논문 2017-12-18

마이크로컨트롤러 환경에서 타깃 바이너리 파일 분석을 통한 최대 스택 메모리 사용량 예측 기법

(Maximum Stack Memory Usage Estimation Through Target Binary File Analysis in Microcontroller Environment)

최 기 호, 김 성 섭, 박 대 진*[1], 조 정 훈*[2]
(Kiho Choi, Seongseop Kim, Daejin Park, Jeonghun Cho)

Abstract : Software safety is a key issue in embedded system of automotive and aviation industries. Various software testing approaches have been proposed to achieve software safety like ISO26262 Part 6 in automotive environment. In spite of one of the classic and basic approaches, stack memory is hard to estimating exactly because of uncertainty of target code generated by compiler and complex nested interrupt. In this paper, we propose an approach of analyzing the maximum stack usage statically from target binary code rather than the source code that also allows nested interrupts for determining the exact stack memory size. In our approach, determining maximum stack usage is divided into three steps: data extraction from ELF file, construction of call graph, and consideration of nested interrupt configurations for determining required stack size from the ISR (Interrupt Service Routine). Experimental results of the estimation of the maximum stack usage shows proposed approach is helpful for optimizing stack memory size and checking the stability of the program in the embedded system that especially supports nested interrupts.

Keywords : Stack memory, Safety, Nested interrupt, ELF, Call graph

Ⅰ. 서 론

임베디드 시스템은 이미 우리 시대를 관통하고 있는 커다란 흐름이다. 도처의 모든 전자 제품은 하나 이상의 임베디드 시스템을 가지고 있으며, 우리의 삶 어느 곳에서나 볼 수 있다. 그 가운데, 자동차 및 항공 산업에서의 임베디드 시스템은 소프트웨어의 안전성을 가장 중요한 핵심 중 하나로 두고 있고, 이러한 소프트웨어의 안전성 보장을 위해 ISO26262 Part6 과 같은 수많은 소프트웨어 테스팅 기법들이 현재 연구 중에 있다. 특히, 소프트웨어의 스택 오버플로 테스팅은 이와 관련한 소프트웨어 안전성 확보 문제에 있어 중요한 이슈들 중 하나이다.

스택 오버플로란 할당되어 있는 스택 메모리 공간 이상의 영역을 프로그램이 메모리로 사용하는 것인데, 스택 오버플로 문제는 시스템 안전성 및 최적화 관점에서 다음과 같은 두 가지 문제를 안고 있다. 첫째, 스택 오버플로는 오직 프로그램 동작 중에 발생하기 때문에 프로그램 설계 시, 발견하기 어렵다는 것이다. 따라서 스택 오버플로는 예기치 못한 시스템의 안전성 저해를 초래할 수 있고, 실제 1995년 독일 철도역 프로그램 오류 사건 [1], 2005년 도요타사의 자동차 급발진 사태 [2] 등이 발생하였다. 둘째, 이러한 예상치 못한 스택 오버플로 문제를 방지하기 위해, 일반적으로 예상되는 스택 사용량 크기의 두 배를 할당하여 스택 메모리 사이즈로 결정하는데, 이는 시스템 최적화 관점에서 매우 큰 낭비이다. 바이너리 실행 이미지가 올라가

*Corresponding Authors ([1]boltanut@knu.ac.kr, [2]jcho@knu.ac.kr)
Received: May 12 2017, Revised: May 20 2017, Accepted: May 23 2017.
K. Choi, S. Kim, D. Park, J. Cho: School of EE, Kyungpook National University
※ 이 논문은 2015학년도 경북대학교 복현학술연구비에 의하여 연구되었음.

ⓒ IEMEK J. Embed. Sys. Appl. 2017 June 12(3) 159-167
ISSN : 1975-5066
http://dx.doi.org/10.14372/IEMEK.2017.12.3.159

는 플래시 메모리의 비용을 고려해 본다면, 이와 같이 관습적으로 두 배 크기의 스택 메모리 할당은 시스템 설계 시 비용적인 측면에서 매우 부적절하다. 따라서 시스템의 안전성 및 최적화 관점에서 스택 오버플로 방지를 위해 프로그램의 최대 스택 사용량을 추정하여, 시스템의 스택 메모리를 결정하는 것은 매우 중요하다 할 수 있다.

