Packet Communication에 의한 Demand-Driven Dataflow 컴퓨터 構造에 관한 研究

(A Study on Demand-Driven Dataflow Computer Architecture based on Packet Communication)

李相範,* 柳根鎬,** 朴圭泰***

(Sang Burm Rhee, Keun Ho Ryu and Kyu Tae Park)

要 約

Dataflow 컴퓨터는 기존 von-Neumann 컴퓨터 構造에서 갖기 어려운 高度의 竝列性을 제공하므로 계산 처리능력을 향상시키게 된다.

본 논문에서는 MIT dataflow 기계구조에 demand propagation network을 포함시킨 demand-driven dataflow 구조를 제안하고 時間加速法에 의한 dataflow 시뮬레이션을 통하여 그 構造의 타당성을 보였다.

또한, 프로그램의 수행시간 및 연산처리 요소의 실효 이용률을 구하여, MIT dataflow 기계구조와 제안된 구조를 비교함으로써, 제안된 구조가 연산처리능력을 향상시킬 수 있음을 보였다.

Abstract

Dataflow computers exhibit a high degree of parallelism which can not be obtained easily with the conventional von-Neumann architecture. Since many instructions are ready for execution simultaneously, concurrency can easily by achieved by the multiple processors modified the data-flow machine. In paper, we describe an improved dataflow architecture which is designed by adding the demand propagation network to the MIT dataflow machine, and show the improved performance by the execution time and the efficiency of processing elements through simulation with the time acceleration method.

*正會員, 檀國大學校 電子工學科
(Dept. of Elec. Eng., Dan Kook Univ.)

**正會員, 忠北大學校 電算統計學科
(Dept. of Com. Soi and Sta., Chung Buk Univ.)

***正會員, 延世大學校 電子工學科
(Dept. of Elec. Eng., Yon Sei Univ.)

(※ 본 논문은 한국과학재단의 지원에 의하여 이루어진 것임.)

接受日字 : 1985年 12月 12日

I. 序 論

현재 사용되고 있는 von-Neumann 構造는 一次元的인 住所空間과 演算處理가 順次的이고 中央集中式 制御이므로, 복잡한 계산이 필요한 과학계산 분야에서 심각한 장해가 되고 있다. 이러한 문제를 해결하려면 컴퓨터는 高度의 竝列性(high degree of parallelism) 을 갖도록 설계되어야 한다. 그 대표적인 例는 SIMD 와 MIMD 構造를 갖는 ILLIAC Ⅳ와 C. mmp[1]시스템을 들 수 있다. 그러나 이들은 첫째, 演算 處理裝置의 數를

임의로 조절하는 것이 곤란하며, 둘째, 여러개의 演算處理裝置가 하나의 기억장치에서 데이타를 동시에 액세스 할 수 없다는 점에서 演算速度를 개선하기에는 한계가 있다.[2] 따라서 컴퓨터의 性能을 향상시키기 위한 새로운 構造의 접근이 필요하게 되었다.

최근 몇년 동안 여러 연구기간에서는 制御裝置 및 프로그램 카운터의 개념을 갖는 von-Neumann 構造의 취약점을 보완하기 위하여 packet communication 기계조직을 바탕으로 하는 dataflow 構造가 제시되고 있다.[3] 이 構造의 기계어 프로그램은 dataflow 프로그램 그래프에서 나타낼 수 있게 되어 高度의 竝列性을 갖게 된다.

Dataflow 프로그램의 이론적 배경은 1966년 Karp와 Miller[4]에 의하여 제시되었으며, 그 이론을 바탕으로 기계구성까지 발전되고 있다. 그 대표적인 例는 MIT의 Dennis가 이끄는 연구팀에서 개발한 dataflow 기계이다.[5,6]

본 논문에서는 MIT dataflow 기계구조를 고찰하고, 이 構造의 취약점을 분석하여 그 개선방법을 제안하였다. 즉, MIT 構造는 non-strict함수[7]를 수행할 때에 불필요한 演算을 수행하게 되어 演算 處理裝置를 효율적으로 이용하지 못하므로 프로그램의 수행 속도가 크게 개선되지 않는다. 이러한 단점을 보완하기 위하여 각 actor가 demand를 받을 때에만 수행되게 하는 demand-driven dataflow 構造를 제안하고, 그 타당성을 컴퓨터 시뮬레이션을 통하여 보였다.

II. Dataflow 演算 組織

1. Dataflow프로그램 그래프

일반적인 dataflow 기계어(Functional Graph Language) 중에서 널리 알려진 Dennis의 언어는 actor, arc 및 token의 세가지 기본개념을 갖는 dataflow 프로그램 그래프로 표시된다.[8] 예를 들어,

IF X>0 THEN output := (A+1) * 4;
ELSE output := ((B-2)*C) * 4;

와 같은 조건문을 수행하는 dataflow 프로그램 그래프는 그림 1과 같다.

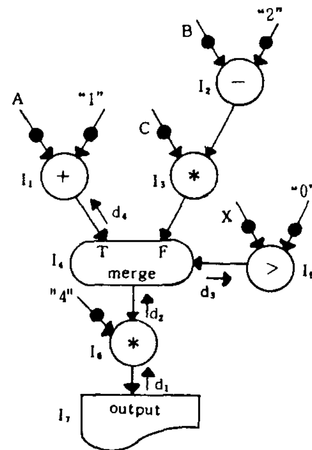


그림 1. Dataflow 프로그램
Fig. 1. A dataflow program graph.

그림 1에 나타내어진 dataflow 프로그램 그래프는 +, −, *, >, merge, output의 기능을 수행하는 actor와 데이타를 나타내는 검은 원의 token, token의 흐름을 표시하는 단일 방향의 arc로 구성되어 있다.

Dataflow 프로그램 그래프의 구성에 기본적으로 필요한 actor는 +, −, * 등의 strict actor와 first, rest, cons 등이 있으며 그 기능은 다음과 같다.

