Cluster Comput
DOI 10.1007/s10586-016-0691-5

Code refactoring techniques for reducing energy consumption in embedded computing environment

Doohwan Kim[1] · **Jang-Eui Hong**[1] · **Ilchul Yoon**[2] · **Sang-Ho Lee**[1]

Received: 5 October 2016 / Revised: 13 November 2016 / Accepted: 18 November 2016

Abstract Code refactoring is the process of restructuring existing computer code without changing its external behavior to enhance reusability and maintainability of software components through improving nonfunctional attributes of the software. However, when we refactor source codes using existing refactoring techniques, those techniques do not consider energy consumption as one of the nonfunctional attributes. Reducing energy consumption is one of the important factors to develop embedded and/or mobile software because it is difficult to provide sustainable services based on limited power resources. This paper suggests new refactoring techniques for reducing energy consumption to support the restructuring of existing source codes. Especially we define energy-consuming constructs as suspicious codes that are predicted to consume a lot of energy, and then develop the techniques to remove these constructs. Our techniques can improve the performance as well as the energy efficiency of legacy codes.

Keywords Code refactoring · Energy consumption · Energy consuming constructs · Energy-efficient constructs · Embedded software

✉ Jang-Eui Hong
jehong@chungbuk.ac.kr

Doohwan Kim
dhkim@selab.cbnu.ac.kr

Ilchul Yoon
icyoon@sunykorea.ac.kr

Sang-Ho Lee
shlee@chungbuk.ac.kr

[1] Department of Computer Science, Chungbuk National University, Cheongju 28644, Republic of Korea

[2] Department of Computer Science, The State University of New York, Korea, Incheon 21985, Republic of Korea

1 Introduction

While the users of mobile devices such as smart phones and tablets are increasing, the applications for mobile devices are becoming more complex and larger in size in order to satisfy and meet diverse user needs. Consequently, the time mobile devices are used has dramatically increased. However, mobile devices have a limitation in their power supply. Therefore, it is an important issue to increase the operational time while providing the functionality and performance of mobile devices in this environment [1–3].

Study on increasing the operation time of mobile devices can be classified into three dimensions: battery capacity increment [4], low-power hardware elements [5–7], and low-energy software development [8–11]. Also, the studies on low-energy software development have been performing into two main streams: model-based analysis of software energy to predict consuming power in the early stages of development [10–13] and code-based analysis to verify the energy requirements [14–17]. The former case has the advantage of being able to perform the analysis to meet low-power requirements, and the latter case has the advantage of obtaining a highly reliable energy analysis due to using the actual implementation code. The purpose of this study was inspired by code-based analysis to suggest code refactoring techniques for reducing energy consumption.

Currently, as many popular techniques for code-based refactoring have become well-formed, these techniques provide the advantage of easy restructuring and enhance the quality of codes through applying the techniques during source code development and its maintenance [18]. In addition, by defining ‘Code Smells’ as potentially corruptible codes, and also defining ‘Bad Smells’ as those code smells which have a negative impact to code quality, it is possible

Published online: 30 November 2016

to refactor codes easily, and easy to apply these techniques to remove the bad smells [18].

There are also some studies about the code refactoring techniques for reducing energy consumption [19–21]. These studies defined 'Energy Bug' or 'Energy Code Smell' with respect to the behaviors of software that cause unnecessary energy consumption in mobile devices, and proposed refactoring techniques to remove these bugs. However, the targets under consideration for the reduction of energy consumption are limited to recovery management for hardware resources, color arrangement on display characteristics, and code removal for pop-up ads.

Even though these considerations can anticipate the reduction of energy consumption, it is not useful to software developers who are concerned with functional code itself, or, who want to analyze or observe the behaviors of the operation system, because these considerations only focus on the auxiliary functions, not the core functions. In addition, the existing methods are different with the code refactoring techniques which consider the functional logic of software, and also there is a disadvantage that the applicable target is limited.

In this paper, we define the energy consuming constructs (ECCs) with code structures that have the possibility of energy reduction, and then propose energy-efficient code refactoring techniques for restructuring the ECCs. The proposed techniques are verified through experiments for several example applications. The suggestion of this paper is summarized as follows:

- Identify ECC: We identified ECC with the code structure that is expected to consume lots of energy through investigating the open source codes. This is identified with the fundamental philosophy of the refactoring techniques by Fowler. The identified ECCs appeared to seven.
- Develop the methods to transform the ECC to energy-efficient code (EEC): We systematically defined the methods to converting ECCs to EECs. These methods are our seven techniques to refactor source codes with respect to the seven ECCs.
- Verify the energy-efficient code refactoring techniques: The seven refactoring techniques were verified through experiments to confirm that provide energy savings. We define a systematic procedure for the experiment, and performed the experiments according to the defined procedure for each technique by applying example source codes

Our suggested refactoring techniques showed an average reduction of 2.4% in energy consumption compared to the original codes in the experiments. This is enough to provide more than 4 min of usage in a typical mobile device. The application of our study results is expected to meet the low-energy or low-power requirements more easily in the development and maintenance of mobile software.

This paper is organized as follows: Sect. 2 surveys related work concerned with reducing the energy consumption of software and the refactoring of source codes with the intention of energy reduction. The ECCs are defined by identifying the code structures that have high energy consumption in Sect. 3, and our refactoring techniques for converting ECCs to EECs are described in Sect. 4. Section 5 contains our experimental results to show the applicability of the techniques, and Sect. 6 shows a case study and its results analysis. Lastly, the conclusive summary and further work explained in Section 7.

2 Related work

Recent studies for reducing energy consumption by focusing on source codes are increasing in the direction of code refactoring techniques [22–29].

First, there are some studies carried out using the Fowler's refactoring techniques [18]. As one of the studies, Silva et al. [22] analyzed the changes between the performance and the power consumption while applying the technique 'Inline Method' repeatedly to a specific application. This study showed that the technique 'Inline Method' can support the performance enhancement and power reduction for a specific application.

Conversely Pérez-Castillo's study [23] showed that the application of the techniques 'Extract Method' and 'Extract Class' (techniques proposed by Fowler) increase the energy consumption of mobile devices due to the incremental increase of message traffic between the objects. Additionally, Park et al. [24] carried out an estimation of energy consumption for the whole 58 refactoring techniques by Fowler, and analyzed the changes in energy consumption when each technique was applied to source codes.

These studies show that energy consumption can be considered as one of the factors of refactoring in addition to code quality like well-formed structure. However, all of the existing code refactoring techniques cannot reduce energy consumption; also they do not offer any solution in cases where energy consumption was increased.

Recently, the studies on code refactoring have been trying to identify the code patterns for large energy consumption, and remove those patterns from source codes.

Pathak et al. [25] has defined 'Energy Bug' as the factors that increase the power consumption in mobile devices. The 'Energy Bug' is classified at the hardware and the application level; the bug at the application level consists of No-Sleep, Loop, and Immortality. The No-Sleep bug is to prevent the release of hardware resources after its use. The Loop bug means that the system performs some operations repeatedly

even unexpected results, and the Immortality bug means that an application is continuously restarted by another application even after the application was terminated.

Gottschalk et al. [26] performed an energy consumption analysis on Android-based mobile devices by defining the 'Energy Code Smell' based on the Application Energy Bug. This study analyzed source codes to identify the Energy Code Smell, and proposed the 'Energy Refactoring' technique through the reconstructing of the code. However, because the smell 'Third-Party Advertisement' (among his proposed five Energy-Code-Smells) removes pop-up ads from the applications' codes, its refactoring is not useful to mobile application developers. Also, even though the smell 'Backlight' has a positive factor on energy consumption, because it depends on the type of liquid crystal displayer and the scheme of color representation in a mobile device, it is not effective in reducing energy consumption because the consideration of the visual design of the application is one of the most important factors from the users' perspective.

