최 종 연 구 보 고 서

고속 분산 처리 시스템 개발에 관한 연구

(차세대 컴퓨터 개발 과제중)

주관연구기관 : 한국과학기술원

협동연구기관 : 서 울 대 학 교

과 학 기 술 처

과학기술처장관 귀하

차세대 컴퓨터 개발에 관한 기술 개발중 고속 분산 처리 시스템 개발에 관한 연구의 최종 연구 보고서를 별첨과 같이 제출합니다

1989년 6월 29일

주관연구기관장 : 한국과학기술원장

중과제 책임자 : 조 정 완

협동연구기관장 : 서울대학교총장

협동연구책임자 : 조 유 

제 출 문

과학기술처장관 귀하

본 보고서를 차세대 컴퓨터 개발에 관한 기술 개발중 고속 분산 처리 시스템 개발에 관한 연구 사업의 최종 연구 보고서로 제출합니다

1989년 6월 29일

협동연구기관 : 서울대학교

연구 책임자 : 조 유 근

선임 연구원 : 신 현 식

전 주 식

한 상 영

연 구 원 : 김 명 주

김 진 환

김 홍 근

김 홍 한

방 대 욱

조 규 찬

요 약 문

I. 제목

"고속 분산 처리 시스템의 개발에 대한 연구"

II. 연구 개발의 목적 및 중요성

o 본 연구는 특정 부류의 문제를 고속으로 해결하는 고성능 컴퓨터를 설계하고, 이들을 기존의 범용 컴퓨터와 연결하는 고성능 통신 시스템, 그리고 전체 시스템을 운영하는 분산 운영체제를 설계하는 것을 목적으로 하고 있다.

o 본 연구는 대형 컴퓨터 시스템을 대체할 수 있는 고속 전용 컴퓨터의 설계를 제시하고, 또한 고속 분산 처리를 위한 시스템 소프트웨어에 대한 설계를 제시함으써 이들에 대한 설계 기술을 확보하고 향후 이들 시스템의 개발을 위한 기초가 된다. 또한 컴퓨터 설계를 위한 고급 기술 인력의 양성으로 중대형 컴퓨터의 개발에 기여할 수 있다.

III. 연구 개발의 내용 및 범위

가) 연구 내용

o 고속의 Math. Package 전용 컴퓨터 개발
o 고속의 Divide and Conquer 알고리즘 전용 컴퓨터 개발
o 고속 고신뢰 상호 결합망 개발
o 자원 공유 분산 운영체제 개발

나) 연구 범위

o 새로운 개념을 갖는, 특정 목적의 비폰노이만 컴퓨터 개발
o 컴퓨터간의 고속의 통신 기법 개발
o 이기종 분산 환경에 적합한 운영체제 개발

IV. 연구 추진 내용

가) 고속의 Math. Package 전용 컴퓨터 개발

1차년도에서 설계된 전용언어의 주요 특징 및 Math. Package 전용 컴퓨터의 개념적인 구조를 바탕으로 하여 전용언어의 full set, Math. Package 전용 컴퓨터에 관한 세부 구조 그리고 전단 컴퓨터와의 인터페이스를 설계하였다.

나) 고속의 Divide and Conquer 알고리즘 전용 컴퓨터 개발

1차 년도에서 수행한 HYPERDAC의 개념적인 구조 설계를 기초로, HYPERDAC의 병렬 구조를 효율적으로 이용할 수 있으며, 본 연구에서 채택한 여러 수준의 병렬성을 이용하는 데이타 플로우 모델을 효과적으로 지원해 줄 수 있는 병렬 기계 언어인 베이스 언어를 설계하였고, 이를 토대로 고급 프로그래밍 언어를 설계하였으며, HYPERDAC의 개념 설계를 완성하고 성능 분석을 통하여 실제의 구현시에 발생할 수 있는 문제점들에 관해 분석하였다. 또한 병렬 수행 환경하에서 효과적인 구조화 자료 조작 기법에 대한 연구도 아울러 수행하였다.

다) 고속 고신뢰 상호 결합망 개발

전용 컴퓨터와 범용 컴퓨터들을 연결하는 상호 결합망으로써 이중 구조로 구성되는 고속 근거리 통신망을 연구하였다. 이에 대한 통신용 하드웨어 인터페이스를 설계하고 통신용 소프트웨어의 개발에 대한 연구를 수행하였으며, 아울러 시뮬레이션을 통한 성능 분석도 수행하였다.

라) 자원 공용 분산 운영체제 개발

1차년도에 수행한 자원 공용 분산 운영체제의 요구 조건 분석, 운영체제 구조의 결정, 최소 커널의 개념 설계와 각종 서버의 기능 분석을 바탕으로 최소 커널과 고속 분산 실행 시스템을 상세 설계하여, 자원 공용 분산 운영체제의 설계를 완성하였다. 또한 분산 실행 시스템 및 분산 IPC를 구현하여 설계된 운영체제의 실용성을 검증하였다.

V. 연구 결과의 활용 방안 및 기대 효과

가) 비 폰노이먼형 컴퓨터 및 함수형 언어 구조의 설계 및 분석 사례 제시

나) 데이타 흐름형 언어 및 함수형 언어의 설계 능력 함양

다) 분산 시스템 구성을 위한 고속 고신뢰 결합망 구성 방안 제시

라) 분산 시스템 용 운영체제 설계

마) 고속 분산 처리 시스템의 개발을 통해 얻은 경험과 지식을 기반으로 첨단형 컴퓨터의 개발 능력 제고

Summary

Parallel and distributed processing has long been seen as a means of increasing the speed of computers. Due to the recent rapid developments in computer hardware and high speed LAN, distributed processing systems are becoming more available. High-performance, special purpose computer systems are typically used to meet specific application requirements or to off-load computations that are especially overloading general-purpose computers.

The primary goal of our project is to design a distributed processing system which consists of several conventional workstations and special purpose computers that execute specific applications very fast. These computers are connected through a high speed interconnection network, and are managed by a distributed operating system.

We proposed two special purpose computers. One is a reduction machine composed of multiple processors which are connected by a X-tree interconnection network, and executes mathematical package programs efficiently by the graph reduction scheme. Also a special purpose programming language is designed so that this machine can manage medium/coarse granularity. This language is based on functionality for parallel processing, and provides processor allocator for efficient resource usage. Performance analysis for some examples is presented.

The other is ,we call, HYPERDAC composed of multiple processors which are connected by a Hypertree interconnection network, and executes the divide-and-conquer algorithm efficiently by data flow scheme. This machine adopts the gradient load balancing strategy and the throttling method to promote the use of system resources. Also we designed a base language as parallel machine language and, a high level special purpose programming language. We also presented performance analysis and further research problems.

The HSLN(High Speed Local Network) is adopted for high speed and reliable interconnection network among the general purpose computers and the special purpose computers. Because HSLN is a dual interconnection network, the throughput as well as the reliability is increased. The hardware interface for the communication and the software based on OSI standard protocol are designed and presented in this report. Also, the performance of this system is analyzed by simulation technique.

Resource sharing distributed operating system is designed to integrate and manage these system resources, and to support the efficient execution of distributed programs. It consists of a minimized kernel and system servers. The minimized kernel is based on UNIX, and composed of process management, memory management, and inter-process communication. To support the distributed execution of light weight processes, network-wide demand paging, and channel concepts are introduced in this kernel. Also the Cross Architecture Procedure EXecution(CAPEX) System, which supports the development and the execution of distributed programs on workstations and special purpose computers, is designed and implemented.

CONTENTS

목 차

1. 서 론

최근의 전산기 하드웨어와 고속 통신망의 급속한 발전은 전산기의 이용과 구성 형태에 많은 변화를 가져왔다. 중소형 컴퓨터를 고속의 통신망을 통해 연결한 형태의 분산 처리 시스템(distributed processing system)이 널리 사용되게 되었다. 분산 처리 시스템은 과거의 중앙 집중형 시스템에 비해 성능, 자원의 공유성, 신뢰도, 확장성, 응용성등이 우수하다. 그러나 중앙 집중형 시스템보다 복잡한 시스템 제어를 요구하므로, 보다 효율적인 분산 운영체제가 요구된다.

새로운 컴퓨터 구조(computer architecture)에 대한 연구의 주요 목표는 다양한 종류의 프로그램을 가급적 빠른 시간내에 수행하도록 지원해주는 것이다. 이러한 목표하에 프로그램의 수행 시간을 단축시키려고 프로그램내에 존재하는 병렬성을 최대한 많이 그리고 효율적으로 처리해주는 방향으로 많은 다중 처리 시스템이 지금까지 연구되어 왔다. 그러나 기존의 폰노이만 컴퓨터로 고속 병렬 처리용 프로그램을 수행하기에는 그 근본 개념에 중요한 제한점을 안고 있음은 이미 잘 알려진 사실이다. 그래서 고속 병렬 처리용으로서 비폰노이만 개념의 컴퓨터들이 연구되어 왔다. 그 중에서 대표적인 것이 리덕션 컴퓨터와 데이타플로우 컴퓨터이다. 데이타플로우 컴퓨터는 임의의 명령에 대해서 해당 피연산자 값들이 준비되는 즉시 해당 명령을 수행하므로써 프로그램상에서 가능한 최대의 병렬 처리성을 지원해준다. 리덕션 컴퓨터는 특정 결과값을 얻기 위하여 그 수행이 필요한 명령에 대해 수행 요구 신호를 보내 수행하도록 한 뒤 그 결과값을 반송받는 방식을 채택함으로써, 함수 언어 자체에 내포된 병렬성을 잘 이용하여 방대한 프로그램을 빠른 속도로 병렬 처리할 수 있다.

그림 1.1은 본 보고서에서 제시하는 고속 분산 처리 시스템의 전체적 구성을 나타낸 것이다. 본 고속 분산 처리 시스템은 위에 제시된 여러 요구를 만족시키 위해 제안된 시스템이다. 프로그램의 고속 병렬 처리를 위해 리덕션 모델을 기초로 한 Math. Package 전용 컴퓨터와 데이타플로우 컴퓨터인 Divide and Conquer 알고리즘 전용 컴퓨터 및 그 언어를 개발하고, 이들 전용 컴퓨터를 고속 고신뢰 상호 결

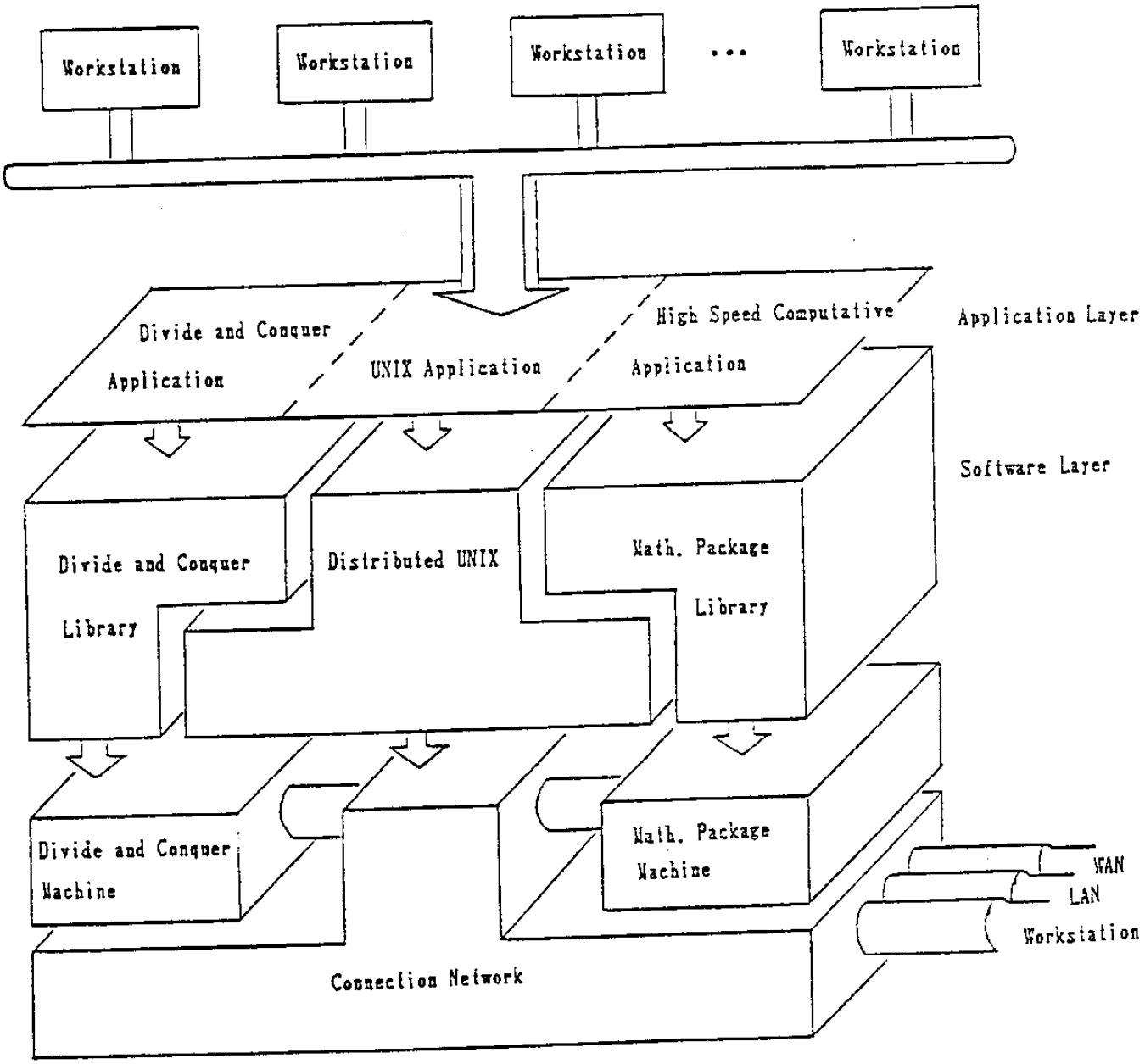


그림 1.1 고속 분산 처리 시스템의 구성

합망을 통해 연결하고, UNIX 호환 분산 운영체제로 전체 시스템을 제어함으로써, 고속 분산 처리 시스템을 구축하였다.

본 보고서의 2장은 하나의 사용자 프로그램이 고속 분산 처리 시스템에서 수행될 때의 시나리오를 기술한다. 이어서 3장과 4장은 각각 본 고속 분산 처리 시스템의 전용 계산 서버로 동작하는 Math Package 전용 컴퓨터, Divide-and-Conquer 알고리즘 전용 컴퓨터의 언어와 하드웨어 설계를 제시한다. 5장에서는 사용자 인터페이스를 담당하는 범용의 워크스테이션과 계산 서버인 전용 컴퓨터를 연결하는 고속 고신뢰 상호 결합망에 대한 하드웨어와 통신 소프트웨어를 설명한다. 6장에서는 본 고속 분산 처리 시스템을 하나의 가상적 단일 컴퓨터 시스템으로 운영하는데 필요한 자원 공용 분산 운영 체제에 대해 기술한다. 마지막 7장에서는 2장부터 6장까지의 부분에 대한 결론과 향후 연구과제를 기술한다.

2. 시스템 개관

이 절에서는 고속 분산 처리 시스템의 구조를 설명하고, 그 작업 과정을 프로그램 수행 예를 통해 설명하겠다.

그림 2.1은 고속 분산 처리 시스템의 하드웨어 구성을 나타내고 있다. 사용자 인터페이스를 제공하는 워크스테이션과 고속의 계산을 지원하는 전용 컴퓨터가 이중 버스의 상호 결합망을 통하여 연결되어 있다.

그림 2.2는 고속 분산 처리 시스템의 한 예제 프로그램이다. 예제 프로그램은 크게 네부분으로 구성된다. (a)에 해당하는 첫번째 부분은 메인 루틴이 포함되어 있는 부분으로서, C 언어로 기술되며 워크스테이션에서 실행된다. (b)에 해당하는 두번째 부분은 분산 실행되는 모듈간의 인터페이스를 기술한다. 이부분은 모듈 인터페이스 기술을 위해 정의된 MIDL(Module Interface Description Language)로 작성되는데, CSG(CAPEX Stub Generator)에 의해 번역되어 분산 실행에 필요한 스터브 모듈로 번역된다. (c) (d)에 해당하는 세번째, 네번째 부분은 각각 Math Package 전용 컴퓨터와 DAC 알고리즘 전용 컴퓨터에서 실행될 부분으로서, 각기 전용 함수 언어와 전용 데이타플로우 언어로 작성된다. 여기서 주목할 것은 전용 언어로 작성하여 모듈을 호출하는 방식이 C 언어로 작성하여 모듈을 호출하는 방식과의 차잇점은 단지 간단한 모듈 인터페이스 기술이 첨가된다는 것이다.

그림 2.2에 보인 예제 프로그램의 번역 과정은 그림 2.4에 나타나있다. 그림 2.3의 번역 과정은 워크스테이션에서 이루어진다. 모듈 인터페이스 기술 화일, intf.mid를 CSG를 통해 번역하여 스터브 모듈 화일, intf.c와 로드 모듈 리스트, main.cnf를 생성한다. main.c와 intf.c는 C 컴파일러, fact.mp는 전용 함수 언어 컴파일러 그리고 mmt.dac는 전용 데이타플로우 언어 컴파일러를 통해 각기 번역되어, 최종 목적 코드 main, fact 그리고 mmt가 생성된다.

사용자는 워크스테이션에서 다음과 같은 명령어를 통해 분산 프로그램을 실행시킨다.

```
$ cxrun main
```

그림 2.1 고속 분산 처리 시스템의 하드웨어 구성

```
#include <stdio.h>
#define  M   100
#define  N   100

main ()
{
        long   A[M][M], B[M][M], C[M][M];
        long   result;

        MAT_read(A, M, M);
        MAT_read(B, M, M);
        C = mmt(A, B, M);
        MAT_print(C);

                .
                .
                .

        result = factorial(20);
        printf("The factorial of 20 is %d\n", result);
}
```

(a) main.c

```
MP     "fact"
                long factorial(int);
HDAC   "mmt"
                long[100][100] mmt(long[100][100], long[100][100], int);
```

(b) intf.mid

```
def factorial(n)
{
        return binary-fact(1,n);
}

def
binary-fact(low, high)
{
        if (low == high) return low;
        else if (low == (high-1)) return low * high;
        else if (low > (high-10)) seq_fact(low, high);
        else
        {
                mid = (low + high) / 2
                v1 = binary_fact(low, mid) $0;
                v2 = binary_fact(mid+1, high) $1;
                return v1 * #v2;
        }
}

seq_def seq_fact(low, high)
int low, high;
{
        int   sum = 1;
        int   index = low;

        while (index <= high)
        {
                sum = sum * index;
                index++;
        }
        return sum;
}
```

(c) fact.mp

```
Function mmt(A, B : array[array[integer]], N : integer;
                                    returns array[array[integer]])
(* This program takes two N x N matrices A and B as inputs and
   returns N x N matrix C which is the product of A by B *)
(* It can be solved according to the divide-and-conquer schema by
   dividing A, B and C into four square matrices each as follows.

        | A11   A12 |     | B11   B12 |     | C11   C12 |
        |           |  x  |           |  =  |           |
        | A21   A22 |     | B21   B22 |     | C21   C22 |

   and computing the four components of C by eight independent
   multiplications, followed by four additions :
   C_ij = A_i1 x B_1j + A_i2 x B_2j for i,j {1, 2}.
   The eight multiplications are performed by eight recursive calls
   that are executed in parallel. *)
(* We assume that N is even, to simplify programming task. *)

{C = array (1,N) (1,N);
      {if N = 2
       then {forall I from 1 to 2 do
                {forall J from 1 to 2 do
                          C[I,J] = A[I,1]*B[1,J] + A[I,2]*B[2,J]}}
       else {C = d_construct([(1,N/2),(1,N/2)] : C%1,1%,
                             [(1,N/2), ((N/2+1),N)] : C%1,2%,
                             [((N/2+1),N),(1,N/2)] : C%2,1%
                             [((N2+1),N),((N/2+1),N)] : C%2,2%);

(* P%x,y% is a component(x row and y column) block matrix of matrix P.
   For instance, P can be represented by block matrices as following :

                        | P%1,1%   P%1,2% |
                    P = |                |
                        | P%2,1%   P%2,2% |

   "d_construct" is an poeration which constructs a matrix,
   consisting of several block matrices. *)
                    A%1,1% from A[(1,N/2),(1,N/2)];
                    A%1,2% from A[(1,N/2),((N/2+1),N)];
                    A%2,1% from A[((N/2+1),N),(1,N/2)];
                    A%2,2% from A[((N/2+1),N),((N/2+1),N)];

                    B%1,1% from B[(1,N/2),(1,N/2)];
                    B%1,2% from B[(1,N/2),((N/2+1),N)];
                    B%2,1% from B[((N/2+1),N),(1,N/2)];
                    B%2,2% from B[((N/2+1),N),((N/2+1),N)];
```

```
(* "from" construct is used to represent a block matrix in a matrix.
   For instance, "A%1,1% from A[(1,N/2),(1,N/2)]" means
   that a block matrix A%1,1% of A consists of rows from 1 to N/2
   and columns from 1 to N/2, of matrix A. *)

        {forall I from 1 to 2 do
                {forall J from 1 to 2 do
                        C%I,J% = mmt(A%I,1%,B%1,J%,N/2) +
                                 mmt(A%I,2%,B%2,J%,N/2)}}}}
in
    C}
```

(d) mmt.dac

그림 2.2 예제 프로그램

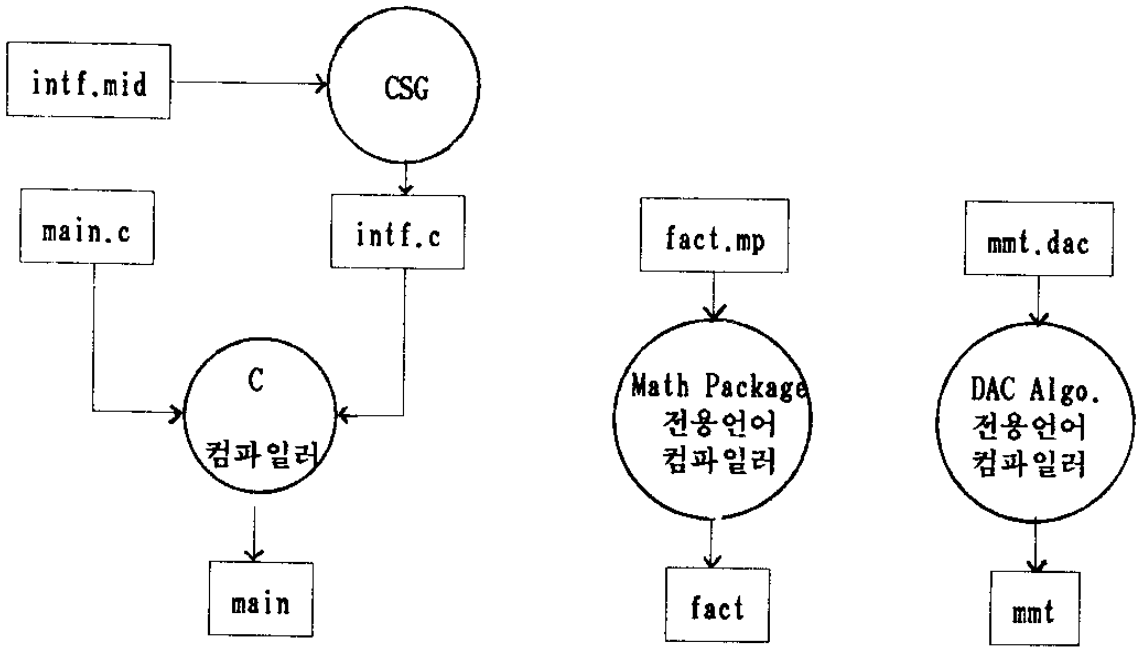


그림 2.3 예제 프로그램의 번역 과정

Cxrun은 분산 실행에 필요한 초기화 작업(즉 분산 실행에 사용할 전용 컴퓨터의 할당, 수행될 사용자 프로세스를 위한 메시지 큐 할당 등)을 수행한 후, main의 실행을 시작시킨다.

워크스테이션에서 실행되고 있는 모듈이 Math Package 전용 컴퓨터 또는 DAC 알고리즘 전용 컴퓨터에서 실행되어야 할 모듈을 호출하는 경우, 고속 분산 처리 시스템의 각 서브 시스템의 작동 과정 및 서브 시스템 사이의 인터페이스는 다음과 같다.

사용자 프로세스(즉 예제 프로그램에서의 main)가 전용 컴퓨터상에서 실행되어야 할 모듈(즉 예제 프로그램에서의 factorial 또는 mmt)의 스터브 모듈을 호출하면 스터브 모듈은 할당된 CAPEX 서버의 메시지 큐 식별자, 실행 화일명, 모듈명, 인자등 필요한 정보를 패킹하여 "분산 실행 요청" 메시지를 생성한다. 이 "분산 실행 요청" 메시지는 같은 워크스테이션상의 CAPEX 클라이언트에 전달된다. CAPEX 클라이언트는, 사용자 프로세스로 부터 전달된 메시지 내용에 실행 코드를 첨가하여 새로운 "분산 실행 요청" 메시지를 생성하고, 최소 커널의 분산 IPC를 거쳐 선정된 CAPEX 서버에 "분산 실행 요청" 메시지를 보내게 되는데, 분산 IPC는 고속 고신뢰 상호 결합망의 필요한 드라이브 루틴을 호출한다.

범용 컴퓨터내에는 두 개의 통신망을 각각 제어하기 위한 두개의 통신망 전용 제어기와 통신 전용 CPU가 구성되어 있다. 각 통신망 전용 제어기는 충돌 회피 CSMA 방식을 이용하여 통신망의 사용 가능성 여부를 탐지하게 된다. 따라서 범용 컴퓨터는 두 개의 통신망 중 하나의 통신망을 사용하게 된다. 그 통신망에 연결된 통신망 전용 제어기는 데이타를 근원지 주소와 목적지 주소가 포함된 프레임 형식으로 구성하여 전송한다. 이때 목적지 주소는 전용 컴퓨터에 직접 연결된 전단 컴퓨터의 주소이다. 전송되는 프레임은 전단 컴퓨터의 기억 장치에 모두 저장된다.

실행 요청" 메시지를 전달받은 CAPEX 서버는 메시지를 언패킹하여 실행 코드, 모듈명, 인자 등 필요한 정보를 추출해낸다. 그리고 CAPEX 서버는 실행 코드, 모듈명 그리고 인자 등을 전용 컴퓨터의 메모리의 일정한 장소에 전달하고, 실행 개시를 지시한다. (또한 후에 그 로드 모듈의 실행 결과가 정확한 위치에 반환되기

위하여 전용 컴퓨터의 실행 결과가 반환될 주소를 할당하고 그 정보를 전용 컴퓨터에 코드와 인자에 부가하여 전달한다).

코드 및 데이타는 전단 컴퓨터로부터 수행 요구의 형태로 인터페이스를 통해 루트 프로세서에 전달된다. 이들은은 다시 전용 컴퓨터내의 상호 결합망을 통하여 전용 컴퓨터내의 해당 범용 프로세서로 전달된다. 각 범용 프로세서는 병렬적으로 주어진 함수를 수행하는데, 다른 함수에 대한 수행 요구나 데이타를 교환할 수도 있다. 범용 프로세서들에 의해 최종 결과값이 생성되면 그것은 루트 프로세서를 통하여 전단 컴퓨터에 전달된다.

전단 컴퓨터에 이동된 로드 모듈은, 고속 분산 처리 시스템에서 실행되는 프로그램이, DAC 알고리즘 전용 컴퓨터(이후 HYPERDAC)에서 그 로드 모듈의 실행을 해당 입력 자료(실행 결과가 저장될 위치를 포함하여)와 함께 요구하면, 전단 컴퓨터는 그 로드모듈에 대한 프로그램과 데이타 패킷을 형성하여 HYPERDAC에 전달한다. HYPERDAC의 각 PE 제어 장치는 컴파일러가 제공한 정보를 근거로 로드 모듈을 HYPERDAC 상에 적절히 로드하고, 해당 입력 자료는 그 입력 자료가 위치할 노드의 구조화 메모리에 할당, 저장하고 동시에 실행결과 값을 전단 컴퓨터에 반환할 때에 필요한 반환 주소에 관한 정보를 PE 제어 장치(대부분이 루트 노드의 PE 제어 장치가 된다)가 유지하며, 구조화 자료에 대한 포인터가 최초의 실행 시작 함수를 구동하기 위하여 그 노드의 Waiting Matching unit으로 전달됨으로써 실행 준비가 완료된다. 그 때부터 HYPERDAC은 실제로 프로그램 수행을 시작하고, 수행이 끝난 뒤에, 수행 결과를 루트 노드를 통해 전단 컴퓨터에 반환함으로써 한 모듈에 대한 작업을 종료한다.

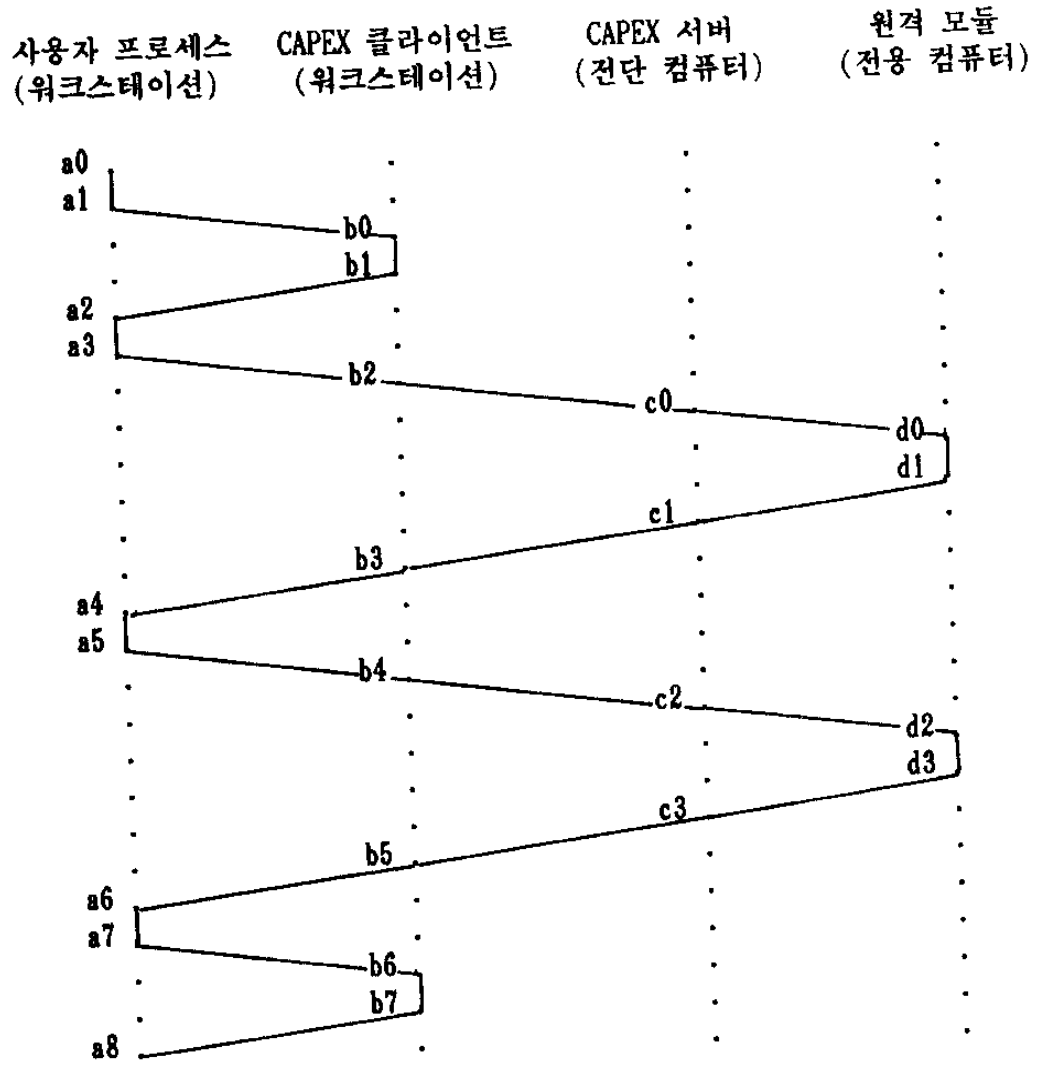
전용 컴퓨터는 실행이 끝나게 되면, 결과값을 전단 컴퓨터의 적당한 주소에 놓고 인터럽트를 발생시킨다. 그러면 CAPEX 서버는 전용 컴퓨터로 부터 전달받은 결과값으로 "분산 실행 회신" 메시지를 생성한다. 그후 CAPEX 서버는 최소 커널의 분산 IPC와 고속 고신뢰 상호 결합망을 통해 결과 메시지를 CAPEX클라이언트에 전달한다.

전단 컴퓨터는 원래 전송된 프레임의 근원지 주소를 목적지 주소로 구성하여 연산 결과를 전용 컴퓨터에 돌려 주게 된다. 이 경우에도 범용 컴퓨터의 송신 과정

과 마찬가지로 전단 컴퓨터는 두 개의 통신망 중 사용 가능한 통신망을 탐지하여 범용 컴퓨터로 전송하게 된다.

CAPEX 클라이언트는 CAPEX 서버로 부터 전달받은 "분산 실행 회신" 메시지 중 적당한 내용을 사용자 프로세스에 전달한다. 사용자 프로세스의 스터브 모듈은 전달된 메시지를 언패킹하여 결과값을 추출하여 귀환한다.

그림 2.4는 그림 2.2의 예제 프로그램의 수행 과정을 도식적으로 나타낸 것이다.



