https://doi.org/10.21289/KSIC.2017.20.1.001 KSIC

VLIW (Very Long Instruction Word) 형식 드론 FCC(Flight Control Computer)의 실시간성 개선을 위한 소프트웨어 성능 가속화 연구

A Study on software performance acceleration for improving real time constraint of a VLIW type Drone FCC

조두산1*

Doo-San Cho[1*]

〈Abstract〉

Most conventional processors execute program instructions in a sequential manner. On the other hand, VLIW processor can execute multiple instructions at the same time. It exploits instruction level parallelism to improve system performance. To that end, program code should be rearranged to VLIW instruction format by a compiler. The compiler determine an optimal execution order of instructions of a program code. This instruction ordering is also called instruction scheduling. The scheduling is an algorithm that decides the execution order for instruction codes in loop parts of a program so that the instruction level parallelism can be maximized. In this research, we apply an existing scheduling algorithm to a VLIW FCC and describe analysis results to further improve its performance. And, we present a solution to solve some limitation of the existing scheduling technique. By using our solution, FCC's performance can be improved upto 32% compared to the existing scheduling only setting.

Keywords : High Performance, Flight Control Computer, Software Performance, Realtime, Drone

1*정회원, 교신저자, 국립순천대학교, 부교수
(E-mail: dscho@scnu.ac.kr)

1*Dept. of Electrical & Electronic Engineering, Sunchon National University

1. 서론

VLIW (Very Long Instruction Word) 타입의 프로세서 아키텍처는 명령어 단계 병렬성을 지원하기 위하여 복수의 연산처리 장치로 구성된다. VLIW의 개념은 Fig. 1에 나타난 바와 같이 4개의 명령어가 4개의 PE(Processing Element, 처리장치)를 이용하여 한 번에 실행될 수 있는 형태의 프로세서 아키텍처와 같다. 이러한 복수개의 연산처리 장치를 최대한 활용하도록 소프트웨어가 구성된다면 소프트웨어 응답 지연 속도와 실행 성능을 개선할 수 있고 결과적으로 전체 시스템의 실시간성이 개선될 수 있다.

소프트웨어 파이프라이닝 [1][2] 혹은 반복 모듈로 스케쥴링 [3][4]은 VLIW 형식의 프로세서에서 명령어 단계 병렬성을 최대한 활용하도록 프로그램 코드(명령어)를 재구성하는 컴파일러 기법이다. 기존의 순차적인 프로그램 코드를 복수의 연산 처리 장치에서 동시에 실행될 수 있도록 명령어 병렬성을 찾아 코드를 재배치 해주는 기능을 수행한다. Fig. 2에 명령어 스케쥴링의 예제가 나타나있다. 순차적으로 기술되어 있는 코드를 여러개의 프로세싱 처리장치 PE에서 동시에 실행되도록 명령어 실행 순서를 재배치한 결과를 나타낸다. 명령어 실행 순서를 결정할 때는 프로그램 결과의 정확성에 오류가 없도록 여러 가지를 고려하여 진행된다.

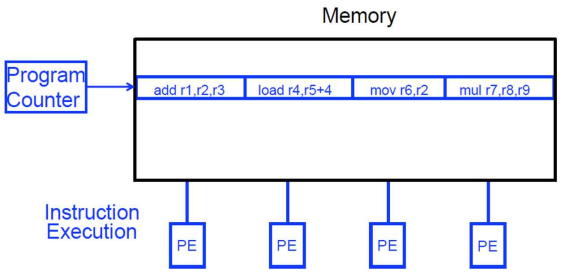

Fig. 1. Concept of VLIW.

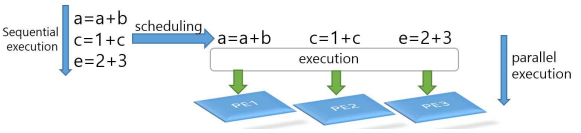

Fig. 2. An example of the scheduling.