Although the existing studies analyze the energy consumption of software, or propose code refactoring techniques with the aim of reducing energy consumption, they do not contain enough detail to assist developers of mobile and/or embedded software who are concerned with the application procedure for restructuring source codes.

3 Identification of energy consuming constructs

3.1 Energy Code Smell

Fowler defined the 'Code Smell' as an indicator that there is a more serious problem within a system [18]. The representative 'Code Smells' noticeable in general code refactoring are presented in Table 1.

The 'Code Smell' is code piece that has a high possibility of inherent errors, or a low performance due to too long a code length. Code refactoring techniques are applied to the identified 'Bad Smells' which are determined as a factor to degrading software quality by analyzing the Code Smell. Generally, Code Smells such as 'Duplicated Code' or 'Long Method' can improve code quality through the application of code refactoring techniques. However, the smell 'Long Parameter List' has the constraint that the dependencies between objects should not occur.

Similarly, Vetro's study [30] defined 'Energy Code Smell' as the probable code pattern which can reduce energy consumption through the modification of code structure. He proposed refactoring techniques for each 'Energy Code Smell', and derived the energy analysis results as shown in Table 2. The experimental results showed that the five Code Smells can be improved to achieve the energy savings through refactoring.

Table 1 Representative refactoring techniques by Fowler [18]

Code Smells	Characteristics
Duplicated Code	The code of same logic appears multiple times
Long Method	The length of method is long and has many functions
Large Class	A class has a lot of functions
Long Parameter List	The number of parameters for method call is large
Data Clump	A bunch of data is used to call at the same time always
Temporary Field	The temporary variables to support complex algorithms
Message Chains	Client takes several steps to reach a particular object
Incomplete Library Class	Functional and structural imperfections in class library

Table 2 Energy Code Smells by Vetro et al. [30]

Energy Code Smells	Description	Impact (%)
Parameter by value	Passing parameters to a function	−0.09
Self assign	Assign the value with own value (e.g., $x = x$)	0.11
Mutual exclusion OR	OR operations has always TRUE value	−0.03
Switch redundant assign	The switch statement without break	0.17
Dead local store	Never used local variable	0.60
Dead local store return	Local variable that is not assigned to return	0.07
Repeated conditionals	Redundant check of condition statement	0.18
Non short circuit	Use the operators &&, || instead of &, | operators	−0.08
Useless control	Control statement does not change the path flow	−0.22

It can be seen that the Code Smells obtaining energy savings appear when causing unnecessary operations due to an error occurring in the execution flow of the codes. However, the effect of energy savings obtained through modification of the pattern is less than 1% which becomes the limit to get the application effects of refactoring technique. In addition, because the Energy Code Smells deal with a few sentences to be removed by a refactoring technique, and the appearance frequency of the code pattern—i.e., Code Smell—is low within a program, the effect of energy savings is inadequate.

3.2 Energy consuming constructs

In this paper, we want to find new ways for refactoring which can provide more energy savings by focusing on the Vetro's Energy Code Smell [30] and his experimental results. In order to develop new refactoring techniques, we observed the following two points:

(1) Verto's Energy Code Smell: The Energy Code Smell by Vetro can reduce the energy consumption through refactoring, but can conversely increase the energy consumption. Therefore, we place our focus on developing new techniques except the techniques presented by Vetro.
(2) Impacts of Execution Logics: Most of the existing refactoring techniques primarily focus on restructuring based on the interaction between functions for increasing the extensibility and understandability of source code. However, these techniques fail to show their effectiveness in terms of energy savings [20]. Therefore, we focused on the code patterns that can provide energy savings through refactoring, especially programming codes that occur frequently, and then we developed refactoring techniques for those code constructs that affect the execution logic and execution path of the program.

Based on the above considerations, we defined the seven 'Energy Consuming Constructs (ECCs)' such as Table 4. In order to derive these ECCs, we analyzed about 70 source files, written in C from http://www.planet-source [31] which in particular provides open sources. We examined closely the source codes in the viewpoints of sequence, branch, and iteration which are basic control structures of Dijkstra [32]. The results of the examination are listed in Table 3. The reason we are interested in procedural languages is that most embedded software is being written in C language.

Table 3 shows that we found a number of codes which contain math functions in preprocessing declaration and global variables in defining and using the variables through analyzing the control type of 'Sequence' for C source codes from the 'planet' website. The code pieces in the control type 'Branch' are generally well-structured, but the recursive function calls make the code structure that its execution flow difficult to understand, as observed from the open source '8 queens on a chess board'. In the control type of iteration, there existed several codes which can be removed one or two indices from loop structure. The open source 'small RSA' is an example code with a nested loop structure containing just one or two statements.

These may be phenomena that can appear in source codes as a coding skill of developers. However, those code styles were determined to be ineffective at reducing energy consumption because they can lead to unnecessary memory access, or repeatedly execute the same operation in the viewpoint of program run. Based on the results of these examinations, we defined ECCs such as Table 4.

Table 3 Examination results for C open sources from 'planet' sites

Control type	# of sources	Names of representative source files	Findings
Sequence	15	Key press event handler	General assignment statements are simple
		A prime factor program	Some function calls pass a little bit complex data type
Branch	30	Absolute recursive factorial function	Some branch paths are confusing to follow
		8 queens on a chess board	Difficult to follow the recursive call path, especially not easy to know returning point
Iteration	24	Network port scanner	A few index variables are abuse
		Lexical analyzer	Superfluous nested loop is used in simple processing
		Small RSA	

For each ECC, we briefly explained the factors that might cause unnecessary energy consumption and the considerations to handle the factors as follows. The refactoring techniques proposed to remove ECCs from the codes are explained in Sect. 4.

3.2.1 Loop Initialization

Complex data structure such as array should be initialized before it is used. The initialization of array data structure is often achieved by loop statements as follows:

```
int array[100];
int i;

for (i = 0; i < 100; i++){
    array[i] =0;
}
```

Initializing an array using a loop statement will consume a lot of energy due to the continuous execution of comparison operations with the increase or decrease operations for the indices of the array. The initialization of a cell of the array causes a jump instruction from L2 to L3 when we check the assembly codes for the above array initialization code, as follows:

Table 4 Identified energy consuming constructs

ECCs	Description
Loop Initialization	The loop that initialize an array with a specific value
Math Function	The function that has only one math formula
Passing by Structure	The function call which uses the parameter with structure data type
Tail Recursion	The codes that is called recursively
Global Variables	A loop that has multiple read/write accesses for global variable
Escape with Flag	A termination condition of loop with checking flag variable
Nested Loop	Loop with a nested structure

```
.file   "test_LoopInitialization.c"
        .text
        .align  2
        .global main
        .type   main, %function
main:
        @ args=0, pretend=0, frame=404
        @ frame_needed=1, uses_anonymous_args=0
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #404
        mov     r3, #0
        str     r3, [fp, #-16]
        b       .L2
.L3:
        ldr     r2, [fp, #-16]
        mov     r3, #-1694498816
        mov     r3, r3, asr #22
        mov     r2, r2, asl #2
        sub     r1, fp, #12
        add     r2, r2, r1
        add     r2, r2, r3
        mov     r3, #0
        str     r3, [r2, #0]
        ldr     r3, [fp, #-16]
        add     r3, r3, #1
        str     r3, [fp, #-16]
.L2:
        ldr     r3, [fp, #-16]
        cmp     r3, #99
        ble     .L3
        mov     r3, #0
        mov     r0, r3
        sub     sp, fp, #12
        ldmfd   sp, {fp, sp, pc}
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.1"
```

Due to this jump control, the array initialization using the loop statement can lead to an increase in energy consumption. Thus, it will be able to reduce energy consumption through the initialization of the array when it is declared, instead of using a loop statement.