이와 관련한 연구로써 다중 인터럽트 환경에서 최대 스택 사용량을 분석하는 다양한 연구들 [3-5]이 진행되었지만, 이전 연구들에서는 인터럽트의 중첩은 허용하지만 각 인터럽트간의 세심한 우선순위 처리 방식이 존재하지 않아, 우선순위에 따른 중첩 인터럽트가 고려되지 않고 있다. 따라서 다양한 외부 인터페이스가 존재하고 이에 대한 제어기의 반응성을 높일 수 있도록, 우선순위를 기반 한 중첩 인터럽트를 제어할 수 있는 컨트롤러를 가지고 있는 고성능 임베디드 시스템에서는 적합하지 않다고 할 수 있다. 즉, 우선순위를 기반 한 세심한 중첩인터럽트 처리를 지원하는 고성능 제어기 시스템에서의 스택 사용량 분석에서는 정확한 최대 스택 분석을 위해 각 인터럽트 우선순위 및 중첩 인터럽트 처리방식에 대한 고려가 필수적이다.

본 논문에서는 각 인터럽트가 우선순위를 가지고, 중첩 인터럽트가 허용되는 Cortex-M4 환경에서 정적인 분석 방식을 통하여 프로그램의 최대 스택 사용량을 추정하는 기법을 제안한다. 실행 프로그램 파일을 분석하여, 프로그램내의 각 사용함수에 대한 스택 사용량을 파악하고 함수 호출 그래프를 구성하고 메인 프로그램의 최대 스택사용량을 추정한다. 그 다음으로는 ARM Cortex-M4에서 제안하는 중첩 인터럽트 제어방식으로 최대로 선점되어 중복되어 질 수 있는 인터럽트의 경우의 수를 파악하여 인터럽트 서비스 루틴으로부터 사용되는 최대 스택 사용량을 구하고, 메인 프로그램의 최대 스택 사용량과 합하는 방식으로 프로그램의 최대 스택사용량을 추정한다.

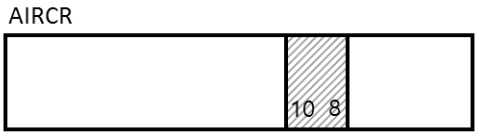
본 논문의 구성은 다음과 같다. Ⅱ장에서는 관련 연구에 관해 살펴보고, Ⅲ장에서는 제안하는 구조 및 알고리즘에 대해 소개하고, Ⅳ장에서는 실험을 통해 제시하는 기법의 타당성을 입증하며, 마지막으로 Ⅴ장에서 결론을 맺는다.

Ⅱ. 배경 지식

1. ARM 중첩인터럽트

ARM Cortex-M4 아키텍처는 중첩 인터럽트를 지원하기 위해 NVIC (Nested Vectored Interrupt Controller)를 제공한다 [6]. 다양한 입출력 장치를 효율적으로 지원하기 위해 단일 인터럽트가 아닌 우선순위에 기반 한 중첩 인터럽트를 제공하고 응용 프로그램에 적합한 우선순위 단계를 설정할 수 있다. 중첩 인터럽트는 이미 인터럽트가 발생하여 ISR (Interrupt Service Routine)이 동작 중인 상황에서 우선 순위가 더 높은 인터럽트가 발생하였을 경우 현재 인터럽트를 선점 (Preemption)하고 높은 우선순위의 ISR이 호출되는 방식을 의미한다. ARM Cortex-M4 아키텍처의 NVIC는 이러한 중첩 인터럽트를 제어하기 위한 유닛이고 다양한 외부 인터페이스가 있는 상황에서 제어기의 반응성을 높일 수 있는 핵심 유닛이다.

ARM Cortex-M4가 제공하는 NVIC는 240개에 대한 외부 인터럽트를 지원하고, 최소 8개에서 최대 256개 까지 우선순위 단계를 설정할 수 있다. 우선순위는 group priority 와 sub priority 로 나뉘는데, group priority는 preemption priority로 발생한 인터럽트의 group priority가 현재 인터럽트의 것보다 크다면 중첩이 발생한다. sub priority는 동일한 group priority를 가진 인터럽트들이 발생하거나 발생했지만 현재 인터럽트의 것보다 낮거나 같아 대기 상태에 있을 경우, 현재 인터럽트 루틴 처리가 끝나면 대기 중이던 동일한 group priority를



PRIGROUP	Group Priority	Sub Priority
0	2^7 level [7:1]	[0]
1	2^6 level [7:2]	[1:0]
2	2^5 level [7:3]	[2:0]
3	2^4 level [7:4]	[3:0]
4	2^3 level [7:5]	[4:0]
5	2^2 level [7:6]	[5:0]
6	2^1 level [7]	[6:0]
7	-	[7:0]

