Automating Energy Optimization with Features

Norbert Siegmund
University of Magdeburg
Magdeburg, Germany
nsiegmun@ovgu.de

Marko Rosenmüller
University of Magdeburg
Magdeburg, Germany
rosenmue@ovgu.de

Sven Apel
University of Passau
Passau, Germany
apel@uni-passau.de

ABSTRACT

Mobile devices such as cell phones and notebooks rely on battery power supply. For these systems, optimizing the power consumption is important to increase the system's lifetime. However, this is hard to achieve because energy-saving functions often depend on the hardware, and operating systems. The diversity of hardware components and operating systems makes the implementation time consuming and difficult. We propose an approach to automate energy optimization of programs by implementing energy-saving functionality as modular, separate implementation units (e.g., feature modules or aspects). These units are bundled as *energy features* into an energy-optimization feature library. Based on aspect-oriented and feature-oriented programming, we discuss different techniques to compose the source code of a client program and the implementation units of the energy features.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Algorithms, Design

Keywords

Software Product lines, Energy consumption, Feature-oriented programming

1. INTRODUCTION

The widespread use of battery-supplied systems such as notebooks and mobile phones leads to a design shift toward energy optimization. Increasing the lifetime of a system is considered more important than an optimal performance. Moreover, due to increasing energy costs, energy optimization techniques such as virtualization are also in the focus of research for systems with direct power supply [37]. It is expected that energy cost of servers will soon exceed the purchase cost of the servers [7, 33]. Beside improvements in hardware architecture, the efficient use of hardware by software is a key factor to reduce energy consumption.

FOSD'10 October 10, 2010, Eindhoven, The Netherlands

Consequently, various compilers have been developed to reduce energy consumption (e.g., by instruction reordering and loop transformation [38, 16, 13]). Furthermore, operating systems provide their proprietary functionality to optimize energy consumption. For example, Windows Mobile[1] comes with an integrated power-management-component that can be used by applications to set different power modes for hardware components within the device. The Advanced Configuration and Power Interface (ACPI) defines an open standard for the power management related configuration of a system's hardware [19]. It enables applications to tune the energy consumption according the current workload. Laptops, desktops, and other hardware systems are deployed with this interface. However, in its current specification,[2] it consists of 727 pages, which reflects the complexity developers have to face and the required implementation effort for its realization.

Although there is a considerable number of energy optimization strategies, it has been observed that many applications do not use them [18, 25]. Reasons are heterogeneity of hardware with its unique energy-management functionality as well as complexity of software solutions (e.g., for ACPI) [4]. Adapting a software to different hardware devices and operating systems often increases development time and cost. Business constraints and release deadlines hamper the implementation of energy optimization techniques further.

The problem of handling variability and heterogeneity of hardware and operating systems in software development is not new. *Software product line (SPL)* engineering has shown to be a suitable approach to tackle this problem [12, 34]. An SPL consists of a set of features that represent end-user visible characteristics of a software. A user generates a tailor-made programs by selecting features that satisfy her requirements. This way, different programs can be generated based on a common code base.

We propose an approach to automate the application of energy-saving techniques (referred to as *energy features*) based on SPL engineering. The energy features are bundled in a *feature library*. The library consists of energy optimization functions for different hardware and operating systems. For example, a feature implements the functionality to activate and deactivate the WLAN for the Windows Mobile OS. By selecting energy features according to given system and user requirements, we can apply energy-saving functions to a target program. We use the feature library to achieve an improved separation of concerns regarding energy-saving techniques. Furthermore, the library allows us to reuse the energy features in different programs and to cope with the required variability (e.g., different hardware and operating systems). We make two contributions: (i) We provide means to ease the development of

[1] We consider OS version 5 and 6.
[2] Revision 4.0: `http://www.acpi.info/`

energy-efficient software and (ii) we propose a method for creating a library of energy-saving techniques. The expected advantages are:

- With the feature library, developers do not require deep knowledge about energy optimization. Energy-efficient features can be added without investigating which energy-saving technique is available for a particular system.
- Reusing energy features of the library reduces the development time of new products.
- We provide a simple configuration interface to utilize energy-reduction techniques. This allows even non-expert developers to optimize their programs.

We think that the energy feature library is needed to handle the diversity of techniques and make them available for practical use. As we describe see in Section 2.3, the kinds of optimization technique can heavily differ. Due to these differences, a variety of interfaces are required to make the features usable. In our current work, we concentrate only on those energy-saving techniques that are accessible by means of method calls. These techniques are commonly cohesive and generally applicable so that a reuse for different programs is possible. We show how feature-oriented programming [5] and aspect-oriented programming [23] can be used to compose a program with energy features. Furthermore, we present an approach to use energy-saving techniques by means of code instrumentation to access energy-management code.