본 연구에서는 VLIW 형식의 프로세서 아키텍처를 드론 FCC로 사용할 때 실시간성 확보를 위하여 기존 소프트웨어 파이프라이닝 기법을 적용하고 그 결과를 분석하여 본다. 이때 발생하는 문제점을 조사하고 그에 대한 해결책을 제시한다. 본 연구 결과를 이용하면 성능면에서 32%, 배터리 소모면에서 40%의 개선이 가능할 것으로 확인되었다.

2장에서 VLIW 형식의 프로세서와 소프트웨어 파이프라이닝 관련 배경지식을 살펴보고, 3장에서 소프트웨어 파이프라이닝 적용 결과 나타난 문제점 분석, 4장에서 그에 대한 해결책을 제시한다. 5장에서 실험결과를 제시하고 결론을 기술하도록 하겠다.

2. 배경지식

프로세서 아키텍처의 성능을 향상시키는 전통적인 방법으로 명령을 부분적으로 분할하여 명령을 부분적으로 동시에 실행할 수 있도록 하는 것(파이프라이닝이라고 불림)을 들 수 있다. 이러한 기술들 중에서 수퍼스칼라(superscalar) 방식은 프로세서의 다른 부분들에서 명령어들을 개별적으로 실행하도록 디스패치(dispatching)하고, 비순차 명령어 처리 방식(out-of-order execution)은 프로그램과는 다른 순서로 명령어들을 실행하는 것을 포함한다. 이러한 기술들은 모든 명령어 실행 순서 결정을 하드웨어 내부적으로 정하기 때문에 하

드웨어를 고비용, 고에너지, 대형으로 복잡하게 구성하게 한다. 대조적으로, VLIW 기법은 어떤 명령들을 동시에 실행하거나 하는 모든 결정을 소프트웨어에 의존한다. 보다 실제적인 문제에 있어서, VLIW는 명령어 순서를 결정하는 컴파일러(스케쥴러)를 보다 복잡하게 만들지만, 하드웨어는 다른 많은 명령어 병렬실행 기법들 보다 간단하게 구성된다. 이는 저비용, 저전력, 소형이라는 키워드로 이어져 최신의 다양한 휴대용 디바이스, 특히 드론에 최적화된다고 할 수 있다.

소프트웨어 파이프라이닝은 VLIW 타입 프로세서를 위해서 명령어를 스케쥴링하는 한가지 기법이다. 프로그램 실행시간의 90%를 차지하는 전체 코드의 10%가 바로 루프 코드인데 루프 코드를 대상으로 적용하는 스케쥴링[5][6] 기법이다. 동시에 실행할 명령어를 루프 코드와 루프 코드 사이에서 찾는 방식이다. Fig. 3에 소프트웨어 파이프라이닝 예제를 나타내었다.

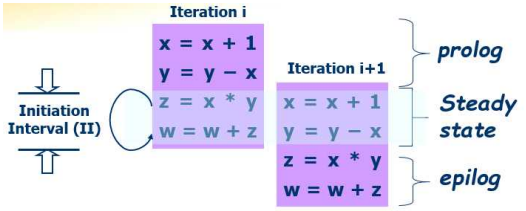

Fig. 3. An example of software pipelining.

예를 들어, 그림에 나타난 바와 같이 x=x+1, ..., w=w+z의 4줄의 코드를 100번 반복 수행하는 루프코드를 대상으로 소프트웨어 파이프라이닝을 수행해보자. 소프트웨어 파이프라이닝은 먼저 루프코드를 2~4회 펼친다. Fig. 3은 2회 펼친 예제이다. 펼쳐진 루프 코드에서 동시 수행 가능한 코드를 원래 루프 코드와 동일한 4개로 구성되도록 찾는다. 4개 코드를 순차적으로 수행하는 것을 가정하고 한 코드에 1싸이클을 소비한다고 하면 총4싸이클이 필요한데, Fig. 3의 경우 한번에 2개 코드를 수행하여 2싸이클에 4개 코드를 모두 수행할 수 있게 된다.