그림 1. AIRCR & PRIGROUP 설정
Fig. 1 AIRCR & PRIOGROUP configurations

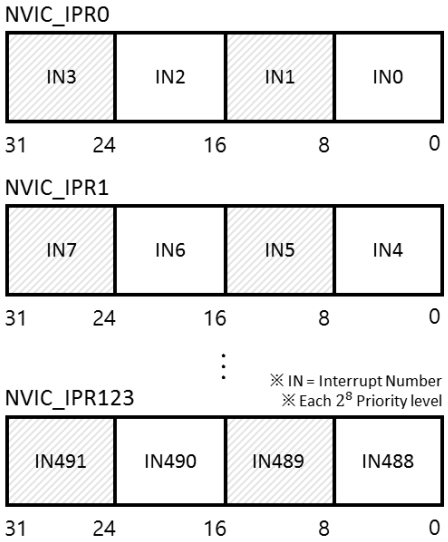


그림 2. NVIC_IPRn 설정
Fig. 2 NVIC_IPRn configurations

가진 인터럽트 중에서 가장 높은 sub priority를 가진 인터럽트가 우선적으로 처리된다.

Group priority는 그림 1과 같이 AIRCR (Application Interrupt and Reset Control Register)의 PRIGROUP bit의 설정을 통해 최대 128개까지 설정할 수 있다. 예를 들어 인터럽트 A의 group Priority가 다른 인터럽트 B의 것보다 크다는 것은 인터럽트 B가 발생하여 ISR이 동작하고 있을 때, 인터럽트 A가 발생한다면 인터럽트 B의 ISR이 중단되고, 인터럽트 A의 ISR이 먼저 수행되는, 즉 인터럽트 A가 인터럽트 B에 대해 선점권을 가지고 있다는 것을 의미한다. 이 때, 각 외부 인터럽트에 대한 우선순위는 그림 2의 해당 NVIC_IPRs 레지스터의 설정을 통해 결정된다.

ARM Cortex-M4는 이전의 M 시리즈와는 달리 빠른 인터럽트 응답성을 위해 스택 프레임 백업 및 ISR 진입을 위해 개발자가 별도의 코드를 작성하지 않아도 된다는 특징을 가지고 있다. 중첩 인터럽트를 지원하는 Cortex-M4의 NVIC는 인터럽트들의 중첩으로 인해 발생하는 스택 프레임 백업이 하드웨어적으로 이루어진다. 즉, 인터럽트 발생 시 레지스터 R0-R3, R12, LR, PSR, PC는 인터럽트를 위한 런타임 스택에 하드웨어적으로 백업이 이루어지므로 이를 위한 소프트웨어적인 코드가 필요하지 않다.

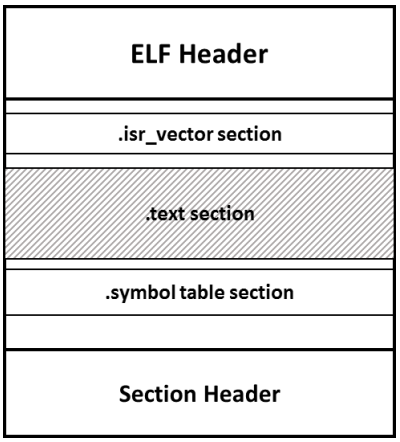


그림 3. Executable ELF 파일의 구조
Fig. 3 Architecture of ELF file

표 1. 각 섹션에 대한 추출정보
Table 1. Extracted data from each section

Section	Contents of the section
ELF Header	ELF file descriptions
Section Header	Each section descriptions
.text	Binary code descriptions (Stack usage, Function call, Interrupt configurations)
.isr_vector	Interrupt vector address
.symbol table	Used symbol descriptions (Function size, address)

2. ELF(Executable and Linkable Format)

ELF 파일은 실행파일, 목적파일, 공유 라이브러리 등을 위한 표준 파일 형식이다 [7]. ELF 실행파일은 그림 3와 같이 ELF 헤더, text section, isr_vector section, symbol table section 등의 각각의 섹션들로 구성되어 있다. 각 섹션에 대한 설명은 표 1에서 정리되어 있다. 우리는 ELF 포맷으로 되어 있는 최종 실행 파일을 분석하여 원하는 정보를 추출하고 전체 프로그램의 스택 사용량을 예측하려고 한다. ELF 헤더는 파일의 전체적인 구조를 나타내고, 각 섹션들은 해당 섹션들에 대한 정보를 담고 있다. text section과 symbol section은 프로그램 내의 각 함수들에 대한 정보를, isr_vector section의 경우 인터럽트에 대한 정보를 가지고 있다.

