Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

From Sensibility to Responsibility: The Impact of System Software on Ecological and Economical Sustainability of Computing Systems

Timo Hönig [1]

Abstract: From an ecological and economic perspective, saving energy is an important factor for the design of today's computing systems. However, building energy-efficient systems is a challenging task for system designers. It requires exact knowledge about the system's components (i.e., software and hardware components) and it is not immediately obvious *how* the energy demand of a system can be improved. This paper presents energy-aware programming techniques that help developers at optimizing the energy demand of software. Such techniques address *sensibility* aspects (i.e., energy awareness) and provide optimization suggestions to developers. The paper further discusses *responsibility* aspects that emerge from the advent of energy-aware programming techniques.

Keywords: Energy-Aware Programming, Energy-Aware Systems, System-Software Design, Sustainability of Computing Systems.

1 Introduction

Energy has become the most precious and critical resource of computing systems for all types of computing systems: from smallest-sized embedded systems up to large-scale data-center systems. For different kinds of computing systems, energy resources need to be handled with caution for different reasons. Embedded systems (e.g., smart dust [KKP99]) suffer from limited energy resources. It is necessary to address restricted energy resources, which arise from limited energy-storage capacities (e.g., supercapacitors, batteries), and the volatile availability of energy [Ka07]. Thus, embedded systems need to conserve available energy resources in the most efficient manner in order to avoid operational disruptions and to proactively prevent system failures caused by depletion of energy resources. Large-scale data-center systems (i.e., high-performance computing systems, cloud-computing systems) enjoy the practically unlimited availability of energy resources. However, large-scale systems commonly suffer from excessive consumption of energy (e.g., caused by operation at maximum performance) and entailing unwanted side effects (e.g., thermal overheating [Mo05], operating costs). Therefore, future large-scale and many-core systems [Es11] must be designed according to their energy characteristics and need to cautiously schedule activities to avoid system failures caused by excessive energy demand.

The effective energy-efficiency of any computing system depends on the characteristics of its hardware *and* software components. Hardware components with low-energy characteristics are optimized at CMOS level [Ja12] which reduce the energy demand of the

[1] Friedrich–Alexander–Universität Erlangen–Nürnberg (FAU), Department of Computer Science, Distributed Systems and Operating Systems, Martensstr. 1, D-91058 Erlangen

transistor circuit, for example, by implementing energy-saving modes and semiconductor scaling. However, hardware components only provide the basis for the energy-efficient operation of computing systems. It is necessary for the system software to operate in an energy-aware manner on the hardware, for example, by avoiding unnecessary system activities and exploiting hardware energy-saving features. Unfortunately, it is a difficult task for system-software designers to establish energy awareness of software components. This circumstance is caused by missing tool support, as programmers often lack energy-analysis tools that would help to identify energy hotspots of program code.

To reduce the energy demand of a computing system it is necessary to optimize the system and application software running on the system. First, the software itself needs to be optimized statically *before* run time (i.e., at the time of program-code creation) at different levels of abstraction (e.g., high-level language, assembly). Second, the software needs to be optimized dynamically *at* run time (i.e., at the time of program-code execution).

Profound tool support is an essential measure to help programmers at designing energy-aware computing systems, which exploit energy resources in the most efficient manner. For one, tool support actively assists programmers at designing energy-aware system software by integrating automated code analysis techniques. For another, tool support bridges the gap between developers and the operating system: important knowledge about the intent of the program code is transferred from the program design and programming phase (before run time) to the program execution phase (at run time) of the software.

The paper is divided into two parts. The first part of the paper (Sections 2 and 3) presents challenges in energy-aware system design and proposes concepts to address these challenges. This includes energy-aware programming techniques that establish the *sensibility* of energy demand. The second part of the paper (Section 4) analyzes the resulting impact on the *responsibility* of programmers with regards to the ecological and economical sustainability of system and application software. In more detail, the paper is structured as follows. Section 2 introduces energy-aware computing systems, presents related work, and discusses distinct energy-optimization methods. Energy-aware programming concepts and proactive system design techniques are presented in Section 3. The paper further discusses the consequences of establishing concepts for proactive energy-aware system design in Section 4. Section 5 outlines how the presented concepts are embedded into the author's doctoral thesis. Section 6 summarizes the proposed approaches and concludes the work.

