https://doi.org/10.21289/KSIC.2024.27.1.87 KSIC

머신러닝 컴파일러와 모듈로 스케쥴러에 관한 연구

A Study on Machine Learning Compiler and Modulo Scheduler

조두산*

Doosan Cho*

〈Abstract〉

This study is on modulo scheduling algorithms for multicore processor in machine learning applications. Machine learning algorithms are designed to perform a large amount of operations such as vectors and matrices in order to quickly process large amounts of data stream. To support such large amounts of computations, processor architectures to support applications such as artificial intelligence, neural networks, and machine learning are designed in the form of parallel processing such as multicore. To effectively utilize these multi-core hardware resources, various compiler techniques are being used and studied. In this study, among these compiler techniques, we analyzed the modular scheduler, which is especially important in one core's computation pipeline. This paper looked at and compared the iterative modular scheduler and the swing modular scheduler, which are the most widely used and studied. As a result, both schedulers provided similar performance results, and when measuring register pressure as an indicator, it was confirmed that the swing modulo scheduler provided slightly better performance. In this study, a technique that divides recurrence edge is proposed to improve the minimum initiation interval of the modulo schedulers.

Keywords : Machine Learning, High Performance, Compiler, Modulo Scheduler

* 정회원, 교신저자, 전자공학과, 국립순천대학교, 교수 * Dept. of Electronics Engineering, Sunchon National Univ.
E-mail: dscho@scnu.ac.kr

1. 서 론

인공지능과 머신러닝을 위한 알고리즘 프레임워크는 Tensorflow, PyTorch, ONNX, mxnet, Caffe, NNAPI 등이 다양한 환경과 목적으로 사용되고 있다. 이러한 응용 프로그램을 가속하기 위하여 armNN, arm COMPUTE LIBRARY, CMSIS-NN 등이 주로 사용되는 ARM 하드웨어를 위한 최적화된 소프트웨어 라이브러리로서 사용되고 있다. 그리고 가장 중요한 하드웨어 IP로는 CPU에서 arm CORTEX, arm DynamiQ, arm NEON, arm NEOVERSE, armv8-A SVE가 있고, GPU로는 arm MALI, NPU로는 Machine Learning Processor가 있다. 모든 응용 프로그램이 이러한 하드웨어의 리소스를 최적으로 사용하도록 설계되어 있는 것이다.

위와 같은 인공지능, 머신러닝, 뉴럴 네트워크 응용들을 위한 하드웨어로 다양한 아키텍처가 사용되는데, 그 특징을 잘 나타내는 것 중에서 캠브리콘 [1]이 있다. 캠브리콘을 비롯하여 뉴럴 네트워크 등을 위한 명령어 집합의 가장 큰 특징은 벡터/행렬을 위한 명령어들을 지원한다는 점이며 다양한 형태의 벡터/행렬 연산이 가능하다는 점이다. 캠브리콘은 크게 4가지로 명령어들을 분류하여 설계되어 있다. 제어/데이터 전송/연산/논리 명령어들이 그것이다. 하나의 뉴런 연산은 복수의 입력과 출력들로 구성되기 때문에 실제의 뉴럴 프로세서는 대단위 입력과 출력을 계산해야 됨으로 행렬/벡터 연산을 위한 명령어는 필수적인 요소가 된다. 이를 위한 VLOAD (vector load), MLOAD (matrix load), VSTORE (vector store), VGTM (Vector Greater Than Merge) 등의 명령어들이 설계되어 있다. 이러한 명령어들을 사용하면 다른 아케텍처에 비하여 훨씬 적은 양의 코드와 실행시간으로 뉴럴 네트워크의 기능을 구현할 수 있게 된다.

본 연구에서는 가장 많이 사용되는 ARM 하드웨어 아키텍처를 분석하여 보고, 이러한 하드웨어 리소스를 최적으로 사용하기 위한 컴파일러 최적화 기법으로 모듈로 스케쥴링을 소개하기로 한다.