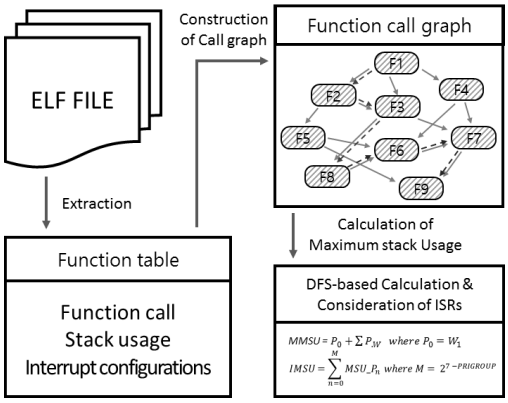


그림 4. 제안하는 전체적인 구조
Fig. 4 Overall architecture of proposed approach

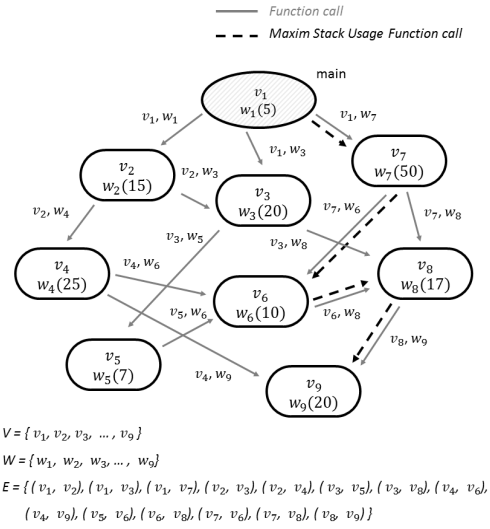


그림 5. 함수 호출 그래프 예제
Fig. 5 Example of function call graph

표 1은 ELF 파일의 각 섹션이 가진 정보를 나타내고 있다.

Ⅲ. 제안하는 구조

본 논문에서 제안하고 있는 전체적인 구조는 그림 4와 같다. 우선 분석하고자 하는 프로그램의 ELF 파일을 통해, 정적 스택 메모리 분석을 위한 function call, stack usage, interrupt configurations 정보를 추출하고, 이를 통해 함수호출 그래프를 구성한다. 다음으로 그래프 탐색을 통해 최대 스택을 사용하는 함수 호출 경로를 찾고, 이를 통해 메인 프로그램에서의 최대 스택 사용량 및 각 ISR의 최대 스택 사용량을 추정한다. 마지막으로, interrupt configurations 정보를 통해 인터럽트 중첩 시 최대로 사용될 수 있는 경우를 찾음으로써 전체 프로그램의 최대 스택 메모리 사용량을 추정할 수 있다.

1. ELF 파일로부터의 데이터 추출

우선, 분석하고자 하는 프로그램의 실행 파일 포멧인 ELF 파일분석을 통해 각 섹션들에서 필요한 정보를 얻는다. Symbol table section을 통해 프로그램 내에 있는 각 함수들에 대한 할당된 메모리 주소, 함수의 크기 정보를 얻고 이를 바탕으로 .text section을 통해 각 함수에 접근하여 각 함수들의 바이너리 코드를 분석한다. 본 논문에서는 ARM Cortex-M4 환경을 바탕으로 하여 실험을 구현하였으므로, ARMv7-M 아키텍처 기반의 Thumb2 instruction을 바탕으로 한 바이너리 코드 분석을 수행하였다. 바이너리 코드 분석을 통해 각 함수의 스택 사용량, 함수 호출 그리고, 인터럽트에 관한 설정 등에 대한 정보를 얻는다. ELF를 이용한 정적 분석은 동적인 바이너리 코드 분석방식 [8] 보다 강력한 분석을 가능케 한다.

2. 함수 호출 그래프

다음으로, ELF 파일 분석을 통해 얻은 함수들의 함수 호출 정보를 통해 프로그램 동작 시 발생하는 함수 호출에 대한 함수 호출 그래프를 구성한다. 함수 호출 그래프는 다음과 같이 정의된다.

정의 1 : 함수 호출 그래프를 G라고 하면 G는 *weighted direct graph*이고 $\{V, E, W\}$로 이루어져 있다.