본 예제에서는 결정된 2싸이클을 루프 시작 간격 (II, Initiation Interval)이라 부른다. II를 시작 간격으로 루프의 새로운 반복이 수행되기 때문에 이렇게 부른다. 소프트웨어 파이프라이닝은 대상 루프코드에서 최소 II 값을 찾고 이에 맞추어 코드를 재배치 하는 기법이다. Fig. 3에서 II=2로 루프코드가 재구성되면 루프의 반복 i번째 코드 z=x*y, w=w+z와 반복 i+1번째 코드 x=x+1, y=y-x가 함께 수행된다. 따라서 새로 구성된 루프 코드의 반복이 앞선 반복의 중간부터 시작되기 (z=x*y 코드부터 시작함) 때문에 루프가 시작되기 이전에 x=x+1, y=y-x 코드가 한 차례 수행되어야 한다. 이 부분을 프롤로그 (prolog)라 한다. 마찬가지로 새로 구성된 루프 코드는 마지막 반복이 수행되면 x=x+1, y=y-x에서 종료된다. z=x*y, w=w+z코드가 한 차례 수행되지 못하기 때문에 루프코드의 아래에 z=x*y, w=w+z 코드가 한 차례 수행되도록 에필로그 (epilog)가 삽입된다. 원래의 루프 코드가 프롤로그, 에필로그 그리고 시작간격 II에 맞추어 수정 완료되면 소프트웨어 파이프라이닝이 완성된다. Fig. 3의 예제의 경우 원래 루프 실행시간이 순차 실행 4싸이클 x 100회 = 400싸이클인데, 파이프라이닝이 적용되면 수정된 루프 코드는 병렬 실행 2싸이클 x 100회 = 200싸이클로 실행시간이 50% 개선된다.

3. 소프트웨어 파이프라이닝 문제점 분석

소프트웨어 파이프라이닝 알고리즘은 크게 3가지 절차로 구성된다.

알고리즘 1. 소프트웨어 파이프라이닝

1. 시작 간격 II값을 계산함
   1.1 루프 코드를 대상으로 디펜던스 싸이클(dependence cycle)을 계산함
   1.2 가장 큰 디펜던스 싸이클 값을 II로 정함
2. 결정된 시작 간격에 맞추어 루프코드를 수정함
   2.1 이때 동시 실행 코드(명령어)의 개수가 PE의 개수보다 많으면 시작 간격 II를 PE 개수로 수정하고 2번을 다시 실행함
3. 프롤로그, 에필로그 코드를 파이프라이닝된 루프 코드 위/아래에 삽입함

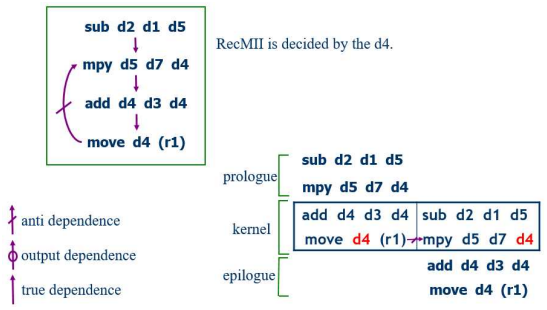

Fig. 4. A software pipelining detail.

상기 기술된 알고리즘에 따라 Fig. 4 코드를 예제로 살펴보겠다. 먼저 디펜던스 싸이클을 계산해야 한다. 프로그램 코드의 디펜던스는 3가지가 있으며, 프로그램의 최종 결과 값에 영향이 없도록 루프 코드를 수정하기 위해서는 이 3가지 디펜던스를 반드시 유지해야 한다. 따라서 디펜던스를 코드안에서 정확히 구분해야 한다.

디펜던스는 값의 흐름으로 구성되는데, 트루/안티/아웃풋 (true/ anti/ output) 3가지 종류가 있다. 트루 디펜던스는 “쓰기 후 읽기”를 유지하도록 값을 쓰기 전에 읽기 연산을 할 수 없음을 나타낸다. 안티 디펜던스는 “읽기 후 쓰기”를 유지하도록 읽기 전에 값을 쓰면 결과가 틀려질 수 있음을 나타낸다. 아웃풋 디펜던스는 “쓰기 후 쓰기”를 유지하도록 앞선 쓰기 연산이 완료되기 전에 뒤의 쓰기 연산이 먼저 수행될 수 없음을 나타낸다.