2 Energy-Aware Systems

Energy-aware systems are computing systems which are conscious regarding the energy impact of individual operations. This energy awareness allows systems to economize available energy resources. The motivation for economizing energy varies and depends on the type of computing system: energy savings lead to extended battery life (i.e., embedded systems), reduced operating costs (i.e., desktop and server systems), and lower energy bills (i.e., large-scale systems).

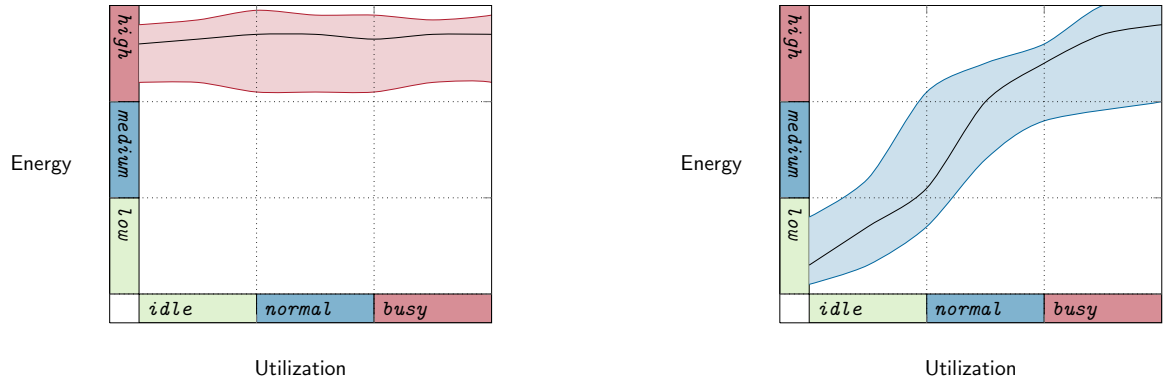

Fig. 1: Schematic representation of energy demand as a function of utilization: energy demand of an energy-unproportional system (left graph) vs. energy-proportional system (right graph).

2.1 Background

The implementation of an energy-aware system requires two main components. First, the system hardware needs to be energy proportional. Energy proportionality is achieved by implementing energy-saving features at hardware level which allow the system software (i.e., operating system) to influence the behavior of hardware components (e.g., activation and deactivation of functional features, switching of operating modes). Second, the system software needs to implement strategies to make efficient use of available energy-saving features at hardware level.

The hardware components implement low-power modes and sleep states at CMOS level and need to expose corresponding control interfaces that are used by the system software (i.e., operating system). Early computer systems did not implement any of such energy-saving features at hardware level, which ruled out software measures to adjust the energy demand depending on the systems' utilization. Today's systems, however, allow fine-grained control of the energy-saving features implemented by the hardware. Figure 1 shows the energy demand as a function of utilization for energy-unproportional systems (left graph) and energy-proportional systems (right graph).

From a system-software point of view, energy demand of computing systems can be decreased in two different ways. First, the program code of the system software can be optimized to exploit energy-saving features at hardware level in the most efficient manner. This includes device-specific low-power states which largely effect the static energy demand of the system. For example, this includes switching off underutilized hardware components such as wireless controllers. Second, software itself can be optimized to implement a required functionality in the most energy-efficient manner. This is achieved by structural optimizations of program code and the utilization of functional units at hardware level (e.g., floating-point units, graphics processing units).

2.2 Related Work

At system level, systems reduce their energy demand by exploiting device-specific energy-saving functions, which are implemented at hardware level by the individual system components (e.g., CPUs, wireless controllers [Ha10]). Today, energy-saving functions like

multi-level low-power modes and sleep states complement traditional approaches such as dynamic voltage and frequency scaling [CSP05]. On top of energy-saving primitives at hardware level, system software components commonly implement different strategies to increase the gain of energy savings. Traditionally, race-to-sleep strategies [DHKC09] have been proven to be effective measures to save energy. However, recent research [Hö14] shows that crawl-to-sleep strategies can be more energy-efficient.

At application level, software needs to be optimized statically (before run time) and dynamically (at run time). Recent proposals for energy-aware programming languages [Sa11] reduce the energy demand of computational resources required to execute the software by applying approximation techniques. Compiler optimizations [PHB15] at architecture level can not address energy-saving features at platform level (e.g., sleep states).

