KIPS Trans. Comp. and Comm. Sys.
Vol.9, No.10 pp.213~220
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2020.9.10.213

Code Visualization Approach for Low level Power Improvement via Identifying Performance Dissipation

Hyun Sik An† · Bokyung Park†† · R.Young Chul Kim††† · Ki Du Kim††††

ABSTRACT

The power consumption and performance of hardware-based mobile and IoT embedded systems that require high specifications are one of the important issues of these systems. In particular, the problem of excessive power consumption is because it causes a problem of increasing heat generation and shortening the life of the device. In addition, in the same environment, software also needs to perform stable operation in limited power and memory, thereby increasing power consumption of the device. In order to solve these issues, we propose a Low level power improvement via identifying performance dissipation. The proposed method identifies complex modules (especially Cyclomatic complexity, Coupling & Cohesion) through code visualization, and helps to simplify low power code patterning and performance code. Therefore, through this method, it is possible to optimize the quality of the code by reducing power consumption and improving performance.

Keywords : Low Power, Performance Visualization, Refactoring, Code Smell

성능 저하 식별을 통한 저전력 개선용 코드 가시화 방법

안 현 식† · 박 보 경†† · 김 영 철††† · 김 기 두††††

요 약

높은 사양이 필요한 하드웨어 기반의 모바일 및 IoT 임베디드 시스템은 저전력과 성능에 중요한 이슈를 갖고 있다. 이는 전력 소비로 발열량 증가 및 기기의 수명 단축 문제가 발생된다. 이러한 환경에서 소프트웨어도 제한된 전력, 메모리 등에서 안정적인 동작을 수행해야하므로 디바이스의 소비전력이 증가한다. 이를 해결하고자, 코드 관점에서 성능을 저하시키는 모듈을 식별하고, 그 모듈의 전력 최소화를 통한 성능 개선 가시화 방법을 제안한다. 이는 코드 가시화를 통해 복잡한 모듈(특히 Cyclomatic complexity, Coupling & Cohesion)을 식별하고, 저전력 코드 패턴화와 성능 코드를 간결화 한다. 이런 코드로 소비전력을 감소 및 성능 개선 함으로써 코드의 품질을 최적화 할 수 있다.

키워드 : 저전력, 성능가시화, 리팩토링, 코드 스멜

1. 서 론

최근 자율주행, 드론, 스마트폰 등 다양한 분야에서 임베디드 기기의 역할과 기능이 점차 확대되고 있다. 임베디드 시스템은 높은 사양을 가진 하드웨어로 구성된다. 이러한 시스템에서 작동하는 소프트웨어는 기존 소프트웨어의 고성능을 유지하면서 제한된 전력, 메모리 등과 같이 한정된 환경에서도 안정적인 작동이 가능하도록 소프트웨어 신뢰성이 요구되고 있다. 고사양의 하드웨어에서 고성능의 소프트웨어가 작동함으로써 소비전력이 증가하고 있으며, 임베디드 시스템의 소비전력 증가는 발열량 증가와 기기의 수명단축으로 이어진다. 이러한 임베디드 시스템의 문제 해결을 위해 본 논문에서 소프트웨어 코드의 전력 측정을 통한 전력 소모 최소화 및 성능개선 방법을 제안한다. 이 방법은 소프트웨어 가시화를 통해서 코드의 복잡도를 추출한다. 추출된 복잡도 중에서 소프트웨어의 성능을 저하시키는 모듈을 선정한다. 개선할 모듈은 가장 높은 복잡도를 가진 모듈들이다. 선택된 모듈은 본

※ 본 논문은 2019년도 산업통상자원부의 '창의산업융합 특성화 인재양성사업'(과제번호 N0000717)과 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2017R1D1A3B03035421).
※ 이 논문은 2020년 한국정보처리학회 춘계학술발표대회에서 "전력 소모 최소화를 통한 성능 개선의 코드 가시화 방법"의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 홍익대학교 전자전산공학과 석사과정
†† 준 회 원 : 홍익대학교 전자전산공학과 박사후연구원
††† 정 회 원 : 홍익대학교 소프트웨어융합학과 교수
†††† 정 회 원 : 한국정보통신기술협회 디지털인프라팀 팀장
Manuscript Received : July 22, 2020
First Revision : August 21, 2020
Accepted : September 1, 2020
* Corresponding Author : Ki Du Kim(kdkim@tta.or.kr)

연구에서 새롭게 정의한 저전력 패턴을 적용하여 소비 전력을 개선하고, 가시화를 통해 성능이 개선됐는지 확인한다. 본 논문의 구성은 다음과 같다. 2장은 소프트웨어 코드 기반의 저전력 연구를 소개한다. 3장은 전력 측정 실험을 통해 새롭게 정의한 저전력 코드 패턴을 언급한다. 4장은 사례연구, 5장은 결론 및 향후 연구를 기술한다.