사용자 프로세스
a0 : 실행 시작
a1 : "서버 할당 요청" 메시지 송신
a2 : "서버 할당 회신" 메시지 수신
a3 : ("mmt"에 대한) "분산 실행 요청" 메시지 송신
a4 : ("mmt"에 대한) "분산 실행 회신" 메시지 수신
a5 : ("factorial"에 대한) "분산 실행 요청" 메시지 송신
a6 : ("factorial"에 대한) "분산 실행 회신" 메시지 수신
a7 : "실행 종료 통보" 메시지 송신
a8 : "실행 종료 회신" 메시지 수신

CAPEX 클라이언트
b0 : "서버 할당 요청" 메시지 수신
b1 : "서버 할당 회신" 메시지 송신
b2 : ("mmt"에 대한) "분산 실행 요청" 메시지 수신 (사용자 프로세스로 부터)
("mmt"에 대한) "분산 실행 요청" 메시지 송신 (CAPEX 서버로)
b3 : ("mmt"에 대한) "분산 실행 회신" 메시지 수신 (CAPEX 서버로 부터)
("mmt"에 대한) "분산 실행 회신" 메시지 송신 (사용자 프로세스로)
b4 : ("factorial"에 대한) "분산 실행 요청" 메시지 수신 (사용자 프로세스로 부터)
("factorial"에 대한) "분산 실행 요청" 메시지 송신 (CAPEX 서버로)
b5 : ("factorial"에 대한) "분산 실행 회신" 메시지 수신 (CAPEX 서버로 부터)
("factorial"에 대한) "분산 실행 회신" 메시지 송신 (사용자 프로세스로)
b6 : "실행 종료 통보" 메시지 수신
b7 : "실행 종료 회신" 메시지 송신

CAPEX 서버
c0 : ("mmt"에 대한) "분산 실행 요청" 메시지 수신 (CAPEX 클라이언트로 부터)
전용 컴퓨터에 원격 모듈 "mmt"의 실행 지시
c1 : ("mmt"에 대한) 결과 수신 (전용 컴퓨터의 원격 모듈로 부터)
("mmt"에 대한) "분산 실행 회신" 메시지 송신 (CAPEX 클라이언트로)
c2 : ("factorial"에 대한) "분산 실행 요청" 메시지 수신 (CAPEX 클라이언트로 부터)
전용 컴퓨터에 원격 모듈 "factorial"의 실행 지시
c3 : ("factorial"에 대한) 결과 수신 (전용 컴퓨터의 원격 모듈로 부터)
("factorial"에 대한) "분산 실행 회신" 메시지 송신 (CAPEX 클라이언트로)

원격 모듈
d0 : 모듈 "mmt"에 대한 실행 지시 받음
"mmt" 실행 시작
d1 : "mmt" 실행 끝
모듈 "mmt"의 결과 전달
d2 : 모듈 "factorial"에 대한 실행 지시 받음

```
     "factorial" 실행 시작
d3 : "factorial" 실행 끝
     모듈 "factorial"의 결과 전달
```

그림 2.4. 예제 프로그램의 수행 과정

3. 고속의 Math. Package 전용 컴퓨터

Math. package를 빠른 속도로 수행하기 위한 방법의 하나로서 math. package 자체가 내포하는 병렬처리성을 최대한 이용하는 것을 생각할 수 있다. 이를 위해서는 math. package 상에서 병렬처리될 부분을 추출하는 단계와 추출된 병렬처리부분을 효율적으로 수행하는 단계를 생각해야 한다. 전통적인 폰노이만(von Neumann) 수행방식을 따르는 컴퓨터로는 이러한 병렬처리성을 효율적으로 지원해주기 어렵다는 것은 이미 잘 알려진 사실이다[BACK78]. 1970년대 후반부터 등장한 함수형 컴퓨터(functional computer)는 폰노이만 컴퓨터의 이러한 한계점을 상당 부분 잘 극복하고 있는데 데이타 플로우 컴퓨터(data flow computer)와 리덕션 컴퓨터(reduction computer)가 그 대표적인 함수형 컴퓨터들이다. 데이타 플로우 컴퓨터는 풍부한 병렬처리성을 보장해 주지만 시스템 사용 측면에서 다소 불리하다. 반면에, 리덕션 컴퓨터는 필요한 부분만을 병렬로 수행해 줌으로써 데이타 플로우 컴퓨터의 약점을 보완해주지만 요구신호전송이라는 부담을 추가로 가지게 된다. 본 보고서에서 제시하는 "고속의 Math. Package 전용 컴퓨터"는 일종의 함수형 컴퓨터(특히 리덕션 컴퓨터)이다. 따라서 프로그램 작성자는 하나의 math. package를 함수언어로 작성하게 되며 작성된 함수 프로그램은 Math. Package 전용 컴퓨터에서 리덕션 방식으로 병렬 수행된다.

3.1. 전용 언어 설계

Math. Package 전용 컴퓨터에서 math. package 프로그램을 수행하기 위해서는 먼저 전용언어에 대한 설계가 필요하다. 앞서 언급했듯이 전용언어는 기본적으로 함수언어이면서도 여러 가지 이유로 인해서 명령형 언어의 특징과 프로그램-프로세서 매핑 특징 등을 갖는다.

3.1.1. 설계 방향

문제에 주어진 병렬처리성을 프로그램 상에서 자연스럽게 나타내기 위해서는 함수언어만큼 효율적인 언어가 없다. 함수언어는 수학적인 함수 개념을 언어의 배경으로 한다. 즉, 임의의 시각에 받아 들인 입력값들에 대해 만일 그 값들이 동일하다면, 프로그램(함수)의 출력값도 유일하게 정의된 동일한 값을 갖는다는 개념이다. 따라서 임의의 프로그램을 병렬로 수행할 때 병렬처리할 부분들이 그 수행순서와 상관없이 원하는 최종 출력값을 생성할 수 있게 된다. 반면에 기존의 병렬 명령형 언어로 프로그램을 작성하여 병렬로 수행할 경우 병렬처리할 부분들의 접속부에 barrier와 같이 적당한 동기화 명령(synchronization command)을 명시해야 한다[BUEH87]. 지금까지 구미 선진국에서 연구되어온 함수언어로는 FP[BACK78], Miranda[TURN85], ParAlfl[HUDA85], HOPE^{+}[PERR87], Standard ML[WILK87] 등을 손꼽을 수 있다. 여기에서 설계된 전용언어 역시 math. package의 병렬처리성을 프로그램 상에 자연스럽게 나타내고 포착하기 쉽도록 하기위하여 기본적으로 함수언어이다. 함수언어에도 약점은 있다. 이 약점 중에서 가장 큰 것은 granularity가 미세(fine)하다는 것인데 이는 함수 프로그램을 수행하는 컴퓨터 구조와의 관계에서 드러난다. 다시 말해서 grain이 작으면 병렬처리성은 증가하지만 다중 프로세서 시스템 상에서 소요되는 통신부하 역시 증가하게 된다. INTEL 80286 프로세서 128개가 Hypercube 형태로 연결되어 있는 iPSC의 경우, CPU 명령 한 단위의 처리시간의 10내지 100배 가량에 해당하는 시간이 걸려야 프로세서 간의 통신 한단위가 처리됨을 볼 때 이러한 통신부하는 심각하다고 아니할 수 없다[HUDA86]. 따라서 설계될 전용언어는 통신부하의 손해를 극복할 만큼 granularity가 크면서도 함수성(functionality)을 만족

해야 한다. 더구나 granularity가 큰 경우, 프로그램 작성능력(progrmmabilty)이 기존의 함수언어들보다 크게 증가할 수 있는 추가적인 장점을 얻을 수 있게 된다.

3.1.2. 전용언어의 특징

앞서 언급되었듯이 설계된 전용언어는 기본적으로 함수언어로서 프로그램 수행순서와 상관없이 그 수행된 최종 결과값은 언제나 동일하게 생성된다는 Church Rosser Property[JONE87]를 만족한다. 이러한 기본적인 특성이외에 여기에서 사용하는 언어는 프로그램 작성자로 하여금 프로세서 지정 및 할당에 관한 option을 제공한다. 구체적으로 이것은 프로그램 수행과정 중에서 일어나는 그래프 생성(graph production) 및 리덕션 수행 위치를 시스템 이외에도 사용자의 지시에 의해서 지정될 수 있도록 해주는 것이다. 따라서 프로그램 작성자가 프로그램의 수행성격을 어느 정도 이해하고 있는 경우에 이를 프로그램 작성시에 반영하면 프로그램을 보다 효율적으로 수행할 수 있을 뿐 아니라 컴파일러가 하는 일을 어느 정도 경감킬시 수 있다는 장점을 얻는다. 물론 option이기 때문에 이를 명시하지않으면 보통의 함수언어로 작성된 프로그램이 되므로 프로그램의 저장 및 수행에 관한 모든 과정이 시스템에 의하여 자동적으로 일어난다. 여기에서 사용하는 언어는 다음과 같은 특징을 갖춘 함수언어이다.

(1) 함수언어로서의 특징

첫째, 설계된 전용언어는 Church Rosser 성질을 만족한다. Church Rosser 성질이란, 임의의 수식 E1, E2에 대하여 E1 <==> E2가 성립하면, 어떤 수식 E에 대하여 E1 ==> E, E2 ==> E가 성립함을 말한다. 단, <== 는 abstraction을 나타내고 ==> 는 reduction을 나타낸다. 이 성질은 프로그램 수행의 결과값이 수식의 수행 순서및 수행 위치에 관계없이 동일하다는 것을 의미한다. 이것을 referential transparency라고 하는데 이러한 referential transparency 성질은 C나 Pascal과 같은 명령형 언어로 쓰여진 프로그램과 함수언어로 쓰여진 프로그램을 구분하는 기준이 될 수 있다. 명령형 프로그래밍 언어에서는 전역변수를 사용함으로써 수식의 값이 동적으로 변할 수 있는 side-effect가 존재한다. 이로 인하여 같은 인자에 대하여 함

수의 결과값이 다를 수가 있다. 이는 함수의 결과값이 전역변수 값에 영향을 받기 때문이다. 반면에 함수언어는 이러한 side-effect가 없으므로 병렬처리에 유리하다는 장점을 갖는다.

둘째, 하나의 프로그램은 블럭(block) 형태의 함수정의(function definition)와 이들간의 함수호출(function call)로 이루어진다. 모든 함수는 다음의 형태를 취한다.

```
function_name(argument)
argument_declaration
{   variable_declaration
    variable1 = expression1;
    variable2 = expression2;
    .......
    return(expressionr);
    .......
    variablen = expressionn;
 }
```

한 함수 블럭내에서는 변수에 수식을 치환하는 문(앞에서는 n개)들과 return문의 순서는 중요하지 않다. 이를 위해서는 프로그램 작성시에, 한 함수블럭내의 모든 변수는 치환문의 왼쪽에 오직 한 번만 나올 수 있다는 단일치환규칙(single assignment rule)을 따라야 한다. 이러한 단일치환규칙을 어기거나 문장들의 순서에 의미를 두려면 연산자 seq를 사용한다. seq안의 모든 문장들은 기존의 폰노이만코드와 유사하게 번역되어 하나의 블럭을 이룬 후 순차로 수행되는데 이에 대한 자세한 설명은 3.1.5에서 하겠다.

셋째, 앞에서도 언뜻 보였듯이 함수호출에 있어서 매개변수(argument)는 'curry' 형태가 아닌 'tuple' 형태를 취한다. 이것은 기존의 명령형 언어와 비슷한 함수호출 방식을 사용함을 의미한다. 기존의 함수언어 중에서 FP[BACK78]나 Miranda[TURN 85]는 매개변수를 'curry' 형태로 사용한다. 이들는 'tuple' 형태의 다른 함수언어에 비하여 독특한 수행과정을 제공하지만 명령형 언어와 구문(syntax) 면에서도 상이하게 되어 기존의 사용자가 첫눈에 거부감을 느끼게 된다. 반면에 여기에서 설계된 전용언어는 'tuple' 형태를 취하므로 명령형 언어와 유사한 구문을 제공하여 사용자

에게 친밀감을 줄 뿐 만 아니라 기존의 명령형 언어와 조화를 이룰 수 있는 터전을 마련해 줌으로써 전용언어의 확장을 꾀할수 있게 해준다.

네째, 순환구문과 리커젼을 둘 다 허용한다. 순환구문은 일종의 tail-recursion이지만, 함수호출로 인한 overhead가 계산의 병렬처리로 얻은 이득을 상쇄시킬 수 있다는 점에서 프로그램 작성자에게 선택의 여지를 주게 된다. 예를 들어, 스트림(stream)의 모든 원소들의 합을 구하는 함수는 다음과 같이 두 가지 구문으로 표현하는 것이 가능하다.

```
[방법 1] seq_def loop_sum(str_var)   /* 순환구문을 사용한 예 */
         stream int str_var;
         {  int sum;
            sum = 0;
            while( !stream_empty(str_var) ){
                   sum = sum + steam_first(str_var);
                   str_var = stream_rest(str_var); }
            return(sum);
          }

[방법 2] def recur_sum(str_var)      /* 리커젼을 사용한 예 */
         stream int str_var
         {      if( stream_empty(str_var) ) return(0);
                else return( stream_first(str_var) +
                             recur_sum(stream_rest(str_var)) );
         }
```

참고로 순환구문은 단일치환규칙을 어기게 되므로 seq_def로 정의된 함수나 seq 연산자 내부에서 사용됨을 밝혀 둔다.

다섯째, 기존 리덕션 컴퓨터의 기본 수행방식인 lazy evaluation 방식의 테두리를 벗어나도록 명시할 수 있다. 비정형 명령(nonstrict instruction)이나 무한 데이타의 처리에 있어서 lazy evaluation에 의한 수행방식은 매우 효과적이다. 일반적으로 비정형명령의 경우, 불필요한 계산부분에는 수행요구신호를 전달하지 않음으로써 시간적인 면이나 자원의 사용면에서 유리하다. 무한 데이타의 경우, 함수성을 만족하면서도 부분적으로는 요구신호전송을 통하여 병렬처리를 가능하게 해준다. 그러나 프

로그램의 모든 부분이 이러한 요구신호전송에 의해 수행된다면 정형 명령(strict instruction)의 경우에 있어서 병렬처리될 수 있는 부분들이 순차적으로 수행되므로 전체적으로 프로그램 수행시간이 길어진다. 따라서 시스템의 요구신호전송에 의하여 결정되는 수행지시보다 프로그램 작성자가 꼭 수행될 것으로 추측되는 명령에 미리 수행 지시를 함으로써 프로그램 수행 속도를 향상시킬 수 있다. 이러한 수행방식은 데이타 플로우 컴퓨터가 기본적으로 채택한 eager evaluation 방식과 개념면에서 유사하다. 자세한 내용은 3.1.3에서 거론된다.

여섯째, 설계된 전용언어는 array, list, stream 세종류의 구조화된 자료(structured data)를 제공한다. Array의 경우, 비결정적인 처리(nondeterministic manipulation)에 의해 각 원소의 비동기적 접근을 허용함으로써 병렬처리성을 높여준다. 다만, 전용언어가 함수성을 만족해야 한다는 조건때문에 한 array의 변경은 새로운 array의 생성을 필요로 하며 이는 프로그램 작성자로 하여금 새로운 array 이름을 선언하도록 요구한다. array에 대한 고유 명령어가 제공되는데 이들은 3.1.6에서 다룬다. List의 경우, LISP언어의 list와 동일한 개념을 갖는데 일종의 tupled-array라고 볼 수 있다. List에 대한 고유 명령어는 cons, appcnd, list, car, cond가 제공된다. Stream은 일종의 nonstrict list인데 그 성격과 사용되는 고유 명령어는 다소 제한적이며 특이하다. Stream은 임의의 원소에 대한 접근을 허용하지 않고 오직 stream_first로만 첫 원소를 접근할 수 있게 한다. 이외에 사용되는 stream 명령어는 stream_empty, stream_rest, stream_conc, stream_cons가 있다.

일곱째, 전용언어는 강한 수형(strongly-typed) 언어이다. 이는 프로그램 수행 중에 발생할 수 있는 수형 불일치에 의한 오류(type miss-match error)를 컴파일 시간에 찾아낼 수 있음을 의미한다.

(2) 부가적인 특징

순수한 함수언어로서의 전용언어는 전술한 특징들을 만족하므로, 이로부터 병렬처리될 부분을 효율적으로 추출할 수 있는 컴파일러를 구현하기가 용이하다. 그러나 이러한 장점만으로 리덕션 컴퓨터에서 프로그램을 효과적으로 수행시키기 힘들다. 따

라서 본 연구에서 제안된 전용언어는 함수언어의 기능외에 다음과 같은 특징을 제공한다.

첫째, 각 함수들이 수행될 프로세서를 지정할 수 있다. 기존 리덕션 컴퓨터들은 함수 호출에 관련된 사항을 컴파일러나 시스템이 추출하여 지원해 주는 형태를 취한다. 따라서 함수 호출에 있어서도 호출된 함수가 어느 프로세서에서 수행되어야 하는+ 에 대한 결정 역시 시스템에 의해 이루어진다. 그러나 이러한 방법은 컴파일러 능력의 한계점으로 인하여 효과적인 수행방식을 제공받기 어렵다. 따라서 프로그램 작성자의 전문 지식을 이용하는 것이 보다 효율적일 수 있다. 이는 함수 계산시 수반되는 통신에 의한 부하 또는 프로세서의 메모리 할당에 따른 부하 등의 측면에서 프로그램 작성자의 경험적인 전문지식이 더 효율적인 수행방식을 제공할 수 있기 때문이다. 그러나 프로그램 작성자가 이러한 특징을 이용하지 않을 경우 기존의 리덕션 컴퓨터에서와 같이 시스템에 의한 자동적인 처리가 이루어질 수 있다. 자세한 내용은 3.1.4에서 설명한다.

둘째, 하나의 math. package 프로그램을 작성할 때 기존의 명령형 프로그래밍 기법을 허용한다. 전용언어는 함수를 def와 seq_def 두 가지 명령으로 정의한다. Def로 정의되는 함수는 순수한 함수형 프로그래밍 기법으로 작성된다. 즉, def로 함수를 정의할 때는 단일치환규칙을 지켜야 한다. 반면에 seq_def로 함수를 정의할 때는 기존의 명령형 프로그래밍 기법 특히, C 프로그래밍 기법을 이용하게 된다. 따라서 기존의 C 언어가 제공하는 특징들을 거의 허용하게 되는데 이는 기존의 C 프로그램들의 유용한 부분들을 상당수 살릴 수 있다는 의미가 된다. 그런데 여기에서 유의할 사항은, seq_def로 작성된 함수의 외부적 행동(external behavior) 역시 함수성을 만족하도록 해야 한다는 사실이다. 따라서 C 언어의 external, static 변수들과 같이 side-effect를 일으키는 feature들은 사용할 수 없다. Def로 정의된 함수내에서 seq라는 연산자가 사용될 수 있는데 이것 역시 부분적으로 명령형 프로그래밍을 제공해준다. Seq 연산자도 외부적 행동이 함수성을 만족해야 한다. Seq 연산자 내부에서 정의되어 값을 반환하는 임의의 변수에 대한 접근은 seq 연산자 내부 전체의 수행을 유도한다. 다시 말해서 seq 연산자는 부분 수행(partial execution)이 아닌

전체 수행(total execution)으로 구현됨을 의미한다. 이는 seq_def로 정의된 함수와 마찬가지로 요구신호전송에 의한 부하를 감수하지 않고 기존의 폰노이만 방식으로 명령 코드들을 수행하고자 함이다.

3.1.3. lazy/predemand evaluation

앞서 3.1.2에서 거론했듯이 본 Math. Package 전용 컴퓨터는 리덕션 수행방식을 수행 모델로 취하고 있으므로 기본적으로 lazy evaluation을 따르면서도 프로그램 작성자의 지시에 의하여 predemand evaluation도 따른다. Predemand evaluation이란 수행요구신호가 시스템에 의해 인위적으로 생성되도록 프로그램 작성자가 프로그램상에서 명시해 주는 것을 말한다. 이러한 predemand evaluation의 명시는 해당 변수나 함수 이름 앞에 #을 접속시키면 된다. 이 때 유의해야 할 점은 #이 명시되는 변수는 1-grain 변수이어야 한다는 점이다. 1-grain 변수란 OGF(One-Grain Function)의 출력값만을 받는 변수를 말하며 OGF란 프로그램 작성자가 작성한 함수 중 가장 안쪽의 함수(user-defined innermost function)를 말한다. 이러한 제약조건은, 본 Math. Package 전용 컴퓨터가 하나의 OGF에 대해서 폰노이만 방식으로 순차처리를 하게 되므로 OGF 내부에서 자신의 다른 내부로의 # 지시가 무의미하기 때문에 제시되었다. Predemand evaluation의 명시는 크게 세 종류로 나눌 수 있다.

첫번째 종류는, 함수를 정의할때 매개변수 이름앞에 명시하는 경우이다. lazy evaluation의 특성중의 하나가 함수의 매개변수는 필요시에 그 값이 요구되어 진다는 것이다. 그러나 반드시 사용될 것이라고 예견되는 매개변수는 미리 요구되어지는 것이 좋다. 예를 들어 다음과 같이 함수 f를 정의할 경우를 보자.

```
def f (x, #y,z)
int x,y,z;
{
    ...
}
```

나중에 함수 f가 호출되어 질 때 매개변수 y의 값을 계산하도록 하는 요구신호 역시 동시에 발생된다. 이와같이 함수 정의시에 매개변수가 predemand 되면 전체적으로 병렬처리성이 높아지게 될 뿐만 아니라 주어진 프로그램을 보다 빠르게 수행할 수 있다.

두번째 종류는, 명령의 피연산자 앞에 붙는 경우다. 다음과 같이 조건문의 비정형 피연산자 앞에 #이 붙을 경우를 보자.

```
x = function1();
.......
z = if (조건수식) #x else y;
```

z를 구하고자 할 때 조건수식의 계산과 동시에 x의 값에 대한 계산도 병렬적으로 수행된다. 또한 다음과 같이 infix 연산자의 피연산자에 #이 붙은 경우를 보자.

```
y = function1();
......
out = x + #y;
```

이 경우도 덧셈 연산의 수행 시작과 동시에 y 값에 대한 요구신호가 발생되므로 병렬처리성을 높일 수 있다. 그러나 여기에서 주의할 것은 수식 #x+y에서 사용된 predemand는 아무런 효과도 없다는 것이다. 이는 '+' 연산자가 정형(strict)이기 때문이며, #x+y는 x+y와 동일한 의미를 갖기 때문이다. 이와 비슷한 경우로서 다음과 같은 수식을 생각할 수 있다.

```
if (#x) y else z; ==  if (x) y else z;
#x && y; == x && y;
```

세번째 종류는, 리스트 구성원소에 명시하는 경우다. 리스트 원소에 #이 붙으면 리스트의 생성과 함께 동시에 그 원소의 계산을 수행한다.

Predemand evaluation의 변형으로서 ##이 제공된다. 이것은 지시된 변수나 함수호출의 결과값을 생성하기 위하여 필요한 다른 함수들의 모든 매개 변수에 대해서 연쇄적으로 predemand evaluation을 명시하는 것으로서 eager evaluation에 가깝다고 볼 수 있다.

3.1.4. 프로세서 할당자

본 연구에서 제안한 전용언어에서는 함수 호출시에 프로세서 할당자(processor allocator)를 사용하여 그 함수가 수행될 범용 프로세서를 명시할 수 있다. 리덕션 방식으로 프로그램을 수행하다보면 함수호출에 의하여 새로운 그래프 생성(graph production)이 자주 일어나기 마련이다. 예를 들어 함수 f1()이 함수 f2()를 호출하며 함수 f2()가 함수 f3(), f4(), f5()를 수행한다고 가정하자. 만일 시스템에 의해 f2()의 그래프 생성 및 그에 따른 추가적인 그래프 생성이, 현재 f1()을 수행중인 범용 프로세서 근처에서 일어난다면 수행 메모리의 할당면이나 부하균형(load balancing)면에서 별로 바람직하지 못하다. 따라서 프로그램 작성자는 f2()의 호출이 가급적 멀리 떨어진 범용 프로세서에서 일어나도록 지정하는 것이 훨씬 효율적일 가능성이 높다. 이 경우 함수 f1()의 정의는 다음과 같이 이루어질 수 있다.

```
f1()
{     ......
      x = f2() $+4;
      ......
      return(r);
 }
```

여기에서 $+4는 현재 f1()을 수행중인 프로세서로부터 오른쪽으로 네번째에 위치한 범용 프로세서에서 함수 f2()를 수행하라는 의미이다. 이것을 프로세서 할당자라 부른다. 프로세서 할당자는 $로 시작하며 다음과 같은 형태를 갖는다.

```
<프로세서_할당자> ::= <상대_할당자> | <절대_할당자>
<상대_할당자>     ::= $ <방향> <할당수식>
<절대_할당자>     ::= $ <할당수식>
<방향>            ::= + | -
<할당수식>        ::= <무부호_정수>
                    | <변수>
                    | $self
                    | <할당수식> + <할당수식>
                    | <할당수식> - <할당수식>
```

상대 할당자는 앞의 예에서 보았듯이 현재의 호출 위치에서 상대적으로 옆에 위치한 범용 프로세서로 함수 수행 요구신호를 보내는 것을 말한다. 이 때 방향이 +이면 오른쪽, -이면 왼쪽을 나타내는데 $+0과 $-0는 현재의 범용 프로세서 자신을 가르킨다. 절대 할당자는 계산된 무부호 정수(unsigned integer)를 자신의 ID-number로 갖는 범용 프로세서로 하여금 지정된 함수를 수행하도록 해준다. 일반적으로 프로세서 할당자의 값은 modulo assignment rule을 적용한다. 이것은 임의의 할당수식의 결과값을 범용 프로세서의 갯수(|GP|)로 나눈 후 그 나머지를 할당수식의 최종값으로 갖는다는 의미이다. 당연한 결과이지만 범용 프로세서들은 0부터 (|GP| - 1)의 범위에 걸쳐 ID-number를 갖는다. 참고로 할당수식 가운데 $self는 현재의 범용 프로세서의 ID-number를 나타낸다. 만일 프로세서 할당자가 명시되지 않으면 전용 컴퓨터에서 채택하는 부하 균형 기법에 따라 동적으로 함수 호출의 위치가 결정된다.

3.1.5. 코드블럭의 생성

한 함수내의 지역참조성(the locality of reference)을 생각해보자. 하나의 함수는 자신의 매개변수(argument)들을 반복해서 사용할 가능성이 크다. 따라서 이들 매개변수들(좀더 확장해서 함수내의 지역변수(local variable)들까지 고려하자)을 한 곳에 모아 놓고 이들을 참조하면서 주어진 명령들을 수행할 경우 함수 수행에 필요한 메모리의 참조 범위는 상당히 작은 영역을 차지한다. 따라서 하나의 함수는 리덕션 수행의 제일 작은 단위로 간주될 수 있기때문에 본 시스템에서는 하나의 함수를 리

덕선의 최소 단위로 보고 이를 하나의 코드블럭(code block)으로 묶어서 번역한다. 예를 들어 앞 3.1.2의 recur_sum()에 대해서 생각해보자. 우선 이 함수는 다음 그림 3.1과 같은 그래프로 나타내질 수 있다.

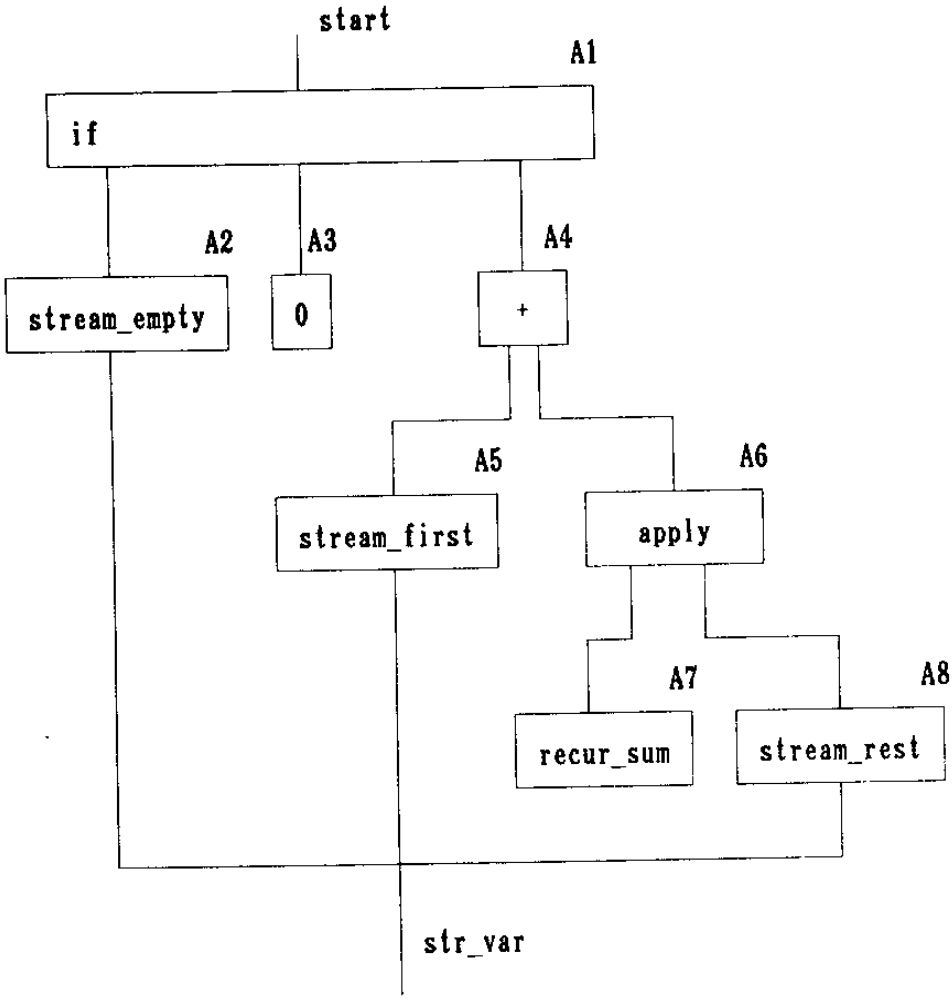


그림 3.1 함수 recur_sum()에 대한 그래프

함수 recur_sum()은 다음과 같은 코드블럭으로 번역된다.

함수 recur_sum:

주소	명령	수신주소	송신주소
start	RETURN	A1	$caller
str_var	ARGUMENT	$caller+2	A2 A5 A8
A1	IF	A2 A3 A4	start
A2	STREAM_EMPTY	str_var	A1
A3	INT	0	A1
A4	+	A5 A6	A1
A5	STREAM_FIRST	str_var	A4
A6	APPLY	A7 A8	A4
A7	FUNCTION	recur_sum	A6
A8	STREAM_REST	str_var	A6

그림 3.2 함수 recur_sum()에 대한 코드블럭

앞의 코드블럭에서 수신주소는 상대주소(relativc address)로 표현될 수 있다. 상대주소 방법으로 함수 recur_sum()에 대한 코드블럭을 다시 표현하면 다음과 같다.

함수 recur_sum:

주소	명령	수신주소	송신주소
start	RETURN	+2	$caller
str_var	ARGUMENT	$caller+2	+2 +5 +8
A1	IF	+1 +2 +3	-2
A2	STREAM_EMPTY	-2	-1
A3	INT	0	-2
A4	+	+1 +2	-3
A5	STREAM_FIRST	-5	-1
A6	APPLY	+1 +2	-2
A7	FUNCTION	recur_sum	-1
A8	STREAM_REST	-8	-2

그림 3.3 상대주소방법에 의한 함수 recur_sum()의 코드블럭

이러한 상대주소방법은 함수내의 지역참조성을 이용하는 방법이면서도, 추후 부하균형을 위하여 범용 프로세서간에 수행 함수를 이동시킬 때 주소교정으로 인한 노력을 크게 줄여준다는 장점이 있다. 한편 코드블럭 내용중에서 $로 시작하는 주소는 전역주소로서 이 함수 recur_sum()가 호출되어 범용 프로세서의 정의 메모리로부터 수행 메모리로 코드블럭이 옮겨질 때 생성된다. $caller는 이 함수를 호출한 위치 즉, APPLY 명령이 사용된 주소를 나타낸다. 따라서 이 주소는 범용 프로세서에 대한 고유주소(P)와 수행 메모리내의 지역주소(L)를 접속한 형태인 PL로 나타내진다. $caller+i는 자신을 호출한 원래의 함수가 주는 i번째 매개변수값에 대한 주소이다. 따라서, i는 1 < i < (tuple의 한계 규정치)를 만족하는 정수이다. 매개변수값을 가져오는 주소규칙 $caller+i를 만족하기 위하여 함수호출에 대한 코드생성은 반드시 다음의 순서를 따라야 한다.

[함수호출의 예] x = function_name(arg1, arg2, arg3);

[코드생성 순서]

주소	명령	수신주소	송신주소
%	APPLY	%+1 %+2 %+3 %+4	.....
%+1	FUNCTION	function_name	%
%+2	arg1	.....	%
%+3	arg2	.....	%
%+4	arg3	.....	%

(여기에서 %는 수행 메모리내의 특정한 지역주소이다)

그림 3.4 함수호출에 대한 코드생성 과정

코드블럭에서 매개변수를 나타내는 "ARGUMENT"는 해당 코드블럭의 수행이 진행됨에 따라 일단 호출이 이루어지면 "CONSTANT"로 바뀌어 실제값을 가지게 된다. 이는 그래프 리덕션의 장점이라고 볼 수 있는 포인터 공유(pointer sharing)에 의한

중복 계산의 회피를 지원해주는 특징이라 할 수 있다. 다음은 CONSTANT가 실인자의 수형(type)에 따라 나타날 수 있는 경우를 요약한 것이다.

[x가 호출되기 전]

주소	명령	수신주소	송신주소
x	ARGUMENT	$caller+2	+3 +4 +5 +9

[x가 호출된 후]

주소	명령	수신주소	송신주소
x	단일수형	value	+3 +4 +5 +9
x	복합수형	pointer	+3 +4 +5 +9

그림 3.5 호출에 의한 ARGUMENT의 변형

여기에서 단일수형이라 함은 CHAR, SHORT, INT, LONG, UNSIGNED, FLOAT, DOUBLE을 말하며, 복합수형이라 함은 STRUCT, UNION, ARRAY, LIST, STREAM을 말한다.

본 시스템은 기본적으로 lazy evaluation 방식으로 리덕션을 수행하므로 함수호출은 단지 그 함수에 대한 코드블럭의 수행을 요구하는 정도로 그친다. 그리고 매개변수에 대한 수행은 코드블럭 수행중에 필요에 따라 일어나도록 되어 있다. 그러나 앞에서 제시했던 predemand evaluation(#)이 매개변수에 명시된 경우는 함수호출과 더불어 predemand evaluation이 명시된 매개변수로의 리덕션 수행 요구가 동시에 발생하게 된다. 따라서 이러한 predemand evaluation에 대한 요구도 코드블럭내에서 명시된다. 예를 들어 x = function_name(#arg1, arg2, #arg3)를 생각해

보자. 이 함수호출에 대한 코드생성은 APPLY 명령을 제외하고는 동일하다. 이 경우 APPLY 명령은 "APPLY %+1 #%+2 %+3 #%+4"이 되며 기본 프로세서는 APPLY 명령의 이러한 용도를 파악하여 %+1, %+2, %+4에 대한 수행요구 신호를 동시에 발생시키고 자신은 대기상태로 접어든다.

이상에서 언급된 코드블럭 생성문제는 def로 정의된 함수에 관한 것이다. Seq_def에 관한 부분은 기존의 폰노이만 코드블럭과 동일하게 번역되어 진다. 단지 함수의 입출력 부분에 관해 요구신호 전송 및 결과값 반환을 위한 영역을 설정해 주면 된다. 따라서 seq_def로 정의된 코드블럭에서 송신 주소 및 수신 주소는 사용되지 않으므로 별의미가 없다. Def로 정의된 함수에서 부분적으로 사용되는 seq 연산자에 관해서도 seq_def로 정의된 함수와 동일하게 코드블럭 생성이 이루어진다.

