리스트 기반 스케줄링 과정의 경합 연산 우선 순위 결정에 대한 연구

오규철, 이준용
홍익대학교 컴퓨터공학과
{kcoh, jlee}@cs.hongik.ac.kr

Improved Priority Decision Making for Competitive Operators in List Based Scheduling Algorithms

KyuCheol Oh, JunYong Lee
Dept. of Computer Engineering, Hong Ik University

요 약

리스트 기반 스케줄링은 하드웨어의 자원이 일정한 수준으로 제한된 상황에서 스케줄링에 필요한 컨트롤 스텝의 수를 최소화하고자하는데 중점을 둔 일련의 스케줄링 기법 중 하나로, 연산의 우선 순위를 결정한 뒤, 그 우선 순위에 의거하여 준비된 일단의 연산들에 대한 스케줄링을 수행하는 방식이다. 따라서 연산의 우선 순위를 결정하는데 고려되는 조건들은 리스트 기반 스케줄링의 성능에 직접적인 영향을 주며, 현재까지 다양한 우선 순위 결정 조건들이 제시되어 있다. 그런데, 최종 합성 결과는 상위 수준 합성의 대상이 되는 입력 그래프의 특성에 따라 그 성능이 좌우되므로 일반적인 의미에서 최적의 우선 순위 결정 조건이란 존재하지 않는다. 본 논문에서는 단일한 우선 순위 조건하에서 경합하는 연산들의 우선 순위 결정시 다양한 우선 순위 결정 조건들을 균형있게 고려하여 보다 효율적인 스케줄링 결과를 얻는 방법을 제시하였다.

1. 서론

상위 수준 합성(High-Level Synthesis)은 하드웨어의 동작적 기술(Behavioral Description)을 구조적 기술(Structural Description)로 변환하는 일련의 과정을 의미한다. 일반적으로 상위 수준 합성은 입력으로 VHDL과 같은 하드웨어 기술 언어(Hardware Description Language)로 표현된 명세(Specification)를 받아들여, 그 결과로는 데이터패스와 그 데이터패스의 동작을 제어하는 컨트롤러로 구성되는 구조적 명세를 생성한다. 상위 수준 합성의 입력과 출력은 그 의미상으로 정확하게 일치(equivalent)하여야 하며, 입력에서 출력까지의 과정에서 하드웨어 기술 언어의 컴파일(Compilation), 스케줄링(Scheduling), 사용될 컴포넌트의 선택(Unit Selection), 선택된 컴포넌트의 결합(Unit Binding)등의 작업을 수행한다. 그 중에서 스케줄링은 연산의 수행 순서를 결정하는 단계로서 최종적인 합성 결과에 지대한 영향을 미치는 매우 중요한 단계이다. 일반적으로 스케줄링은 두 가지 범주로 나누어 구분하는데, 그 하나는 고정된 시간 내에서 하드웨어 컴포넌트의 사용을 최소화하는 방법(Time Constrained Scheduling)이고, 다른 하나는 하드웨어 컴포넌트로 인한 비용이 고정된 상태에서 디자인에 필요한 컨트롤 스텝의 수를 최소화하는 방법(Resource Constrained Scheduling)이다. 후자의 경우에서 리스트 기반 스케줄링(List Based Scheduling)은 논문[5]를 통해 제시된 스케줄링 방법으로, 준비된 연산들의 우선 순위에 관한 리스트(Priority List)를 참조하여, 스케줄링을 수행한다. 우선 순위 리스트는 우선 순위 함수(Priority Function)에 의해 결정되는데, 이 우선 순위 함수는 스케줄링의 결과에 직접적으로 영향을 준다. 따라서, 적절한 우선 순위 함수의 선정은 리스트 기반 스케줄링의 성능과 직결되는 매우 중요한 문제이다. 그런데, 일반적인 경우, 상위 수준 합성의 대상이 되는 입력 그래프의 특성에 따라 최종 합성 결과의 성능이 좌우된다. 따라서 최적의 우선 순위 함수를 결정하기란 대단히 어렵다.