2. 관련 연구

2.1 Energy Code Smell

저전력을 위한 소프트웨어 최적화 기술은 코드를 개선하여 전력소모를 줄이는 방법으로 연구되고 있고, 이것을 Energy Code Smell로 정의하고 있다. Vetro가 제안한 Energy Code Smell은 소모 전력을 감소시킬 수 있는 가능성이 높은 패턴이다[1]. 이 패턴은 총 9개의 패턴으로 구성되고 이를 리팩토링하여 소비전력을 감소시킨다. 하지만 에너지의 소모가 증가되는 경우도 있어서 모든 기법이 소비 전력을 절감하는데 적합하다고 볼 수 없다. 또한 Energy bad smells 기반 소모전력 절감을 위한 코드 리팩토링 기법을 사용하면 전력을 절감할 수 있다[2]. 하지만 코드를 유지보수 하는 과정에서 전체 소프트웨어의 어떤 모듈이 전력을 많이 소비하는지 알 수 없다. 이를 해결하기 위해 소프트웨어를 가시화하여 그 중 전력을 많이 소비하고, 성능을 저하시키는 모듈을 식별한다. 해당 모듈을 리팩토링하여 소비전력과 성능을 개선한다. 기존 연구에서는 반복문에 대해서만 논하였다[3]. 이를 보완 및 확장하는 패턴은 3장에서 언급한다.

2.2 성능 가시화

소프트웨어의 특징 중 하나로 비가시성을 예로 들 수 있다. 소프트웨어 가시화는 비가시성인 소프트웨어를 가시화하는 기법을 의미한다. 가시화 종류로는 코드 가시화, 성능 가시화가 존재한다. 코드가시화는 기본적인 가시화 메커니즘을 언급한다[4]. 성능가시화 절차는 다음과 같다. 1) 이미 개발된 소프트웨어의 내부구조를 파악하기 위해 파서를 이용하여 소스코드의 정보를 추출하고 추출물을 데이터베이스에 저장한다. 2) 내부구조를 표현할 모듈을 정의 한 뒤 소프트웨어의 복잡도 지표를 측정한다. 3) 성능 저하요소의 관한 패턴을 정의한다. 4) 측정한 결합도 지표와 성능 저하요소 정보를 통해 소프트웨어의 내부를 가시화한다. 7) 해당 소프트웨어의 문제점을 파악한다.

3. 코드 가시화 방안

3.1 성능 저하 식별 방법

전력소모를 최소화하기 위해서 먼저 소프트웨어의 어떤 모듈에서 전력을 많이 소모하는지, 성능을 저하시키는 요소가 있는지 식별이 필요하다.

Fig. 1은 본 연구에서 성능 저하 및 전력을 많이 소비하는 모듈을 식별하는데 사용한 성능 가시화의 흐름도이다. 먼저 소스내비게이터를 통해 소프트웨어 내부 정보를 SNDB 파일로 추출한다. SNDB는 바이너리 파일이다. 이 파일은 DBdump를 이용하여 텍스트로 변환한다. 그 후 변환된 데이터를 데이터베이스 생성 후 각 테이블에 저장한다. 품질지표 추출을 위한 쿼리문을 생성한다. 결과물들을 Graphviz에 Dot Script로 입력하면 가시화 그래프에서 소프트웨어의 성능을 확인 할 수 있다[5-7].

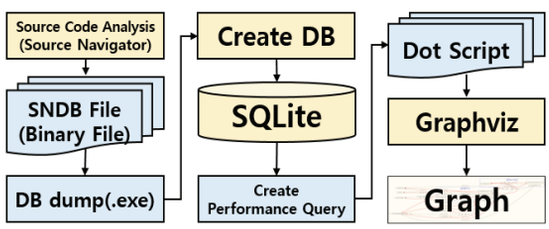


Fig. 1. Performance Visualization Flow Chart

3.2 전력 측정 절차

소프트웨어 코드의 전력 소모 최소화를 위한 전력 측정 방법을 제시한다. 소프트웨어에 의한 소모전력 절감 방법은 개발 초기 단계에서 소모 전력을 예측하는 기법과 개발된 코드의 검증을 통해 소모 전력을 분석하는 두 가지 방법이 있다. 본 연구에선 실제 구현된 코드를 가지고 검증을 하기 때문에 신뢰성이 높은 결과 값을 가질 수 있는 후자의 방법을 사용했다. 본 연구에서는 ARM사의 ULINK Plus 모듈을 사용하여 C코드의 소비전력량을 측정하고 Language 패러다임을 기반으로 패턴을 분류하여 저전력 코드 패턴을 제시한다.