3.1.6. 전용언어의 구문과 의미

(1) 프로그램의 구조

전용언어로 작성된 프로그램은 workstation의 다중 언어 프로그래밍 환경하에서 사용된다. 따라서, 전용 언어로 작성된 프로그램(일종의 math. package)의 결과를 생성하는 함수는 workstation에서 사용되는 언어로 작성된 프로그램에 의하여 부함수처럼 호출되어 사용된다. 앞의 3.1.2에서 언급했듯이 전용 언어로 작성된 프로그램은 하나 또는 그 이상의 함수 정의로 이루어 진다. 전용 언어에서 함수 정의에는 순수한 함수(즉, 병렬 함수)에 대한 정의와 명령형 형태의 함수(즉, 순차 함수)에 대한 정의가 있다. 특히 이 순차 함수의 구문은 기존 C 언어의 함수를 약간 수정한 형태를 갖는다. 순수한 함수 정의와 순차 함수 정의는 함수 이름 앞에 각각 def와 seq_def를 붙여 구분한다고 이미 앞에서 밝혔다. Def로 정의된 함수 내에서 부분적으로 순차처리가 이루어지기를 원할 경우, seq 연산자를 사용한다. 프로그램의 전체적인 구조에 대한 구문은 다음과 같다.

```
<프로그램>          ::= <함수정의_리스트>

<함수정의_리스트>   ::= <함수정의>
                      | <함수정의> <함수정의_리스트>
<함수정의>          ::= def <함수선언> <병렬함수본체>
                      | seq_def <함수선언> <순차함수본체>

<함수선언>          ::= <수형> <함수이름> <매개변수_리스트> <선언_리스트>

<병렬함수본체>      ::= { <선언_리스트> <병렬문장_리스트> }
<병렬문장_리스트>   ::= <병렬문장>
                      | <병렬문장> <프로세서_할당자>
                      | <병렬문장> <병렬문장_리스트>
                      | seq { <순차문장_리스트> }
                      | seq { <순차문장_리스트> } <병렬문장_리스트>

<순차함수본체>      ::= { <변수선언> <순차문장_리스트> }
<순차문장_리스트>   ::= <순차문장>
                      | <순차문장> <순차문장_리스트>

<매개변수_리스트>   ::= <명칭>
```

```
| #<명칭>
| <명칭> , <매개변수_리스트>
| #<명칭> , <매개변수_리스트>
```

(2) 기본 수형과 연산

함수를 정의하기 위해 필요한 수형은 기본 수형과 구조 수형 두 가지로 나뉜다. 기본 수형으로는 C 언어의 기본 수형인 char, short, int, long, unsigned, float, double이 제공된다. 구조 수형으로는 C 언어에서 제공되는 struct, union 이외에 array, list , stream이 제공된다. 여기에서 관심을 둘 사항은 C 언어의 pointer 수형이 제공되지 않는다는 점이다. 이것은 pointer 사용에 의한 side-effect 유발 가능성을 배제시키자는 의도에 의한 것이다. 따라서 pointer에 의한 array 선언이 허용되지 않는 대신 첨자범위가 정해지지 않는 array 선언이 제공된다. Array는 최대 3차원(dimension)까지 허용되며 각 차원마다 차원변수가 제공되는데 $x, $y, $z가 바로 그것이다. 이들은 각각 1차원, 2차원, 3차원을 나타낸다. 다음은 전용언어에서 허용하는 변수들의 대한 선언의 형식을 나타낸다.

```
<선언_리스트> ::= <선언>
                | <선언> <선언_리스트>
<선언> ::= <선언_명시> ;
         | <선언_명시> = <초기값_지정> ;
<선언_명시> ::= <수형_명시> <변수_리스트>
              | typedef <수형_명시> <수형정의_이름>

<수형_명시> ::= char
              | short
              | int
              | long
              | unsigned
              | float
              | double
              | <struct_또는_union_명시자>
              | <array_명시자>
              | <list_명시자>
              | <stream_명시자>
              | <수형정의_이름>
```

```
<초기값_지정> ::= <상수>
                | ( <원소_리스트> )
                | [ <원소_리스트> ]
                | { ( <원소_리스트> ) }
                | { ( <원소_리스트> ) ( <원소_리스트> ) }
                | { ( <원소_리스트> ) ( <원소_리스트> ) ( <원소_리스트> ) }

<struct_또는_union_명시자> ::=
              | struct { <struct_선언_리스트> }
              | srtruct <수형정의_이름> { <struct_선언_리스트> }
              | union { <struct_선언_리스트> }
              | union <수형정의_이름> { <struct_선언_리스트> }
              | union <명칭>
<struct_선언_리스트> ::= <struct_선언>
                       | <struct_선언> <struct_선언_리스트>
<struct_선언> ::= <수형_명시> <수형정의이름>

<array_명시자> ::= <수형_명시> <명칭> []
                 | <수형_명시> <명칭> [ <상수> ]
<list_명시자> ::= list <수형_명시> <명칭>
<stream_명시자> ::= stream <수형_명시> <명칭>
```

이들 수형에 대한 연산으로는 산술 연산, 논리 연산, 관계 연산 그리고 구조 연산이 있다. 산술 연산에는 덧셈, 뺄셈, 곱셈, 나눗셈 그리고 modulo 연산이 제공된다. 논리 연산에는 &&(and), ||(or), !(not) 연산이 제공되고 관계 연산에는 < (작다), <= (작거나 같다), == (같다), > (크다), >= (크거나 같다), != (같지 않다) 등의 연산이 제공된다. 이들 관계 연산자는 관계가 성립하지 않으면 0을 그렇지 않으면 0이 아닌 값을 반환한다. 즉 0은 FALSE를 의미하고 0이 아닌값은 TRUE를 의미한다. 구조 연산에는 구조 수형에 따라 array 연산, list 연산, stream 연산이 존재한다.

Array 연산에는 array_apply가 있다. 이 연산은 다음과 같은 구조를 취한다.

<array_이름> ::= array_apply(범위수식,생성함수이름);

범위수식은 array의 각 차원의 첨자범위를 나타내며 array의 차원에 따라 세 개의

범위수식이 나열될 수 있다. 범위수식은 시작첨자와 종료첨자가 ":"으로 접속된 형태를 취한다. 생성함수는 array의 각 원소의 값을 생성하는 함수로서 차원변수 $x, $y, $z를 매개변수로 취하며 array의 차원과 동일한 매개변수 갯수를 갖는다. 특별히 new_array라는 생성함수가 미리 주어지는데 이것은 array를 최초로 생성하여 적당한 초기치를 설정해주는 함수이다.

List 연산에는 car, cdr, cons, append, list 등이 제공되고 있다. car은 인자로 전달된 리스트의 첫번째 원소를 반환하고, cdr은 인자로 전달된 리스트에서 첫번째 원소를 제외한 나머지 원소들로 구성된 리스트를 반환한다. cons는 값과 리스트를 받아서 이들을 원소로 하는 리스트를 반환한다. append는 두개의 리스트를 전달받아 이들을 합병함으로써 생성되는 리스트를 반환한다. List는 임의의 갯수의 리스트를 전달받아 이들을 원소로 하는 리스트를 반환한다.

Stream 연산에는 stream_empty, steam_first, stream_rest, stream_cons, stream_conc 다섯 가지가 있다. Stream_empty는 주어진 스트림이 비었는지의 여부를 알려주는 함수로서 비었으면 1(true), 그렇지 않으면 0(false)의 값을 반환한다. Stream_first는 스트림의 첫번째 원소를 추출하며, stream_rest는 그 나머지 원소들의 스트림을 반환한다. Stream_cons는 하나의 원소를 하나의 스트림 뒤에 넣는 것이며, stream_conc는 하나의 스트림 뒤에 또하나의 스트림을 접속시키는 것이다. 구조연산에 대한 구문은 다음과 같다.

```
<수식> ::= <순수수식>
         | <순수수식> <프로세서_할당자>
<순수수식> ::= <산술수식>  | <논리수식> | <관계수식>
             | <array수식> | <list수식> | <stream수식>
             | <기본수식>  | <치환수식>
<산술수식> ::= <수식> <산술연산자> <수식>
<산술연산자> ::= + | - | * | / | %
<논리수식> ::= <논리연산자1> <수식>
             | <수식> <논리연산자2> <수식>
<논리연산자1> ::= !
<논리연산자2> ::= & | '|'
<관계수식> ::= <수식> <관계연산자> <수식>
<관계연산자> ::= < | <= | == | > | >= | !=
```

```
<array수식> ::= array_apply ( <범위수식_리스트> , <생성함수이름> )
<범위수식_리스트> ::= <범위수식>
                    | <범위수식> , <범위수식>
                    | <범위수식> , <범위수식> , <범위수식>
<범위수식> ::= <첨자수식> : <첨자수식>
<첨자수식> ::= <첨자상수>
             | <첨자상수> <산술연산자> <첨자수식>
<첨자상수> ::= <상수>
             | <명칭>
             | <첨자변수>
<첨자변수> ::= $x | $y | $z
<생성함수이름> ::= new_array
                 | <명칭>

<list수식> ::= cons(<리스트원소>, <리스트구조>)
             | append(<리스트구조> , <리스트구조>)
             | list(<리스트원소_리스트>)
             | car(<리스트구조>)
             | cdr(<리스트구조>)
<리스트구조> ::= ()
               | nil
               | ( <리스트원소_리스트> )
<리스트원소_리스트> ::= <리스트원소>
                      | <리스트원소>, <원소_리스트>
<리스트원소> ::= <상수>
               | <명칭>
               | #<명칭>

<stream수식> ::= stream_empty ( <스트림구조> )
               | stream_first ( <스트림구조> )
               | stream_rest  ( <스트림구조> )
               | stream_cons  ( <원소> , <스트림구조> )
               | stream_conc  ( <스트림구조> , <스트림구조> )
<스트림구조> ::= []
               | eos
               | [ <스트림원소_리스트> ]
<스트림원소_리스트> ::= <스트림원소>
                      | <스트림원소> , <스트림원소_리스트>
<스트림원소> ::= <상수>

<기본수식> ::= <명칭>
```

```
                | <상수>
                | (<수식>)
<치환수식> ::= <명칭> = <수식>
```

(3) 문장

전용언어의 문장은 크게 병렬문장과 순차문장으로 구분된다. 구문(syntax) 상 순차문장은 병렬문장을 포함하나 의미(semantics) 상 동일한 구문에 대해 지켜지는 규칙이 다르다. 즉, 단일치환규칙 준수여부가 동일한 문장에 대하여 병렬문장인지 순차문장인지를 구분시킨다. C 언어의 문장 중에서 의미 상으로 함수언어의 성질을 무시하는 문장은 순차문장에만 존재한다. 예를 들어 +=, -= 등과 같은 치환문장과 while, for 등과 같은 순차문장 그리고 goto와 같은 분기문장은 순차문장이며 병렬문장이 아니다. 다음은 전용언어에서 제공하는 문장들을 나타낸다.

```
<병렬문장> ::= if ( <수식> ) { <병렬문장_리스트> }
             | if ( <수식> ) { <병렬문장_리스트> } else { <병렬문장_리스트> }
             | switch ( <수식> ) { <case_문장_리스트> }
             | default : <병렬문장_리스트>
             | return ;
             | return <병렬문장> ;
             | <lvalue> = <병렬수식> ;
             | <명칭> ;
             | #<명칭> ;
             | ;
<case_문장_리스트> ::= case <수식> : <병렬문장_리스트>
                     | case <수식> : { <병렬문장_리스트> } <case_문장_리스트>
<lvalue> ::= <명칭>
<rvalue> ::= <상수>
           | <명칭>
<병렬수식> ::= <rvalue>
             | & <병렬수식>
             | - <병렬수식>
             | ! <병렬수식>
             | ~ <병렬수식>
             | ( <수형_이름> ) <병렬수식>
             | <병렬수식> <이진연산자> <병렬수식>
<이진연산자> ::= * | / | % | + | - | >> | << | < | > |
               | <= | >= | == | != | & | ^ | '|' | && | '||'
```

```
<순차문장> ::= if ( <수식> ) { <순차문장_리스트> }
             | if ( <수식> ) { <순차문장_리스트> } else { <순차문장_리스트> }
             | while ( <관계수식> ) { <순차문장_리스트> }
             | do { <순차문장_리스트> } while ( <관계수식> );
             | for ( <순차문장_리스트>; <관계수식>; <순차문장_리스트>})
                 { <순차문장_리스트> }
             | switch ( <수식> ) { <case_문장_리스트> }
             | default : <순차문장_리스트>
             | break;
             | continue;
             | return;
             | return <순차문장>;
             | goto <라벨>;
             | <라벨> : <순차문장>;
             | <lvalue> = <순차수식>;
             | ;
<case_문장_리스트> ::= case <수식> : <순차문장_리스트>
                     | case <수식> : { <순차문장_리스트> } <case_문장_리스트>
<순차수식> ::= <rvalue>
             | & <순차수식>
             | - <순차수식>
             | ! <순차수식>
             | ~ <순차수식>
             | ++ <순차수식>
             | -- <순차수식>
             | <lvalue>++
             | <lvalue>--
             | ( <수형_이름> ) <순차수식>
             | <순차수식> <이진연산자> <순차수식>
             | <순차수식> ? <순차수식> : <순차수식>
             | <lvalue> <치환연산자> <순차수식>
             | <순차수식> , <순차수식>
<이진연산자> ::= * | / | % | + | - | >> | << | < | > |
               | <= | >= | == | != | & | ^ | '|' | && | '||' | ?:
<치환연산자> ::= = | += | -= | *= | /= | %= | >>= | <<= | &= | ^= | !=
```

3.1.7. 프로그램 작성의 예

제안된 전용언어를 사용하여 프로그램을 작성한 예를 세 가지만 살펴 보겠다. 첫번째 예는 factorial 계산이고 두번째 예는 n × n linear system의 solution vector를 LU-factorization 방법으로 구하는 것이며 세번째 예는 두번째 예와 동일한 문제를 Cramer's rule로 구하는 것이다. 첫번째 예제 프로그램의 경우, granularity 조절이 어떻게 일어나며 프로세서 할당자가 어떻게 유용하게 사용될 수 있는지에 대하여 보여준다. 두번째와 세번째 예제 프로그램의 경우, 프로세서 할당자와 더불어 array의 비결정적 처리(nondeterministic manipulation)에 대해서 보여 주는데 두번째 예제는 pipeline parallelism을 위주로 하며 세번째 예제는 1-master/multi-slave 형태의 parallelism을 위주로 한다. 특히 세번째의 경우 뒤에 나오는 3.3.2에서 성능 분석의 예제 모델로 사용된다.

(1) 예제 프로그램 I: factorial 계산

프로그램의 첫번째 예로서, 주어진 정수 n에 대한 factorial 값을 구하는 프로그램을 제시하고자 한다. Factorial을 구하는 문제를 순차 처리 프로그램으로 작성할 경우, 순환구문에 의한 곱셈의 누산으로만 가능하지만 이 예제에서는 제안된 전용언어의 특징을 잘 드러내도록 다음과 같이 병렬처리성을 제공하는 부함수 binary_fact를 사용한다. 그리고 지나친 통신부하로 인하여 병렬처리성에 의한 이득이 상쇄되는 것을 방지하기 위해 어느 정도 한계에 다다르면 순환 구문에 의한 순차처리가 이루어지도록 순차 함수 seq_fact를 사용한다.

```
def fact (n)
{
     return binary_fact(1,n);
}
```

```
def  binary_fact (low,high)
{
     if (low == high) return low;
     else if (low == (high-1)) return low * high;
     else if (low > (high-10)) seq_fact (low,high);
     else
     {
          mid = (low+high)/2;
          v1 = binary_fact (low,mid) $+0;
          v2 = binary_fact (mid+1, high) $+1;
          return v1 * #v2;
     }
}
```

```
seq_def seq_fact(low, high)
int low, high;
{
     int sum = 1;
     int index = low;

     while (index <= high)
     {
          sum = sum * index;
          index++;
     }
     return sum;
}
```

함수 binary_fact 는 조건에 따라서 recursive하게 자신을 두번 호출한다. 그리고 이들 호출은 predemand evaluation의 명시에 의해 병렬로 처리된다. 그리고 병렬수행되는 v1과 v2의 두 함수 호출은 현재의 프로세서($+0)와 바로 오른쪽($+1)에 이웃한 범용 프로세서에서 수행되도록 프로세서 할당자에 의하여 명시되어 있다. 한편 seq_fact 호출은 프로세서 할당자에 의해 명시되어 있지 않으므로 함수 binary_fact를 수행중인 프로세서에 의해서 수행이 이루어지되 순수한 폰노이만 방식으로 함수의 코드블럭이 수행된다. 함수 binary_fact안에서 함수 seq_fact를 호출하는 조건부에 의하여 low와 high의 값이 조절됨에 따라 다양한 granularity를 가지게 된다. 따라서 통신부하와 병렬처리성을 총체적으로 고려하여 가장 이득이 큰 경우를 선택할 수 있게 된다.

(2) 예제 프로그램 II: Solving n × n Linear System (LU-factorization method)

다음과 같이 n개의 미지수에 대한 n개의 선형방정식의 해를 구하는 문제를 생각해 보자.

$$
\begin{aligned}
(E_1)\ & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
(E_2)\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
(E_3)\ & a_{31}x_1 + a_{32}x_2 + \ldots + a_{3n}x_n = b_3 \\
& \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\
(E_n)\ & a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n
\end{aligned}
$$

여기에서 a_{ij}와 b_{ij}는 scalar이고 (i,j in [1..n]), $x_1, x_2, \ldots, x_n$은 미지수이다. 이 문제는 AX=B 라는 행렬식에 대해 vector X를 구하는 문제로 변환된다. 이 때, $A = (a_{ij})_{nxn}$ (coefficient matrix), $X = [x_1\ x_2\ \ldots\ x_n]^T$ (solution vector), $B = [b_1\ b_2\ \ldots\ b_n]^T$ (constant vector) 이다. 여기서 유념해야 할 사항은 행렬 A가 nonsingular(invertable)할 경우만 vector X를 구할 수 있다는 것이다. 이러한 n × n linear system을 해결하는 방법은 여러 가지가 존재한다.

한 가지 방법은, A의 역행렬을 이용하여 푸는 것이다. A가 nonsingular하다고 가정했으므로 A의 역행렬 A^{-1}가 존재한다. 따라서 X는 $A^{-1}B$의 계산을 통하여 구할 수 있다.

다른 한 가지 방법은, Gaussian Elimination에 의하여 푸는 것이다. 이는 행렬 A에 Interchange, Scale, Subtract를 사용킴으로써 U(unit upper triangular matrix)로 변형한 후 backward substitution을 적용시켜 solution vector X를 구하는 것이다.

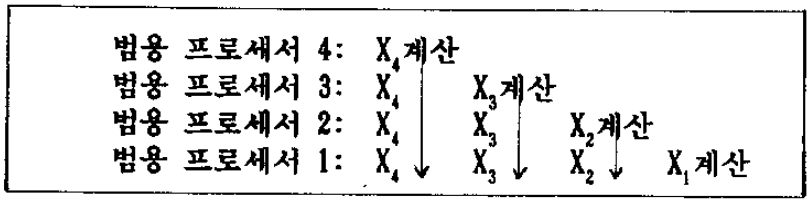
또다른 한가지 방법은 LU-factorization에 의하여 푸는 것이다. 이 방법은 세 단계로 구성된다. 단계 1은 LU-factorization (A = LU)에 의하여 행렬 A를 두개의 행렬 L과 U의 곱(product)으로 나타낸다. 이때 L은 lower triangular 행렬 ($l_{ij} = 0$ for $i < j$)이고 U는 upper triangular 행렬($u_{ij} = 0$ for $i > j$)이며 A,L,U 모두 n x n 행렬이다. 하나의 A에 대하여 L,U는 여러 가지가 가능하다. 여기 예제에서는 특별히 U의 경우 $u_{ii} = 1$인 unit upper triangular 행렬을 생각하자. 단계 2는 forward substitution (LC = B)으로서 행렬식 LC = B를 만족하는 vector C를 구한다. 단계 3은 backward substitution (UX = C)으로서 행렬식 UX = C를 만족하는 vector X를 구한다. 이상에서 살펴보면, $C = L^{-1}B$이고 $X = U^{-1}C$이므로 $X = U^{-1}C = U^{-1}(L^{-1}B) = (U^{-1}L^{-1})B = (LU)^{-1}B = A^{-1}B$가 되어 원래 구하고자 하는 X를 구할 수 있음을 알 수 있다. 본 예제 프로그램에서는 단계 3을 한정적으로 취급하겠다. 단계 1과 단계2는 약간의 변형에 의하여 단계3과 같은 유형으로 변형이 가능하기 때문이다.

단계 3은 행렬식 UX=C에서 X를 구하는 문제이다. 여기에서 U는 unit upper triangular matrix (n x n)이고, X는 Solution Vector = $[x_1\ x_2\ \dots\ x_n]^T$이며, C는 Constant Vector = $[c_1\ c_2\ \dots\ c_n]^T$이다. 따라서 이 문제는 다음과 같은 알고리즘으로 풀어진다.

```
for i = n downto 1 step -1 do
    Xi = Ci - the summation of UikXk for k > i
```

이 때 병렬처리성을 분석해 보자. n = 4일 경우를 생각해 보면, 다음과 같은 pipeline

parallelism과 data dependancy를 알 수 있다.



최초의 요구신호가 범용 프로세서 1에서의 X_1 계산에 주어질 경우 이를 위해서 연속적인 요구신호가 발생하여 나머지 X 값을 계산한다. 중복된 요구는 단한번의 계산만으로 충족되어 graph reduction의 장점을 만족시킨다. 일반적으로 해 X_i를 구하는 과정에서 (i−1)번의 요구신호가 전송되지만 계산은 단한번에 이루어진다. 다음은 설계된 전용언어로 프로그램을 작성한 결과이다.

```
def int third_step[](n,U,C)
int n, U[], C[];
{
    int X[], Current_X[][];

    return(X);
    X = array_apply(1:n, back_sub);
    back_sub($x) = C[$x] - summation(n, $x, 0) $$x;
        /* 범용 프로세서 [1..n] 모두에 함수 summation()을 저장한다.
           이때 배열 C의 원소는 자신의 첨자와 일치하는 범용 프로세서에서
           각각 생성되는 것이 통신부하면에서 유리하게 된다 */
    summation(from, to, sum) =
        if (from == to) sum
        else summation(from-1, to, sum + Current_X[to][n-from+1]*U[to][from]);
    Current_X = array_apply(1:n, 1:n-$x+1, setup) $$x;
        /* 2차원 배열 Current_X는 pipeline parallelism을 지원해준다 */
    setup($x, $y) = if (n-$x+1 == $y) X[$x]
                    else Current_X[$x+1][$y];
        /* 연쇄적인 요구신호 전송을 지원해주는 함수이다 */
}
```

(3) 예제 프로그래밍 III: Solving n × n Linear System (Cramer's rule)

이번에는 앞의 예제 II에서 제기된 n × n linear system을 determinant를 이용하여 푸는 방법인 Cramer's rule에 대하여 살펴보자. Cramer's rule은 다음과같이 정리된다.

$$X_i = \det(A_i) \,/\, \det(A) \quad \text{for } i \text{ in } [1..n]$$

여기에서 det(A)는 행렬 A의 determinant를 나타내며 $Det(A_i)$는 행렬 A의 i번째 열을 constant vector B로 대입한 n × n 행렬(A_i)의 determinant를 나타낸다. Det (A)와 $det(A_i)$는 다음과 같이 구할 수 있다.

단계 1 LU-Factorization (A or A_i = LU)

단계 2 $\det(A) \text{ or } \det(A_i) = (-1)^t \det(L)$
단, t는 단계 1에서 pivot을 구하기 위해 행해졌던 행교환의 횟수이다.

이 경우에 대해서 병렬처리성을 분석하면 다음 그림 3.6과 같이 정리된다.

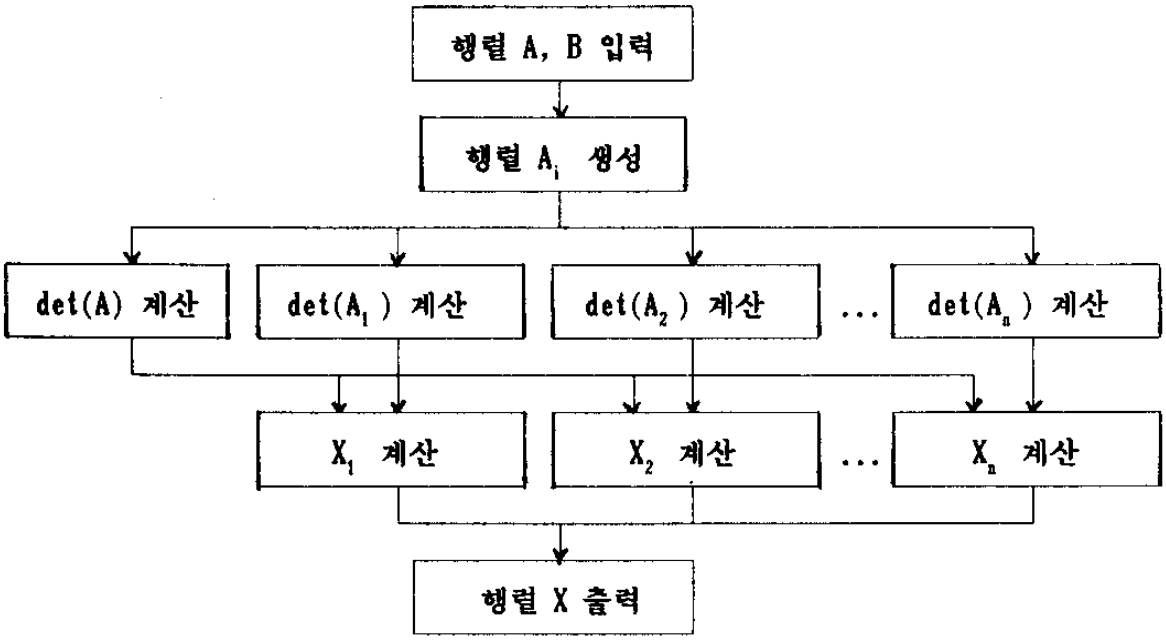


그림 3.6 Cramer's rule의 병렬처리성

이상의 사실들을 토대로 프로그램을 작성하면 다음과 같다.

```
def real Cramer_Rule[](n,A,B)
int  n;
real A[][], b[];
{
     real divider;
     real Ai[][][], X[];

     return(X);
     X = array_apply(1:n, solve);
     solve($x) = det(#n,#Ai[$x]) / #divider $$x;   /* (n + 1) 개의 병렬처리성 */
     divider = det(#n,A) $$self;
     Ai = array_apply(1:n, 1:n, 1:n, column_replace); /* 각 입력 행렬 생성 */
     column_replace($x,$y,$z) = if($x == $z) B[$y] else A[$y][$z];
}
```

```
seq_def det(n,A)              /* predefined evaluation */
int  n;
real A[][];
{
     int   i, j, m, interchange_row;
     real  flag, tmp[], Pivot[], L[][], U[][];

     flag = 1.0;    /*  +1.0 또는 -1.0  */
     tmp = array_apply(1:n, new_value);   /* unbounded array의 초기화 */
     L = U = array_apply(1:n, 1:n, new_value);
     U = array_apply(1:n, 1:n, new_value);
     for(m = 1; m <= n; m++) {  /* outer loop */
         for(i = m; i <= n; i++) {    /* 행렬 L 구하는 과정 */
             L[i][m] = A[i][m];
             j = 1;
             while(j < m) L[i][m] -= L[i][j] * U[j][m]; }
         Pivot[m] = L[m][m];      /* Partial Pivoting Method */
         interchange_row = m;
         i = m + 1;
         while (i <= n)
             if ( abs(Pivot[m]) < abs(L[i][m]) ) {  /* 가장 큰 절대값의 원소 */
                  Pivot = L[i][m];
                  interchange_row = i; }
         if (interchange_row <> m) {  /* 행렬 A의 행 교환 */
```

```
            for(i = 1; i <= n; i++) {
                tmp[i] = A[m][i];
                A[m][i] = A[interchange_row][i];
                A[interchange_row][i] = tmp[i]; }
            for(i = 1; i <= m; i++) { /* 행렬 LU의  행 교환 */
                temp[i] = L[m][i];
                L[m][i] = L[interchange_row][i];
                L[interchange_row][i] = temp[i]; }
            temp[1] = B[m];  /* 행렬 B의 행 교환 */
            B[m] = B[interchange_row];
            B[interchange_row] = temp[1];
            flag = - flag; }
        for(i = m + 1; i <= n, i++)   /* 행렬 U 구하는 과정 */
            U[m][i] = A[m][i];
            j = 1;
            while(j < m) U[m][i] -= L[m][j] * U[j][i];
            U[m][i] = U[m][i] / L[m][m]; }
    }
    determinant = 1;
    for(m = 1; m <= n; m++) determinant *= Pivot[m]; /* Pivot 곱셈 */
    return( flag * determinant);   /* determinant 값을 반환 */
}
```

3.1.8. 컴파일러 구성을 위한 문제 제기

전용언어를 위한 컴파일러를 구성할 경우 다음과 같은 사항들에 유념해야 한다.

첫번째 사항은, 프로세서 할당자 지정에 따른 코드블럭 생성 문제이다. 상수로 이루어진 절대 할당자의 경우, 컴파일러는 해당 함수의 코드블럭을 지정된 범용 프로세서에만 저장하면 된다. 그러나 그외의 경우, 다시 말하면 변수로 이루어진 절대 할당자나 상대 할당자는 프로그램 수행도중에 범용 프로세서의 위치가 결정되므로 함수 코드블럭을 중복 저장해야 한다. 이것은 코드블록의 이동에 따른 통신부하를 최소화시켜보려는 목적에서 취하는 전략이다.

두번째 사항은, recursive function에 대한 코드블럭 생성문제이다. 이러한 부류의 함수는 프로그램 수행 도중에 부하균형에 의하여 수행요구신호가 다른 범용 프로세서로 이동될 수 있으므로 이에 따른 통신부하를 경감시킨다는 목적에서 이 함수의 코드블럭 역시 범용 프로세서 사이에 중복 저장시킨다.

세번째 사항은, predemand evaluation이 명시된 부분을 지원해주는 문제이다. 앞에서 언급 했듯이 predemand evaluation은 1-grain 변수에 한정하여 유용하므로 1-grain 변수가 아닌 경우에는 이를 찾아 내어 프로그램 작성자에게 알려줄 필요가 있다.

네번째 사항은, 일반적인 오류보고(error report)에 관한 문제이다. 전용언어가 강한 수형의 언어이므로 수형 불일치에 의한 오류보고가 필요하다. 또한 전용언어가 함수언어이므로 def로 정의된 함수내에서 단일치환규칙이 지켜지는지 살펴볼 필요가 있으며 sed_def로 정의된 함수내에서 C 언어에서 처럼 external 변수나 static 변수가 사용되어 함수성을 어기고 있는지 점검해야 한다.

이러한 것들이외에도 많은 사항이 컴파일러 구성을 위하여 연구되어야 한다. 이에 관한 상세한 사항은 본 보고서의 수준을 넘으므로 생략한다.

3.2 Math. Package 전용 컴퓨터의 설계

1차년도 연구에서는 범용 프로세서, 통신 프로세서와 루트 프로세서가 X-트리 형태로 연결된 구조를 갖는 전용 컴퓨터의 윤곽을 제시하였다. 전용 컴퓨터의 구성도는 그림 3.7과 같다.

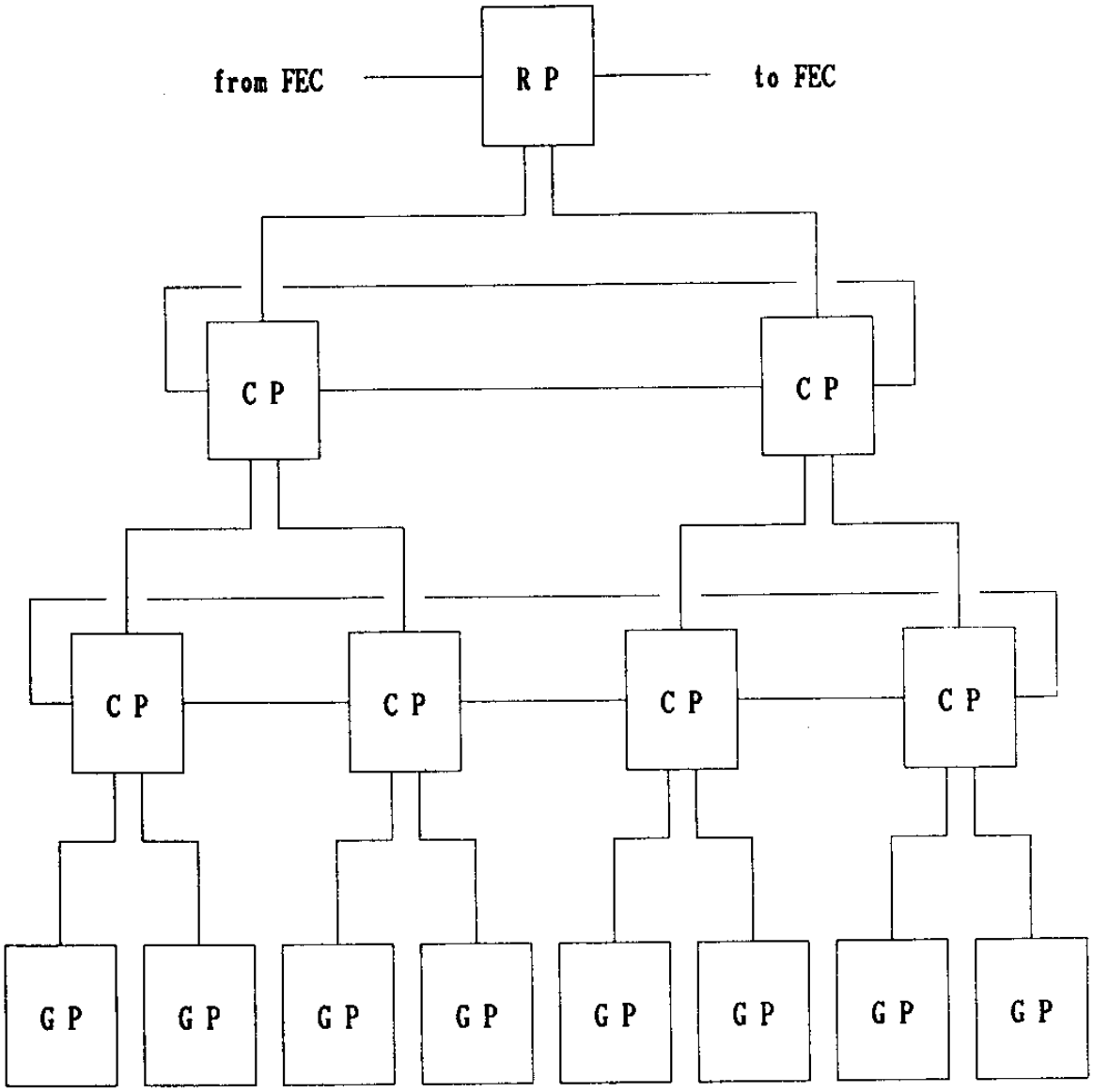


그림 3.7: Math. Package 전용 컴퓨터의 구성도.

3.2.1 설계 방향

본 연구팀에서 설계한 전용 컴퓨터는 기본적으로 리덕션 컴퓨터이며, 다중 프로세서 시스템이다. 따라서 다중 프로세서 시스템이 만족해야 하는 기준들에 대한 검토가 필요하며, 리덕션 컴퓨터 고유의 특징을 살릴 수 있는 사항들을 고려해야 한다.

첫째, 본 시스템은 하나의 프로세서가 자신의 지역 메모리를 독립적으로 제어하는 약결합성 구조(Loosely-coupled Architecture)를 채택하고 있다. 일반적으로 버스 등을 써서 공유 메모리에 접근하는 강결합성 시스템(Tightly-coupled System)은 성능 향상에 대한 한계를 가지고 있을 뿐만 아니라, VLSI 기술발전의 측면에서 확장성을 고려할 경우에도 약결합성 시스템에 뒤진다고 알려져 있다. 따라서 본 전용 컴퓨터는 약결합성 구조로 설계되었다.

둘째, 본 전용 컴퓨터는 각 프로세서들만의 상호 연결망으로 X-트리를 채택하였다. 여기서는 편의상 가장 간단한 형태인 이진트리로 전체 구조를 제한하고 있다. 'X(cross)'는 트리의 한 내부 노드들과 그 인접 노드 사이에 환형 링크(Circular Link)가 존재한다는 것을 의미하는데, 이 환형 링크를 X-링크라고 정의한다. 또한 상호 연결망의 각 프로세서들은 패킷을 사용하여 통신하는데, 패킷은 트리의 branch나 X-링크를 통해 전송된다.