1) strict actor $([a_i], [b_i], nil) = nil$
strict actor $([a_i], [b_i], [c_i]) = f(a_i, b_i, c_i)$
2) first(nil) = nil
first$([a_i]) = a_1$
3) rest(nil) = nil
rest$([a_i]) = [a_2, a_3 \cdots\cdots]$
4) cons(nil, $[a_i]$) = nil
cons$([a_i], nil) = [a_1]$
cons$([a_i], [b_i]) = [a_1, b_1, b_2, b_3 \cdots\cdots]$

또한 복잡한 演算이 요구되는 고급 dataflow 프로그램 그래프의 용이한 구성을 위하여 다음과 같은 actor를 정의할 수 있다.

1) t-gate$([a_i], [T, F, T, T \cdots\cdots]) = [a_1, a_3, a_4 \cdots]$
2) f-gate$([a_i], [F, T, T, F \cdots\cdots]) = [a_1, a_4 \cdots\cdots]$
3) merge$([a_i], [b_i], [T, F, T, T \cdots\cdots]) = [a_1, b_1, a_2, a_3 \cdots\cdots]$
4) if-then-else$([a_i], [b_i], [T, F, T, T \cdots\cdots]) = [a_1, b_2, a_3, a_4 \cdots\cdots]$
5) fanout$([a_i]) = [a_i], [a_i]$
6) condfanout$([a_i], [T, F, F, T \cdots\cdots]) = [a_1, a_4 \cdots]_T, [a_2, a_3 \cdots\cdots]_F$

Dataflow 컴퓨터를 구성하는 여러가지 방법 중에서 MIT dataflow 컴퓨터 구조는 크게 메모리 장치(memory unit), arbitration/distribution network 및 처리장치(processing unit)의 3부분으로 나누어졌고, 각 장치들 사이에서 서로 독립적으로 패키트를 pipeline형태로 주고 받는 packet communication 기계조직에 바탕을 두고 있는 data-driven 기계조직이다.[9]

이때 메모리 장치는 여러 命令語 素子(instruction cell)들로 구성되며, arbitration/distribution network

은 패키트들을 전달하는 일종의 스위칭 망이며, 각 장치들 사이에서 패키트 전달은 非同期的 two-way handshake protocol에 의하여 이루어진다.[10]

2. Data-driven 演算 組織의 問題點 및 解決 方案

Dataflow 프로그램 그래프에서 actor의 수행조건(firing rule)을 만족시키기 위하여 필요한 入力 token을 공급하는 방법으로는 data-driven과 demand-driven이 있다.

Data-driven이란, actor의 수행에 필요한 모든 token들이 존재(available)할 때에만 수행이 이루어지는 演算 組織(computational organization)이다. Demand-driven이란, 수행이 필요한 actor에만 demand라는 신호를 보내어 入力 token을 공급받는 演算 組織이다. 이러한 演算 組織의 특징을 비교하기 위하여 strict 함수와 non-strict함수를 정의한다.

(정의)

1. 함수 f가 매개변수(argument) X에 대하여 strict하다.
   ≡ X가 정의되지 않으면 f가 정의되지 않는다.
2. 함수 f가 strict하다.
   ≡ 모든 매개변수에 대하여 f가 strict하다.

(참고1) 함수 f가 strict하지 않으면 non-strict라 한다.

(참고2) Strict함수에서는 함수가 수행되면 모든 入力 매개변수는 소멸되며, non-strict 함수에서는 사용된 매개변수만 소멸된다.

Dataflow 構造에서 프로그램이 高度의 並列性을 갖기 위하여는 strict 및 LISP의 演算子와 같은 first, rest, cons, merge, arbit 등의 non-strict함수의 命令語들이 기본적으로 필요하다.[7]

Data-driven에서는 모든 actor에 필요한 token이 공급되면 命令語가 수행되므로 strict함수를 수행하기에 적당하다. 그러나 프로그램이 non-strict 함수를 포함하고 있으면 수행이 필요없는 actor까지도 수행하게 되어 자원의 낭비와 처리장치의 효율을 저하시키는 단점을 갖고 있다. 즉, 그림 1에서 actor I_4(merge)는 non-strict함수이므로 actor I_4에서 X의 값이 1이면 actor I_1의 수행만 필요하고 actor I_2, I_3의 수행은 불필요하게 된다. 또한 어떤 프로그램이 non-strict 함수를 포함하고 있을 때에, 결과값을 구하는데 도움이 안되는 무한히 긴 연산(unbounded computation)을 제한된 처리장치가 수행하게 된다면 처리장치의 효율이 저하되어 프로그램 수행속도를 저해하는 원인이 된다. 이러한 문제를 해결하는 방법으로는 non-strict함수의 入力중에 불필요한 token을 공급하지 않는 방법도 있으나, multiprocessor에서 프로그램이 수행되는 동안 이를 시행하기에는 극히 비효율적이다.[7] 다른 해결방법은 결과값을 구하는데 필요한 actor만 수행되도록 선택적으로 demand를 공급하는 demand-driven 演算組織이다. 이때 demand는 프로그램 수행순서와 반대방향으로 전달되며, 각 actor는 demand를 받아서 入力 token이 필요한 actor로 보낼 수 있어야 한다. 그러나 strict함수에서도 전달되어야 하므로 data-driven에 비하여 demand를 전달하는 지연시간이 추가되는 단점이 있다.

그러나, demand의 전달과 actor의 수행을 concurrent하게 구성하면 demand를 전달하는 추가시간을 줄일 수 있다. 또한 演算處理裝置의 처리시간은 demand 전달시간에 비하여 상당히 크므로 demand 전달시간은 실제로 거의 무시할 수 있다.

Ⅲ. Demand-driven Dataflow 컴퓨터 構造의 提案

MIT dataflow 기계구조는 non-strict 함수를 수행할 때 처리장치의 효율이 저하된다. 이러한 문제를 해결하기 위하여 MIT dataflow 기계구조에 demand propagation network을 포함시켜 그림 2와 같은 demand-driven dataflow 構造를 설계하였다.

이 기계구조는 MIT 기계와 다음과 같은 점에서 달리한다.

1) MIT dataflow 기계구조는 data-driven인데 반하여, 제안된 구조는 demand-driven dataflow 기계구조이다.

2) MIT dataflow 기계구조에 비하여 demand를 전달하는 demand propagation network이 추가된다.

3) 명령어 형식이 data-driven에 비하여 demand를 전달할 source address 및 demand 상태를 표시할 수 있는 demand state의 필드가 포함되어야 한다.