2.3 Static Energy Optimizations of Software

To implement *static optimizations* for reducing the energy demand of software before run time, developers need to optimize the program code at the time of program-code creation. Static optimization steps concern different levels of abstraction: First, program code needs to be altered in order to efficiently exploit available hardware functions (e.g., floating-point units, graphics processing units) and explicitly using available energy-saving functions (e.g., low-power modes, sleep modes) of the target platform. To apply static optimizations at level of the programming language, developers need to analyze the target hardware platform and adapt their program code according to the hardware characteristics. Second, different code transformations at lower language levels (e.g., intermediate language, machine language) than the originating high-level programming language apply code optimizations which influence the energy demand of the program code.

2.4 Dynamic Energy Optimizations of Software

To implement *dynamic optimizations* for reducing the energy demand of software at run time, the operating system must coordinate potentially disaccording interests (i.e., conflicting policies at task and system level) and the operating system continuously needs to execute the corresponding decision making process. However, the basis of this decision making process often lacks important information about the intended behavioral aspects of program code which currently executes on the computing system.

2.5 Problem Statement

The different energy-saving opportunities and the various static and dynamic approaches to optimize software for energy demand make it difficult for programmers to find the right set of measures to build energy-aware systems. Depending on hardware and software characteristics, programmers need to find the correct settings in a case-by-case decision process. This process is especially complicated as sufficient tool support often is missing. During the task of programming, developers commonly are left without orientation, as they have no feedback on the energy-demand of their programming decisions.

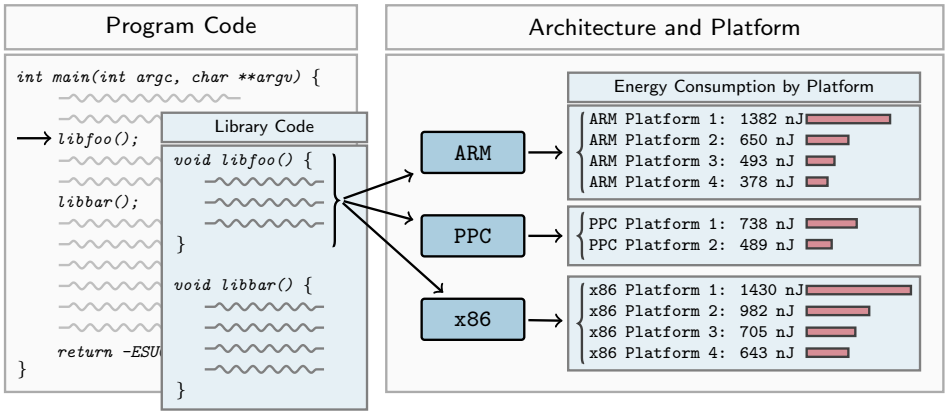

Fig. 2: Energy consumption of program code at function level: the individual energy consumption for executing a function depends on the hardware architecture and the hardware platform.

3 Energy-Aware Programming

Energy-aware programming techniques unify different concepts and increase the energy awareness of the developer during program-code creation. To initially create a basis of decision-making for the developer, energy-aware programming techniques provide feedback on the energy demand of program code. The techniques include code analyses that provide direct feedback on the energy demand of program code. As the energy demand differs for heterogeneous hardware architectures and platforms, analysis techniques must address this diversity, for example, by using virtualization technologies. The energy-function mapping in Figure 2 shows that energy demand for executing the program code of function `libfoo` differs on heterogeneous hardware architectures and platforms.

Energy demand of program code can be evaluated at different levels of abstraction. Programmers are most commonly interested in high-level energy analysis (i.e., energy analyses at function-level), as this allows them to reason about the effectiveness of code restructuring measures with regards to the energy efficiency. Depending on the analysis method, it requires often several intermediate steps to obtain the end result: energy-analysis methods evaluate the energy demand of the program code at other levels of abstraction (i.e., assembly level, basic-block level) and consolidate the results in a bottom-up approach.

As a consequence of energy-aware programming, programmers reveal *energy faults*[2] at design time and programmers can verify whether corresponding countermeasures (e.g., restructuring of program code) lead to the desired effect (i.e., reduction of energy demand). To implement a reliable chain of tool support for establishing energy-aware programming it is required to provide developers with energy consumption data of program code (Section 3.1), to assist at analyzing different code paths (Section 3.2), and to implement automation aspects to address the complexity of analyses (Section 3.3). The resulting infrastructure allows the implementation of analysis techniques, which automatically extract actual optimization suggestions for developers (Section 3.4).