세째, 본 전용 컴퓨터로 함수 수준의 granularity를 채택하고 있다. 일반적으로 granularity가 클수록 병렬 처리성은 적고 그로 인한 통신 부하는 줄어드는 반면, granularity가 작을수록 병렬 처리성은 많아지고 통신 부하는 증가한다. 본 전용 컴퓨터는 앞서서 설명한 것과 같이 프로그래머가 정의한 함수 크기 정도의 granularity가 적합하다고 판단하고 있는데, 이는 리덕션 수행방식이 수행 요구 신호의 전송이라는 추가적인 통신 부하를 가지고 있고, 또한 프로그래머가 정의하는 순차처리의 단위로 함수 수준이 적절하기 때문이다.

3.2.2 시스템의 기본 개요

본 전용 컴퓨터는 세 종류의 프로세서로 구성되며, 이들은 X-트리 형태의 상호 연결망으로 결합되어 있다.

범용 프로세서(GP: General Processor)는 X-트리의 제일 하단에 위치하며, 프로그램의 수행을 담당한다. 통신 프로세서(CP: Communication Processor)는 범용 프로세서간의 통신 및 부하 균형을 담당하며 상호 연결망의 중앙 부분에 위치한다. 루트 프로세서(RP: Root Processor)는 전단 컴퓨터와의 통신과 부하 균형을 담당하는데, X-트리의 제일 상단에 위치한다.

각 프로세서와 상호 연결망의 구성을 간략히 설명하면 다음과 같다.

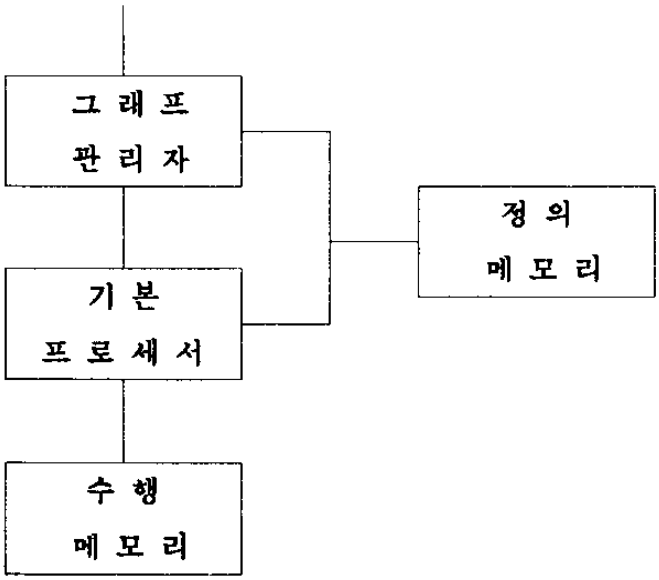


그림 3.8: 범용 프로세서의 구성도.

범용 프로세서는 그림 3.8과 같은 구조를 가진다. 하나의 범용 프로세서는 그래프 관리자(Graph Manager), 기본 프로세서(Primary Processor), 정의 메모리(Definition Memory)와 수행 메모리(Execution Memory)로 구성된다.

그래프 관리자는 기본 프로세서에게 작업을 지시하고, 생성된 결과를 지정된 주소로 전달하며, 기본 프로세서가 동작하면서 발생시키는 요구들을 처리한다. 또한 기본 프로세서의 부하 상태를 상위의 통신 프로세서에게 전달하고, 필요시에는 작업 이동을 지시한다.

기본 프로세서는 그래프 관리자의 수행 요구에 따라 수행 메모리상에서 폰 노이만 방식에 의해 작업을 처리하는데, 수행 메모리상의 코드는 동일 범용 프로세서 내의 정의 메모리로부터 복사되거나 외부 범용 프로세서로부터 전달된다. 기본 프로세서가 함수를 수행하면서 또 다른 요구 신호에 접할 경우, 해당 함수는 문맥교환(Context Switch)되고 새로운 작업이 이어서 수행된다.

정의 메모리는 함수에 대한 코드 블럭을 저장하며 컴파일러의 선택에 따라 일부 함수는 다수의 범용 프로세서에 중복해서 저장된다. 즉 특정 함수를 저장하는 범용 프로세서들이 환형 리스트(Circular List) 혹은 선형 리스트(Linear List)를 구성한다.

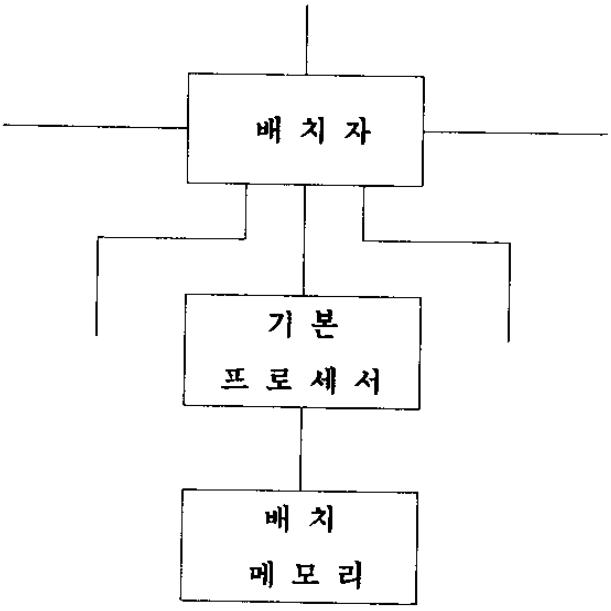


그림 3.9: 통신 프로세서의 구성도.

수행 메모리는 현재 기본 프로세서에 의해 수행중인 코드 블럭들을 저장하고 있는 메모리로 파괴적(destructive) 수행 방식을 지원한다.

통신 프로세서는 그림 3.9와 같은 구조를 가진다. 하나의 통신 프로세서는 배치자(Router), 기본 프로세서와 배치 메모리(Routing Memory)로 구성된다.

배치자는 자신이 속한 통신 프로세서를 상위의 프로세서나 하위의 프로세서에 연결시키고, 인접 통신 프로세서와의 통신 경로를 제공한다. 배치자는 6개의 링크들을 통해 store & forward 방식으로 패킷들을 전송한다.

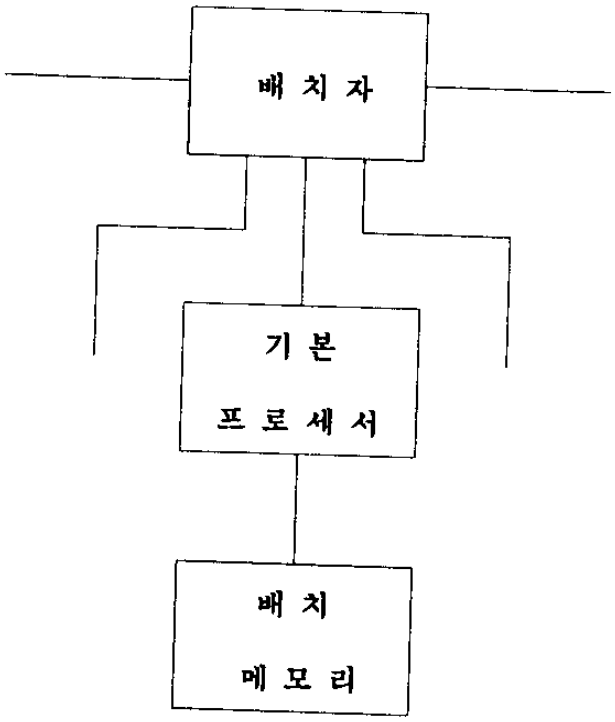


그림 3.10: 루트 프로세서의 구성도.

기본 프로세서는 범용 프로세서의 그것과 달리 부하 균형만을 담당한다. 하위 프로세서로부터 부하 균형에 관한 정보를 전달받으며, 필요에 따라서는 상위 프로세서에게 자신이 가지고 있는 정보를 전달한다. 또한 부하 이동의 필요성이 발생하는

경우 하위 프로세서에게 부하 이동에 관한 정보, 즉 부하를 받아들일 수 있는 프로세서의 위치를 알려주며, 통신을 위해 어떤 링크를 사용할 지 결정하는 역할도 담당하는데, 이것은 X-링크를 효율적으로 사용할 수 있도록 지원한다는 것을 의미한다.

배치 메모리는 기본 프로세서가 부하에 관한 정보를 보관하는 메모리로, 필요에 따라 교정되어 최근의 상태를 유지한다. 하위의 프로세서에서 전달되는 부하값들은 배치 메모리에 기록되며, 이것들의 해석에 의해 상위의 프로세서에 전달되는 정보가 생성된다.

루트 프로세서는 그림 3.10과 같은 구조를 가진다.

하나의 루트 프로세서는 통신 프로세서와 같이 배치자, 기본 프로세서, 배치 메모리로 구성되는데, X-링크가 통신 프로세서의 그것과 달리 단방향의 통신 경로로 사용되며, 상위의 프로세서에 대한 통신 경로를 가지지 않는다. 따라서 본 전단 컴퓨터가 필요로 하는 입출력은 모두 루트 프로세서를 통해서 일어나며, 루트 프로세서는 전단 컴퓨터와의 인터페이스를 통해 이러한 역할을 수행한다.

배치자는 루트 프로세서를 전단 컴퓨터나 하위 통신 프로세서에 연결시키는데, 통신 프로세서의 그것과는 달리 다른 목적으로 사용된다. 즉 통신 프로세서간의 통신 경로가 아닌, 전단 컴퓨터와 통신 프로세서간의 연결 통로로 사용된다. 왜냐하면 하위의 통신 프로세서간의 X-링크가 통신 프로세서들을 서로 연결시키기 때문이다.

배치 메모리는 기본적으로 통신 프로세서의 그것과 같다.

3.2.3 수행 방식

프로그래머에 의해 작성되어 전용 컴퓨터에서 처리되는 프로그램은 응용면에서 볼 때 Math. Package로 분류될 수 있다. 이것은 워크 스테이션, 전단 컴퓨터, 전용 컴퓨터와의 유기적인 결합 속에서 수행된다. 전용 컴퓨터에서 수행되어야 할 프로그램은 워크 스테이션상의 컴파일러에 의해 전용 컴퓨터가 인식할 수 있는 형태의 목적 코드들로 번역(cross-compile)된다. 이 때 컴파일러는 각 함수에 대해 전용

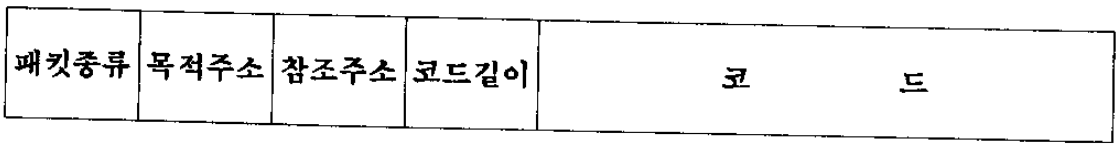
컴퓨터상의 실제 주소를 할당하는데, 각 범용 프로세서에 함수들을 고루 분포시킬 수 있도록 배려해야 한다. 전용 컴퓨터의 주소는 범용 프로세서의 번호와 범용 프로세서내의 지역 주소의 접속으로 만들어진다. 또한, 컴파일러는 특정 함수를 여러 범용 프로세서에 중복시키는 것이 바람직하다고 판단할 경우, 그 함수에 대해 다수의 실제 주소를 배정한다.

이렇게 생성된 코드 블럭들은 해당 전단 컴퓨터에 미리 전달되고, 각 전단 컴퓨터가 전용 컴퓨터에서 수행되어야 할 코드들을 관리한다. 워크 스테이션상에서 수행되던 프로그램이 전용 컴퓨터를 통한 계산 결과를 요구할 경우, 수행에 필요한 데이타가 전용 컴퓨터로 넘겨지고, 전단 컴퓨터는 해당하는 코드 블럭들과 전달받은 데이타들을 수행 요구의 형태로 전용 컴퓨터에 송신한다. 루트 프로세서는 전단 컴퓨터와의 인터페이스를 통해 정해진 규칙에 따라 수행 정보를 전달받고, 이것들은 전용 컴퓨터내의 통신 규약에 따라 적절한 패킷 형태로 변형되어 하위의 통신 프로세서로 넘겨진다. 각 단계의 통신 프로세서는 상위의 프로세서로부터 전달된 패킷들을 주어진 주소에 따라 하위의 프로세서로 넘겨 주고, 최종적으로 범용 프로세서가 자신에게 할당된 코드 블럭들을 받게 된다. 동시에 전단 범용 프로세서(Front GP)로 수행 요구에 관한 정보가 전달되고 이것에 의해 수행이 시작되는데, 각 범용 프로세서는 각각 병렬적으로 자신에게 맡겨진 작업을 처리한다. 이 과정에서 범용 프로세서사이에는 수행에 필요한 요구 신호들이 데이타와 함께 교환되고, 경우에 따라서는 코드 블럭들도 이동되어야 한다. 이제 전단 범용 프로세서로 각 범용 프로세서에서의 수행에 의한 결과가 모아지고, 이것은 패킷 형태로 통신 프로세서를 통해 루트 프로세서로 반환된다. 다시 루트 프로세서는 전단 컴퓨터와의 통신 규약에 따라 적절한 형식의 결과 패킷을 생성하여 전단 컴퓨터로 전달하며, 이것이 최종적으로 워크 스테이션에 전달됨으로써 계산을 마친다.

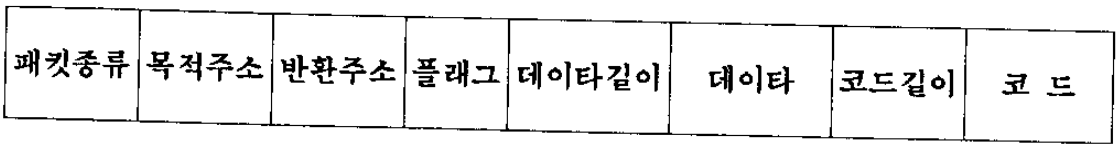
3.2.4 패킷 구조

범용 프로세서, 통신 프로세서와 루트 프로세서 사이의 정보 전달을 위해 사용되는 패킷들은 전송하는 정보의 성격에 따라 코드 패킷, 함수적용 패킷, 요구 패킷,

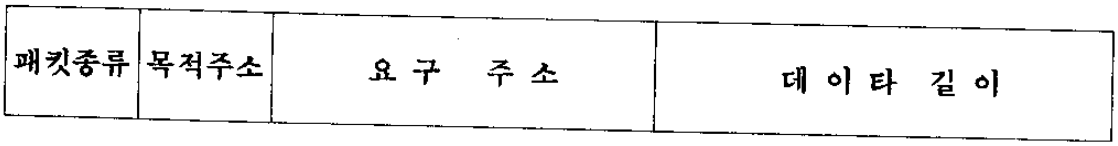
결과 패킷으로 구별할 수 있다. 각 패킷들은 그림 3.11과 같은 구조를 가지며, 전용 컴퓨터에서 배타적으로 사용된다.



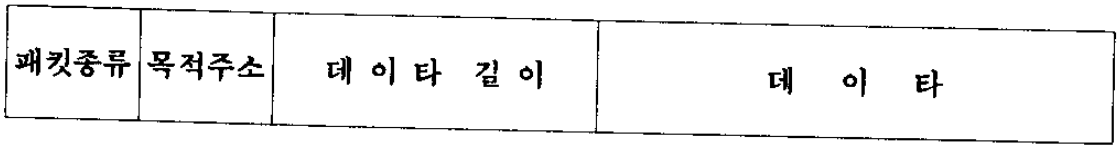
(a) 코드 패킷



(b) 함수적용 패킷



(c) 요구 패킷



(d) 결과 패킷

그림 3.11: 패킷 구조.

'패킷종류' 필드는 패킷들의 성격을 규정하는데 사용되며 '목적주소' 필드는 패킷이 전달될 주소를 나타내는데, 범용 프로세서의 번호나 지역 주소를 포함한다. 패킷의 처리를 용이하게 하기 위해 위의 두개의 필드는 같은 위치에 배치되도록 고려하

였다.

코드 패킷은 루트 프로세서로 전달된 코드 블럭들을 범용 프로세서로 옮기는데 사용된다. '참조주소' 필드는 해당 함수가 중복 저장되는 경우 환형 혹은 선형 리스트상의 다음 범용 프로세서 번호를 가지며, 코드 블럭들은 '코드' 필드에 저장된다.

함수적용 패킷은 범용 프로세서간의 함수 수행 요구 혹은 루트 프로세서로부터 발생된 전단 범용 프로세서로의 수행 요구를 전달하는데 사용된다. '반환주소' 필드는 해당 함수를 수행한 후 결과가 보내져야 할 주소를 가리키며, '플래그' 필드는 코드의 전이 여부를 표시하는데 쓰인다. 즉 수행 요구가 전달되는 범용 프로세서가 해당 함수에 대한 코드 블럭을 가지고 있지 않은 경우, 코드 부분도 수행 요구와 함께 동시에 전달되어야 하는데, 이것을 나타내기 위해 플래그가 사용된다. 따라서 플래그의 값에 따라 뒤의 '코드길이' 필드나 '코드' 필드가 생략될 수 있다.

요구 패킷은 범용 프로세서간의 데이타 요구를 처리하는데 사용된다. '요구주소' 필드는 원하는 데이타가 위치하고 있는 주소를 뜻하며, '데이타길이' 필드는 전송될 데이타의 길이를 나타낸다.

마지막으로 결과 패킷은 함수적용 패킷 혹은 요구 패킷에 의한 결과를 목적지에 전달하는데 사용된다. '목적주소' 필드는 함수적용 패킷의 반환 주소나 요구 패킷의 요구 주소로부터 생성되며, '데이타' 필드는 전달되어야 할 데이타들을 포함한다.

3.2.5 범용 프로세서

일반적인 관점에서 보면, 각 범용 프로세서는 자신에게 할당된 코드 블럭들을 정의 메모리에 보관하면서, 필요시 수행 메모리로 복사한 후 기본 프로세서에 의한 파괴적 수행 방식에 의해 결과를 산출한다. 이 때 그래프 관리자는 기본 프로세서가 함수를 수행하는데 필요한 제반 환경을 조성하는 역할을 담당한다.

우선 범용 프로세서가 주어진 함수를 수행하면서 관리해야 하는 시스템 표에 대해 설명한다. 기본 프로세서가 코드 블럭들과 데이타의 조합으로 함수를 수행하기

위해서는 그래프 관리자가 함수 주소표(FAT: Function Address Table), 함수 수행표(FET: Function Execution Table)와 함수 문맥표(FCT: Function Context Table)를 유지해야 한다.

함수 주소표는 워크 스테이션상의 컴파일러에 의해 생성된 각 함수별 주소를 관리한다. 이것은 범용 프로세서가 기호로 표현된 함수 이름으로부터 실제 주소, 즉 범용 프로세서의 번호와 범용 프로세서내의 지역 주소를 획득하는데 사용된다. 루트 프로세서로부터 각 범용 프로세서에 코드 블럭들이 전달될 때 함수 주소표에 관한 패킷도 함께 전파되어야 한다. 함수 주소표의 각 엔트리는 다음과 같다.

- 함수 이름
- 지역 주소: 해당 범용 프로세서내의 주소. 만약 해당 함수를 가지고 있지 않다면 NULL
- 참조 GP 번호: 컴파일러에 의한 판단에 따라 다른 범용 프로세서에 중복 저장된 경우, 해당 범용 프로세서가 참조할 수 있는 인접 범용 프로세서의 번호. 만약 해당 범용 프로세서만 그 함수를 가지는 경우 NULL
- 참조 지역 주소: 해당 범용 프로세서가 참조할 수 있는 범용 프로세서내에서 그 함수가 위치하는 주소

함수 수행표는 각 범용 프로세서가 수행하고 있거나 수행을 마친 함수들에 대한 정보를 유지하는데, 그래프 리덕션 수행방식에 따른 중복 계산을 피하는데 사용된다. 즉 계산이 완료된 함수에 대해서는 인자들의 리스트와 함께 결과값을 저장하며, 계산중인 함수의 경우는 수행에 필요한 정보와 함께 반환 주소를 유지한다. 함수 수행표의 각 엔트리는 다음과 같다.

- 함수 이름
- 인자 길이와 인자들에 대한 포인터: 해당 함수를 수행하는데 사용된 인자들에 관한 정보
- 결과 길이와 결과에 대한 포인터: 위의 인자들을 해당 함수에 적용시켜 산출한 결과값에 대한 정보

- 함수 문맥표에 대한 포인터: 함수 수행에 관련된 정보 혹은 반환 주소를 획득하기 위한 포인터

함수 문맥표는 각 범용 프로세서내에서 수행되고 있는 함수들에 대한 반환 주소와 함께 수행 환경에 관한 정보를 관리하는데, 이는 문맥 교환시에 유용하게 사용된다. 함수 문맥표의 각 엔트리는 다음과 같다.

- 함수 이름
- 반환 주소들의 리스트에 대한 포인터: 해당 함수를 수행한 후 결과를 전달해야 할 주소들의 리스트로 중복 계산 방지에 사용된다.
- 수행 상태: 해당 함수의 현재 상태를 표시하는데, R(unning), C(omputable), S(uspended), W(aiting) 혹은 H(ibernate) 등으로 표현한다. H-상태는 문맥만 만들어지고 전혀 수행되지 않았음을 의미하며, W-상태는 다른 함수로부터의 결과나 혹은 특정 주소로부터의 데이타를 기다리고 있음을 뜻한다. S-상태는 자원등의 부족으로 수행이 중단된 경우이며, C-상태는 S-상태 혹은 W-상태로부터 계산이 가능하도록 바뀐 상태로써 항상 수행될 수 있도록 대기하고 있음을 의미한다.
- 대기 주소들의 리스트에 대한 포인터: W-상태의 함수가 결과를 요구하는 곳의 위치들로써, 결과 패킷이 도착했을 때 수행 상태를 변화시키기 위해 사용된다.
- 프로그램 카운터: 해당 함수가 dispatch될 경우 수행이 계속되거나 혹은 시작될 명령어의 주소를 유지한다.
- 수행 메모리의 크기: 해당 함수를 수행하기 위해 확보해야 될 수행 메모리의 크기로 정의 메모리가 복사되어 수행되는 공간을 의미하는데, 기본 프로세서가 수행할 함수를 선택할 때 참조한다.
- 스택 포인터: 기본 프로세서가 해당 함수를 수행하면서 사용하는 스택에 대한 포인터를 뜻한다.
- 스택의 상/하한계: 할당한 스택 공간의 최상위/최하위 주소를 가리킨다.
- 다음 엔트리에 대한 포인터: 수행 상태별 문맥들이 리스트를 유지하기 위한 포인터로 자신과 같은 상태를 갖는 다음 문맥을 가리키는데, 기본 프로세서에서

의 스케쥴링을 위해 사용된다.

본 전용 컴퓨터는 병렬도를 높이면서, 통신 부하를 줄이기 위해 일부 코드 블럭들을 다수의 범용 프로세서에 중복 저장한다. 물론 중복 저장의 유무와 중복도는 컴파일러에 의해 결정되며, 외형적으로 보았을때 수행 과정상에 어떤 변화를 요구하지도 않는다. 중복 필요성이 나타나는 대표적인 경우로 recursion을 생각할 수 있지만, 일반화시켜서 본다면 컴파일러가 판단한 모든 경우에 이것을 적용시킬 수 있다. 기본적으로 특정 함수를 중복 저장하는 범용 프로세서들은 환형 리스트 혹은 선형 리스트 구조로 연결된다. 일단 컴파일러가 중복도를 결정하고 대상 범용 프로세서를 선정하여 실제 주소를 중복 지정한다는 것을 가정한다. 이렇게 선정된 범용 프로세서들은 위상에 따라 배열되어 리스트 형태를 구성하는데, 리스트의 맨앞에 위치하는 범용 프로세서가 주 범용 프로세서(Master GP)가 되며, 나머지 범용 프로세서는 종 범용 프로세서(Slave GP)로 지정된다. 특히 선형 리스트의 경우는 리스트의 마지막 범용 프로세서를 단말 범용 프로세서(Terminal GP)라고 부른다. 리스트 구조가 의미하는 것은 해당 범용 프로세서로부터의 반복적 수행 요구가 우선적으로 다음 범용 프로세서로 옮겨진다는 것이다. 그러나 선형 리스트에서의 단말 범용 프로세서는 참조할 범용 프로세서를 할당받지 못하기 때문에, 반복적인 수행 요구라고 하더라도 자기 자신이 처리하는 것을 원칙으로 하며, 이것은 코드의 중복없이 recursion을 처리하는 것과 같은 효과를 가진다. 만약 이 경우 똑같은 함수에 대해 지나치게 많은 수행 요구가 발생하여 수용 불능 상태에 빠지면, 다음에 설명하는 부하 이동 방법에 따라 적절한 범용 프로세서를 선택하여 코드를 이동시킨다. 따라서 일정 길이 혹은 합리적인 최대 길이를 가지는 recursion의 경우나 recursion이 아니더라도 빈번히 수행되어야 할 함수는 그 예상 빈도에 따라 적절한 정도로 중복될 수 있으며, 해당 함수의 성격에 따라 선형이나 환형 여부를 결정할 수 있다. 덧붙여서 전단 범용 프로세서(Front GP)를 정의할 수 있는데, 이것은 전단 컴퓨터에서 직접 수행 요구를 받는 범용 프로세서로, 전용 컴퓨터내에서 수행되는 함수의 메인 블럭을 가진다.

이제 범용 프로세서 내에서 함수를 수행하는 일반적인 시나리오를 설명한다. 기

본적으로 그래프 관리자에 함수 적용 패킷이 전달되면서 해당 함수에 대한 수행이 개시된다고 볼 수 있다. 이것은 요구큐에서 대기하다가 그래프 관리자에 의해 처리된다. 즉 함수 수행표나 함수 문맥표 등의 정보를 생성하고 수행 메모리로 코드를 복사하며, 스택등의 자원도 할당한다. 이후에 해당 함수가 기본 프로세서에 의해 선택되면 과괴적 수행 방식에 따라 결과를 산출하는데, 다른 함수로부터의 결과나 특정 주소의 데이타를 요구할 경우 대기 상태로 변환되었다가, 결과값 혹은 데이타를 받음으로써 계속 수행될 수도 있다. 기본 프로세서에 의해 생성된 결과는 패킷 형태로 생성되어서 그래프 관리자에게 전달되고 이것은 결과큐에서 대기하였다가 적절한 곳으로 전송된다. 범용 프로세서의 동작과정은 그림 3.12와 같다.

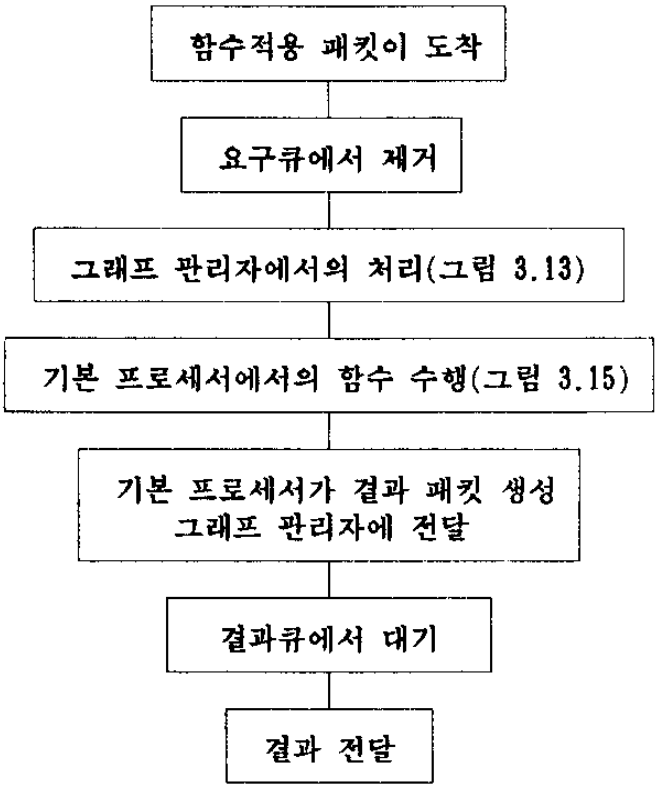


그림 3.12: 범용 프로세서의 동작 과정.

그래프 관리자는 요구큐와 결과큐를 관리하면서 통신 프로세서와의 인터페이스를 담당하고, 기본 프로세서가 함수를 수행하는데 필요한 환경을 조성한다. 그림 3.13은 그래프 관리자의 동작 과정을 도표화한 것이다. 먼저 그래프 관리자는 요구큐와 결과큐를 검색하면서 각 패킷들을 처리한다. 이 때 결과큐를 요구큐보다 우선 처리함으로써 시스템의 안정성을 도모한다. 즉 결과큐가 비어있을 때만 요구큐를 처리한다는 것을 의미한다.

코드 패킷의 경우 그래프 관리자는 해당 패킷을 정의 메모리내의 지정된 위치로 복사하며 함수 위치표상에 관련된 정보를 기록한다.

그래프 관리자가 함수 적용 패킷을 선택하면 일단 함수 수행표상에서 그 함수의 수행 여부를 확인해야 하는데, 이것은 그래프 리덕션 방식을 지원하기 위한 배려이다. 즉 이미 수행된 경우는 함수 수행표상의 결과값으로부터 결과 패킷을 생성하여 이것을 결과큐에 삽입하며, 현재 수행중인 경우는 함수 문맥표상의 반환 주소를 추가함으로써 처리를 마친다. 반면 아직 수행요구가 없었던 경우에 대해서는 목적 주소내의 범용 프로세서 번호를 확인하게 되는데, 자기 자신이거나 외부의 범용 프로세서 혹은 미확정된 경우로 구별된다. 첫째, 외부의 범용 프로세서로 지정된 경우는 상위 통신 프로세서로 이 패킷을 전달한다. 둘째, 미 확정된 경우는 과도한 부하에 의한 다른 범용 프로세서로의 작업 이동을 의미하는데, 목적 주소가 NULL인 요구 패킷을 발생시켜 이것으로부터 부하가 작은 범용 프로세서를 찾아내야 한다. 따라서 요구 패킷을 상위 통신 프로세서로 전달하고 함수 적용 패킷에 대해서는 대기 표시를 한다. 세번째로 자신이 속한 범용 프로세서내에서 수행할 수 있는 경우는 함수 수행표와 함수 문맥표상의 엔트리를 할당하고, 정의 메모리 혹은 함수적용 패킷의 코드 부분으로부터 수행 메모리로 코드를 복사한다. 또한 정의 메모리상에 스택 공간을 할당하고 몇가지 시스템 변수들을 초기화한 후 해당 함수에 대해 H-상태를 할당하는데, 이 함수는 적절한 시간에 기본 프로세서에 의해 수행된다.

그래프 관리자가 처리해야 할 패킷이 요구 패킷인 경우는 해당 주소에서 데이타를 획득하고 이것을 결과 패킷으로 생성하여 결과큐에 삽입한다.

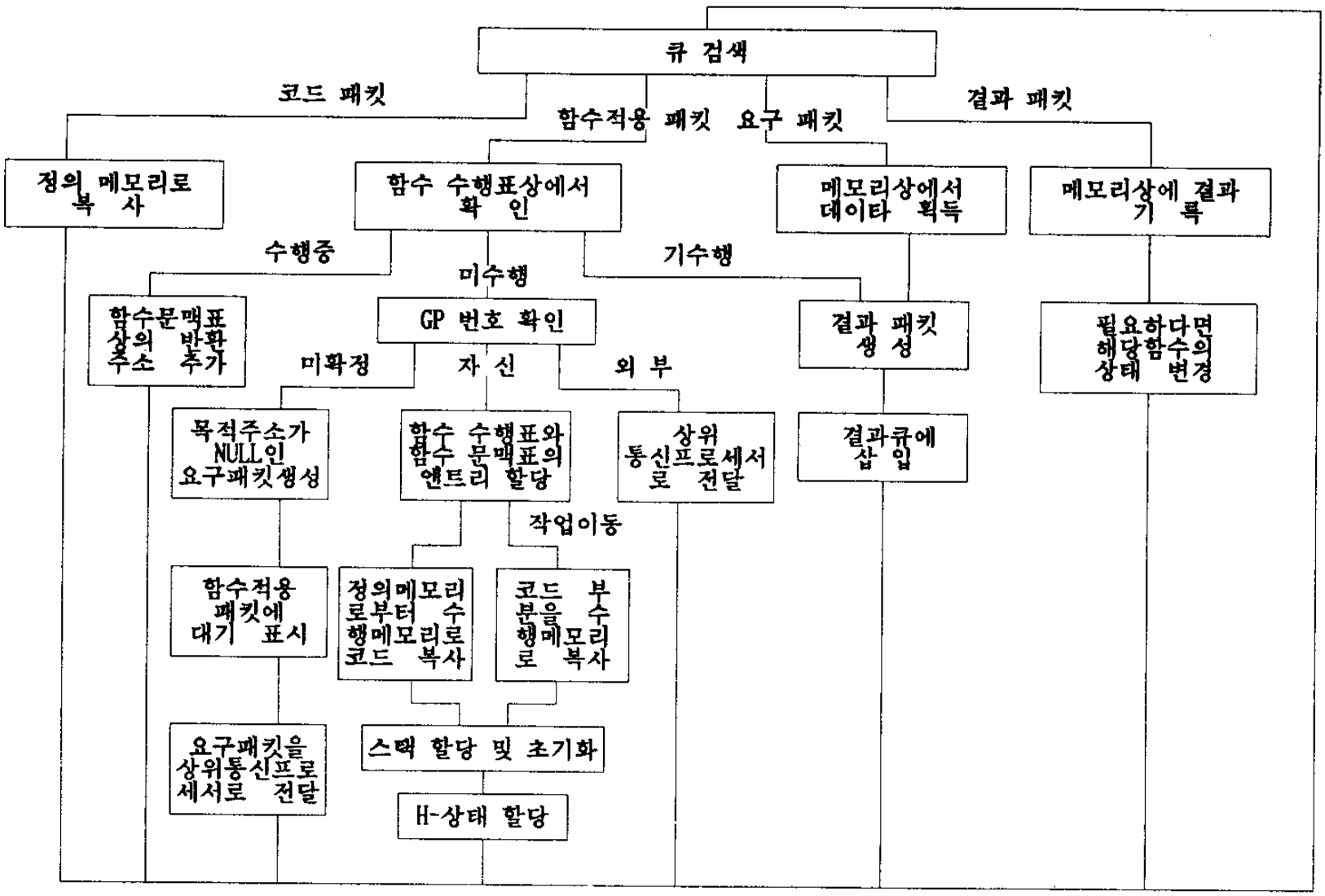


그림 3.13: 그래프 관리자의 동작 과정.

결과 패킷에 대해서는 메모리상에 결과값을 기록하고, 필요하다면, 즉 해당 함수가 기다리고 있는 결과값 혹은 데이타가 모두 도착했다면, 수행 상태를 C-상태로 변경한다. 물론 함수적용 패킷내의 범용 프로세서 번호가 미확정된 경우에 대한 결과, 즉 할당된 범용 프로세서의 번호도 결과 패킷에 의해 처리된다.