4) 프로그램의 수행에서 data-driven이 유리하면 명령어의 demand state를 data-driven으로 初期化시킴으로써 data-driven과 똑같이 프로그램이 수행될 수 있다.

그림 2의 각 부분의 기능은 다음과 같다.

1. 메모리 장치

메모리 장치는 여러 命令語 素子로 구성된다. 프로그램이 수행되려면 demand를 전달해야 하므로 명령어 소자에 demand상태를 표시하는 demand state(i, p, s) 필드를 두어야 하며 그 명령어 형식은 다음과 같다.

명령어 형식 :

〈opcode, operand, destination/port, source addr., demand state〉

예를 들어, 다음과 같이 non-strict함수를 포함하는

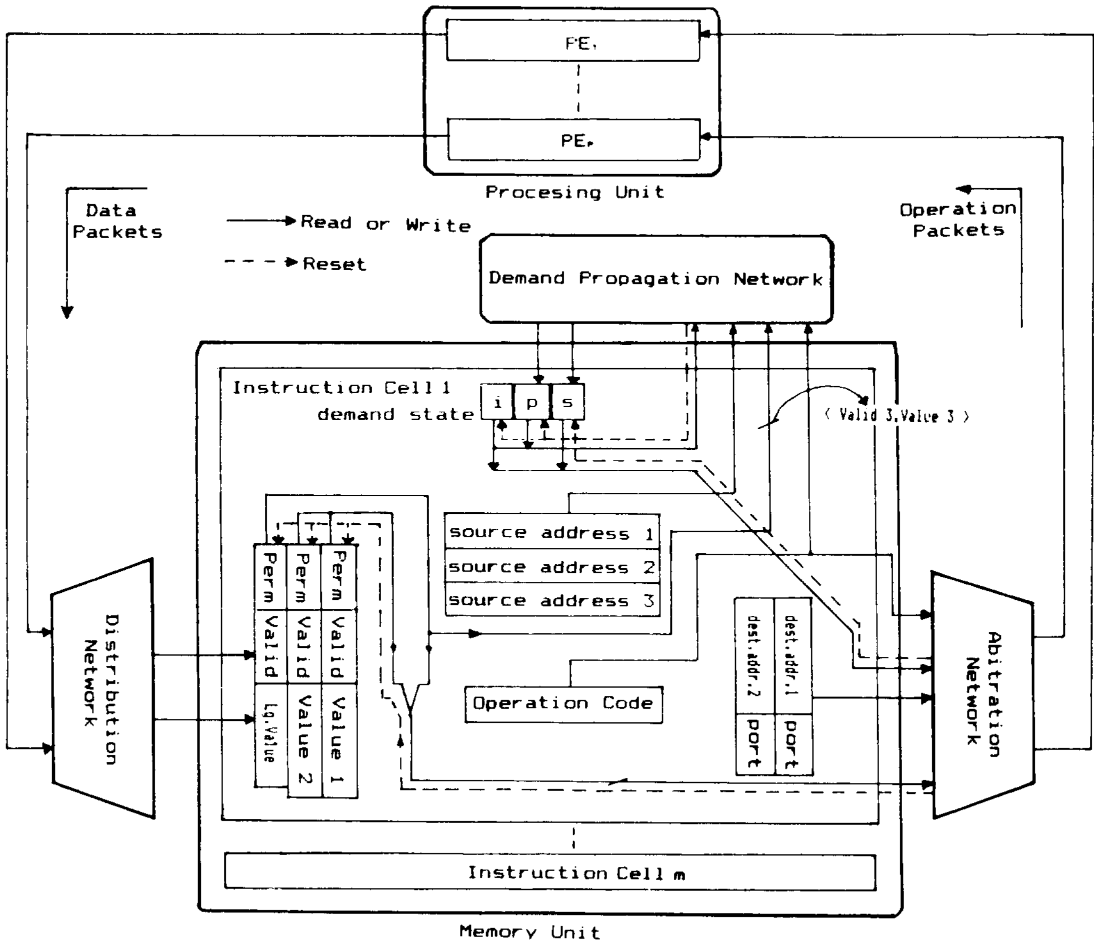


그림 2. 제안된 demand-driven dataflow 컴퓨터 構造

Fig. 2. The proposed demand-driven dataflow computer architecture.

프로그램을 제안된 demand-driven dataflow 構造에서

```
PROGRAM example (input,output);
VAR A,B,C,D,E:real;
    I,J,X,Y:integer;
BEGIN
  read (A,C,D,E,X);
  B := (A-1)*2;
  FOR I := 1 TO X DO
    BEGIN
      B := B-2;
      Y := B-B;
      IF B > 0 THEN B := ((B/2)-(B*0.1))+1
              ELSE FOR J := 1 TO Y DO
                     B := (E*E-(C/2+D))/A;
    END;
  writeln ( B );
END.
```

수행시키는 dataflow 프로그램 그래프는 그림 3 과 같다.

그림 3 의 dataflow 프로그램 그래프에서 명령어 소자의 初期 상태는, 표 1 과 같은 명령어 형식으로 데이타가 메모리에 저장되어야 한다.

명령어 소자의 opcode는 수행해야 될 기능(function)을 표시하고, 오퍼런드의 valid는 value가 있는 상태를 표시하며, perm은 수행된 후 오퍼런드가 리세트되는 상태를 나타낸다. Source address는 demand를 전달할 명령어 소자를 표시하며, demand state(i, p, s) 의 i는 초기 비트이며, p는 demand 전달상태, s는 demand를 받은 상태를 표시한다. 이때 demand를 순환전달(circular propagation)하기 위하여, 메모리 장치는 다음 그림 4 의 알고리즘과 같은 知能(intelligence) 를 갖도록 설계되어야 한다.

이때 output 명령어는 入力 token이 없을 때 처음 demand를 전달할 수 있게 되며, condfanout은 첫번째

그림 3. Dataflow 프로그램 그래프
Fig. 3. A dataflow program graph.

표 1. 명령어 소자의 초기값
Table 1. Initial values of the instruction cells.

