논문 2010-47SP-5-18

소스코드의 분석을 통한 알고리즘 레벨에서의 소프트웨어 복잡도 측정 방법

(The Software Complexity Estimation Method in Algorithm Level by Analysis of Source code)

임 웅*, 남 정 학*, 심 동 규**, 조 대 성***, 최 웅 일***

(Woong Lim, Jung-Hak Nam, Dong-Gyu Sim, Dae-Sung Cho, and Woong-Il Choi)

요 약

프로그램은 실행파일 내의 각 명령어를 수행함으로써 전력을 소비한다. 소비 전력은 복잡도와 비례하기 때문에 프로그램의 복잡도를 측정함으로써 예측될 수 있다. 일반적으로 소프트웨어의 복잡도는 마이크로프로세서 시뮬레이터를 사용하여 측정한다. 그러나 시뮬레이터를 사용한 복잡도 측정방법은 하드웨어를 트랜지스터 레벨과 같은 낮은 레벨에서 모델링하기 때문에 수행시간이 오래 걸리고, 단순히 정량적 측정치만을 제공한다. 본 논문에서는 소프트웨어의 최상위 레벨인 프로그램의 소스코드를 분석하고, 복잡도 매트릭을 생성하여 프로그램 전체에 대한 복잡도를 수식화하여 표현하는 방법을 제안한다. 또한 복잡도 매트릭을 함수 단위로 생성함으로써 연산이 집중되는 모듈에 대한 세분화된 정보를 제공할 수 있다. 제안한 알고리즘의 성능 분석은 게이트 레벨 마이크로프로세서 시뮬레이터인 SimpleScalar와의 비교를 통해서 수행하였다. 분석을 위해 사용된 소프트웨어는 최신 비디오코덱인 H.264/AVC에서 사용되는 4×4 정수변환, 화면 내 예측, 화면 간 예측 모듈이다. 각각의 소프트웨어에 대하여 정량적으로 측정된 성능 분석을 위하여 입력된 각 모듈에 대한 실행 명령어의 수를 비교하였으며, 정확도는 SimpleScalar를 통하여 측정된 시뮬레이션 결과 대비 약 11.6%, 9.6%, 3.5%의 오차를 보였다.

Abstract

A program consumes energy by executing its instructions. The amount of cosumed power is mainly proportional to algorithm complexity and it can be calculated by using complexity information. Generally, the complexity of a S/W is estimated by the microprocessor simulator. But, the simulation takes long time why the simulator is a software modeled the hardware and it only provides the information about computational complexity quantitatively. In this paper, we propose a complexity estimation method of analysis of S/W on source code level and produce the complexity metric mathematically. The function-wise complexity metrics give the detailed information about the calculation-concentrated location in function. The performance of the proposed method is compared with the result of the gate-level microprocessor simulator 'SimpleScalar'. The used softwares for performance test are 4×4 integer transform, intra-prediction and motion estimation in the latest video codec, H.264/AVC. The number of executed instructions are used to estimate quantitatively and it appears about 11.6%, 9.6% and 3.5% of error respectively in contradistinction to the result of SimpleScalar.

Keywords: complexity estimation, software optimization, power consumption

* 학생회원, ** 정회원, 광운대학교 컴퓨터공학과
(Dept. of Computer Engineering, Kwangwoon University)
*** 정회원, 삼성전자(주)
(Samsung Electronics)
※ 본 연구는 "서울시 산학연 협력사업(10560)"의 지원으로 수행되었음.
※ 본 연구는 삼성전자(주)의 지원으로 수행되었음
접수일자: 2009년11월30일, 수정완료일: 2010년7월8일

I. 서 론

최근 모바일 시장이 급속히 확장됨에 따라 다양한 모바일 장치들이 개발되고 있다. 이와 동시에 다양한 형태의 멀티미디어 콘텐츠들이 개발되고 있으며, 사용자들의 요구에 따라, 고효율의 압축 및 고속의 데이터 처

리가 요구되는 높은 복잡도를 수반하는 콘텐츠 들이 증가하고 있다. 이러한 추세에 발맞추어 멀티미디어 데이터를 모바일 환경에서 고속으로 처리하려는 노력이 지속되고 있다. 하드웨어의 빠른 발전에 따라 많은 양의 데이터를 저장하고, 고속으로 처리하는 것이 가능해 졌지만 동시에 고성능 비디오 코덱인 H.264/AVC와 같은 높은 복잡도의 소프트웨어가 지속적으로 개발되고 있으며, 이를 실시간으로 동작시키기 위해서는, 여전히 소프트웨어적인 최적화가 요구되고 있다. 소프트웨어의 최적화는 프로그램의 동작 속도뿐만 아니라, 소비전력과도 높은 관계가 있다. 모바일 장치는 배터리를 사용하기 때문에 휴대시 소비할 수 있는 전력이 제한적이므로, 이러한 모바일 장치에서 동작하는 소프트웨어는 소비전력이 낮아야 하며, 이러한 저전력 프로그램은 복잡도를 낮춤으로써 달성될 수 있다. 특히, H.264/AVC와 같은 높은 복잡도의 최신 비디오 코덱을 지원하는 장치에서는 고속 복호화를 위한 저복잡도, 저전력 구현이 매우 중요하다. 여기서, 복잡도와 소비전력은 높은 상관관계가 존재한다. 하드웨어 상에서의 소비전력은 실행되는 명령어의 수와 비례하게 되는데, 이는 복잡도와 관련이 있다. 따라서 소비전력과 복잡도 둘 중 하나를 구할 수 있다면 다른 하나를 예측할 수 있다. 모바일 장치는 제한된 자원을 가지고 있으며, 전원의 한계로 인하여 모바일 장치에서 실행되는 프로그램의 전력 소비를 최소화하는 것이 매우 중요하다. 이러한 모바일 장치의 특성 때문에 소프트웨어의 소비전력을 측정하는 연구가 활발히 진행되고 있다. 그 예로, 하드웨어 상에서 소비전력 측정을 위한 장치를 사용하여 프로그램 실행간 소비전력을 측정하는 방법과, 시뮬레이터를 통하여 측정하는 방법, 소비전력 모델을 고안하여 측정하는 방법 등이 있다.

컴퓨터 프로그램은 명령어의 연속이며, 하드웨어 상에서는 명령어들이 동작될 때 전력을 소비한다. 이러한 소비 전력은 실행되는 프로그램의 복잡도와 매우 밀접한 관련이 있으며, 복잡도는 또한 프로그램이 동작하는 동안 실행되는 명령어의 수에 비례한다. 이러한 실행 명령어의 수는 프로그램의 알고리즘 레벨 분석을 통해 계산될 수 있다. 본 논문에서 제안하는 방법은 소프트웨어의 소비전력 측정을 위하여 알고리즘 레벨에서 소프트웨어의 복잡도를 예측하는 것이다.

일반적으로 실행 명령어의 수는 하드웨어 시뮬레이터를 사용하여 측정이 가능하다. 그러나 이러한 시뮬레이터들은 속도가 매우 느리고, 단순히 프로그램의 수행간에 실행되는 명령어를 분석하는 것이며, 알고리즘의 분석이라고 할 수 없다. 만일 알고리즘의 분석을 통하여 복잡도가 특히 높은 모듈이나 문맥의 위치를 제공할 수 있다면 최적화 과정을 효과적으로 수행하는데 도움을 줄 수 있을 것이다. 하지만 기존의 시뮬레이터를 통해서 제공되는 정보들은 소프트웨어 개발자에게 어떤 모듈 또는 문맥에서 연산이 집중되는지를 알려줄 수 없다.