Fig. 1에 나타난 바와 같이 머신러닝을 위하여 최적으로 설계된 동적 공유 유닛 (DynamiQ shared unit, DSU)은 7개의 코어 클러스터와 비동기 브리지 그리고 기타 주변 모듈로 이루어져 있다. 이러한 동적 공유 유닛이 대량의 데이터 스트림을 병렬 처리하도록 구성되어 있다. 대량의 데이터를 병렬처리하려면 각 코어들이 독립적으로 계산할 수 있는 데이터를 분리하여 코어들에게 할당하여야 한다. 그렇기 때문에 DynamiQ [2]는 컴파일러와 같은 코드 생성기를 정교하게 설계해야 한다.

DynamiQ와 같은 멀티코어 프로세서의 성능은 컴파일러와 같은 코드 생성기와 태스크 스케쥴러 그리고 각 코어에서 모듈로 스케쥴링을 사용하여 병렬성을 극대화하는 알고리즘에 의하여 결정된다. 본 논문에서는 각 코어에서의 성능을 결정하는 모듈로 스케쥴링을 소개하는데, 특히 상용화된 제품에 많이 사용되는 반복 모듈로 스케쥴러 (iterative mmodulo scheduler [3])와 스윙 모듈로 스케쥴러 (swing modulo scheduler [4])에 대하여 깊이 있게 고찰하여 본다.

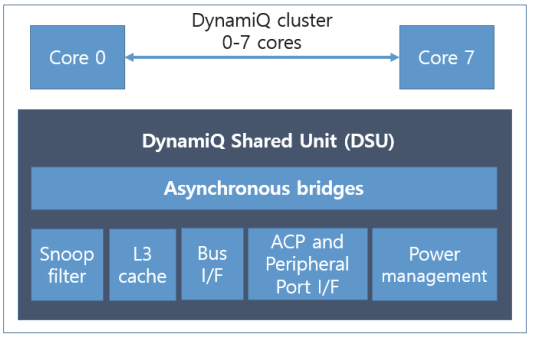

Fig. 1 DynamiQ shared unit

다음 장에서는 반복 모듈로 스케쥴러에 대하여 소개하고 3장에서는 스윙 스케쥴러에 대하여 소개한다. 4장에서는 알고리즘들을 비교 분석하여 보고, 5장에서는 본 연구에 대한 결론을 논의하도록 한다.

2. 반복 모듈로 스케쥴링 (Iterative Modulo Scheduling)

반복 모듈로 스케쥴러는 arm의 DynamiQ 아키텍처와 같은 멀티 프로세서를 위한 명령어 레벨 스케쥴링을 위한 알고리즘이다. DynamiQ는 다양한 조합의 멀티프로세서를 구성할 수 있는데, 여기서는 8개 프로세싱 코어를 가정하여 스케쥴러를 설명하겠다.

반복 모듈로 스케쥴러는 크게 3단계로 구성된다.

1. 의존도 그래프 구성 (dependency graph build) : 명령어들 사이의 true, anti, output 의존도 분석
2. 명령어 스케쥴: 명령어들의 비용 (cost)를 계산하여 resource reservation table를 구성하여 명령어의 스케쥴을 계산
3. 루프 코드의 prolog/epilog code를 구성 및 코드 변환

반복 모듈로 스케쥴러는 주로 루프 (loop) 코드를 대상으로 수행한다. 대부분의 응용 프로그램의 수행시간의 대부분을 루프 코드가 차지하기 때문이다. 루프 코드를 있는 그대로 사용할 경우 명령어 병렬성이 멀티코어 리소스를 사용하기에는 부족할 수 있기 때문에 이러한 경우 루프 펼침 (loop unrolling) [5] 최적화를 적용할 수 있다. 우리는 루프 펼침 최적화를 적용한 상태에서 모듈로 스케쥴링을 적용하기로 한다.

반복 모듈로 스케쥴링을 적용하기 위한 첫 번째 단계는 의존도 그래프 구성이다. 의존도 그래프는 명령어들 사이의 의존성을 분석하여 명령어들의 실행 순서를 분석하는 정보를 제공한다.