본 논문에서는 리스트 기반 스케줄링시 연산의 수행 순서를 결정하는 단계에서, 단일한 우선 순위 함수의 결과를 사용하는 대신, 다양한 우선 순위 함수의 결과를 종합하여 보다 효율적인 스케줄링 결과를 얻는 방법을 제시하고, 기존의 스케줄링 방식과 비교하였다.

2. 관련연구

2.1 리스트 기반 스케줄링 알고리즘

리스트 기반 스케줄링 알고리즘은 하드웨어 자원이 한정된 상황을 극복하기 위해 연산별로 우선 순위 리스트(Priority List)를 운영한다. 우선 순위 리스트에는 스케줄링 될 준비가 끝난 연산들이 우선 순위 결정 기준에 따라 정렬된 형태로 저장된다. 여기서 스케줄링 될 준비가 끝났다는 의미는 그 연산의 모든 선행 연산에 대한 스케줄링이 완벽하게 이루어져있다는 의미이다. 우선 순위 리스트의 구성이 이루어지면 스케줄러는 준비된 연산 중에 높은 우선 순위를 가지는 연산부터 현재의 컨트롤 스텝(Control Step)에 할당하며, 이 할당 작업은 주어진 하드웨어 자원이 고갈될 때까지 계속된다.

2.2 우선 순위 결정 기준

우선 순위 결정 기준은 실제 알고리즘에서는 우선 순위 함수로 나타나는데, 현재까지의 연구들에서는 '연산의 이동성 범위[1,4]', '최종 후행 연산까지의 길이[3]', '병렬성 반발 정도[2]', '후행 연산의 개수'등의 다양한 우선 순위 결정 조건들이 제시되어 있다.

2.2.1 연산의 이동성 범위(Mobility Range)

연산의 이동성 범위이란 어떤 연산이 스케줄 될 수 있는 컨트롤 스텝의 영역을 의미한다. 이동성 범위가 좁은 연산의 스케줄링이 지연된다면 전체 스케줄링이 지연될 가능성이 크기 때문에 스케줄러는 이동성 범위가 좁은 연산에 높은 우선 순위를 두어야 한다.

2.2.2 최종 후행 연산까지의 길이(Criticality)

어떤 연산에 대해, 그 연산의 종속관계에 있는 연산 중 가장 뒤에 나오는 연산까지의 길이를 우선 순위 결정 기준으로 사용할 수 있다. 두

개 이상의 연산들이 서로 경합하는 경우, 최종 후행 연산까지의 길이가 가장 긴 연산에 높은 우선 순위를 두어야 한다. 그렇지 않다면 전체 스케줄링의 임계경로(Critical Path)가 길어져 스케줄링의 효율이 떨어질 수 있다.

2.2.3 병렬성 반발 정도(Force)

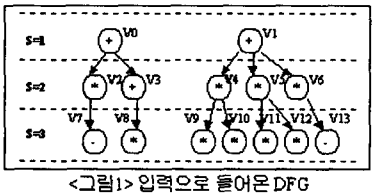
고정된 수의 하드웨어 자원을 효율적으로 사용하기 위해서는, 각각의 컨트롤 스텝에서 주어진 수의 하드웨어 자원을 모두 사용하도록 스케줄링하여, 연산 상호간의 병렬성을 높여 주어야 한다. 따라서 병렬성의 반발 정도가 높은 연산이 높은 우선 순위를 가져야 한다.

2.2.4 후행 연산의 개수

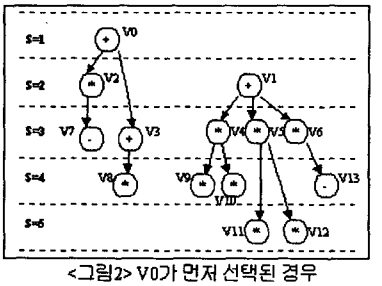
어떤 연산이 스케줄링되기 위해서는 그 연산의 선행 연산들이 모두 스케줄링되어야 한다. 그러므로, 후행 연산의 개수가 많은 연산에 높은 우선 순위를 두어, 다음 스텝의 우선 순위 결정시 보다 많은 연산들이 고려될 수 있도록 하는 편이 좋다.