기본 프로세서는 기본적으로 스택 머신의 성격을 가진다. 각 함수에 대해 각각의 스택이 할당되며 이 스택위에서 대부분의 수행 과정을 처리한다. 즉 하나의 명령어를 수행한다는 것은 수신 주소로부터 데이타를 가져와서, 해당 연산을 수행한 후 송신 주소에 전달한다는 것을 의미한다. 따라서 그래프의 각 노드는 계산되었는지의 여부를 기록해야 하므로 일종의 태그가 필요하다. 그림 3.14는 그 예이다.

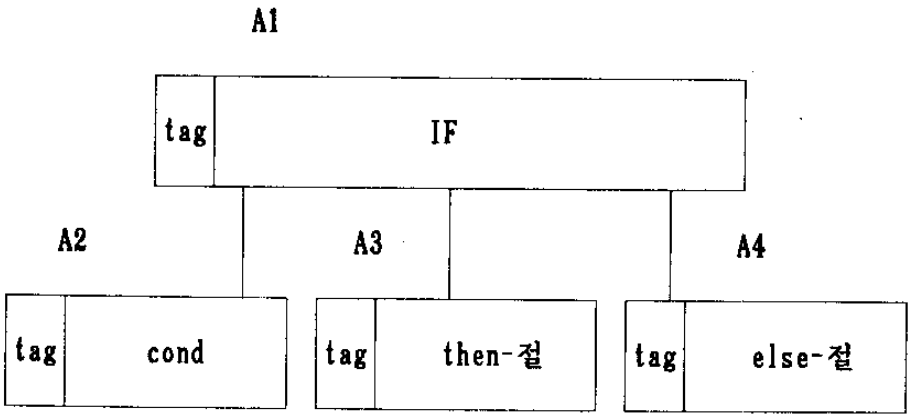


그림 3.14: 그래프의 예.

A1 노드는 'IF' 명령어로 이것을 계산하는 과정을 설명함으로써 수행에 따르는 메카니즘을 제시한다. 우선 A2 노드의 계산 여부를 확인하여 만약 계산되지 않았으면 A1의 주소를 스택에 넣고 A2 노드를 계산한다. A2 노드의 계산이 끝나면 A1의 주소를 스택에서 가져오는데, A2 노드의 값에 따라 참이면 A3 노드에 대해, 거짓이면 A4 노드에 대해 A2 노드와 같은 과정을 반복한다. 이제 다시 A1의 주소를 획득하고 A3 노드의 값이나 A4 노드의 값을 송신 주소로 전달한다. 따라서 스택은 수행 과정에서 필요한 주소들을 관리하는데 사용된다.

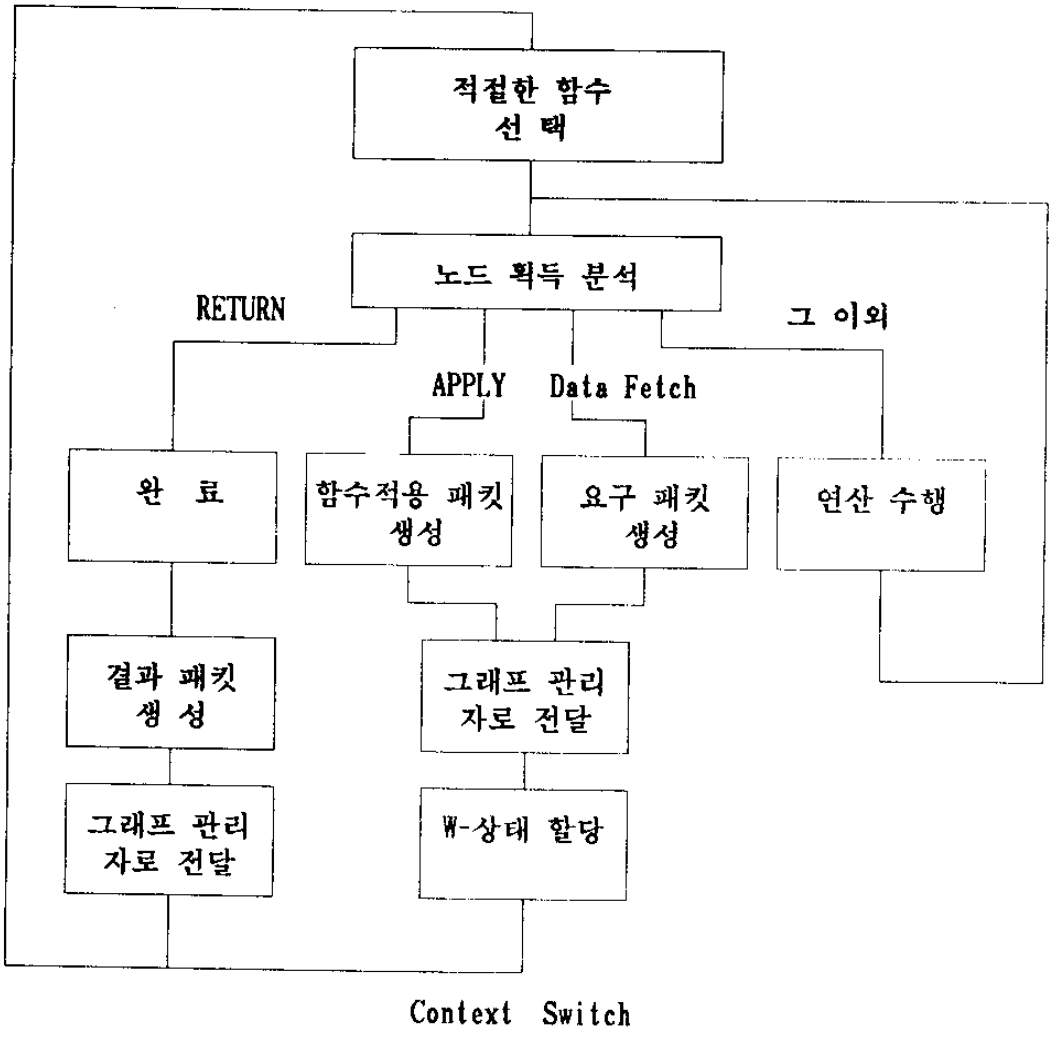


그림 3.15: 기본 프로세서의 함수 수행 과정.

그림 3.15는 기본 프로세서가 함수를 수행하는 과정을 보여준다. 그래프 관리자가 관리하는 함수 문맥표에 따라 C-상태나 H-상태의 함수중에서 적절한 함수를 선택하고 각 노드들을 따라가면서 수행한다. 이 때 노드의 명령어가 'RETURN'인 경우 해당 함수의 수행을 완료하고 결과 패킷을 생성하여 그래프 관리자로 전달한다. 노드의 명령어가 'APPLY'나 특정 주소의 데이타 획득에 관련된 경우 함수 적용

패킷이나 요구 패킷을 생성하여 그래프 관리자로 전달하며, 해당 함수에 대해서는 W-상태를 할당한다. 그 이외의 명령어를 포함하는 노드에 대해서는 위에서 설명한 바와 같이 수행한다. 또한 현재 함수의 수행을 마쳤거나 W-상태로 변환된 경우에는 다른 함수로 문맥 교환한다.

기본 프로세서가 각 노드를 수행하면서 주의해야 되는 점중의 하나는 predemand에 대한 처리이다. 함수 블럭내의 함수에 대한 predemand나 함수내의 인자에 대한 predemand는 해당 함수에 대한 함수 적용 패킷을 생성하는 것과 동시에, predemand된 함수에 대한 함수적용 패킷과 predemand된 인자에 대한 요구 패킷을 생성해야 한다. 이것들은 똑같이 요구큐에 넣어지게 되므로 원하는 효과를 거둘 수 있다.

또한 기본 프로세서는 함수들을 수행하기 위한 환경으로 최소한의 자료 구조를 유지해야 한다. 현재 처리중인 노드에 대한 주소 즉 프로그램 카운터와 현재 사용되고 있는 스택에 대한 스택 포인터, 스택의 상/하한계에 대한 정보를 보유해야 하며, 각 수행 상태별로 함수 문맥표상의 엔트리에 대한 포인터를 가져야 한다.

범용 프로세서 내부의 메모리는 정의 메모리와 수행 메모리로 구분된다. 정의 메모리는 컴파일러에 의해 초기에 할당된 코드 블럭들을 보관하며, 그래프 관리자가 사용하는 시스템표 및 큐를 유지한다. 반면 수행 메모리는 파괴적 수행 방식을 지원하기 위한 코드들을 기억하는데, 이것은 정의 메모리로부터 복사된다. 특이한 점은 그림 3.8과 같이 정의 메모리는 그래프 관리자와 기본 프로세서에 의해 공유되는 반면, 수행 메모리는 기본 프로세서에 의해 배타적으로 사용된다는 것이다. 이것은 메모리 엑세스에 따르는 충돌을 줄이기 위해 동적인 성질을 갖는 부분들을 적절히 배분한 결과이다. 특히 수행 메모리는 그래프 형태를 기억하기 위한 노드 구조를 지원할 수 있도록 설계되어야 한다.

3.2.6 통신 프로세서

통신 프로세서는 하위의 범용 프로세서나 통신 프로세서의 부하 정보를 유지하면서 부하 이동에 관련된 작업들을 전담한다. 즉 하위 프로세서들의 부하가 어떤

수준인지 기억하고 부하 이동의 필요성이 발견되면 어떤 범용 프로세서에 부하를 전달할 지 결정하며, 실제적으로 부하가 이동하는 경우에는 그 통로로써의 역할을 수행한다. 또한 프로세서 할당자에 의해 이미 특정 범용 프로세서가 할당되었다고 하더라도 해당 범용 프로세서가 그 함수를 수행할 수 없을 경우, 부하 균형 방법에 의거해 동일한 방법으로 작업을 이동시킨다.

통신 프로세서는 그림 3.9와 같은 구조를 가진다. 특히 배치자는 6개의 링크를 가지는데, 상위의 통신 프로세서 혹은 루트 프로세서와의 접속을 위한 하나의 링크, 하위의 범용 프로세서나 통신 프로세서를 위한 두 개의 링크, 인접 통신 프로세서와의 두 개의 X-링크와 기본 프로세서와의 연결을 위한 전용 링크가 그것이다. 배치 메모리는 하위 프로세서의 수행 제어에 필요한 정보들을 기억하기 위한 작은 규모의 메모리이다.

먼저 부하의 수준을 결정하는 기준에 대해 설명한다. 부하의 정도를 표시하는 기준값에는 허용 한계값, 일반 상한값, 일반 하한값이 있다. 허용 한계값은 해당 범용 프로세서가 작업을 받아 들일 수 있는 지의 여부를 결정하는데 사용되는 값으로, 수행 메모리의 할당율에 의거하여 결정된다. 일반 상한값과 일반 하한값은 수행 메모리의 할당율과 더불어 요구큐의 유효 부하 길이도 고려하는데, 이것들이 부하 이동 여부를 결정하는 척도로 사용된다. 유효 부하의 길이는 요구큐의 총 길이에 대해 프로세서 할당자에 의한 요구의 갯수와 수행 메모리내에 저장되어 수행중인 요구의 갯수를 제외한 값으로 계산된다.

따라서 범용 프로세서가 일반 상한값을 넘어서는 부하를 보유하고 있는 경우 일반 하한값 이하의 부하를 가지는 범용 프로세서로 작업을 이동시키게 된다. 그러므로 각 통신 프로세서들은 하위의 프로세서들의 부하가 어떤 영역에 해당하는지를 기록해야 하며, 그것은 배치 메모리에 저장된다.

또한 각 프로세서들간의 X-링크는, 코드나 그 이외의 패킷이 이동할 경우 전송 거리를 단축시키고 루트 부근에서 흔히 발생할 수 있는 병목 현상을 제거하는 목적으로 사용된다. 우선 X-링크를 통한 전송을 효과적으로 제어하는데 필요한 개념을

정의한다.

[정의] 두개의 범용 프로세서의 번호가 각각 N_1, N_2일때 해당 범용 프로세서들이 다음과 같은 식을 만족시키면, 두 범용 프로세서를 "깊이 m으로 인접하다"라고 표현한다.

$$| N_1 \text{ div } 2^{m-1} - N_2 \text{ div } 2^{m-1} | = 1$$
$$\text{Min}(N_1, N_2) \text{ div } 2^{m-1} + 1 = \text{Max}(N_1, N_2) \text{ div } 2^{m-1}$$

따라서 두개의 범용 프로세서가 깊이 m으로 인접하면 두 범용 프로세서간의 패킷 전송은 (m−1)-번의 상향 전송, 한번의 수평 전송, 그리고 (m−1)-번의 하향 전송으로 구성된다. 이것은 X-링크를 사용하지 않는 경우 최소한 m-번의 전송이 필요하다는 사실과 비교될 수 있다. 그림 3.16은 이런 성질을 보여준다.

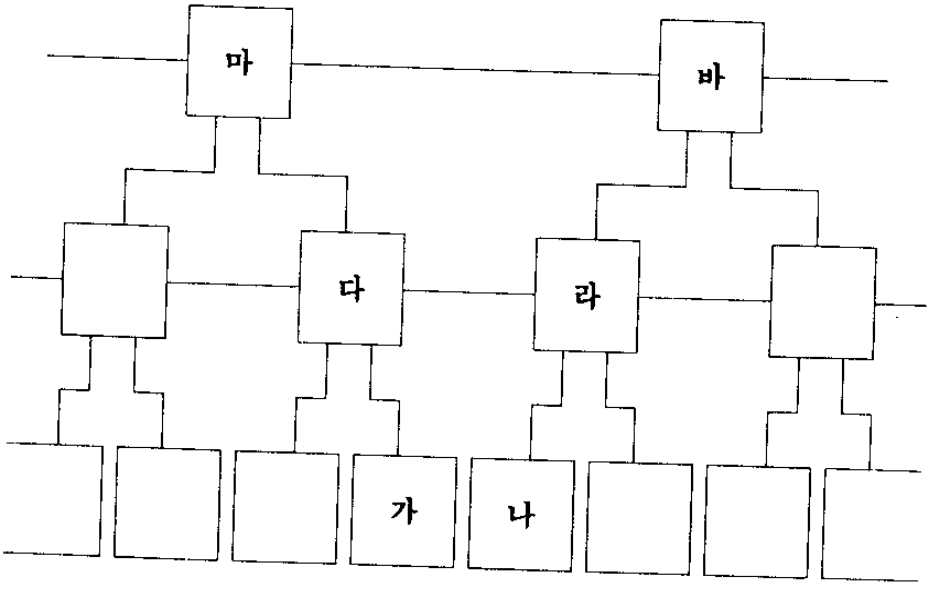


그림 3.16 X-링크의 사용 예.

범용 프로세서 '가'로부터 범용 프로세서 '나'로의 전송은 통신 프로세서 '다'와

통신 프로세서 '라'의 X-링크를 통해 이루어진다. 이것은 X-링크를 사용하지 않는 경우 최소한 통신 프로세서 '마'와 통신 프로세서 '바'를 거쳐야 된다는 점을 고려하면 상당한 양의 전송 길이를 감소시킨다. 그러므로 각 통신 프로세서는 목적 주소의 범용 프로세서 번호와 자신이 가지는 고유 번호와의 비교에 의해 X-링크를 사용할지의 여부를 결정해야 한다. 만약 인접하지 않다고 판단하면 상위의 통신 프로세서로 전달한다.

앞에서 언급한 바와 같이 단말 범용 프로세서로부터의 반복적인 함수 적용을 포함하는 함수수행 요구가 해당 범용 프로세서에서 수용되지 못할 경우 적절한 범용 프로세서로 수행 요구가 이동되어야 하는데, 이것은 통상의 수행 요구와 목적 범용 프로세서의 번호가 할당되지 않는다는 점에서 다르다. 즉 앞에서 설명한 것과 같이 범용 프로세서내의 그래프 관리자에 의해 적절한 대상 범용 프로세서를 찾기 위한 요구 패킷이 발생되고, 이것들이 통신 프로세서를 통해 상위로 전달된다. 이제 통신 프로세서가 이 요구 패킷을 받고, 하위의 통신 프로세서나 범용 프로세서에서 전달된 부하값에 따라, 여유가 있을 경우에는 하위의 해당 프로세서로, 그렇지 않는 경우는 계속해서 상위의 통신 프로세서로 패킷을 전달한다. 특히 하위 프로세서로 패킷이 전달되는 경우에 대해서는 요구 주소와의 주소 차이를 크게 하지 않는 쪽의 하위 프로세서를 우선적으로 선택하는데, 이것은 X-링크를 사용하는 통신 경로를 효과적으로 이용하기 위함이다. 이것을 반복하면서 부하가 이동될 범용 프로세서의 번호가 결정되면, 이것에 관한 정보를 결과 패킷으로 만들어 최단 거리의 전송 통로, 즉 X-링크를 사용하는 전송 통로를 따라 전송한다. 이것을 수신한 범용 프로세서는 범용 프로세서의 번호가 주어진 경우와 같은 방식으로 패킷을 전달하게 된다.

3.2.7 루트 프로세서

루트 프로세서는 전단 컴퓨터와의 인터페이스를 통한 통신을 담당하며 하위 두 개의 통신 프로세서의 부하 정보를 유지한다. 즉 하위 통신 프로세서로 전단 컴퓨터로부터의 코드 블럭을 전달하고, 각 범용 프로세서로부터의 계산 결과를 전단 컴퓨터에 넘겨준다. 또한 통신 프로세서와 마찬가지로 하위의 프로세서에 대한 부하

정보를 이용하여 부하 이동 경로를 제시하지만, 스스로가 부하 이동 경로에 포함되지는 않는다.

루트 프로세서는 그림 3.10과 같이 구성된다. 배치 메모리는 통신 프로세서의 그것과 동일하다. 배치자에는 5개의 링크가 연결되는데, 3개의 양방향 링크와 2개의 단방향 링크로 구별된다. 즉 하위의 프로세서에 연결시키기 위한 2개의 양방향 링크와 기본 프로세서와의 연결을 위한 하나의 양방향 링크, 그리고 전단 컴퓨터와의 통신을 위한 2개의 단방향 링크로 구성된다.

전체적인 시스템의 관점에서 본다면 워크 스테이션의 컴파일러에 의해 생성된 코드 블럭들은 시스템내의 통신 규약에 따라 적절한 헤더를 붙여서 전단 컴퓨터에 넘겨지고, 전단 컴퓨터는 전용 컴퓨터가 요구하는 형식의 패킷 형태로 변형하면서, 전단 컴퓨터와 루트 프로세서 사이의 인터페이스가 요구하는 헤더를 부가시켜 전달한다. 따라서 루트 프로세서는 전단 컴퓨터에서 전달된 패킷으로부터 헤더를 제거하는 외에는 전혀 변형을 가하지 않고 하위 통신 프로세서로 전달한다. 또한 전단 범용 프로세서로부터의 결과에 대해서는 전단 컴퓨터와의 인터페이스에 따라 적절한 헤더를 부가시켜야 하며, 이것은 전단 컴퓨터내에서 해석되어 시스템내의 통신 규약에 따라 워크 스테이션에 전달된다.

또한 루트 프로세서는 하위의 프로세서의 부하 관리 기능에 따라 발생된 요구 패킷을 받고 다른 프로세서로 그것이 전달되도록 지시하며, 모든 프로세서가 과부하일 경우 해당 함수의 수행을 연기하도록 지시해야 한다. 만약 부하를 이동시킬 경우에는 하위의 X-링크에 의해 그 통로를 설정하므로 자신은 부하를 이동시키는 부담에서 벗어날 수 있다.

3.2.8 전단 컴퓨터와의 인터페이스

루트 프로세서와 전단 컴퓨터를 연결시켜주는 인터페이스는 아주 간단한 논리적 관점을 제공하여 소프트웨어적으로 쉽게 처리할 수 있도록 지원할 뿐만 아니라 전용 컴퓨터를 통해 유지되는 패킷 전달의 규약들에 일관성을 부여한다. 전단 컴퓨터

와 루트 프로세서간의 인터페이스 구성도는 그림 3.17과 같다.

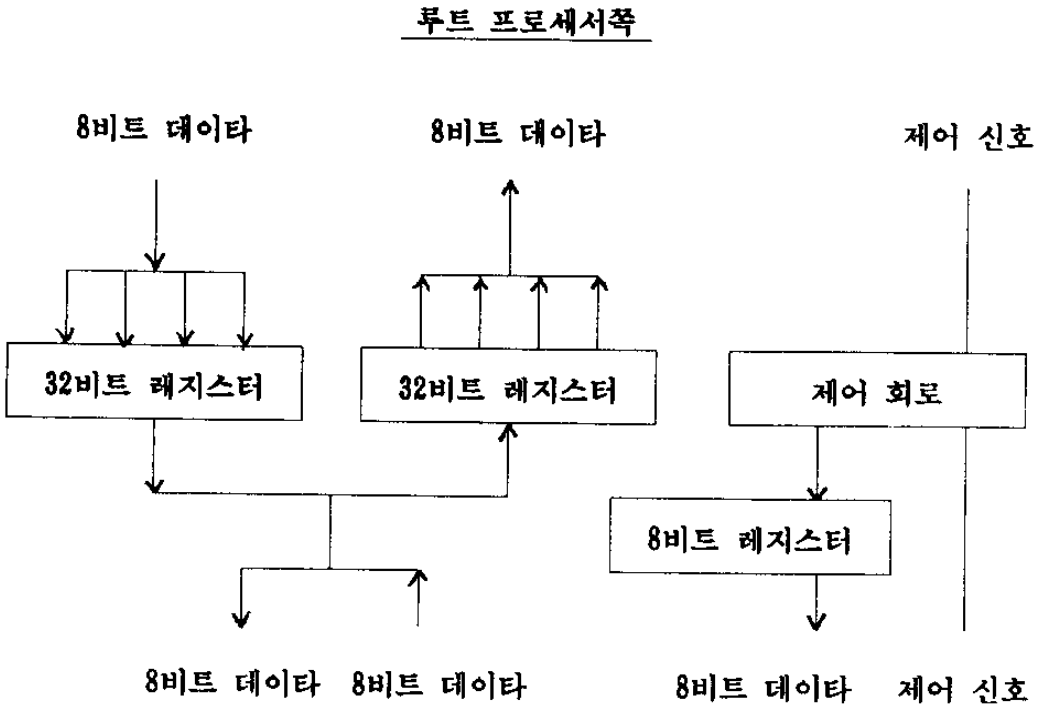


그림 3.17: 전용 컴퓨터와 루트 프로세서간의 인터페이스.

전단 컴퓨터쪽의 인터페이스는 시스템 버스를 엑세스하여 작동하며, 3개의 I/O mapped address에 의해 수신 레지스터, 송신 레지스터, 상태 레지스터를 정의한다. 수신 레지스터는 루트 프로세서로부터의 16비트 데이타를 저장하고, 송신 레지스터는 전단 컴퓨터에서 생성된 16비트 데이타를 저장한다. 상태 레지스터는 두 비트로 구성되는데, 하나는 수신 요구 신호로 사용되고 나머지 하나는 송신 준비 신호로 사용된다.

수신 요구 신호가 1이라는 것은 루트 프로세서로부터 데이타가 수신 레지스터에 보내져서 사용될 수 있다는 뜻이고, 송신 준비 신호가 1이라는 것은 송신 레지

스터가 비어 있으므로 데이타를 넣을 수 있다는 뜻이다. 전단 컴퓨터 입장에서 보면 수신 레지스터는 읽기 용도로, 송신 레지스터는 쓰기 용도로, 상태 레지스터는 읽기 용도로만 사용된다.

루트 프로세스쪽에서 본 인터페이스는 쓰기를 위한 8 비트 포트와 읽기를 위한 8 비트 포트가 있어 전이중(full duplex) 방식의 통신을 구현한다. 각 포트에 대해 hand shaking을 위한 두 개의 제어 신호와 동기를 위한 클럭 신호가 필요하다.

전단 컴퓨터가 통신 정보를 전용 컴퓨터에 적합하도록 처리하는 작업이 소프트웨어적으로 수행되어야 하고, 인터페이스와의 작업에 대한 범용성이 유지되어야 하기 때문에, 인터페이스와의 데이타 전달은 상태 레지스터를 검색(poll)하는 방식을 취한다. 그러나 루트 프로세서쪽에서는 속도를 높여 통신에 의한 시간 지연을 줄여야 할 뿐만 아니라 패킷 형태가 전단 컴퓨터에서 대부분 완성되므로 데이타 전달 방식이 하드웨어적으로 이루어 진다.

전단 컴퓨터에서 루트 프로세서로 전달되는 코드 패킷은 가변적인 길이를 가지므로 루트 프로세서도 이 패킷의 길이를 알 필요가 있다. 따라서 패킷의 헤더에 그 길이를 포함시켜 전달한다.

헤더의 길이는 32 비트이며 모든 전송의 단위는 32 비트로 고정된다. 즉 인터페이스에 대한 한번의 읽기나 쓰기 작업을 수행함에 있어서, 전단 컴퓨터는 16 비트 크기 단위로 두 번에 나누어 처리하며, 루트 프로세서에서는 8 비트 크기로 네 번에 걸쳐서 처리된다.

루트 프로세서나 전단 컴퓨터는 헤더를 받고 '패킷 길이' 필드로부터 패킷의 나머지 부분의 길이를 계산하여 헤더를 받을 때와 동일한 작업을 반복한다. 전단 컴퓨터에서는 코드 패킷 형태로 정보를 구성하고 워드 카운터를 유지하면서 상태 레지스터 내용에 따라 데이타를 쓰거나 읽고, 루트 프로세서에서는 헤더 패킷의 워드 길이에 따라 워드 카운트를 유지하면서 해당 길이의 데이타를 읽거나 쓴다.

인터페이스 하드웨어는 읽기와 쓰기를 위해 각각 32 비트의 버퍼와 상태 레지

스터를 위한 8 비트 버퍼, 그리고 이것들을 제어하기 위한 제어 논리회로로 구성된다.

3.2.9 시스템 구현을 위한 문제 제기

첫째, 각 프로세서가 동작하기 위한 최소한의 커널이 정의되어야 한다. 범용 프로세서에 대해서는 적절한 함수를 선택하고 그 수행 과정을 제어할 프로그램이 커널 형태로 구성되어야 하며, 앞서서 언급한 시스템 표가 커널 자료 구조로 정의되어야 한다. 이 외에 수행해야 할 여러 함수들에 대해 적절한 함수를 선택할 수 있도록 스케줄러의 기능도 보강되어야 한다. 또한 통신 프로세서나 루트 프로세서에 대해서도 수행해야 할 역할들과 일치하는 제어 프로그램이 상주할 수 있도록 고안되어야 한다.

둘째, 범용 프로세서나 통신 프로세서 혹은 루트 프로세서에서 사용하는 메모리에 대해 그 구조가 정의되어야 한다. 범용 프로세서의 수행 메모리는 그래프 리덕션 방식을 지원하기 위한 태그 필드등을 포함하고 가변 길이의 노드를 만족시킬 수 있도록 구성되어야 한다. 또한 통신 프로세서나 루트 프로세서가 사용하는 배치 메모리의 경우는 부하 정보를 유지하는 적절한 구조가 정의되어야 한다.

세째, 범용 프로세서가 프로그램을 수행하면서 발생되는 비정상적인 상태, 즉 수행 메모리등의 자원을 더 이상 할당할 수 없는 경우등에 대한 하드웨어적인 처리 방법이 제시되어야 한다. 전체 시스템에 대한 과부하로 더 이상 프로그램을 수행할 수 없을 경우 적절한 함수를 선택하여 그것으로부터 자원을 되돌려받는 방법도 고안되어야 한다. 또한 특정 프로세서의 결함(fault)에 의한 시스템의 비정상적인 작동을 방지하기 위해 해당 프로세서를 시스템내에서 배제하는 방법도 고려해야 한다.

네째, 범용 프로세서내의 수행 메모리에 대한 가비지 수집 전략이 수립되어야 한다. 기본적으로 수행과정중에 가비지가 수집되는 것을 원칙으로 하지만, 각 함수에 대한 수행을 완료하면서 전체적으로 가비지를 수집할 수 있도록 설계해야 한다.

다섯째, 전용 컴퓨터내의 상호 연결망에서 패킷이 전송되는 방법에 대해 정의해

야 한다. 구현을 위해서는 고정 길이의 패킷들로 분할해야 하며 이들의 순서를 제어하기 위해 적절한 헤더를 부착해야 한다. 또한 이러한 헤더를 다루는 논리가 설계되어야 한다.

3.3. 시스템 성능 분석

설계된 Math. Package 전용 컴퓨터의 성능을 정확하게 예측하여 평가한다는 것은 수행 시나리오의 동적 작업 배치 성질로 인하여 거의 불가능하다. 그러나 여기에서는 유사 시스템과의 성능비교 및 예제 프로그램에 의한 성능분석을 통하여 제안된 Math. Package 전용 컴퓨터의 성능을 예측하여 보겠다.

3.3.1. 유사 시스템과의 성능 비교

본 시스템의 상호연결망으로 사용된 X-Tree는 MIT 정적 데이타 플로우 컴퓨터[DENN82]에 있어서 distributer와 arbiter를 묶어 놓은 위치에 해당된다. 이것은, 연속적으로 수행될 두 계산간의 데이타 이동 거리가 현저하게 단축될 수 있음을 의미하는 것으로서, 이러한 장점은 하나의 함수를 하나의 코드블럭으로 번역하여 순차처리함으로써 얻어지는 통신부하의 경감에서 잘 드러난다. AMPS[KELL79]의 약점이라 할 수 있는 Tree의 루트부근에서의 병목현상은 X-Tree의 X-링크를 사용함으로써 치명적인 순간을 벗어날 가능성을 크게 해준다. 사용언어에서 option으로 주어진 프로세서 할당자는 프로그램 작성자의 전문 예상 지식을 이용할 수 있다는 장점을 제공한다는 면에서 AMPS보다 프로그램 수행방식의 융통성이 더 주어진다.

3.3.2. 예제 프로그램에 의한 성능 분석

한 개의 프로세서만을 사용하는 기존의 폰노이만 컴퓨터 상에서 임의의 math. package를 수행하는 시간을 1로 볼때, p개의 범용 프로세서(GP_1, GP_2, ..., GP_p)를 가지고 구성되는 Math. Package 전용 컴퓨터 상에서 이것을 수행하면 최선의 경우 1/p의 수행시간만이 필요해진다 (이 때 Math. Package 전용 컴퓨터가 X-Tree임을 감안하면 p는 2의 지수승인 2^k (k = 1,2,3,...)을 값으로 갖게 된다). 그러나 실제로는, 1보다는 작지만 1/p보다는 큰 수행시간이 필요해진다. 이와 같이 프로세서의 추가에 따른 성능향상도를 기대치보다 낮추는 요소로서 다음과 같은 사항을 생

각할 수 있다.

첫째, 주어진 문제(math. package)의 성격상 병렬처리성이 풍부하지 못할 경우 성능향상도는 기대치보다 못하게 된다. 예를 들어 Newton Raphson 방법과 같은 반복대입법(Repeated Substitution Method)의 경우, 이전의 상태에 따라 현재의 상태가 결정되어 진행될 수 있으므로 이런 형태의 math. package는 병렬처리성이 뛰어나지 못하다. 여기에서 유념할 사항은 병렬처리의 granularity가 본 Math. Package 전용 컴퓨터에서는 user-defined innnermost function 수준이라는 점이다. 만일 granularity가 산술연산 정도로 작다면 반복대입법 역시 상당한 병렬처리성을 가지고 있다고 볼 수 있지만 본 Math. Package 전용 컴퓨터에서는 통신부하면을 고려하여 다소 큰(medium/coarse) granularity를 취한다. 물론 통신부하를 감소하더라도 병렬처리성을 높이는데 주안점을 두겠다고 한다면 지원해 줄 수는 있다. 또한 문제의 성격상 병렬처리성이 다소 뛰어나다고 해도 그 정도가 범용 프로세서의 갯수와 어떠한 대소관계가 있느냐에 따라 성능향상도는 크게 변한다. 예를 들어 어느 기간(period)에 동시 수행가능한 함수(OGF: One Grain Function: user-defined function)의 갯수(n)가 범용 프로세서의 갯수(p)보다 작은 경우 성능향상도는 기대치에 미치지 못한다. 반면에 n이 p이상인 경우, 성능향상도는 기대치에 수렴할 가능성이 커지므로 본 Math. Package 전용 컴퓨터에서는 부하균형 메카니즘에 의해 가급적 오랜 기간에 걸쳐 이 상태가 만족되도록 부하를 조절하게 된다.

둘째, 본 Math. Package 전용 컴퓨터가 요구신호 전송에 의한 리덕션 수행방식을 취함으로써 감수하는 overhead로 인하여 성능향상도가 기대치보다 못하게 된다. 이러한 overhead는 요구신호 전송과 결과값 전송에 따른 일종의 통신부하로서 크게 두 부류로 나눌 수 있다. 하나는 범용 프로세서 내(intra-GP)의 통신부하이고, 다른 하나는 범용 프로세서 간(inter-GP)의 통신부하이다.

Intra-GP 통신부하는 하나의 OGF를 범용 프로세서가 수행할 때 프로그램 카운터의 값에 코드블럭에 지정된 offset만큼을 가감시킴으로써 요구 구동형 수행방식을 지원해주는데 필요한 부하이다. 따라서 if 구문과 같이 선택적 매개변수 전달(selective parameter passing)에 의하여 불필요한 OGF를 수행하지 않을 수 있다는

장점을 제공하지만 기존의 폰노이만 수행방식도 조건부 분기를 통하여 이를 지원해 주기 때문에 결국 intra-GP 통신부하는 폰노이만 컴퓨터가 하나의 OGF를 수행하는데 필요한 부하의 거의 두 배가 된다고 볼 수 있다. 그러나 이것은 def로 정의된 OGF에 한정된 이야기이며 seq나 seq_def로 정의된 OGF에 대해서는 그렇게 심각한 부하가 되지않는다. 왜냐하면 seq나 seq_def로 정의된 OGF는 기존의 폰노이만 수행방식에 외거하여 수행되는 코드블럭으로 번역되기 때문이다. 이외에도 Intra-GP 통신부하 속에는 대기지연(queueing delay)이 포함되는데 이것은 주어진 math. package의 병렬도 및 시스템 상에서 이루어지는 부하균형상태와 밀접한 관계가 있다.

Inter_GP 통신부하는 동시병렬처리될 수 있는 OGF(f_1, f_2, ..., f_n)들이 각각 하나의 범용 프로세서에서 수행될 것을 지시하는 n개의 수행요구신호 전송 부하이외에 수행완료된 OGF들의 결과값이 이를 필요로 하는 범용 프로세서로 반환 전송되는데 필요한 부하를 합한 것을 이른다. Inter-GP 통신부하는 크게 데이타 전송용 패킷을 packing/unpacking하는 시간과 link를 거치는데 필요한 시간으로 나뉘게 되는데 predemand evaluation을 명시하지 않는 이상 demand forward와 result back에 의하여 이중의 통신부하가 필요하다.

이상의 사항을 고려하여 앞서 거론되었던 예제 프로그래밍 III (Solving n × n Linear System by Cramer's rule)을 본 Math. Package 전용 컴퓨터에서 수행할 경우 기존의 폰노이만 컴퓨터의 경우와 성능을 비교분석하면 다음과 같다. 먼저 성능분석을 간략하게 하기 위해서 Inter-GP 통신부하 중에서 packet을 packing/unpacking하는 시간은 없는 것으로 가정한다 (이 시간은 link에서 소요되는 시간에 비해서 아주 작다). n × n linear system을 하나의 프로세서로 구성된 폰노이만 컴퓨터에서 수행할 경우 소요되는 시간을 T_{VNM}, p개의 범용 프로세서로 구성된 Math. Package 전용 컴퓨터에서 수행할 경우 소요되는 시간을 T_{MPM}라 하자.