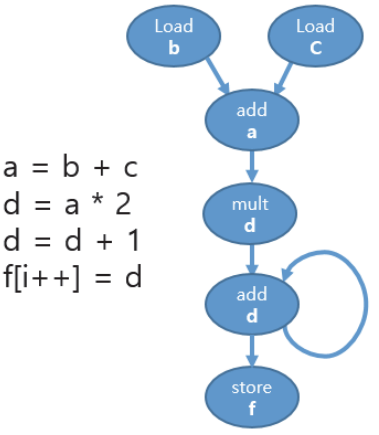

Fig. 2 Dependence graph

Fig. 2를 보면 왼쪽에 예제 코드와 오른쪽에 의존도 그래프가 나타나있다. 의존도 그래프는 예제 코드에서 연산에 따라 계산되는 각각의 변수로 나타내었다. 먼저 코드에서 첫 번째 연산은 변수 b와 c를 읽어서 변수 a를 계산한다. 두 번재로 변수 a에 상수 2를 곱하여 d를 계산한다. 변수 d는 1증가하여 d에 값을 재저장한다. 마지막으로 d는 배열 f에 저장되는 것으로 루프를 구성하는 마지막 문장이 계산된다. 이와 같은 계산흐름을 의존도 그래프로 나타낸 것이 Fig. 2의 오른쪽 그래프이다.

이 의존도 그래프를 사용하여 명령어 스케쥴을 만든다. 명령어 스케쥴은 다음과 같은 얼리 스타트(Early start) 함수를 기준으로 결정된다.

$$Estart(P) = Schedtime(Q) + Dealy(Q, P) - II \times Distance(Q, P) \quad (1)$$

각 명령어들에 대한 Estart 값을 계산한다. 그리고 모듈로 예약 테이블 (Modulo Reservation

```
function Iterative Schedule:
    begin
        compute HeightR;
        while(the list of unscheduled operations is-
        -not empty) and (Budget > 0) do
            begin
                operation = Highest Priority-
                -Operation();
                Estart = Calculate Early-
                -Start(operation);
                MinTime = Estart;
                MaxTime = MinTime + II -1;
                TimeSlot = FindTimeSlot(operation,-
                -MinTime, MaxTime);
                schedule(operation, TimeSlot);
                Budget = Budget -1;
         end (while)
    Iterative Schedule = the list of unscheduled-
    -operations is empty;
end (Iterative Schedule)
```

Fig. 3 Iterative Schedule

Table)을 사용하여 스케쥴될 위치를 결정한다. 전체 스케쥴 결과는 시작 간격 II (Initiation Interval)의 최소값인 Minimum II를 기준으로 결정한다. Minimum II는 리소스 시작간격 (resource II)와 재귀 시작 간격 (recurrence II) 중에서 더 작은 값으로 결정된다. 리소스 시작 간격 (resource II)는 스케쥴될 루프 코드의 명령어 수를 명령어들을 실행할 연산 유닛의 수로 나눈 몫으로 계산된다. 재귀 시작 간격 (recurrence II)는 의존도 그래프에서 가장 긴 재귀 회로 (recurrence circuit)의 지연시간 (delay) 값이다. 이렇게 계산된 Minimum II와 Estart값으로 반복 모듈로 스케쥴 알고리즘은 Fig. 3과 같이 구성된다. (-는 줄연결 표시)

Fig. 3에 반복 모듈로 스케쥴링 알고리즘[3]을 나타내었다. 작동 순서는 다음과 같다. 루프를 구성하는 명령어들의 스케쥴 우선 순위를 부여하기 위하여 의존도 그래프의 높이 (height)를 계산한다. 그래프의 높이에 따라서 각 명령어들이 순서대로 스케쥴이 계산된다. 두 번째로는 각 명령어

Time Slot	Data Bus	Load/Store Unit	ALU 1	ALU 2	Multiplier
0	b	b			a
1	c	c	d		
2	f	f			