[2] 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).

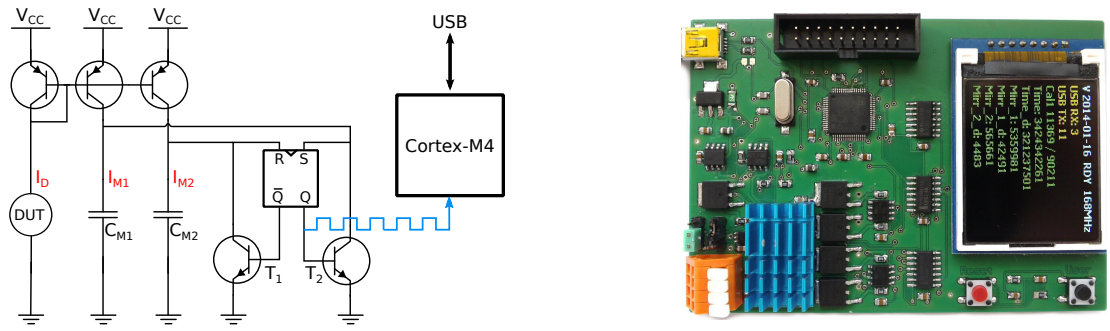

Fig. 3: The PEEK energy measurement device implements automated energy measurements and performs a current-to-frequency conversion for non-discrete measurements of energy demand.

3.1 Sensing Energy Demand

Energy-aware programming concepts rely on methods to sense and to quantify the energy demand of program code. This is either done by using energy-modeling [WB02, Hä12] or energy-measuring methods [TMW94, Fo08]. Model-based energy analyses are implemented by software components only. Therefore, programmers do not need to instrument their setup with additional hardware components (i.e., measurement equipment). As hardware complexity is steadily increasing, the creation of reliable and accurate energy models is a difficult task. For example, non-deterministic effects (i.e., cache misses, pipeline stalls) need to be considered by energy models. Results from model-based energy-analyses need to be verified periodically by control measurements to retain accuracy criteria. Measurement-based energy analyses use hardware instruments (e.g., multimeter, oscilloscope) to determine the energy demand of software during execution. In contrast to model-based energy analyses, measurement-based energy analyses are integrated into the setup of the system which is analyzed (device under test, DUT). As energy measurements follow a black-box approach, non-deterministic effects are included implicitly.

PEEK [Hö14] implements a non-discrete energy measurement device (i.e., an energy measurement device which does *not* sample) to avoid sampling limitations. The measurement device features microcontroller operation via USB, automatic calibration functions, and is capable of running automated energy measurements. The device implements a transistor circuit consisting of a current mirror and a flip-flop to implement a current-to-frequency conversion. Three PNP transistors mirror the input current I_{DUT} of the device under test to the mirrored currents I_{M1} and I_{M2}. The circuit operates as follows while the input current I_{DUT} is being drawn: controlled by an RS flip-flop, 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 toggles its 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}. The device uses the signal to determine the energy consumption of the DUT. Figure 3 shows an excerpt of the board schematics and the PEEK measurement device.

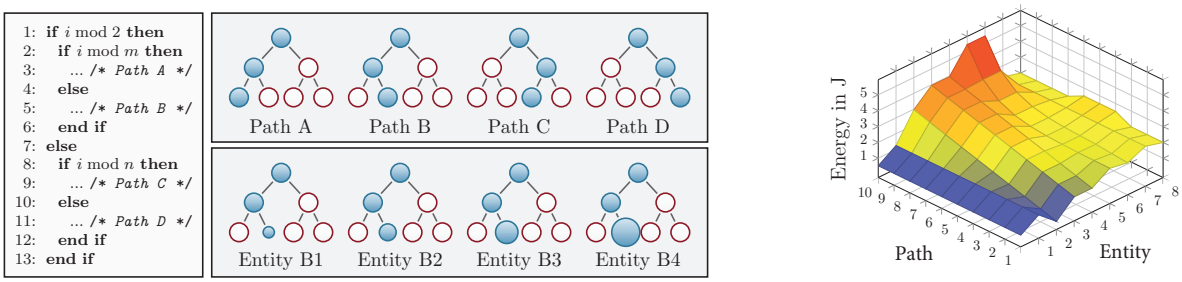

Fig. 4: The energy demand of program code at run time differs for various code paths and depends on the actual input data of the program (e.g., input parameters, processing data).