3.3 환경구축

먼저 Fig. 2와 같이 소스코드 전력 측정을 위한 환경을 구성한다. 본 연구에서 사용한 도구는 ARM 사의 ULINK Plus 모듈과 Keil MCBSTM32F400보드다.

3.4 전력 측정 방법

Fig. 3은 소스 코드 전력 측정을 통한 전력소모 최소화 방안이다.

① Keil uVision5 IDE에서 프로젝트 생성하고 전력 측정하고자 하는 소스 코드를 입력한다. ② 환경 구성을 마치면 프로젝트를 실행한다. 프로젝트 실행은 3단계의 세부 단계로 구성되어있다. 먼저 입력한 소스코드를 빌드하여 컴파일 실행을 한다. 오류가 없으면 Load하여 보드에 업로드한다. 그

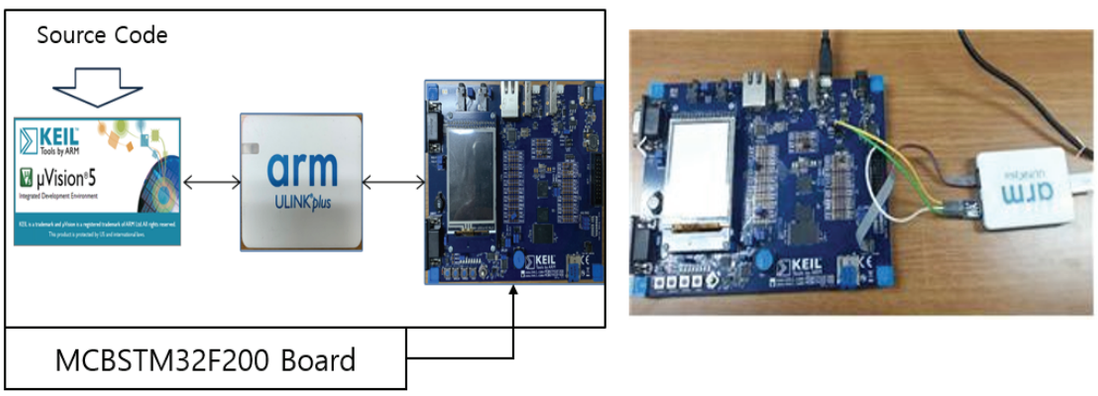


Fig. 2. Building a Power Measurement Experimental Environment

후 디버그모드를 실행한다. ③ 디버깅 모드에 들어가면 입력된 소스코드의 소비전력을 측정할 수 있다. 측정결과는 ④와 같은 그래프로 나온다. ⑤ 측정된 전력 값과 그래프를 참고하여 소스코드의 품질 만족 여부를 결정한다. ⑥ 만약 만족하지 못한다면 전력소모를 최소화하기 위한 코드로 개선한다. ⑦ 만족한다면 소스 코드 전력 측정을 통한 전력소모 최소화 절차를 종료한다.

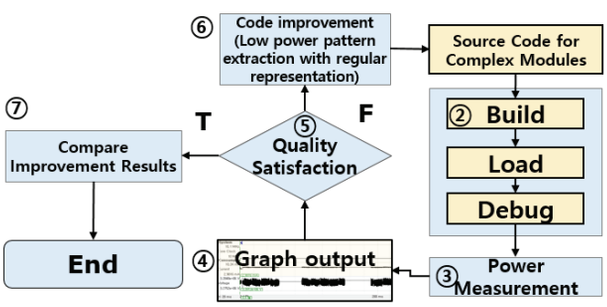


Fig. 3. Power Consumption Minimization Procedure Flow Chart

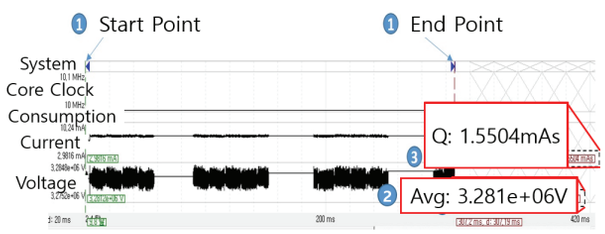


Fig. 4. Power Measurement Graph

Fig. 4는 측정 그래프이다. ①번은 프로그램의 시작점과 종료지점을 나타낸다. 프로그램이 실행되는 동안의 전력을 측정하기 위해서는, 이 시작점과 종료지점까지 사용된 전류를 측정해야 한다. ②에서 전압은 3.3V가 일정하게 공급된다. 실제로 그래프 상에서는 차이가 있지만 그 차이가 1/1,000,000 단위에서 변화하는 극소량이기 때문에 3.3V로 측정했다. ③에서 나오는 데이터는 현재의 전류량, 시작점과 종료지점에서 종료지점까지의 누적 전류량을 나타낸다. 이 누적 전류량을 전력을 구하는 공식인 Equation (1)에 대입하면 전력량을 측정할 수 있다.