Fig. 4 Modulo Reservation Table

들의 얼리 스타트 타임 (early Estart)이 수식 (1)에 따라 계산된다. MinTime은 얼리 스타트 타임이 되고, MaxTime은 MinTime + II -1로 계산된다. FindTimeSlot 함수는 모듈로 예약 테이블에서 현재 스케쥴하려는 명령어의 타임 슬롯을 계산한다. 이렇게 모든 명령어가 스케쥴될 때까지 반복하여 모든 명령어의 스케쥴을 계산한다. 하지만 스케쥴 결과가 정해지지 않는 경우 스케쥴에 실패함을 보고하게 된다. 스케쥴 구성은 각 명령어 별로 우선 순위에 따라 Estart에 맞추어 모듈로 예약 테이블의 타임슬롯에 할당하여 결정하게 된다.

Fig. 4는 모듈로 예약 테이블 예시를 나타내었다. 타임 슬롯은 싸이클 단위로 각 연산 유닛의 실행 시간을 나타내며, 로드/스토어 유닛, 연산 유닛 2개, 곱셈기 1개와 데이터 버스를 타나태어 명령어 스케쥴을 구성하도록 나타내고 있다. 스케쥴된 명령어들을 순서대로 테이블에 채워서 최종 스케쥴을 선택할 수 있다. 스케쥴이 최종 결정되면 루프 코드를 재구성하고, 루프의 시작전에 수행하는 프롤로그 (prolog) 코드와 루프 코드의 실행이 종료된 이후 실행되는 에필로그 (epilog) 코드를 생성하여 반복 모듈로 스케쥴 코드를 완성하고 알고리즘이 종료하게 된다.

3. 스윙 모듈로 스케쥴링 (Swing Modulo Scheduling)

스윙 모듈로 스케쥴링은 앞서 살펴본 반복 모듈로 스케쥴링의 단점을 보완하는 형태로 설계되

었다. 반복 모듈로 스케쥴러는 얼리 스타트 타임을 기준으로 스케쥴을 찾기 때문에 명령어 들이 최대한 모듈로 예약 테이블의 상단에 배치되도록 스케쥴을 계산한다. 이렇게 스케쥴을 계산하면 우선 레지스터 압력 (register pressure)이 높아지게 되어 유출 코드 (spill code) [6]가 발생할 가능성이 높아지게 되어 성능 저하를 유발할 수 있다.

이러한 문제점을 해결하기 위해서는 스윙 모듈로 스케쥴러는 Estart time 이외에 지연된 시작 타임 (latest start time) Lstart time을 스케쥴 결정 변수에 추가하여 사용한다. 예를 들면, Fig. 5에 나타난 의존도 그래프를 스케쥴 할 때 제일 지연시간이 긴 경로인 〈n1, n,5, n8, n10, n11〉을 먼저 스케쥴하고 난 후, 경로 〈n2, n3. n4〉를 스케쥴 할 때는 Lstart time을 기준으로 스케쥴을 찾을 수 있다.

이렇게 되면 두 가지 방식의 스케쥴을 계산하게 되는데, 첫 번째 방식을 톱다운 스케쥴 (Top-down), 다른 방식을 보톰업 스케쥴 (Bottom-up)이라고 부른다. 스윙 모듈로 스케쥴러는 의존도 그래프를 사용하여 톱다운과 보톰업 방식을 동시에 사용한다. 이렇게 되면 최소 시작 간격 (Minimum II)의 범위 안에서 명령어가 타임 슬롯 상단에 몰리지 않기 때문에 레지스터 압력 (register pressure)가 적절히 분산되는 효과를 얻을 수 있다.

