ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

소스코드와 실행코드의 상관관계 분석을 통한 최악실행시간 측정 방법☆

Measuring Method of Worst-case Execution Time by Analyzing Relation between Source Code and Executable Code

서 용 진[1] 김 현 수[1*]
Yongjin Seo Hyeon Soo Kim

요 약

내장 소프트웨어는 실시간성 및 실행 환경으로부터의 독립성을 요구사항으로 갖는다. 실시간성 요구사항은 탑재된 태스크의 최악 실행 시간으로부터 영향을 받는다. 따라서 실시간성을 보장하기 위해서는 정적 분석 기반의 최악 실행 시간 분석 방법을 사용하여 프로그램의 최악 실행 시간을 파악하여야 한다. 그러나 기존의 최악 실행 시간 분석은 실행 환경으로부터 독립성을 고려하지 않는다. 이에 우리는 실행 환경으로부터 독립성을 제공하기 위해 소스코드로부터 실행 시간을 측정하는 방법을 제시한다. 이를 위해 실행 코드가 아닌 소스코드로부터 생성된 제어 흐름 그래프를 통해 실행 시간을 측정한다. 또한 소스코드로부터 생성된 제어 흐름 그래프에는 실행 시간 정보가 존재하지 않기 때문에, 이를 제공하기 위해 소스코드의 문장과 실행코드의 명령어와의 관계를 분석한다. 결과적으로 실행 시간 측정이 가능한 제어 흐름 그래프를 생성할 수 있다. 이를 통해 프로세서로부터 종속적인 부분을 매개변수화할 수 있기 때문에, 최악 실행 시간 분석 도구의 유연성을 향상시킬 수 있다.

☞ 주제어 : 최악 실행 시간, 사이클 테이블, 소스 코드 기반의 실행 시간 측정

ABSTRACT

Embedded software has requirements such as real-time and environment independency. The real-time requirement is affected from worst-case execution time of loaded tasks. Therefore, to guarantee real-time requirement, we need to determine a program's worst-case execution time using static analysis approach. However, the existing methods for worst-case execution time analysis do not consider the environment independency. Thus, in this paper, in order to provide environment independency, we propose a method for measuring task's execution time from the source codes. The proposed method measures the execution time through the control flow graph created from the source codes instead of the executable codes. However, the control flow graph created from the source code does not have information about execution time. Therefore, in order to provide this information, the proposed method identifies the relationships between statements in the source code and instructions in the executable code. By parameterizing those parts that are dependent on processors based on the relationships, it is possible to enhance the flexibility of the tool that measures the worst-case execution time.

☞ keyword : Worst-case execution time, Cycle table, Source code based execution time measurement

1. 서 론

실시간성을 요구사항으로 갖는 내장 소프트웨어들은 주어진 시간 내에 주어진 임무를 수행할 수 있어야 한다. 이러한 내용을 보장하기 위해서는 내장 소프트웨어의 스케줄 가능성에 대한 검증이 이루어져야 한다. 스케줄 가능성에 대한 검증을 위해 사용되는 방법에는 최악 실행 시간(Worst-case execution time, 이후 WCET) 분석이 있다. WCET 분석을 수행하기 위해서는 소프트웨어가 갖는 모든 실행 경로를 파악하고 각 실행 경로마다의 실행 시간을 측정하여야 한다. 이후, 측정된 실행 시간 중 가장 긴 실행 시간을 소프트웨어의 최악 실행 시간으로 간주한다. 따라서 WCET 분석을 통해 측정된 실행 시간이 소프트웨어에게 할당된 시간보다 짧다면, 해당 소프트웨어는 스케줄 가능성을 만족한다고 할 수 있다[1].

WCET 분석은 크게 측정 기반의 분석 방법과 정적 분

[1] Department of Computer Science & Engineering, Chungnam National University, Daejeon, 34134, Korea.
* Corresponding author (hskim401@cnu.ac.kr)
[Received 14 April 2016, Reviewed 21 April 2016, Accepted 17 May 2013]
☆ 이 연구는 충남대학교 학술연구비에 의해 지원되었음

석 방법으로 나뉜다. 측정 기반의 분석 방법은 소프트웨어를 실행시켜 최악 실행 시간을 파악하는 반면, 정적 분석 방법은 다양한 소프트웨어 정적 분석 기법을 활용하여 최악 실행 시간을 파악한다. 측정 기반의 분석 방법은 실제 소프트웨어를 실행시켜야 하므로, 데이터 준비가 필수적이다. 또한 해당 데이터는 소프트웨어의 모든 경로를 확인할 수 있도록 준비되어야 한다. 그러나 소프트웨어의 모든 경로를 확인할 수 있는 데이터를 준비하는 것은 무척 어렵다[2,3]. 정적 분석 방법은 측정 기반의 분석 방법의 문제를 해결하기 위해 제시된 방법이다. 정적 분석 방법은 소프트웨어의 행위 분석(예, 흐름 분석, 반복문의 반복 횟수 분석, 데이터 분석 등)뿐만 아니라 하드웨어의 행위 분석(예, 파이프라인 분석, 캐시 분석 등)을 수행한다. 또한 소프트웨어의 실행 코드를 기반으로 소프트웨어의 행위 분석을 수행하기 때문에, 실행 코드를 생성하는 컴파일러에 대한 분석을 수행하는 경우도 있다[4,5,6].

최근에는 내장 소프트웨어의 또 다른 요구사항으로 실행 환경으로부터의 독립성이 요구되고 있다. 여기서 실행 환경은 운영체제 및 하드웨어를 의미한다. 이런 요구사항을 만족하기 위해 다양한 분야에서 여러 플랫폼이 제시되고 있다. 예를 들면, 자동차 분야의 AUTOSAR[7], 항공 분야의 IMA 아키텍처[8], 우주 분야의 SAVOIR[9]와 cFE[10] 등이 있다. 이와 같은 플랫폼에서 실행 환경으로부터의 독립을 제공하는 이유는 내장 소프트웨어의 이식성(Portability)을 제공하여 재사용성을 증대시키기 위함이다. 이를 통해서 내장 소프트웨어는 본래 목표로 했던 시스템과 다른 실행 환경을 갖는 시스템에서도 동작할 수 있다.

