KIPS Tr. Software and Data Eng.
Vol.5, No.5 pp.209~220 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2016.5.5.209

Code Refactoring Techniques Based on Energy Bad Smells for Reducing Energy Consumption

Jae-Wuk Lee† · Doohwan Kim†† · Jang-Eui Hong†††

ABSTRACT

While the services of mobile devices like smart phone, tablet, and smart watch have been increased and varied, the software embedded into such devices has been also increased in size and functional complexity. Therefore, increasing operation time of mobile devices for serviceability became an important issue due to the limitation of battery power. Recent studies focus on the software development having efficient behavioral patterns because the energy consumption of mobile devices is caused by software behaviors which control the hardware operations. However, it is often difficult to develop the embedded software with considering energy-efficiency and behavior optimization due to the short development cycle of the mobile services in many cases. Therefore, this paper proposes the refactoring techniques for reducing energy consumption, and enables to fulfill the energy requirements during software development and maintenance. We defined energy bad smells with the code patterns that can excessively consume the energy, and our refactoring techniques are to remove these bad smells. We performed some case studies to verify the usefulness of our refactoring techniques.

Keywords : Software Behavioral Patterns, Energy Consumption, Energy Bad Smells, Code Refactoring

Energy Bad Smells 기반 소모전력 절감을 위한 코드 리팩토링 기법

이 제 욱† · 김 두 환†† · 홍 장 의†††

요 약

최근 스마트폰, 태블릿과 같은 기기의 사용량이 증가하면서, 이에 탑재되는 소프트웨어는 더욱 복잡해지고 규모가 커지고 있다. 배터리의 전력으로 구동되는 모바일 기기들은 전력 공급의 한계로 인해 운용시간을 증가시키는 것이 중요한 이슈이다. 최근에는 소프트웨어 동작이 하드웨어 구동을 통해 전력 소모를 일으킨다는 점에서, 효율적인 동작 패턴을 갖는 소프트웨어 개발에 대한 연구들이 진행되고 있다. 그러나 모바일 기기에 탑재되는 소프트웨어는 그 개발 주기가 짧은 경우가 많아 최적화와 전력 소모량을 반영하기 어려운 경우가 많다. 따라서 본 연구에서는 소모전력 절감을 위한 코드 리팩토링 기법을 제안하여, 소프트웨어 개발 및 유지보수에서 보다 용이하게 저전력 요구사항을 충족시키고자 한다. 이를 위해 전력 소모량을 감소시킬 수 있는 코드 패턴에 대하여 Energy Bad Smell을 식별하고, 이를 제거하기 위한 새로운 코드 리팩토링 기법을 제안하며, 실험을 통해 그 효용성을 검증하였다.

키워드 : 소프트웨어 행위패턴, 에너지 소모, Energy Bad Smells, 코드 리팩토링

1. 서 론

최근 스마트폰 및 태블릿과 같은 모바일 기기의 이용자들이 증가하면서, 모바일 기기를 위한 응용 프로그램들은 사용자들의 다양한 요구사항을 만족시키기 위해 더욱 복잡해지고 규모가 커지고 있으며, 결과적으로 사용자들이 모바일 기기를 이용하는 시간을 증가시키고 있다. 그러나 모바일 기기는 배터리의 전력으로 구동되므로 전력 공급에 한계성이 있다. 이러한 환경에서 모바일 기기의 성능과 기능을 유지한 채 운용시간을 늘리는 것은 중요한 이슈이다[1].

모바일 기기의 운용시간을 늘리기 위한 연구는 크게 배터리 용량증가, 저전력 하드웨어 모듈 개발, 소프트웨어 최적화의 세 가지로 분류할 수 있다[2]. 소프트웨어에 의한 소모전력 절감 연구는 크게 개발 초기 단계에서 소모 전력을 예측하는 기법들[3-5]과 개발된 코드의 검증을 통해 소모 전

※ 이 논문은 정부(교육부)의 재원으로 한국연구재단-일반연구자지원사업의 지원을 받아 수행된 연구임(No. NRF-2014R1A1A4A01005566).
† 준 회 원 : 충북대학교 컴퓨터과학과 박사과정
†† 준 회 원 : 충북대학교 컴퓨터과학과 박사후과정
††† 종신회원 : 충북대학교 컴퓨터과학과 교수
Manuscript Received : January 12, 2016
First Revision : April 11, 2016
Accepted : April 11, 2016
* Corresponding Author : Jang-Eui Hong(jehong@chungbuk.ac.kr)

력을 분석하는 연구들[6-9]로 구분할 수 있다. 전자의 경우 소프트웨어 개발 초기 단계에서부터 저전력 요구사항을 충족시키기 위한 활동들을 수행할 수 있다는 장점이 있으며, 후자의 경우 실제 구현된 코드를 검증하기 때문에 신뢰성 높은 분석 값을 얻을 수 있다는 장점이 있다. 이에 착안하여 본 연구에서는 소모전력 절감을 위한 코드 리팩토링 기법을 연구하였다.

현재 코드 리팩토링 기법의 많은 부분은 패턴화 되어 제시되고 있으며, 소프트웨어 코드 개발 및 유지 보수 단계에서 쉽게 적용 가능하다는 장점을 제공한다. 또한 리팩토링이 가능한 Code Smells을 정의하고, 특히 소프트웨어 품질에 악영향을 미치는 코드 패턴을 Bad Smell로 정의함으로써, 코드 리팩토링 기법의 적용 기준이 되고 있다[10].

소모전력 절감을 위한 코드 리팩토링 기법들 또한 연구되고 있다[11-13]. 이들 연구에서는 모바일 기기에서 불필요한 소모 전력을 유발하는 소프트웨어 동작을 Energy Bug 또는 Energy Code Smell로 정의하고, 이를 제거하기 위한 리팩토링 기법을 제안하고 있다. 그러나 분석 대상이 하드웨어 리소스 반납 관리, 디스플레이 특성에 따른 색상 배치, 광고 팝업 코드 제거 등으로 구성되어 있다. 이는 소모전력 절감 효과는 기대할 수 있으나, 운영체제의 기능 및 동작과 연계된 정밀한 분석이 요구되거나, 소프트웨어의 독창성, 수익성 등과 연관되어 있어 소프트웨어 개발자들에게 유용성이 부족할 것으로 판단된다. 또한 소프트웨어의 기능 로직에 대한 코드 리팩토링 기법과는 차이가 있으며, 적용 가능한 대상이 한정되어 있다는 단점이 있다.

따라서 본 논문에서는 프로그램 로직을 구성하는 코드를 분석하여, 소모 전력이 절감될 가능성이 있는 코드를 식별하여 Energy Bad Smell로 정의하고, 이를 해결하기 위한 소모 전력 절감 리팩토링 기법을 제안한다. 제안한 기법은 소모 전력 분석 실험을 통해 그 타당성을 검증하였다. 이를 통해 소프트웨어 개발 및 유지보수 과정에서 보다 용이하게 저전력 소프트웨어 요구사항을 충족시킬 수 있을 것으로 기대된다.

본 논문의 구성은 다음과 같다. 2장에서는 소프트웨어 소모전력 절감을 위한 연구들 및 코드 리팩토링 연구들을 소개하고, 3장에서는 에너지 소비량이 많은 코드 패턴을 식별하여 Energy Bad Smell을 정의한다. 4장에서는 Energy Bad Smell을 제거하기 위한 소모전력 절감을 위한 코드 리팩토링 기법들을 제안하며, 이에 대한 검증 실험 결과를 5장에서 기술하고, 6장에서 결론 및 향후 연구를 기술한다.