This paper discusses ideas to apply feature-oriented software development to new domains such as energy optimizations. Our work concentrates on challenges that arise during design time of a software. Although, this work is at a theoretical level, it identifies problems that exist in other domains and present possible solutions.

2. BACKGROUND

Our approach is based on a library of features that implement different energy optimization techniques. We use *feature-oriented programming (FOP)* [35, 5] as a technique for modularizing the code that implements a feature. Next, we give a short overview of modeling features and how they can be implemented using FOP. Finally, we present some important representatives of energy-saving techniques.

2.1 Feature Modeling

We use a *feature model* to design the features of our library and their relations. A feature model is a hierarchical structure consisting of features and their constraints among each other [22, 14]. Figure 1 shows a feature model for the energy feature library. Mandatory features (e.g., feature *OS* of Figure 1) have to be included in every variant (denoted with a filled bullet) when the parent feature is selected. Optional features (denoted with an empty bullet; e.g., feature *Backlight*) are not required for each variant; it is up to the user to select them during the product configuration process. There are also groups of features representing additional variability. *Alternative groups* require the selection of exactly one feature of the group (not more than one); *or groups* allow the selection of one or more features of the group.

2.2 Feature-Oriented Programming

FOP is a technique that enables the implementation of features as fundamental modules of abstraction and composition [5]. This means, a feature exactly corresponds to a *feature module*. A feature module encapsulates the source code otherwise scattered across different classes inside a single modular code unit. It can define new classes or extend existing classes (as *refinements*) originally

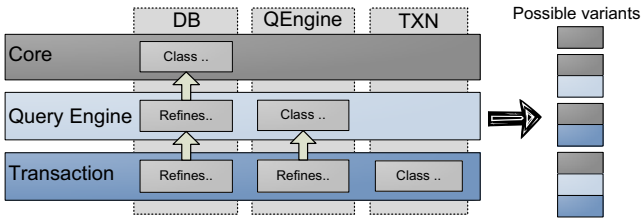

Figure 2: Decomposition of classes (vertical bars) with respect to features (horizontal layers).

defined by other feature modules. To derive a program, a set of feature modules is incrementally composed. This composition obtains final classes from classes and their refinements. In Figure 2, we show the architecture of a database management system (DBMS) implemented with FOP. Horizontal layers represent the features of the DBMS and the vertical bars represent classes. Starting from a base implementation of a class, multiple refinements, which belong to different features, are added resulting in a *layered design*. Refinements add new members to classes such as methods and attributes or extend existing methods with additional functionality. For example, if a user selects feature *Query Engine*, class *DB* would consist of the base implementation and a refinement to handle SQL queries.

2.3 Energy-Saving Techniques

There are different kinds of energy-saving techniques. We divide them into two classes: code transformation techniques and hardware- and operating-system–dependent techniques. Before we describe the two classes, we clarify the differences between power and energy, because it is necessary for understanding the techniques. Unfortunately, power and energy are often used interchangeably in literature [40]. The consumed power P of a system is the consumed energy E per time unit t:

$$P = \frac{E}{t} \tag{1}$$

Given that a system consumes energy[3], the intended goal is to save energy for a given task (in contrast to saving power). Considering Equation 1, reducing the power consumption may also reduce the energy consumption, but not necessarily. For instance, a task can either be performed with high power in a short time period or with low power, but for a longer time period. It depends on the task and algorithm what saves more energy. For example, constant time tasks should be performed with low power.

Code Transformation Techniques. The first category of energy optimization techniques contains different code transformation approaches. We first describe some solutions based on compilers and continue with source-to-source transformations that are compiler independent. Compiler approaches transform a software such that the energy consumption is reduced. One of the first solutions was proposed by Tiwary et al. [38]. The main idea is to reorder processor instructions such that less switches in CPU registers are necessary. Furthermore, they show how a reduction of memory operands in a program can lead to energy savings. Other optimizations are based on method inlining and loop unrolling [26] as well as on an energy efficient scheduling of a program's instruction set [39]. A more recent approach allows developers to define

[3]For example, batteries are charged with a certain amount of energy instead of power.