3.2 Combining Static and Dynamic Code Analysis

It is necessary to extract run-time characteristics to determine the energy demand of program code. Different program paths of an application may have entirely different energy demands. Furthermore, different program-path entities (i.e., identical program paths executed with different input data) again show distinct energy requirements. As an example, Figure 4 shows the excerpt of a program code that consists of four code paths. The value of the input parameter `i` determines which path is being executed at run time as visualized by the tree structure of the control flow graph. Moreover, the variable `i` influences the further proceeding of the code path, which results in path entities exposing distinct energy demands. The energy map in Figure 4 visualizes this.

To reason about the energy demand of individual program-code entities requires the use of static or dynamic code-analysis techniques. Static code-analysis techniques analyze the syntactical structure of code without actually executing it, whereas dynamic code analysis-techniques extract run-time information during the execution of the program code. Symbolic execution [Ki76, CDE08, Ca11] is an analysis technique that combines static and dynamic code-analysis techniques. Results from the code analysis are followed by a transformation step where energy demands are calculated on top of the analysis results.

The energy-aware programming technique SEEP [Hö12] uses symbolic execution techniques to automatically reveal possible program paths and to extract individual path constraints. Based on these path constraints, SEEP builds and executes binaries of the corresponding path entities. During this execution step SEEP extracts relevant run-time information (e.g., number and type of executed instructions) to calculate the energy demand of the individual path entities. Developers use such tool support to analyze whether specific programming decisions (e.g., restructuring of program code, using different program libraries) have a positive (or negative) impact on the energy demand.

3.3 Automated Analysis

Carrying out energy analysis for program code entails time-consuming operations. The amount of analysis work increases with the complexity of program code (i.e., growing number of code paths) and with the amount of potential target platforms that are considered by the developer. For this reason, energy-analysis tools need to be automated and run

unattended (i.e., without requiring manual interaction). Further, scalability aspects have to be considered and addressed at level of the tooling infrastructure of energy-aware programming techniques. For example, code-analysis methods use multiple threads or distributed computing systems to reduce the required analysis time.

3.4 Automated Energy Optimization

Energy-aware programming techniques which provide the infrastructure to sense energy demand and implement the execution of automated energy analyses are the basis for further extensions, for example, the automatic generation of energy-optimized program code.

Developers require hints and suggestions on *how* to actually improve their program code. A corresponding technique [Hö14] implements the generation of *energy optimization hints*. Energy optimization hints are being created during the code-analysis phase of energy-aware programming. If potential improvements for the program code of the developer are found, the infrastructure proposes a structural change to the original source code (e.g., as source-code patch). This step significantly improves the analysis process as the developer only needs to verify whether the suggested improvement introduces functional regressions.

FIGAROS [Hö15] improves energy-aware programming techniques by transferring knowledge from the developer to the run-time environment (i.e., operating system). The operating system kernel of FIGAROS is energy-aware and senses different system activities for energy analysis at run time. Future research work will integrate FIGAROS with the energy-aware programming kit PEEK [Hö14].

4 From Sensibility to Responsibility

The advent of energy-aware programming techniques improves the development of energy-efficient system software significantly. However, for developers the introduced *sensibility* of energy demand of program code implies a new *responsibility* as a direct consequence. Hence, this section discusses societal aspects tied to—and resulting from—the previously presented technical advances (i.e., energy awareness at programming level).

4.1 Ecological and Economic Impact

On the one hand, energy-aware programming techniques enable programmers to *evaluate*[3] quality aspects of their program code from an ecological point of view. For example, if a specific functionality can be implemented in different ways (i.e., different program structure) and with varying energy demand this leads to a new *value system* to evaluate program code. On the other hand, energy-aware programming techniques create an opportunity to address economical interests at the design time of computing systems. By saving prospective energy resources at design time, this builds new ways of optimizing systems from the bottom (i.e., system software) up to high-level program code (i.e., application software). The increased resource efficiency has an economic impact (i.e., monetary savings).

[3] The word *evaluate* is a compound word which originates from *ex-* (Latin for *out* or *from*) and *value* (Old French, Gallo-Romance dialect, for *value* → French: *évaluer* → English: *evaluate*).