2. 관련 연구

최근의 코드 기반 소프트웨어 소모전력 절감 연구들은 코드 리팩토링 기법에 기반을 둔 사례가 증가하고 있다.

먼저 Fowler가 제시한 코드 리팩토링 기법들을 대상으로 한 연구들이 존재한다. Silva의 연구[14]는 Inline Method 기법을 반복적으로 적용하면서 성능과 전력량 변화를 분석하였다. 이 연구에서는 Inline Method 기법의 적용이 성능향상과 소모전력 감소 효과가 있음을 보였다. 이와 반대로 Rica의 연구[15]는 Extract Method 기법과 Extract Class 기법의 적용이 객체 사이의 메시지 트래픽을 증가시켜 소모 전력량을 증가 시키고 있음을 보였다. 그 외에 Park의 연구[16]는 Fowler가 제시한 코드 리팩토링 기법 전체를 대상으로 소모전력 분석을 수행하여, 각 기법들을 적용하였을 때의 전력 소모량 변화를 분석하였다. 이러한 연구들은 기존의 코드 리팩토링 기법이 가진 소프트웨어 품질 향상 기능에 더하여 전력 소모량 또한 고려할 수 있음을 보이고 있다. 그러나 기존 코드 리팩토링 기법 모두가 전력 소모량이 감소하는 것은 아니며, 오히려 증가하는 경우에 대한 명확한 해법을 제안하지 못했다는 한계가 있다.

최근에는 소스 코드로부터 전력 소모량이 큰 패턴을 식별하여 이를 해결하기 위한 다양한 코드 리팩토링 연구들이 시도되고 있다.

Pathak의 연구[17]는 모바일 기기에서 전력 소모량을 증가시키는 요인을 Energy Bug로 정의하였다. Energy Bug는 하드웨어 레벨과 어플리케이션 레벨로 분류되며, 이 중 어플리케이션에 대한 Energy Bug는 No-Sleep, Loop, Immortality로 구성된다. No-Sleep Bug는 하드웨어 리소스의 사용이 종료되는 것을 막는 것이며, Loop Bug는 시스템이 동작 결과를 얻지 못하지만 반복적으로 특정 작업을 수행하는 것을 의미한다. Immortality Bug는 종료된 어플리케이션이 다른 어플리케이션에 의해 지속적으로 재시작되는 상태를 의미한다.

Gottschalk의 연구[18]는 Application Energy Bug를 기반으로, Energy Code Smell을 정의하여 안드로이드 기반 모바일 기기의 소모전력 분석을 수행하였다. 이 연구에서는 소스코드를 분석하여 Energy Code Smell을 식별하고, 해당 코드를 재구성하는 과정을 통한 Energy Refactoring 방법을 제안하였다. 그러나 정의된 5개의 Energy Code Smell 중 “Third-Party Advertisement”는 어플리케이션 내부에서 광고 팝업 코드를 제거하는 것으로써, 실제 모바일 어플리케이션 개발자들에게 효용성이 적을 것으로 보인다. 또한 “Backlight” Smell의 경우, 모바일 기기 액정의 종류와 색상 표현 방법에 따른 소모 전력이 서로 다르게 나타난다는 것에 착안하고 있으나, 사용자 관점에서 어플리케이션의 시각 디자인이 중요한 요소임을 감안하면 효용성이 크지 않을 것으로 판단된다.

3. 에너지 코드 유형

3.1 Energy Code Smell

M. Fowler는 Code Smell을 “시스템 내부에 더 심각한 문제가 있다는 일종의 지표”로 정의하였으며, 일반적인 코드 리팩토링에서 주목하는 대표적인 Code Smell은 Table 1에 제시하였다[10].

Table 1. Representative Code Smells by M. Fowler [10]

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

Code Smell은 코드의 길이가 지나치게 길어 프로그램 성능이 저하되거나 오류를 내재할 가능성이 높은 코드들이다. Code Smell을 분석하여 실제로 소프트웨어 품질을 떨어뜨리는 요인으로 판별될 경우 Bad Smell로 식별하여 코드 리팩토링 기법을 적용한다. 일반적으로 Duplicated Code나 Long Method와 같은 Code Smells은 소스 코드에 대한 리팩토링 기법의 적용을 통해 코드의 품질을 향상시킬 수 있다. 그러나 Long Parameter List의 경우 객체간의 종속성이 생기지 않아야 한다는 제약조건이 있다.

이와 유사하게 Vetro의 연구[19]에서는 수정을 통해 소모 전력을 감소시킬 수 있는 가능성이 높은 코드 패턴들을 Energy Code Smell이라고 정의하였다. 각 Energy Code Smell에 대하여 코드 리팩토링 기법을 제안하고 소모전력 분석을 통해 Table 2와 같은 결과를 도출하였다. 실험 결과 5가지의 Code Smells이 리팩토링을 통해 소모전력 절감 효과가 있음을 보였다.

Table 2. Energy Consumption for Energy Code Smells [19]

Energy Code Smells	Description	Impact (%)
Parameter By Value	Passing parameters to the 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

소모전력 절감 효과를 얻은 Code Smells은 코드 실행 흐름의 오류로 인하여 불필요한 동작을 유발한 경우임을 알 수 있다. 그러나 해당 코드 패턴의 수정을 통한 소모전력 절감 효과가 1% 미만으로 그 효과가 적다는 한계가 있다. 또한 Energy Code Smells은 수정되는 코드가 적으며, 프로그램에서 해당 코드의 사용빈도가 적어 소모전력 절감 효과가 미비한 것으로 판단된다.

3.2 Energy Bad Smell

본 논문에서는 Vetro가 정의한 Energy Code Smell 및 실험 결과에 착안하여 보다 에너지 절감 효과를 제공할 수 있는 새로운 리팩토링 방법을 발견하고자 하였다. 이와 같은 새로운 리팩토링 기법의 개발을 위하여 다음과 같은 두 가지 사항에 주목하였다.

(1) Verto's Energy Code Smell : Vetro가 제시하는 Energy Code Smell은 리팩토링을 통하여 에너지의 소모를 줄일 수도 있지만, 반대로 에너지의 소모가 증가되기도 하였다. 따라서 제시한 모든 기법이 에너지 절감 효과를 제공한다고 단언할 수 없다. 따라서 본 연구에서는 코드 리팩토링을 통하여 에너지 절감 효과를 제공하는 소스 코드의 패턴을 Energy Bad Smell로 정의하였다. 따라서 Energy Bad Smell을 갖는 소스 코드들은 제안하는 리팩토링 기법의 적용에 의해 에너지 효율성이 증가할 것이라고 판단한다.

(2) Impacts of Execution Logics : 소스 코드의 확장성, 이해성 등을 높이기 위한 방법으로 함수 시그너춰(Signature)를 기반으로 하는 리팩토링 기법이 있다. 그러나 이러한 기법들은 에너지 절감 측면에서 그 효용성을 제공하지 못하는 경우가 대부분이다[16]. 따라서 본 연구에서는 리팩토링을 통해 소모 절감 효과를 제공할 수 있는 코드 패턴에 주목하였다. 즉 프로그램의 실행 로직 및 실행 경로에 영향을 미치며, 또한 일반적인 코드 작성에서 빈번히 발생할 수 있는 Code Constructs을 대상으로 리팩토링 기법을 개발하였다.