Instruction Cell	Operation Code	Operand oprd no.	Operand perm	Operand valid	Operand value	Destination oprd no.	Destination address	Destination port	Source Address		Demand State	
1	—	1	False	True	A	1	2	1	1	0	i	False
		2	True	True	1	2	0	0	2	0	p	False
		3	True	True	0				3	0	s	False
2	*	1	False	False	0	1	3	2	1	1	i	False
		2	True	True	2	2	0	0	2	0	p	False
		3	True	True	0				3	0	s	False
3	merge	1	False	False	0	1	4	1	1	17	i	False
		2	False	False	0	2	0	0	2	2	p	False
		3	False	False	0				3	8	s	False
45	output	1	False	False	0	1	0	0	1	18	i	False
		2	True	True	0	2	0	0	2	0	p	True
		3	True	True	0				3	0	s	True

수행된 후 入力 token이 없을 때 demand를 선달할 수 있게 된다.

2. Demand Propagation Network

Demand propagation network는 actor의 demand전달 조건에 따라 그림5와 같이 demand를 전달하는 기능을 갖게 된다.

```
PROGRAM [illegible]
BEGIN
   WHILE running DO    { while program execution
      FOR i:=1 TO number of instruction cell DO
         WITH instruction cell[i]
         CASE opcode OF
            merge : IF p,s = True,True THEN i := True;
            fanout : IF p,s = False,True THEN i := True;
                     IF i = True THEN i := False;
            fork : IF destination[1].demand state(p,s) = False,True
                       THEN i:=True;    { for destination address select }
            condfanout : BEGIN
                           IF p,s = False,True THEN i := True;
                           IF i = True THEN i := False;
                           IF i,s = True,False
                              AND operand[1].valid = False
                              AND operand[2].valid = False
                                 THEN p,s := True, True;
                         END;
            output : IF operand[1].valid = False AND
                        demand state(p,s) = False,False
                         THEN p,s:=True,True;    { for initial demand propagation }
         END;
END.
```

그림 4. 메모리 장치의 知能
Fig. 4. The intelligence of memory unit.

그림5에서 보인 demand 전달조건 중에서 d-gate actor가 포함된 fork의 기능을 살펴보자. 2개의 出力 arc를 갖고 있는 fork는 어느 한 出力 arc로부터 demand를 받으면 入力 token을 해당되는 出力 arc로 공급하게 된다. 이때 어느 한 쪽에서 d-token을 받은 d-gate는 fork의 入力 token을 요구하는 demand를 생성하여 出力시킨다. d-gate가 최초의 d-token을 어느한 쪽에서 받은 다음에 다른 쪽에서 요구된 d-token은흡수되어 다른 出力 arc로는 出力이 이루어지지 않는다.

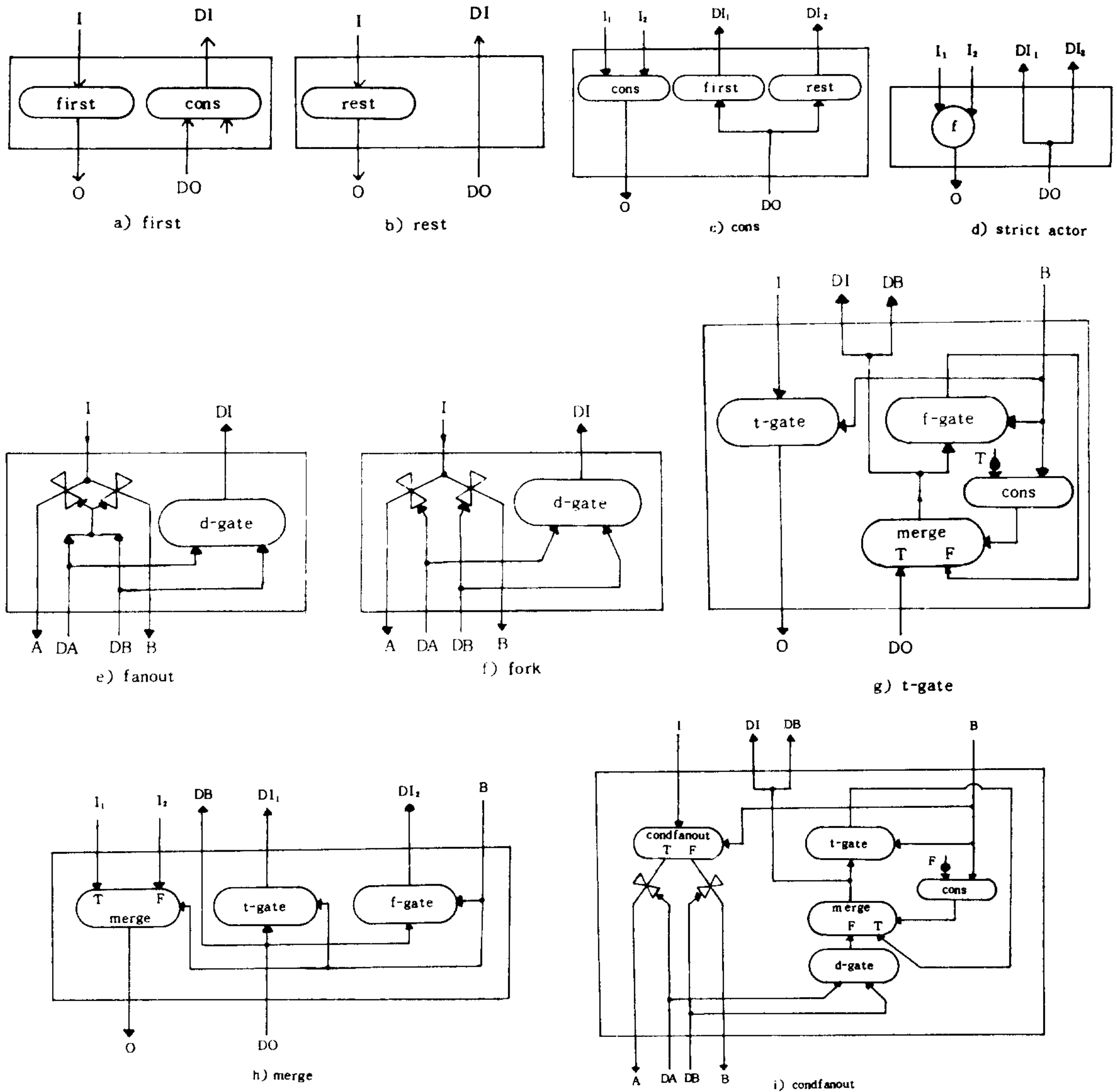


그림 5. Demand 전달조건
Fig. 5. Propagating demands.