$$P = V \times I \tag{1}$$

3.5 절차식 언어(C) 패러다임에 대한 저전력 코드 패턴 정의

소프트웨어 코드의 전력을 측정하기 위해 절차식 언어 패러다임에 대한 코드 전력 패턴을 분석하였다. 본 연구에서는 총 3개의 절차식 코드 패턴(Loop Up, Call by Reference, Multiple if then else)을 정의하고 전력을 측정하였다. 각 패턴은 최소 50번씩 전력을 측정하여 평균을 낸 값이다. 또한 이 패턴들을 전체 소프트웨어에서 추출하기 위해 Cppcheck와 정규표현식을 사용했다[8].

Table 1은 Loop Up과 Loop Down의 측정 결과이다. i

Table 1. Loop Up and Loop Down Pattern Measurement Results

Item	Pattern Name	
	Loop Up	Loop Down

Code Pattern	int main(){
	for (int i=0; i<100; i++){
	printf("*");
	}
	return 0;
	}
		int main(){
		for (int i=100; i>0; i--){
		printf("*");
		}
		return 0;
		}

Average Power Consumption	5.2670697mW	5.2263594mW
Regular Expression	[a-zA-Z_]([a-zA-Z_0-9])* [&⟨] ([a-zA-Z_0-9])+ ; [a-zA-Z_]([a-zA-Z_0-9])* \+\+	

가 0부터 100까지 1씩 증가하면서 “*”를 출력한다. Loop Up 패턴의 소비전력은 5.2670697mW이다. Loop Down은 i가 100부터 0까지 1씩 감소하면서 “*”를 출력한다. 소비전력은 5.2263594mW이다. 따라서 Loop Down 패턴을 사용하는 것이 전력소모를 줄일 수 있다. 다음은 정규표현식에 대한 설명이다. for문 내부의 조건식에서 ‘i’는 변수이기 때문에 ‘[a-zA-Z_]([a-zA-Z_0-9])*’로 표현한다. 변수는 맨 처음에 숫자가 올 수 없으므로 ‘[a-zA-Z_]’로 표현한다. 이 문장은 소문자 a부터 소문자 z, 대문자 A부터 대문자 Z까지와 언더바‘_’가 올 수 있다는 뜻이다. 그리고 그 뒤엔 문자와 숫자, 그리고 언더바‘_’가 올 수 있다. 이를 ([a-zA-Z_0-9])*로 표현한다. 뒤에 붙는 ‘*’는 해당 구문이 0개 이상 존재한다는 의미다. 이로 인해 변수의 이름, 길이에 상관없이 검출 할 수 있다. 조건식 내부의 값을 비교하기 위해 ‘[&〈]’을 사용했고 조건을 비교하기 위해 변수 혹은 숫자가 올 수 있으므로 이를 ‘([a-zA-Z_0-9])+’로 표현했다. ‘+’는 해당 구문이 1개 이상 존재한다는 의미다. 증감 식에서 ‘i++’를 표현한 방법은 변수를 표현하는 ‘[a-zA-Z_]([a-zA-Z_0-9])*’뒤에 ‘++’를 표현해주는 ‘\+\+’를 사용했다. 이 정규표현식을 Cppcheck에서 사용하면 Loop Up 패턴의 ‘i〈100; i++’와 같은 변수가 증가하는 반복문을 추출할 수 있다.

Table 2는 Call by Reference와 Call by Value의 측정결과다. func 함수에서 int double 두 가지 자료형으로 각각 Call by Reference, Call by Value로 매개변수를 받고 각각 2배로 곱해준다. 전력측정 결과 5.67433656mW, 5.3083668mW가 측정됐다. 따라서 Call by Value를 사용하면 전력소모를 줄일 수 있다. Call by Reference의 정규표현식은 함수의 매개변수 시작을 ‘\(’로 표현하고 모든 자료형을 검출하기 위해‘((char)*(short)*(int)*(long)*(longlong)*(float)*(double)*(long double)*)’으로 표현했다. 자료형 뒤에 오는 변수를 나타내는 ‘[a-zA-Z_]([a-zA-Z0-9_])*’을 사용하고 앞에 *로 포인터 변수를 표현했다. 이 패턴을 한 번 더 사용하여 매개변수 두 개를 Call by Reference하는 패턴을 검출할 수 있다. 매개변수가 3개, 4개, n개로 늘어 날 때에는 ‘[,]’를 추가하고 자료형 + 변수의 정규표현식을 더해주는 패턴을 이어가면 된다.