3.2.2 Mathematics Function

If the function including an expression is declared as a separate user-defined function with its body, it generates a calling operation of the declared function. Typically, such user-defined function is sometimes frequently used during its execution, or is defined to provide the clarity or modularity of the function. The below shows the definition of ADD() function, an example that returns the value of the expression, $2 \times x + y$.

```
int ADD(int x, int y){
     return 2 * x + y;
}

int main(){
     int a = 1,b = 2,c;
     c = ADD(a,b);
return 0;
}
```

This code structure provides the advantage of high readability and functional understandability for the user. However, due to a call for the function, it is required for the operation of branches to the function. Thus, in the case of simple mathematical functions, it can refactor the code structure in the form to perform the operation without branching through a systematic code restructuring.

3.2.3 Passing by Structure

The below example code shows the declaration of a data type 'struct' and the function call using the data type as a parameter. In general, function calls can be represented in various forms according to the way of their parameter passing. It is possible to pass a parameter using the contents of structure itself (i.e., call by value) or using the address of the structure (i.e., call by reference) [33] when we use a complex data type like array or struct. We can choose and use one of the two aforementioned ways depending on the intention of the function call—i.e., the caller will have affected the result of the execution of the called function.

```
struct s {
  double a;
  double b;
  float c;
};

int sub1(struct s temp){
  return 0;
}
```

However, if we pass a parameter to the callee (i.e., called function) without any particular intention, passing the contents of a struct is inefficient in terms of energy consumption because it requires more memory usage and a copy operation of the contents. From the assembly code level, the more members of the structure data type, the more operations will be performed.

3.2.4 Tail Recursion

The recursive calling function may increase the energy consumption by the usage of the accumulated memory arising from the call history of the function. Recursive function can be identified as a method that calls itself [34]. The modification of this recursion function may reduce the energy consumption by using an iterative loop, instead of the recursive function calls.

```
int func(int a) {
    if ( a > 0 ) {

        // stattements;
        return func (a - 1);
    } else {

        // stattements;
        return 0;
    }
}
```

We can find unnecessary instructions like bl (branch command) to call a recursive function and ldr (load command) to receive return values from the assembly codes of recursion function call. This recursion has adverse effect on energy efficiency due to repetitive memory allocation and additional behaviors caused from a number of commands in the function body and the multiple calls of the recursion function itself. Therefore, this code structure should be refactored.

3.2.5 Global Variable

The loop structure like 'for' statement generally causes a lot of execution of internal statement of the loop when it entered the loop once; the block within the loop is repeatedly executed until the escape condition of the loop was satisfied. If the loop contains some global variables within its body block, a problem will arise that the global memory will be repeatedly accessed for every iteration. Contained in the box below is an example code that has the global variable 'sum' inside a 'for' statement.

```
int sum;

void main(){
    int i;
    for (i = 0; i < 10; i++){
        sum = sum + i;
    }
}
```

The use of a global variable in a loop causes the pointer movement to the global memory area (it is different from stack operations for local variables) and additional instructions to handle the pointer to the level of assembly code. Therefore, refactoring global variables to local variables is necessary in order to save energy.

3.2.6 Escape with Flag

The loop structure requires the check of break conditions to exit the loop. As shown in the code below, we sometimes add the particular comparison clause—the 'flag' variable in the below code—to check the break condition of the loop.

```
for(i=0; (i<100)&&(flag==1); i++){

    if(i > 30)
        flag = 0;
}
```

However, it requires four branches which are to check whether the condition of the variable is satisfied or not for every iteration. Even though this code style can offer the advantage of being able to clearly understand the exit condition of the loop intuitively, it consumes more energy at running times. Therefore, we can get energy savings when refactoring the code style by separating the exit conditions with the definition of the loop.

3.2.7 Nested Loop

When performing some operations using the two-dimensional array, we generally access the array elements through a double loop (nested loop) structure. In particular, when the operation of the inner loop consists of a few, simple operations, it may be a burden. The example code below shows a nested loop structure to assign all elements in the array to the value of '1'.

```
int array[m][n];

for(i = 0; i < m; i++){
   for(j = 0; j < n; j++) {
       array[i][j] = 1;
   }
}
```

Although this nested loop structure contains simple operation within its inner block, the number of executions of the simple operation is dramatically increased due to the increment operations, comparison operations, and branch operations. Therefore, we can obtain the benefits of energy savings as well as runtime performance by refactoring the loop structure into a single loop structure.

4 Refactoring techniques

In order to drive the EECs through removing the ECCs identified in Sect. 3, we proposed new refactoring techniques with consideration to energy efficiency as well as good code structure. The proposed techniques will be defined with motivation, mechanics, and an example for each technique. Definition 1, as below, represents the elements of our code-based refactoring technique.

Definition 1 (*ECC to EEC*) Let the ECC2EEC be a refactoring technique. In order to generate EECs from ECCs, the ECC2ECC is defined with 4 elements as follows;

$ECC2EEC = \langle D_r, C_r, S_r, E_r \rangle$, where

- D_r: definition of a refactoring technique
- C_r: description of the contextual motivation to apply a refactoring technique
- M_r: procedure or mechanics to refactor source code
- $E_r = \{(S_c, R_c) | S_c$ is original source code and R_c is refactored code$\}$

We proposed seven refactoring techniques that can be applied for each of the ECCs presented in Table 4. Generally, code-based refactoring techniques are intended to improve the code quality through the restructuring of existing codes. The purpose of our proposed refactoring techniques is to obtain a result in the reduction of energy consumption of refactored codes while keeping the functionality of the original source code. Thus our techniques can be accounted useful ones if the refactored code generated by our technique has reduced energy consumption in company with the ECC removal.

4.1 ECC2EEC: Eliminate Loop Initialization

[Definition] This technique is to restructure the code structure that initializes a large-sized array using the loop structure.

[Motivation] The initialization of data structure, especially array, is essentially required to use data structure in some operations. However, the comparison operation and index processing can cause overhead when the initialization of the array is performed by loop structure. Therefore, it needs a simple and efficient way to reduce overhead.

[Mechanics]
① Identify the initialization statement for array from original source code.
② Extract the name of the array from the initialization statement.
③ Find the declaration of the array from the source code
④ Add an assignment statement to initialize the array with predefined values
⑤ Delete the loop structure for initializing the array.
⑥ Compile and test to confirm the execution results.

[Example] The example code initializes the array 'array[]'. The original example code (Fig. 1a) can be modified like Fig. 1b by the procedure explained in the above Mechanics section. This modification improves energy efficiency through reducing the number of instructions as well as the readability of the code.

```
#include <stdio.h>

int main(){
   int array[100];
   int i;

   for(i = 0; i < 100; i++){
      array[i]=0;
   }
   return 0;
}
```

(a)

```
include <stdio.h>

int main(){

   int array[100]={0};

   return 0;

}
```

(b)

Fig. 1 An example of the application of the technique 'Eliminate Loop Initialization'

4.2 ECC2EEC: Inline Mathematics Function

[Definition] This technique simplifies the user-defined function to a mathematical macro function.

[Motivation] Overhand can occur when managing and using a function that was developed separately. Thus, we can remove the function, and declare a macro function using the expression within the function to reduce the overhead.