그러나 내장 소프트웨어의 이식성 증대는 WCET 분석에 있어서 또 다른 문제를 야기한다. 기존의 WCET 분석 도구들은 실행 코드를 기반으로 동작하기 때문에, 같은 소프트웨어라고 할지라도 새로운 실행 환경에 동작하여야 하는 상황이 온다면 WCET 분석 도구가 새로 구현되어야 한다. 하드웨어가 변경되면 실행 코드를 구성하는 명령어도 변경되며 이전의 도구로는 새로운 환경을 위한 소프트웨어를 분석할 수 없기 때문이다. 결국, 실행 환경으로부터의 독립성 제공을 통해 소프트웨어의 개발 시간은 단축되지만, 소프트웨어의 검증이 이를 뒷받침하지 못하는 문제가 발생하는 것이다. 이를 해결하기 위해서는 WCET 분석 역시 내장 소프트웨어의 추세에 발맞추어 실행 환경으로부터 독립성을 가질 필요가 있다.

이와 같은 문제를 해결하기 위해 본 논문에서는 실행 코드 대신 소스 코드로부터 프로그램의 실행 시간을 측정하는 방법을 제시한다. 실행 코드는 소프트웨어가 동작하게 될 프로세서가 결정되어야 얻을 수 있는 것이므로, 실행 코드를 통해 WCET 분석을 수행한다는 것은 실행 환경에 종속성을 갖는다는 의미가 된다. 따라서 소스 코드만을 이용하여 WCET 분석을 수행할 수 있다면 실행 환경으로부터 독립성을 가질 수 있다. 이를 위해서 소스 코드를 구성하는 단위 문장과 어셈블리 명령어 사이의 관계를 분석하고, 이를 바탕으로 소프트웨어의 실행 시간을 예측하는 방법을 제시한다. 소스 코드 기반의 WCET 분석을 수행하기 위해서는 소스 코드만으로 소프트웨어의 실행 시간을 측정할 수 있어야 한다. 따라서 본 논문에서는 WCET 분석 자체보다는 소스 코드 기반의 소프트웨어 실행 시간 측정에 대한 방법에 초점을 맞추어 연구를 진행한다.

2. 관련 연구

최악 실행 시간 분석은 프로그램을 구성하는 여러 실행 경로 중에서 가장 긴 실행 시간을 갖는 경로의 실행 시간을 해당 프로그램의 실행 시간으로 간주하는 방법이다. 최악 실행 시간 분석은 크게 정적 방법과 측정 기반의 방법으로 나뉜다. 정적 방법은 코드를 실행시키지 않고 최악 실행 시간을 분석하는 방법으로, 일반적으로 다양한 분석 기법을 사용하여 최악 실행 시간을 예측한다. 정적 분석 방법에서 사용하는 분석 기법은 다음과 같다.

- 제어 흐름 그래프 (Control Flow Graph)
- 값 분석 (Value Analysis)
- 반복 횟수 분석 (Loop Bound Analysis)
- 경로 분석 (Path Analysis)
- 파이프라인 분석 (Pipeline Analysis)
- 캐시 분석 (Cache Analysis)

위 분석 중에서 제어 흐름 그래프 구축, 값 분석, 반복 횟수 분석, 경로 분석은 모두 소프트웨어의 행위에 대한 분석이다. 소프트웨어의 행위라고 표현했지만 결국은 소프트웨어가 갖는 실행 경로를 분석하기 위한 요소들이다. 파이프라인 분석과 캐시 분석은 하드웨어의 행위에 대한 분석이다. 정적 방법을 사용하는 최악 실행 시간 분석 도구는 다음과 같다. aiT[4]는 AbsInt GmbH에서 만든 상용 도구로, 실행 코드를 분석하여 최악 실행 시간을 분석한다. 실행 코드로부터 제어 흐름 그래프를 구축하고,

이를 기반으로 값 분석, 반복 횟수 분석, 경로 분석을 수행한다. 또한 모사된 프로세서를 정의하여 파이프라인 및 캐시 분석도 병행한다. Bound-T[5]는 경성 실시간 임베디드 소프트웨어에 해당하는 인공위성의 탑재 소프트웨어의 최악 실행 시간을 측정하기 위해 개발되었다. Bound-T는 실행 코드로부터 제어 흐름 그래프를 구축한 뒤, 각 경로의 존재하는 어셈블리 명령어의 수를 계산하여 최악 실행 시간을 측정한다. TU Vienna[6]는 그래프 기반의 접근 방식을 사용한다. 여기서 사용하는 그래프는 T-Graph(Timing Graph)로, 흐름과 함께 가중치를 이용하여 실행 시간을 표현할 수 있다. TU Vienna는 프로그램을 T-Graph로 표현한 뒤, 최악 실행 시간 측정을 그래프 이론 관점에서 수행한다.

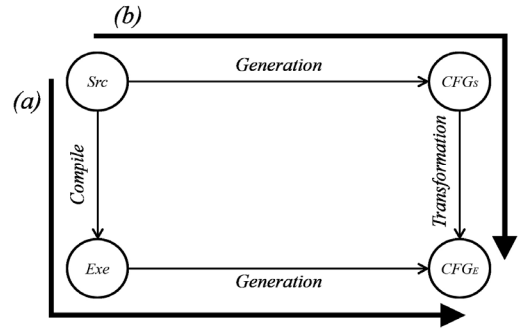
중요한 것은 위 연구들은 모두 실행 코드로부터 제어 흐름 그래프를 구축한다는 것이다. 실행 코드는 어셈블리 명령어로 이루어지기 때문에, 실행 코드를 기반으로 제어 흐름 그래프를 만들면 실행 시간을 간접적으로 측정할 수 있는 사이클의 수를 파악할 수 있다. 즉, 제어 흐름 그래프를 통해 소프트웨어의 실행 경로를 파악하는 것만으로도 실행 시간을 측정할 수 있다는 의미이다. 그러나 실행 코드로부터 제어 흐름 그래프를 도출하기 위해서는 실행 코드를 구성하는 어셈블리 명령어의 종류 및 형식을 파악하여야 한다. 또한 프로세서의 특성도 파악하여야 한다. 이렇게 파악된 내용을 바탕으로 실행 코드를 분석하는 도구를 개발하여야 한다. 만일, 새로운 환경에 대해 WCET 분석을 수행하기 위해서는 실행 코드를 구성하는 어셈블리 명령어에 대해 새로 분석하여야 할뿐만 아니라 분석된 내용을 바탕으로 실행 코드를 분석하는 도구를 새로 개발하여야 한다. 이러한 이유로 위 WCET 분석 도구는 자신들이 지원할 수 있는 컴파일러 및 프로세서의 목록을 제공하고 있으며, 각각마다 별도의 실행 파일을 제공한다.