Table 3은 Multiple if then else와 Switch-Case의 측정 결과이다. 평균 소비전력은 각각 6.117389mW, 5.7922722mW가 측정되었다. 따라서 Multiple if then else보다 switch case문을 사용하는 것이 전력소모를 줄일 수 있다. Multiple if then else는 a의 값을 만족하는 조건을 모두 확인해야 한다. 하지만 switch case문은 a값을 만족하는 case문이 실행되면 break문을 통해 종료된다. 이러한 이유로 switch case문이 더 낮은 소비전력을 소모한다. 정규표현식은 Multiple if

Table 2. Call by Reference, Call by Value Pattern Measurement Results

Item	Pattern Name	
	Call by Reference	Call by Value

Code Pattern	void func(int *a, double *b){
	a *= 2;
	b *= 2;
	}
	int main(){
	int x=1;
	double y=3;
	func(&x,&y);
	return 0;
	}
		void func(int a, double *b){
		a *= 2;
		b *= 2;
		}
		int main(){
		int x=1;
		double y=3;
		func(x, y);
		return 0;
		}

Average Power Consumption	5.67433656mW	5.3083668mW
Regular Expression	\(((char)*(short)*(int)*(long)*(long long)*(float)*(double)*(long double)*) * [a-zA-Z_]([a-zA-Z_0-9])* [,] ((char)*(short)*(int)*(long)*(long long)*(float)*(double)*(long double)*) * [a-zA-Z_]([a-zA-Z_0-9])* \)	

Table 3. Multiple if then Else, Switch Case Pattern Measurement Results

Item	Pattern Name	
	Multiple if then else	Switch Case

Code Pattern	int main(){
	int a=80;
	if(a==80){
	printf(“A”);
	}
	else if(a==70){
	printf(“B”);
	}
	else if(a==90){
	printf(“C”);
	}
	else if(a==60){
	printf(“D”);
	}
	return 0;
	}
		int main(){
		int a=80;
		switch(a){
		case 80:
		printf(“A”);
		break;
		case 70:
		printf(“B”);
		break;
		case 90:
		printf(“C”);
		break;
		case 60:
		printf(“D”);
		break;
		}
		return 0;
		}

Average Power Consumption	6.1117386mW	5.7922722mW
Regular Expression	if \([a-zA-Z_]([a-zA-Z_0-9])* [&〈!=&〉&|]* [a-zA-Z_0-9]+ \)	

then else패턴을 추출하는데 사용된다. 조건문의 시작을 'if \('로 표현하고 내부의 변수를 '[a-zA-Z_]([a-zA-Z_0-9])*'로 나타냈다. 조건식에 올 수 있는 모든 식을 포함하기 위해 '[&<!=&>&|]*'을 사용했다. 비교대상을 '[a-zA-Z_0-9]+'로 표현하고 조건식의 마지막을 '\)'로 표현할 수 있다.

3.6 저전력 코드 패턴 추출 과정

앞서 작성한 3가지 패턴을 검출할 수 있는 정규표현식을 Cppcheck 명령어를 통해 추출한다. 추출한 다음 전력을 상대적으로 덜 소비하는 코드로 수정한다.

Fig. 5는 Loop Up 패턴을 검출하는 과정이다.

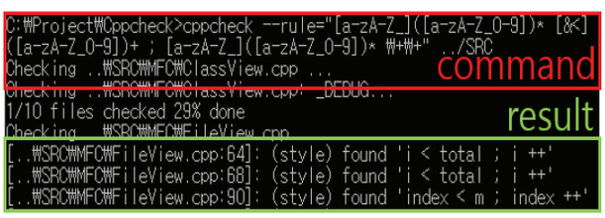


Fig. 5. Loop Up Pattern Extraction Results

정규표현식을 작성한 뒤 타겟소스의 주소를 입력한다. Fig. 5에서는 '../SRC'가 주소다. 이를 실행하면 주소에 있는 모든 파일에서 Loop Up패턴을 검출한다. 검사 결과 FileView. cpp 파일의 64, 68, 90번 째 줄에서 Loop Up 패턴이 검출됨을 확인 할 수 있다. 이 3개의 Loop Up패턴들을 Loop Down 패턴으로 수정하면 소비전력을 감소시킬 수 있다. 업 카운트 대신에 다운 카운트를 사용하면 최종 값을 저장하기 위한 레지스터를 할당하지 않고 매번 0과 값을 비교하는 작업의 비용이 들지 않기 때문이다.