위와 같은 고려사항을 기반으로 하여 Table 4와 같은 5가지의 Energy Bad Smells을 정의하였다. 특히 이러한 5가지의 Bad Smells을 도출하기 위하여 Dijkstra가 제시하는 제어구조의 기본 패턴인 순차(Sequence), 분기(Branch), 그리고 반복(Iteration)을 기준으로 오픈 소스를 제공하는 www.planet-source-code.com 사이트[20]로부터 C로 작성된 70여개의 소스코드를 분석하여 Table 3과 같은 결과를 발견하였다.

Table 3으로부터 'plate' 오픈 소스 사이트로부터 분석한 C 소스 코드에 대하여 순차 제어 구조를 갖는 코드 부분에서는 일반적으로 관계 혹은 논리 연산자를 포함하는 수식이 복잡하게 표현되는 경우가 다수 발견되었으며, 분기 제어 구조를 갖는 코드 부분은 대체적으로 체계적이나, '8 queens on a chess board' 오픈 소스에 나타난 것처럼 재귀적 호출

을 사용하는 경우, 이해가 어렵고 복잡한 실행 구조를 보이고 있었다. 반복의 구조에서는 한 두 개의 불필요한 루프 인덱스(index)를 제거할 수 있는 사례들이 보였으며, 아주 단순한 예를 들면 한 두 개의 문장만으로 구성된 이중 루프 구조도 오픈 소스 'small RSA'에 나타냈다.

Table 3. Inspection results for C open sources from 'planet' open sources site

Control Type	# of Sources	Names of Representative Source Codes	Findings
Sequence	15	- key press event handler - a prime factor program	- General assignment statements are simple. - Some expressions are a little bit complex.
Branch	30	- absolute recursive factorial function - 8 queens on a chess board	- Some branch paths are confusing to follow. - Difficult to follow the recursive call path, especially not easy to know returning point.
Iteration	24	- network port scanner - lexical analyzer - small RSA	- A few index variables are abuse. - Superfluous nested loop is used in simple processing

이러한 것들은 기본적으로 개발자의 코딩 스킬에 따라 나타나는 현상일 수 있으나, 실행의 관점에서 불필요한 메모리 접근을 유발하거나 동일 연산을 반복적으로 수행하는 결과를 초래할 수 있어 에너지 절감측면에서 효율적이지 못한 코딩 패턴으로 판단하였다. 이러한 코드 분석 결과를 토대로 Table 4와 같은 Energy Bad Smell을 정의하였다.

Table 4. Definition of Energy Bad Smells

Energy Bad Smells	Description
Complex Expressions	The codes that contain highly complex expressions
Common Sub-expressions	The multiple codes that contain the same operation
Tail Recursion	The codes that is called recursively
Loop Structure	The loop that can be optimized with restructuring
Dead Code	The codes that will not be run at any condition

3.2.1 Complex Expressions

수식을 표현하는 연산문이 Fig. 1과 같이 지나치게 복잡하게 작성되는 경우, 연산을 위한 명령어가 많이 생성되어 소모 전력이 불필요하게 증가할 수 있다. 특히 이러한 문장이 조건문에 사용되는 경우, 가독성이 떨어짐으로 인해 오류가 내재될 수 있다.

```
if((A == B) && (B == C)||(A + D >= 0)){
    //statements
}
else if(!(A == B) && (B == C)||(A + D < 0)){
    //statements
}
else{
    //statements
}
```

Fig. 1. An Example Pattern of the Complex Expressions

이러한 코드 패턴은 복잡한 연산식이나 if-else 코드를 대상으로 식별할 수 있으며, 실행의 측면에서 분석한 어셈블리 코드를 살펴보면, if-else 체인의 모든 조건문을 수행함에 있어서 각 조건식(A==B와 같은)의 결과를 얻기 위해 분기를 수행하고, 이에 대한 결과 값을 레지스터에 저장한 후, 모든 조건식의 결과를 얻은 후에야 각 결과에 대한 논리 연산을 순차적으로 실행하였다. if-else 체인을 구성하고 있는 Fig. 1의 각 조건문에서는 (A==B), (B==C), 그리고 (A+D<0)과 같은 연산이 반복적으로 사용되고 있는 점에서 매번 비교할 때마다 이미 결과가 구해진 조건식에 대한 결과를 다시 구해야 하는 불필요한 동작을 수행하고 있음을 알 수 있다. 따라서 이러한 현상으로 인하여 에너지 소모가 증가하게 될 것이고, 이들은 조건문을 간소화하거나 코드 구조를 재구성하는 일반적인 리팩토링 방법에 의해 해결될 수 있다.

3.2.2 Common Sub-expressions

연산식이 포함된 연속된 코드에서 동일한 연산식이 나타나는 경우, 중복된 동작이 발생한다. 특히 반복문 내부에 포함된 경우 중복 수행 횟수가 더 증가하여 전력 소모를 증가시킬 수 있다.

```
for(//condition)
{
  ...
  a = x + y + 1;
  b = (x + y) * z;
  ...
}
```

Fig. 2. An Example Pattern of the Common Sub-expressions

이러한 코드 패턴은 수식이 포함된 연산문이 연속적으로 배치되었을 때 발생할 수 있다. 동일한 연산식을 포함하고 있더라도, 연산문이 먼 위치에 배치된 경우 변수의 값이 달라질 수 있음을 주의해야 한다. 동일 연산의 반복 코드에 대한 어셈블리 코드의 분석에 따라, 주어진 반복문의 내부를 나타내는 서브루틴에서 Fig. 2에서 나타내는 바와 같이 간단한 구조를 가진 연산이라도 하나의 연산에 대하여 두 개의 명령어가 추가되는 것을 확인할 수 있었다. Fig. 2의 경우 해당 연산이 2회 반복되므로 총 4개의 명령어가 추가되었으며, 반복한 횟수만큼 전력이 증대할 수 있음을 알 수 있다. 따라서 중복된 코드를 추출하여 연산 결과를 임시 변수에 저장한 뒤, 기존 코드를 임시 변수로 대체함으로써 에너지 소모를 줄일 수 있을 것으로 기대한다.

3.2.3 Tail Recursion

재귀 호출 함수는 호출 이력이 누적되어 메모리를 사용함으로써 소모 전력을 증가시킬 수 있다. 재귀 호출 함수는 자기 자신을 호출하는 코드를 내재하고 있는 메서드로 식별될 수 있다. 이에 대한 수정은 재귀문을 반복문으로 변경하여 소모 전력을 절감할 수 있다.

```
int func(int a){

    if ( a > 0 ){
        //statements;
        return func(a - 1);
    } else {
        //statements;
        return 0;
    }
}
```

Fig. 3. An Example Pattern of the Tail Recursion

재귀 호출에 대한 어셈블리 코드의 분석에 의해 재귀 호출이 발생하는 경우, 재귀적 함수를 호출하기 위한 bl(분기) 명령어와 반환 값을 받기 위한 ldr(load) 등의 불필요한 명령어가 다수 추가됨을 알 수 있었다. 함수 내부에서 추가되는 명령어의 개수뿐만 아니라 해당 함수 자체가 재귀 조건에 따라 여러 차례 호출되기 때문에 이에 따른 메모리 할당과 같은 추가적인 행위가 반복적으로 생성되어 전력 소모에 대해 악영향을 미친다. 따라서 이러한 부분은 리팩토링 되어야 하는 부분이다.