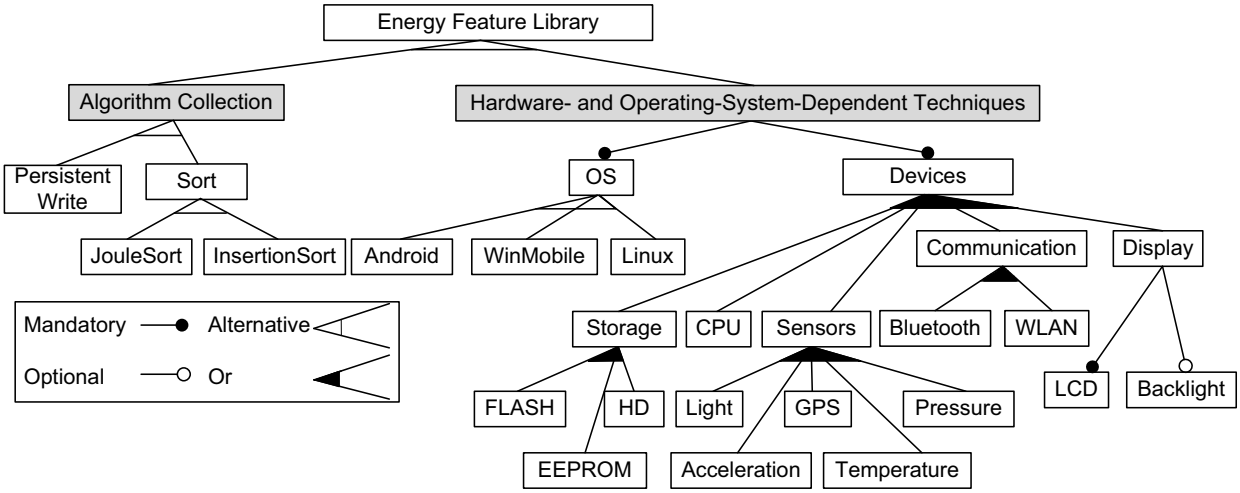

Figure 1: Feature model of the energy feature library.

objective functions for the compilation process [13]. Based on the objective function, the compiler selects suitable code transformations out of a pool.

Beside compiler-based optimizations, source-to-source transformations can also save energy. Fei et al. have shown that transforming the source code based on the analysis of the program's control flow can achieve energy savings of up to 37.9% [15]. The transformations reduce inter-process communication (reduces OS overhead for context switches), minimize the number of concurrent processes, and reallocation of computations of one process to another (reduces synchronization between processes).

Another study provides insights into energy consumption of language constructs in the C# programming language [10]. For example, dynamic anonymous methods consume 3 to 4 times more energy than dynamic methods. Another interesting fact is that protected attributes require two times more energy than private or public attributes. Of course, this may be different in other runtime environments; however, we observe that simple changes in the source code can have a large impact on energy consumption.

Beside application-independent transformations of the source code, the selection of a suitable algorithm is often more important for large energy savings. Researchers proposed energy-efficient algorithms such as for sorting large amounts of data [36] as well as studies about choosing the optimal algorithm to reduce energy consumption [11, 9] which can even be taken a step further to combine hardware-dependent techniques with energy-efficient algorithms. For example, Pisharath et al. reduces energy consumption of queries in an in-memory database system by selectively setting different memory banks into a certain power state [31]. Controlled by an intelligent query engine of the database system, energy consumption could be decreased by up to 68%. Solutions that are very specific for certain application scenarios are very difficult to encapsulate into a single implementation unit. Thus, reuse in different programs is unlikely. Therefore, we currently focus only on generic algorithms for the feature library that fulfill reoccuring tasks in different programs.

Hardware- and Operating-System–Dependent Techniques. Beside code transformation techniques, software can also efficiently control hardware components to save energy. The most prominent techniques are dynamic voltage scaling [29, 21] and frequency scaling of CPUs [30, 32]. Both approaches are used to reduce the power when a system has a low workload. Since the power P depends proportionally on the frequency f and the voltage v squared, reducing the frequency as well as the voltage decreases the used power:

$$P \propto f * v^2 \tag{2}$$

For constant-time tasks, reducing power consumption proportionally reduces energy consumption (as it follows from Equation 1). Therefore, both factors (cf. Equation 2), frequency and voltage, should be adjusted depending on the current workload of the system. Some approaches analyze applications statically to include the scaling commands into the application (e.g., when certain events occur) [41]. Other solutions dynamically change the frequency or voltage in a pre-defined time interval [3]. These techniques use profiling mechanisms to predict the future workload of the system so that dynamic adaptations are possible. Currently, we do not address this kind of optimization because it is difficult to provide a general applicable approach that saves more energy than already existing solutions provided by operating systems.

ACPI is an industry-driven specification for power management of hardware devices. It allows the operating system to control different power states of the hardware. For example, a device could be set in four different states; beginning with a full powered on state (*D0*) and ending with powered off state (*D3*). The different states can be set by programs by means of operating system API calls.

Operating systems for mobile devices provide special power management solutions similar to ACPI. For example, Windows Mobile comes with its own power management.[4] Applications can control the power states of different devices such as the backlight, the screen, etc. Setting the device into a specific state requires API calls to the operating system. Another example is the *Android*[5] power management that requires *Wake Locks* to request CPU resources. If there is no wake lock active, the CPU will shut down (and saves energy). As Windows Mobile, Android supports different power states. Unfortunately, setting the device into a specific state differs in both operating systems.