Fig. 6은 동일한 방법으로 Call by Reference패턴을 검출한 것이다. 실행결과 MFCToolBarEx.cpp파일에서 (자료형 *변수)패턴을 찾아낸다. 총 6번 검출된 것을 확인할 수 있다. Call by Value를 사용하면 Call by Reference에 비해 메모리 사용량은 증가하지만 어셈블리 명령어를 덜 실행하기 때문에 총 6개의 Call by Reference 패턴을 Call by Value 패턴으로 수정하면 소비전력을 감소시킬 수 있다.

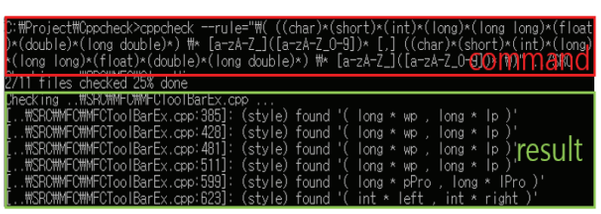


Fig. 6. Call by Reference Pattern Extraction Results

Fig. 7은 Call by Value와 Call by Reference의 어셈블리 명령어를 비교한 것이다.

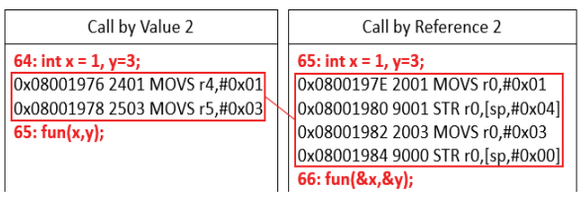


Fig. 7. Compare Call by Value and Call by Reference Assembly Commands

Call by Value의 경우 MOVS명령어만을 실행한다. MOVS명령어는 각각 #0x01, #0x03에 있는 매개변수 x, y의 값을 읽어서 r4, r5 레지스터에 저장하는 명령을 수행한다. 반면 Call by Reference의 경우 MOVS 명령어를 수행한 뒤 STR명령어를 추가로 수행한다. STR 명령어는 sp의 주소에 r0 레지스터 값을 저장한 뒤 다음 저장할 곳을 지정하기 위해 sp를 각각 #0x04, #0x00만큼 증가시키는 명령을 수행한다.

Fig. 8은 Multiple if then else를 추출할 수 있는 정규표현식을 Cppcheck를 통해 실행한다. 실행 결과 123개의 조건문이 검출되었다. 실제 코드를 확인해보면 Multiple if then else가 총 7번 발생한다. 발생하는 기준은 if문 다음에 else if문이 2번 이상 발생한 경우 Multiple if then else패턴으로 인식하도록 설정했다. else문은 횟수에 포함하지 않았다. 7번 발생한 Multiple if then else패턴을 Switch-Case문으로 변경하면 소비전력을 줄일 수 있다. 그 이유는 Multiple if then else패턴은 실행될 때 모든 조건을 비교한 뒤 조건에 만족되는 분기문에서 기능이 수행한다. 하지만 Switch-case문은 점핑테이블을 만들어 인덱싱하기 때문에 여러 번의 비교가 필요 없이 한 번에 찾아 기능을 수행하게 된다.

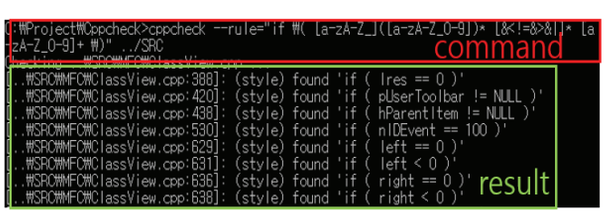


Fig. 8. Multiple If Then Else Pattern Extraction Results

4. 단계별 적용 사례

본 연구에서는 컨트롤러를 도구 내에 두고 다관절 로봇의 움직임을 제어하는 프로그램을 대상으로 실험을 진행했다[9].

Fig. 9는 성능가시화 그래프 결과물이다. 동그라미 모듈은 헤더파일로 외부참조 모듈이다. 각 모듈간의 호출 관계는 화살표로 표현하고 호출할 때 발생하는 결합도가 표시된다. D는 Data Coupling이고 결합도 수치 1을 갖는다. S는 Stamp Coupling이고 결합도 수치 2, CL은 Control Coupling을 의미하고 3의 결합도 수치를 갖는다. E는 External Coupling이고 6의 결합도 수치, CM은 Common Coupling 결합도 수치 8, CT는 Content Coupling이고 10의 결합도 수치를 갖는다. 결합도 수치가 30을 넘으면 호출관계를 나타내는 화살표가 빨간색으로 표시된다. 사각형 모듈은 모듈이름(호출횟수(번), 실행시간(ms), 성능저하요소(개))로 이루어져 있다. 성능저하요소가 3개 이상인 모듈을 '복잡한 모듈'이라 정의하고 해당 모듈을 빨간색으로 표시한다. 가시화된 '복잡한 모듈'에 저전력 패턴들을 적용해야 소비전력과 성능을 개선할 수 있다.