[Mechanics]
① Identify user-defined function that contains a mathematical expression only from the original code.
② Extract the name of the function and the math expression within the function.
③ Declare a macro function to perform the same operations using the extract information.

```
#include <stdio.h>
int ADD(int x, int y){
    return 2 * x + y;
}

int main(){
    int a = 1, b = 2, c;

    c = ADD(a, b);
    return 0;
}
```

(a)

```
#include <stdio.h>
#define ADD(x, y) 2*x+y

int main(){
    int a = 1, b = 2, c;

    c = ADD(a, b);
    return 0;
}
```

(b)

Fig. 2 An example of the application of the technique 'Inline Math Function'

④ Delete the user-defined function from the code.
⑤ Compile and test to confirm the execution results.

[Example] The example codes show that the user-defined function 'ADD()'in Fig. 2a can be substituted with a macro function 'ADD()' like in Fig. 2b. This is to refactor the original code for reducing the overhead of function call, by substituting the function call with the expression of the declared macro function by preprocessor.

4.3 ECC2EEC: Substitute Complex Parameter

[Definition] This technique changes the complex data type (like struct in C) to a pointer type for function parameter.

[Motivation] It requires a copy operation of the structure, dynamic memory allocation for large structures when passing a structure data type as the parameter of function call. Thus, it can use a pointer instead of the structure, to gain energy-efficient code when we want to be directly affected by the computation results from the called function.

[Mechanics] ① Find the function which passes the structure data type as the parameter of function call.

```
#include <stdio.h>
struct s{
    double a;
    double b;
    float c;
};

int func(struct s temp){
    return 0;
}

int main(){
    struct s ab;

    ab.a = 0;
    ab.b = 10;
    ab.c = 100;
    func(ab);
    return 0;
}
```

(a)

```
#include <stdio.h>
struct s{
    double a;
    double b;
    float c;
};

int func(struct s *temp){
    return 0;
}

int main(){
    struct s ab;

    ab.a = 0;
    ab.b = 10;
    ab.c = 100;
    func(&ab);
    return 0;
}
```

(b)

Fig. 3 An example of the application of the technique 'Substitute Structure-typed Parameter'

② Substitute the structure with a pointer for the parameter declaration.
③ Find all statements to call the function.
④ Substitute the name of the structure with the address of the structure for the statements.
⑤ For all function calls to pass the structure data type, perform the step (1) to (4).
⑥ Compile and test to confirm the execution results.

[Example] The code in Fig. 3a uses the structure data type 's' as the parameter of the function 'func()'. To improve energy efficiency, the structure is

```
int countDown(int n) {
    if (n == 0) {
       printf("Completed!");
       return;
    }
    printf("%d ", n);
    countDown(n - 1);
}
```

(a)

```
void countDown(int n) {
    while(n > 0) {
       printf("%d + ", n);
       n -= 1;
    }
     printf("Completed!");
}
```

(b)

Fig. 4 An example of the application of the technique 'Transform Recursion'

replaced with the address of the structure (i.e., pointer to the structure) like Fig. 3b.

4.4 ECC2EEC: Transform Recursion

[Definition] This technique refactors a recursion function to a simple iteration statement.

[Motivation] Using the recursion function can provide short and simple code structure. But it inversely causes multiple function calls and even stack overflow. Thus, the recursion function can be substituted with iterative structure.

[Mechanics] ① Identify a recursion function from the original source code.
② Extract the exit condition for the function
③ Create an iteration statement using the exit condition.
④ Substitute the recursion function with the developed iteration statement.
⑤ Compile and test the refactored code to confirm the execution results

[Example] The example code (Fig. 4) shows the substitution of the recursive function call (i.e., the function 'countDown()') with 'while' statement.

4.5 ECC2EEC: Transform Global to Local

[Definition] This technique replaces the global variable that is used within the iteration statement with local variables.

```
#define size = 100;

int sum;

void main() {
     int i;
     int a[size];

     for(i = 0; i < size; i++) {
        sum += a[i];
     }
}
```

(a)

```
#define size = 100;

int sum;

void main() {
     int i;
     int a[size];
     int sum_local;

     for(i = 0; i < size; i++) {
        sum_local += a[i];
     }
     sum = sum_local
}
```

(b)

Fig. 5 An example of the application of the technique 'Transform Global to Local'

[Motivation] Since an iteration statement repeatedly executes the same codes of its block whenever it is invoked, the shared memory access repeatedly occurs if the body includes a global variable. Therefore, after replacing the global variables as local variables within the iteration block, then assign the value of local variables to the global variable at the next point of the iteration exit. We may receive a lot of energy savings by this method of refactoring.

[Mechanics] ① Identify a global variable from the original source code.
② Identify the iteration statement which includes the global variable within its inner block.
③ Declare a local variable to replace the global variable within the iteration block.
④ Replace all global variables with the declared local variable within the block.
⑤ Add a statement to assign the final value of the local variable to the global variable at the next point of the iteration exit.
⑥ Compile and test the refactored code to confirm the execution results

[Example] The example code (Fig. 5) shows the replacement of the global variable 'sum' to the local variable 'sum_local' within the iteration block.

4.6 ECC2EEC: Use Explicit Escape

[Definition] This technique simplifies the control variable (i.e., flag variable) and its condition check to escape from the iteration statement.

[Motivation] It should be provided adequate control flow to exit the iteration. However, inappropriate control structure (for example, the control variable 'flag' and its condition check repeatedly) was used to escape form the iteration, it leads to lots of overhead due to a number of condition checks and branch operations. Therefore, we refactor the inappropriate control structure to the simple statement 'Break' to exit from the iteration.

[Mechanics]
① Check whether any control variable is used or not, instead of an index variable to control the iteration exit.
② If a control variable is used, check whether the control variable is used from the outside of the iteration, excluding the declaration part.
③ If the control variable was used from the outside, leave it at unchanged.
④ If the control variable was not used from the outside, delete the declaration of the control variable.
⑤ Remove the condition statement that uses the control variable from the condition expression to check the iteration exit.
⑥ Replace the expression that uses the control variable with a 'Break' statement.
⑦ Compile and test the refactored code to confirm the execution results

[Example] The below example code (Fig. 6) shows that the control variable 'flag' in the original code was replaced with a 'Break' statement at refactored code. This refactoring provides the benefits of simplified code structure and improved energy efficiency.

4.7 ECC2EEC: Simplify Nested Loop

[Definition] The double loop (nested loop) is commonly used to handle a two-dimensional array. However, this technique refactors the nested loop structure to a single loop if possible.

```
#include <stdio.h>
int main(){
    int i;
    int n = 25;
    int flag = 1;

    for(i=0;(i<100) && (flag==1);i++){
       if(i > n)
           flag=0;
    }
    return 0;
}
```

(a)

```
#include <stdio.h>
int main(){
    int i;
    int n = 25;

    for(i = 0; i < 100; i++){
        if(i > n)
            break;
    }
    return 0;
}
```

(b)

Fig. 6 An example of the application of the technique 'Use Explicit Escape'

[Motivation] If the operations within loop block were described with the statements using the index to access array elements in nested loop structure, this nested loop structure may be substituted with the structure using array pointer and single loop. The single loop structure generated by this refactoring technique can provide the effect of reducing a great deal of energy consumption due to the reduction in the number of iterations.