Fig. 6에 스윙 스케쥴링의 명령어 순서화 (ordering) 알고리즘[4]을 나타내었다. 기본적으로 스윙 모듈로 스케쥴링 알고리즘은 반복 모듈로 스케쥴링 알고리즘과 동일한 순서로 루프 코드의 스케쥴을 계산한다. 우선 루프 코드의 의존도 그래프를 구성

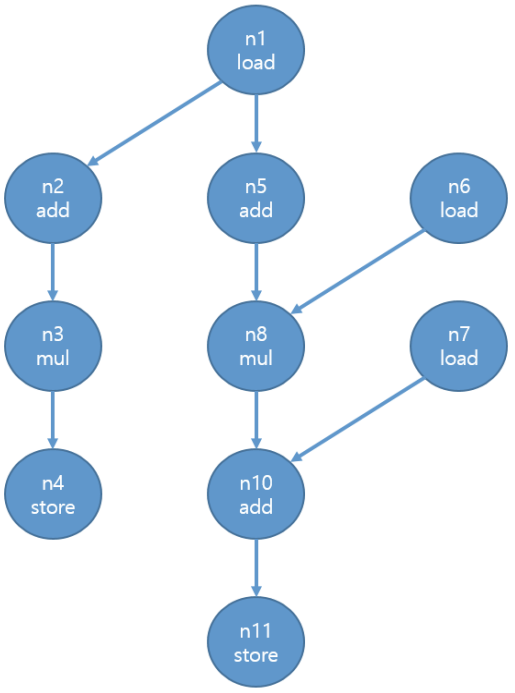

Fig. 5 Example of dependence graph

```
O := Empty_list
For each set of nodes S in decreasing priority do
  if Pred_L(O) ≠ ∅ and Pred_L(O) ⊆ S then
    R := Pred_L(O) ∩ S
    order := bottom-up
  else if Suc_L(O) ≠ ∅ and Suc_L(O) ⊆ S then
    R := Suc_L(O) ∩ S
    order := top-down
  else
    R := {node with the highest ASAP value in S} ;
      if more than one, choose anyone
    order := bottom-up
  end if
  Repeat
    if order = top-down
      while R ≠ ∅ do
        v := Element of R with the highest Hv ;
          if more than one, choose node with lowest  MOVu
        O := O | <v>
        R := R - {v} ∪ (Suc (v) ∩ S)
      endwhile
      order := bottom-up
      R := Pred_L(O) ∩ S
    else
      while R ≠ ∅ do
        v := Element of R with the highest Dv ;
          if more than one, choose node with lowest  MOVu
        O := O | <v>
        R := R - {v} ∪ (Pred(v) ∩ S)
      endwhile
      order := top-down
      R := Suc_L(O) ∩ S
    endif
  until R = ∅
endfor
```

Fig. 6 Swing modulo scheduling

한다. 의존도 그래프를 사용하여 각 명령어의 Estart time과 Lstart time을 계산한다. Fig. 6의 알고리즘에 따라 각 명령어들의 톱다운 스케쥴과 보톰업 스케쥴을 계산한다. 스케쥴이 완성되면 루프 코드의 프롤로그 코드와 에필로그 코드를 생성하고 스케쥴 코드를 완성하고 종료한다.

Fig. 6의 알고리즘을 다시 살펴보면, 먼저 스케쥴을 계산하기 위해서 의존도 그래프의 명령어들의 스케쥴링 순서를 정해야 한다. 이러한 명령어들의 순서는 먼저 상향식 방법으로 의존도 그래프의 노드들을 방문하여 순서를 정렬하고, 다시 하향식으로 스윙하여 노드들의 순서를 재정렬한다. 보통은 재귀 회로 (RecMII)를 결정하는 경로를 중심으로 순서가 결정된다. 스케쥴 순서가 결정되면, 이 순서에 따라 명령어들의 실제 스케쥴 위치가 결정된다. 먼저 하향식으로 타임 슬롯을 결정하고 한곳에 2개 이상의 명령어가 있을 경우 명령어의 모빌리티 (MOVu = Lstart time - Estart time)이 작은 것을 먼저 결정한다. 이렇게 상향식/하향식을 스윙하듯이 위아래로 스케쥴링하면 작은 레지스터 압력을 유지하면서 스케쥴링에 성공할 가능성을 높일 수 있게 된다.