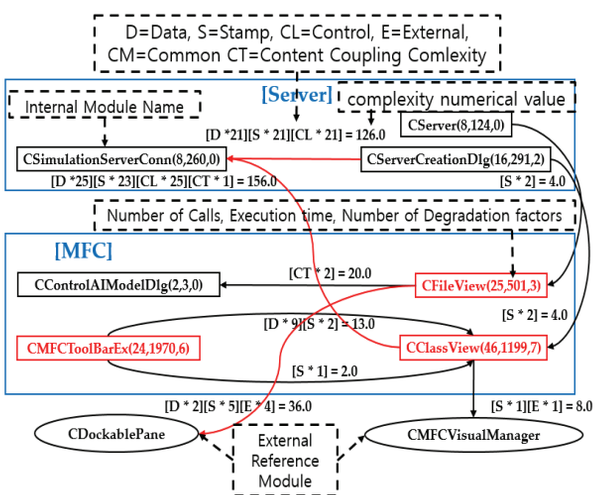
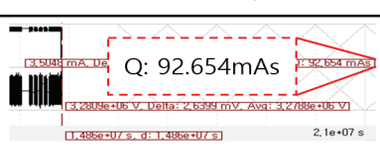
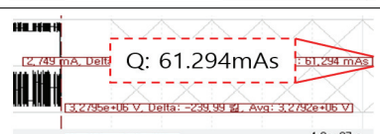


Fig. 9. Target Source Performance Measurement Results Graph

Table 4는 다관절 로봇 시뮬레이터 프로그램에 저전력 패턴을 적용 전, 후를 나타내는 전력측정 그래프이다.

Table 4. Multi-joint Robot Simulator Program Power Measurement Results Graphs

Before	
After	

개선 전 누적전류량이 92.654mAs이다. 전력량을 계산하면 305.7582mW값이 나온다. 이를 저전력 패턴을 적용한 후의 전류량은 61.294mAs로 적용 전보다 전류량이 감소했다. 전력량을 계산해 보면 202.2702mW이다. 총 33.8%의 전력감소효과를 얻을 수 있었다.

Table 5는 CMFCToolBarEx모듈에 저전력 패턴을 적용 전, 후 성능측정 비교테이블이다. CMFCToolBarEx 모듈의 Call by Reference패턴을 Call by Value패턴으로 변경한 결과 성능 저하요소 개수가 0개로 감소한 것을 통해 심각한 성능저하를 유발할 수 있는 패턴이 모두 사라졌음을 알 수 있다. 실행시간도 1970ms에서 1493ms로 감소하여 24.2% 성능의 향상을 볼 수 있다.

Table 5. CMFCToolBarEx Module Performance Improvement Comparison

Before	CMFCToolBarEx(24,1970,6)
After	CMFCToolBarEx(24,1493,0)

Table 6은 CClassView모듈의 성능개선 전, 후 비교테이블이다.

Table 6. CClassView Module Performance Improvement Comparison

Before	CClassView(46,1199,7)
After	CClassView(46,1056,0)

CClassView모듈에서 7번 발생하는 Multiple if then else 패턴을 Switch-case패턴으로 변경했을 때, 성능저하요소 개수가 7개에서 0개로 모두 사라졌다. 실행시간이 1199ms에서 1056ms로 11.9% 성능의 향상을 보인다.

Table 7은 CFileView모듈의 성능개선 전, 후 비교테이블이다.

Table 7. CFileView Module Performance Improvement Comparison

Before	CFileView(25,501,3)
After	CFileView(25,382,0)

CFileView모듈에서 3번 발생하는 Loop Up 패턴을 Loop Down으로 변경한 결과 성능저하요소가 모두 제거됐고 실행시간도 501ms에서 382ms로 23.7% 성능 향상을 보였다.