프로그램의 개발과정에 있어서 최적화 과정을 통하여 복잡도를 낮출 수 있으며, 이 과정에서 많은 노력과 시간이 수반되고, 알고리즘의 세부적인 분석이 필요하다. 알고리즘의 분석을 통해서 불필요한 연산과, 시간지연이 긴 명령어들을 최소화하여 프로그램의 성능을 높일 수 있기 때문이다. 이러한 최적화 과정에서 높은 복잡도를 갖는 위치를 찾는 것과, 그 위치에서 얼마나 많은 양의 연산이 집중되어 있는가가 계산되어야 한다. 연산의 양은 프로그램 내에 존재하는 순환문과 조건 분기문 등에 의하여 달라질 수 있기 때문에 반드시 소스코드의 길이와 비례한다고 볼 수 없다. 따라서 순환문과 분기문 등이 존재하는 위치와 그 위치에서 실행되는 명령어의 수를 분석해야 한다. 즉, 단순히 순환문의 횟수를 구하는 것은 의미가 없으며, 순환문 내에서 어떤 명령어가 얼마나 많이 반복되는지 알아야 한다.

본 논문에서 제안하는 소프트웨어의 분석을 위한 알고리즘 레벨의 복잡도 측정 방법은 기존의 시뮬레이터와 달리 소프트웨어의 소스 코드 분석을 통하여 함수별 복잡도 매트릭을 구성함으로써, 소프트웨어의 각 기능 모듈별 복잡도를 파악할 수 있는 방법이다. 제안한 방법을 통해서 프로그램의 세분화된 복잡도와 실행 명령어의 수를 예측할 수 있다.

먼저, Ⅱ장에서는 소프트웨어의 소비전력을 구하기 위하여 제안되었던 기존의 연구들과 연관하여 본 논문의 아이디어와 이론적 배경을 설명한다. Ⅲ장에서는 알고리즘 레벨에서의 복잡도 분석 방법 및 결과를 보이고, Ⅳ장에서는 결론 및 향후의 연구 진행방향을 제시한다.

Ⅱ. 소프트웨어의 소비전력 측정을 위한 기존의 방법

기존의 소프트웨어 소비전력 측정 방법은 크게 다음

의 세 가지 방법으로 나눌 수 있다. 하드웨어를 사용하여 직접적으로 측정하는 프로그램 실행 간 소비전력 측정 방법, 소프트웨어로 구현된 시뮬레이터를 사용하여 측정하는 시뮬레이션 기반의 소비전력 측정 방법, 소비전력을 측정하기 위한 요소들을 수학적으로 모델링하여 측정하는 모델기반의 소비전력 측정 방법이다.

1. 프로그램 실행 간 소비전력 측정 방법

프로그램 실행간 소비전력 측정 방법은 하드웨어에 포함되어 있거나, 별도의 외부장치를 사용하여 하드웨어 상에서 프로그램이 동작하는 동안 소비되는 전력을 실제로 측정하는 방법이다.[1~2] 이 방법은 프로그램의 개발 중간 과정에서 마이크로프로세서가 탑재된 보드 등을 사용하여 개발 중인 프로그램을 실행시켜보는 경우에 사용될 수 있다. 일부 보드 중에서 소비전력 측정을 위하여 일정한 주기(샘플링 주파수)로 전류, 전압 등을 측정할 수 있는 별도의 장치가 탑재되어 있는 것도 있다. 그러나 이러한 기능이 제공되지 않는 경우에는 별도의 소비전력 측정을 위한 장치를 연결하여 사용할 수 있다.

프로그램 실행간 소비전력 측정 방법에서 요구되는 측정 장치는 프로그램의 실행 간에 소비되는 전류, 전압 등에 대한 값을 모든 시간에 대하여 측정하는 것이 아니라, 주기적으로 각 요소들에 해당하는 값을 측정하여 제공하기 때문에 소비전력에 대한 완벽한 측정치를 제공한다고 할 수는 없다. 하지만 측정 장치가 갖는 정확도 내에서 가장 간단하고 정확하게 소비전력을 측정할 수 있는 방법 중의 하나라고 할 수 있다.

프로그램 실행간 소비전력 측정을 위해서는 그러한 기능을 제공하는 추가적인 장치가 필요하며, 이러한 장치는 모든 프로세서 플랫폼에 제공되거나 탑재되어 있는 것은 아니다. 이러한 방법은 전력 측정 장치에 대한 추가적인 비용이 따르는 단점이 있다.

2. 시뮬레이션 기반의 소비전력 측정 방법

프로그램의 소비전력 측정을 위한 또 다른 방법은 시뮬레이터를 사용하는 방법이다.[3, 5] 시뮬레이터는 특정 프로세서와 주변장치를 모델링하여 구현한 소프트웨어이다. 시뮬레이터를 사용한 측정 방법의 장점은 다양한 정보의 제공에 있다. 시뮬레이터는 하드웨어의 하위레벨인 트랜지스터 레벨 혹은 게이트 레벨까지 매우 자세하게 모델링하며, 메모리와 같은 주변장치 또한 유연하게 설정할 수 있기 때문에 다양한 상황에 대한 다양한 정보를 제공할 수 있다. 또한, 소프트웨어적으로 구현되므로 프로그램이 실행되는 모든 단계에 대한 정보를 프로파일링할 수 있다. 일반적으로 시뮬레이터는 시뮬레이터가 실행시킨 프로그램의 실행시간과 실행된 명령어의 수, 각 단계별 메모리 접근 횟수 및 지연시간 등의 다양하고 세분화된 정보를 제공한다. 시뮬레이터를 사용하면 유용한 정보를 손쉽게 측정할 수 있지만, 일반적으로 특정 프로세서에 의존적이므로, 범용적이지 않다. 이는 각 프로세서 제조사별로 제작하는 프로세서의 구조가 다르기 때문이며, 대부분의 경우 이러한 정보를 제공하지 않기 때문에 구현이 매우 어렵다.

비교적 널리 알려진 마이크로프로세서 시뮬레이터로 SimpleScalar가 있다.[4] 이는 무료 공개 소프트웨어이면서 소스 코드까지 제공된다. SimpleScalar는 마이크로프로세서를 게이트 레벨에서 모델링한 시뮬레이터이면서, 다양한 구조의 마이크로프로세서를 구성하여 시뮬레이션할 수 있도록 옵션을 통하여 많은 주변장치와 명령어 처리에 관한 정책 등을 변경할 수 있도록 구현되어있다. 옵션 중에는 데이터 캐시 및 명령어 캐시의 사이즈, 각 메모리에 연결되는 포트의 사이즈, 명령어 패치 속도, out-of-order프로세싱, 분기 예측기 등의 매우 다양한 사항들을 사용자가 임의로 설정하여 시뮬레이션할 수 있도록 되어있다. 또한 SimpleScalar에 추가적으로 소비전력 측정을 위한 모듈을 포함시킨 Sim-PAnalyzer를 사용하여 프로그램이 해당 프로세서에서 동작되는 동안의 소비전력을 측정할 수 있다.

SimpleScalar나 Sim-PAnalyzer는 다양한 옵션을 통해서 사용자에게 상당한 유연성을 제공할 수 있지만, 실제적으로 특정 프로세서를 완벽하게 모델링하는 데에는 한계가 있다. 이 외에 ARMulator와 같은 상용제품이 있지만, SimpleScalar와 Sim-PAnalyzer를 포함한 이러한 시뮬레이터들은 대부분 실행속도가 매우 느리다는 단점이 있다.

3. 모델 기반의 소비전력 측정 방법

모델 기반의 소비전력 측정 방법은 시뮬레이션을 통해 얻을 수 있는 명령어의 실행 횟수, 메모리 hit/fault, 파이프라인 stall, 등의 요소를 사용하여 수학적으로 모델링한 매트릭에 적용함으로써 소비전력을 구할 수 있다.[6~9] 프로세서가 프로그램을 실행시키는 데에 있어서 전력을 가장 많이 소비하는 부분은 명령어의 실행이다.

명령어가 메모리로부터 패치되어, 계산되고, 결과를 처리하는 파이프라이닝 과정에서 대부분의 에너지가 소비되며, 이 과정에서 발생하는 메모리 fault나 stall 등에 의해서 부수적인 에너지의 소비가 발생한다. 따라서 다음의 수식 (1)과 같은 방법으로 에너지 소비를 모델링 할 수 있다.

