. Wang, Y. Li, W. Shi, L. Fan, and A. Agrawal. Safari: Function-level power analysis using automatic instrumentation. In *Proceedings of the 3rd International Conference on Energy Aware Computing*, pp. 1–6, 2012.

[16] ARM Incorporated. ARM Cortex-M0+ Technical Reference Manual, Revision r0p1, 2012.

[17] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[18] J. Hamano and L. Torvalds. Git Revision Control System. `http://www.git-scm.com`.

[19] M. Mackall. Mercurial Revision Control System. `http://mercurial.selenic.com`.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[21] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference*, pp. 263–272, 2005.

[22] Freescale Semiconductor. KL02 Sub-Family Reference Manual, Revision 3.1, 2013.

[23] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh. Energy-aware high performance computing with graphic processing units. In *Proceedings of the 1st Workshop on Power-Aware Computing and Systems*, pp. 1–5, 2008.

[24] C. C. Evans. YAML Ain't Markup Language. http://www.yaml.org/.

[25] Texas Instruments Incorporated. MSP430FR57xx Family User's Guide, Revision c, 2013.

[26] STMicroelectronics Incorporated. STM32F405xx and STM32F407xx Family Datasheet, Revision 4. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf.

[27] P3 International Corporation. Kill-A-Watt P4400. http://www.p3international.com/.

[28] T. Hönig, R. Kapitza, and W. Schröder-Preikschat. ProSEEP: A proactive approach to energy-aware programming. In *Proceedings of the 2012 USENIX Annual Technical Conference (Poster)*, 2012.

[29] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on Instrumentation & Measurement*, 57(4),797–804, 2008.

[30] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pp. 136–146, 2010.

[31] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pp. 121–130, 2002.

[32] S. Dawson-Haggerty, A. Krioukov, and D. E. Culler. Power optimization: A reality check. Technical report, EECS Department, University of California, Berkeley, October 2009.

[33] A. Dunkels, B. Gronvall, and T. Voigt. Contiki: A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th International Conference on Local Computer Networks*, pp. 455–462, 2004.

[34] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. H. Katz. NapSAC: Design and implementation of a power-proportional Web cluster. In *Proceedings of the 1st Workshop on Green Networking*, pp. 15–22, 2010.

[35] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras. Fix the code. Don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization*, pp. 262–272, 2014.

[36] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for Android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pp. 69–76, 2012.

[37] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 238–246, 2002.