2.3 퍼지 이론

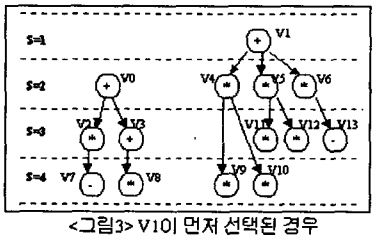
본 논문에서는 위와 같은 다양한 우선 순위 결정 기준을 종합하여 일관성있는 값으로 표현하기 위해 퍼지 로직을 이용하였다. 각각의 우선 순위 결정 기준들은 선호도:규칙(Preference Rule)을 통해 보다 세밀하게 보정하였으며, 다양한 기준들을 보다 폭넓게 수용하기 위해 보상 연산자(Compensatory Operator)를 사용하였다.



<그림1> 입력으로 들어온 DFG



<그림2> V0가 먼저 선택된 경우



<그림3> V1이 먼저 선택된 경우

3. 경합하는 연산에 대한 우선 순위의 결정

3.1 기존 스케줄링 기법의 문제점

기존의 리스트 기반 스케줄링 기법들에서는 단일한 우선 순위 결정 기준을 적용하여 스케줄링을 실행했다. 이때 동일한 우선 순위를 가진 두 개 이상의 연산들이 존재한다면, 이 경합하는 연산들 사이의 우선 순위 결정은 무작위로 이루어지게 된다. 아래의 예제들은 승산기 3개, 가산기 1개, 감산기 1개의 하드웨어 자원이 사용 가능한 상황에서 리스트 기반 스케줄링을 실행한 결과이다. 우선 순위 결정 조건은 연산의 이동성 범위를 기준으로 하였다.

<그림 1>은 스케줄링의 입력으로 주어진 DFG(Data Flow Graph)를 나타낸 것으로, 이때 연산 V1과 V2는 선행 연산이 없으므로 스케줄될 준비가 되어있다. 그런데, 주어진 가산기의 수가 1개로 한정되어 있으므로 두 연산 V1과 V2는 우선 순위 리스트에 저장되기 위해 서로 경합한다. 이 때, 두 연산의 이동성 범위는 '0'으로 동일하다. <그림 2>는 연산 V2가 선택되어 먼저 스케줄링 된 예를 보인 것이고, <그림 3>은 연산 V1이 선택된 경우의 예를 나타낸 것이다. 결과적으로 <그림 3>, 즉 후행 연산의 개수가 많은 연산을 선택한 편이 더 우수한 스케줄링 성능을 보임을 알 수 있다.

일반적으로 동일한 우선 순위 함수 값을 가지는 경합하는 두 개 이상의 연산이 존재하는 경우, 무작위로 연산의 우선 순위를 결정하는 대신, 다른 종류의 우선 순위 결정 조건을 고려하여, 연산들의 우선 순위를 결정하는 것이 더 우수한 스케줄링 결과를 가져올 것이다. 이러한 방법을 적용하면, 최악의 경우라도 기존의 리스트 기반 스케줄링의 성능을 유지한다.

3.2 퍼지 논리를 이용한 우선 순위의 결정

3.2.1 퍼지 제어 규칙

본 논문에서는 기본적인 우선 순위 결정 조건으로 연산의 이동성 범위를 적용하였다. 동일한 우선 순위 함수 값을 갖는 경합하는 연산들에 대해서는 아래와 같은 퍼지 제어 규칙을 적용하여 최종적인 우선 순위를 결정하였다.

IF

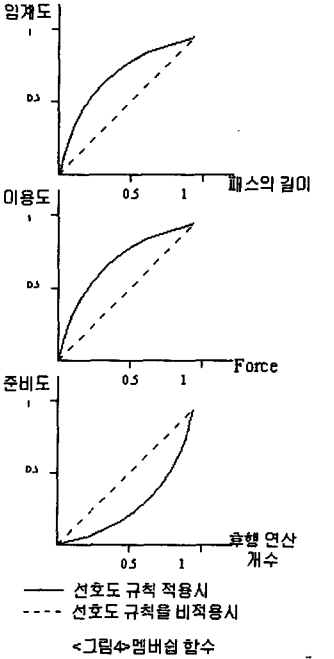
최종 후행 연산까지의 패스가 길다(임계도) ***AND***
연산 상호간의 병렬성 반발도가 크다(이용도) ***AND***
후행 연산의 개수가 많다(준비도) ***AND***

THEN

연산의 우선 순위가 높다.