The described operating-system techniques are used by means of

[4]For detailed information, see `http://msdn.microsoft.com/en-us/library/aa923906.aspx`.

[5]`http://www.android.com/`

API calls. Thus, a feature that utilizes the provided energy management functionality of operating systems can be easily implemented as a separate, self-contained implementation unit. The energy feature library contains especially such features because a reuse between different programs is easy to achieve.

3. TOWARDS AN ENERGY FEATURE LIBRARY

Due to the multitude of possible energy-saving techniques, it is hard for developers to know every possible mechanism wasting a large quantity of possible energy savings. This problem even grows if the target hardware and operating systems varies for the same program. In this section, we present our approach to cope with the heterogeneity of techniques by developing a library of energy-saving technique. Our idea for an automated optimization is the integration of code of the feature library in a client program. Depending on the execution of a feature's code of the client program, we (i) replace this code with the code of the library or (ii) (de)activate hardware components. Unfortunately, we cannot implement every mechanism to reduce energy consumption with this approach. For example, compiler-based optimizations cannot be applied with a feature library.

Thus, we include only operating-system-dependent techniques as well as some code transformations in the energy feature library. However, the idea is that the library can be extended by others providing their own techniques. Next, we describe the envisioned features of the library. We illustrate how application engineers can use the energy feature library to select proper energy-saving techniques for their program.

3.1 Modeling the Energy Feature Library

We use a feature model to relate the different techniques for energy management. This allows us to define a view on the energy optimization domain as well as to use the feature model for the configuration process. In Figure 1, we depict an initial proposal for the energy feature library. The energy feature library contains of two main parts: *Algorithm Collection* and *Hardware- and Operating-System–Dependent Techniques*.

Algorithm Collection. We create a library of algorithms that minimize the energy consumption. Subfeatures of *Algorithm Collection* implement common, recurring algorithms in an energy-efficient manner. For example, feature Sort in Figure 1 contains two subfeatures that sort data with a minimal energy consumption [36, 8]. Developers of different domains can further add their domain-specific algorithms to enrich this algorithm library. Possible examples are energy-efficient read and write of data from and to persistent storage [20], different data aggregation strategies for network communication [24, 6], or data caching algorithms to reduce communication [28]. Although a large amount of such algorithms are often only reusable in a certain domain, we expect that the benefit for the domain is still considerable.

The idea of algorithm collection is to replace the algorithms in the target application with the algorithm features of the *Algorithm Collection*. This way, we can exchange an algorithm of a program with another one that has an equal functionality but requires less energy. To use an algorithm, a developer has to encapsulate the corresponding algorithms with a proper interface in her program. For example, if a developer wants to exchange the algorithm for sorting data, she has to separate the sort algorithm into a single implementation unit, i.e., a feature module. This implementation unit can then be replaced by the feature module of the algorithm collection (e.g., feature *InsertionSort*). If the interface of the algorithms do not match, developers have to implement an adapter to integrate the algorithm of the library into the program. The rational behind different sorting algorithms is that the efficiency of a sorting algorithm regarding energy consumption depends on the amount and kind of data to be sort. Depending on these factors, a user selects the sorting algorithm that fits best to her workload.

Hardware- and Operating-System–Dependent Techniques. To ease the selection of energy-saving techniques, we use features for operating systems (feature *OS* and subfeatures) and for hardware components (feature *Devices* and subfeatures). A selection of these features together map to the feature module(s) that implement the energy-saving technique(s). In detail, feature *OS* provides energy management functions for different hardware components as described in Section 2.3. Since these energy management functions strongly dependent on the used device, we have to model the hardware, too. The possible available hardware components of a system are represented as subfeatures of feature *Devices*, in Figure 1. Together with the selection of the operating system, we can select a feature module that efficiently utilizes the hardware. For example, if a developer selects feature *WinMobile* and feature *Backlight*, we include the feature module in the program that powers the backlight on and off. The place for inclusion is defined separately, as describe below. In Section 4, we show the usage of features *WinMobile* and *WLAN*. Selecting the two features enables a program to trigger the power mode of a WLAN device depending on the current executed functionality. Overall, selecting the subfeature of feature *OS* defines which operating system API calls have to be composed and selecting a subfeature of feature *Devices* defines which hardware component has to be controlled by the program.