예를 들어, d-gate의 入力線 A, B에 공급된 d-token의 순서가 A(1), A(2), B(1), B(2), B(3)… 일 때 fork의 出力은 A(1), A(2), B(3)…이 된다. 그림 ▷◁는 단순히 gate를 나타내며 d-token을 받으면 入力된 token을 出力시키는데, 그 동작은 다음과 같이 나타낼 수 있다.

gate(X, nil) = nil

gate(X, d) = X

이러한 demand의 전달 기능을 그림 3에서 살펴보면, output actor로부터 첫번째 demand d_1을 전달하게 되고, 순차적으로 d_8이 actor I_8까지 전달되면 actor I_8은 수행조건이 만족되어 수행하게 되고, 그동안 d_9, d_{10} 등이 처리장치의 수행과 concurrent하게 전달된다. 처음 demand를 전달하기 위하여 output actor가 메모리에서 demand state(i, p, s) = False, True, True의 상태로 되면 demand propagation network은 p비트를 리세트시키고 source address가 지정하는 명령에 소자로 demand를 전달한다. 그후 actor I_{17}로 부터 똑같은 방법으로 demand가 전달된다. 예외로 non-strict함수 merge인 경우 demand를 두번 전달해야 하

므로, 처음 demand를 전달할 때에는 demand state i 비트를 리세트시키고 두번째 전달할 때에는 demand state p비트를 리세트시킨다. 이때 actor가 strict 함수인 경우 모든 入力 port에 demand를 전달해야 하고, non-strict함수인 경우에는 入力 token이 필요한 방향으로 demand를 전달하여야 한다. 이러한 demand propagation network의 기능은 그림 6과 같다.

또한 프로그램을 data-driven으로 수행시키려면, demand state (i, p, s) 필드의 p와 s의 flag 비트를 하드웨어적으로 p=false, s=true로 세트시키면 가능하게 된다.

```
PROGRAM DF-netSimulator;
BEGIN
   WHILE running { program running }
      DO BEGIN
         FOR i := 1 TO number of instruction cell
            DO WITH instruction cell[i]
               DO BEGIN { operation of demand propagation network }
                  IF demand status(p,s) = True, True
                     THEN BEGIN
                        CASE opcode OF
                           first: IF demand status(i)
                                     THEN demand status(p,s) := False, False;
                                     ELSE    DI - propagation
                           rest:     DI - propagation
                           cons: IF demand status(i) THEN    DI 2 - propagation
                                                     ELSE    DI 1 - propagation
                           strict actor:     DI n - propagation
                           fanout: IF demand status(i)
                                     THEN demand status(i, p) := False, False
                                     ELSE BEGIN
                                        DI - propagation
                                       demand status(i, p) := True, False;
                                     END;
                           form: BEGIN
                                    IF destination[1]. demand ststus(s)
                                      THEN demand status(i) := True;
                                       DI- PROPAGATION
                                 END;
                           tior ti-gate: BEGIN
                                             DI, DB - propagation
                                          IF operand[3]. valid
                                             THEN BEGIN
                                                IF operand[3]. value = 0
                                                   THEN demand status(i) := True;
                                             END;
                                       END;
                           merge: BEGIN
                                     IF NOT demand status(i)
                                        THEN BEGIN
                                              DB - propagation
                                          demand status(i, p) := True, True;
                                        END
                                        ELSE BEGIN
                                          IF operand[3]. valid
                                             THEN BEGIN
                                                IF operand[3]. value = 1
                                                   THEN    DI 1 - propagation
                                                   ELSE    DI 2 - propagation
                                                demand status(i) := False;
                                             END;
                                        END;
                                  END;
                           condfanout: BEGIN
                                              DI, DB - propagation
                                          IF operand[3]. valid
                                             THEN BEGIN
                                                IF operand[3]. value = 1
                                                   THEN demand status(i) := True;
                                             END;
                                       END;
                           output:      DI - propagation
                        END
                        ELSE BEGIN
                           IF opcode = t or f-gate OR opcode = condfanout
                              THEN BEGIN { demand refresh }
                                 IF demand status(i)
                                    THEN demand status(p, s) := True, True;
                              END;
                           IF opcode = output AND (NOT operand[1]. valid) AND
                                 (NOT demand status(s))
                              THEN demand status(p, s) := True, True;
                        END;
                     END;
      END;
END.
```

그림 6. Demand propagation network의 기능

Fig. 6. The operation of demand propagation network.

3. 演算 處理裝置

연산처리장치는 여러개의 연산처리요소(processing elements)로 구성되며 arbitration network로부터 다음 형식의

operation packet

〈opcode, operand, destination/port〉

를 받아, 데이타를 처리한다. 이러한 연산처리장치는 그림 7과 같은 데이타 처리경로를 통하여, 다음 형식의

data packet

〈destination/port, result value〉

를 distribution network에 보내게 된다. 또한 연산처리장치는 산술, 논리연산, 비교 뿐만 아니라 入.出力장치도 포함된다.

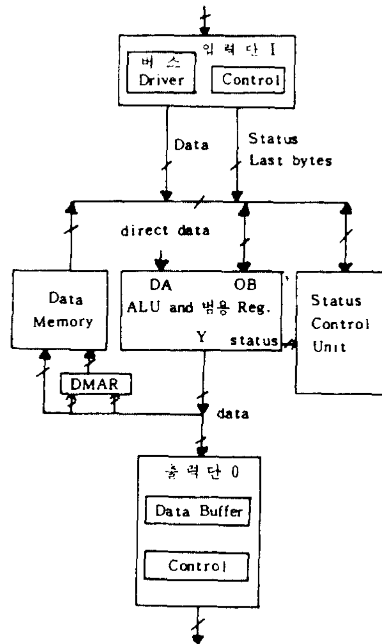


그림 7. 연산처리요소의 데이타선송경로

Fig. 7. Data paths of the processing elements.