Fig. 4에서 sub(뺄셈)명령어가 d2와 d1 레지스터의 값을 서로 뺄셈하여 결과 값을 d5에 저장한다. mpy(곱셈) 연산에서 d5와 d7을 곱하여 d4에 저장한다. sub 연산과 mpy 연산은 트루 디펜던스 관계로 sub이 완료되어야 mpy를 수행할 수 있다. 왜냐하면 sub에서 d5값을 계산하면 d5를 mpy의 피연산자로 사용하기 때문이다. add(덧셈) 연산은 d4와 d3을 더하여 d4에 저장한다. mpy 연산과 add 연산은 d4 값으로 인하여 안티/아웃풋 디펜던스가 구성된다. mpy연산이 d4값을 계산하면 add연산이 d4를 피연산자로 사용하고 또한 결과 값을 쓰는 장소로 사용하기 때문이다. move(쓰기) 연산은 d4의 값을 메모리 (r1)주소에 저장한다. r1은 메모리 주소를 저장하는 레지스터이다. d로 시작하는 레지스터는 데이터를 저장하는 레지스터이다. 따라서 add연산과 move연산은 트루 디펜던스를 구성한다.

소프트웨어 파이프라이닝은 루프를 구성하는 반복과 반복 사이의 코드를 동시에 실행하도록 수정한다. 따라서 반복과 반복 사이의 디펜던스를 유지할 필요가 있다. 여기서 반복이라 함은 Fig. 4에서 루프가 루프를 구성하는 전체 코드(sub-mpy-add-move)를 한번 수행하고 그 다음 수행(sub-mpy-add-move)을 반복한다는 것을 뜻한다. 반복 코드와 반복 코드 사이의 데이타들 사이에 디펜던스를 디펜던스 싸이클이라 한다. 그림에 화살표로 나타내면 싸이클을 구성하기 때문이다. Fig. 4 코드에서 가장 큰 디펜던스 싸이클은 레지스터 d4에 의해서 구성된다. 루프 반복 i번째에서 move연산이 d4를 읽기로 사용하고 루프 반복 i+1번째에서 d4값을 mpy연산이 쓰기로 사용

한다. 따라서 i번째 move와 i+1번째 mpy가 안티 디펜던스 사이클을 구성한다. 여기서 각 명령어가 1 싸이클씩 실행시간을 소비한다면, mpy 연산 명령어가 1싸이클, add 연산 1 싸이클이 실행된 후 move연산이 실행된다. 따라서 d4로 구성되는 안티 디펜던스 싸이클의 길이는 2싸이클이 된다. 시작간격 II는 안티 디펜던스 싸이클 2로 결정된다.

```
add d4 d3 d4        sub d2 d1 d5
move d4 (r1)        mpy d5 d7 d4
```

안티 디펜던스 싸이클을 (읽기 후 쓰기)유지하면서 II=2로 루프 코드를 파이프라이닝하면 위와 같이 구성된다. 최종 코드는 move의 d4 읽기와 mpy의 d4 쓰기가 동시에 시작하지만 프로세서 내부적으로 d4 읽기가 d4 쓰기 보다 선행되어 동작하기 때문에 안티 디펜던스가 유지되어 실행된다. 루프 코드가 파이프라이닝 완료되면 위/아래로 프롤로그, 에필로그 코드가 삽입된다. Fig. 4의 왼쪽 코드는 순차적으로 4싸이클에 실행되던 루프를 나타내고, 오른쪽 하단 파이프라이닝된 루프는 2 싸이클 마다 반복 수행이 시작되는 형태로 개선된 것을 나타낸다.

파이프라이닝 완료된 루프 코드를 보면 안티 디펜던스는 주로 한 개의 레지스터를 반복적으로 사용하면서 발생하는 것을 알 수 있다. 이러한 발생은 컴파일러가 프로세서에 제한된 숫자로 구성된 레지스터 메모리를 최적으로 사용하기 위하여 항상 최소 개수로 분배하도록 레지스터 할당정책 알고리즘이 구성되었기 때문이다. 따라서 소프트웨어 파이프라이닝을 적용할 때 다른 레지스터를 추가로 사용한다면 안티 디펜던스를 제거하여 보다 개선된 소프트웨어 파이프라이닝 루프를 구성할 수 있을 것이다.