4.2 Values and Sustainability

As software programmers reveal that the energy demand of program code is different among heterogeneous hardware platforms, this leads to decision situations that concern hardware aspects. For example, depending on the software of a computing system it can be ecologically reasonable to substitute the current hardware platform with a different hardware platform on which the program code has a smaller energy demand. However, such decisions need to consider further consequences. For example, sustainability aspects need to be respected. If a target hardware platform has already been obtained it is not always reasonable to move to a new hardware platform if the entailed demand of resources (e.g., production, energy, money) is higher than the savings of the platform substitution.

4.3 Verification and Certification

For large-scale projects among different partners (i.e., research groups, industry partners) it is necessary to consolidate energy demands of all subsystems. Energy-aware programming concepts are a complementary extension to existing project development processes (i.e., at design time) where individual components are verified by corresponding unit tests. For industrial use it is important to consider certification requirements (e.g., ISO 50001 [Mc10]).

5 Embedding into the Doctoral Thesis

The author's doctoral thesis proposes several distinct energy-aware programming concepts which are presented in extracts in Section 3. This work is basis for further research on reducing the energy demand of computing systems, including large-scale systems in distributed environments. The entailed responsibility (Section 4) is a logic consequence and needs to be considered by future work targeting at energy-aware system design. Research work contributing to this paper builds the science base of the author's doctoral thesis. The individual publications [Hö12, Hö14, Hö15] are presented with their abstract in the following three subsections.

5.1 SEEP: Exploiting Symbolic Execution for Energy-Aware Programming

The research [Hö12] presents the initial work on energy-aware programming and exploits the use of symbolic execution techniques to analyze program code for energy demand.

> 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 that 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 knowledge to take energy demand into account during the task of writing programs.

5.2 Proactive Energy-Aware Programming with PEEK

PEEK [Hö14] proposes a programming kit around the task of energy-aware programming. The work focuses on automation aspects and energy measurements.

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 %.

5.3 The FigarOS OS Kernel for Fine-Grained System-Level Energy Analysis

The research on FIGAROS [Hö15] takes energy-aware programming aspects to the operating-system level and it investigates how system kernels can be made self-conscious with regard to energy and how static and dynamic optimizations are addressed at operating-system level.

Abstract—Energy has become the most important operating resource for computing systems of all sizes—from embedded systems to large-scale high-performance computing systems. However, at system level, engineers remain challenged at efficiently handling energy as first-class operating system resource. The reasons for this are twofold: First, increasingly complex hardware circuits are inherently difficult to model which makes the creation of accurate energy models practically impossible. Second, available energy measurements at system level are coarse-grained and they are insufficient for fine-grained system level energy measurements of the operating system.

The current advent of power constrained many-core systems and the road ahead towards the era of dark silicon requires efficient energy control mechanisms in the system software layer. In this paper, we present FIGAROS, an operating system kernel which implements primitives required for fine-grained system-level energy analysis. Our implementation of FIGAROS orchestrates energy measurements at hardware level by a low-level system software infrastructure at kernel level.

6 Conclusions

This paper presents approaches for energy-aware programming to address ecological and economical sustainability aspects for computing systems. Energy-aware programming is an essential measure to establish sensibility for energy demand of program code right at the time of development. Assisted by corresponding tool support programmers are able to reason about their programming decisions with regard to energy demand. The introduction of sensibility for energy demand at programming level has several consequences for programmers—and their design decisions. Besides optimizing their program code by reducing its energy demand, programmers are now in position to control further ecological and economical aspects affected by system and application software.

References

[Ca11] Cadar, Cristian; Godefroid, Patrice; Khurshid, Sarfraz; Păsăreanu, Corina S; Sen, Koushik; Tillmann, Nikolai; Visser, Willem: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. IEEE, pp. 1066–1071, 2011.

[CDE08] Cadar, Cristian; Dunbar, Daniel; Engler, Dawson R: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 12th Symposium on Operating Systems Design and Implementation. USENIX, pp. 209–224, 2008.

[CSP05] Choi, Kihwan; Soma, Ramakrishna; Pedram, Massoud: Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(1):18–28, 2005.

[DHKC09] Dawson-Haggerty, Stephen; Krioukov, Andrew; Culler, David E: Power optimization: A reality check. Technical report, EECS Department, University of California, Berkeley, October 2009.