3.2.4 Loop Structure

반복문은 한번 호출될 때마다 수행 횟수가 많다는 특징이 있다. 반복문 내부의 코드 블록이 반복 횟수만큼, 반복문의 종료 시점을 결정하기 위한 조건문 검사가 매 반복마다 수행된다. 따라서 반복문 내부에 불필요한 코드가 존재하는 경우 전력 소모량이 누적되어 나타날 수 있다. 반복문에서 에너지 소모를 증가시키는 요인은 Table 5와 같이 식별할 수 있다.

Table 5. Energy Bad Smells of Loop Structure

Energy Consuming factors	Description
Induction Variables	Many increment variables within loop structure
Nested Loop	Loop with a nested structure
Global Variables	A loop that has multiple read/write accesses for global variable

Table 5와 같이 반복문에 나타나는 Bad Smells은 앞서 설명한 것과 같이 반복문에 과도한 인덱스 변수의 사용(Induction Variables), 단순한 연산에 대한 이중 루프(Nested Loop), 그리고 루프내에서의 광역 변수의 사용 등이다. 불필요한 변수를 제거하는 것과 이중 루프를 단일 루프 구조로 변환함으로써, 코드의 단순화 및 에너지 절감 효과를 얻을 수 있다. 광역 변수에 대한 루프내에서의 사용은 어셈블리 코드 수준에서 광역 변수 영역으로 포인터가 이동하고(일반적인 지역 변수를 다루기 위한 스택 연산과는 다름), 이를 핸들링하기 위한 명령어가 추가되는 현상을 유발한다. 따라서 에너지 절감을 위하여 광역 변수에 대한 지역 변수로의 리팩토링이 필요하다.

3.2.5 Dead Code

Dead Code는 어떠한 경로로도 실행되지 않는 코드이지만 메모리에 적재됨으로써 전력소모를 유발한다. 이러한 코드 패턴은 주로 메서드 단위로, 혹은 전처리과정에서 포함되는 라이브러리 파일로 존재하며, 소스코드에 대한 정적 분석을 통해 검출할 수 있다. 검출된 Dead Code는 제거함으로써 메모리 적재에 의한 소모 전력을 절감할 수 있다.

Dead Code가 발생하는 경우는 여러 가지 원인이 있을 수 있다. 일반적으로는 분기 조건문의 연산에서 특정 코드 블록으로의 점프를 위한 연산 결과가 발생하지 않는 경우를 고려할 수 있으나, 이러한 부분은 테스트를 통해 쉽게 찾아질 수 있는 결함이다. 그러나 사용 중인 코드의 유지보수 과정에서 기존 코드에 대한 과도한 수정, 또는 기존 동작을 대체하는 과정에서 불필요한 함수 등을 삭제하지 않음으로써 Dead Code가 의도하지 않게 생성될 수 있다.

4. 소모전력 절감 리팩토링 기법

3장에서 식별한 Energy Bad Smells을 제거하기 위하여 각 Bad Smell에 대한 코드 리팩토링 기법들을 제안하고, 각 기법들에 대하여 기법 적용의 동기와 적용 방법, 그리고 적용 예제를 제시한다.

Complex Expression -> Simplify Expression 기법
Common Sub-Expression -> Eliminate Common Sub-Expression 기법
Tail Recursion -> Transform Recursion 기법
Loop Structure -> Transform Loop Structure 기법
Dead Code -> Eliminate Dead Code 기법

코드 기반 리팩토링 기법은 기존의 개발된 코드들에 대한 재구성 작업을 통하여 코드의 품질 특성을 향상시키고자 하는데 그 의도가 있다. 본 연구에서 제시하는 에너지를 고려한 코드 리팩토링 기법은 소스코드가 갖는 Bad Smell을 제거하기 위하여 해당 리팩토링 기법을 적용하고, 이를 통해 궁극적으로 코드의 기능을 유지하면서 에너지 소모를 줄이는 결과를 얻는 것이다. 따라서 제안하는 리팩토링 과정을 통해 생성된 Refactor Code의 에너지 소모가 절감된다면 해당 기법이 에너지를 고려한 리팩토링 기법으로써 유용하다고 판단할 수 있다.

4.1 Simplify Expression

[Definition] 이 패턴은 코드의 조건식 및 연산 수식 등에 나타나는 복잡한 표현식을 재구성하여 단순한 표현식들로 기술한다.

[Motivation] 문장을 구성하는 코드가 지나치게 많은 연산을 포함하는 경우, 코드에 대한 가독성 및 이해성이 떨어져 오류를 내재할 가능성이 있으며, 복잡한 연산을 수행하기 위한 명령어가 많이 생성되어 전력 소모량 역시 증가할 수 있다. 이러한 패턴은 복잡한 수식을 구현한 코드나, 조건 검사 변수를 많이 포함하는 조건문의 경우 발생할 수 있다.

[Mechanics]

① 소스코드로부터 Complex Expression을 식별, 선택한다.
② 선택한 표현식에 상수간의 연산이 존재하는 경우 연산의 결과 값으로 대체한다.
③ 다수의 논리연산이 포함된 경우, 논리식 간소화 방법을 이용하여 정리한다.
④ 다수의 Complex Expression이 중복적으로 존재하는 경우 공통부분 추출한다.
⑤ 추출된 공통부분의 연산 결과를 포함하는 새로운 문장을 생성한다.
⑥ 생성된 문장을 표현식이 첫 번째로 출현하는 문장 앞에 삽입한다.
⑦ 공통으로 나타난 모든 표현식을 삽입한 문장의 결과 값으로 대체한다.
⑧ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 4)는 스코어 계산을 위한 if 조건식문의 표현을 간단하게 재구조화함으로써, 가독성을 높이는 한편, 명령어 수의 절감을 통한 에너지 효율성을 향상시킨다.

```
void scoreCalculate(StdInfo stdInfo){
    int calculatedScore = 0;

    if(stdInfo.grade == 'A' ||
        (stdInfo.position = undergraduate &&
            stdInfo.attend != FAIL && stdInfo.report == 'A'))
    {
        calculatedScore = (stdInfo.score + 20) * 3 + 10;
    }else if(stdInfo.grade == 'B' ||
        (stdInfo.position = undergraduate &&
            stdInfo.attend != FAIL && stdInfo.report == 'B'))
    {
        calculatedScore = (stdInfo.score + 20) * 2 + 10;
    }else if(stdInfo.grade == 'C' ||
        (stdInfo.position = undergraduate &&
            stdInfo.attend != FAIL && stdInfo.report == NONE))
    {
        calculatedScore = (stdInfo.score + 20) * 1 + 10;
    }else{
        calculatedScore = (stdInfo.score + 20) * 1;
    }
}
```

```
void scoreCalculate(StdInfo stdInfo){
    int calculatedScore = 0;

    int score = stdInfo.score + 20;
    bool isPosAtt = FALSE;

    if(stdInfo.position = undergraduate && stdInfo.attend != FAIL){
        isPosAtt = TRUE;
    }

    if(stdInfo.grade == 'A' || (isPosAtt && stdInfo.report == 'A'))
    {
        calculatedScore = score * 3 + 10;
    }else if(stdInfo.grade == 'B' || (isPosAtt && stdInfo.report == 'A'))
    {
        calculatedScore = score * 2 + 10;
    }else if(stdInfo.grade == 'C' || (isPosAtt && stdInfo.report == 'B'))
    {
        calculatedScore = score + 10;
    }else
    {
        calculatedScore = score;
    }
}
```

Fig. 4. Applying Example of Simplify Expression Technique

4.2 Eliminate Common Sub-Expression

[Definition] 반복되는 부분 연산식을 사전 정의하고, 이를 이용하여 표현식을 간소화 한다.