본 논문에서는 기존 연구가 가지고 있는 실행 환경의 종속성 문제를 해결하기 위해 소스코드 기반의 실행 시간 측정 방법을 제시한다. 이를 위해 기존 연구와 달리 소스코드로부터 제어 흐름 그래프를 생성한다. 기존 연구에서는 제어 흐름 그래프로부터 실행 시간을 측정하기 때문에, 실행 시간을 파악할 수 있는 실행코드로부터 제어 흐름 그래프를 생성한다. 이로 인해 실행 환경의 종속성이 발생하기 때문에, 본 연구에서는 소스 코드로부터 제어 흐름 그래프를 생성한다. 대신 소스코드로부터 실행 시간에 대한 정보를 얻을 수 없기 때문에, 소스 코드와 실행코드 사이의 상관관계를 분석한다. 여기서 수행하는 분석은 특정 소스코드로부터 생성되는 실행코드를 파악하는 것이며, 이를 통해 간접적으로 소스코드로부터 실행 시간을 파악할 수 있다. 다만, 이 정보는 추상적인 형태로 유지되어야 한다. 다시 말해, 본 논문에서는 소스코드로부터 제어 흐름 그래프를 생성하고 소스코드와 실행코드 사이의 상관관계를 분석함으로써 실행 시간을 측정할 수 있는 제어 흐름 그래프를 생성한다. 이 방법을 이용하면 기존의 연구처럼 컴파일러 및 프로세서마다 별도의 실행 파일을 제공하는 것이 아니라 소스코드와 실행코드 사이의 상관관계 정보만을 제공함으로써 다양한 실행 환경에 대해 최악실행시간을 측정할 수 있다.

3. 단위 문장과 어셈블리 명령어 사이의 관계 분석을 통한 소프트웨어 실행 시간 측정

3.1 소프트웨어 실행 시간 측정 전략

그림 1은 소프트웨어의 실행 시간을 측정하기 위한 두 가지 방법을 도식화한 것이다. 여기서 그림 1(a) 방법은 기존 연구에서 사용하는 방법이며, 그림 1(b) 방법은 본 연구에서 제시하는 방법이다.



(그림 1) 소프트웨어의 실행 시간 측정 방안
(Figure 1) Measuring method for software execution time

그림 1(a) 방법은 먼저 소스 코드로부터 실행 코드를 도출하는 것으로 시작한다. 이 작업은 컴파일러를 통해 수행되며, 컴파일러와 프로세서의 종류에 영향을 받는다. 실제로 이 과정을 통해 생성될 수 있는 실행 코드의 수는 매우 많다. 결국 그림 1(a) 방법을 이용하기 위해서는 해당 소프트웨어의 동작 환경이 결정되어야 함을 알 수 있다. 다음으로 실행 코드를 기반으로 제어 흐름 그래프를

구축한다. 이를 구축하는 이유는 흐름 분석, 경로 분석, 반복 횟수의 분석 등을 수행하기 위함이다. 앞서 말했듯이 생성 가능한 실행 코드의 종류는 매우 많기 때문에, 이 과정에서 생성될 수 있는 제어 흐름 그래프의 종류도 다양할 수밖에 없다. 이것이 내장 소프트웨어가 실행 환경으로부터 독립성을 가짐으로써 발생할 수 있는 기존의 WCET 분석의 문제이다. 새로운 동작 환경이 생겨날 때마다 새로이 제어 흐름 그래프를 구축하는 방법을 연구하고 개발하여야 한다. 어쨌든 이와 같은 과정을 통해 제어 흐름 그래프를 구축한 이후에는 IPET (Implicit Path Enumeration Technique) 방법을 통해 각 경로의 실행 시간을 측정할 수 있다. 실행 코드 기반의 제어 흐름 그래프를 구성하는 각 노드는 어셈블리 명령어로 구성되기 때문에, 각 노드마다 얼마큼의 시간을 소요하는지 예측할 수 있다.

그림 1(b) 방법은 소스 코드로부터 제어 흐름 그래프를 구축하는 것으로부터 시작한다. 이후, 소스 코드 기반의 제어 흐름 그래프를 실행 코드 기반의 제어 흐름 그래프로 변환한다. 이와 같은 과정이 가능한 이유는 다음과 같다. 먼저, 소스 코드와 실행 코드는 구성되는 문장의 종류는 다르더라도 행위적으로 동치이다. 또한 소스 코드와 실행 코드로부터 생성된 제어 흐름 그래프는 각 코드의 행위적 모델이다. 다시 말해, 소스 코드 기반의 제어 흐름 그래프는 소스 코드가 갖는 행위를 바탕으로 구성되었으며, 실행 코드 기반의 제어 흐름 그래프는 실행 코드가 갖는 행위를 바탕으로 구성된다. 서로 다른 코드를 바탕으로 구성되었지만 두 제어 흐름 그래프는 동치를 이룬다. 따라서 소스 코드 기반의 제어 흐름 그래프로부터 실행 코드 기반의 제어 흐름 그래프를 도출하는 작업을 굳이 수행하지 않더라도, 소스 코드 기반의 제어 흐름 그래프는 이미 실행 코드 기반의 제어 흐름 그래프와 동일한 작업을 수행할 수 있는 그래프임을 알 수 있다. 문제는 실행 코드 기반의 제어 흐름 그래프의 경우에는 어셈블리 명령어로 각 노드가 구성되기 때문에, 추상적으로나마 실행 시간을 파악할 수 있다. 그러나 소스 코드 기반의 제어 흐름 그래프는 각 노드가 고수준의 프로그래밍 언어 문장들로 구성된다. 이와 같은 내용을 통해서는 직접적으로 실행 시간을 파악하는 것이 어렵다.