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \quad (1)$$

수식 (1)에서 E_p는 프로그램의 실행 전체에서 소비되는 에너지이며, B_i는 i번째 명령어 한 개가 실행될 때 기본적으로 소비되는 에너지, N_i는 i번째 명령어가 실행된 횟수이다. 다양한 종류의 명령어가 실행될 때 동일한 명령어가 연속될 수도 있고, 서로 다른 명령어가 실행될 때도 있다. 서로 다른 명령어가 연속하여 실행될 때에는 회로의 상태변화에 따른 추가적인 에너지 소비가 발생하며, 이를 $O_{i,j}$로 표현하고, $N_{i,j}$는 서로 다른 명령어가 실행된 횟수를 표현한다. 마지막으로 E_k는 프로그램의 실행 간에 발생하는 캐시 miss, 파이프라인 stall과 같은 부수적인 에너지 소비 요소이다.

수식 (1)에 제시된 매트릭을 통해 프로그램이 소비하는 에너지를 구하기 위해서는 B_i, $O_{i,j}$, E_k에 대한 각각의 소비에너지를 측정해야하고, 발생횟수인 N_i, $N_{i,j}$는 해당 프로그램 프로파일러나 캐시 시뮬레이터 등을 통해서 얻을 수 있다.

모델기반의 소비전력 측정 방법은 비교적 단순한 수학적 모델을 통하여 프로그램의 소비전력을 측정할 수 있지만, B_i, $O_{i,j}$, E_k와 같은 소비에너지 요소들은 앞서 설명한 프로그램 실행간 소비전력 측정이나 시뮬레이션 기반의 소비전력 측정 방법을 사용하여 해당 명령어 아키텍쳐에 따른 모든 명령어에 대한 값을 측정해야 하는 어려움이 있다.

Ⅲ. 제안하는 알고리즘 레벨 복잡도 측정 방법

Ⅱ장에서 설명한 기존의 소비전력 측정방법들은 하드웨어에 대한 높은 비용이나, 소프트웨어의 구현, 소비전력 측정을 위한 실험의 관점에서 효율성과, 다양성이 떨어진다. 즉, 높은 비용이 수반되거나, 다양한 종류의 마이크로프로세서에 대한 실험이 힘들다는 단점이 있다.

본 논문의 서론에서 언급한 것과 같이 프로세서의 소비전력은 실행되는 명령어의 수에 비려한다. 즉, 동일한 프로그램의 입력에 대하여 입력된 프로그램에 대한 실행명령어의 수를 알 수 있다면, 소비전력을 상대적으로 예측할 수 있다. 이는 기존의 소비전력 측정방법과 동일한 프로그램 입력에 대하여 실행될 명령어를 분석함으로써 달성될 수 있다.

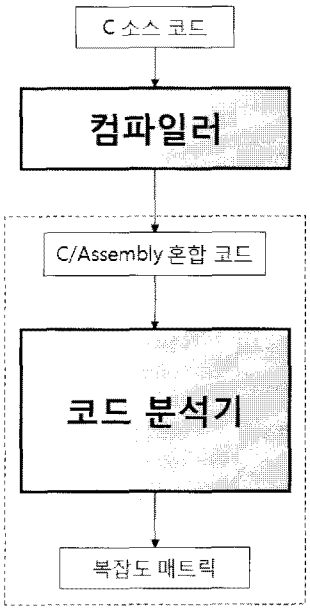

그림 1. 제안하는 알고리즘의 전체적인 구조
Fig. 1. The general structure of the proposed algorithm.

본 논문에서는 제안하는 알고리즘 레벨 복잡도 측정 방법은 소스 코드의 컴파일 과정에서 얻어지는 C와 어셈블리어가 혼합된 코드를 분석하여 복잡도 매트릭을 생성한다. 그림 1은 제안하는 알고리즘의 전체적인 구조를 보여준다.

그림 1의 컴파일 과정에서 C컴파일러를 통해서 C/Assembly 혼합 코드가 생성된다. 이를 입력으로 분석하여 프로그램의 복잡도 매트릭을 생성한다. 본 논문에서는 ARM 명령어 아키텍쳐를 바탕으로 하여 구현 및 실험을 수행하였다. C/Assembly 혼합 코드를 파일로 저장하기 위해서 ARM gcc 크로스 컴파일러를 사용할 경우 다음의 옵션을 사용하여 컴파일하면 된다.

```
arm-linux-gcc -c -g -Wa,-a,-ahl=file.lst file_name.c
```

C/Assembly 혼합 코드의 예는 그림 2와 같다.

그림 2와 같이 C/Assembly 혼합 코드에는 C 소스 코드와 어셈블리어가 함께 존재하며, 각 C 소스 코드 라인마다 그에 해당하는 어셈블리 명령어들이 나열된

```
245   32:sixtab_C.c     ****      int i, j, t;
246   196                 .stabn 68,0,32,.LM9-six_tab
247   197                 .LM9:
248   198                 .LBB3:
249    33:sixtab_C.c     ****
250    34:sixtab_C.c     ****      int row = ROW * 4 + 17;
251   199                 .stabn 68,0,34,.LM10-six_tab
252   200                 .LM10:
253   201 0060 24339FE5      ldr r3, .L66
254   202 0064 20300BE5      str r3, [fp, #-32]
255    35:sixtab_C.c     ****      int col = COL * 4 + 17;
256   203                 .stabn 68,0,35,.LM11-six_tab
257   204                 .LM11:
258   205 0068 20339FE5      ldr r3, .L66+4
259   206 006c 24300BE5      str r3, [fp, #-36]
260    36:sixtab_C.c     ****
261    37:sixtab_C.c     ****      for (i = 0 ; i < 8 ; i++)
262   207                 .stabn 68,0,37,.LM12-six_tab
263   208                 .LM12:
264   209 0070 0030A0E3       mov r3, #0
265   210 0074 14300BE5       str r3, [fp, #-20]
266   211                 .L6:
267   212 0078 14301BE5       ldr r3, [fp, #-20]
268   213 007c 070053E3       cmp r3, #7
269   214 0080 200000DA       ble .L9
270   215 0084 3D0000EA       b   .L7
```

그림 2. C/Assembly 혼합 코드의 예
Fig. 2. An example of C/Assembly combination code.

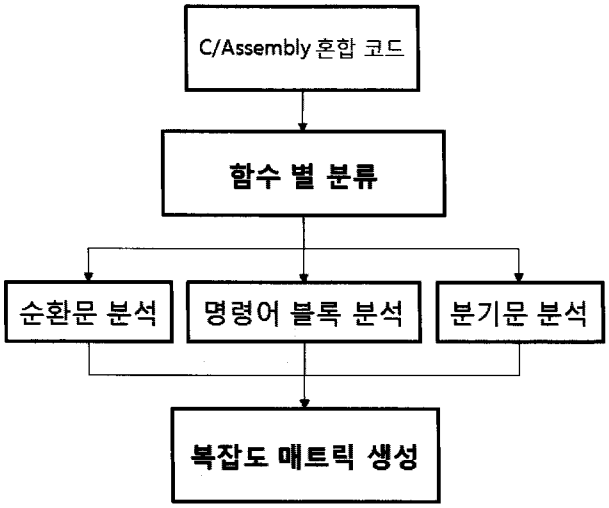

그림 3. C/Assembly 혼합 코드 분석기의 구조
Fig. 3. The structure of C/Assembly combination code analyzer.

다. 입력된 C/Assembly 혼합 코드를 분석하여 복잡도 매트릭을 구성하는 방법은 다음 그림 3과 같이 크게 네 가지 하위요소로 구분된다.

먼저, 프로그램 내에 정의된 각 모듈(함수)을 분류한다. 분류된 함수 내에서 순차적으로 수행되는 명령어들을 하나의 블록으로 묶고, 각각의 순환문과 분기문을 분석한다. 즉, 프로그램 전체로부터 각 모듈의 하위 단계로의 분석을 수행하여 복잡도 매트릭을 생성한다.