4. Arbitration 및 Distribution Network

Arbitration network은 연산처리요소(PE)가 수행가능할(idle) 때, 수행조건을 만족하는 명령어 소자의 데이타를 액세스하여 operation packet을 만들어 보낸다. 또한 distribution network은 연산처리장치로부터 데이타 패키트를 받아 destination이 지정하는 명령어 소자의 오퍼런드에 결과값을 저장하는 그림 8과 같은 스위칭 망이다.

예를 들어, N×N 스위칭 망은 $(N/2)\log_2 N$개의 2×2 스위칭 망으로 구성되고, 패키트를 전달할 때의 지연시간은

```
PROGRAM Abitration/DistributionNetwork
BEGIN
  WHILE running DO  {  program running  }
    BEGIN
      FOR i := 1 TO number of instruction cell
        WITH instruction cell[i]
        BEGIN { operation of abitration }
          go := True ;  { firable instruction cell }
          FOR j := 1 TO number of input port
            IF operand[j].valid = False { no token }
              THEN go := False ; { not firable }
          FOR j := 1 TO number of output port DO
            IF instruction cell[dest[j].addr].operand[dest[j].port].valid
              THEN go := False;
          cp := 0;
          FOR j := 1 TO number of processing element DO
            IF processing element[j] = busy { executing }
              THEN cp:=cp+1;
          IF cp = number of processing element { not idle }
            THEN go := False;
          IF go { firable instruction cell }
            THEN BEGIN
              { mark operand buffers as empty }
              FOR k=1 TO number of input port DO
                IF operand[k].perm = False
                  THEN operand[k].valid := False;
              BEGIN
                { send operation packet to idle processor };
                  processor execution & make data packet }
              END;
              BEGIN  { operation of distribution network }
                FOR m := 1 TO output port DO
                  BEGIN
                    ds := dest[m].addr; {data packet}
                    dp := dest[m].port; {data packet}
                    IF ds <> 0
                     THEN BEGIN
                      instruction cell[ds].operand[dp].valid:=True;
                      instruction cell[ds].operand[dp].value := result;
                     END;
                  END;
              END; { end of distribution network }
            END;
        END; { end of abitration network }
    END;
END.
```

그림 8. Arbitration 및 distribution network의 기능

Fig. 8. The operation of arbitration/distribution network.

$$t_{nd}(N \times N) = t_{nd}(2 \times 2) \cdot \log_2 N \qquad (1)$$

이 된다. 2×2 스위칭 방은 2 개의 入力 port에서 패키트를 받아들여, 패키트에 포함된 住所에 따라 선택된 임의의 한 出力 port로 그 패키트를 보내게 한다. 패키트는 도착한 순서대로 통과하며 2 개의 出力 port는 동시에 出力을 내보낼 수도 있다.

Ⅳ. 性能 解析 및 시뮬레이션 結果

1. 性能 解析

Dataflow 컴퓨터의 성능은 프로그램 수행시간에 따라 결정될 수 있다. 프로그램 수행시간을 결정하기 위하여 다음을 정의한다.

- p : 연산처리요소(PE)의 數
- n : 수행될 actor의 총수(n' : data-driven, n'' : demand-driven)
- α : 연산처리장치의 수행확률(α' : data-driven, α'' : demand-driven)
- t_{fi} : i번째 actor의 수행시간
- t_a : arbitration network의 스위칭 시간
- t_s : distribution network의 스위칭 시간
- n_d : 연산처리장치가 idle할 때의 demand전달 數
- t_d : demand 전달 시간

이상과 같은 정의에 의하여 data-driven 구조의 프로그램 수행시간은 n'의 명령어를 메모리에서 액세스하여 연산처리장치로 보내는 시간, 연산처리장치의 수행시간 및 그 결과를 메모리에 저장하는 시간의 合이 된다. 즉,

$$T'_{ex}(\text{data-driven}) = \frac{1}{\alpha' p}\left(n' \cdot t_a + \sum_{i=1}^{n'} t_{fi} + n' \cdot t_s\right) \qquad (2)$$

Demand-driven 구조의 프로그램 수행시간 data-driven에 비하여 연산처리장치가 idle할 때 demand 전달 시간이 포함되어야 한다.

$$T''_{ex}(\text{demand-driven}) = \frac{1}{\alpha'' p}\left(n'' \cdot t_a + \sum_{i=1}^{n''} t_{fi} + n'' \cdot t_s\right) + n_d \cdot t_d \qquad (3)$$

프로그램이 高度의 竝列性을 유지할 경우 프로그램 수행시간을 비교하기 위하여 t_{fi} = 상수, $\alpha' = \alpha''$로 가정하고 T''_{ex}/T'_{ex}를 구하면

$$\frac{T''_{ex}}{T'_{ex}} \fallingdotseq \frac{n''}{n'}\left[1 + \frac{(n_d \cdot t_d) \cdot \alpha' p}{n''(t_a + t_{fi} + t_s)}\right] \qquad (4)$$

가 된다. Demand-driven이 data-driven에 비하여 수행시간이 빠를 조건은

$$\frac{T''_{ex}}{T'_{ex}} < 1 \qquad (5)$$

이 되어야 한다. 이 조건이 만족되려면 $n''/n' < 1$이되어야 한다. 또한 프로그램에 따라서 n', n''가 정하여지면 식 (4)의 우변의 값이 작아지기 위해서는 연산처리요소의 數 p가 작은 경우임을 알 수 있다. 즉 demand-driven 구조는 프로그램이 non-strict함수를 포함하고, 프로그램의 크기에 비하여 연산처리요소의 數가 제한된 경우에 효율적임을 알 수 있다. 그리고 실제로는 $\alpha' \geq \alpha''$임에도 불구하고 동일한 프로그램 그래프를 수행할 때 앞에서와 같이 $T'_{ex} > T''_{ex}$이 되어 연산처리요소가 다른 연산을 할 수 있으므로 고도의 병렬처리가 가능하다.