3.2 Mapping Energy Features to Application Functionality

To use energy features, we need a mapping between the functionality that has to be optimized and the feature module of the energy feature. Thus, a developer configures the energy feature library for client application feature that has to be optimized. This means, each feature of the client program maps to a configuration of the library. Based on this configuration, we derive the correct feature module that maps to the functionality of the program. In Figure 3, we show such a mapping between a program and the energy feature library. In this example, the program is a variant of an SPL for data management solutions targeting resource-constrained systems. On the left side of Figure 3, we depict the features of the DBMS SPL that should be optimized regarding energy consumption. For each feature, we configure the energy feature library (right side of Figure 3) using the feature model of Figure 1. The configuration of the energy library results in a set of feature modules that realize the energy-saving functions. These feature modules are mapped to the corresponding modules of the DBMS SPL (center of Figure 3). For example, to power the WLAN device only when feature *Distribution* of the DBMS SPL is executed, we create a mapping between this feature and the feature module *W_W* which is derived from the configuration of the energy feature library. The result is that the energy management for the WLAN device is controlled by the program. The device is powered only when the program uses its communication functionality. Establishing such mappings is a straightforward process. It requires only little expert knowledge about energy management of operating systems and hardware components.

If a developer wants to use energy-efficient algorithms, she has to create a mapping between the algorithm feature of the algorithm

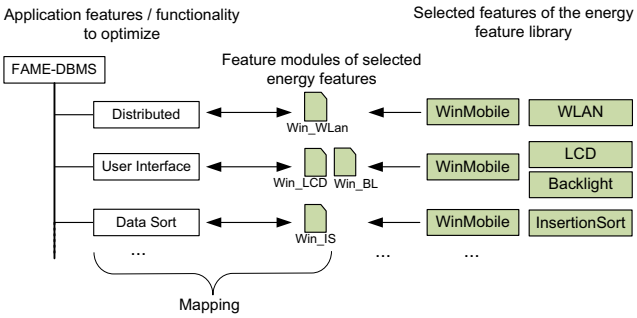

Figure 3: Mapping application features to energy features.

collection and the algorithm feature of her program. For example, the mapping between feature *Data Sort* and feature module *W_IS* in Figure 3 results in a DBMS variant that uses the InsertionSort algorithm for sorting data; instead of the original implementation. Beside the reduced energy consumption, the developer has to take additional effects on non-functional properties into account. For example, by applying feature *InsertionSort* the performance may be decreased. If performance is more important than energy consumption, the mapping should not be established.

4. PRODUCT GENERATION

A key contribution of our approach is the application of energy-saving techniques to programs with a very small implementation effort for the developer. The technique requires only a mapping between features of the application SPL and features of the energy feature library. Depending on the selection of an algorithm feature or an hardware-dependent feature, different composition mechanisms have to be used. For algorithm features, we have to exchange the implementation unit of the program with a feature module of the feature library. When using FOP, this is a straightforward process because either the existing feature modules must be physically replaced or the configuration of the application SPL must be changed accordingly. For hardware-dependent techniques, additional code from the library must be integrated into the program. This requires new implementation techniques to connect application features with energy-optimization code. We propose three different implementation variants: *Manual Layered Composition*, *Automated Layered Composition*, and *Energy Manager*. We explain all approaches in detail and discuss their advantages and drawbacks.

4.1 Manual Layered Composition

The first approach to realize the integration of energy optimization code is using the layered composition process of FOP. Features modules are represented as layers. Similar to classes, methods are composed from multiple refinements resulting in a refinement chain executing the functionality of all existing refinements sequentially. This way, functionality of the method from different feature modules is executed depending on the order of the refinement chain (or layers). To access the energy management, a developer has to implement a feature module that refines the application's functionality. This means, a developer has to write method refinements (consisting glue code) for those methods that have to be optimized regarding energy consumption. To simplify manual implementation, we envision a skeleton that enables a semi-automated generation process. Such a skeleton can provide the implementation of the energy code for each feature of the energy feature library. It requires only the declaration of the method that is the target for energy optimization. In Figure 4, we show an example for such a skeleton based on FeatureC++[6]. FeatureC++ is a C++ language extension for FOP [1]. It comes with a source-to-source compiler. FeatureC++ uses keyword **super** (Line 9) to execute the next (above) method refinement of the composed method. In square brackets, a developer has to include the name of the class and method (Lines 1-2) that has to be refined with the energy-saving code. Furthermore, if there is a return value, it has to be stored temporarily (Line 8) and returned after the device is powered off (Line 13). The benefits of this approach are the fine granularity of changes to the program's code (i.e., only selected methods are surrounded with energy management code) and the direct control of code changes by developers. This allows developers to have a direct influence on the code changes.