4. 안티 디펜던스 제거 방안

앞서 Fig. 4의 예제에서 add연산의 결과 값을 d4가 아닌 새로운 d9 레지스터에 저장한다면 move연산이 d9를 사용하게 되고 결과적으로 mpy와 move연산 사이의 안티 디펜던스는 제거된다. 새로운 레지스터 추가된 변경 코드를 Fig. 5에 나타내었다. mpy에서 add와 add에서 move로 d4를 사용하던 값의 흐름이 add 결과 값을 d9에 저장하는 형태로 변경하면서 디펜던스 흐름이 끊어지게 되었다. 이렇게 되면 2싸이클 길이로 연결된 안티 디펜던스 싸이클이 제거되면서 II도 변경되게 된다. 안티 디펜던스가 d9에 의하여 분리되어 안티 디펜던스 싸이클은 mpy와 add 그리고 add와 move사이에 존재한다. 따라서 안티 디펜던스 길이는 기존 2에서 1로 변경되고, 시작간격도 II=1로 변경하게 된다.

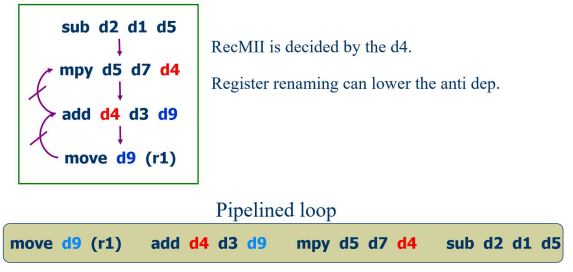

Fig. 5. Register renaming.

시작 간격이 1싸이클로 변경되면 파이프라인된 루프도 반복이 매 1싸이클 간격으로 시작될 수 있다. 즉, 원래의 루프가 한번 수행되는데 4싸이클을 소비하는데, II=1로 파이프라이닝된 루프는 1싸이클에 반복 수행 1회가 완료되어 실행시간이 4배 빨라지게 된다.

수정된 파이프라이닝 코드가 Fig. 5 하단에 나타나 있다. 한번에 4개 코드가 수행됨을 확인할

수 있다. 이상과 같이 새로운 레지스터를 추가하여 안티 디펜던스 싸이클을 제거하는 (혹은 디펜던스 싸이클 길이를 짧게하는) 것을 레지스터 리네이밍 (register renaming) [7]이라 부른다. 우리는 기존의 소프트웨어 파이프라이닝 알고리즘에 레지스터 리네이밍 스텝을 추가하여 파이프라이닝 성능을 개선하도록 하였다. 개선된 소프트웨어 파이프라이닝 알고리즘은 다음과 같다.

알고리즘 2. 개선된 소프트웨어 파이프라이닝

1. 시작 간격 II값을 계산함
   - 1.1 루프 코드를 대상으로 디펜던스 싸이클(dependence cycle)을 계산함
   - 1.2 안티 디펜던스는 싸이클을 구성하는 레지스터를 추출함
   - 1.3 가용한 레지스터를 이용하여 싸이클 구성 레지스터를 새로운 레지스터로 변경함
   - 1.4 남아있는 가장 큰 디펜던스 싸이클 값을 시작간격 II로 정함
2. 결정된 시작 간격에 맞추어 루프코드를 수정함
   - 2.1 이때 동시 실행 코드(명령어)의 개수가 PE의 개수보다 많으면 시작 간격 II를 PE 개수로 수정하고 2번을 다시 실행함
3. 프롤로그, 에필로그 코드를 파이프라이닝된 루프 코드 위/아래에 삽입함

5. 실험 및 결론

제안하는 알고리즘을 검증하기 위하여 우리는 현재 가장 많이 사용하는 컴파일러인 GCC 4.0을 이용하였다. 실험 대상 프로그램은 드론용 FCC에 사용하는 오픈소스 코드 4개 (FCC1,2,3,4)를 이용하였다. 특히 FCC를 구성하는 핵심 루프 코드들을 대상으로 결과를 평가하였다. 본 실험에서는 순수하게 제안된 기법만을 평가해 보기 위하여 모든 최적화 옵션을 제거하고 실험을 진행하였다. 원래의 소프트웨어 파이프라이닝 기법(알고리즘1)과 개선된 소프트웨어 파이프라이닝 기법(알고리즘2)를 대상으로 비교 실험을 진행하였다. 실험 환경 구성은 FCC를 TI c6000 core로 가정하여, 테스트용 보드를 PC에 연결하여 진행하였다. 실험 결과는 다음과 같다.