1. 프로그램 내 모듈(함수)별 분류

C/Assembly 혼합 코드 내에는 C 소스 코드에 대한 모든 내용이 어셈블리 명령어로 기술되어 있다. 프로그램 내에 존재하는 함수의 시작위치 또한 특정 규칙을

```
131   122                 .stabs "main:F(0,1)",36,0,5,main
132   123                     .global main
133   124                     .type    main,function
134   125                 main:
135     1:paper.c       **** #include<stdio.h>
136     2:paper.c       ****
137     3:paper.c       **** int
138     4:paper.c       **** main(void)
139     5:paper.c       **** {
140   126                 .stabn 68,0,5,.LM1-main
141   127                 .LM1:
142   128                     @ args = 0, pretend = 0, frame = 20
143   129                     @ frame_needed = 1, current_functio
144   130 0000 0DC0A0E1       mov ip, sp
145   131 0004 00D82DE9       stmfd   sp!, {fp, ip, lr, pc}
146   132 0008 04B04CE2       sub fp, ip, #4
147   133 000c 14D04DE2       sub sp, sp, #20
148     6:paper.c       ****      int i, j;
```

그림 4. C/Assembly 혼합 코드 내 함수 시작의 예
Fig. 4. An example of start position of function in C/Assembly combination code.

통해 기술되어 있기 때문에 각 함수별 분석이 가능하다. 다음 그림 4는 C/Assembly 코드 내에서 함수의 시작 위치를 보여준다.

그림 4의 표시된 영역 내의 어셈블리 코드를 통하여 'main'이라는 'function'이 시작됨을 알 수 있다. 함수의 시작으로부터 끝나는 위치는 중괄호('{}')의 개수를 통해서 알 수 있다. 중괄호는 함수의 시작에서 열리고 함수의 끝에서 닫힌다. 중괄호는 함수 내에서도 사용되지만, 열린 중괄호의 수와 닫힌 중괄호의 수는 반드시 같아야하기 때문에 함수의 시작에서 열린 중괄호와 짝이 되는 중괄호를 간단히 찾음으로써 함수의 끝을 알 수 있다.

2. 명령어 블록의 분석

명령어 블록은 함수, 루프 또는 분기문을 경계로 구분한 어셈블리 명령어들의 집합이다. 이는 복잡도 매트릭 분석의 가장 기본 단위가 된다.

명령어 블록의 예는 그림 5와 같으며, 프로그램의 코드에서 분석되는 순서에 의하여 n번째 명령어 블록을

```
169    12:paper.c       ****      num1++;
170   149                 .stabn 68,0,12,.LM6-main
171   150                 .LM6:
172   151 0028 18301BE5       ldr r3, [fp, #-24]
173   152 002c 012083E2       add r2, r3, #1
174   153 0030 18200BE5       str r2, [fp, #-24]
175    13:paper.c       ****      num2++;
176   154                 .stabn 68,0,13,.LM7-main
177   155                 .LM7:
178   156 0034 1C301BE5       ldr r3, [fp, #-28]
179   157 0038 012083E2       add r2, r3, #1
180   158 003c 1C200BE5       str r2, [fp, #-28]
181 ^LARM GAS  /tmp/ccUBCVBX.s                  page 4
182
183
184    14:paper.c       ****      num3++;
185   159                 .stabn 68,0,14,.LM8-main
186   160                 .LM8:
187   161 0040 20301BE5       ldr r3, [fp, #-32]
188   162 0044 012083E2       add r2, r3, #1
189   163 0048 20200BE5       str r2, [fp, #-32]
```

그림 5. C/Assembly 혼합 코드 내의 명령어 블록의 예
Fig. 5. An example of the instruction block in C/Assembly combination code.

```
191   16:paper.c        ****      for(i = 0 ; i < 10 ; i++)
192   164               .stabn 68,0,16,.LM8-main
193   165               .LM9:
194   166 004c 0030A0E3     mov r3, #0
195   167 0050 10300BE5     str r3, [fp, #-16]
196   168               .L3:
197   169 0054 10301BE5     ldr r3, [fp, #-16]
198   170 0058 090053E3     cmp r3, #9
199   171 005c 170000DA     ble .L6
200   172 0060 240000EA     b   .L4
201   173               .L6:
202    17:paper.c       ****      {
203    18:paper.c       ****          num1++;
204   174               .stabn 68,0,18,.LM10-main
205   175               .LM10:
206   176 0064 18301BE5     ldr r3, [fp, #-24]
207   177 0068 012083E2     add r2, r3, #1
208   178 006c 18200BE5     str r2, [fp, #-24]
209    19:paper.c       ****          num2++;
210   179               .stabn 68,0,19,.LM11-main
211   180               .LM11:
212   181 0070 1C301BE5     ldr r3, [fp, #-28]
213   182 0074 012083E2     add r2, r3, #1
214   183 0078 1C200BE5     str r2, [fp, #-28]
215    20:paper.c       ****          num3++;
216   184               .stabn 68,0,20,.LM12-main
217   185               .LM12:
218   186 007c 20301BE5     ldr r3, [fp, #-32]
219   187 0080 012083E2     add r2, r3, #1
220   188 0084 20200BE5     str r2, [fp, #-32]
221   189               .stabn 68,0,16,.LM13-main
222   190               .LM13:
223   191               .L5:
224   192 0088 10301BE5     ldr r3, [fp, #-16]
225   193 008c 012083E2     add r2, r3, #1
226   194 0090 10200BE5     str r2, [fp, #-16]
227   195 0094 130000EA     b   .L3
228   196               .L4:
229    21:paper.c       ****      }
```

그림 6. 명령어 블록 구분의 예
Fig. 6. An example of division of block.

B_n으로 표시한다. 그림 5의 표시된 세 영역은 각 영역의 상단에 존재하는 C코드에 대응되는 어셈블리 명령어 집합이며, 표시된 각각의 영역을 합하여 하나의 명령어 블록으로 정의한다. 이 세 영역을 하나의 블록으로 처리하는 이유는 세 블록 모두가 한 번씩 처리되기 때문이다. 세 영역을 서로 다른 명령어 블록들로 처리할 경우 블록의 개수가 매우 많아져, 본 논문에서 제안하는 알고리즘을 통해 생성되는 복잡도 매트릭이 소스코드의 길이가 길어짐에 따라서 너무 복잡해질 수 있기 때문이다. 다음 그림 6의 명령어 블록은 그림 5의 명령어 블록과 연속한 소스 코드의 일부이다. 표시된 영역 중 첫 번째와 두 번째는 순환문과 관련된 명령어 블록이다. 이 두 블록의 처리는 다음절의 순환분 분석에서 다루도록 한다. 나머지 세 개 블록은 순환문 내에 있으므로 그림 5의 블록과 구분되어야 한다.

즉, 그림 5의 명령어 블록은 1번만 실행되는 블록이고, 그림 6의 명령어 블록은 순환문의 반복 횟수에 따라서 실행되는 횟수가 결정되므로 두 블록을 구분해야 한다. 이러한 명령어 블록은 분기문에서도 구분되어야 하며, 결론적으로 명령어 블록은 순환문과 분기문을 기준으로 구분되어야 하며, 구분된 영역 내에서 합쳐져야 한다.

3. 순환문의 분석

프로그램의 복잡도에서 가장 중요한 요소 중 하나는 순환문이다. 순환문의 반복 횟수는 가능한 한 줄이는 것이 좋지만, 많은 경우에 필요한 반복 횟수가 고정적이다. 또한, 순환문 자체를 위한 명령어들은 대부분의 경우에는 복잡도에 큰 영향을 미치지 않는다. 대신, 순환문 내에서 반복되는 명령어가 차지하는 비중은 경우에 따라서 프로그램 자체의 성능에 큰 영향을 미칠 수 있다. 그러므로 순환문 내에서 실행되는 명령어 블록은 그 크기나 반복횟수에 따라서 프로그램 최적화의 중요한 요인이 될 수 있다.