4. 모듈로 스케쥴러의 비교

스윙 모듈로 스케쥴러 (swing modulo scheduler)와 반복 모듈로 스케쥴러 (iterative modulo scheduler) 그리고 슬랙 모듈로 스케쥴러 (slack modulo scheduler) [7] 등을 비교하여 분석한 연구 결과가 발표되었다 [8].

다양한 모듈로 스케쥴러의 성능을 비교한 연구 [8]은 다양한 스케쥴러의 지표들을 비교하였다. 퍼펙트클럽 벤치마크와 SPECfp95 벤치마크의 서브루틴 1936개 루프에 대하여 결과는 표1과 같이 정리되었다.

Table 1. Results for the scheduling performance

	Metric	IMS	SMS	SLACK
효율성	non-scheduled loops	0	0	9
병렬성	$\sum II$	14885	14843	15307
레지스터 압력	$\sum registers$	37380	31417	31705
실행시간	total cycles	39185	38628	41223

Table 1에서 IMS 열은 반복 모듈로 스케쥴러에 대한 결과를 나타내고, SMS 열은 스윙 모듈로 스케쥴러의 결과를 나타내며, SLACK은 슬랙 스케쥴러의 결과를 나타낸다. 각각 효율성, 병렬성, 레지스터 압력, 실행시간을 지표로 스케쥴이 안된 루프의 개수, 시작 간격의 총합, 사용된 레지스터의 개수, 총 실행시간을 싸이클로 나타내었다. 우선 가장 빠른 스케쥴링 알고리즘은 SMS로 파악되었다. 스케쥴에 실패한 스케쥴러는 SLACK 스케쥴러로 1936개에서 9개를 제외한 1927개에서 스케쥴 결과를 얻을 수 있었고, SMS와 IMS는 1936개 모두 스케쥴에 성공하였다. 시작 간격 II의 총합이 SMS가 가장 우수하며, IMS는 SMS와 거의 근사하게 나왔으며, SLACK은 다소 성능이 떨어지는 것으로 나타났다. II의 총합이 SMS가 가장 작게 구성되기 때문에 실행시간도 SMS가 가장 작게 측정되었다. 또한 레지스터의 총합도 SMS가 가장 낮게 요구되었다. 결과적으로 SMS가 가장 빠른 모듈로 스케쥴링 알고리즘이면서 또한 가장 작은 레지스터를 요구하는 알고리즘이기 때문에 성능이 가장 우수하게 측정된 것으로 파악할 수 있다. 이러한 결과가 나온 이유는 IMS 스케쥴링 알고리즘은 하향식 명령어 스케쥴링 알고리즘으로 최적의 타임슬롯을 Estat time을 기준으로 검색하기 때문에 레지스터 압력이 높아져 유출 코드가 발생하는 경우가 있을 수 있기 때문으로 판단된다. SMS는

변수들의 라이프타임을 고려하여 레지스터 압력을 낮추도록 양방향으로 스케쥴링하도록 구성된 알고리즘이다. 다만 SMS와는 스케쥴링 방식이 다르게 구성되어 레지스터 압력은 낮으나 시작간격이 높게 계산되어 실행시간이 낮게 구성되는 경향이 있는 알고리즘이다. 결과적으로 아키텍처의 설계 목적에 따라 다르게 선택되겠으나 현재의 컴파일러들은 SMS 알고리즘을 선호하는 경향이 있다.

4.1 개선사항

반복 모듈로 스케쥴링 알고리즘과 스윙 모듈로 스케쥴링 알고리즘을 앞서 살펴보았다. 이들 알고리즘은 스케쥴링된 코드가 최상의 성능을 갖도록 설계되어 있었다. 반복 모듈로 스케쥴링 알고리즘은 레지스터 압력을 고려하지 않고 스케쥴링을 계산하기 때문에 의도치 않게 유출 코드 (spill code)를 생성하여 성능을 저하시키는 문제점을 내재하고 있었다. 이러한 문제점을 해결한 것이 스윙 모듈로 스케쥴링이다. 스윙 모듈로 스케쥴링은 레지스터 압력을 낮추기 위하여 상향식, 하향식 스케쥴링을 차례로 수행한다. 하지만 이러한 스윙 모듈로 스케쥴링도 복잡한 의존도 그래프의 에지를 고려하지는 않고 있다. 본 연구에서는 이러한 복잡한 에지를 분리하는 기법을 추가하여 스윙 모듈로 스케쥴링의 성능을 개선하도록 하였다.