[Motivation] 특정 수식 패턴이 반복적으로 등장하며, 그 결과 값이 변하지 않는 경우, 수식의 결과 값을 미리 계산하여 대체한다. 특히 반복문 내부에 Common Sub-Expression이 존재하는 경우, 반복 횟수만큼 소모전력 절감 효과가 누적하여 나타날 수 있다.

[Mechanics]

① 소스코드로부터 Common Sub-Expression을 식별한다.
② 각 Common Sub-Expression에 속한 변수들의 값이 변경되는 지점을 식별한다.
③ 변수 값이 변경되지 않는 구간 내에 속하는 Common Sub-Expression들을 식별한다.
④ 구간 내의 Common Sub-Expression의 연산 결과를 저장할 변수를 선언한다.
⑤ 변수에 Common Sub-Expression의 결과를 저장하는 새로운 문장을 생성한다.
⑥ 생성된 문장을 최초의 Common Sub-Expression이 발견된 앞에 삽입한다.
⑦ 새롭게 선언한 변수로 Common Sub-Expression을 대체한다.
⑧ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 5) 코드는 수식에서 “x*y”의 반복 출현을 제거하기 위해 재구조화 되었다.

```
void function(){
    //statements

    a = (x * y)/2;
    b = x * y + 30;

    //statements
}
```

```
void function(){
    //statements

    tmp = x * y
    a = tmp/2;
    b = tmp + 30;

    //statements
}
```

Fig. 5. Applying Example of Eliminate Common Sub-Expression Technique

4.3 Transform Recursion

[Definition] 재귀함수 구조를 반복문 등의 구조로 재구조화 한다.

[Motivation] 재귀함수는 알고리즘을 직관적으로 구현하고, 코드가 간결해진다는 장점이 있다. 그러나 함수 호출 비용이 높고, Stack Overflow를 유발할 수 있다는 단점

이 있다. 따라서 동일한 기능을 수행할 수 있는 반복문으로 변환한다.

[Mechanics]

① 소스코드로부터 재귀함수를 식별한다.
② 재귀함수의 반복 횟수 또는 종료 조건을 식별한다.
③ 식별된 종료 조건을 조건으로 하는 반복문으로 생성한다.
④ 생성한 반복문을 이용하여 재귀함수를 대체한다.
⑤ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래 예제(Fig. 6)는 함수 “countDown”의 재귀호출을 반복문 구조로 변경한 것이다.

```
int countDown(int n){
    if (n == 0) {
        printf("Completed!");
        return;
    }

    printf("%d ", n);
    countDown(n - 1);
}
```

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

    printf("Completed!");
}
```

Fig. 6. Applying Example of Transform Recursion Technique

4.4 Transform Loop Structure (TLS)

루프 구조에 대한 리팩토링을 수행하기 위한 기법에 대해서는 Table 5에서 제시한 것과 같이 3가지의 기법으로 구체화할 수 있다.

4.4.1 TLS: Reduce Increment Variable

[Definition] 반복문 구조에서 반복 횟수를 제어하거나 배열 원소 등의 접근을 위한 인덱스 변수가 사용될 때, 불필요하게 사용되는 인덱스 변수를 제거한다.

[Motivation] 반복문에서 반복 횟수를 제어하는 인덱스 변수와 동일한 값의 변화 패턴을 갖는 첨자형 인덱스 변수가 사용될 때, 그 첨자형 변수는 루프 제어 인덱스 변수를 이용하여 대체될 수 있다. 이는 선언 및 사용되는 변수의 수를 줄임으로써, 에너지 절감 효과를 얻을 수 있다.

[Mechanics]

① 소스코드로부터 반복문 내부에 인덱스 변수를 사용하는 부분을 식별한다.
② 인덱스 변수들에 대한 값의 변화 패턴을 확인한다.
③ 값의 변화 패턴이 동일한 인덱스 변수 중에서 하나의 변수만 선택한다.
④ 선택한 인덱스 변수를 제외하고 나머지 인덱스 변수를 삭제한다.
⑤ 삭제된 변수가 사용되는 문장 부분에 선택한 인덱스 변수로 대체한다.
⑥ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 7)는 for문의 조건식에 포함된 다수의 인덱스를 단일의 인덱스에 의해 처리될 수 있도록 변경하였다.

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

int sum;

void main(){
     int i1, i2, i3;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];

     for (i1=0, i2=0, i3=0; i1 < SIZE; i1++){
         a[i2++] = b[i3++];
     }

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

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

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

int sum;

void main(){
     int i1;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];

     for (i1 = 0; i1 < SIZE; i1++){
         a[i1] = b[i1];
     }

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

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

Fig. 7. Applying Example of TLS: Reduce Increment Variable Technique

4.4.2 TLS: Simplify Nested Loop

[Definition] 반복문 구조에서 Nested Loop에 의한 이중 루프를 갖는 구조는 일반적이다. 그러나 이중 루프가 단일 루프로 재구조화할 수 있다면 이들을 리팩토링한다.

[Motivation] 이중 루프를 갖는 Nested Loop 구조에서 반복문 블록 내부의 연산이 배열에 접근하는 인덱스를 사용하는 문장들로 구성되었을 때, 이들은 배열 포인터와 단일 루프에 의거하여 치환가능하다. 이러한 루프 단일화는 반복 횟수의 감소와 함께 많은 소모 전력을 절감하는 효과를 제공한다.

[Mechanics]

① 소스코드로부터 Nested Loop 구조를 식별한다.
② Nested Loop 블록 내부의 연산들이 배열 인덱스에 의해 처리되는 문장들로 구성되어 있는지 점검한다.
③ 배열을 접근하기 위한 포인터 변수를 선언한다.
④ 선언된 포인터 변수를 이용하여 반복문 블록내부의 연산을 대체한다.
⑤ Nested Loop 구조의 안쪽 루프를 삭제한다.
⑥ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 8)는 Nested Loop로 구조화된 for문에 대하여 단일의 for문으로 단순화하여 재구조화 하였다.

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

int sum;

void main(){
     int i1;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];

     for (i1 = 0; i1 < SIZE; i1++){
         a[i1] = b[i1];
     }

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

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

⇩

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

int sum;

void main(){
     int i1;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];
     int *p = &arrayA[0][0];

     for (i1 = 0; i1 < SIZE; i1++){
         a[i1] = b[i1];
     }

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

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

Fig. 8. Applying Example of TLS: Simplify Nested Loop Technique

4.4.3 TLS: Transform Global to Local

[Definition] 반복문은 반복문 내부 블록에 정의된 동일한 연산을 반복적으로 수행하는 구조이다. 따라서 내부 블록의 연산에서 광역(Global) 변수를 사용하고 있다면, 이를 지역(Local) 변수로 대체한다.

[Motivation] 반복문은 호출될 때마다 같은 코드를 반복적으로 실행되기 때문에, 반복문내에 포함된 광역 변수를 사용하기 위하여 반복적인 공유 메모리 접근이 발생한다. 따라서 반복문 블록 내의 광역변수를 지역변수로 대체한 후, 루프가 종료되는 시점에서 지역변수의 값을 광역변수로 배정한다면 많은 소모전력 절감효과가 나타날 수 있다.

[Mechanics]

① 소스코드로부터 선언된 광역 변수를 확인한다.
② 확인된 광역 변수가 반복문의 블록내에 사용되는 부분을 식별한다.
③ 함수에 광역 변수를 대체할 지역변수를 선언한다.
④ 지역변수를 이용하여 광역변수를 대체시킨다.
⑤ 반복문이 종료되는 블록의 바로 다음에 지역변수의 값을 광역변수로 배정하는 연산식을 추가한다.
⑥ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 9)는 for문의 조건식에 포함된 인덱스의 초기화를 반복문의 밖에서 정의하여 사용하는 예이다.

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

int sum;

void main(){
     int i1;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];
     int *p = &arrayA[0][0];

     for (i1 = 0; i1 < SIZE; i1++){
         a[i1] = b[i1];
     }

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

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

⇩

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

int sum;

void main(){
     int i1;
     int i, j;

     int a[SIZE], b[SIZE];
     int arrayA[sizeM][sizeN];
     int *p = &arrayA[0][0];

     int sum_local;

     for (i1 = 0; i1 < SIZE; i1++){
         a[i1] = b[i1];
     }

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

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

Fig. 9. Applying Example of TLS: Transform Global to Local Technique

4.5 Eliminate Dead Code

[Definition] 코드에 나타나는 절대 실행 불가능한 부분을 제거하기 위해 재구조화 한다.

[Motivation] 프로그램이 실행되는 동안 절대 실행되지 않는 코드더라도 메모리에 적재되어 전력 소모를 일으킬 수 있다. 이러한 Dead Code의 패턴은 절대 실행될 수 없는 Unreachable Code와 프로그램 실행 흐름에 전혀 영향을 주지 않는 Dead Store 패턴을 포함한다.

[Mechanics]

① 소스코드로부터 선언 및 초기화 후 사용되지 않는 변수를 식별한다.
② 식별된 변수를 제거한다.
③ 소스코드의 Control Flow 분석을 수행한다.
④ 호출되지 않는 Unreachable 메소드를 찾는다.
⑤ 찾아진 Unreachable 메소드를 제거한다.
⑥ 메소드의 최종 return문 앞에서 값을 저장하지만 사용하지 않는 지역변수의 배정문을 삭제한다.
⑦ 메서드의 최종 return()문 다음에 존재하는 Unreachable Code를 제거한다.
⑧ 컴파일 및 테스트를 통해 실행 결과를 확인한다.

[Example] 아래의 예제(Fig. 10)는 코드의 재사용 과정에서 나타나는 상황으로써, 사용되지 않는 변수 b와 변수 c에 값 30을 배정하는 코드를 제거한 것이다.