복잡도 매트릭에서 n번째 순환문에 대한 반복 횟수를 N_n으로 표시한다. 예를 들어, 순환문 내의 명령어 블록이 존재하면 $N_n \times B_m$이 되어 B_m 내의 명령어들 각각이 N_n번씩 실행됨을 알 수 있다.

순환문으로 주로 사용되는 for문과 while문은 어셈블리어 레벨에서 차이가 있다. for문은 일반적으로 변수의 초기화, 종료조건, 반복문(증감문)으로 구성된다. 이 세 가지 중 초기화는 순환문의 시작에서 한 번만 수행되며, 종료조건 비교와 반복문은 순환문의 순환 횟수만큼 반복된다. for문의 어셈블리 명령어 코드는 그림 7을 통해서 알 수 있는데, 단순하게 그림 7과 같이 표현할 수 있다.

그림 6의 첫 번째 표시 영역은 해당 for문의 변수 I에 대한 초기화 명령어들이고, 두 번째 표시 영역은 순환문의 종료 조건을 확인하는 명령어들에 해당된다. 그림 6과 7을 통해서 알 수 있듯이 for 순환문에서 초기화에 해당하는 명령어들은 순환문 내에 포함되지 않

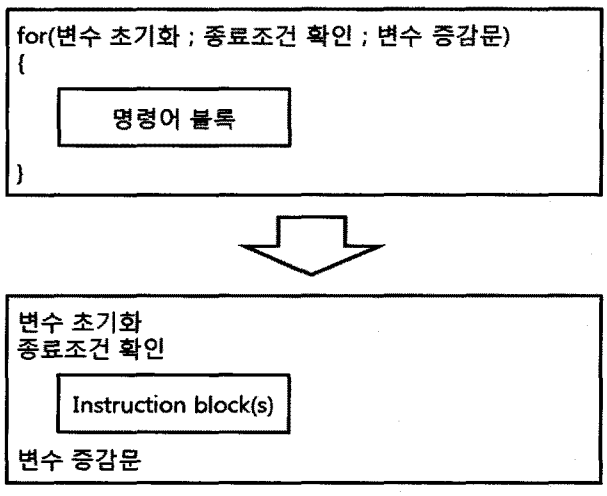

그림 7. for문의 단순화 된 어셈블리 구조
Fig. 7. The simplified assembly frame of for loop.

```
197   18:paper_while.c ****      while(i < 10)
198  168                    .stabn 68,0,18,.LM10-main
199  169                    .LM10:
200  170                    .L3:
201  171 0054 10301BE5          ldr r3, [fp, #-16]
202  172 0058 090053E3          cmp r3, #9
203  173 005c 170000DA          ble .L5
204  174 0060 240000EA          b   .L4
205  175                    .L5:
206   19:paper_while.c ****      {
207   20:paper_while.c ****          num1++;
208  176                    .stabn 68,0,20,.LM11-main
209  177                    .LM11:
210  178 0064 18301BE5          ldr r3, [fp, #-24]
211  179 0068 012083E2          add r2, r3, #1
212  180 006c 18200BE5          str r2, [fp, #-24]
213   21:paper_while.c ****          num2++;
214  181                    .stabn 68,0,21,.LM12-main
215  182                    .LM12:
216  183 0070 1C301BE5          ldr r3, [fp, #-28]
217  184 0074 012083E2          add r2, r3, #1
218  185 0078 1C200BE5          str r2, [fp, #-28]
219   22:paper_while.c ****          num3++;
220  186                    .stabn 68,0,22,.LM13-main
221  187                    .LM13:
222  188 007c 20301BE5          ldr r3, [fp, #-32]
223  189 0080 012083E2          add r2, r3, #1
224  190 0084 20200BE5          str r2, [fp, #-32]
225   23:paper_while.c ****
226   24:paper_while.c ****          i++;
227  191                    .stabn 68,0,24,.LM14-main
228  192                    .LM14:
229  193 0088 10301BE5          ldr r3, [fp, #-16]
230  194 008c 012083E2          add r2, r3, #1
231  195 0090 10200BE5          str r2, [fp, #-16]
232   25:paper_while.c ****      }
```

그림 8. while문의 단순화 된 어셈블리 구조
Fig. 8. The simplified assembly frame of while loop.

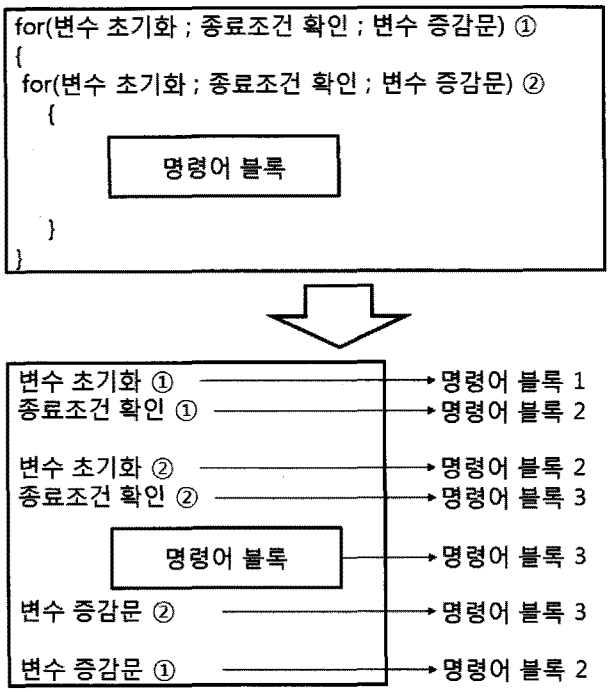

그림 9. 다중 for문의 예
Fig. 9. The example of multiple nested for loop.

아야 하며, 종료 조건 비교와 반복문은 순환문 내의 명령어 블록에 포함되어야 한다.

for문과 비교하여 while문은 비교문 만을 통하여 순환의 여부가 결정되므로 순환문 내에 초기화 영역이 없다. while문의 예는 그림 8과 같다.

그림 8의 코드는 그림 6의 for문을 단순히 while문으로 대체한 코드이다. 표시된 영역이 비교문을 수행하는 부분이며, 이 영역은 while문의 반복 횟수만큼 수행된다. 따라서 while문의 경우는 while문 내부의 모든 명령어들을 하나의 명령어 블록으로 처리할 수 있다.

비디오 데이터와 같은 이차원 데이터를 처리하거나, 컨벌루션과 같은 연산이 사용되는 소프트웨어에서는 다중 순환문 구조가 많이 사용된다. 위에서 설명한 일중 순환문의 경우와는 다르게 다중 순환문은 순환문 분석의 중간에서 또 다른 순환문이 나오게 된다. 이러한 경우, 내부에 존재하는 순환문의 분석을 시작하기 전에 이전에 분석 중이던 순환문에 대한 정보를 저장해야한다.

새로운 순환문의 분석이 끝나게 되면, 저장되어 있던 이전의 순환문에 대한 정보를 꺼내어 남은 코드에 대한 분석을 계속한다. 이와 같은 다중 순환문의 분석은 가장 바깥쪽의 순환문이 가장 마지막에 완료되기 때문에 스택을 사용하여 분석 중이던 순환문에 대한 정보를 저장하는 것이 적절하다. 그림 9는 for문을 사용한 이중 순환문의 구조를 보여준다.

그림 9에서 예로 든 상단의 C코드는 이중 순환문 구조를 가진다. 바깥쪽에 있는 1번 순환문에 대하여 내부에 있는 2번 순환문은 1번 순환문의 순환횟수 만큼 반복된다. 따라서 2번 순환문의 내부에 있는 명령어 블록은 1번 순환문의 반복횟수 N_1과 2번 순환문의 반복횟수 N_2에 대하여 $N_1 \times N_2$번 실행된다. 그림 7의 하단의 어셈블리어 레벨에서 분석을 하면 다음과 같다. 가장 먼저 1번 순환문에 대한 초기화 구문은 한 번만 수행되므로 명령어 블록1로 구분된다.

다음으로 1번 순환문에 대한 조건 비교가 수행되는데, 이는 1번 순환문의 순환횟수만큼 실행되므로 명령어 블록2가 새로이 생성된다.