2. 시뮬레이션 結果

식 (2), 식 (3)의 T'_{ex}, T''_{ex}를 구하는 dataflow 프로그램 수행 알고리즘은 時間加速法(time acceleration)으로 시뮬레이션하였고,[13] 그중 T''_{ex}를 구하는 알고리즘은 그림 9와 같다.

이때 각 actor의 수행시간은 비트 슬라이스 마이크로 프로세서 Am2903의 마이크로 싸이클 시간[14]을 기준으로,

$t_f(+) = 184\text{ns}$, $t_f(-) = 184\text{ns}$, $t_f(*) = 200\text{ms}$

$t_f(/) = 228\text{ns}$, $t_f(\text{merge, fanout, condfanout}) = 143\text{ns}$

로 정하였으며 arbitration/distribution network은 이상적인 routing network이라 가정하고, 각 network의

```
PROGRAM DemandDrivenDataflowSimulator;
BEGIN
  read << number of instruction cells & procesors, memory unit >>.
  FOR processors := 1 TO number of processors
    DO BEGIN
      prog := save; { save initial state of instruction cells }
      running := true; { program execution }
      << initialize variables go(firable instr.), fetch(instr. fetch state),
        processor, ptime & paddr & pstate (end execution time & executing
        instr. cell & processor executing state), demand status(i, p, s) >>
      WHILE running
        DO BEGIN
          << check demand status of instuction cells >>
          << check fetch state of processors >>
          IF demandcheck { exist demand propagation }
            THEN << demand propagation >>
          IF demcheck AND (NOT fetcheck)
            { demand propagation and all processors is idle }
            THEN BEGIN
              totdmtime := totdmtime + dmtime;
              { total demand propagation time }
              presenttime := presenttime + dmtime;
            END;
          IF NOT demandcheck { not exist demand propagation }
            THEN BEGIN
                << select the least fetchtime & paddr of processor >>
              presenttime := leasttime;
              { the least fetch time of the processors }
                << execute processor have least fetchtime & make data packets >>
              << operation of distribution network >>
              IF all instruction cells executed
                THEN BEGIN
                  running := False;
                    << select max fetchtime of processors, write execution time >>
                END;
            END;
          FOR i := 1 TO number of instruction cells
            DO BEGIN { satisfy the firing rule }
                << check firable state of instruction cell[i] >>
              IF << instruction cell[i] is firable >>
                THEN instruction cell[i]. go := true;
            END;
          FOR i := 1 TO number of instruction cells
            DO BEGIN { arbitration network }
              << check any idle processors ( exist executable processor ;
                firable instruction fetch >>
              instruction cell[i]. go := false;
              fetchtime := presenttime + executiontime;
            END;
        END;
    END;
END.
```

그림 9. 시뮬레이션 알고리즘

Fig. 9. Simulation algorithm.

스위칭 시간을 정하기 위하여 packet module이 명령어 소자를 검색하는 시간은 CAM(content addressable memory)의 속도를 기준[15]으로 하여 35ns로 하고, router의 전달지연 시간은 75ns로 하였을 때

$t_a = 110\text{ns}$, $t_s = 110\text{ns}$

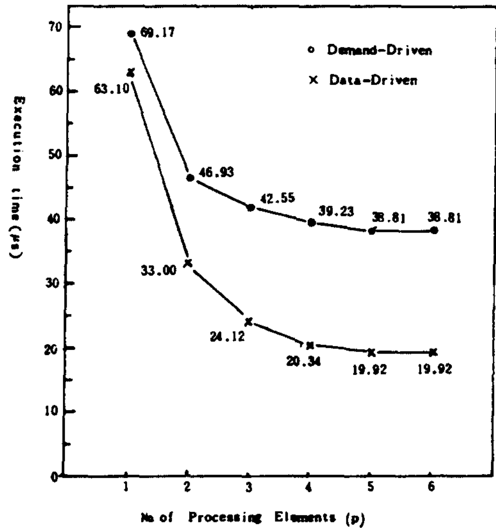
로 하였다. 이때 router의 전달지연 시간은 명령어 소자의 수를 256개 이내로 하여 256×256router로 구성했을 때 2×2router의 8단계 지연시간이 필요하므로 NAND gate(SN74S30)의 지연시간 3ns, MUX(SN74 AS158)의 지연시간 1.4ns, D-latch(SN74AS33) 의 지연시간을 기준으로 하면 2×2router의 지연시간은 9.4ns이 되어 256×256router의 총 지연시간은 약 75 ns가 된다. 또한 demand propagation network의 demand 전달시간은 arbitration network 및 distribution network의 지연시간과 동일하다고 가정하여

$t_d = 110\text{ns}$

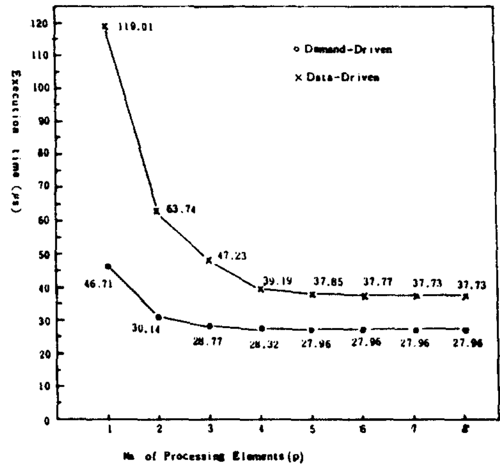
로 하였다.

연산처리요소의 數를 변화시켜가며 프로그램의 수행시간(execution time)을 비교한 결과는 그림10과 같다.