[3][4]에 기술된 바와 같이 최소 시작간격 (Minimum II)은 리소스의 개수에 의하여 계산되는 ResII (Resource Initiation Interval) 와 의존도 그래프의 순환에지 (recurrence edge)에 의하여 계산되는 RecII (Recurrence Initiation Interval) 값을 사용하여 MII = Max(ResII, RecII)와 같이 계산된다. 보통의 경우 ResII 〈 RecII이기 때문에 여기서는 MII = RecII로 가정한다. 따라서 최소 시작간격이 최소값을 갖도록 하기 위하여 RecII을 줄이는 것이 성능 개선의 핵심이 된다.

Fig. 7의 왼쪽은 루프 코드에서 의존도 그래프를 명령어로 나타내고 있다. mpy는 곱셈연산, mac은 곱하기-더하기를 동시에 수행하는 복합 연산자이다. 각 명령어가 각 1 cycle 시간을 소요한다고 가정할 경우, 적색 원으로 표시한 레지스터 d3에 의하여 recII가 4로 구성됨을 확인할 수 있다. tfr명령은 변수 복사 명령으로 d3을 d7로 복사한 것을 볼 수 있다. 변수 복사를 이용한 의존도 에지 분리를 Fig. 7의 오른쪽에 코드에 나타내었다. 이렇게 순환 에지 (recurrence edge)를 분리하면 recII를 줄일 수 있게 된다. Fig. 7의 예제의 경우 MII가 왼쪽 4 cycle 에서 오른쪽 3 cycle로 축소되었다. 결과적으로 해당 루프코드가 100 만번 수행된다면, 전체 수행시간의 25%인 100만 cycle 타임을 개선 시키는 결과를 얻을 수 있게 될 것이다.

Fig. 8은 제안된 순환 에지 분리 기법의 성능을 분석하기 위한 코드를 나타낸다. 스타코어140 [9] 컴파일러를 사용하여 하프 레이트 GSM (half rate Global System for Mobile) 루프 코드 (loop code)를 컴파일하여 얻은 머신 코드와 그것의 의존도 그래프를 나타내었다. 의존도 그래프에서 AAU 명령어는 로드/스토어 관련 명령어이며, 나머지는 ALU 연산 명령어이다. 백에지로 구성된 것이 순환 에지이다. 아래 그림에서 보듯이 tfr 전달 명령어로 변수 복사를 하고, 그것으로 의존도 그래프의 순환 에지를 분리할 수 있다. 본래의 의