[Es11] Esmaeilzadeh, Hadi; Blem, Emily; St Amant, Renee; Sankaralingam, Karthikeyan; Burger, Doug: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th International Symposium on Computer Architecture. ACM, pp. 365–376, 2011.

[Fo08] Fonseca, Rodrigo; Dutta, Prabal; Levis, Philip; Stoica, Ion: Quanto: Tracking energy in networked embedded systems. In: Proceedings of the 8th Symposium on Operating Systems Design and Implementation. USENIX, pp. 323–338, 2008.

[Ha10] Halperin, Daniel; Greenstein, Ben; Sheth, Anmol; Wetherall, David: Demystifying 802.11 n power consumption. In: Proceedings of the 2010 Workshop on Power-Aware Computing and Systems. USENIX, pp. 1–5, 2010.

[Hä12] Hähnel, Marcus; Döbel, Björn; Völp, Marcus; Härtig, Hermann: Measuring energy consumption for short code paths using RAPL. ACM SIGMETRICS Performance Evaluation Review, 40(3):13–17, 2012.

[Hö12] Hönig, Timo; Eibel, Christopher; Kapitza, Rüdiger; Schröder-Preikschat, Wolfgang: SEEP: Exploiting symbolic execution for energy-aware programming. ACM SIGOPS Operating System Review, 45(3):58–62, January 2012.

[Hö14] Hönig, Timo; Janker, Heiko; Eibel, Christopher; Mihelic, Oliver; Kapitza, Rüdiger; Schröder-Preikschat, Wolfgang: Proactive Energy-Aware Programming with PEEK. In: Proceedings of the 2014 Conference on Timely Results in Operating Systems. USENIX, pp. 1–14, 2014.

[Hö15] Hönig, Timo; Herzog, Benedict; Janker, Heiko; Schröder-Preikschat, Wolfgang: The FigarOS Operating System Kernel for Fine-Grained System-Level Energy Analysis. In: DAC Workshop on System-to-Silicon Performance Modeling and Analysis. ECSI, 2015.

[Ja12] Jan, C-H; Bhattacharya, U; Brain, R; Choi, S-J; Curello, G; Gupta, G; Hafez, W; Jang, M; Kang, M; Komeyli, K et al.: A 22nm SoC platform technology featuring 3-D tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density SoC applications. In: Proceedings of the 2012 International Electron Devices Meeting. IEEE, pp. 1–4, 2012.

[Ka07] Kansal, Aman; Hsu, Jason; Zahedi, Sadaf; Srivastava, Mani B: Power management in energy harvesting sensor networks. ACM Transactions on Embedded Computing Systems, 6(4):32, 2007.

[Ki76] King, James C: Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[KKP99] Kahn, Joseph M; Katz, Randy H; Pister, Kristofer SJ: Next century challenges: Mobile networking for "Smart Dust". In: Proceedings of the 5th International Conference on Mobile computing and Networking. ACM, pp. 271–278, 1999.

[Mc10] McKane, Aimee: Thinking globally: How ISO 50001 energy management can make industrial energy efficiency standard practice. Lawrence Berkeley National Laboratory, 2010.

[Mo05] Moore, Justin D; Chase, Jeffrey S; Ranganathan, Parthasarathy; Sharma, Ratnesh K: Making scheduling "cool": Temperature-aware workload placement in data centers. In: Proceedings of the 2005 Annual Technical Conference. USENIX, pp. 61–75, 2005.

[PHB15] Pallister, James; Hollis, Simon J; Bennett, Jeremy: Identifying compiler options to minimize energy consumption for embedded platforms. BCS The Computer Journal, 58(1):95–109, 2015.

[Sa11] Sampson, Adrian; Dietl, Werner; Fortuna, Emily; Gnanapragasam, Danushen; Ceze, Luis; Grossman, Dan: EnerJ: Approximate data types for safe and general low-power computation. In: Proceedings of the 32nd Conference on Programming Language Design and Implementation. ACM, pp. 164–174, 2011.

[TMW94] Tiwari, Vivek; Malik, Sharad; Wolfe, Andrew: Power analysis of embedded software: A first step towards software power minimization. IEEE Transactions on Very Large Scale Integration, 2(4):437–445, 1994.

[WB02] Weissel, Andreas; Bellosa, Frank: 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. ACM, pp. 238–246, 2002.