5. 결 론

IoT 기반 서비스 산업의 확장으로 IoT 디바이스들에 대한 수요가 증가하고 있다. 그러나 이러한 서비스들은 장기간 운용이 필요하다. 장기간 IoT 디바이스들을 운용하기 위한 저전력 사용 방법 및 성능 개선 등에 대한 연구가 필요하다. 본 연구에서 소프트웨어 코드의 전력 사용 효율화 및 성능 개선 방법을 제안했다. 이 방법은 전체 소스 코드의 복잡도를 가시화 하고 가장 복잡한 모듈들을 개선함으로써, IoT 디바이스 소프트웨어의 소비전력 감소 및 성능 향상이 가능하다. 소스코드의 내부를 가시화하기 위해 파서를 통해 소스코드의 정보를 추출한다. 추출된 정보는 데이터베이스에 저장한다. 저장된 정보를 이용하여 코드를 가시화하고 3가지(Loop Up, Call by Reference, Multiple if then else)의 성능저하요소 및 전력을 많이 소비하는 패턴들을 Cppcheck와 정규표현식을 활용하여 추출한다. 소프트웨어의 소비전력 감소와 성능 향상을 위해 위 코드들을 저전력패턴((Loop Down, Call by Value, Switch-case)으로 리팩토링한다. 그 결과 소비전력은 33.8%의 절감효과를 보였다. 성능에 대한 변화(실행시간이 각각 23.7%↓, 24.2%↓, 11.9%↓)를 알 수 있었다. 향후 연구로는 이번 연구에서 줄이지 못한 모듈간의 호출관계에서 발생하는 결합도를 개선하여 전력을 감소하고 성능을 개선하는 연구를 진행할 것이다. 또한 Java, Go Language, Python 등 성능 가시화를 적용할 예정이다.

References

[1] A. Vetro, L. Ardito, G. Procaccianti, and M. Morisio, "Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System," *The Third International Conference on Smart Grids*, pp.34-39, 2013.

[2] Jae-Wuk Lee, Doohwan Kim, and Jang-Eui Hong, "Code Refactoring Techniques Based on Energy Bad Smells for Reducing Energy Consumption," KIPS Tr. Software and Data Eng. Vol.5, No.5, pp.209-220, 2016.

[3] HyunSik Ahn, WonYoung Lee, and R. Young Chul Kim, "Guideline of extracting Low Power-Consumpted Code Mechanism with Power Consumption in High-Level Code," *2019 ICT Platform 2019*, pp.15-18, 2019.

[4] Bo Kyung Park, Geon-Hee Kang, Hyun Seung Son, Byung-Kook Jeon, and R. Young Chul Kim, "Code Visualization for Performance Improvement of Java Code for Controlling Smart Traffic System in the Smart City," *Applied Sciences*, Vol.10, Issue 8, pp.1-22, 2020.

[5] 강건희, 박보경, 장우성, 황준순, 권하은, 이한솔, 이현준, 김영철, 소프트웨어 성능 가시화를 위한 툴 체인 개발, KCSE 2016, Vol. 18, No.1, pp.395-398, 2016.

[6] Source Navigator NG. http://sourcenav.sourceforge.net/

[7] Graphviz, http://www.graphviz.org/

[8] CppCheck http://cppcheck.sourceforge.net/

[9] Bo Kyung Park, Byungkook Jeon, and R. Young Chul Kim, "Improvement Practices in the Performance of a CPS Multiple-Joint Robotics Simulator," *Applied Sciences*, Vol.10, pp.185-198, 2019.

안 현 식

https://orcid.org/0000-0003-0851-6346
e-mail : ahn@selab.hongik.ac.kr
2019년 홍익대학교 컴퓨터정보통신공학과 (학사)
2019년 ~ 현 재 홍익대학교 전자전산공학과 석사과정
관심분야 : 소프트웨어 공학, 소프트웨어 가시화, 소프트웨어 테스팅, 저전력 및 성능 가시화

박 보 경

https://orcid.org/0000-0002-7007-852X
e-mail : park@selab.hongik.ac.kr
2008년 홍익대학교 컴퓨터정보통신공학과 (학사)
2012년 홍익대학교 소프트웨어공학(석사)
2020년 홍익대학교 소프트웨어공학(박사)
2020년 ~ 현 재 홍익대학교 전자전산공학과 박사후연구원
관심분야 : 요구공학, 소프트웨어 역공학, 소프트웨어 가시화, 저전력 및 성능 가시화, AI 기반 소프트웨어 개발법

김 영 철

https://orcid.org/0000-0002-2147-5713
e-mail : bob@hongik.ac.kr
2000년~2001년 LG산전 중앙연구소
Embedded System 부장
2001년 ~ 현 재 홍익대학교
소프트웨어융합학과 교수
관심분야 : 테스트 성숙도 모델(TMM), 모델 기반 테스팅, 메타모델, 소프트웨어 프로세스 모델, 소프트웨어 가시화

김 기 두

https://orcid.org/0000-0002-6723-5950
e-mail : kdkim@tta.or.kr
2003년 홍익대학교 컴퓨터정보통신학과(학사)
2005년 홍익대학교 소프트웨어공학(석사)
2014년 홍익대학교 소프트웨어공학(박사)
2005년 ~ 현 재 한국정보통신기술협회 소프트웨어시험인증연구소
디지털인프라팀 팀장
관심분야 : 소프트웨어공학, 테스트프로세스, HPC, 3D Printing SW검증, 항공용 SW 품질 검증, AI 기반 소프트웨어 테스트 방법론