$V = \{v_1, v_2, \ldots, v_n\}$
: 런타임 스택을 생성하는 함수들의 집합
$E = \{e_1, e_2, \ldots e_m\}$
$e_k = (v_i, v_j)$, $1 \le i, j \le n, 1 \le k \le m$
: v_i는 호출자, v_j는 피호출자
$W = \{w_1, w_2, \ldots, w_n\}$
w_i : 각 함수에서 사용하는 스택 프레임 크기

그림 5는 함수 호출 그래프를 구성한 예제를 보이고 있다. 함수 호출 그래프가 구성되면 깊이 우선

표 2. 호출 경로에 따른 스택 사용량
Table 2 Stack usage according to call path

No.	*Function call path*	*Stack usage*
1	v_1, v_2, v_3, v_5, v_6, v_8, v_9	*94*
2	v_1, v_2, v_4, v_6, v_8, v_9	*92*
3	v_1, v_2, v_4, v_9	*65*
4	v_1, v_2, v_4, v_8, v_9	*77*
5	v_1, v_3, v_5, v_6, v_8, v_9	*79*
6	v_1, v_3, v_8, v_9	*62*
7	v_1, v_7, v_6, v_8, v_9	***102***
8	v_1, v_7, v_8, v_9	*92*

탐색 방식을 바탕으로 한 그래프 탐색 [9]을 통해 프로그램의 초기 동작 시점인 시스템 리셋으로부터 main 함수 종료 시점까지의 스택 사용량을 분석할 수 있다. 즉, 다시 말해 .symbol table section과 .text section을 통해 얻은 각 함수의 스택 사용량 및 함수 호출 정보를 통해 함수 호출 그래프를 구성하고 그래프 탐색을 통해, 프로그램 동작 시 사용되는 스택 사용량을 분석할 수 있는 것이다.

예제에서는 main 함수를 포함한 9개의 함수 V들이 존재하고, 각 함수 호출 시에 사용되는 스택 프레임의 크기 W가 표기되어 있다. 또한 각 함수 사이의 호출 E이 나타나 있다. main 함수로부터 시작되는 그래프 G_{main}의 호출 경로 및 경로에 따른 스택 사용량을 나타내면 아래의 표 2와 같다. 표 2에서 예제 프로그램에서 경로 7이 가장 큰 스택을 사용하는 것을 알 수 있다.

그러나 위 경우에서 측정한 프로그램의 스택 사용량은 인터럽트 발생 시 인터럽트 서비스 루틴에 의해 사용되는 스택 사용량을 고려하지 않고 있다. 본 논문에서는 용어의 편의를 위해 앞서 논의한 프로그램 초기화 및 메인 프로그램의 동작 시 사용되는 최대 스택 사용량을 MMSU (Main Maximum Stack Usage)라고 지칭하고, 다음에 살펴볼 다수의 인터럽트들의 중첩으로 인한 인터럽트 서비스 루틴들로부터 사용될 수 있는 최대 스택 사용량을 IMSU (Interrupt Maximum Stack Usage)라고 지칭하였다. 최대로 스택을 사용하는 함수 호출 경로에 존재하는 함수의 스택프레임 크기를 w', 선택된 함수의 수를 N이라 할 때, MMSU를 수식적으로 나타내면 다음과 같다.