따라서 이 논문에서는 소스 코드로부터 실행시간을 측정하기 위해 단위 문장-어셈블리 명령어의 관계를 분석한다. 소스코드의 단위 문장마다 관련된 어셈블리 명령어를 파악하고, 이를 통해 소요되는 사이클 수를 분석한다. 이 방법은 “프로그램은 어셈블리 명령어로 변환되어 실행되며, 각 어셈블리 명령어는 고유의 사이클 수를 갖는다.”는 사실에 기반을 둔다. 여기서 사이클은 클럭(Clock)이라고도 하며, 프로세서에서 명령어를 수행하는 각 단계(예를 들면, Fetch, Decode, 등)를 수행하는데 소요되는 시간을 의미한다. 만일, C 언어로 작성된 문장에 대응되는 어셈블리 코드를 알 수 있다면, 각 문장마다 소요되는 사이클의 수뿐만 아니라 프로그램의 총 소요 사이클 수를 예측할 수 있다.

궁극적으로 실행 시간을 측정하는데 있어 프로세서로부터 자유로울 순 없다. 그러나 기존 WCET 분석 도구의 문제점은 변경 가능성(Modifiability)을 만족하지 않는다는 것이다. 프로세서의 변경으로 인해 도구 자체가 재개발되는 문제가 발생되는 것이다. 본 논문의 방법은 프로세서와 종속적인 부분을 설정(Configuration)으로 다루기 때문에, 설정 값만을 바꿈으로써 새로운 프로세서에 맞는 WCET 분석을 수행할 수 있다. 즉, 본 논문의 전략은 기존의 방법에 비해 높은 유연성을 갖는 WCET 도구를 구현할 수 있게 해준다. 이를 위해서 반드시 수행되어야 하는 작업이 단위 문장과 어셈블리 명령어 사이의 관계를 분석하는 것이며, 이 내용은 다음 절에서 자세히 다룬다.

3.2 단위 문장-어셈블리 명령어 관계 분석

3.1절의 전략을 달성하기 위해서는 소스코드의 단위 문장으로부터 생성될 어셈블리 명령어를 파악하여야 한다. 소스코드로부터 어셈블리 명령어로의 변환은 컴파일러를 통해 수행된다. 이 논문에서는 TI(Texas Instruments)의 TMS320C67x DSP[11]를 대상으로 분석하였다. 문장에 대한 사이클 관계를 분석하기 위해서는 (1)입력으로 사용될 문장의 종류를 분류하고, (2)각 문장에 따라 도출되는 어셈블리 명령어의 패턴을 분석하여야 한다. 각 단계마다 수행하는 자세한 내용은 세부 절에서 다룬다.

3.2.1 소스코드를 구성하는 문장 분류

이 논문에서 사용하는 방법은 문장을 컴파일 하였을 때 생성되는 어셈블리 코드가 무엇인지 확인함으로써 문장과 어셈블리 명령어 사이의 관계를 파악하는 방법이다. 효율적으로 대응 관계를 파악하기 위해서 입력으로 사용할 문장을 분류하는 작업이 필요하다. 이 논문에서 사용한 프로그래밍 언어는 C 언어이기 때문에, C 언어를 기준으로 문장을 분류한다. C 언어의 문장은 크게 다음과 같이 분류할 수 있다.

- 표현식
- 분기문
- 반복문
- 함수 및 함수 호출부

표현식은 C 언어에서 제공하는 연산자를 바탕으로 기술된 문장을 의미하며, 분기문은 if문과 switch문을 포함한다. 반복문은 for문, while문 그리고 do while문을 포함한다. 마지막으로 함수 및 함수 호출부는 함수를 선언하거나 그것을 호출하는 문장을 의미한다. 이와 같은 요소들은 C 언어에서 제공하는 연산자와 키워드를 통해 기술된다. 따라서 C 언어의 문장을 분류할 때는 C 언어에서 제공하는 연산자와 키워드를 기준으로 한다. 예를 들어, C 언어에서 제공하는 모든 연산자(대입 연산자, 단항 연산자, 이항 연산자, 삼항 연산자, 캐스팅 연산자 등)를 이용하여 표현식을 작성하고, 이를 입력 문장으로 사용하여야 한다. if문의 경우에는 if와 else라는 두 가지 키워드를 이용하여 if, if-else, if-else if, if-else if-else와 같이 네 가지 형태로 작성할 수 있으므로, 이를 모두 입력 문장으로 사용하여야 한다.

다른 키워드와 달리 자료형 키워드는 독특한 성질을 갖는다. 단독으로 어셈블리 명령어와 관계를 갖지 않지만, 연산자와 결합하는 경우에는 생성되는 어셈블리 명령어에 영향을 준다. 예를 들어, TMS320C67x DSP에서는 short 자료형의 곱셈과 int 자료형의 곱셈을 각각 MPY 명령어와 MPYI 명령어로 변환한다. MPY 명령어는 2 사이클이, MPYI 명령어는 9 사이클이 소요되는 명령어이다. 즉, 동일한 연산자임에도 불구하고 자료형의 차이로 인해 생성되는 명령어가 달라질 수 있다. 또한 생성되는 명령어의 차이로 인해 소요 사이클의 수 역시 달라진다는 것을 알 수 있다. 그러므로 자료형 키워드 역시 입력 문장으로 고려하여야 하며, 특히 연산자와 결합하여 입력 문장으로 사용하여야 한다.