Feature Module WLAN_WinMobile_On (Win_WLan)

```
1  refines class [CLASS_NAME] {
2    [RETURN_TYPE METHOD_NAME]([ARGS]) {
3    //Turns the WLAN device on
4    _energySplClass.SetDevicePower
5      (_energySplClass.getWirelessDeviceName(),1,0);
6
7    //Execute the send functionality
8    [RETURN_TYPE RETURN_VALUE=]super::[METHOD_NAME(ARGS)];
9
10   //Turns the WLAN device off
11   _energySplClass.SetDevicePower
12     (_energySplClass.getWirelessDeviceName(),1,4);
13   [return RETURN_VALUE];
14  }
15 };
```

Figure 4: Skeleton of energy-optimization feature modules (FeatureC++ example).

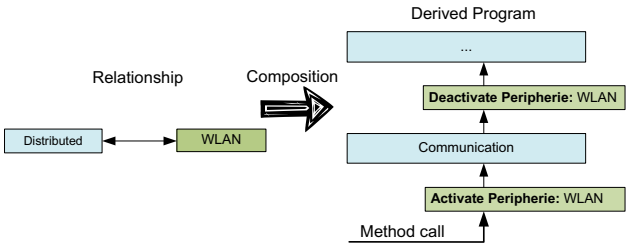

Figure 5: Composing application features with energy optimization code.

4.2 Automated Layered Composition

Another solution to compose the energy features with the program is based on aspectual feature modules[7] [2], as illustrated in Figure 6. The idea is to enclose the code of the program's feature with code of the feature library (cf. Figure 5) to manage hardware components. In this solutions, features are implemented as aspectual feature modules. An aspectual feature module can introduce aspects, method refinements, and new classes to a program. The aspects are weaved into the application to access the energy management code. The energy management code can be implemented, in turn, as classes, also part of the aspectual feature module. In order to access the energy management functionality, we use an around advice (Line 2–3) for each method belonging to a

[6] http://fosd.de/fcc

[7] Aspectual feature modules combine the concepts of FOP and aspect-oriented programming (AOP) [23].

Aspectual Feature Module WLAN_WinMobile (Win_WLan)

```
1  aspect WLAN_Energy_Management {
2   pointcut exWLAN() = execution("%.Send_Receive%(...)");
3   advice exWLAN() : around (){
4    //turn wlan on
5    _energySplClass.SetDevicePower
6    (_energySplClass.getWirelessDeviceName(),1,0);
7
8    proceed();//Execute the send functionality
9
10   //turn wlan off
11   _energySplClass.SetDevicePower
12   (_energySplClass.getWirelessDeviceName(),1,4);}
13 }
14 class EnergySplClass {
15  //Load ossvc library
16  EnergySplClass(){..}
17  //Set different power modes
18  SetDevicePower(..){..}}
19 }
```

Figure 6: An Aspectual feature module to apply WLAN (de) actionvation for feature Send_Receive (FeatureC++ example).

feature (of the application SPL) in a mapping. Depending on the energy-optimization technique, the piece of advice activates or deactivates different hardware. For example, we activate the WLAN device in Lines 5–6 and deactivate it in Lines 11–12 (cf. Figure 6). The whole aspectual feature module is stored in the library. The only part which needs to be generated is the pointcut for each public method of the program's feature. In Figure 6, we include the *Send_Receive* feature name in the execution pointcut. In order to select join points of aspects of a single feature, the feature has to be part of the pointcut definition.

4.3 Energy Manager

Another implementation technique is based on an energy manager that stores the mappings between features of the application SPL and the library. The energy manager is statically composed with the program including the mappings. It is accessed each time when the program execution reaches the code of a feature. Before feature code is executed, the manager checks whether it has to activate or deactivate a certain device. For example, when the DBMS calls the *send* method of feature *Send_Receive*, the manager is called to check for existing mappings. In this case, we have to execute the energy management to turn the WLAN on. To query the manager for mappings, we have to integrate a method call at the entry points of every method refinement of a feature. Such a code instrumentation techniques is also used in dynamic AOP approaches in which hook methods are generated to enable execution of additional code at runtime [17, 27]. FeatureC++ can be used to generate such method calls during the composition process for each program feature, which is the main difference to the aspectual-feature-module solution. The method calls were originally intended to support activation and deactivation of features at runtime. We use this technique to include a method *OnExecute(FEATURE_NAME)*. This method accesses the energy manager to evaluate if there is an existing mapping for the current feature. If present, this code is executed (e.g., the WLAN device is powered on). The device is turned off when the execution of the refinement has finished. This can be achieved by generating a method *OnExecuteExit(FEATURE_NAME)* before each return statement.

4.4 Discussion

Reuse of Energy Features. Usually, features are reused within different variants of a single software product line. In such a case, reuse is the main goal of the design and architecture of features. When using a library for energy optimization techniques, features have to be reused in very different programs. For common application features, a reuse would be very difficult to achieve. Obvious reasons are application-dependent interfaces, heterogeneous data structures, etc. However, the characteristics of energy optimization techniques suggest that their reuse in very different programs is possible. Energy features that rely on code transformation, are very generic. There is no application specific code in such a feature. Features for utilizing the energy management for different hardware components are often cohesive. That is, turning a hardware component on and off does not require complex adaptations of the application. An example for this case is ACPI that provides an interface for energy management accessible in different operating systems.