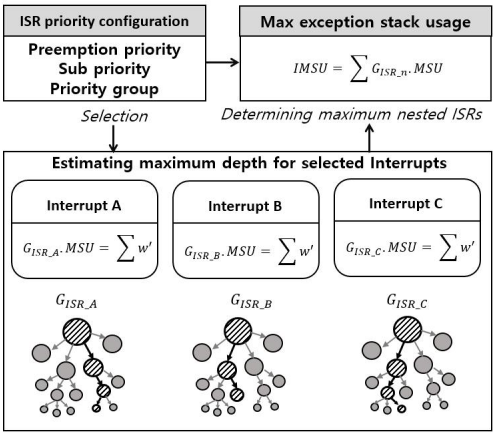


그림 6. 중첩 인터럽트 분석을 위한 제안구조
Fig. 6 Proposed structure for nested interrupt

$$MMSU = \sum_{i=0}^{N} w'_i \qquad (1)$$

3. 우선순위 중첩 인터럽트

그림 6은 IMSU 분석을 위한 전반적인 구조를 나타내고 있다. 우선 ELF 파일의 .isr_vector table section을 통해 프로그램에서 사용되는 인터럽트 목록을 구성하고, 각 인터럽트에 대한 인터럽트 서비스 루틴의 스택 사용량과 해당 인터럽트의 우선순위 레벨, 그리고 인터럽트 우선순위 레벨의 크기를 .symbol table section 및 .text section을 통해 추출한다. 인터럽트 우선순위 레벨의 크기 및 각 인터럽트의 우선순위 레벨과 스택 사용량을 고려하여, 발생할 수 있는 인터럽트 중첩의 경우들을 파악하고, 이 중 최대 스택 사용량인 경우를 선택한다. 이 때, 우선순위 레벨의 크기는 AIRCR.PRIGROUP bit를 설정하는 명령어를 통해 확인하고, 각 인터럽트의 우선순위 레벨은 NVIC_IPRn 레지스터를 설정하는 명령어를 통해 확인할 수 있다.

그림 7은 인터럽트의 중첩으로 인한 최대 스택 사용량 측정을 위한 알고리듬을 나타내고 있다. 본 논문은 Cortex-M4 환경을 중점으로 우선순위 중첩 인터럽트를 처리하고 있으므로, ARMv7-M 아키텍처에서 적용되는 NVIC 기반의 처리방식을 알고리즘에 적용하였다. 추출한 우선순위 레벨의 크기를 토대로 동일 우선순위 레벨 즉, 동일한 group

```
Function IMSU Determination

PRIGROUP = getPriorityGrouping();
set_group_priority_table(ISR_priority_table, PRIGROUP);
For isr_priority < 2^(7-PRIGROUP) do
 For isr_n do
  if get_ISR_priority_table(isr_n) == isr_priority
    temp = G_isr_n.MSU
    if temp > MSU[isr_priority]
      MSU[isr_priority] = temp
  done
  IMSU += MSU[isr_priority]
done
```

그림 7. 중첩 인터럽트를 고려한 IMSU 결정 알고리즘
Fig. 7 IMSU determination algorithm

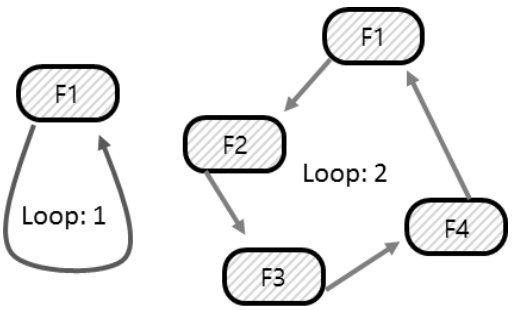


그림 8. 함수 호출 루프 제약조건
Fig. 8 Constraints of function call loop

priority를 가진 인터럽트 중 스택 사용량이 가장 많은 인터럽트를 선택하고, 각 우선순위 레벨별로 합하여 IMSU를 구하였다. 본 논문에서는 최대 스택 사용량을 측정을 그 목표로 하고 있으므로, 인터럽트의 최대 중첩이 발생할 수 있는 group priority 설정만을 고려 대상으로 삼았다. IMSU를 수식적으로 나타내면 다음과 같다.

$$IMSU = \sum_{n=0}^{M} MSUP_n \qquad (2)$$

$$where\ M = 2^{7-PRIGROUP}$$

4. 제약 조건

본 논문에서는 프로그램의 최대 스택 사용량 즉, Worst case에 대해서만 고려하므로 조건문이 프로그램에 존재하는 경우, 조건문의 실행 여부에 상관없이 포함된 모든 함수 호출이 이루어진다고 가정하고 있다. Worst case만을 고려하는 이유는 서론부에서도 언급한 것처럼, 스택 오버플로 문제는 얼마나 많이 스택을 사용하는가의 정도보다는 발생하느냐, 발생하지 않느냐가 프로그램이 동작하는 시스템의 안전성을 결정하기 때문이다.

또한 본 논문에서는 함수 호출 루프가 발생하는 경우를 배제하고 있는데, 함수 호출 루프란 첫째, 함수가 자기 자신을 호출하는 재귀호출을 가지어 루프를 형성하는 것과 둘째, 함수 호출 과정에서 루프가 형성되는 것을 말한다. 그림 8은 두 가지 경우의 함수 호출 루프를 나타내고 있다. 본 논문에서 함수 호출 루프를 배제하는 이유는 함수 호출 루프가 형성되는 함수의 스택 사용량은 정적인 예측이 불가능하기 때문이다. 그러나 MISRA (Motor Industry Software Reliability Association) 가이드라인에 따르면 재귀 호출을 엄격히 금하고 있는데 [10], 이는 재귀호출이 시스템의 안전성을 크게 해칠 가능성을 내포하고 있기 때문이다. 재귀 함수가 명시적으로 존재하는 첫 번째 경우는 물론이고, 함수 호출 과정에서 루프가 형성되는 두 번째 경우 역시 표면적으로는 다른 함수를 호출하는 것 같지만 실제로는 다시 자기 자신을 호출하므로 재귀호출이 이루어진다고 할 수 있다. 따라서 본 논문에서는 시스템의 안전성을 저해할 수 있는 함수 호출 루프를 고려하지 않더라도 제안하는 기법이 유효할 수 있다고 판단하였다.