3.2.2 멤버쉽 함수(Membership Function)

임계도, 이용도, 준비도에 대한 각각의 멤버쉽 함수는 아래와 같다.



<그림4>멤버쉽 함수

3.2.3 퍼지 보상 연산자(Compensatory Operator)

상기의 퍼지 제어 규칙에서 퍼지 AND 연산자는 아래와 같은 보상 연산자(Compensatory Operator)를 사용하였다. 보상 연산자를 사용한 이유는 각각의 우선 순위 결정 조건들을 보다 규형있게 고려하기 위해

서이다. 비보상 연산자를 이용하는 경우, 연산의 약요소(Week Element)는 강요소(Strong Element)를 보상할 수 없고, 따라서 약요소에 담긴 의미는 무시된다. 아래의 식은 본 논문에서 사용된 퍼지 교집합(Intersection)연산자이다.

파라미터 α의 값이 결정되었을 때

$\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}$ if $\mu_A, \mu_B \in [\alpha, 1]$

$\mu_{A\cap B}(x) = (\mu_A(x) \cdot \mu_B(x)) / \alpha$ otherwise

3.2.4 퍼지 선호도 규칙(Preference Rule)

경합하는 연산의 우선 순위를 결정할 때 적용하는 우선 순위 결정 조건들 사이에 위상을 재정립하기 위해 퍼지 선호도 규칙을 적용할 수 있다. 선호도의 정도(Degree of Preference)를 결정하는 특별한 규칙은 없으나, 본 논문에서는 임계도와 이용도의 퍼지 멤버쉽 함수 값에 1/2 승을 하여, 낮은 선호도를 주었고, 이용도의 멤버쉽 함수 값에는 2제곱을 하여, 선호도를 높혀 주었다.

4. 개선된 알고리즘

개선된 알고리즘을 최종적으로 정리하면 다음과 같다.

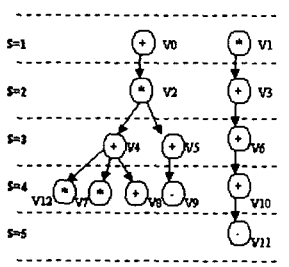
```
DETERMINE_OPs_PRIORITY(Scurrent)
INSERT_READY_OPS(V, PList_t1, PList_t2, ... PList_tm)
Cstep = 0
while((PList_t1 ≠ Null)or... or(PList_tm ≠ Null)) do
  Cstep = Cstep + 1
  for k = 1 to m do
    for funit = 1 to Nk do
      if Plisttk ≠ Null then
        SCHEDULE_OP(Scurrent, FIRST(PList_tk), Cstep)
        Plist_tk = DELETE(Plist_tk, FIRST(PList_tk))
      endif
    endfor
  endfor
  DETERMINE_OPs_PRIORITY(Scurrent)
  INSERT_READY_OPS(V, PList_t1, PList_t2, ... PList_tm)
endwhile
```

* 함수 ***DETERMINE_OPs_PRIORITY(Scurrent)*** 는 각 연산의 우선 순위 함수 값을 구하여 선호도 규칙을 적용한다.

* 함수 ***INSERT_READY_OPS(V, PList_t1, PList_t2, ... PList_tm)***는 준비된 연산들을 이동성 범위에 입각하여 우선 순위 리스트에 저장하되, 동일한 우선 순위 값을 갖고 경합하는 연산에 대해서는 다른 우선 순위 결정 조건들을 참조하여 최종 우선 순위를 결정한다.

5. 스케줄링 예제

제안한 알고리즘의 작동을 설명하기 위해 <그림5>의 DFG를 입력으로 하는 스케줄링을 실시하였다. 본 예제에서는 리스트 스케줄링의 기본 우선 순위로 연산의 이동성 범위를 이용하였으며, 하드웨어 자원은 승산기 ,가산기, 감산기 각각 1기씩으로 제한하였다. <그림6>의 스케줄링 결과를 통해 새롭게 제시된 알고리즘을 이용한 경우가 기존의 방법에 비해 더 좋은 성능을 보임을 알 수 있다. 이 것은 스케줄링의 4번째 단계에서 V4, V5, V10이 동일한 우선 순위로 경합할 때, 새로 제시한 알고리즘이 후행 연산의 개수가 더 많은 V4를 선택하였기 때문이다.