이어서 새로운 2번 순환문이 시작되므로 현재까지의 상태를 스택에 저장하고 2번 순환문의 분석을 시작한다. 2번 순환문의 초기화 구문은 1번 순환문의 순환횟수만큼 실행되므로 기존의 명령어 블록 2번에 포함시킨다. 다음으로 나오는 2번 순환문에 대한 조건 비교문을 새로운 명령어 블록 3번을 생성하여 포함시킨다. 2번 순환문의 내부에 존재하는 명령어 블록은 2번 순환문의 조건 비교문과 동일한 횟수만큼 실행되므로 기존의 명령어 블록 3번에 추가한다. 2번 순환문에 대한 변수값

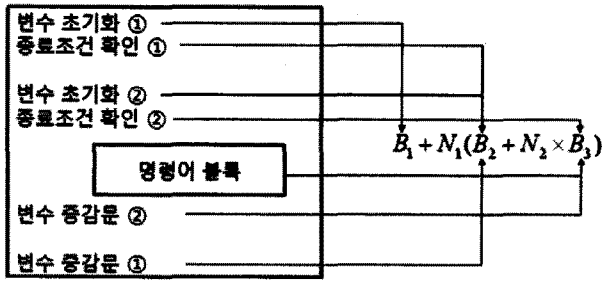

그림 10. 다중 for문의 복잡도 분석과 결과 매트릭의 예
Fig. 10. The example of complexity analysis of multiple nested for loop and complexity metric.

증가 구문 또한 2번 순환문의 순환횟수만큼 수행되므로 명령어 블록 3번에 추가된다. 2번 순환문의 분석이 완료되었으므로 스택으로부터 아직 완료되지 않은 1번 순환문에 대한 정보를 꺼내오고, 1번 순환문에 대한 변수 증가 구문이 명령어 블록 2번에 추가된다. 이러한 방식으로 다중 순환문을 분석할 수 있으며, 그림 9에서 제시된 이중 for문에 대한 매트릭은 그림 10과 같이 표현될 수 있다. 이와 같은 방법으로 명령어 블록의 수를 최소화 하면서 동시에 정확하게 다중 루프를 분석할 수 있다.

4. 분기문의 분석

일반적인 프로그램에서 특정 조건을 만족할 경우에 특정 동작을 수행하도록 하기 위해서 분기문이 사용된다. 이는 분기가 발생할 확률에 따라서 특정 명령어들

```
231   23:paper.c        ****      if(i > 5)
232  197                 .stabn 68,0,23,.LM14-main
233  198                 .LM14:
234  199 0098 10301BE5       ldr r3, [fp, #-16]
235  200 009c 050053E3       cmp r3, #5
236  201 00a0 300000DA       ble .L7
237   24:paper.c        ****      {
238   25:paper.c        ****          num1++;
239  202                 .stabn 68,0,25,.LM15-main
240  203                 .LM15:
241 ^LARM GAS  /tmp/cc8Uxnua.s          page 5
242
243
244  204 00a4 18301BE5       ldr r3, [fp, #-24]
245  205 00a8 012083E2       add r2, r3, #1
246  206 00ac 18200BE5       str r2, [fp, #-24]
247   26:paper.c        ****          num2++;
248  207                 .stabn 68,0,26,.LM16-main
249  208                 .LM16:
250  209 00b0 1C301BE5       ldr r3, [fp, #-28]
251  210 00b4 012083E2       add r2, r3, #1
252  211 00b8 1C200BE5       str r2, [fp, #-28]
253   27:paper.c        ****          num3++;
254  212                 .stabn 68,0,27,.LM17-main
255  213                 .LM17:
256  214 00bc 20301BE5       ldr r3, [fp, #-32]
257  215 00c0 012083E2       add r2, r3, #1
258  216 00c4 20200BE5       str r2, [fp, #-32]
259  217                 .L7:
260   28:paper.c        ****      }
```

그림 11. 분기문의 C/Assemnly 코드 예
Fig. 11. The example of conditional branch in C/Assembly combination code.

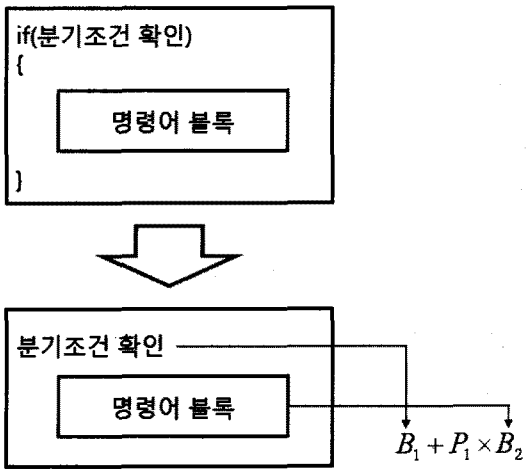

그림 12. 분기문의 단순화된 어셈블리 구조와 생성된 복잡도 매트릭
Fig. 12. The simplified assembly frame of conditional branch and complexity metric.

이 수행되거나 그렇지 않을 수 있음을 의미한다. 따라서 분기문을 기준으로 명령어 블록이 나뉘어져야 하며, 제안하는 알고리즘 레벨 복잡도 분석 방법에서는 n번째 분기분에 대하여 P_n을 사용하여 표현한다. 다음 그림 11은 분기문에 대한 C/Assembly 혼합 코드이며, 그림 12는 분기문의 단순화된 어셈블리 구조에 대하여 제안한 알고리즘을 통해 생성된 복잡도 매트릭의 예이다.

그림 11의 분기문 내의 표시된 영역인 조건 비교문은 분기의 여부에 상관없이 수행된다. 그러나 분기문 내의 명령어 블록은 분기 조건을 만족할 경우에 대하여 수행된다. 즉, 조건 비교문과 분기문 내의 명령어 블록은 실행되는 횟수가 다를 수 있기 때문에, 조건 비교문은 분기문 이전의 명령어 블록에 포함되어야 하며, 분기문 내의 명령어 블록은 새로운 명령어 블록으로 생성되어야한다.

5. 알고리즘 레벨 복잡도 매트릭 구성

본 논문에서 제안한 알고리즘 레벨 복잡도 측정 방법은 C/Assembly 혼합 코드의 분석을 통하여 복잡도 매트릭을 생성하는 것이다. 개발된 소프트웨어가 마이크로프로세서 상에서 실행될 때, 프로세서는 컴파일된 결과로 생성된 기계어 시퀀스를 정해진 규칙에 의해 하나씩 실행시킨다. 이는 어셈블리 명령어를 실행시키는 것과 동일하기 때문에 어셈블리 코드를 분석함으로써 프로세서가 개발된 소프트웨어를 동작시키기 위해서 얼마나 많은 명령어를 실행시키는지를 알 수 있다. 본 논문에서는 C/Assembly 혼합 코드를 분석할 수 있는 프로

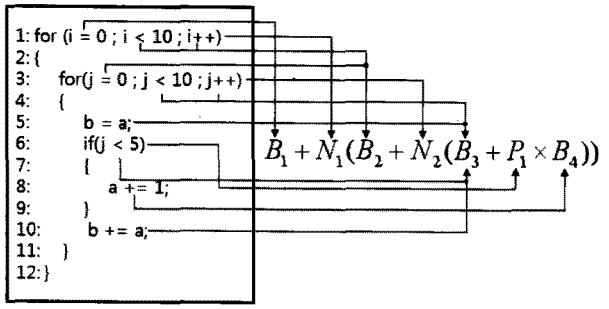

그림 13. 소스코드 분석을 통한 복잡도 매트릭의 예
Fig. 13. An example of the complexity metric of a C source code.

그램을 작성하여 실험하였다. 이는 코드 내에 정해진 몇몇 규칙들을 찾아내어 파일을 분석하는 방법이므로, 기존의 시뮬레이터나 프로파일러와는 비교할 수 없을 정도로 빠른 분석이 가능하다.