표 3. 검증을 위해 사용된 벤치마크
Table 3. Benchmark used for verification

No.	*Benchmark*	*Number of interrupts*
1	*Whetstone*	*0*
2	*Dhrystone*	*1*
3	*Program A*	*1*
4	*Program B*	*5*
5	*Program C*	*7*

Ⅳ. 실험 및 결과

본 논문에서 제안하는 기법의 타당성을 검증을 위해 Cortex-M4 아키텍처를 기반으로 하는 STM32F407 Board가 사용되었다. 해당 보드에 총

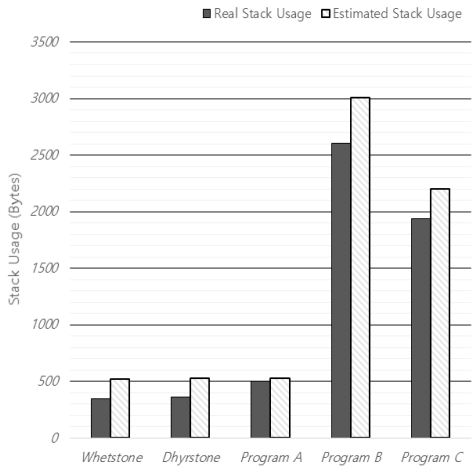


그림 9. 제안하는 기법의 실험 결과
Fig. 9 The result of proposed approach

5개의 벤치마크를 사용하였으며, 표 3은 실험에 사용된 5개의 벤치마크에 대한 정보를 나타내고 있다. 대부분의 벤치마크들은 프로세서의 성능을 검증하기 위해 고안되었으므로, 본 논문에서 중점으로 다루고 있는 중첩 인터럽트에 대한 처리를 가지고 있지 않다. 따라서 기존의 벤치마크 및 중첩 인터럽트를 처리하는 프로그램을 구성하여 실험을 진행하였다.

동적인 스택 사용량 분석 방법으로 테스팅 기법에 대한 많은 연구들이 진행되었지만 [11-12], 본 실험에서는 실험의 편의 및 정확성을 위해 검증된 디버그 툴을 사용하여 제안된 기법의 검증을 수행하였다. 우선, 프로그램 실행 전 스택 메모리 영역에 특정 값 (0xFFFFEEEE)을 쓰는 스택 메모리 초기화 프로그램을 실행시켜 스택 메모리를 특정 값으로 초기화 시키고, 벤치마크 프로그램을 일정시간 동안 동작시켜 사용된 스택 메모리 영역을 확인하는 방법으로 실 스택 사용량을 확인하였다. 그림 10은 벤치마크 프로그램의 실 스택 사용량과 제안하는 기법으로 측정된 최대 스택 사용량 및 정확도를 나타내고 있다.

그림 9의 실험 결과에서 첫 번째 칼럼 RSU (Real Stack Usage)은 디버그 툴을 사용하여 얻은 실 스택 메모리 사용량이고, 두 번째 칼럼 ESU (Estimated Stack Usage)은 본 논문에서 제안하는 기법을 통해 얻은 최대 스택 사용량을 의미한다. Whetstone의 경우 ESU가 524Byte, Dhystone, Program A의 경우, ESU가 532Byte로 비슷한데, 이는 벤치마크 프로그램의 스택 사용량이 시스템 리셋으로부터 시작되는 시스템 초기화 과정에서 main 함수를 호출하지 않는 경우일 때, 최대로 스택 메모리를 사용하기 때문이다. 다만, Dhystone, Program A의 경우 하나의 인터럽트에서 8bytes가 사용되어 Whetstone과 차이가 존재한다. 이와 달리, Program B와 Program C의 경우, main 함수 및 각 ISR에서 큰 스택 메모리 사용하여 서로 다른 ESU를 보인다.

모든 벤치마크 실험 결과에서 RSU는 ESU보다 작은 값으로 ESU가 본 논문에서 제안하는 기법으로 측정된 프로그램 동작 시 사용될 수 있는 최대 스택 사용량이라는 것과 본 논문의 제약조건을 고려한다면 타당성 있는 결과라 할 수 있겠다.