```
int fuction1(void){
    //statements
    return 0;
}

void function2(int a){
    //statements
}

int main(int){
    int a = 10;
    int b = 25;
    int c;

    fuction1();

    c = a << 2;
    return c;

    c = 30;
    return 0;
}
```

```
int fuction1(void){
    //statements
    return 0;
}

int main(int){
    int a = 10;
    int c;

    fuction1();

    c = a << 2;
    return c;
}
```

Fig. 10. Applying Example of Eliminate Dead Code Technique

5. 제안한 리팩토링 기법 검증

5.1 리팩토링 기법 실험 환경

4장에서 제안한 에너지 절감을 위한 리팩토링 기법의 효용성을 검증하기 위한 실험 환경은 Table 6, 그리고 검증을 위한 실험 절차는 Fig. 11과 같다.

본 논문에서 사용한 XEEMU[21]는 소모 전력을 분석하기 위해 알려진 시뮬레이션 도구로써, ARM RISC 기반의 플랫폼에 대한 소모 전력 예측의 높은 정확도를 갖는다. 따라서 제안한 기법의 효용성을 검증하는데 충분할 것으로 판단된다.

Table 6. Target Platform of XEEMU

CPU	Intel XScale 80200, 266-733 MHz in 66MHz
Architecture	ARM pipelined RISC
RAM	32 MByte Micron SDRAM, 100MHz
X-Compiler	arm-elf-g++ 4.1.1
Measurement tap	CPU core current, IO current, System peripherals

Fig. 11에 설명한 것처럼 본 실험을 수행하기 위해서 우리는 먼저 Original Code와 Refacor Code에 대한 Functional Equivalence를 점검하였다. 이는 Refactor Code가 Original Code와 동일한 실행 결과를 제공함을 보장받기 위한 것이다.

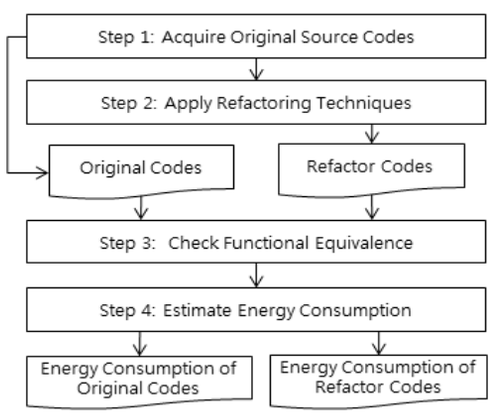


Fig. 11. Verification Procedure for Our Proposed Techniques

5.2 기본 코드 분석

제안한 기법을 검증하기 위해, 먼저 각 리팩토링 기법에 대한 기본 코드를 분석하여 소모전력 절감 가능성을 검증하였다. 기본 코드는 4장에서 기술한 각 예제 코드를 실행 가능한 코드로 변환하여 실험을 수행하였다. 실험 결과는 Table 7과 같다. Transform Loop Structure는 세부적으로 세 가지 Energy Bad Smells에 대한 실험을 각각 수행하였다.

이 실험에서 Simplify Expression 기법의 경우 절감효과가 3%대 낮은 편에 속하고 있으나, 복잡한 코드를 간결하게 하여 프로그램 이해도를 증가시킴으로써 소프트웨어의 안정성과 저전력 요구사항을 동시에 충족할 수 있다는 장점이 있다. Eliminate Dead Code의 경우 소모전력 절감 효과가 크지 않은 것으로 나타났으나, 기법의 특성상 Dead Code가 많이 발견될수록 절감효과가 증가할 것으로 예측된다. Transform Recursion과 Transform Loop Structure 또한 소모전력 절감 효과를 확인하였으며, 대상 코드의 특성상 반복 횟수가 증가할수록 절감 효과 또한 증가할 것으로 기대된다.

Table 7. Energy Consumption Analysis Results for the Given Example Codes

Techniques		Energy (nJ)		Difference (nJ)	Efficiency (%)
		Before	After		
Simplify Expression		35277	33992	1285	3.64
Eliminate Common Sub-Expression		6762	6115	647	9.57
Eliminate Dead Code		14556	14261	295	2.03
Transform Recursion		89375	85355	4020	4.50
Transform Loop Structure	Induction Variables	4939	4819	120	2.43
	Nested Loop	945508	753183	192325	20.34
	Global Variables	6967	6590	377	5.41

5.3 예제 코드 적용 및 분석

제안한 기법의 검증을 위해 실제 프로그램 개발에 사용되는 코드를 대상으로 소모전력 분석을 수행하였다. 실험 결과의 정확성을 높이기 위해, 사용자 입력을 받는 코드를 최대한 배재하고 실험대상을 선택하였다. 또한 코드 실행을 위한 데이터가 필요한 경우, Original Code와 Refactor Code에 동일하게 적용하였다. 실험을 위해 선정한 코드는 Table 8과 같다.

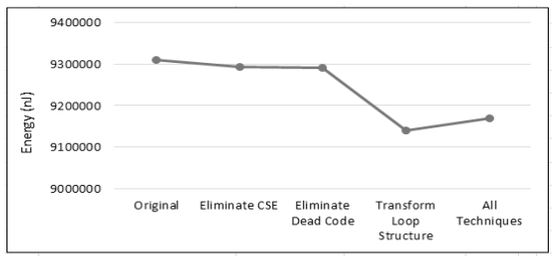
Table 8. Experimental Codes for Applying the Techniques

Exp. Codes	Description
Huffman Code	Compression Technique with data Lossless
K-Means	Clustering the given data to the k clusters
Binary Search	Location search of particular value in the sorted list

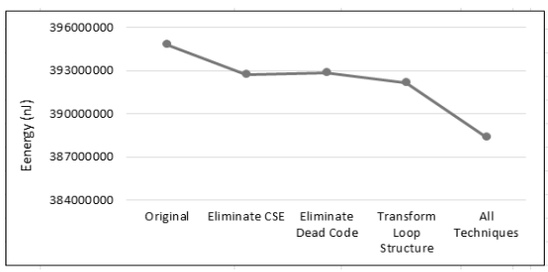
먼저 Huffman Code에 대한 실험 결과는 Fig. 12와 같다. Huffman Code에는 Eliminate Common Sub-Expression, Eliminate Dead Code, Transform Loop Structure 기법을 적용할 수 있었다. 여기서 Transform Loop Structure에 대해서는 3가지의 리팩토링 방법을 모두 적용하여 실험하였다. 각각의 기법을 적용한 결과와 적용 가능한 모든 기법을 이용하여 실험한 결과를 측정하였으며, 입력 데이터는 두 종류를 사용하였다. Fig. 12(A)는 10Kbyte, Fig. 12(B)는 200 Kbyte의 텍스트 파일을 각각 입력 데이터로 사용하였다.

실험 결과, 모든 경우에 대해 전력 소모량이 감소한 것으로 나타났다. Fig. 12(A)에서 모든 기법을 적용한 경우보다 Transform Loop Structure 기법을 단독으로 적용한 경우에 소모전력 절감 효과가 가장 큰 것으로 나타났으나, Fig. 12(B)에서는 모든 기법을 적용한 경우가 소모 전력량이 가장 많이 감소한 것으로 나타났다. 따라서 이는 입력 데이터의 크기에 따른 현상인 것으로 판단된다.