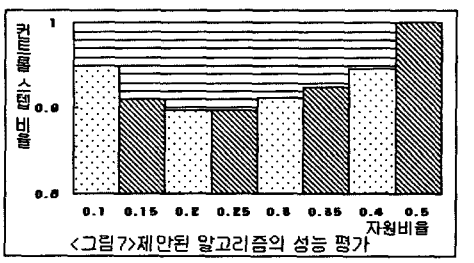
<그림5> 예제 DFG입력

	기존의 방식	제안된 알고리즘
스텝1	V0, V1	V0, V1
스텝2	V2, V3	V2, V3
스텝3	V6	V6
스텝4	V5	V4
스텝5	V9, V10	V7, V10
스텝6	V4, V11	V12, V5, V11
스텝7	V7, V8	V8, V9
스텝8	V12	---

<그림6> 스케줄링 결과

6. 실험결과

본 논문에서는 제안한 알고리즘과 기존 알고리즘의 성능 비교를 위해 Visual C++을 이용하여 각각을 구현하였다. 임의로 다수의 DFG를 생성하여 그 입력으로 하였으며, 하드웨어 자원의 비율을 조정하여, 각각의 결과를 살펴보았다. <그림7>은 기존 알고리즘에서 요구되어지는 컨트롤 스텝의 수를 1로 보았을 때의 성능비교이다. 실험 결과, 제안된 알고리즘을 사용하는 경우가 컨트롤 스텝의 수를 적게 요구하는 스케줄링 결과를 생성함을 알 수 있다. 컨트롤 스텝 비율의 극점, 즉, 유효한 하드웨어 자원의 수가 극단적으로 많다고 가정한 경우에는 성능 향상의 정도가 둔화된다. 이것은 경합하는 연산이 적게 발생하기 때문이다. 그러나, 이 경우에서도 기존 리스트 스케줄링의 성능 이하로 저하되는 경우는 없으며, 또한 이러한 극단적인 경우에서는 하드웨어의 자원 제약 문제에 중점을 두고있는 리스트 스케줄링 자체가 커다란 의미를 갖지 못한다. (단, 자원비율 = 유효한 하드웨어 자원 / 총 자원, 컨트롤 스텝 비율 = 제안된 알고리즘의 컨트롤 스텝의 수 / 기존 알고리즘의 컨트롤 스텝 수)



<그림7>제안된 알고리즘의 성능 평가

7. 결론

본 논문에서는 단일한 우선 순위 조건하에서 경합하는 연산들의 우선 순위 결정시 다양한 우선 순위 결정 조건들을 균형있게 고려하여 보다 효율적인 스케줄링 결과를 얻는 방법을 제시하였다. 제시된 우선 순위 결정 조건들은 퍼지 선호도 규칙을 조정하여 강조하거나 반대로 약화할 수 있었다. 또한 우선 순위 결정 조건이 지닌 의미를 폭넓게 반영하기 위해 퍼지 보상 연산자를 이용하였다. 새롭게 제시된 알고리즘은 최악의 경우라도 기존의 리스트 기반 스케줄링의 성능을 유지할 수 있다.

참고문헌

[1] Daniel D. Gajski and Nikil D. Dutt and Allen C-H Wu and Steve Y. L. Lin, "High-level synthesis: Introduction to chip and system design," Kluwer Academic Publisher, 1992.

[2] Pierre G. Paulin and John P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," IEEE Trans. Computer-Aided Design, vol. 8, pp. 661~679, 1989.

[3] Jan Vanhoof et. al., "High-Level Synthesis for Real-Time Digital Signal Processing," Kluwer Academic Publishers, 1993.

[4] B. M. Pangrle and D. D Gajski, "Slicer: A state synthesizer for intelligent silicon compilation," in Proc. of IEEE Int. Conf. on Computer Design, Oct. 1987.

[5] S. Davidson et. al., "Some experiments in local microcode compaction for horizontal machines", IEEE Trans. on Computers, pp. 460-477, July 1981.