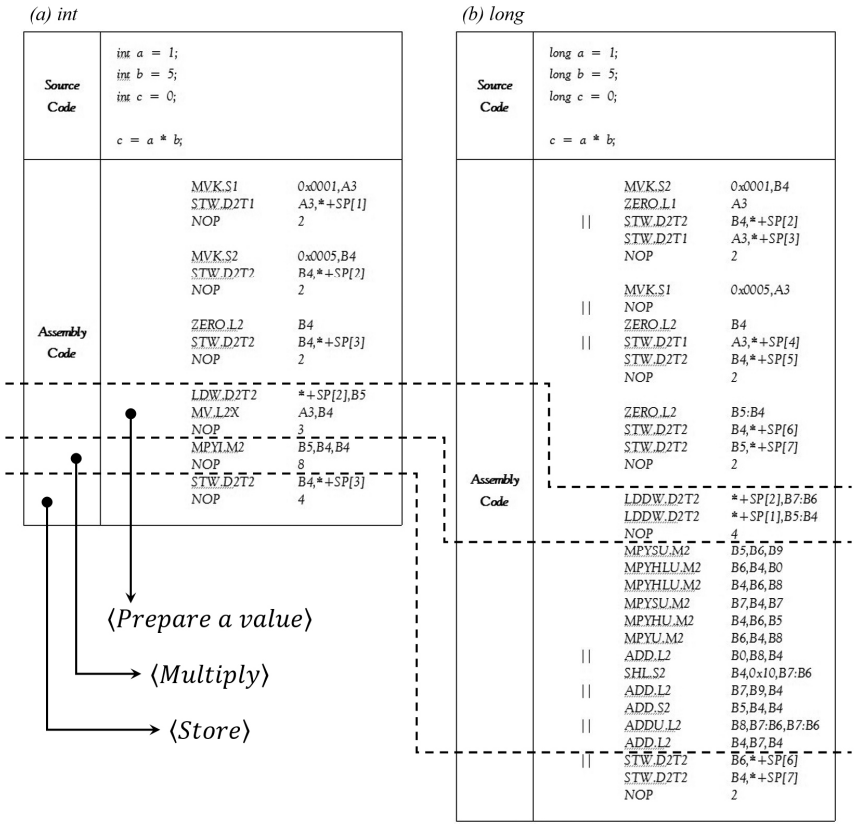
3.2.2 문장과 어셈블리 명령어 사이의 패턴 도출

3.2.1절의 내용을 바탕으로 문장과 어셈블리 코드 사이의 패턴을 도출한다. 문장으로부터 어셈블리 코드를 생성하는 것은 컴파일러를 통해 수행된다. 컴파일러는 어셈블리 코드를 생성할 때 일정한 규칙에 따라 코드를 생성한다. 즉, 유사한 입력에 대해서는 유사한 코드를 생성한다. 따라서 문장과 어셈블리 코드의 대응 관계를 파악할 때, 문장에 대해 생성되는 어셈블리 코드와 직접적인 관계를 설정하는 대신 어셈블리 패턴으로 간접적인 관계를 설정할 수 있다. 이와 같은 관계 설정을 통해 추상적인 실행 시간 측정이 가능해진다.

이 논문에서는 3.2.1절의 내용에서 소개한 단위 문장으로부터 생성되는 어셈블리 코드를 확인하기 위해 총 160 가지의 예제 코드를 작성하였으며, 이를 바탕으로 어셈블리 패턴을 도출한다. 어셈블리 패턴 요소는 160 가지의 예제 코드로부터 생성된 어셈블리 코드를 추상적으로 표현하기 위해 사용되며, 이는 생성된 어셈블리 코드들 사이의 유사한 부분을 추상적으로 표현함으로써 도출할 수 있다. 어셈블리 패턴 요소의 도출 및 사용에 대한 예를 설명하기 위해 본 논문에서 사용한 예제 코드 중 일부를 소개한다.

그림 2는 int 자료형의 곱셈(그림 2(a)), long 자료형의 곱셈(그림 2(b))에 대한 예제 코드와 그로부터 도출된 어셈블리 코드를 보여준다. 그림의 예제 코드는 모두 유사한 어셈블리 코드로 시작된다. 그림 2(a)는 LDW와 MV 명령어를 사용하며, 그림 2(b)는 LDDW 명령어를 사용한다. LDW 명령어는 32비트 크기의 값을 메모리에서 레지스터로 옮기는데 사용하는 명령어이며, LDDW 명령어는 64비트 크기의 값을 메모리에서 레지스터로 옮기는데 사용하는 명령어이다. 여기서 int 자료형은 32비트이고, long 자료형은 48비트이다. 즉, 이 부분은 자료 크기에 맞는 로드(LD) 명령어를 사용하여 피연산자에 해당하는 값을 준비하는 코드이다. 이와 같은 코드는 그림의 예제 코드 외에도 모든 표현식에 등장한다. 어떤 연산이든지 수행하기 위해서는 피연산자가 필수적으로 요구되기 때문이다. 이와 같이 유사한 코드에 대해서 패턴 요소로 추상화하는 작업을 수행하였다. 이 부분은 <Prepare a value>라는 패턴 요소로 추상화된다. 만일, 어떤 단위 문장의 어셈블리 패턴에 <Prepare a value> 패턴 요소가 포함된다면, 이는 LD나 MV 명령어를 통해 필요한 값을 레지스터에 저장한다는 의미가 된다.

다음으로 각 예제 코드에 맞는 연산을 수행한다. 그림 2(a)의 곱셈 연산과 그림 2(b)의 곱셈 연산은 자료형의 차이로 인해 서로 다른 어셈블리 코드를 갖지만 본질적으로 두 곱셈 연산은 동일한 작업을 수행한다. 그림 2(a)의 MPYI는 두 개의 32비트 크기의 값을 곱하는데 사용하는 명령어이다. 이 명령어의 연산 결과로는 64비트 크기의 값이 도출된다. 그림 2(b)의 곱셈 연산은 그림 2(a)에 비해 매우 복잡한 것을 볼 수 있다. 즉, 두 개의 48비트 크기의 값을 곱하는 명령어가 존재하지 않기 때문에, 다수의 명령어를 통해 이를 실현하는 것이다. 어찌되었든 그



(그림 2) 어셈블리 패턴 도출 예제 코드
(Figure 2) Example code for deriving assembly pattern

림 2(a)와 (b)의 어셈블리 명령어는 모두 곱셈 연산을 수행하기 위한 것이다. 또한 각 자료형에 따른 곱셈 연산은 항상 그림과 같은 어셈블리 코드를 통해 수행된다. 따라서 우리는 두 곱셈 연산에 대해서 <Multiply>라는 패턴 요소로 표현할 수 있다. 다만, 같은 패턴 요소라고 할지라도 자료형에 따라 다른 사이클을 가진다는 사실은 염두에 두어야 한다.