그림10(a)는 strict함수로만 구성된 2차미분방정식 $y''+2xy'+2y=0$(정의구역 $x \leq 2$, x의 증분(step size) $dx=0.2$, 초기조건 $y'(0)=0$, $y(0)=1$)의 解를 구하는 프로그램을 수행시킨 결과로 $n'=n''$가 되어 data-driven이 유리함을 알 수 있다. 그러나, 그림10(b)는 non-strict함수를 포함한 그림3의 프로그램 例인 (A=20, C=2, D=1, E=2, X=5)를 수행시킨 결과이다. 이 경우에 $n' > n''$가 되어 demand-driven이 data-driven에 비하여 효율적임을 볼 수 있다. 또한 그림3의 dataflow 프로그램 그래프에서 출력값을 구하는데 이용된 연산처리장치의 實效利用率 e는



(a) strict 함수의 프로그램



(b) non-strict 함수가 포함된 프로그램

그림10. 프로그램 수행시간

Fig. 10. Program execution time.

$$e = \frac{\sum_{i=1}^{n''} (t_a + t_{fi} + t_s)}{\text{프로그램수행시간} \times \text{연산처리요소(PE)의 수}} \times 100$$

으로 정의할 수 있으며 이를 구하면 그림11과 같이 되므로, demand-driven의 경우에 연산처리요소가 불필요한 actor를 수행하지 않음을 알 수 있다. 그리고 이

프로그램을 수행할 때 105단계(2.1 μs)의 demand 를 전달하였으나 연산처리장치의 수행과 69단계의 demand는 actor의 수행과 concurrent함으로써 n_d가 40단계(0.8 μs)의 demand를 전달한 것으로 되었다. 따라서 실제적으로는 이 demand 전달시간은 연산처리장치의 수행시간에 비하여 무시할 수 있는 시간이다.

그러므로 non-strict함수를 포함하는 프로그램에서 연산처리요소의 數가 제한될 경우에는 demand-driven dataflow構造가 효율적임을 알 수 있다.

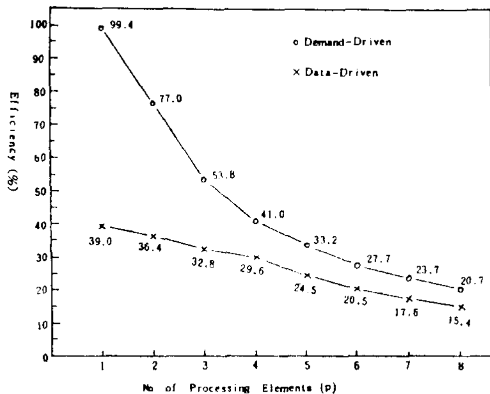


그림11. 연산처리장치의 실효이용률
Fig. 11. The efficiency of processing elements.

V. 結 論

Dataflow 컴퓨터 構造는 프로그램이 高度의 竝列性을 갖고 수행될 수 있으나, 프로그램이 non-strict 함수를 포함할 때 MIT 기계구조는 불필요한 演算을 수행하게 되어 性能이 저하된다.

본 논문에서는 이러한 취약점을 개선하기 위하여 MIT dataflow 기계구조에 demand propagation network을 포함시키어 선택적으로 demand를 공급함으로써, 결과값이 필요한 actor만을 수행시킬 수 있는 demand-driven dataflow 構造를 提案하였다.

제안된 기계구조는 demand를 전달하는 추가의 시간이 필요하나, 처리장치가 命令語를 수행하는 시간에 비하여 demand의 전달시간이 훨씬 적으며, 실제로 프로그램의 수행시에는 처리장치와 concurrent하게 demand가 전달되므로 거의 무시할 수 있다.

이와같은 기계구조의 기능을 時間加速法으로 시뮬레이션하여 타당성을 보였다. 또한 계산능력의 성능해석을 위하여 수행시간의 비교, 연산처리장치의 실효이용률을 구하여 제안된 구조의 연산속도가 개선됨을 알 수 있었다.

본 논문에서 제안된 構造는 data-driven 기계에 demand를 전달하는 demand propagation network이 포함되어 더욱 복잡한 기계구조를 갖고 있다.

그러나, 이러한 문제점은 dataflow 기계의 실현이 VLSI의 기술과 상호보완적인 발전관계를 유지하고 있으므로, VLSI의 기술발전과 함께 해결될 수 있다고 생각된다.

參 考 文 獻

[1] I. Watson & J. Gurd, *A Prototype Dataflow Computer with Token Labelling.* AFIPS Con. Proc. Ncc, N.Y., pp. 623-628, June 1979.

[2] Arvind & Robert A. Iannucci, *A Critique of Multiprocessing von-Neumann Style,"* Proc. ACM SIGPLAN 1983 Conf., pp. 426-436, Apr. 1983.

[3] Treleaven, P.C. et al, "Data-driven and demand-driven computer architecture," *ACM Computing Surveys,* vol. 14, no.1, pp. 93-143, Mar. 1982.

[4] Karp, R.M. & R.E. Miller, "Properties of a model for parallel computations: determinancy, termination, queueing," *SIAM J. Applied Mathematics,* vol. 14 pp. 1390-1411, Nov. 1966.

[5] Dennis J.B. & D.P. Misunas, *A Preliminary Architecture for a Basic Dataflow Processor.* Proc. 2nd Ann. IEEE Symposium on Computer Architecture, pp. 126, Jan. 1974.

[6] Dennis J.B., D.P. Misunas & C.K. Leung, "A highly parallel processor using a dataflow machine language," CSG Memo 134. Lab. for Computer Science, MIT, Jan. 1977.

[7] Keshav Pingali & Arvind, "Efficient demand-driven evaluation," Technical MEMO 242. Lab. for Computer Science, MIT, July 1984.

[8] Alan, L. Davis & M. Keller, "Dataflow program graph," *IEEE Computer, pp. 26-41, Feb. 1982.*

[9] Dennis, J.B., "Data-flow supercomputers," Computer 13,11, pp. 48-56, Nov. 1980.

[10] Kenneth W. Todd, "Function sharing in a

static Data Flow Machine," Proc. of the 1982 Int. conf. on parallel processing, pp. 137-139, 1982.

[11] Glenford J. Myers, Advances in Computer Architecture, John Wiley & Sons, pp. 463-496, 1978.

[12] Chi-Yuan Chin & Kai Hwang, "Packet Switching Networks for Multiprocessors and Data Flow Computers," IEEE Trans. on Computers, vol. C-33 no. 11, pp. 991-1003, Nov. 1984.

[13] Randal E. Bryant, "Simulation on a Distribution System," CSG Memo 182 Lab. for Computer Science, MIT July 1979.

[14] Am 2900 Family Data Book, Bipolar Microprocessor Logic and Interface, AMD Inc, pp. 5-32~5-71, 1983.

[15] "Intel Component Data Catalog," Intel Corporation, 3065 Bovers Av., Santa Clara, CA 95051, 1978.