V. 결 론

본 논문에서는 제한된 임베디드 환경에서 소스코드 없이 ELF파일을 통해 프로그램의 최대 스택 사용량을 추정함으로써, 메모리 사용의 최적화 및 프로그램의 안전성 보장에 그 목적을 두고 있다. 메인 프로그램의 최대 스택 사용량을 추정하기 위해 시스템 리셋부터 시작되는 함수 호출 그래프를 구성하고, 최대 스택 사용량을 추정하였다. 그리고 우선순위 에 따른 중첩 인터럽트 발생 시 최대로 중복될 수 있는 경우를 구해 인터럽트의 중첩으로부터 사용될 수 있는 최대 스택 사용량을 추정하였다. 최대 스택 사용량을 정적인 방식으로 분석함으로써, 다양한 외부 인터페이스 환경에서 제어기의 반응성을 높일 수 있도록 설계된 고성능 임베디드 시스템의 스택 메모리 최적화 및 안전성 향상에 기여할 수 있을 것이라 생각된다.

Reference

[1] http://catless.ncl.ac.uk/Risks/16.93.html#subj.
[2] http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences.
[3] D. Bucur, M. Kwiatakoska, “On Software Verification for Sensor Nodes,” Journal of Software and Systems, Vol. 84, No. 10, pp. 1693-1707, 2011.
[4] J. Regehr, A. Reid, “HOIST: A System for Automatically Deriving Static Analyzers for

Embedded Systems," Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 32, No. 5, pp. 133-143, 2004.

[5] D. Brylow, N. Damgaard, J. Palsberg, "Static Checking of Interrupt-driven Software," Proceedings of the 23rd International Conference on Software Engineering, pp. 47-56, 2001.

[6] http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf

[7] http://refspecs.linuxbase.org/elf/elf.pdf

[8] D. Park, "Low-Power IoT Microcontroller Code Memory Interface Using Binary Code Inversion Technique Based on Hot-Spot Access Region Detection," IEMEK J. Embed. Sys. Appl, Vol. 11, No. 2, pp. 97-105, 2016

[9] K. Mehlhorn, S. Naher, P. Sanders, "Engineering DFS-based Graph Algorithm," Partially supported by DFG grant SA 933/3-1, 2007.

[10] https://misra.org.uk/

[11] J. Regehr, "Random Testing of Interrupt-driven Software," Proceedings of the 5th ACM international conference on Embedded software, pp. 290-298, 2005.

[12] G. Grancioli, S. Fischmeister, "Tracing Interrupts in Embedded software," Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems Vol. 44, No. 7, pp. 137-146, 2009.

Kiho Choi (최 기 호)

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2017. He is currently a M.S student in department of electronics engineering at Kungpook National university, Deagu, Korea. His research interests include Memory-embedded microprocesor architecture and its optimizing control algorithm.

Email: posjkh22@gmail.com

Seongseop Kim (김 성 섭)

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2017. He is currently a M.S student in department of electronics engineering at Kungpook National university, Deagu, Korea. His research interests include hardware-accelerated signal processing algorithm and high performance microprocessor-based system implementation

Email: kss92318@gmail.com

Daejin Park (박 대 진)

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2001, the M.S. degree and Ph.D. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2003, and 2014, respectively. He was a Research Engineer at Major Semiconductor Companies such as SK Hynix Semiconductor, Samsung Electronics over 12 years from 2003 to 2014, respectively and have worked on processor architecture design and low-power ASIC implementation with custom designed software algorithm optimization. Dr. Park is now with School of Electronics Engineering as full-time assistant processor in Kyungpook National University, Daegu, Korea and presidential research fellow.
Email: boltanut@knu.ac.kr

Jeonghun Cho (조 정 훈)

He received the B.S. degree in EE, the M.S. and the Ph. D degree in EECS from the Korea Advanced Institute of Science and Technology (KAIST), Deajeon, Korea in 1996, 1998, and 2003, respectively. He was a senior engineer at Hynix Semiconductor from 2003 to 2005. Main role was development of a C compiler for 8-bit microcontrollers. He is currently a professor with the School of EE of Kyungpook National University, Daegu, South Korea since 2005. His research interest includes a binary translation, software safety and security for automotive, AUTOSAR, run-time monitoring and logging for embedded system. Dr. Cho is a member of IEEE.
Email: jcho@knu.ac.kr