그러나 이렇게 패턴 요소만을 도출해서는 소스 코드로부터 실행 시간을 측정할 수 없다. 도출된 패턴 요소가 각각 소요하는 사이클 수를 계산해낼 수 있어야 한다. 어셈블리 패턴은 유사한 어셈블리 코드를 추상화한 것이기 때문에, 각 어셈블리 패턴 요소를 도출할 때 사용한 어셈블리 코드를 통해 어셈블리 패턴 요소의 소요 사이클을 계산할 수 있다. 예를 들어, int 자료형의 <Multiply> 패턴 요소의 경우에는 MPYI 명령어를 통해 도출되었으므로 소요 사이클의 수가 9가 된다. long 자료형의 <Multiply> 패턴 요소의 경우는 그림 2(b)의 어셈블리 코드의 각 명령어의 소요 사이클을 더한 값으로 12가 된다. 이 때, 특정 어셈블리 명령어에 대한 소요 사이클 수는 해당 프로세서의 명령어 명세를 통해 알 수 있다. 이런 식으로 모든 패턴 요소에 대해서 소요 사이클의 수를 계산할 수 있으며, 이 내용을 정리하면 표 1과 같다. 본 논문에서는 이를 사이클 테이블이라고 부른다. 결국, 단위 문장으로부터 도출되는 어셈블리 패턴을 파악하고 어셈블리 패턴을 구성하는 패턴 요소의 소요 사이클의 수를 모두 더함으로써 단위 문장의 소요 사이클의 수를 계산할 수 있다.

앞서 언급한대로 실행 시간을 측정하는데 프로세서로부터 자유로울 순 없기 때문에 사이클 테이블은 각 프로세서마다 준비하여야 한다. 각 프로세서마다 사이클 테이블이 준비되어 있다면, 특정 사이클 테이블을 선택하도록 설정 값만 바꿈으로써 새로운 프로세서에 맞는 WCET 분석을 수행할 수 있다. 이런 이유로 본 논문의

전략은 기존의 방법에 비해 높은 유연성을 갖는 WCET 도구를 구현할 수 있게 해준다.

(표 1) 사이클 테이블
(Table 1) Cycle Table

	char	short	int	long	float	...
Add	1	1	1	1	4	
Subtract	1	1	1	3	4	
Multiply	2	2	9	12	4	
Division	26	34	50	264	128	
Modular	24	32	48	285	-	
And	9	9	9	9	9	
Or	9	9	9	9	9	
Not	2	2	2	2	2	
Equal	1	1	1	2	1	
NotEqual	2	2	2	3	2	
GreatThan	1	1	1	4	1	
GreatOrEqual	2	2	2	5	2	
LessThan	1	1	1	4	1	
LessOrEqual	2	2	2	5	2	
BitwiseAnd	1	1	1	2	-	
BitwiseOr	1	1	1	2	-	
BitwiseXor	1	1	1	2	-	
BitwiseNot	1	1	1	2	-	
ShiftLeft	1	1	1	1	-	
ShiftRight	1	1	1	1	-	
Negate	1	1	1	1	3	
Nop	1	1	1	1	1	
Branch	6	6	6	6	6	
Move	1	1	2	3	2	
Store	3	3	3	6	3	
Load	5	5	5	5	5	

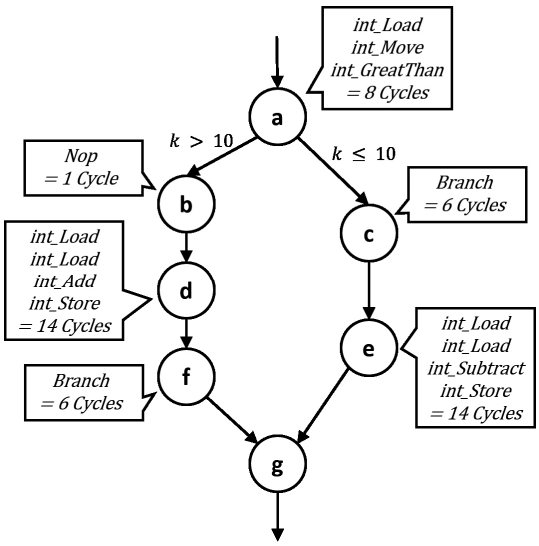
3.3 실행 시간 측정 방법

이번 절에서는 3.2.2절에서 제시한 어셈블리 패턴과 사이클 테이블을 이용해 프로그램의 실행 시간을 측정하는 방법을 설명한다. 우리는 프로그램을 하나의 그래프로 표현할 수 있다. 각 문장은 가중치 노드가 되며, 실행 흐름에 따라 노드와 노드 사이에 에지가 형성된다. 이와 같은 상황에서 프로그램의 실행 경로는 노드의 순열로 표현되며, 해당 경로의 실행 시간은 순열 내에 존재하는 노드의 가중치의 합으로 나타낸다. 예를 들어, 그림 3과 같은 코드로 구성된 프로그램이 있다고 가정한다. 각 문장을 노드로 표현한다면, 그림 3의 코드는 그림 4와 같은 그래프로 구성된다.