Rapidly Switching Between Different Features. A problem arises if we quickly enter and leave the code of a feature that belongs to a mapping. Putting a device in a certain power mode consumes time, the performance decreases, and energy is wasted. A possible solution is to use timers for the deactivation of hardware. The time interval for deactivation should be defined during the configuration and is stored together with the mapping. A finer granularity may be used to set different timers for individual features.

Performance. The execution of additional code due to the integration of the energy features may decreases the performance. This is especially the case when using the energy manager implementation because we execute the *OnExecute()* method for each feature. Hence, there is a trade-off between performance and energy optimization. We have to evaluate how large the impact of such a realization is. We will address this issue in further work. Nevertheless, when an application performs a constant time task, performance does not matter. This means, when we decrease the performance, the performed task may not require more time to finish. For these tasks, applying energy optimizations which come with small performance degradation will not have a negative effect.

Architectural Adaptations. We need a feature model of the program to create the mapping between program features and energy features. In some cases, a program's feature has to be decomposed into two features when it is not feasible to apply the energy management to the whole feature. For example, feature *Distribution* of the data management SPL consists of various functions that realize data distribution for different databases. Only a small part of this feature actually requires the WLAN device to send and receive data. Thus, only this part should be mapped to an energy feature of the library. Such an architectural change represents a way to apply the energy optimization code only at the point where it is really needed. Thus, there is a trade-off between a good program design and energy optimization. With additional implementation effort by the application developers the manual layered composition and automated layer composition can solve this problem. For example, pointcuts can be manually defined to address only the important methods. This hampers a complete automated generation, but may be more appropriate than a restructuring of the application SPL. The first approach of the layered implementation already requires

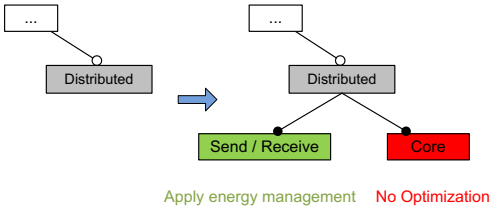

Figure 7: Dividing feature Distribution into two subfeatures to apply the optimization only to the Send / Receive functionality.

a partial manual implementation of energy feature modules.

Handling Method Calls Inside Encapsulated Methods. If the control flow leaves the functionality of a feature with energy management before it is completely executed, we may waste energy because hardware components only are powered off at the end of the feature's functionality. As an example consider the *Send_Receive* feature, if we have to wait for user input (e.g., to type in a password) in the send method, we might have to wait a long period of time and thus waste energy. The question is whether to turn the device off before leaving the method or keep it active. We think that an appropriate decomposition of application SPL features can reduce the occurrence of such problems.

Application SPL. Currently, we consider programs developed as SPLs, i.e., decomposed into features. However, this is not necessarily required. We only need a description of the functionality of the program that maps one-to-one to implementation artifacts (e.g., a feature model or a component model). We use this description to attach energy optimizations to the application's functions. This way, we can determine which functionality requires which device and can thus be optimized regarding energy consumption. To sum up, a serious reduction of energy consumption may require a design shift of software development towards energy optimizations. This means, a program needs an appropriate design of its features with respect to energy optimization (as we already discussed).

5. CONCLUSION

We presented an approach that allows programmers to use energy-saving techniques without the need of special knowledge about energy optimization. Developers do not have to invest time to learn different energy saving techniques and to apply them to their programs. We use software product line (SPL) techniques to model and implement different energy-saving techniques such as operating-system- and hardware–dependent functions. The functionality is implemented as separate implementation units (feature modules) and bundled in an energy feature library. Researchers and developers can add their own energy-efficient algorithms as features to the library.

To integrate the energy features into the program, we use mappings between program functionality and energy features. For example, a feature for sending data in a software maps to a WLAN feature of the energy feature library. Such a mapping is used to compose the source code of the energy features with the code of the program. We presented three approaches to implement this composition. The first two approaches use the layered design of feature-oriented programming. Layers represent features (or functionality) of a program. A program's feature can be composed with a feature of the library. The composition can be performed semi-automated (based on a skeleton class that contains energy-optimization code) or automatically by generated pointcuts of aspects. Furthermore, code instrumentation can also be used to insert an access method each time the program reaches feature code. The access method calls an energy manager that checks for existing mappings. If a mapping of the current feature exists the related energy optimization is executed.