두 번째 실험은 K-means 알고리즘에 대하여 Simplify Expressions, Eliminate Common Sub-Expression, Eliminate Dead Code 기법을 적용하여 수행하였다. Fig. 13의 A, B, 그리고 C는 각각 100, 1000, 10000개의 입력 데이터를 사용하여 소모전력 분석을 수행했다. K-means 알고리즘에 대해서는 Simplify Expressions 기법을 적용했을 때 소모 전력량이 가장 감소되었으며, 다른 기법의 경우는 입력 데이터에 따라 소모전력 절감 효율이 서로 달라짐을 알 수 있었다.

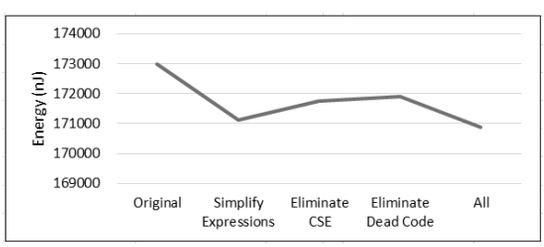


(A) Case of 10KByte Input Size

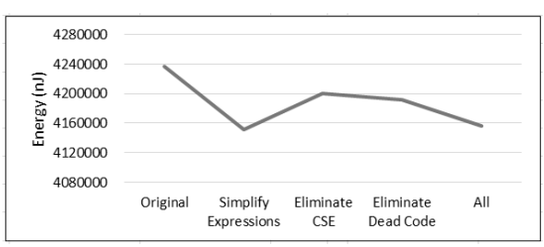


(B) Case of 200KByte Input Size

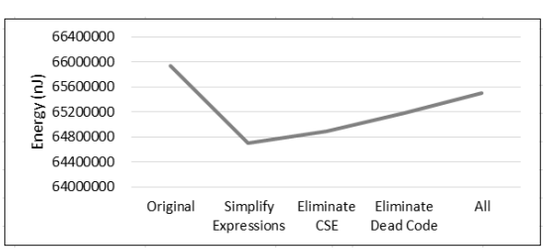
Fig. 12. Experiment Results for Huffman Code



(A) Case of 100 Input Data



(B) Case of 1000 Input Data



(C) Case of 10000 Input Data

Fig. 13. Experiment Results for K-means algorithm

Fig. 14는 Binary Search에 대한 실험 결과를 보이고 있다. 이 실험에서는 Recursion 코드를 Iteration 코드로 변환하였으며, 입력 데이터의 양에 변화를 주어 전력 분석을 수행하였다. 데이터의 크기와 무관하게 Iteration 코드가 전력 소모량이 적은 것으로 나타났다. 또한 입력 데이터의 증가량에 비해 전력 소모량의 증가율은 크지 않은 것으로 나타났다.

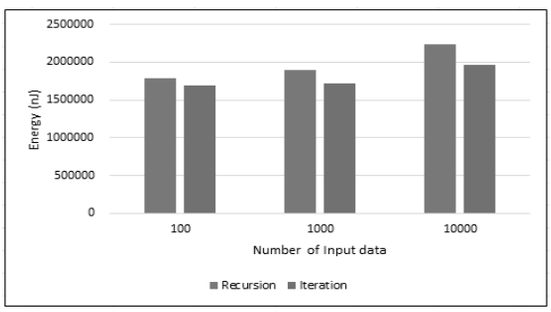


Fig. 14. Experiment Result for Binary Search

5.4 실험 가정 및 한계 요인 분석

다양한 실험 분석을 통해 제안한 기법이 소모 전력 절감에 효과가 있는 것을 알 수 있었다. 그러나 제안하는 기법이 소모 전력의 절감이라는 효과를 제공한다고 하더라도 다음과 같은 몇 가지 사항은 향후 연구에 있어서 고려해야 할 사항이다.

첫 번째는 입력 데이터에 따른 소모전력 효율의 변화에 대한 고려이다. Fig. 10의 Huffman code 실험에서 Transform Loop Structure의 기법에 대한 실험 결과를 보면, 입력 데이터의 크기에 따라 소모전력 절감 효율이 다르게 나타남을 알 수 있다. 또한 Fig. 11의 K-means 실험에서는 그래프에 나타나는 소모 전력의 패턴이 서로 다르게 나타난다. 이는 입력 데이터의 크기에 따라 소모 전력의 패턴이 일정하게 나타나지 않는다는 것을 의미한다. 비록 Refactor Code의 소모전력이 Origin Code보다 절감되기는 하였지만, 입력 데이터의 크기, 특성, 데이터 패턴 등도 함께 고려한다면 소모전력 절감에 대한 보다 상세한 분석이 가능할 것으로 판단된다.

두 번째는 리팩토링 기법의 적용으로 인하여 실험 대상 시스템의 소모전력 절감의 효과가 크게 나타나지 않았다. Table 7로부터 기본 코드에 대한 소모전력 절감의 비율은 최소 2.03% 수준이며, 평균적으로 Origin Code 대비 6.8%의 절감 효과를 보인다. 리팩토링을 수행하기 위한 노력 대비 얻어진 효과가 미흡하다고 보일 수 있으나, 전체 소프트웨어의 여러 부분에서 해당 기법이 적용되어 지는 경우, 상대적으로 더 많은 소모 전력이 절감될 것으로 예상하고 있다.

6. 결론 및 향후 연구

코드 리팩토링은 본래 코드의 기능을 유지하면서, 코드의 품질을 향상시키고자 하는 의도로 수행된다. 본 논문에서는 특히 소프트웨어의 소모 전력 특성을 고려하는 리팩토링 기법을 제안하였다. 이를 위하여 먼저 구조 변환을 통해 전력 소모량을 절감할 수 있는 잠재력을 가진 코드 패턴들을 식별하여 Energy Bad Smell로 정의하였으며, 각각에 대한 코드 리팩토링 기법을 제시하였다. 제시된 기법은 소프트웨어 개발에 일반적으로 사용되는 코드 패턴을 중심으로 분석을 수행하였으며, 효용성 검증을 위한 분석 실험을 통해 소모전력 절감 효과를 얻을 수 있음을 증명하였다.

실험 결과로부터 코드 구조의 변화가 소모전력 절감에 효과가 있음을 알 수 있었고, 제시한 리팩토링 기법의 적용으로 평균 6.8%의 소모전력 절감을 얻을 수 있음을 또한 확인하였다. 적용 기법에 따라 소모 전력 절감의 효과가 많은 차이를 보였지만, 실제 환경에서 1시간의 스마트폰 사용에서 3.6분 정도를 추가로 사용할 수 있다고 한다면, 위험 상황에서 긴급 통화를 이용하기에 충분하다고 판단할 수 있다. 또한 하나의 소프트웨어에서 여러 부분에 기법이 적용되는 경우 더 큰 소모 전력을 절감할 수 있을 것이다.

향후의 연구로는 본 연구에서 제안한 기법을 확장 및 조합하여 다양한 코드 패턴에 대한 소모전력 절감 기법을 도출할 수 있을 것으로 기대된다. 이를 위해서 본 논문에서 제시하는 기법간의 관계성과 상호 작용으로 인한 효과를 분석하여, 다수의 기법을 조합한 다측면 리팩토링(Multi-Aspects Refactoring) 기법이 연구되어야 할 것이다. 또한 보다 체계적인 기법의 적용을 지원을 위해서는 일반적인 코드 개발 도구와 통합된 리팩토링 도구를 개발하여야 할 것이다.

References

[1] Jang-Eui Hong and Doo-Hwan Kim, “Task Extraction from Software Design Models to Improve Energy Efficiency of Embedded Software,” *The KIPS Transactions: PartD*, Vol.18D, No.1, pp.45-56, 2011.

[2] S. Hao, D. Li, W. G. Halfond, and R. Govindan, “Estimating mobile application energy consumption using program analysis,” in *Software Engineering (ICSE), 2013 35th International Conference on,* IEEE, pp.92-101, 2013.

[3] T. K. Tan A. Raghunathan, and N. K. Jha, “Energy Macromodeling of Embedded Operating Systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, Vol.4, No.1, pp.231-254, 2005.

[4] H. Jun, L. Xuandong, Z. Guoliang, and W. Chenghua, “Modeling and Analysis of Power Consumption for Component-Based Embedded Software,” *Proc. Embedded Ubiquitous Computing Workshops 2006*, pp.795-804, 2006.