[Mechanics]
① Identify a nested loop structure from the original code.
② Check whether the operations within the loop block were composed with the statement to be processed by the array index.
③ Declare a pointer variable to access the array.
④ Substitute the inside operations of the loop with the declared pointer.
⑤ Remove the inner loop from the nested loop structure.
⑥ Compile and test the refactored code to confirm the execution results

[Example] The below example code Fig. 7) shows the refactoring for simplifying the nested loop (double 'for' statement) to a single 'for' statement.

```
#define sizeM = 100;
#define sizeN = 100;

void main() {
     int i, j;
     int arrayA[sizeM][sizeN];

     for(i = 0; i < sizeM; i++){
         for(j = 0; j < sizeN; j++){
            arrayA[i][j] = 100;
     }
}
```

(a)

```
#define sizeM = 100;
#define sizeN = 100;

void main() {
     int i;
     int arrayA[sizeM][sizeN];
     int *p = &arrayA[0][0];

     for(i = 0; i < sizeM*sizeN; i++){
         *p++ = 100;
     }
}
```

(b)

Fig. 7 An example of the application of the technique 'Simplify Nested Loop'

Table 5 Specification of our experimental environments

CPU	Intel XScale 80200, 266–733 MHz in 66 MHz
Architecture	ARM pipelined RISC
RAM	32 MByte Micron SDRAM, 100 MHz
Cross Compiler	arm-elf-gcc 4.1.1
Measurement tap	CPU core current, IO current, system peripherals

5 Experiments

5.1 Experimental environments

In order to verify the usefulness of our refactoring techniques and to check the energy efficiency of our techniques, we set up the experimental environment, as listed in Table 5. In addition, the experimental procedure is represented in Fig. 8.

The simulation tool, XEEMU [35], which is well-known for energy analysis, was used for our experiments. The tool, XEEMU is considered to be sufficient to support our experiments because it provides high accuracy for analyzing and predicting the power consumption based on ARM RISC-based platforms.

Figure 8 shows the procedure of our experiment. In order to perform the experiment, we first choose an appropriate source code to be restructured. Then, we get the refactored

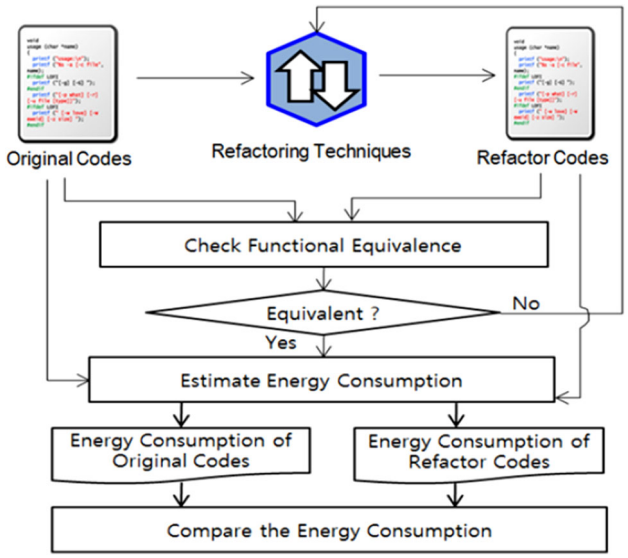


Fig. 8 The experimental procedure to confirm the energy efficiency of our refactoring techniques

code for the chosen code through applying our refactoring technique. The next work is to check the functional equivalence between the original code and the refactored code. Some test cases were used for checking equivalence. This is intended to guarantee that the original and refactored codes will provide the same execution results.

5.2 Example analysis

In order to verify the proposed technique, we first analyzed the basic code for each refactoring technique to check the possibility of reducing energy consumption. The basic codes are the example codes which were explained in Sect. 4, and they are developed by adding some statements for compiling and running to example codes. The experiment results for the basic codes, from XEEMU, are listed in Table 6.

The technique 'T2: Inline Math Function' shows low efficiency at 2% for energy reduction in this experiment. But, it provides some advantage in that it can meet the requirements of software safety and low-energy consumption as well as improve the understandability of refactored codes at the same time. In the case of the user-defined function, branching is performed to obtain the result of the operation from the function, while in the case of the inline function only the registry operation is performed without branching. Also, it can be said that the cost of context switching is saved because no branch occurs.

The techniques 'T1: Eliminate Loop Initialization' and 'T7: Simplify Nested Loop' which are related to the iterative code structure can lead to very high degree of energy reduction, the remaining 4 techniques provide on average 4% to 5% of energy efficiency. When we look at the assembly code

Table 6 Experiment results for our Refactoring Techniques

Techniques	Energy (μJ)		Difference (μJ)	Efficiency (%)
	Before	After		
T1: Eliminate Loop Initialization	5.873	4.674	1.226	20.08
T2: Inline Math Function	4.319	4.226	0.093	2.15
T3: Substitute Complex Parameter	4.522	4.341	0.181	4.00
T4: Transform Recursion	89.375	85.355	4.020	4.50
T5: Transform Global to Local	6.967	6.590	0.377	5.41
T6: Use Explicit Escape	4.745	4.556	0.189	3.98
T7: Simplify Nested Loop	945.508	753.183	192.325	20.34

for the cause analysis, the structures of two modified codes are very similar. However, in the case of direct assignment to each element of the array, the stack pointer performs an operation to point out an element of the array, while in other cases only a registry operation is performed as if it were a pointer. Therefore, the cost of handling the local stack seems to be the main cause of this difference.

The refactoring for iteration structure gives an efficiency of roughly 20% from the experimental results and the refactoring for other sequential or conditional statements has the efficiency of approximately 4.5% in consumed energy reduction. Also we can determine that energy saving efficiency will increase if the number of iterations increases depending on the nature of the target codes.

6 Case study and its analysis

6.1 Contexts of case study

In order to verify the proposed refactoring techniques, we carried out a case study for a practical program to demonstrate energy efficiency. We looked at a lot of programs to select the sample program for our experiments, and finally we selected the program as follows [36]. Also, to improve the accuracy of experimental results, the programs which receive a user input during its running were excluded. In addition, if it were necessary for some input data to execute the program, the same input data was used for the original code and the refactored code.

- Program Name and version: Lame (Ver. 3.95)
- Developer and distribution date: Mark Taylor et al., June, 2000
- Major function: Encoding/decoding of mp3 file from/to other format of media files
- Size in LOC: 12,367 LOC
- Programming Language: C language

Our target program is Lame which is a very famous open source based mp3 encoder. Following the information provided by the official site of Lame, there are more than 100 applications (including various mobile software) that are implemented using this program. In other words, we use this open source program very often in our life from desktop computers to smartphones even though we did not realize that. This program mainly provides encoding and decoding of mp3 files with various options. Because these functions are implemented with complex options and the nature of encoding and decoding (for example, the buffer as the array, branches and iterations that are used frequently and arithmetic operations) the source code of this program is suitable to verify our proposed refactoring techniques. Therefore, we used Lame for our case study.

The case study in this experiment was conducted in the same environments as the one described in Sect. 5.1 to evaluate the energy savings of refactoring techniques. At first, we analyzed the energy consumption of the 'Lame' program. After restructuring the 'Lame' program using the refactoring techniques, we generated the object code of the program using arm-elf-gcc 4.1.1 compiler. The object code estimated its energy consumption by the XEENU tool. This process was carried out by the procedure, presented in Fig. 8.

Figure 9 shows the example screens of our tool that supports our refactoring techniques.

6.2 Goals and procedure

The basic goals of performing this case study is intended to confirm the usefulness of our refactoring techniques by comparing the experiment results for energy efficiency described in Table 6 and the case study results for the source code of the practical example program. To this end, we defined the following goals:

Q1: All of our seven refactoring techniques can provide energy efficiency for the example system.
Q2: Our refactoring techniques can provide energy efficiency even when a combination of two or more techniques is used.