T_{VNM}은 함수 main을 수행하는 시간과 (n+1)개의 determinant 함수를 수행하는 시간의 합이다. 함수 determinant는 n에 대하여 대략 $n^3/3$ 정도의 복잡도(complexity)를 갖고 있으므로[MARO82] T_{VNM}은 다음과 같이 정리된다.

$$T_{VNM} = \{G_{main} + (n+1)\ G_{det}\}\ t_e$$
$$\simeq (n+1)\ (n^3/3)\ t_e$$
$$= \{n^3(n+1)/3\}\ t_e.$$

여기에서 G_{main}과 G_{det}는 각각 함수 main과 함수 determinant의 granularity(CPU 명령수)를 나타내며 t_e는 하나의 CPU 명령을 수행하는데 걸리는 시간이다.

T_{MPM}은 통신에 걸리는 시간(T_{comm})과 계산에 걸리는 시간(T_{comp})의 합으로 볼 수 있다. T_{comp}는 주어진 예제의 병렬처리성 때문에 함수 main을 수행하는 시간(T_{main})과 함수 determinant 1개를 수행하는 시간(T_{det})의 합만으로 표시된다. 반면에 T_{comm}는 최악의 경우 X-Tree의 최장 link path를 통과하는대 걸리는 시간과 (n-1)개의 요구신호가 전송되는데 걸리는 시간의 총합의 2배가 된다. (n-1)개의 요구신호 전송시간이라 함은 예제 프로그램이 1-master/multi-slave 형태의 병렬처리성을 제공하기 때문에 고려된 것이며, 2배라 함은 demand forward와 result back 때문에 생기는 요소이다. 하나의 패킷이 link 1개를 통과할 때 걸리는 통신시간을 t_c라 할 때 t_c는 t_e보다 상당히 큰 것이 상식이므로 이므로 여기에서는 r배 가량 크다고 가정하여 $t_c = r\ t_e$라고 두겠다. 이 때 T_{MPM}은 다음과 같이 정리된다.

$$T_{MPM} = T_{comp} + T_{comm}$$
$$= (T_{main} + T_{det}) + 2\ \{\ MAX\ T_{comm}(i\ in\ [1..n]) + (n-1)\ t_c)$$
$$= (G_{main} + G_{det})\ t_e + 2\ \{\ (2\ \log_2 n - 1) + (n-1)\}\ t_c$$
$$\simeq \{\ n^3/3 + (4\log_2 n + 2n - 4)r\}\ t_e.$$

여기에서 $r << n^3/3$이라고 가정한다면 Math. Package 전용 컴퓨터가 기존의 폰노이만 컴퓨터에 비해서 (n + 1)배의 성능향상을 가져옴을 보여 주는데 만일 (n +1)의 값이 p보다 클 경우는 p배의 성능향상을 가져 오게 된다. 이것은 제안된 Math. Package 전용 컴퓨터가 거의 이상적인 시스템임을 보여주는데 주된 원인은 예제 프로그램의 효율적인 병렬처리성에서 기인하며 부차적으로 $r << n^3/3$이라

는 상당히 유리한 가정에서 기인한다. 이 이야기는 math. package 프로그램의 병렬 처리성에 따라 성능이 다소 변할 수 있음을 암시하며, t_c가 t_e보다 상당히 큰 경우 성능이 떨어질 수 있음을 나타낸다. 그러나 math. package의 대부분이 상당한 병렬 처리성을 내포하고 있을 뿐만 아니라 기술상의 발전으로 $r << n^3/3$의 가정을 만족하리만큼 r이 점점 작아지고 있으므로 설계된 Math. Package 전용 컴퓨터에 의하여 위의 예제가 수행될 경우, 기존의 폰노이만 컴퓨터보다 p 배 정도 빠르다고 할 수 있다.

4. 고속의 Divide-and-Conquer 알고리즘 전용 컴퓨터

Divide-And-Conquer (DAC) 알고리즘을 효과적으로 수행하면서 일반적인 다른 알고리즘도 무리없이 수행할 수 있도록 제안된 고속의 다중처리 시스템인 HYPERDAC은 데이타 플로우 모델을 계산 모델로 채택하고, DAC 알고리즘의 논리적 계층 구조를 구조상에 효과적으로 반영하기 위하여 Hypertree를 상호연결망으로 채택하고, DAC 알고리즘의 성격상 문제에 따라 분할되는 부문제의 수 및 할당되는 자료의 양이 동적으로 결정되는 문제를 다루기 위해 Gradient 분산 부하 균형 정책을 사용하고, 제한된 자원하에서 자원을 효과적으로 이용하기 위하여 병렬성의 제어 기법도 사용한다.

본 연구에서는 HYPERDAC 시스템의 베이스 언어와 이를 토대로 한 프로그래밍 언어를 설계하고, 구조에 관한 개념 설계를 완성하고 성능을 분석한다.

4.1 병렬 계산에 적합한 구조화 자료의 조작 (Structure Handling for Parallel Processing)

데이타 플로우 모델은 병렬처리 시스템에 매우 적합한 것으로 알려져 있으나 Kuck을 비롯한 몇몇 학자들이 제기한 것처럼 그 모델의 함수성 및 개념적인 메모리의 부재로 인해 큰 구조화 자료의 조작에는 상당한 문제점을 안고 있다 [GAPA 82].

데이타 플로우 모델에서 각 계산단위 실행법칙의 요체는 데이타 종속관계에 기초한 데이타의 흐름에 있다. 즉, 계산단위는 해당 입력 아크(arc)상에 자료가 도착하면(이용 가능하면) 실행이 가능해지고, 실행후 그 결과 자료를 필요로 하는 모든 계산단위로 전송하므로써 계산을 수행해 나간다. 따라서 자료는 일단 한번 생성되면 수정되지 않으며, 각 계산단위는 실행에 필요한 모든 입력 데이타가 완전히 생성되어 이용이 가능할 때에만 실행할 수 있다. 그러므로 구조화 자료를 입력으로 하는 계산단위는 해당 자료 전체가 생성된 후에 실행이 가능하고 또한 구조화 자료 전체를 이동해야 하는 문제점이 있으며, 모델의 함수성 때문에 구조화 자료의 극히 일

부분만을 수정해도 새로이 구조자체를 생성해야 하는 문제점이 있다. 구조화 자료 전체를 한꺼번에 이동하는 문제는 통신상의 막대한 오버헤드(overhead)로 인해 병렬 수행을 통한 이득이 상실될 수 있기 때문에 실제로는 거의 대부분의 시스템이 구조화 메모리를 두고 각 구조화 자료는 그곳에 저장하고, 그것에 대한 포인터를 이동하는 기법이 사용된다.

여기서 주의해야 할 점은 구조화 자료를 입력으로 하는 계산단위에 구조화 자료에 대한 구조화 메모리 상의 포인터를 넘겨 주는 시점이다. 순수한 데이타 플로우 모델에서는 구조화 자료의 생성이 완성된 후에야 비로서 포인터를 넘겨 주도록 하고 있으나, 그럴 경우에 구조화 자료를 생성하는 계산단위와 그자료를 소비 하는 계산단위는 병렬수행이 불가능하게 되므로써 일반적으로 빈번한 구조 조작 연산을 요하는 많은 과학,공학 분야의 문제에 내재한 병렬성을 이용하지 못하게 되어 성능이 저하되는 문제점이 있다. 이러한 문제점은 구조화 자료 전체에 대해 동기화(synchronization)를 행하기 때문에 발생한다. 따라서 구조화자료 전체에 대해 동기화를 행하지 않고 구조화 자료의 각 원소(components)에 대해 동기화를 행하므로써 구조화 자료의 생산자와 소비자 간에 병렬수행을 통해 성능향상을 꾀할 수 있을 것이고, 그러한 관점에 기초하여 MIT 대학의 연구팀은 I-structure를 제시하였다 [ARTH 80]. 그러나 I-structure는 이러한 계산의 nonstrictness를 이용하여 병렬수행의 이득을 보고 있으나, 동일한 이유로 인한 제약점 또한 있다(accumulation 문제와 accumulation의 종료를 탐지하는 문제) [ARNP 86]. 특히 구조화 자료의 일부분을 수정해야 할 경우에 전 자료를 복사,생성해야 하는 문제점이 있다. 이점은 빈번한 복사를 통한 메모리의 낭비와 복사에 소요되는 처리시간의 오버헤드로 인해 문제시 되며, 흔히 완전한 복사, 생성 기법과 병렬성의 이용은 적으나 전반적인 성능향상을 위한 적절한 공유(sharing) 기법의 절충문제로 알려져 있다.

또한 일반적으로 다중처리 시스템에서 문제시되는 자료의 분산문제 (partitioning/allocation)도 효과적인 성능향상을 위해 고려되어져야 할 것이다.

본 절에서는 우선 병렬수행 환경하에서 구조화 자료의 효과적인 처리를 위한 요건 및 자료구조의 형태에 대해 살펴보고, 설정된 요건하에 기존의 여러 구조화

자료 조작 방법에 대해 분석을 행하고, 마지막으로 본 연구에서 제안한 구조화 자료 조작 기법을 논의한다.

4.1.1 구조화 자료 조작의 요건 (Requirements for the Structure Handling)

병렬수행 환경하에서 함수언어의 구현시에 구조화 자료의 효과적인 처리를 위한 요건은 다음과 같다.

(1) 구조화 자료 조작의 함수성 및 병렬성 유지

데이타 플로우 모델의 함수성을 약화시키거나 이용가능한 병렬성을 억제하는 구조화 자료의 수정은 방지되어야 한다.

(2) 성능

구조화 자료에 대한 접근(Access)이 신속해야 한다. 따라서 가능한 불필요한 계산을 수행하지 않고, 지역성을 이용하여야 한다.

(3) 하드웨어 오버헤드(Hardware Overhead)

채택한 구조화 자료 조작 기법은 하드웨어를 과도하게 복잡하게 만들어서는 안된다.

(4) 메모리 관리

채택한 구조화 자료 조작 기법이 과도하게 메모리를 요구하거나 메모리 관리를 복잡하게 만들어서는 안된다.

(5) 프로그래밍의 용이성

채택한 구조화 자료 조작의 semantics가 프로그래밍 작업과 더 나아가서는 컴파일 작업까지 복잡하게 만들어서는 안되며, 가능한 높은 수준에서 효과적인 구조화 조작을 표현할 수 있는 construct나 방법이 있으면 바람직할 것이다.

(6) 그외에 구조화 자료 조작의 병렬성을 효율적으로 이용하기 위한 자료의 적절한 partitioning/allocation 기법과 기호 처리(Symbolic Processing)를 빈번히 행하는 여러 응용분야에의 적용을 위해 효과적인 리스트(스트림)의 처리기법이 필요하다.

4.1.2 자료 구조의 종류

자료 구조는 일반적으로 다음과 같이 크게 3가지 형태로 분류할 수 있다.

(1) 스칼라(Scalar Values)

정수, 실수, 불린 값과 같이 값 자체가 데이타 플로우 그래프의 아크상에서 값 전체가 이동되는 단순한 값들이 이 범주에 속한다.

(2) 리스트/스트림(List/Stream)

일종의 구조화 자료로서 비교적 접근 형태가 순차적인 성격을 갖는다. 물론 구현 방법에 따라 별도의 메모리를 가질수도 갖지 않을 수도 있으며, 후자의 경우에 구조화 자료라고 말하기는 곤란한 점도 있다. 또한 리스트와 스트림은 엄밀하게 말해서 구분되어야 하나 nonstrict 리스트는 스트림도 포괄할 수 있기 때문에 편의상 같은 범주로 구분하였다.

(3) 임의 접근 구조(Random Access Data Structure)

구조화 자료의 임의의 원소를 동일한 시간내에 접근할 수 있는 구조화 자료로서 배열(Array)은 이범주에 속한다. 물론 리스트도 tuple 형태로 구현하는 경우에는 임의 접근 구조에 가깝다고 말할 수 있으나 논의의 편의상 위와 같이 구분하였다.

4.1.3 관련된 연구의 분석

함수언어의 구현에 있어서 구조화 자료를 다루는 문제는 단순하지 않아서 이를 효과적으로 처리할 수 있는 방법에 관한 연구가 지속적으로 연구되어 왔다. 그러나 아직까지도 만족할 만한 해결책이 발견되지 않은 실정이다.

본 절에서는 앞서 언급한 자료 구조중 배열과 리스트(특히 배열)를 효과적으로 표현하고 구현하는데 관련된 여러 연구에 대해 살펴보고자 한다 [ARNP 86].

(1) 배열(Array)

함수언어에서 배열을 효과적으로 사용하기 위해 제안된 방법들은 incremental 방법과 monolithic 방법으로 분류할 수 있다.

a. Incremental 방법

이 방법은 단어 자체가 의미하는 것처럼 배열의 내용을 점차적으로 증가, 변형시키는 방법으로 세가지의 연산을 사용한다.

i. newarray (n) :

n개의 원소가 어떤 default 값(nil)으로 초기화된 배열을 생성하는 연산이다.

ii. select(a,i) :

배열 a의 i번째 원소의 값을 반환하는 연산이다.

iii. update(a,i,x) :

새로운 배열(a')를 반환하는 연산이다. 여기에서 select(a',i) = x 이고, 만약 $i \neq j$ 이면 select(a',j) = select(a,j) 이다.

이 방법의 가장 큰 문제점은 update 연산시에 함수언어의 함수성을 유지시켜 주는 문제이다. 가장 단순한 방법은 update 연산마다 새로운 배열을 복사,생성하는 것인데 이 방법은 앞서 언급하였듯이 처리시간과 메모리 사용에 있어서 상당한 오버헤드를 초래하게 되어 성능이 저하되기 때문에 거의 사용하지 않는다.

전 구조를 생성하지 않고 함수성을 유지하는 한 방법으로는 Dennis가 제안한 Heap 구조를 사용하는 방법을 들 수 있다 [DENN 74]. 그러나 이 방법은 생성작업의 수행시에 배열에 관한 접근이 방지되고(즉 순차적으로 처리되어야 함) 또한 reference count를 유지해야 하는 문제점이 있다.

또 다른 방법으로는 변경되는 배열이 변경시에 그것에 대한 reference count가 1일 경우에 그 내용을 파괴적으로 변경시키는(in place update) 방법이 있다. 이 방법은 불필요한 메모리의 낭비를 막고 처리시간의 감소를 꾀할 수 있으나, 과연 그러한 파괴적 변경이 가능한 가를 검토해야 하는 문제점이 있다(이러한 파괴적 변경이 가능한 배열의 성질을 single-threaded 라 말한다). 이와 같이 배열이 single-

threaded 인지를 판별하는 작업은 reference counting을 통해 실행시에 행해질 수도 있으며, 컴파일시에 프로그램의 정적분석을 통해 행해질 수도 있다. 전자의 경우에 빈번한 reference count 조작을 위해 특별한 하드웨어를 사용하고 있으나 그러한 하드웨어가 복잡하고 incremental 방법 자체가 배열 원소 생성의 병렬성을 억제하고 있기 때문에 앞서 제시한 효과적인 구조화 자료 조작의 요건을 만족시키는데 문제가 있다. 후자의 경우에 single-threading의 효과적인 탐지를 위해 intelligent한 컴파일러가 필요하고 그렇지 않은 경우 언어상에 그러한 탐지가 용이한 construct를 사용하거나 사용자가 그러한 표현을 하도록 요구하므로써 함수언어의 선언적(declarative) 성격을 상실할 수 있으며, 또한 incremental 방법 자체의 한계인 배열 원소 생성시에 병렬성 이용이 용이하지 않은 단점이 있다.

b. Monolithic 방법

이 방법은 단어 자체가 의미하는 바와 같이 배열을 한번에 생성하고 일단 생성된 후에는 수정을 할 수 없는 방법으로 일반적으로 아래의 세가지 연산을 사용한다.

i. mka(혹은 make-array) (1,u) f :
index bound가 (1,u)이고 i번째 원소의 값이 (f i)인 배열을 생성하는 연산이다.

ii. mka-1 (1,u) f :
(f i)는 (j,v)를 반환하고, 그 배열의 j번째 원소의 값이 v인 배열을 생성하는 연산이다. 이 연산은 mka 연산만으로는 표현할 수 없는 분제(예를 들면 inverse permutation 문제 - A[B[i]] = i)를 표현하기 위하여 도입된 연산으로 mka와는 달리 f가 원소값 뿐만 아니라 index도 계산해야 한다.

iii. select(a,i) :
배열 a의 i번째 원소값을 반환하는 연산이다.

이 방법은 incremental 방법과는 달리 update 연산이 없다. 따라서 배열의 일부분을 변경하려 한다면 새로운 배열을 생성해야 한다. 그러나 주의해야 할 점은

그러한 배열의 일부분의 변경이 굳이 배열 자체의 변화를 통해 행해져야 하느냐 하는 것이고, 또한 과연 그러한 표현 방법이 실제의 응용분야에서 발생하는 문제들의 표현에 빈번히 사용되어 필수적인가 하는 점이다. 이점은 응용분야의 문제에 대한 연구가 보다 진전된 후에만 정확한 판단이 가능하겠지만, MIT 대학의 Arvind의 연구에 의하면 monolithic 방법만으로도 대부분의 문제를 해결할 수 있다는 점을 지적하고 있다. 그러나 monolithic 방법의 한 변형인 I-structure를 제시한 Arvind는 비교적 많이 사용되면서 I-structure로는 표현하기 어려운 문제들을 제시하고 있다.

monolithic 방법은 구현하기 쉽고 또한 상당한 병렬성을 갖고 있다(즉 mka(-1) 연산의 구현시에 해당되는 원소들의 값을 동시에 계산할 수 있으며, 배열의 생산자와 소비자가 명확히 구분되어 보다 표현이 간단해 진다). 또한 mka(-1) 연산을 nonstrict 하게 구현하므로써 배열의 생산자와 소비자의 overlapping을 통한 병렬성도 이용할 수 있다.

I-structure는 mka(-1) 연산을 nonstrict 하게 구현한 한 예이다. 본 연구에서는 I-structure를 기본으로 하고 I-structure의 문제점을 해결하는 새로운 방법을 제안하였기 때문에 I-structure에 대해 보다 자세히 살펴보고자 한다.

c. I-structure

I-structure는 앞서 언급한 바와 같이 배열의 생성 및 소비의 계산에서 이용할 수 있는 모든 병렬성을 이용하는 구조이다. 본 절에서는 I-structure와 관련된 연산과 그러한 연산을 이용하여 mka(-1)를 표현하는 방법, I-structure의 구현방법, 그리고 I-structure의 제약점을 살펴보고자 한다.

i. I-structure 연산

- 할당(혹은 초기 생성) - array (1,u)
  index bound가 (1,u)인 배열을 할당(생성)하는 연산이다.
- 쓰기(write 혹은 constraint statement) - A[i] = v
  배열 A의 i번째 원소의 값을 v로 하는 연산이다. 주의해야 할 점은 배열의

한 원소에 값이 할당되면 더이상의 할당은 허용되지 않는다는 점이다.

. 읽기(read 혹은 select) - A[i]

배열 A의 i번째 원소의 값을 읽는 연산이다. 주의해야 할 점은 I-structure는 배열 전체에 동기화를 행하지 않고 배열의 각 원소에 대해 동기화를 행하는 nonstrict structure 이기 때문에 쓰기에 앞서 읽기가 먼저 수행될 수 있기 때문에 이러한 경우를 효과적으로 처리하기 위해 지연된 읽기(deferred reads)를 처리할 수 있어야 한다.

ii. I-structure를 이용한 array abstraction

프로그래밍 기법의 관점에서 볼 때, 프로그래머가 높은 수준의 array abstraction을 사용하고, 후에 이러한 abstraction을 구체적인 연산으로 바꾸어 주는 방법은 매우 바람직하고, 따라서 본절에서는 앞서 언급한 mka(-1)을 I-structure 연산을 사용하여 표현하고자 한다.

```
. Def mka (1,u) f
  = { A = array (1,u) ;
      { For i From 1 To u Do
          A[i] = f i }
      In
        A } ;
. Def mka-1 (1,u) f
  = { A = array (1,u) ;
      { For i From 1 To u Do
          j, v = f i ;
          A[j] = v }
      In
        A } ;
```

위에서 사용한 언어는 일반적으로 통용되는 함수언어의 syntax와 semantics와

유사하다. 위의 표현에서 보는 바와 같이 우리는 자세한 I-structure 연산을 사용하기 보다 mka(-1)를 사용하여 프로그래밍 할 수 있고 그것은 컴파일러를 통해 자동으로 I-structure 연산으로 변환이 가능하기 때문에 프로그래밍이 보다 용이해 진다.

iii. I-structure의 구현

I-structure는 배열의 원소 각각에 대해 동기화를 행하는 nonstrict structure 이기 때문에 각 원소별로 값이 존재하는 가를 표시해 주는 별도의 비트(bit)가 필요하고, 또한 읽기/쓰기의 순서가 일정치 않기 때문에 지연된 읽기를 처리해 주는 방법이 또한 필요하다.

I-structure는 그림 4.1과 같은 형태로 되어 있다.

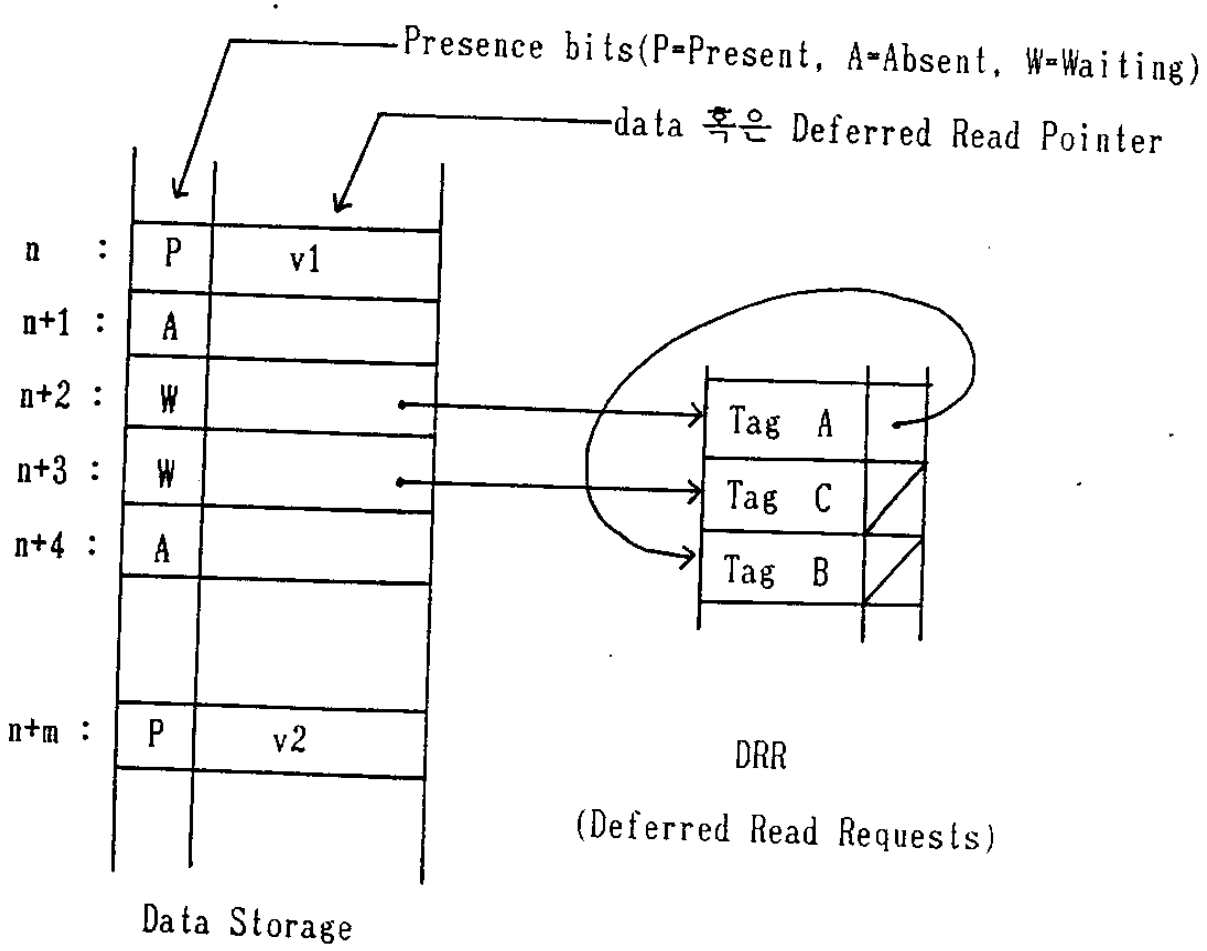
. array(1,u) 연산은 I-structure controller에 index bound가 (1,u)인 배열을 생성할 것을 요청하고 controller는 크기가 u 이고 각 원소의 presence bit가 'A'인 배열을 할당하고 그 배열의 첫 주소를 반환한다.

. A[i] = v 연산은 배열 A의 i번째 원소의 주소에 v 값을 쓴다. 주의해야 할 점은 쓰기 연산을 수행할 때에 이미 presence bit가 'P'이면 error가 된다는 것과, DRR(Deferred Read Requests)에 대한 포인터가 있으면(즉 presence bit가 'W'이면) controller는 DRR에 있는 읽기 요구들을 실행한 후 결과를 해당 계산단위로 보낸다.

. A[i] 연산은 배열 A의 i번째 원소값을 읽는 연산으로 presence bit가 'P'이면 직접 해당값을 읽고 반환하면 되고, presence bit가 'W' 이거나 'A'이면 그 원소값이 아직 생성되지 않았다는 사실을 표시하기 때문에, 후에 생성되면 그값이 곧장 반환될 수 있도록 DRR에 저장하고 그것에 대한 포인터를 생성한다. 이미 DRR에 여러 읽기 연산이 저장되어 있는 경우 chained 기법을 사용하여 queue의 끝에 저장한다.

iv. I-structure의 제약점

I-structure가 병렬성의 이용이 용이하고, 구현이 간단하고, 프로그래밍이 쉬운



위의 구조를 생성하는 하나의 가능한 실행 순서

. 계산단위 A - A[2]

. A[m] = v2

. 계산단위 C - A[3]

. A[0] = v1

. 계산단위 B - A[2]

. A[0]

그림 4.1 I-structure

장점이 있으나, Arvind가 지적한 바와 같이 어떤 특정 문제에 있어서는 효과적이지 못하다 [ARNP 86].

. Histogram 문제 : 각각 한 수를 생성하는 많은 생성자가 있고, 우리는 10개의 구간으로 나눈뒤 각 구간에 대한 이러한 값들의 빈도(frequency)를 계산하려 한다. 효과적인 병렬 계산 방법의 하나는 각 원소가 accumulator로서 0으로 초기화된 10개의 원소를 갖는 배열을 생성한 후에, 각 생성자는 병렬로 수를 생성하고, 그것의 결과는 임의의 구간 j에 분류되고 j번째 acculuator가 증가된다. 물론 accumulation의 순서는 중요하지 않다. 그러나 이 경우에 주의해야 할 점은 histogram 문제의 최종 결과는 마지막 생성자가 생성한 값이 완전히 감안된 시점에서야 비로서 생성되는데, 우리는 이 시점에서 I-structure로는 accumulator를 표현할 수 없을 뿐만 아니라 고육책을 사용하여 설혹 표현한다 하더라도 accumulation의 종료를 표현할 방법이 없다는 사실을 알 수 있다(즉 I-structure의 nonstrictness 성질이 수정되어야 한다).

. Polynomial 문제 : symbolic algebra 계산을 수행하는 시스템에서, polynomial을 곱하는 문제를 생각해 보자. 다음과 같은 polynomial에 대한 한 표현방법은 각 원소가 각각 $a_0,a_1,a_2,\ldots,a_n$ 값을 갖는 n+1 개의 원소로 구성된 배열을 사용하는 것이다.

$$a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \qquad \text{(polynomial A)}$$

$$b_0 + b_1x + b_2x^2 + \ldots + b_nx^n \qquad \text{(polynomial B)}$$

위의 두 polynomials A,B를 곱하기 위해, 우리는 각각 0으로 초기값을 갖는 accumulators 역할을 하는 2n 개의 원소를 갖는 배열 C를 생성하고, 각 원소 j에 대해 j+1 개의 (a_0 x b_j, a_1 x b_{j-1}, ..., a_j x b_0) 프로세스를 생성한 뒤, 이러한 각 프로세스들이 완성할 때마다 그것의 결과가 j번째 원소에 더해지도록 하면 될 것이다. 물론 각 index에 대해 accumulation의 순서는 중요하지 않다. 이 문제에서 주의해야 할 점은 배열의 각 원소에 대한 accumulation이 종료되는 시점을 파악해서 종료된 원소에 대해서는 I-structure의 nonstrictness를 이용할 수 있지만, I-structure의 오직 한번 쓰기가 가능하다는 성질로 인해, I-structure로는 이 문제를 표현할 수 없다.

위의 문제점을 요약하면 다음과 같다.

. accumulator를 표현할 수 없다.

. accumulation의 종료를 표현할 수 없다.

(2) 리스트/스트림

본 연구는 수치계산 뿐만 아니라 인공지능 분야에서 널리 이용되는 기호처리(symbolic processing)도 효과적으로 수행할 수 있는 범용컴퓨터를 설계하는 것이 목표이기 때문에, 비록 리스트중 알고리즘에 따라 배열로 리스트를 구현할 수 있고 그러한 연구가 부분적으로 행해지고 있으나, 인공지능 분야 개발에 초석이 되었던 LISP이나 PROLOG 등은 리스트를 기본적인 자료구조로 채택하여 발전하여 왔고, 배열로는 자연스럽게 표현할 수 없는 많은 문제가 있기 때문에 우리는 이러한 리스트를 효과적으로 수행할 수 있는 방안에 대해 연구하였다. 또한 원소의 생성과 소비가 순차적으로 수행되는 스트림은 특히 history sensitive한 계산에(예를 들면 I/O 문제) 많이 사용되기 때문에 그 필요성 또한 역설되어 왔다. 그러나 lenient cons가 가능한 경우에 리스트가 스트림을 표현할 수 있기 때문에, 우리는 스트림을 별도로 취급하지 않고 리스트에 관해 중점적으로 연구를 해왔다.

기존의 데이타 플로우 시스템이 주로 수치계산 분야에의 적용을 목표로 설계되었기 때문에 이와 관련된 부분은 주로 Reduction Machine에서 행해졌으나, 일본에서는 5세대 컴퓨터의 기본 언어로 PROLOG 형태의 논리 프로그래밍 언어를 채택하고, 기본 머신으로 데이타 플로우 시스템을 채택하여 연구를 수행하였기 때문에 비교적 리스트 처리에 관한 연구가 많이 수행되어 왔고, 대표적인 연구로는 NTT에서 Amamiya 연구팀의 list-processing-oriented data flow machine [ARTH 86]이 있고, 우리는 이들의 idea를 대부분 채택하기로 하였다.

4.1.4 제안된 구조화 자료 조작 기법

앞서 언급한 것처럼 우리는 수치계산뿐만 아니라 기호처리도 효과적으로 행할수 있으며, 뒤에서 자세히 논의되겠지만 두가지 형태의 함수 적용을 지원하는 동적 데

이타 플로우 모델을 지원해 줄 수 있는 고속의 다중처리 시스템을 설계하려 하기 때문에, 배열로는 Heap 구조의 배열과 I-structure 계통의 배열을 사용하고, 리스트는 앞서 말한 바와 같이 lenient cons와 lazy cons를 지원하는 NTT의 방법을 채택하였다.

배열의 경우에 Heap 구조의 배열을 포함한 이유는 다음과 같다. 두 가지의 형태의 함수 적용중 (이에 대한 자세한 내용은 4.2 절에서 제시된다) 노드내에서 처리되는 함수 적용은 함수 수준의 nonstrictness를 이용한 부분적 계산을 통해 병렬성을 올리는 형태로 특별히 함수의 인자들에 대한 구조화 자료의 사용을 통한 동기화 기법이 불필요한 반면에, 다른 노드로 함수를 전가시켜서 계산을 수행하는 exportable 함수 적용의 경우에는 전달할 함수의 인자들을 함께 모아서 그것을 패킷 형태로 만든후에 전달하여야 하기 때문에, 그러한 인자들을 하나의 구조화 자료로 모아야 될 필요가 있다. 만약 해당 인자들을 모으지 않고 부분적 함수 적용을 통해 계산을 수행하는 경우에 관련된 노드간에 인자의 참조 및 전달 과정이 상당한 시간과 노력을 필요로 하기 때문에, 인자의 참조가 적은 경우를 제외하고는, 오히려 함수적용을 다른 노드로 이동해서 병렬성을 올림으로써 계산 속도를 증진시키려는 우리의 의도가 퇴색될 가능성이 높다. 그러나 이 경우에도 물론 인자에 대한 구조화 자료로 I-structure를 사용할 수 없는 것은 아니나, I-structure의 성격상 각 원소에 대해 동기화를 수행하기 때문에 인자들이 다 계산되어 구조화 메모리에 쓰여졌는 지를 확인하기 어렵고 (이 문제는 앞서 언급한 I-structure가 갖는 accumulation 종료 표현의 한계와 일치한다), 뒤에서 논의하겠지만 이러한 문제를 해결하는 E1 구조 배열을 사용한다 하더라도, 어차피 그것의 생성이 끝난후에 전달되기 때문에 구조화 자료에 대한 포인터를 때이르게 전달해서 발생할 수 있는 문제점들을 피하기 위해, Heap 구조 배열도 부분적으로 사용하기로 한다.

또한 본 연구에서는 I-structure가 갖는 두 가지의 문제점을 해결하는 방안으로 두 가지의 확장된 I-structure를 제시한다. Heap 구조 배열은 이미 널리 알려진 배열의 형태이고 리스트 처리는 4.2 절에서 논의될 데이타 플로우 베이스 언어 부분에서 논의될 것이기 때문에, 본 절에서는 주로 확장된 I-structure에 대해 논의한다.

전절에서 언급한 바와 같이 I-structure는 병렬처리, 프로그래밍, 그리고 구현이 용이한 반면에 histogram/polynomial 문제등을 해결하지 못하는 단점이 있다. 우리가 생각하기에 이러한 제약점은 아래와 같은 문제때문에 기인한다고 생각한다고 생각한다.

. histogram 문제는 문제의 성격상 각 원소값의 계산은 병렬로 계산하는 것이 가능할 지라도 원소에 값을 저장하는 것은 정확한 실행을 위해서 순차적으로 행해져야 하며, 그러한 배열을 소비하는 계산단위는 모든 원소의 계산이 완성된 후에야 비로서 그 배열을 사용할 수 있다는 점이다. 즉, 문제의 성격상 배열의 각 원소의 저장 및 수정이 순차적으로 이루어져야 하고 배열의 소비는 모든 원소가 완전히 생성된 후에야 비로서 가능하다는 것이다. 따라서 I-structure가 이용할 수 있는 두가지 형태의 병렬성(다른 원소에 대한 동시 저장, 생산자/소비자 간의 overlapping) 모두가 허용되지 않는다는 점이다.

. polynomial 문제는 histogram 문제와는 달리 I-structure가 이용할 수 있는 두가지 형태의 병렬성을 모두 이용할 수는 있지만, I-structure가 갖는 '오직 한번의 쓰기 허용' 문제로 인해 각 원소에 부가적인 accumulation이 불가능하다는 점이다.