그림 4의 그래프는 총 일곱 개의 노드로 구성된다. 이 중에서 a, b, c, f 노드는 if-else 구문으로부터 도출된 노드이며, d, e 노드는 각각 if-else 구문 내부의 표현식으로부터 도출되었다. g 노드는 제어 흐름 상 추가된 노드로 실행 시간 측정에 아무런 영향을 주지 않는다.

```
if (k > 10)
    n = i + j;
else
    n = i - j;
```

(그림 3) 예제 코드
(Figure 3) Example Code



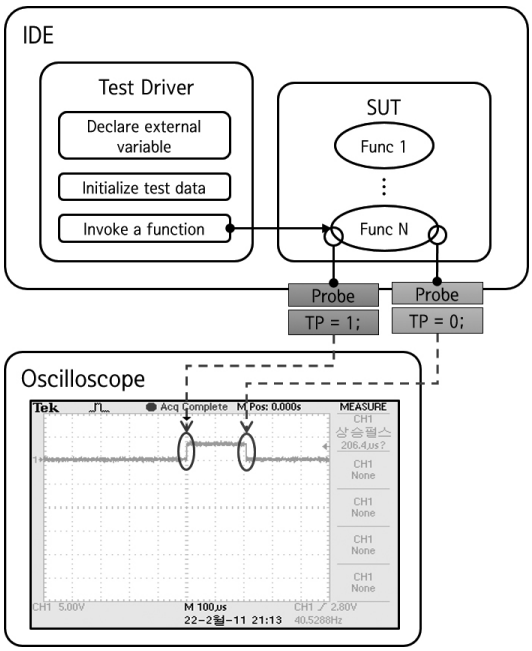
(그림 4) 예제 코드에 대한 그래프
(Figure 4) Graph for Example Code

다음으로 각 노드의 가중치를 계산하여야 한다. 그림 4에서 어셈블리 패턴을 보면 <Branch> 패턴 요소를 볼 수 있는데, 이는 그림 3의 코드에서 if-else 구문의 조건식 결과에 따라 2번째 줄 혹은 4번째 줄로 이동함을 의미한다. 따라서 a 노드에서는 <Branch> 패턴 요소 이전의 패턴 요소가 수행된다. 그러므로 a 노드의 가중치는 <Prepare a value> + <Relational op.> 가 된다. <Prepare a value> 패턴 요소는 조건식의 피연산자를 준비하는데 소요되는 사이클의 수를 의미한다. 변수의 값은 메모리로부터 준비되고 상수의 값은 레지스터로부터 준비되기 때문에 변수의 값을 준비할 때에는 Load가 사용되고 상수의 값을 준비할 때에는 Move가 사용된다. 또한 피연산자의 자료형을 고려하여 각각 int_Load, int_Move만큼의 사이클을 소요한다. <Relational op.> 패턴 요소도 동일한 방

법으로 계산하면 int_GreatThan만큼의 사이클을 소요한다. 따라서 a 노드의 가중치는 int_Load + int_Move + int_GreatThan인 8 사이클이 된다.

각 노드의 가중치를 모두 계산하였다면, 프로그램의 실행 시간을 측정하는 것은 어렵지 않다. 그림 4의 그래프는 두 개의 실행 경로를 가지며, 각각은 {a,b,d,f,g}와 {a,c,e,g}이다. 각 실행 경로에 포함되는 노드의 가중치를 모두 더하면 29 사이클과 28 사이클이 되고, 각각이 실행 경로의 실행 시간이다. 이 결과를 통해 {a,b,d,f,g}가 {a,c,e,g}보다 더 긴 실행 시간을 갖는다는 것을 알 수 있다. 이와 같은 방법으로 실행 시간을 측정할 수 있다.

4. 적용 예제



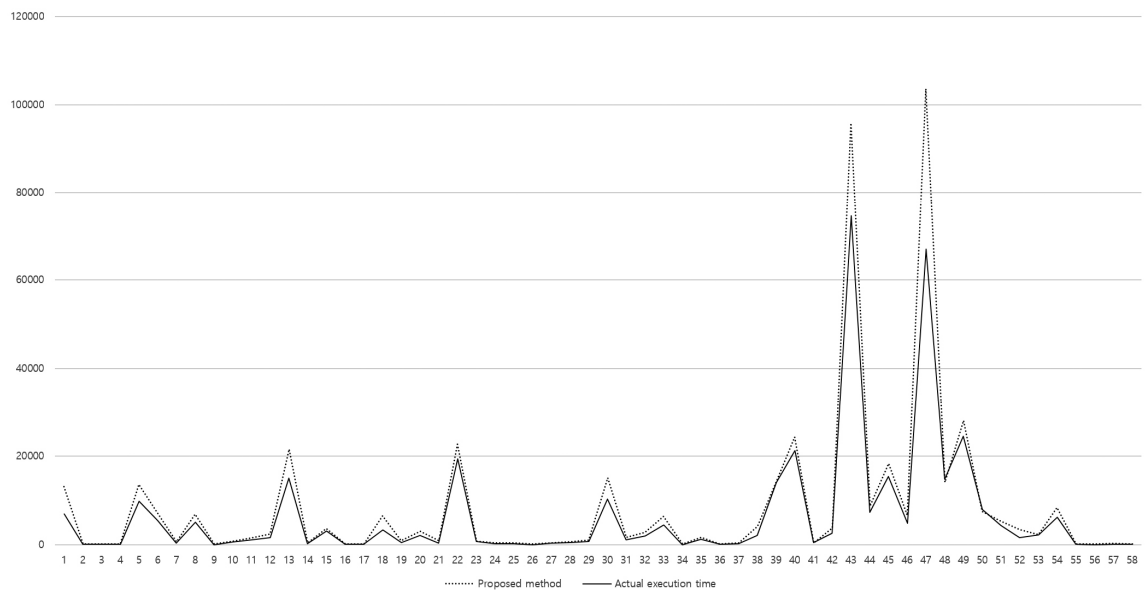
(그림 5) 실제 실행 시간 측정을 위한 실험 환경
(Figure 5) Experimental environment for measuring actual execution time

이번 절에서는 논문에서 제시한 방법으로 측정한 실행 시간과 실제 실행 시간의 비교를 통해 논문에서 제안한 방법의 타당성을 확인한다. 실험을 위해 원자력발전소에서 사용되는 제어 소프트웨어를 사용하였으며, 해당 소프트웨어는 실시간성이 매우 중요한 내장형 소프트웨어이다. 본 논문에서 제시한 방법은 3.3절의 내용과 같이 실행 시간을 측정하였으며, 실제 실행 시간은 제시한 방법과 동일한 경로를 지나도록 데이터를 준비하고 이를 바탕으로 실행 시간을 측정하였다. 실제 실행 시간 측정을 위한 환경은 그림 5와 같이 타겟 보드 위에서 오실로스코프를 이용하여 측정하였다. 그림 5의 IDE는 TI사에서 제공하는 CCS(Code Composer Studio) 3.1이며, 타겟 보드는 TI사의 TMS320C67x DSK(DSP Starter Kit)를 사용하였다.