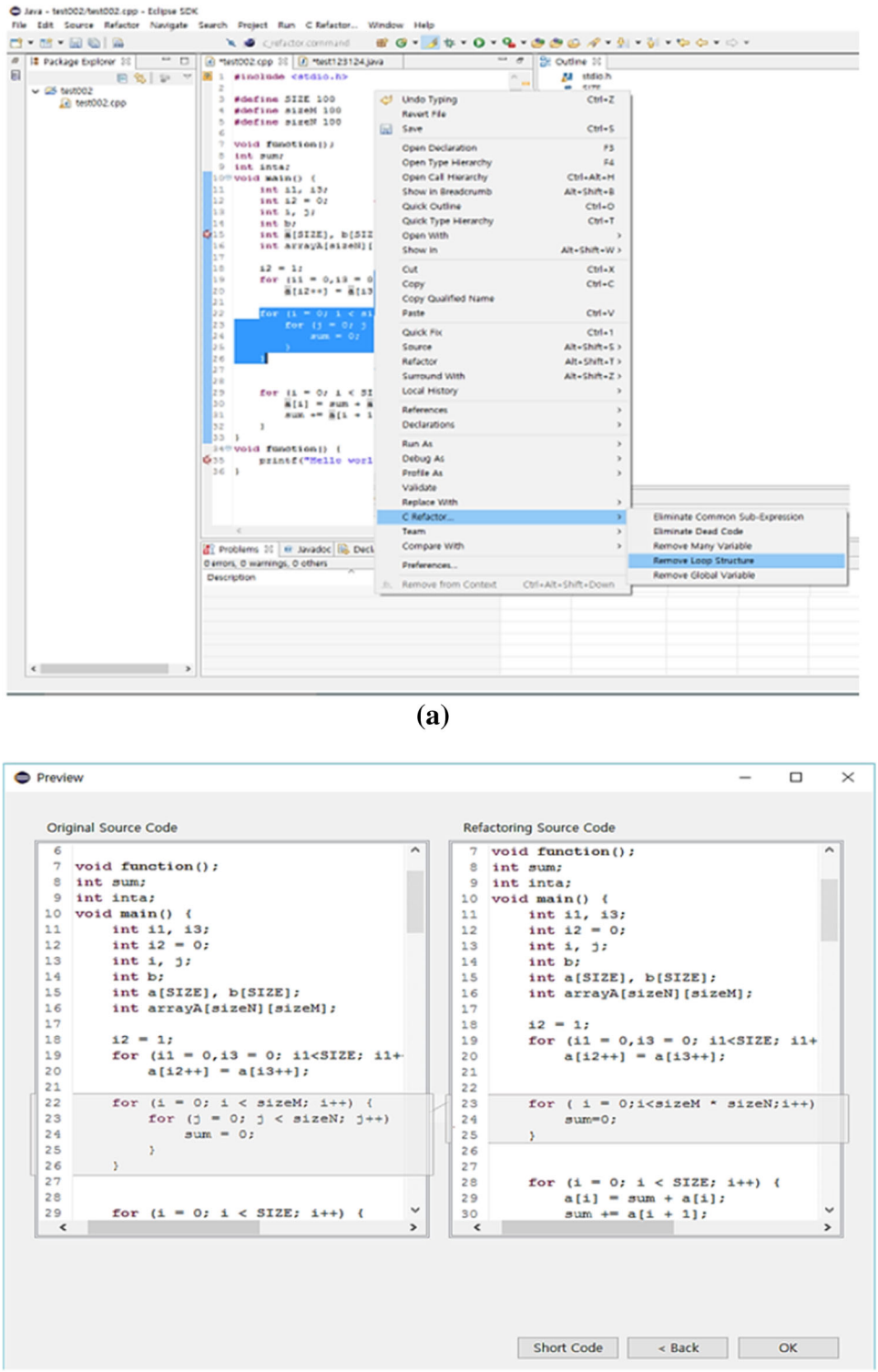


Fig. 9 Example screen shots for supporting our refactoring techniques: **a** choosing a refactoring technique for the original code, and **b** the refactoring result (right side) of the original code (left side)

In order to confirm whether the goal, Q1 can be achieved or not, we compared the energy consumption between the original code and the refactored code. Some experiments were also carried out through the combination of the refactoring techniques, such as Table 7 to confirm the achievement of the goal, Q2.

As shown in Table 7, we examined three combinations of the applicable techniques. The number of all possible combinations for the seven techniques can cause a lot of cases (i.e., ${}_7C_2$), but almost of them were excluded from the target cases because they do not affect each other by performing code restructuring.

Table 7 How to combine our techniques for refactoring of case study application

Techniques	Combination 1	Combination 2	Combination 3
T1: Eliminate Loop Initialization			
T2: Inline Math Function			X
T3: Substitute Complex Parameter	X		
T4: Transform Recursion	X		
T5: Transform Global to Local		X	X
T6: Use Explicit Escape		X	
T7: Simplify Nested Loop			

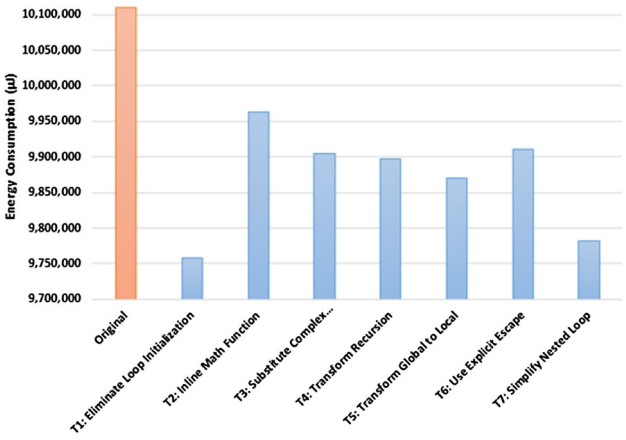


Fig. 10 Experiment results of energy consumption for the original code and the refactored codes by refactoring techniques

6.3 Results

First, we carried out the experiments using the Lame program to confirm the goal Q1. The results are shown in Fig. 10. The energy consumption for the original code of the Lame program is 10,111,345 μJ, and the results obtained from applying each of the seven refactoring techniques have shown that all techniques can drive energy savings.

Among them, the technique T1 provided the most energy-saving with similar trends as appeared in Table 6. Although the technique T2 showed the least energy savings on Table 6, it shows a relatively higher energy-saving effect than those of the techniques T5 and T6 in this case study.

This is because the T5 and T6 did not provide energy-savings due to a small number of effected codes when the T5 and T6 were applied to the original code of the Lame. More specifically, the codes applied with the T5 were executed 14 times during its run-time, but the codes applied with the T6 were executed only two times during the run-time.

Figure 10 shows that the goal Q1 is achieved. This means that the ECCs identified in Sect. 3 and the seven ECC2EECs suggested in Sect. 4 are good for the application of refactoring practices. Table 8 lists the energy consumptions for the seven techniques and their saving effects for the Lame program.

Table 8 shows the information of the number of modified code lines, energy consumptions, reduced energy consumption, and the consumption per modified one line by applying a refactoring technique.

As shown in Table 8, all the codes were modified within 10 lines through refactoring techniques. Especially, the technique T2 was applied by changing just 2 line. The fourth column of Table 8 shows the savings of energy consumption between the original code and the refactored code. In general, it was revealed that the code associated with the loop structure gives more effect in energy savings. The next column shows the efficiency of the energy saving that is calculated by dividing the energy savings with the energy consumption of original code. Some of the efficiencies are very small, even undistinguishable. However, this result is relatively better compared with the results of Vetro, as listed in Table 2.