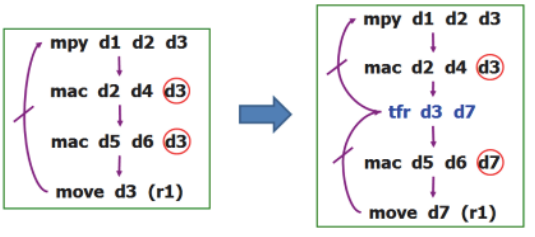

Fig. 7 Example of recurrence edge splitting

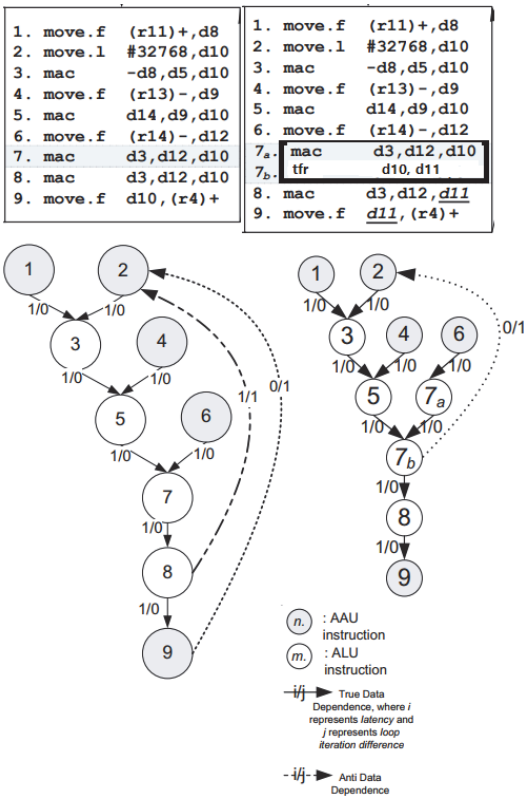

Fig. 8 Effect of recurrence edge splitting

존도 그래프를 보면 순환에지 RecII가 5로 구성되어 있어서, 해당 루프 코드는 5 cycle로 모듈로 스케쥴이 구성 되어진다. 제안된 순환 에지 분리 기법을 적용하면, 그림 8의 오른쪽 의존도 그래프를 보듯이 RecII가 3 cycle로 구성될 수 있게 된다. 결과적으로, 루프 전체 수행시간의 40%를 개선시킬 수 있게 되는 것이다.

5. 결 론

본 논문은 인공지능을 위한 하드웨어 아키텍처의 특징에 대하여 간략히 살펴보고 그러한 하드웨어 특징을 효과적으로 활용하기 위한 명령어 스케쥴러로서 모듈로 스케쥴러를 분석하여 보았다. 많이 사용되고 연구되고 있는 반복 모듈로 스케쥴러 (Iterative Modulo Scheduler, IMS)와 스윙 모듈로 스케쥴러 (Swing Modulo Scheduler, SMS), 그리고 슬랙 모듈로 스케쥴러 (Slack Modulo Scheduler, SLACK)들에 대한 실험 결과를 살펴보고 그 알고리즘들의 특징들을 분석하여 보았다. 그리고 모듈로 스케쥴링을 개선시킬 순환 에지 분리 기법을 제안하였다.

참고문헌

[1] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen, “Cambricon: an instruction set architecture for neural networks,” SIGARCH Comput. Archit. News 44, 3, 393–405, (2016).

[2] ARM DynamiQ data sheet, [online] https://www.arm.com/technologies/dynamiq, (2018).

[3] Ramakrishna Rau., “Iterative modulo scheduling: an algorithm for software pipelining loops,“ In Proceedings of the 27th annual international symposium on Microarchitecture (MICRO 27), 63–74., (1994).

[4] J. Llosa, A. Gonzalez, E. Ayguade and M. Valero, "Swing module scheduling: a lifetime-sensitive approach," Proceedings of the Conference on Parallel Architectures and Compilation Technique, pp. 80-86, (1996)

[5] Huang, J.C. and Tan Siek Leng. “Generalized loop-unrolling: a method for program speedup.” Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99, 244-248, (1999).

[6] G. J. Chaitin, “Register allocation & spilling via graph coloring,” In Proceedings of the 1982 SIGPLAN symposium on Compiler construction (SIGPLAN '82). pp. 98–105, (1982).

[7] R.A. Huff, “Lifetime-sensitive modulo scheduling,” In Proc. of the ACM SIGPLAN'93 Conference on Programming Language, Design and Implementation, pages 258-267, (1993).
[8] Josep M. Codina, Josep Llosa, and Antonio González, “A comparative study of modulo scheduling techniques,” In Proceedings of the 16th international conference on Supercomputing (ICS '02),, 97–106, (2002).
[9] Freescale semiconductor, SC140 DSP core reference manual, [online] https://www.nxp.com/docs/en/reference-manual/MNSC140CORE.pdf. (2005).

(접수: 2024.01.09. 수정: 2024.01.16. 게재확정: 2024.01.23.)