[5] B. Nogueira, P. Maciel, E. Tavares, E. Andrade, R. Massa, G. Callou, and R. Ferraz, “A Formal Model for Performance and Energy Evaluation of Embedded Systems,” *EURASIP Journal on Embedded Systems*, 2011.

[6] Brandolese, Carlo, et al., “The impact of source code transformations on software power and energy consumption,” *Proc. of Journal of Circuits, Systems, and Computers*, 2002.

[7] J. Jelschen, et al., “Towards Applying Reengineering Services to Energy-Efficient Applications,” *Software Maintenance and Reengineering (SCMR), 16th European Conference on IEEE*, 2012.

[8] D. Li, S. Hao, W. G. Halfond, and R. Govindan, “Calculating Source Line Level Energy Information for Android Applications,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp.78-89, 2013.

[9] Y. W. Kwon and E. Tilevich, “Reducing the Energy Consumption of Mobile Applications Behind the Scenes,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp.170-179, 2013.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code,” Addison Wesley, 2002.

[11] Gottschalk, Marion, et al., “Removing Energy Code Smells with Reengineering Services,” *Proc. of GI-Jahrestagung,* 2012.

[12] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, “Detecting Energy Bugs and Hotspots in Mobile Apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp.588-598, 2014.

[13] G. Pinto, F. Castor, and Y. D. Liu, “Understanding Energy Behaviors of Thread Management Constructs,” *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ACM, 2014.

[14] W. G. P. da Silva, L. Brisolara, U. B. Correa, and L. Carro, “Evaluation of the impact of code refactoring on embedded software efficiency,” I Workshop de Sistemas Embarcados, 2010.

[15] R. Pérez-Castillo and M. Piattini, “Analyzing the Harmful Effect of God Class Refactoring on Power consumption,” *IEEE Software*, Vol.3, pp.48-54, 2014.

[16] J. J. Park, J. E. Hong, and S. H. Lee, “Investigation for Software Power Consumption of Code Refactoring Techniques,” *International Conference on Software Engineering & Knowledge Engineering*, 2014.

[17] A. Pathak, Y. Charlie Hu, and M. Zhang, “Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices,” in *Hotnets'11*, November, 2011.

[18] M. Gottschalk, J. Jelschen, and A. Winter, “Energy-Efficient Code by Refactoring,” 15. Workshop Software-Reengineering, 2013.

[19] A. Vetro, L. Ardito, G. Procaccianti, and M. Morisio, “Definition, Implementation and Validation of Energy Code Smells: an exploratory study on an embedded system,” in *ENERGY 2013, The Third International Confrence on Smart Grids, Green Communications and IT Energy-aware Technologies*, pp.34-39, 2013.

[20] Planet Source Code [Internet], https://www.planet-source-code.com/ available at April, 04, 2016.

[21] Z. Herczeg, D. Schmidt, and et al., “Eergy simulation of embedded XScale systems with XEEMU,” *Journal of Embedded Computing*, Vol.3, No.3, pp.209-219, 2009.

이 재 욱

e-mail : jwlee@selab.cbnu.ac.kr
2010년 충북대학교 컴퓨터공학과(학사)
2012년 충북대학교 컴퓨터과학과(석사)
2012년~현 재 충북대학교 컴퓨터과학과 박사과정
관심분야 : Software Safety, Software Refactoring, Low-Energy Software Development

김 두 환

e-mail : dhkim@selab.cbnu.ac.kr
2007년 충북대학교 컴퓨터공학과(학사)
2009년 충북대학교 컴퓨터과학과(석사)
2016년 충북대학교 컴퓨터과학과(박사)
2016년~현 재 충북대학교 컴퓨터과학과 박사후과정
관심분야 : Software Quality, Software Reuse, Low-Energy Software Development

홍 장 의

e-mail : jehong@chungbuk.ac.kr
2001년 KAIST 전산학과(박사)
2003년 국방과학연구소 선임연구원
2004년 ㈜솔루션링크 기술연구소장
2004년~현 재 충북대학교 컴퓨터과학과 교수
관심분야 : Software Quality, Software Reuse, Model-Based SW Engineering, Software Process Improvement