The last column of Table 8 shows how much energy savings per modified one line were obtained from the refactoring process. It says that the technique T7 can provide the largest energy saving with the least effort for this case study.

The experiment results of the energy consumption analysis to confirm the goal Q2 are shown in Fig. 11. This figure shows the energy consumptions for the original codes and the refactored codes by the technique combinations as shown in Table 7.

From the graph of Fig. 11, we know that the goal Q2 has been achieved. The techniques T3 and T4 which belong to the positive side of energy savings could save considerable energy through the combination of the two.

The techniques T5 and T6, which showed the lowest energy consumption in Fig. 10, also give energy savings, even a small amount, in their combination. Similarly, the combination of the T2 and T5 can reduce energy consumption, but the energy savings of the combination is not equal to the mathematical addition of two energy savings for the T2 and T5 techniques.

Simply, the reductions (i.e., (A) − (B) column in Table 8) of energy consumptions for the T3 and T4 are 206,144 μJ and 213,017 μJ, respectively, and their sum is 419,161 μJ. However, the energy reduction of the combination T3 + T4 is slightly less than the sum, as 418,246 μJ. This was caused

Table 8 Results of energy consumptions and their efficiency for refactored code

Techniques	Modified LOCs	Energy consumption [(B)] (μJ)	Energy Savings [(A) − (B)] (μJ)	Efficiency of energy savings (%)	Energy reduction by modified 1 LOC (μJ)
Original code	–	(A) 10,111,345	–	–	–
T1: Eliminate Loop Initialization	9	9,759,350	351,995	3.5	39,111
T2: Inline Math Function	2	9,964,255	147,090	1.5	73,545
T3: Substitute Complex Parameter	5	9,905,201	206,144	2.0	41,229
T4: Transform Recursion	4	9,898,328	213,017	2.1	53,254
T5: Transform Global to Local	6	9,871,291	240,054	2.4	40,009
T6: Use Explicit Escape	3	9,911,332	200,013	2.0	66,671
T7: Simplify Nested Loop	4	9,783,123	328,222	3.2	82,056

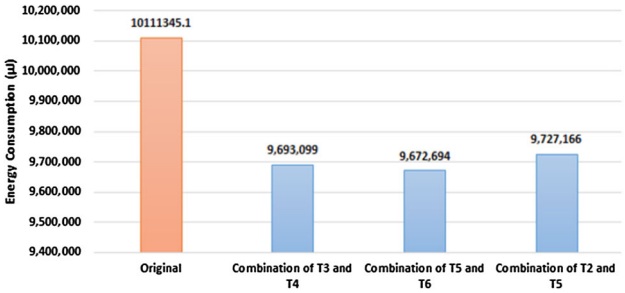


Fig. 11 Experiment results for the combinations of refactoring techniques

from the change of detail behaviors by the application of the refactoring techniques.

6.4 Threats to validity

It can be seen that our proposed techniques are effective in reducing energy consumption through various analysis experiments. However, even if the techniques may provide energy efficiency, there are a few things to be considered with the threats to validity [37], as the following:

Conclusion Validity: There can be doubt as to whether the ECC2EEC refactoring techniques proposed in this paper are best or only a solution to resolve the issue of each ECC. A new refactoring technique can be suggested for our identified ECC. However, we confirmed our refactoring techniques show the effects of energy savings through the experiments even though other techniques can be developed. In addition, we performed experiments for three combinations of our refactoring techniques. Although it is possible that more combinations, with two or three techniques, exist, programmers will not apply those combinations during their development process.

Construct Validity: We have proposed only seven techniques for the identified seven ECCs. Thus, it is possible to identify new ECCs through some more analysis. However, because the ECC was defined as a piece of code which is possible to save energy consumption, our proposed techniques for the identified seven ECCs can reduce energy consumption with their application toward the original code.

Internal Validity: The energy consumption can be slightly differed according to how the original code was modified by programmers. However, our techniques lead to reducing the energy consumption if the programmers modify their code by complying with the mechanics of our ECC2EEC, as shown in Table 6 and Fig. 7.

External Validity: We used the XEEMU tool to analyze the energy consumption for the original code and refactored code for the example program. However, there are a variety of tools for analyzing the energy consumption based on source code [38,39]. If we used a different tool, we would be able to get slightly different results in energy consumption. This is because energy consumption is dependent on the instruction set and hardware platform that covers the tools. However, the XEEMU tool was very useful in our experiments because it supports code based analysis and also provides accuracy in estimating results with approximately 1.6% average error rate compared to the actual measurement.

7 Conclusion and future work

Code refactoring is carried out with the intention to improve quality while maintaining the original functionality of the code. In this paper, we proposed seven refactoring techniques, especially considering the energy consumption characteristics of software. In order to develop the techniques, we first identified the code pattern with potential to reduce energy consumption to ECCs (Energy Consuming Constructs), and proposed techniques to transform each ECC to EEC (Energy-Efficient Construct) through structural transformation. The proposed techniques to restructure code patterns were developed in order to be used by programmers in software development, and they were demonstrated

to show the energy efficiency gained through experimental analysis.

Our experiments show that the changes in code structure have an effect on energy consumption reduction and that the reduction of average energy consumption is up to 2.4% (242,672 μJ) by applying our refactoring techniques. Although energy reduction can be different according to the technique, the reduction results give the opportunity of using a smartphone for an additional 1.5 min when we use 1 h in the real world. This is deemed sufficient to use an emergency call in a hazardous situation. Additionally, it will also be possible to reduce energy consumption further if the technique is applied to several different portions within one source code program.

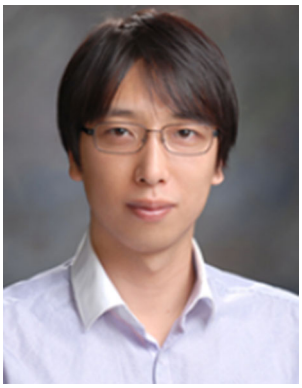
Future studies are expected to be able to extend and combine two or more techniques which are proposed in this study to enhance energy efficiency for various code patterns. This is a multi-aspects refactoring technique: the combination of refactoring techniques. In order to achieve this goal, studies need to analyze the relationship between our techniques and the effects of interaction between these techniques. Also, further work needs to extend the techniques to be applied to a variety of programming language.