In future work, we will evaluate the different approaches regarding energy-savings and the impact on performance. Furthermore, we will analyze if such an approach can also be used to optimize other non-functional properties such as performance and memory consumption.

Acknowledgement

Norbert Siegmund is funded by the ministry of education and science BMBF, number 01IM08003C. Marko Rosenmüller is funded by the German research foundation, project number SA 465/34-1. Apel's work is supported in part by the DFG projects #AP 206/2-1 and #AP 206/4-1. The presented work is part of the ViERforES[8], MultiPLe[9], FeatureFoundation[10], and SafeSPL projects.

6. REFERENCES

[1] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, Sept. 2005.

[2] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the International Conference on Design, automation and test in Europe*, page 168. IEEE Computer Society, 2002.

[4] L. A. Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[6] L. Becchetti, A. Marchetti-Spaccamela, A. Vitaletti, P. Korteweg, M. Skutella, and L. Stougie. Latency-constrained aggregation in sensor networks. *ACM Trans. Algorithms*, 6(1):1–20, 2009.

[7] C. Belady. In the data center, power and cooling costs more than the it equipment it supports. *Electronics Cooling*, 13(1), 2007.

[8] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Proceedings of the International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 600–607. IEEE Computer Society, 2009.

[9] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. In *Proceedings of the International Conference*

[8] http://vierfores.de
[9] http://fosd.de/multiple
[10] http://fosd.de/ff

on Software and Data Technologies (ICSOFT), pages 199–206, 2009.

[10] K. Chantarasathaporn and C. Srisa-an. Object-oriented programming strategies in c# for power conscious system. *International Journal of Computer Science {Online}*, 1(1), 2006.

[11] A. Chatzigeorgiou and G. Stephanides. Energy metric for software systems. *Software Quality Control*, 10(4):355–371, 2002.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[14] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Trans. Embed. Comput. Syst.*, 7(1):1–26, 2007.

[16] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *J. Embedded Comput.*, 1(4):509–520, 2005.

[17] W. Gilani and O. Spinczyk. Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays*, pages 94–109. Gesellschaft für Informatik, 2005.

[18] Global Action Plan. An inefficient truth (white paper), 2007. http://greenict.org.uk/sites/default/files/An%20Inefficient%20Truth%20-%20Full%20Report.pdf.

[19] A. Grover. Modern system power management. *Queue*, 1(7):66–72, 2003.

[20] I. Hong and M. Potkonjak. Power optimization in disk-based real-time application specific systems. In *Proceedings of the International Conference on Computer-aided design*, pages 634–637. IEEE Computer Society, 1996.

[21] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low power electronics and design*, pages 197–202. ACM, 1998.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[24] P. Korteweg, A. Marchetti-Spaccamela, L. Stougie, and A. Vitaletti. Data aggregation in sensor networks: Balancing communication and delay costs. *Theor. Comput. Sci.*, 410(14):1346–1354, 2009.

[25] M. S. Lane, A. Howard, and S. Howard. The energy inefficiency of office computing and potential emerging technology solutions. *Journal of Issues in Informing Science & Information Technology*, 6:795–808, 2009.

[26] Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded hw/sw systems. pages 259–264, 2002.

[27] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[28] M. Pedram. Power optimization and management in embedded systems. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 239–244. ACM, 2001.

[29] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low power electronics and design*, pages 76–81. ACM, 1998.

[30] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 89–102. ACM, 2001.

[31] J. Pisharath, A. Choudhary, and M. Kandemir. Reducing energy consumption of queries in memory-resident database systems. In *Proceedings of the International Conference on Compilers, architecture, and synthesis for embedded systems*, pages 35–45. ACM, 2004.

[32] C. Poellabauer, T. Zhang, S. Pande, and K. Schwan. K.: An efficient frequency scaling approach for energy-aware embedded real-time systems. In *Proceedings of the International Conference on Architecture of Computing Systems*, 2005.

[33] M. Poess and R. O. Nambiar. Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results. *Proc. VLDB Endow.*, 1(2):1229–1240, 2008.

[34] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[35] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[36] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the International Conference on Management of Data*, pages 365–376. ACM Press, 2007.

[37] E. Saxe. Power-efficient software. *Queue*, 8(1):10–17, 2010.

[38] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of Symposium on Low Power Electronics*, pages 38–39, 1994.

[39] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Proceedings of the International Conference on Design, automation and test*, pages 855–860. IEEE Computer Society, 1998.

[40] M. A. Viredaz, L. S. Brakmo, and W. R. Hamburgen. Energy management on handheld devices. *Queue*, 1(7):44–52, 2003.

[41] T. Yokoyama, G. Zeng, H. Tomiyama, and H. Takada. Analyzing and optimizing energy efficiency of algorithms on dvs systems a first step towards algorithmic energy minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 727–732. IEEE Press, 2009.