그림 6은 측정한 결과를 그래프 차트로 표현한 것이다. 점선으로 표시된 것이 제안된 방법으로 측정된 실행 시간이며, 실선으로 표시된 것이 실제 실행 시간이다. x축은 실험에 사용된 소프트웨어의 구성 함수들을 나타내며, y축은 사이클 수를 나타낸다. 그림 6에서 확인할 수 있듯이 제안된 방법으로 측정된 실행 시간은 실제 실행 시간보다 높다. 그 이유는 제안한 방법에서는 컴파일러의 최적화 작업을 고려하지 않은 상태로 실행 시간을 측정하기 때문이다. 실험을 통해 제안된 방법의 측정 시간은 실제 실행 시간의 상한선(upper bound)으로 간주될 수 있고, 또한 소스 코드만으로 소프트웨어의 실행 시간을 측정할 수 있음을 확인할 수 있었다.

5. 결 론

이 논문에서는 기존 연구가 가지고 있는 실행 환경의 종속성 문제를 해결하기 위해 소스코드 기반의 실행 시간 측정 방법을 제시하였다. 이를 위해 기존 연구와 달리 소스코드로부터 제어 흐름 그래프를 생성하였다. 또한 소스코드로부터 생성된 제어 흐름 그래프는 실행 시간과 관련된 정보를 담고 있지 않기 때문에, 이러한 내용을 포함시키기 위해 소스코드와 실행코드 사이의 상관관계를 분석하였다. 이 방법을 이용하면 소스코드와 실행코드 사이의 상관관계 정보, 즉 각 컴파일러 및 프로세서에 적합한 사이클 테이블을 제공함으로써 다양한 실행 환경에 대해 최악실행시간을 측정할 수 있다. 이는 최악실행시간 측정 소프트웨어의 수정 없이도 새로운 실행 환경에 대한 지원이 가능함을 의미한다. 마지막으로 실험을 통해 소스 코드로부터 생성된 제어 흐름 그래프를 이용하여 실행 시간을 측정하는 방법에 대한 타당성도 보였다. 그러나 본 논문의 연구는 실현 가능성에 초점을 두고 연구되었기 때문에, 실제 실행 시간과 어느 정도 격차가 나는 것을 확인하였다. 따라서 향후에는 제시된 방법의 결과의 정확도를 높이는 연구를 진행할 예정이다.



(그림 6) 실험 결과
(Figure 6) Experimental result

참 고 문 헌 (Reference)

[1] R. Wilhelm, J. Engblom, A. Ermedahi et al. "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools", TECS, 2008. http://dx.doi.org/10.1145/1347375.1347389

[2] P. Puschner, R. Nossal, "Testing the results of static worst-case execution-time analysis", In Proc. of the 19th IEEE Real-Time Systems Symposium, pp. 134-143, 1998. http://dx.doi.org/10.1109/REAL.1998.739738

[3] R. Kirner, P. Puschner, et al. "Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation". In WCET'04 Proceedings, 2004. https://www.irisa.fr/manifestations/2004/wcet2004/Papers/Kirner.pdf

[4] D. Sandell, A. Ermedahl, et al. "Static timing analysis of real-time operating system code", In Proc. of the 1st Int'l Symposium on Leveraging Applications of Formal Methods, 2004. http://dx.doi.org/10.1007/11925040_10

[5] N. Holsti, T. Langbacka, S. Saarinen, "Using a worst-case execution-time tool for real-time verification of the DEBIE software", In Proc. of the DASIA 2000, 2000. https://www.researchgate.net/publication/248257120_Using_a_Worst-Case_Execution_Time_Tool_for_Real-Time_Verification_of_the_Debie_Software

[6] P. Puschner, A. Schedl, "Computing maximum task execution times - A graph-based approach", J. Real-Time Syst, vol. 13, no. 1, pp. 67-91, 1997. http://dx.doi.org/10.1023/A:1007905003094

[7] O. Scheid, Autosar Compendium, Part 1: Application & RTE, 1st Ed., CreateSpace Independent Publishing Platform, 2015.

[8] C. Watkins, "Integrated modular avionics: managing the allocation of shared intersystem resources", Proceedings of the 25th Digital Avionics Systems Conference (DASC), 2006. http://dx.doi.org/10.1109/DASC.2006.313743

[9] J.-L. Terraillon, "SAVOIR: Reusing specifications to improve the way we deliver avionics", In Embedded Real Time Software and Systems 2012, 2012.

http://web1.see.asso.fr/erts2012/Site/0P2RUC89/6C-1.pdf
[10] D. McComas, NASA/GSFC's Flight Software Core Flight System, BiblioGov, 2013.
[11] Texas Instruments Inc., SPRU733A, TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide.

◐ 저 자 소 개 ◑

서 용 진 (Yongjin Seo)
2011년 충남대학교 컴퓨터공학과 졸업(학사)
2011년~현재 충남대학교 컴퓨터공학과 (박사 과정, 석박사통합)
관심분야 : 소프트웨어 테스팅, 스마트폰, UX/UI
E-mail : yjseo082@cnu.ac.kr

김 현 수 (Hyeon Soo Kim)
1988년 서울대학교 계산통계학과 졸업(학사)
1991년 한국과학기술원 전산학과 졸업(석사)
1995년 한국과학기술원 전산학과 졸업(박사)
1995년~1995년 한국전자통신연구원 Post Doc.
1996년~2001년 금오공과대학교 조교수
2001년~현재 충남대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어 공학, 소프트웨어 테스팅, 소프트웨어 아키텍처
E-mail : hskim401@cnu.ac.kr