본 논문에서 제안하는 알고리즘 레벨 복잡도 측정 방법으로 생성되는 매트릭은 상세한 수식을 통하여 각 모듈의 복잡도를 설명한다. 그러나 B, N, P를 통하여 표현되기 때문에 각 순환문의 순환횟수와 분기문의 조건에 대한 확률은 제공되지 않는다. 각 순환문의 순환 횟수와 분기문의 조건을 만족할 확률은 프로그램에 따라 다르지만, 이러한 것들은 개발자의 입장에서 이미 알고 있는 값들이거나, 단순한 실험을 통해서 구할 수 있으며, 고정된 경우가 많다. 따라서 N, P의 값을 구해 매트릭에 대입함으로써 프로그램이 수행되는 동안 실행되는 명령어의 수를 계산할 수 있다.

그림 13과 같은 소스코드와 분석결과를 가정해보자.

앞서 설명한 소스코드 분석방법을 사용하여 그림 13의 예와 같은 매트릭을 생성할 수 있다. 이 매트릭의 각각의 명령어 블록에 포함되는 명령어의 종류와 실행횟수는 입력된 C/Assembly 혼합 코드의 분석을 통해 정확하게 측정됐다고 가정하자. 이때 그림 13의 프로그램이 실행하는 명령어의 수를 얻기 위해서 N_1, N_2, P_1의 값을 알아야 한다. 위의 코드를 통해 N_1과 N_2는 각각 10, P_1은 0.5임을 알 수 있다. 이를 통해서 프로그램이 수행하는 명령어의 수를 예측할 수 있다. 일반적으로 프로그램 내 순환문의 순화횟수는 위와 같이 명시적이다. 그러나 분기분의 확률은 알고리즘의 특성에 기인하므로 별도의 계산이 필요한 경우가 있다. 다음 절의 실험 결과에서는 이러한 분기에 대한 확률을 계산하여 대입한 측정치이다.

Ⅳ. 제안하는 알고리즘의 성능 분석 결과

본 논문에서 제안한 방법의 성능을 검증하기 위하여 복잡도 분석을 위한 소프트웨어의 구현을 통하여 각 명령어 블록에 대한 분석 결과와 복잡도 매트릭을 파일로 저장하도록 하였다. 저장된 명령어 블록은 각 명령어 블록에서 실행되는 모든 명령어의 수와 명령어의 종류별 실행횟수를 포함한다. 각 명령어 블록들은 함수별로 생성되며, 해당 함수의 복잡도 매트릭이 명령어 블록들에 이어서 표시되도록 하였다.

그림 15는 H.264/AVC에서 사용되는 6-탭 보간필터 모듈에 대한 소스코드를 입력으로 하여 제안한 알고리즘의 구현을 통해 분석된 결과파일의 일부분이다. 그림 14와 같이 각 명령어 블록에서 수행되는 명령어들의 총합과 명령어 별 수행횟수를 포함하여 순차적으로 저장되며, 각 함수에 해당하는 복잡도 매트릭을 표시한다. 따라서 해당 매트릭에 각 명령어 블록에 포함되는 명령어의 수와 순환횟수 N, 조건 분기 확률 P를 각각 대입함으로써 각 모듈별로 실행되는 명령어의 수와 순환문의 순환횟수 등을 통해 함수 전체에서 실행되는 명령어의 수와 연산이 집중되는 문맥을 알 수 있다.

본 논문에서 제안한 알고리즘 레벨 복잡도 측정 결과의 정확도를 측정하기 위해서 Ⅱ장에서 설명한 시뮬레이터인 SimpleScalar를 사용하여 측정한 프로그램의 총 실행 명령어의 수와 제안한 알고리즘을 통해 예측된 총 실행 명령어의 수를 비교하였다. 즉, 동일한 프로그램을 입력으로 하여 각각의 프로그램에서 실행되는 명령어의 실행 횟수를 비교함으로써 제안한 알고리즘 레벨 복잡

```
===B23===
InstSum : 12
mov : 1
str : 2
ldr : 3
sub : 1
cmp : 1
blt : 1
b : 2
add : 1

===B24===
InstSum : 537
ldr : 102
add : 170
mov : 152
rsb : 33
sub : 2
ldrb : 22
str : 9
cmp : 18
movge : 9
movlt : 9
strb : 9
b : 2

METRIC :
( B1 + N1 ( B2 + N2 ( B3 + B4 ) ) + N3 ( B5 + N4 ( B6 + B7 ) ) + N5 ( B8
```

그림 14. 제안한 알고리즘에 대한 결과의 일부
Fig. 14. Some part of result of the proposed method.

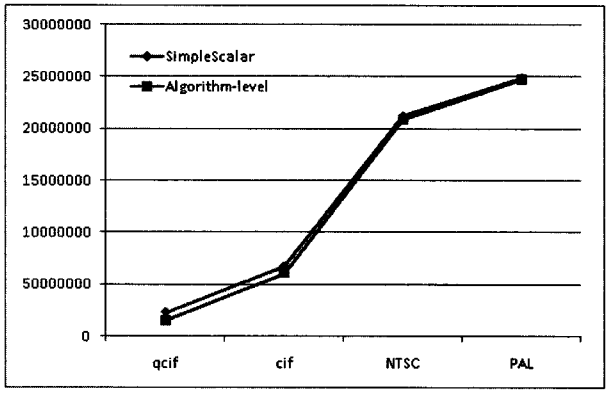

그림 15. H.264/AVC 4×4 정수변환에 대한 제안한 알고리즘의 성능 비교

Fig. 15. Performance comparison for the proposed algorithm with 4×4 integer transform in H.264/AVC.

도 측정의 정확도가 비교될 수 있다. 실험을 위하여 사용된 프로그램은 최신 비디오 코덱인 H.264/AVC에서 압축 성능에 중요한 비중을 차지하는 4×4 정수변환, 화면 내 예측, 화면 간 예측 모듈이다. 입력 영상의 크기에 따른 비교를 위하여 qcif(176×144), cif(352×288), NTSC(720×486), PAL(720×576) 영상을 사용하였다.

다음 그림 15는 4×4 정수변환에 대한 알고리즘 레벨 복잡도 측정 결과와 SimpleScalar를 통해 측정된 실행 명령어의 수를 비교한 것이다. 4×4 정수변환은 영상의 부호화를 블록 단위로 처리함에 있어서, 공간축의 신호를 주파수 축으로 옮겨 신호의 에너지가 DC방향으로 집중되는 특성을 이용하기 위하여 수행되는 모듈이다.

그림 15의 Y축은 실행된 명령어의 수이며, X축은 영상의 크기를 나타낸다. 실험은 각 크기별 영상 한 장에 대하여 수행되었다. 실험에 사용된 영상들에 대하여

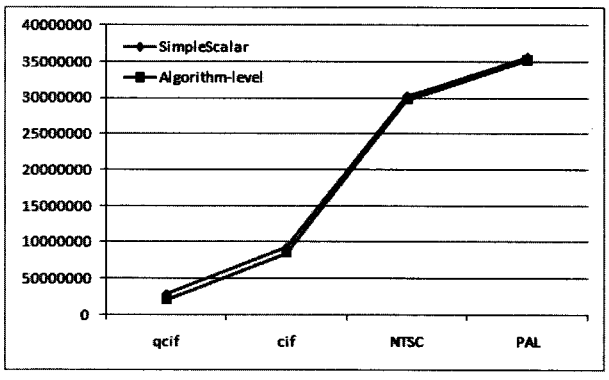

그림 16. H.264/AVC 화면 내 예측 부호화기에 대한 제안한 알고리즘의 성능 비교

Fig. 16. Performance comparison for the proposed algorithm with intra-prediction in H.264/AVC.