Acknowledgements This research was supported by the National Research Foundation, funded by the Ministry of Education Korea (No. NRF-2014R1A1A4A01005566)

References

1. Bunse, C., Hopfner, H., Mansour, E., Roychoudhury, S.: Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. ln: The 10th IEEE International Conference on MDM, pp. 600–607 (2009)
2. Zong, Z., Nijim, M., Manzanares, A., Qin, X.: Energy efficient scheduling for parallel applications on mobile clusters. Clust. Comput. J. **11**(1), 91–113 (2008)
3. Valentini, G.L., Lassonde, W., et al.: An overview of energy efficiency techniques in cluster computing systems. Clust. Comput. J. **16**(1), 3–15 (2013)
4. Julian, N., et al.: Power consumption modeling and characterization of the T1C620. IEEE Micro **23**(5), 40–49 (2003)
5. Chang, N., Kim, K.H., Lee, H.G.: Cycle-accurate energy consumption measurement and analysis: case study of ARM7TDMI. ln: The International Symposium on Low Power Electronics and Design, pp. 185–190 (2000)
6. Tiwari, V., Malik, S., Wolfe, A.: Power analysis of embedded software: a first step towards software power minimization. IEEE Trans. VLSI Syst. **2**(4), 437–445 (1994)
7. Kiertscher, S., Zinke, J., Schnor, B.: CHERUB: power consumption aware cluster resource management. Clust. Comput. J. **16**(1), 55–63 (2013)
8. Tan, T.K., Raghunathan, A., Jha, N.K.: Energy macromodeling of embedded operating systems. ACM Trans. Embed. Comput. Syst. **4**(1), 231–254 (2005)
9. Jun, H., Xuandong, L., Guoliang, Z., Chenghua, W.: Modeling and analysis of power consumption for component-based embedded software. ln: Proceedings of the Embedded Ubiquitous Computing Workshops, pp. 795–804 (2006)
10. Hao, S., Li, D., Halfond, W.G., Govindan, R.: Estimating mobile application energy consumption using program analysis. ln: The 35th IEEE International Conference on Software Engineering, pp. 92–101 (2013)
11. Wang, Z., Xu, X., Xiong, N., Yang, L.T., Zhao, W.: Energy cost evaluation of parallel algorithms for multiprocessor systems. Clust. Comput. J. **16**(1), 77–90 (2013)
12. Kim, D.H., Hong, J.E.: ESUML-EAF: a framework to develop an energy-efficient design model for embedded software. Softw. Syst. Model. **14**, 795–812 (2015)
13. Nogueira, B., Maciel, P., Tavares, E., Andrade, E., Massa, R., Callou, G., Ferraz, R.: A formal model for performance and energy evaluation of embedded systems. EURASIP J. Embed. Syst. (2011). doi:10.1155/2011/316510
14. Jelschen, J., et al.: Towards applying reengineering services to energy-efficient applications. lm: The 16th IEEE European Conference on Software Maintenance and Reengineering (2012)
15. Brandolese, C., et al.: The impact of source code transformations on software power and energy consumption. J. Circuit Syst. Comput. **11**, 477 (2002)
16. Li, D., Hao, S., Halfond, W.G., Govindan, R.: Calculating source line level energy information for android applications. ln: The 2013 International Symposium on Software Testing and Analysis, pp. 78–89 (2013)
17. Kwon, Y.W., Tilevich, E.: Reducing the energy consumption of mobile applications behind the scenes. ln: The 29th IEEE International Conference on Software Maintenance, pp. 170–179 (2013)
18. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley, Boston (2002)
19. Pinto, G., Castor, F., Liu, Y.D.: Understanding energy behaviors of thread management constructs. ln: The ACM International Conference on Object Oriented Programming Systems Languages & Applications (2014)
20. Gottschalk, M., et al.: Removing energy code smells with reengineering services. ln: Proceedings of the GI-Jahrestagung (2012)
21. Banerjee, A., Chong, L.K., Chattopadhyay, S., Roychoudhury, A.: Detecting energy bugs and hotspots in mobile apps. ln: The 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 588–598 (2014)
22. da Silva, W.G.P., Brisolara, L., Correa, U.B., Carro, L.: Evaluation of the impact of code refactoring on embedded software efficiency. ln: Workshop de Sistemas Embarcados (2010)
23. Pérez-Castillo, R., Piattini, M.: Analyzing the harmful effect of god class refactoring on power consumption. IEEE Softw. **31**, 48–54 (2014)
24. Park, J.J., Hong, J.E., Lee, S.H.: Investigation for software power consumption of code refactoring techniques. ln: International Conference on Software Engineering & Knowledge Engineering (2014)
25. Pathak, A., Charlie Hu, Y., Zhang, M.: Bootstrapping Energy debugging on smartphones: a first look at energy bugs in mobile devices. ln: Hotnets'11, November (2011)
26. Gottschalk, M., Jelschen, J., Winter, A.: Energy-efficient code by refactoring. ln: The 15th Workshop on Software-Reengineering (2013)
27. Kimura, S., et. al.: Move code refactoring with dynamic analysis. ln: The 28th IEEE International Conference on Software Maintenance, pp. 575–578 (2012)
28. Kannangara, S.H., Wijayanake, W.M.J.I.: An empirical evaluation of impact of refactoring on internal and external measures of code quality. Int. J. Softw. Eng. Appl. **6**(1), 51–67 (2015)

29. Ouni, A., Kessentini, M., Sahraoui, H.: Search-based refactoring using recorded code changes. ln: The 17th European Conference on Software Maintenance and Reengineering, pp. 221–230 (2013)
30. Vetro, A., Ardito, L., Procaccianti, G., Morisio, M.: Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. ln: ENERGY 2013, The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, pp. 34–39 (2013)
31. Planet Source Code. https://www.planet-source-code.com/. Available at 4 January 2016
32. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliff (1976)
33. Milanova, A., Rountev, A., Ryder, B.G.: Precise call graphs for C programs with function pointers. Autom. Softw. Eng. **11**(7), 7–26 (2004)
34. Moschovakis, Y.N., van den Dries, L.: Arithmetic complexity. ACM Trans. Comput. Log. **10** (2009). doi:10.1145/1459010.1459012
35. Herczeg, Z., Schmidt, D., et al.: Eergy simulation of embedded XScale systems with XEEMU. J. Embed. Comput. **3**, 209–219 (2009)
36. Mark, T., et al.: Lame (Ver. 3.95:2000). http://lame.sourgeforge.net. Accessed 12 May 2016
37. Wohlin, C., et al.: Experimentation in Software Engineering: An Introduction, pp. 64–74. Kluwer Academic Publishers, Norwell (2000)
38. Senn, E., Laurent, J., Julien, N., Martin, E.: SoftExplorer: estimating and optimizing the power and energy consumption of a C program for DSP application. EURASIP J. Appl. Signal Process. **16**, 2641–2654 (2005)
39. Tan, T.K., Raghunathan, A., Jha, N.K.: EMSIM: an energy simulation framework for an embedded operating system. ln: International Symposium of Circuits and Systems, pp. 464–467 (2002)



Doohwan Kim is a Post.Doc researcher working for the Software Engineering Laboratory at the School of Electrical and Computer Engineering, Chungbuk National University, Korea. He received his M.S degree from Chungbuk National University, Korea, in 2009, and his Ph. D in computer science from also Chungbuk National University, Korea, in 2015. His interesting research areas are software architecture, software quality, embedded software, low-energy software development, and component-based software development.

Jang-Eui Hong is a professor of department of computer science at the Chungbuk National University since 2004. He obtained his master degree in computer engineering from the Chung-Ang University of Seoul, Korea in 1988 and his Ph. D in computer science from KAIST, Korea in 2001. He has many years of experience as a researcher and a consultant in software development and software quality improvement during working at the Agency for Defense Development and the SolutionLink Co., Korea. His research interests include model-based quality engineering, energy-efficient software development, and software safety.

Ilchul Yoon is an assistant professor in the department Computer Science at State University of New York, Korea. He received the B.S. degree from Sogang University and the M.S. degree from Korea Advanced Institute of Science and Technology. In 2010, he received the Ph.D. degree in Computer Science at University of Maryland, College Park. His research interests include collaborative software testing, component-based software engineering, software process, and high-performance computing.

Sang-Ho Lee was born in PuSan, Korea in 1953. He received the M.S. degree and Ph. D in the department of computer science, Soongsil University in February 1981 and 1989 respectively. He is currently working as a professor in the department of Computer Science, Chungbuk National University. He is currently the president of Convergence Society for Small and Medium Business. His research interests include Protocol Engineering, Mobile Network Security, and Software-define Network Architecture.