물론 histogram 문제나 polynimial 문제 같은 것이 우리가 적용할 응용분야에 얼마만큼 비중있게 사용될 것인지는 보다 연구되어져야 하겠지만, 여하튼 I-structure가 대부분의 경우에 매우 효과적이라는 것을 알고, 또한 그 구조가 해결할 수 없는 문제가 있다면, 대부분의 경우에는 I-structure를 사용하고 I-structure로 해결할 수 없는 문제에 대해서는 약간 변형된 구조를 사용하는 것이 현시점에서 가장 바람직한 접근 방향이라고 생각한다.

이러한 시각에서 우리는 I-structure와 유사하나 histogram 문제와 polynomial 문제의 해결을 위한 두가지 구조인 EI-structure-1(Extended I-structures 1)과 EI-structure-2(Extended I-structure 2)를 제시한다.

(1) EI-structure-1(줄여서 E1)

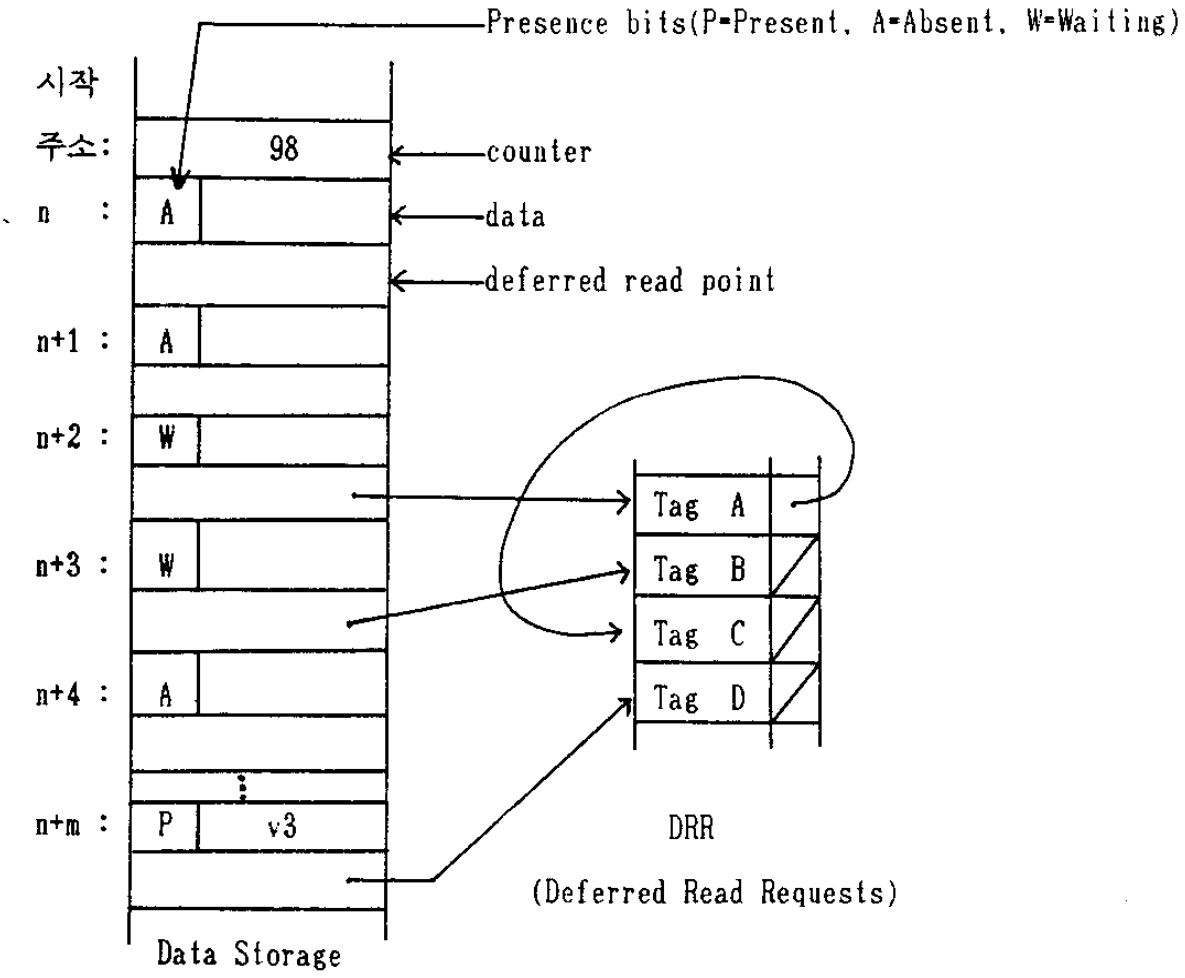
이구조는 histogram 문제를 해결하기 위해서 제안한 것으로 그림 4.2와 같다. E1 구조의 조작은 다음과 같다.

. 할당 - array-1 (1,u,k)
이것은 배열 메모리 controller에 index bound가 (1,u)인 E1 구조의 배열을 할당할 것을 요구하고 controller는 그 배열이 E1 구조이고 원소 생성에 간여하는 계산단위가 k개임을 인지하고 k값을 갖는 counter를 set하고, 배열을 할당, 초기화하고 포인터를 반환한다.

. 쓰기 - A[i] = v
배열 A가 E1 구조를 갖는다면(그러한 사실이 controller로 부터 할당시에 전달받은 패킷에 수록되어 있음을 가정한다) 이 연산은 구조화 메모리의 counter 값을 하나 감소시키고 그값을 해당 주소에 쓰고, controller에 counter 값을 하나 감소시킨다. 또한 DRR에 있는 모든 연산에 그값을 전달하여 수행토록 한다. E1 구조의 counter 항이 0일 때 쓰려고 하는 연산은 error로 간주된다.

. 읽기 - A[i]
배열 A가 E1 구조를 갖는다면 읽기 연산은 counter 항을 검토하고 만약 0이면 곧장 읽기가 수행되어 그값을 사용할 수 있지만 만약 0이 아니면 DRR에 저장하고 포인터를 연결한다.

그러나 I-structure와 같이 DRR을 유지하는 것이 효과적인 가에 관해서는 보다 연구가 되어져야 할 것이다. 이것의 이유는 E1 구조의 소비는 E1 구조의 원소 전체가 완전히 생성된 후에야 비로서 가능하기 때문에 I-structure와 동일하게 DRR을 유지할 경우에 E1 구조의 생성과 동시에 구조화 메모리의 처리율과 통신율이 폭발적으로 증가할 것이 예견되기 때문이다. 이 문제는 기본적으로 E1 구조의 할당후 그것에 대한 포인터를 E1 구조를 소비하는 계산단위에 즉각 넘겨 주는 현재의 전략에 기인한다. 따라서 이러한 전략을 수정하는 것이 DRR을 유지하는 것보다 효과적인 것인지는 보다 연구가 행해져야 할 것이다.



위의 구조를 생성하는 하나의 가능한 실행 순서

(처음에 counter가 100으로 set 되었다고 가정했을 때)

. 계산단위 A - A[2]

. A[m] = v2

. 계산단위 B - A[3]

. A[m] = v3

. 계산단위 C - A[2]

. 계산단위 D - A[m]

그림 4.2 EI-structure-1

주의해야 할 점은 전체 배열에 하나의 counter를 유지해야 하기 때문에 배열 메모리를 분할하면 문제가 보다 복잡하기 때문에 배열의 매핑시에 한 모듈에 할당되는 것이 바람직하다는 점이다.

(2) EI-structure-2(줄여서 E2)

이 구조는 polynimial 문제를 해결하기 위해 제안된 것으로서 그림 4.3과 같다. E2 구조의 조작은 다음과 같다.

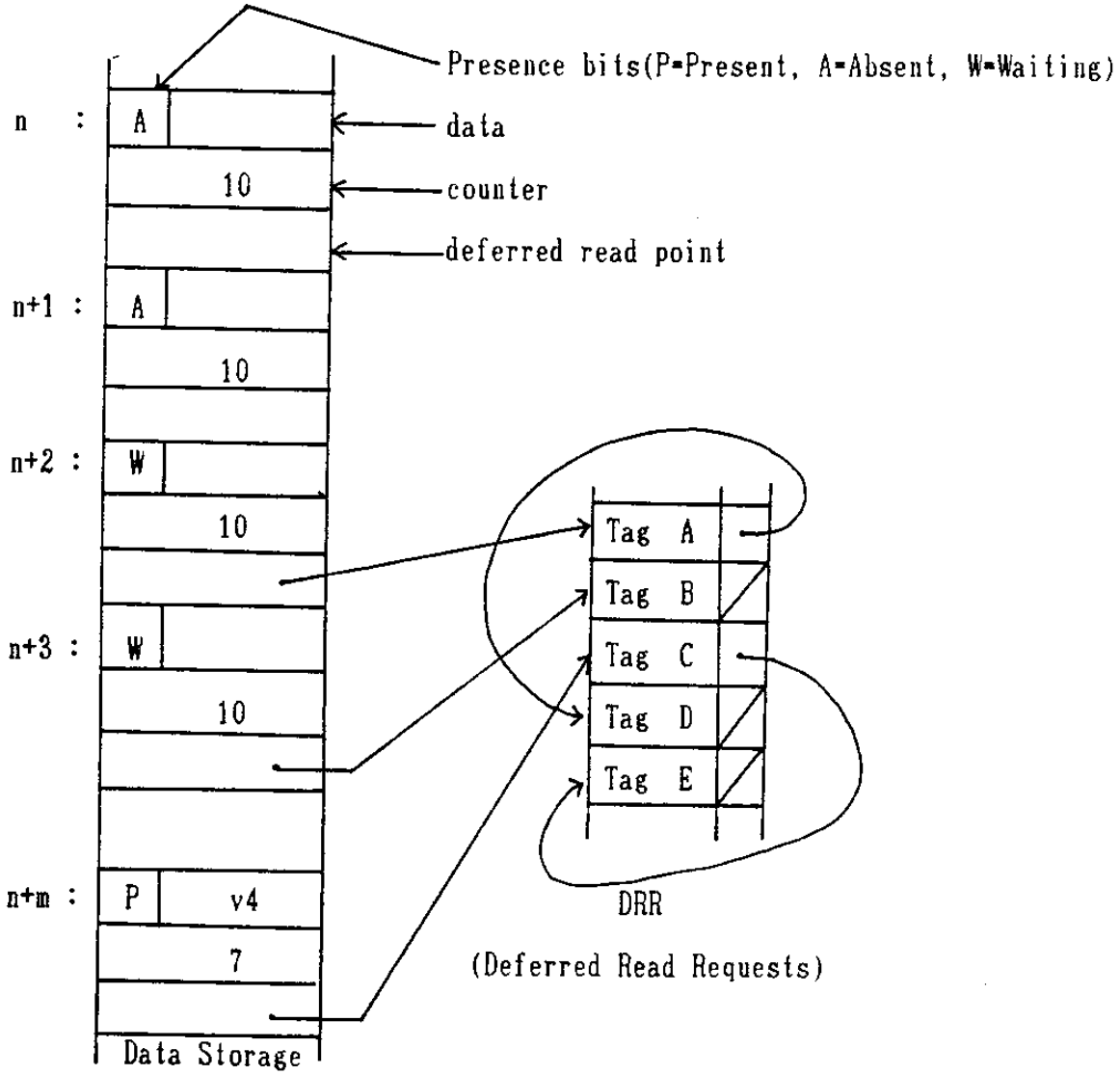
. 할당 - array-2 (1,u,n)
이것은 배열 메모리 controller에 index bound가 (1,u)인 E2 구조의 배열 할당을 요구하고, controller는 그 배열이 E2 구조이고 각 원소 생성에 간여하는 계산단위가 n 개임을 인지하고, 배열을 할당,초기화하고 포인터를 반환한다(초기화 시에 각 원소의 counter 항에 n을 기록하고 나머지는 E1과 유사함).

. 쓰기 - A[i] = v
배열 A가 E2 구조를 갖는다면 쓰기는 해당 주소에 직접 값 v를 쓰고 counter를 하나 감소시킨다. 만약 counter 값이 0이 되면, DRR을 검토하고 지연된 읽기 요구를 처리한다. counter 항이 0일때 쓰려고 하는 연산은 error로 간주된다.

. 읽기 - A[i]
배열 A가 E2 구조를 갖는다면 읽기 연산은 해당 주소의 counter를 검토하고 만약 0이면 곧장 읽기를 수행하고 만약 0이 아니면 DRR에 저장하고 포인터를 연결한다.

여기서 주의해야 할 점은 E2 구조가 I-structure와 마찬가지로 병렬 수행을 증진시키기 위해 분할된 메모리에 저장되어도 정확한 수행에 아무런 지장이 없다는 점이고, 이것은 E2 구조가 원소에 대한 동기화를 행하기 때문에 가능하다.

4.2 데이타 플로우 베이스 언어 (DDFBL: Divide-and-conquer Data Flow Base Language)



위의 구조를 생성하는 하나의 가능한 실행 순서

(각 원소에 대한 accumulation counter가 10이라 가정할 때)

1) 계산단위 A - A[2]　　2) A[m] ← v2

3) 계산단위 B - A[3]　　4) A[m] ← v3

4) 계산단위 C - A[m]　　5) 계산단위 D - A[2]

5) A[m] ← v4　　6) 계산단위 E - A[m]

그림　4.3　EI-structure-2

데이타 플로우 베이스 언어는 데이타 플로우 프로그래밍 언어의 기본적인 의미 구조(semantic constructs)에 대한 연구 모델로 사용되며 프로그래밍 언어의 중요한 기능에 대한 최종적인 표현이 되며(흔히 병렬 기계 언어라 불린다), 프로그래밍 언어와 그것을 구현하는 구조와의 개념적 교량 역할을 담당한다. 따라서 베이스 언어에 대한 연구는 프로그래밍 언어 및 구조에 관한 연구의 기본이 된다 [DENN 74].

그러므로 본 연구에서는 실제의 프로그래밍 언어와 구조의 세부적 개념 설계에 앞서, 1차 년도에서 제시한 HYPERDAC의 기본적인 원칙들, 앞서 제시한 구조화 자료 조작, eager/lazy evaluation, 그리고 두 가지 형태의 함수 적용을 효과적으로 지원하여 줄 수 있는 베이스 언어에 대해 연구하였다.

4.2.1 제안된 베이스 언어의 특징

1차 년도의 연구에서 제시한 HYPERDAC의 기본적인 원칙들을 효과적으로 지원하면서 보다 광범위한 응용 범위를 갖는 베이스 언어를 설계하였으며, 그것의 특징은 다음과 같다.

- 수치 계산 및 기호 처리 분야를 포함하는 광범위한 응용 분야에의 사용을 위해 배열 및 리스트 수형을 효과적으로 지원할 수 있도록 하였다.
- 제한된 자원 하에서 효과적인 병렬성의 제어를 위해 lazy evaluation을 지원하고, 자원이 허용하는 환경하에서 고도의 병렬성을 이용할 수 있도록 eager evaluation도 지원하며, nondeterminate 계산과 nonstrict 계산을 지원해 줌으로써 보다 효과적이고 강력한 표현력을 갖는다.
- 함수의 부분적 적용을 통한 고도의 병렬 처리와 함수를 유용한 다른 처리 요소로 전달하여 함수 수준의 병렬성도 꾀할 수 있는 두가지 형태의 함수 적용을 지원함으로써 최대한의 병렬 처리가 가능토록 하였다.

4.2.2 데이타 플로우 그래프에서의 기본적인 계산단위들

데이타 플로우 모델에서 프로그램은 데이타 플로우 프로그램 그래프라 불리는

directed graph로 표현된다. 여기서 노드는 프로그램에서 수행되는 계산에 해당하며 directed arc는 생산자로부터 계산되거나 혹은 생성된 값이 소비자로 전달되는 데이타 경로이다. arc는 논리적으로 FIFO 기억 장치로 작용하며 개념적으로 무한대의 길이를 가진 queue이며, arc를 통해 전달되는 값(value)은 거의 모든 종류의 형태가 될 수 있다. 가장 기초적인 데이타 플로우 모델에서는 값의 이동 단위인 토큰이 지닐 수 있는 값의 범위가 정수, 실수, 부울린, 구조화 자료 등의 데이타 자체에 국한되고, 보다 발전된 데이타 플로우 모델에서는 값을 갖고 있는 cell, 구조화 자료, 그리고 프로그램 코드에 대한 포인터를 토큰이 값으로 가질 수 있다. 이 절에서는 토큰이 정수, 실수, 부울린, 스트링 등의 가장 기초적인 값(primitive value)을 가지는 경우에 관련된 계산단위를 살펴보고 그외의 계산단위는 다음 절부터 계속해서 다룰 것이다 [한전 87a].

(1) 기초적인 연산자 계산단위(노드) (primitive operator nodes)

+, -, *, / 등의 산술 연산자와 <, <, =, >, > 등의 관계 연산자와 같이 기계적으로 직접 수행될 수 있는 기초적인 연산자 노드로 구성된다. 각 계산단위의 실행법칙은 그림 4.4에 제시되어 있다.

(2) 복사 노드 (DUP 혹은 IDENTITY 노드)

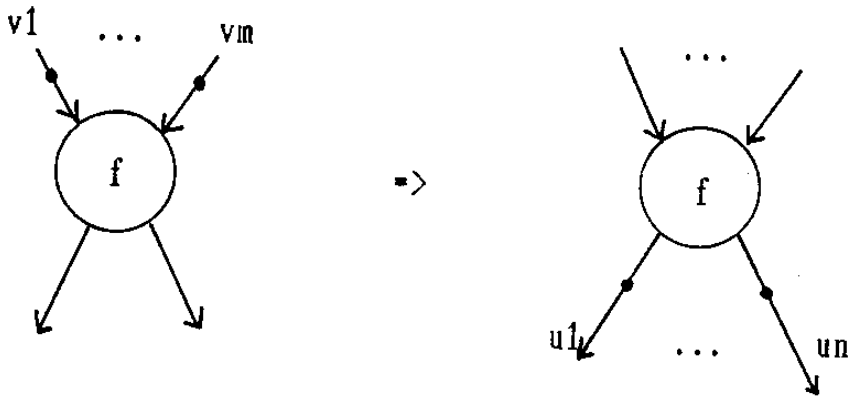
단순히 입력 값을 복사하는 노드로 동일한 값을 필요로 하는 여러 노드로 전달해야 할 경우에 사용한다.

(3) 분배 노드 (Distributor or Dist node)

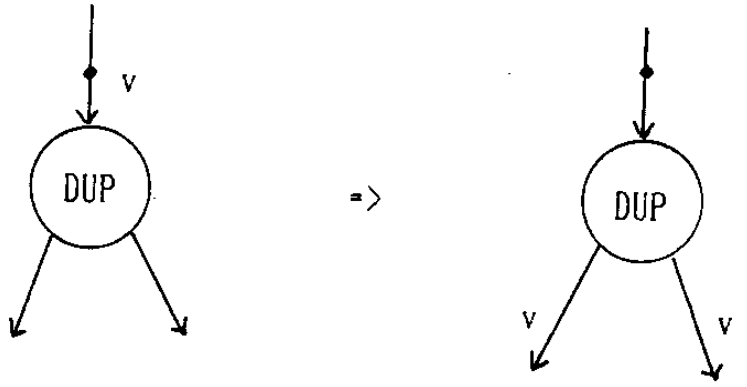
제어 입력 값에 따라 두개의 입력 값중 적절한 값을 출력하는 노드로 값의 흐름을 적절하게 제어하는 기능을 수행한다.

(4) 선택 노드 (Select or Sel node)

제어 입력 값에 따라 입력 값을 두개의 출력 노드중 하나의 적절한 출력 arc로 전달하는 노드로 값의 흐름을 제어하는데 사용된다.

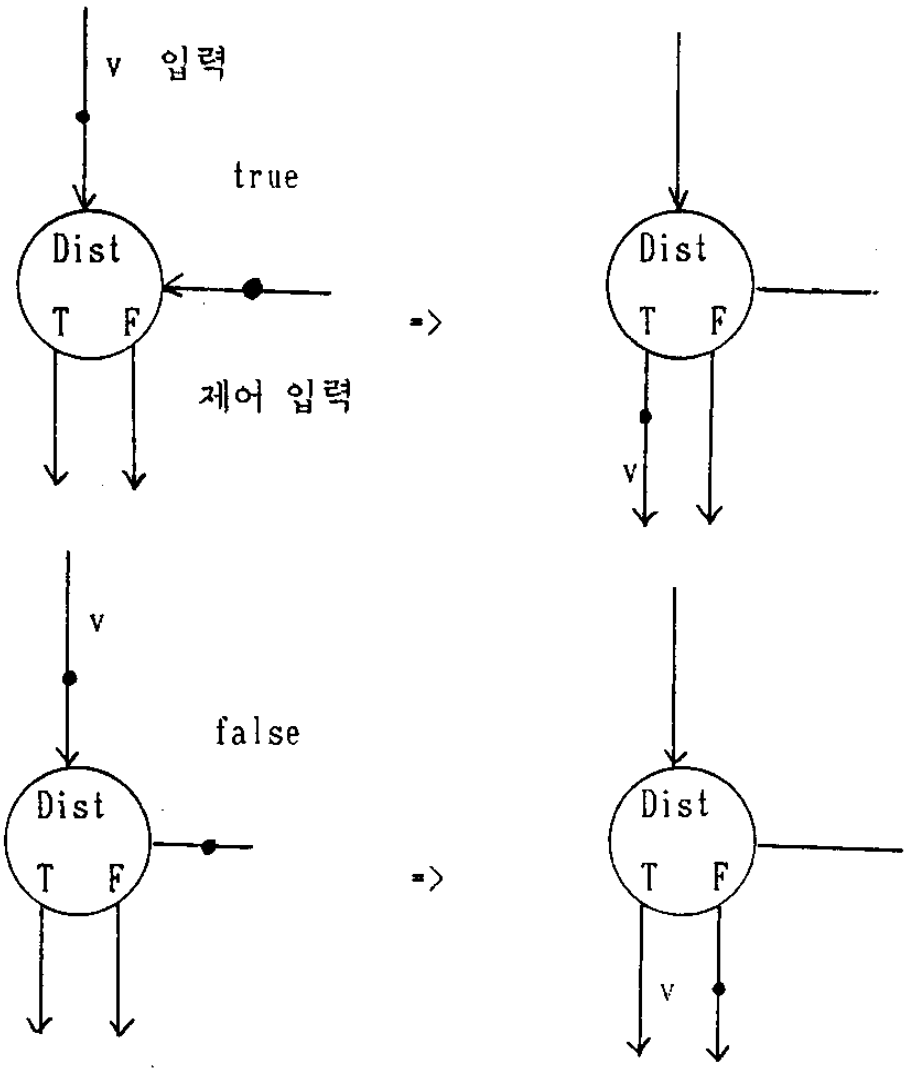


i. 기초적인 연산자 계산 단위



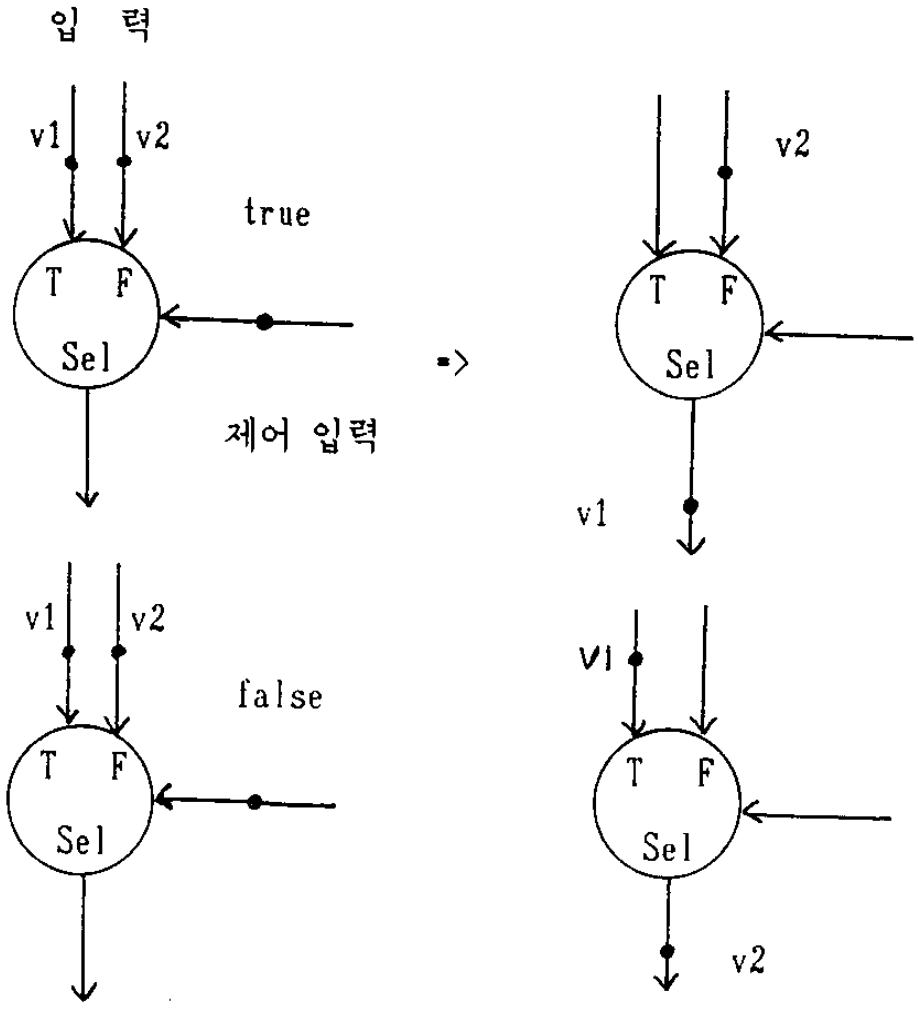
ii. 복사 노드 (DUP 혹은 IDENTITY 노드)

그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)



iii. 분배 노드 (Distributor or Dist node)

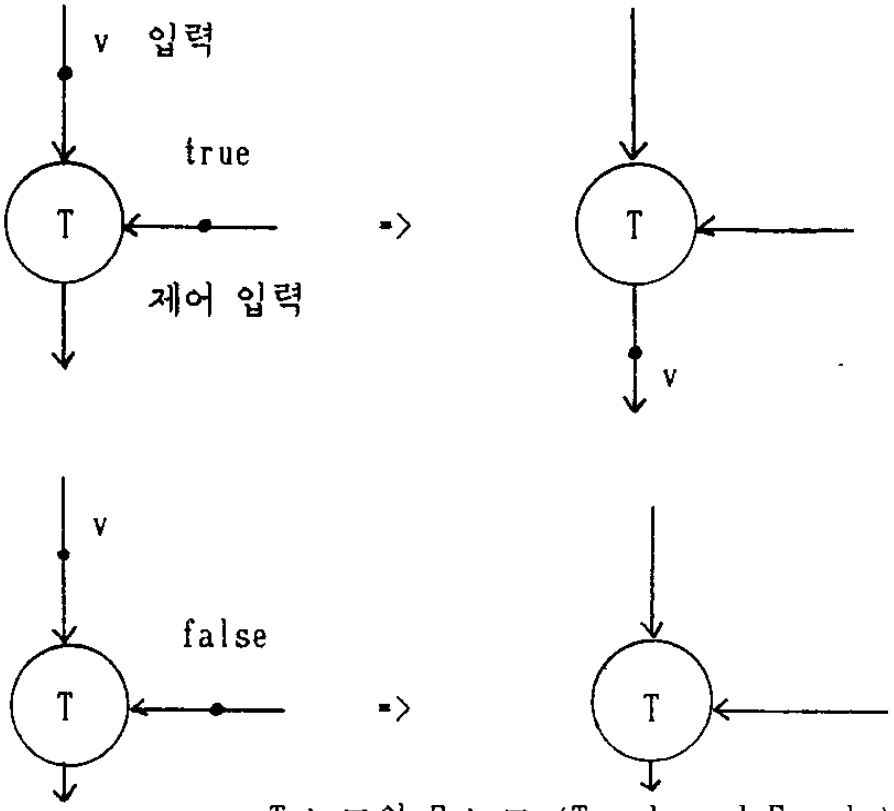
그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)



iv. 선택 노드 (Select or Sel node)

그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)

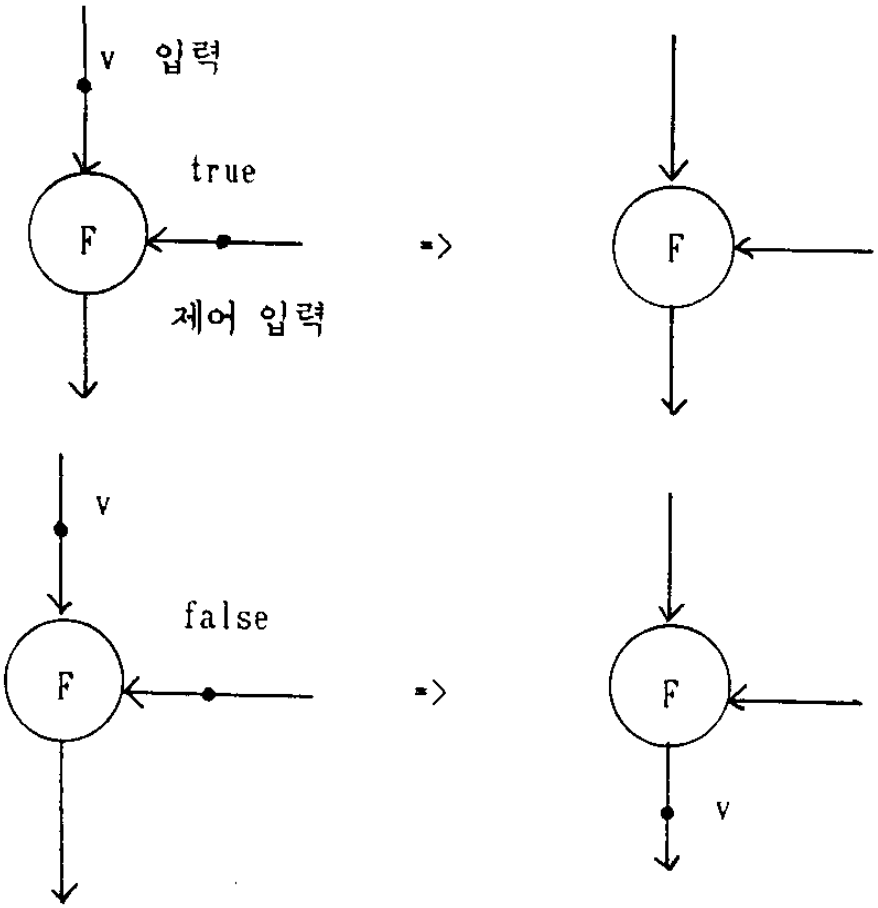
. T 노드



v.a T 노드와 F 노드 (T node and F node)

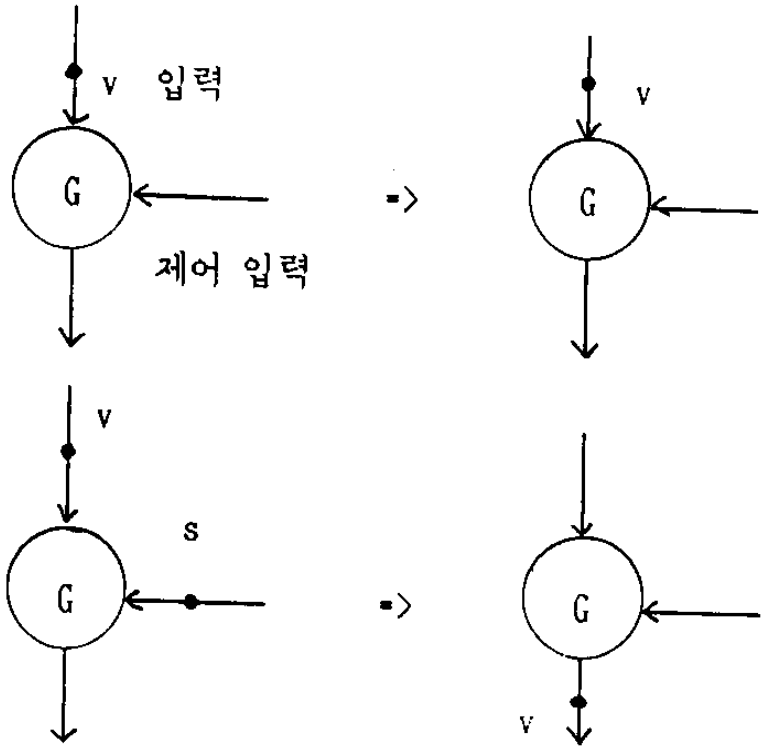
그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)

. F 노드



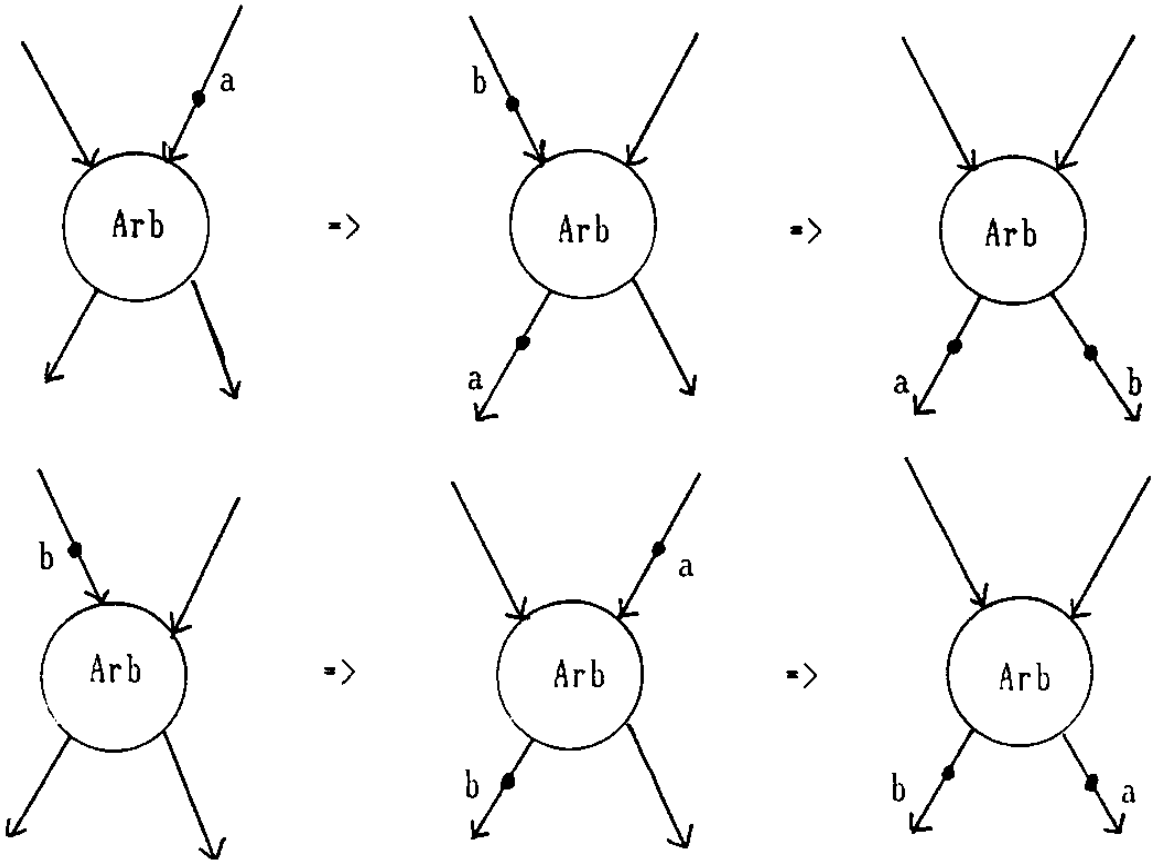
v.b T 노드와 F 노드 (T node and F node)

그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)

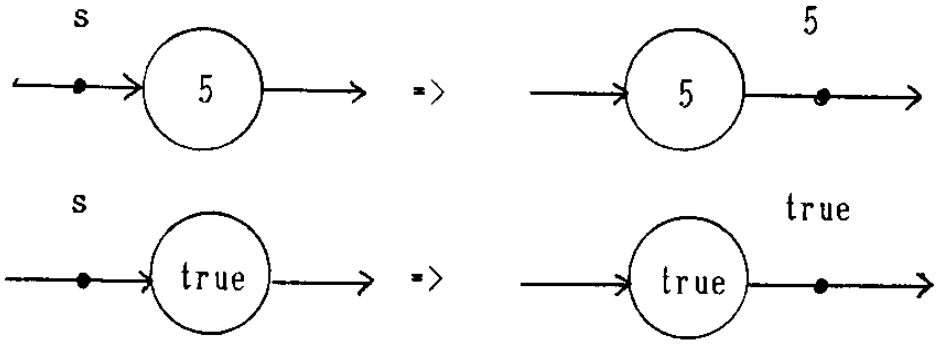


vi. G 노드 (G node)

그림 4.4 기본적인 계산 단위의 실행 법칙 (다음 장에 계속)



vii. Arbiter 노드



viii. 상수 생성 노드

그림 4.4 기본적인 계산 단위의 실행 법칙

(5) T 노드와 F 노드 (T node and F node)

제어 입력 값에 따라 입력 값을 전달하거나 흡수하는 노드로 값의 흐름을 제어하는데 사용된다.