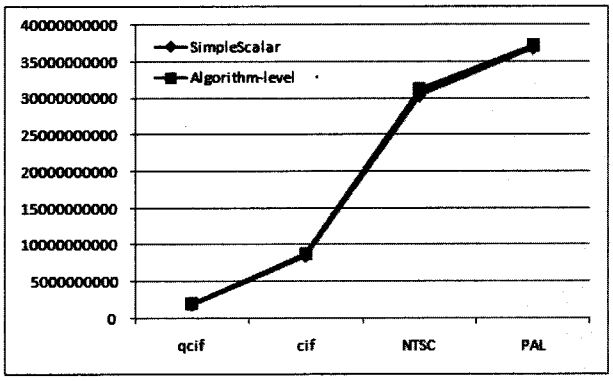

그림 17. H.264/AVC 화면 간 예측 부호화기에 대한 제안한 알고리즘의 성능 비교

Fig. 17. Performance comparison for the proposed algorithm with inter-prediction in H.264/AVC.

SimpleScalar와 제안한 알고리즘간의 실행명령어 수에 대한 오차의 평균은 약 11.6%로 측정되었다.

다음의 그림 16은 제안한 알고리즘의 성능평가 중 화면 내 예측 모듈에 대한 실험 결과이다. 영상의 화면 내 예측은 영상의 공간상의 지역적 상관도가 높은 특성을 사용하여, 중복성을 제거함으로써 낮은 값의 차분영상을 얻기 위하여 사용된다.

H.264/AVC의 화면 내 예측 모듈에는 예측 방향별로 0~8까지 총 9개의 예측모드가 존재한다. 이에 따라 cif 영상을 사용하여 각 모드에 대한 분기조건별 확률을 구하여 해당 모듈의 복잡도 매트릭에 적용하였으며, 다른 크기의 영상에도 이와 동일한 확률을 적용하였다. 그 결과 각 크기별 영상에 대하여 SimpleScalar 대비 평균 약 9.6%의 오차를 보였다.

그림 17은 H.264/AVC의 화면 간 예측 모듈에 대한 실험 결과를 나타낸다. 화면 간 예측 모듈은 부호화되고 있는 현재 영상과 이전 영상 간의 높은 시간적 상관도를 사용하여 중복성을 제거하는 모듈로써, H.264/AVC 인코더에서 가장 높은 복잡도를 갖는 모듈 중의 하나이다.

그림 17에 대한 실험 결과, 화면 간 예측 모듈은 SimpleScalar 대비 약 3.5%의 오차를 보였다.

위의 세 가지 실험 결과를 통하여 제안한 방법을 통한 복잡도 측정 결과가 입력되는 영상이 커짐에 따라서 시뮬레이터로 측정한 결과와 유사한 비율로 증가함을 알 수 있다. 이는 알고리즘 레벨에서의 복잡도 측정 방법을 사용하여 프로그램을 실행시키지 않고도 실제 시뮬레이션 결과에 근접하게 추정할 수 있음을 의미한다.

V. 결론 및 향후 연구 진행방향

본 논문에서는 프로그램의 최적화 과정에 반드시 필요한 복잡도 분석을 위하여 알고리즘 레벨 복잡도 분석 방법을 제안하였다. 제안한 알고리즘은 C/Assembly 혼합 코드를 분석하여 각각의 명령어 블록, 순환문, 분기문에 대하여 B, N, P를 사용한 복잡도 매트릭을 생성함으로써 각 모듈별 복잡도를 쉽게 알 수 있도록 하였다.

제안한 알고리즘에 대한 검증을 위하여 최신 비디오 코덱인 H.264/AVC의 4×4 정수변환, 화면 내 예측 부호화기, 화면 간 예측 모듈을 입력으로 하여 동일한 입력에 대한 SimpleScalar의 시뮬레이션 결과를 통해 측정된 실행 명령어 수를 비교하였다. 그 결과, 각각의 툴에 대하여 11.6%, 9.6%, 3.5%의 오차를 확인하였다. 이를 통하여 실제 하드웨어를 모델링하여 실행간 측정을 하는 기존의 시뮬레이터들과는 달리, 소스코드의 분석을 통해서 비교적 정확하게 복잡도를 측정할 수 있음을 증명하였다.

참 고 문 헌

[1] Coleman D. Bagwell, Emil Jovanov, Jeffery H. Kulick, "A Dynamic Power Profiling of Embedded Computer Systems" IEEE System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium, pp. 15-19, Huntsville, Alabama, March 2002.

[2] Chun-Hao Hsu, Jian Jhen Chen, Shiao-Li Tsao, "Evaluation and modeling of power consumption of a heterogeneous dual-core processor", IEEE Transactions on Volume 24, Issue 7, Computer-Aided Design of Integrated Circuits and Systems, pp. 1030 - 1041, Hsinchu, Taiwan, July 2005.

[3] Yu Hu, Qing Li, C.-C. Jay Kuo, "Run-time Modeling and Estimation of Multimedia System Power Consumption", 11th IEEE International Conference, Embedded and Real-Time Computing Systems and Applications, pp.353-356, 17-19, Hong Kong, China, Aug. 2005.

[4] D. Burger, T. Austin, "The simplescalar tool set version 2.0", Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[5] Toshinori SATO, Yukio OOTAGURO, Masato NAGAMATSU, Haruyuki TAGO, "Evaluation of Architecture-level Power Estimation for CMOS RISC Processors", Low Power Electronics, IEEE Symposium, pp. 44 - 45, San Jose, CA, 9-11 Oct. 1995.

[6] Vivek Tiwari, Sharad Malik, Andrew Wolfe, "Power analysis of embedded software: a first step towards software power minimization", IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 2, no. 4, pp. 437-445, December 1994.

[7] Vivek Tiwari, Sharad Malik, Andrew Wolfe, "Instruction level power analysis and optimization of software", IEEE VLSI Design, 1996. Proceedings., Ninth International Conference, pp. :326 - 328, Bangalore, India, Jan. 1996.

[8] Chi-ying Tsui; Marculescu, R.; Marculescu, D.; Pedram, M., "Improving the efficiency of power simulators by input vector compaction", 33rd IEEE Design Automation Conference, pp. 165-168, Las Vegas, NV, 3-7 June 1996.

[9] Diana Marculescu, Radu Marculescu, Massoud Pedram, "Stochastic Sequential Machine Synthesis Targeting Constrained Sequence Generation", the 33rd annual Design Automation Conference, pp. 696-701, Las Vegas, Nevada, United States, January, 1996.

저 자 소 개

임 웅(학생회원)

2008년 광운대학교 컴퓨터공학과 학사
2010년 광운대학교 컴퓨터공학과 석사
2010년~현재 광운대학교 컴퓨터공학 박사과정.

<주관심분야 : 영상압축, 컴퓨터 비젼, 영상신호처리>

심 동 규(정회원)

1999년 서강대학교 전자공학과 공학박사
1999년~2000년 (주) 현대 전자
2000년~2002년 (주) 바로 비젼
2002년~2005년 Univ. of Washington
2005년~현재 광운대학교 컴퓨터공학과 (부교수)

<주관심분야 : 영상신호처리, 영상압축, 컴퓨터비전>

최 웅 일(정회원)

2000년 성균관대학교 전기전자 및 컴퓨터공학부 학사
2002년 성균관대학교 전기전자 및 컴퓨터공학부 석사
2010년 성균관대학교 전기전자 및 컴퓨터공학부 박사
2006년~2008년 삼성종합기술원 컴퓨팅랩 전문 연구원
2008년~현재 삼성전자 DMC 연구소 멀티미디어 연구팀 책임연구원

<주관심분야 : 멀티미디어 프로세싱, 비디오코딩>

남 정 학(학생회원)

2006년 광운대학교 컴퓨터공학과 학사
2008년 광운대학교 컴퓨터공학과 석사
2008년~현재 광운대학교 컴퓨터공학 박사과정

<주관심분야 : 영상압축, 멀티프로세서>

조 대 성(정회원)

1994년 서강대학교 전자공학과 학사
1996년 서강대학교 전자공학과 석사
1996년~2008년 삼성종합기술원 컴퓨팅랩 전문연구원
2008년~현재 삼성전자 DMC 연구소 멀티미디어 연구팀 수석연구원

<주관심분야 : 영상처리, 영상압축>