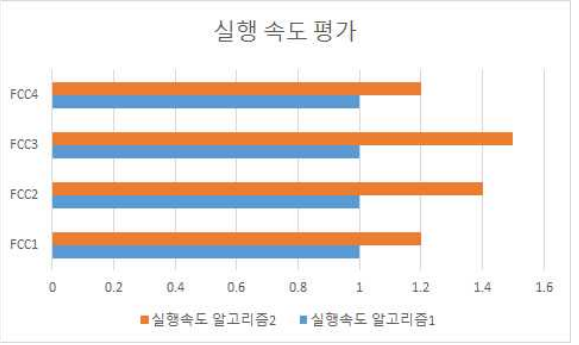

Fig. 6. Experimental results on execution time.

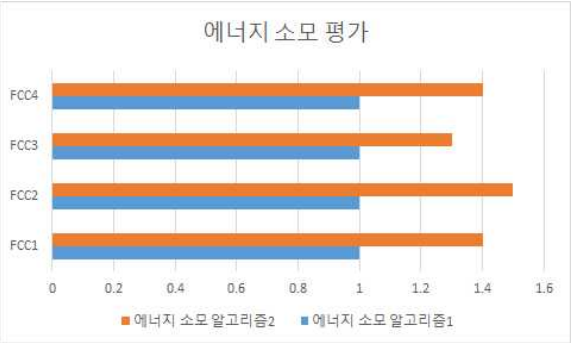

Fig. 7. Experimental results on energy consumption.

실험은 프로그램 실행속도(시간) 개선과 에너지 소모량 개선 두개의 측면에서 진행되었다. Fig. 6에 FCC flight stack에서 루프 코드의 실행속도

(시간) 알고리즘 1과 2, Fig. 7에 에너지소모 알고리즘 1과 2의 비교 결과가 바그래프 형태로 나타나 있다. 알고리즘 1의 결과를 기준값(baseline) 1로 두고 2의 개선 정도를 나타낸 것이다. 실행속도에서 32% 개선되었고 에너지 소모에서 40% 개선됨을 확인하였다.

본 연구에서는 새로운 형태의 스케쥴링 알고리즘이 아닌 기존의 스케쥴링 기법을 드론 FCC 환경에서 그대로 사용하면서 발생되는 한 가지 문제점을 개선하는데 목표가 있다. 이 문제점을 레지스터 리네이밍 기법을 이용하여 해결하였으며 결과적으로 FCC의 실행속도 및 에너지 소모량을 개선하였다. 이러한 결과는 드론의 실시간성 개선(응답속도 개선)에 상당한 도움을 줄 것으로 예상한다.

참고문헌

[1] M. Lam : Software pipelining: an effective scheduling technique for VLIW machines. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp.318-328, (1988).

[2] H. Allan, B. Jones, M. Lee, J. Allan : Software Pipelining. ACM Computing Surveys, 27, 3, (1995).

[3] B. Rau : Iterative modulo scheduling. HP Laboratories Technical Report, HPL94115, (1995).

[4] D. Lavery and W. Hwu : Unrolling-Based Optimizations for Modulo Scheduling. Proceedings of the 28th annual international symposium on Microarchitecture, pp.327-337, (1995).

[5] R. Huff : Lifetime-Sensitive Modulo Scheduling. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp.258-267, (1993).

[6] B. Rau and D. Glaser : Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proceedings of the 14th Microprogramming Workshop, pp.183-198, (1981).

[7] J. Hennessy and D. Patterson : Computer Architecture - a Quantitative Approach. Morgan Kaufmann, (2011).

(접수:2016.12.23. 수정: 2017.1.25. 게재확정: 2017.2.14.)