(6) G 노드 (G node)

입력 값을 유지하고 있다가 제어 입력 값이 도달하면 그값을 전달하여 줌으로써 값의 흐름을 제어하는 노드로서 특히 eager/lazy evaluation을 구현할 때 사용된다.

(7) Arbiter 노드

Arbiter 노드는 2개의 입력과 2개의 출력을 갖는데, 2개의 입력 중 먼저 도착한 입력을 왼쪽 출력 arc에 출력하고 다른 한 값을 오른 쪽 출력 arc에 출력한다.

이 노드는 비결정형 계산(nondeterminate computation)을 표현하기 위하여 필요한 계산단위로서 스트림 합병(stream merge)이나 guarded command 등의 구현에 사용된다.

(8) 상수 생성 노드

입력 신호가 도착할 때에 해당되는 상수를 생성한다.

4.2.3 구조화 자료와 관련된 계산단위들

본 절에서는 구조화 자료에 속하는 배열과 스트림/리스트에 관련된 계산단위들에 대해 논의한다. 특히 배열과 스트림에 관련된 연산에 대해 자세히 논의하고 리스트에 관한 계산단위들은 4.2.6에서 다룰 것이다.

(1) 배열 연산 (Array operations)

본 연구에서는 배열을 효과적으로 다루기 위하여 4 가지 형태의 배열(heap 구조의 배열, I-structure, EI-structure-1, EI-structure-2)을 제공하며, 본 절에서는 각

각의 형태에 대한 기본적인 계산단위에 대해 다룬다.

a. Heap 구조 배열 (Arrayh)

이 배열은 주로 exportable 함수 적용에 대한 인자에 사용되며, 뒤에서 논의될 nonstrict 배열인 I-structure 등과는 달리 reference counter 등을 사용한 자료의 공유 및 동기화가 이루어지기 때문에 병렬성은 적은 반면에 배열 조작에 소요되는 과도한 동기화를 피함으로써 함수 적용을 처리요소간에 이동시킴으로써 병렬 계산을 꾀하는 경우에 효과적으로 사용될 수 있다.

이 구조와 관련된 기본적인 계산단위들은 다음과 같고, 실행법칙은 그림 4.5에 제시되어 있다.

i. Append 노드

입력 배열의 입력 인덱스 값이 입력 값으로 (추가)변형된 배열을 출력한다.

ii. Select_Arrayh 노드 (Select_h node)

입력 배열의 입력 인덱스에 해당되는 값을 출력한다.

iii. Concatenate 노드 (Conc or ‖ node)

두 개의 입력 배열을 결합한 배열을 출력한다.

iv. Join 노드 (element-by-element merge or array_join(A,B))

두 개의 입력 배열을 각 구성요소 간에 교차하여 결합된 배열을 출력한다.

v. Index 노드 (Index_h and Index_l nodes)

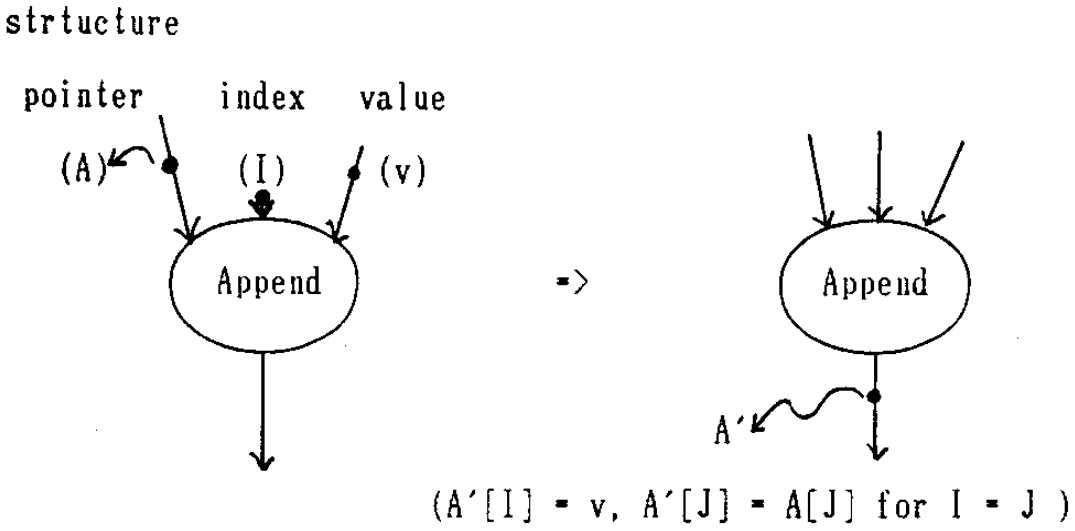
입력 배열의 가장 높은 (혹은 가장 낮은) 인덱스 값을 출력한다.

vi. Create 노드 (Create node)

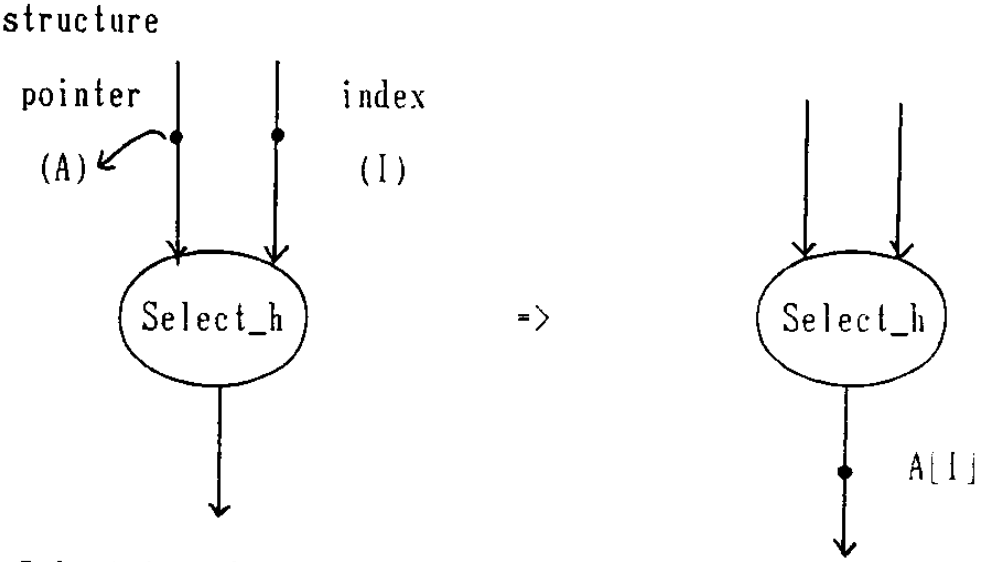
입력 수형을 구성 원소의 수형으로 가지며 구성 원소로는 아무것도 갖고 있지 않는 빈 배열을 출력한다.

vii. Create/fill 노드 (Crefil node)

가장 낮은 인덱스가 L이고 가장 높은 인덱스가 H이며 구성 원소의 모든 값이 v인 배열을 출력한다.

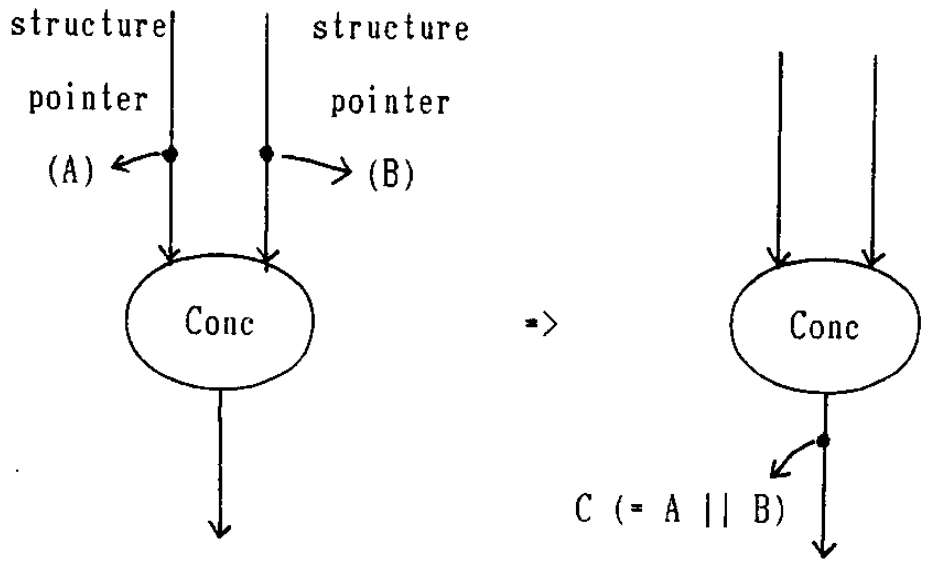


i. Append 노드

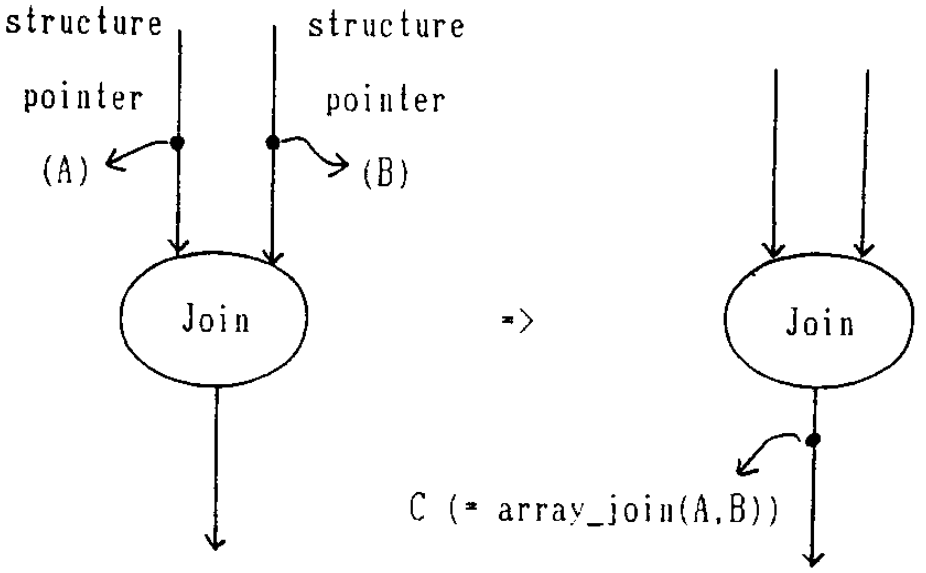


ii. Select_Arrayh 노드 (Select_h node)

그림 4.5 Heap 구조 배열의 계산 단위의 실행 법칙 (다음 장에 계속)

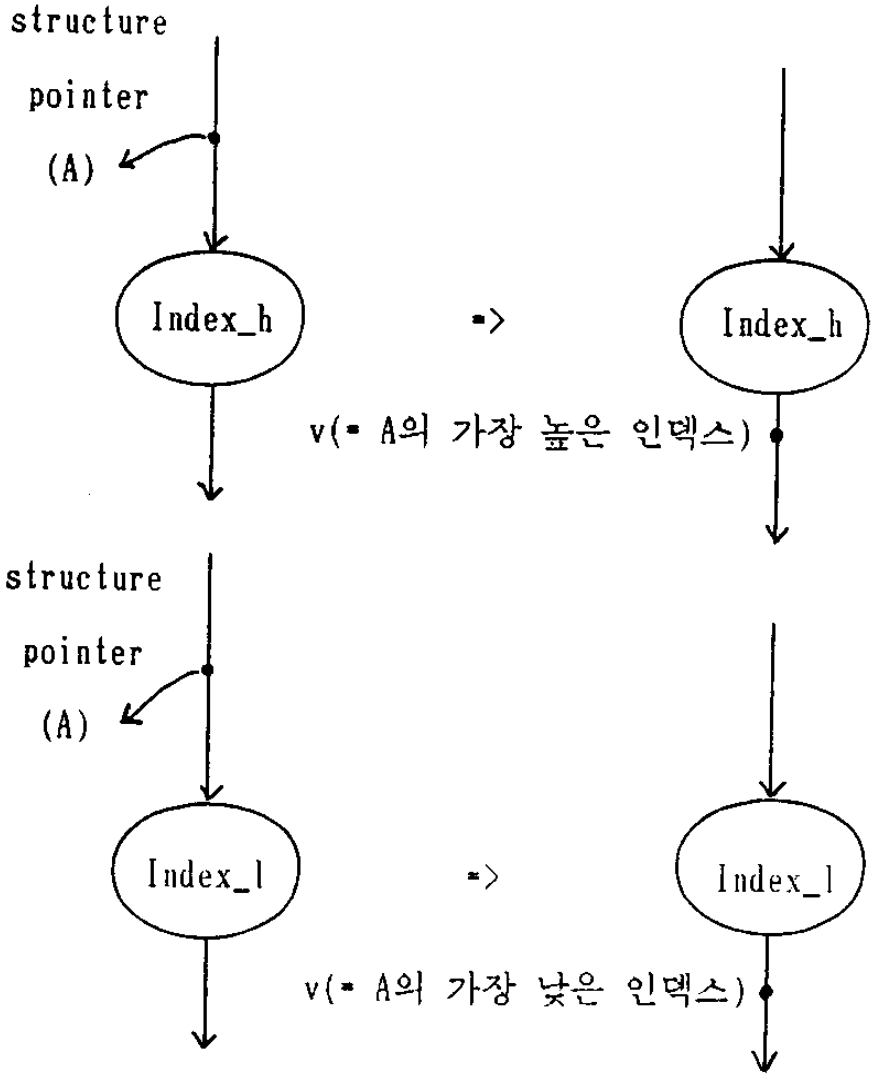


iii. Concatenate 노드 (Conc or || node)



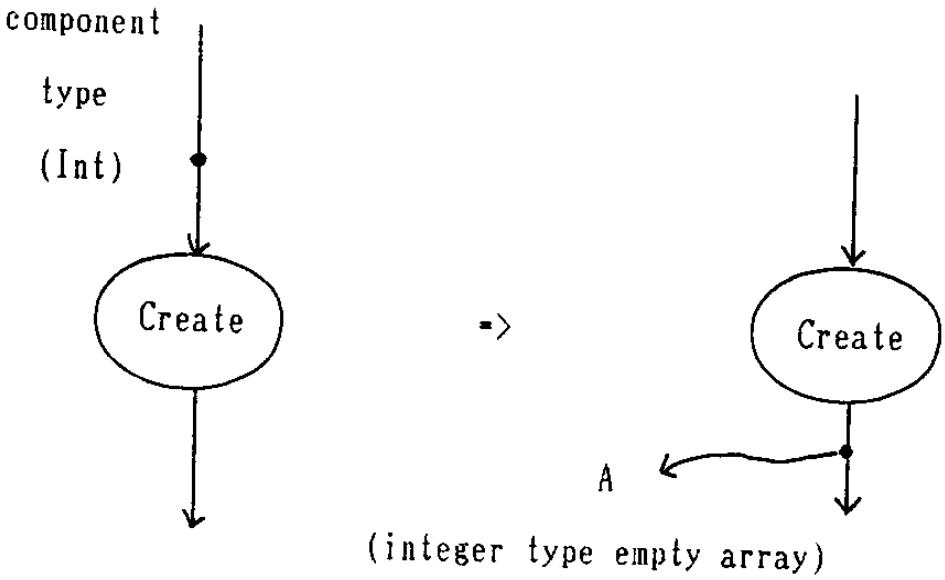
iv. Join 노드 (element-by-element merge or array_join(A,B))

그림 4.5 Heap 구조 배열의 계산 단위의 실행 법칙 (다음 장에 계속)

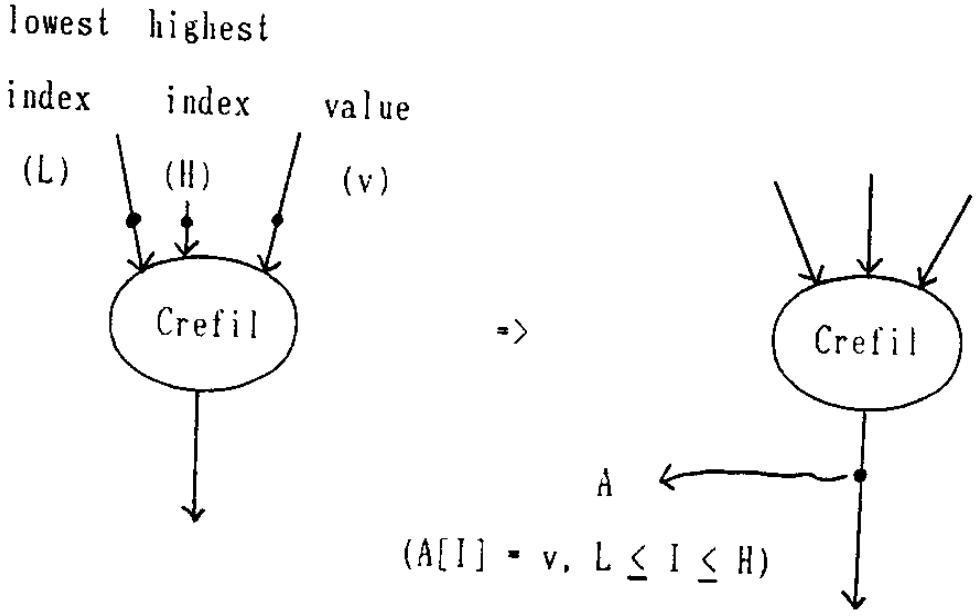


v. Index 노드 (Index_h and Index_l nodes)

그림 4.5 Heap 구조 배열의 계산 단위의 실행 법칙 (다음 장에 계속)

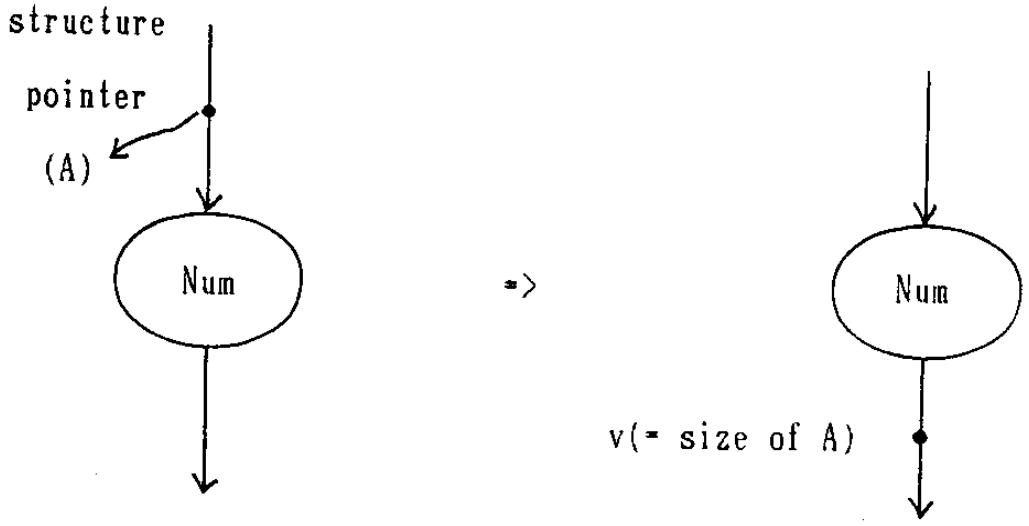


vi. Create 노드 (Create node)

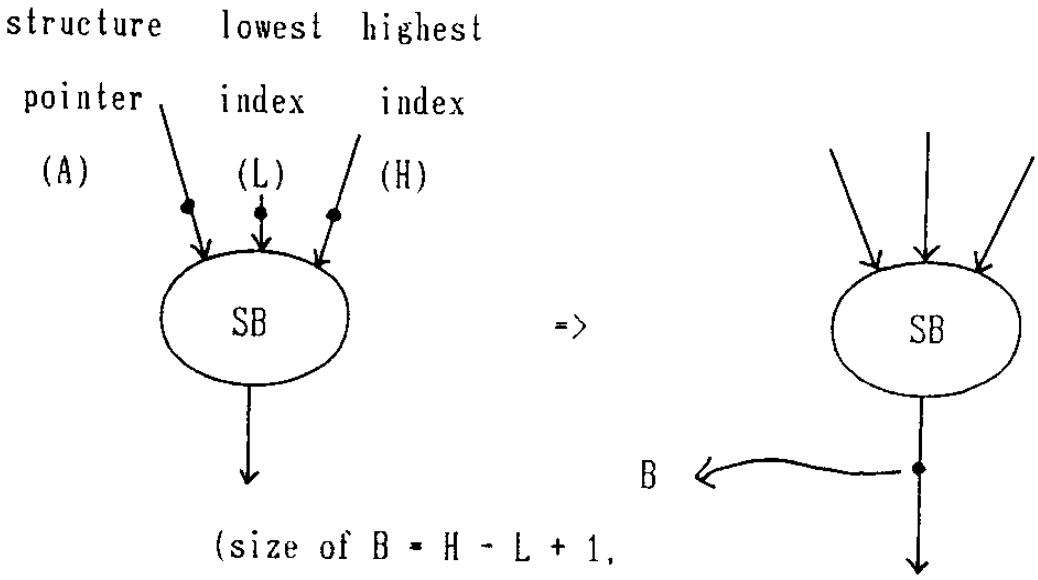


vii. Create/fill 노드 (Crefil node)

그림 4.5 Heap 구조 배열의 계산 단위의 실행 법칙 (다음 장에 계속)



viii. Number 노드 (Num node)



(size of B = H - L + 1,

B[I] = A[I], where Index_1(A) < L

B[I] = missing elements, otherwise)

ix. Set Bounds 노드 (SB node)

그림 4.5 Heap 구조 배열의 계산 단위의 실행 법칙

viii. Number 노드 (Num node)

입력 배열의 크기를 출력한다.

ix. Set Bounds 노드 (SB node)

입력 배열중 가장 낮은 입력 인덱스(L)와 가장 높은 입력 인덱스(H) 내에 있는 원소로 구성된 새로운 배열을 출력한다. 만약 L이 입력 배열의 가장 낮은 인덱스 보다 작거나 H가 입력 배열의 가장 높은 인덱스 보다 클 경우에 나머지 구성 원소들은 빈 원소(missing elements)로 간주된다.

b. I-structure

I-structure는 nonstrict 성질을 갖고 있는 배열로서 각 원소의 생성 및 소비에 대한 동기화 메카니즘을 갖고 있어서 앞서 설명한 Heap 구조 배열 보다 병렬성의 이용이 우월하며 exportable 함수 적용과 histogram 문제 그리고 polynomial 문제 등을 제외하는 대부분의 배열은 이러한 I-structure를 사용한다. 각 계산단위에 대한 실행법칙은 그림 4.6에 제시시되어 있다.

i. 할당 노드 (Allocation or Alloc node)

가장 낮은 인덱스가 L이고 가장 높은 인덱스가 H인 I-structure를 할당하여 그것에 대한 포인터를 출력한다.

ii. 읽기 노드 (Read node)

입력 I-structure의 입력 인덱스에 해당하는 값을 출력한다.

iii. 쓰기 노드 (Write node)

입력 I-structure의 입력 인덱스에 입력 값을 쓰고 그 배열을 출력한다.

iv. Bounds 노드 (Bounds node)

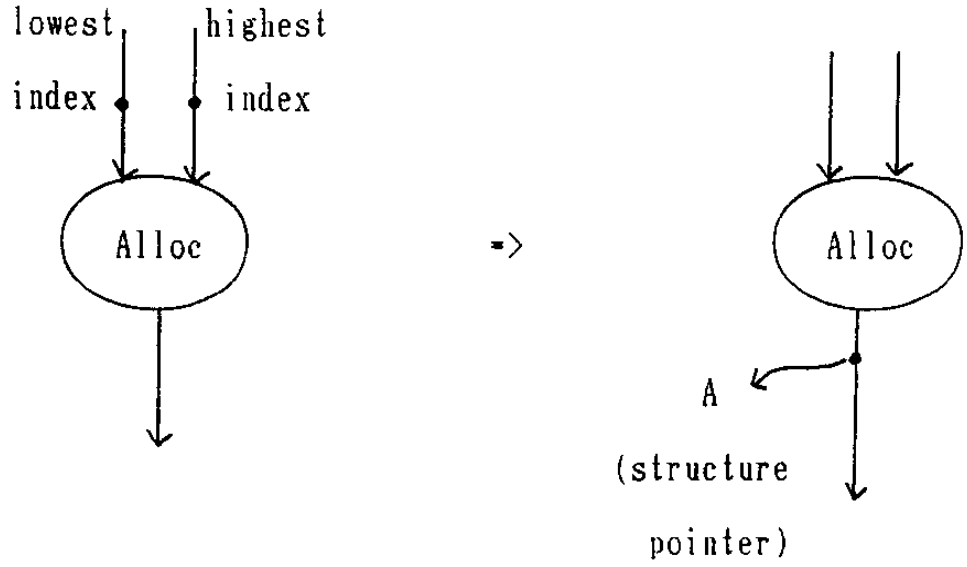
입력 I-structure의 가장 낮은 인덱스와 가장 높은 인덱스를 출력한다.

v. 크기 노드 (Size node)

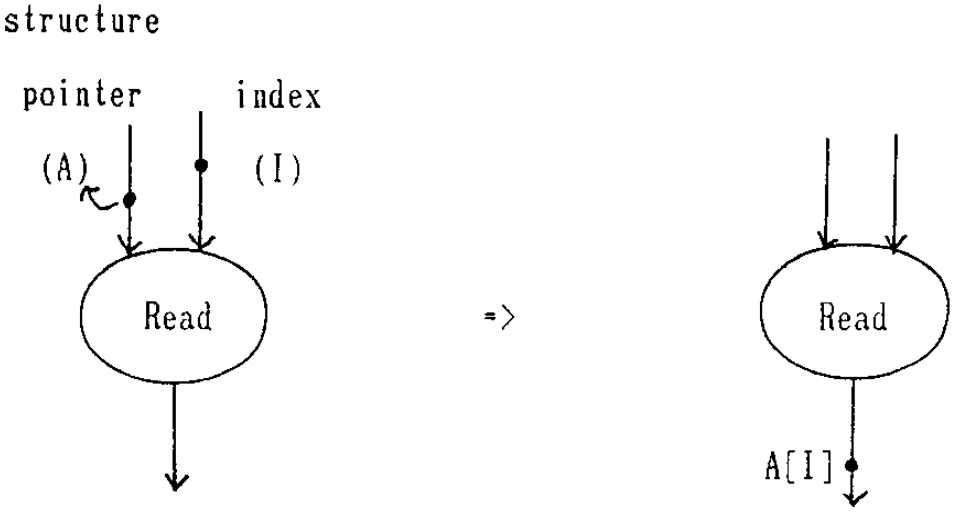
입력 I-structure의 크기를 출력한다.

c. EI-structure-1 (Extented I-structure-1 or E1)

E1은 I-structure가 갖는 histogram 문제를 해결하기 위하여 제안된 것으로 E1과

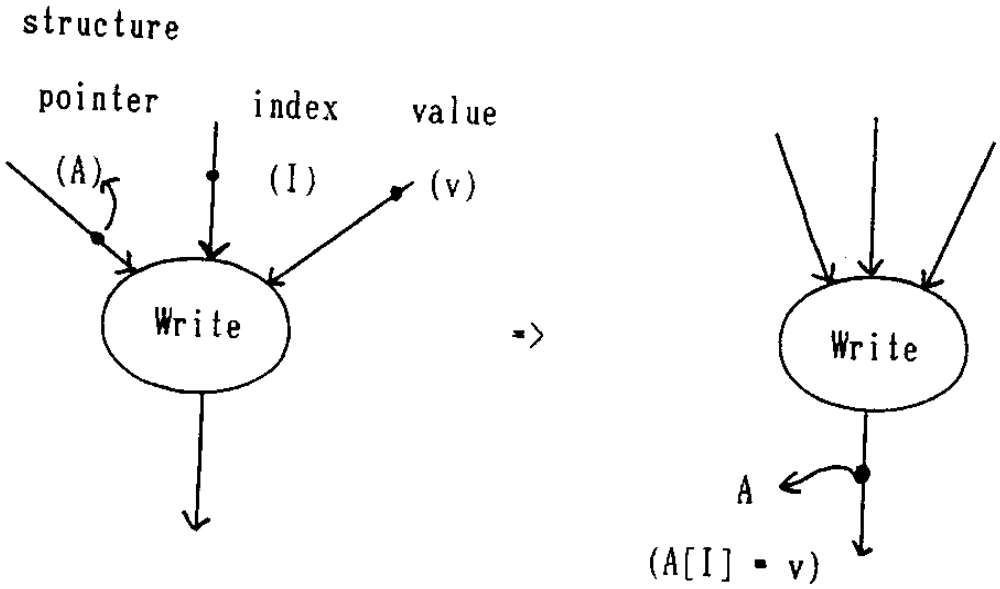


i. 할당 노드 (Allocation or Alloc node)

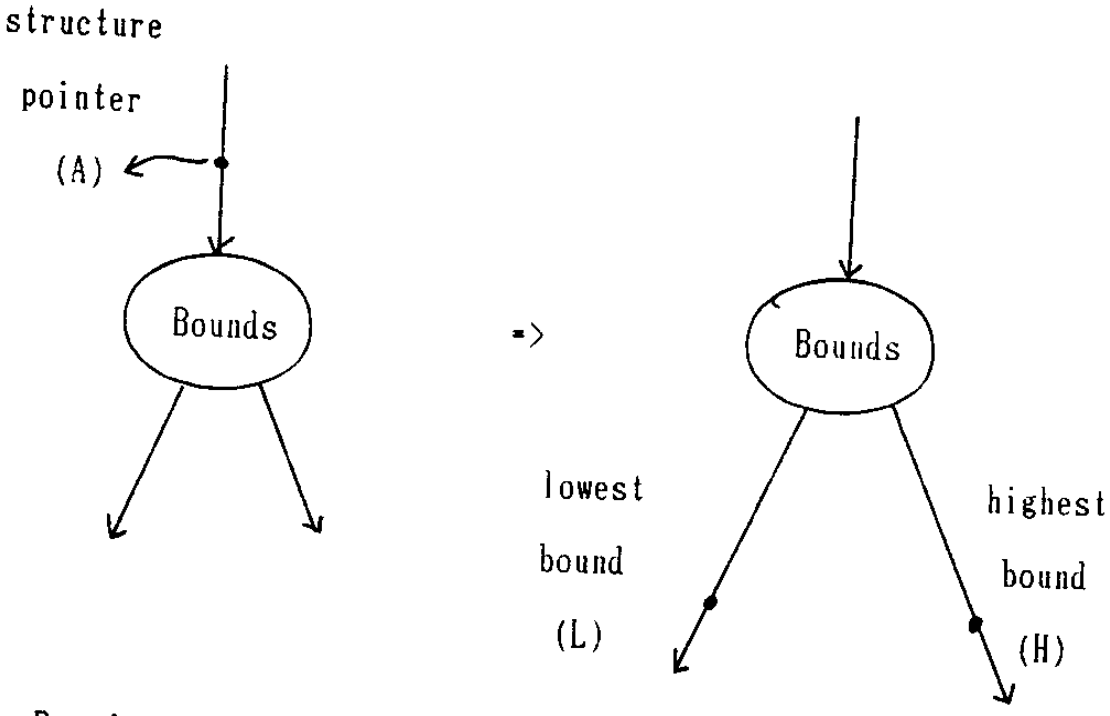


ii. 읽기 노드 (Read node)

그림 4.6 I-structure 계산 단위의 실행 법칙 (다음 장에 계속)

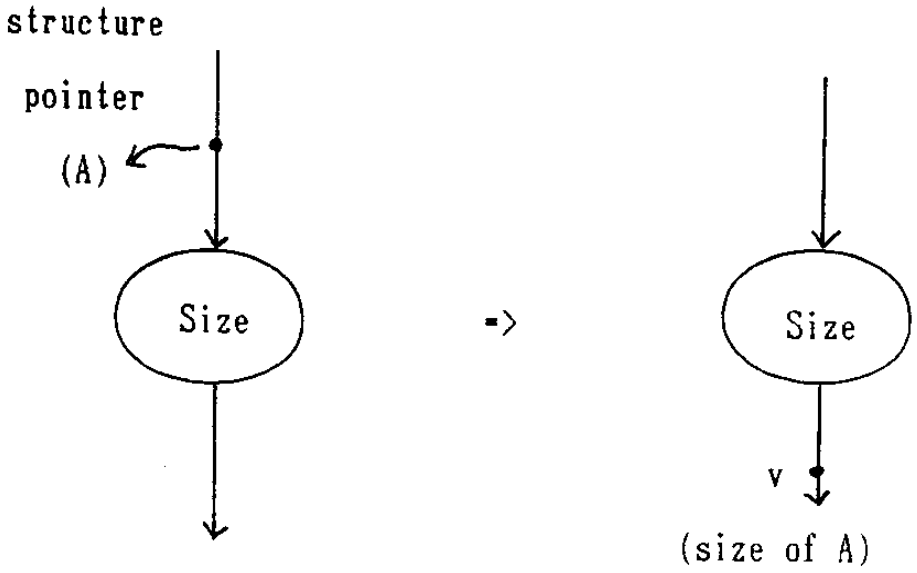


iii. 쓰기 노드 (Write node)



iv. Bounds 노드 (Bounds node)

그림 4.6 I-structure 계산 단위의 실행 법칙 (다음 장에 계속)



v. 크기 노드 (Size node)

그림 4.6 I-structure 계산 단위의 실행 법칙

관련된 계산단위는 다음과 같고, 실행법칙은 그림 4.7에 제시되어 있다.

i. 할당 노드(Allocation or Alooc1 node)
가장 낮은 인덱스가 L이고 가장 높은 인덱스가 H이며 원소 생성에 간여하는 계산단위가 K 개인 E1 구조의 배열을 할당하여 그것에 대한 포인터를 출력한다.

ii. 읽기 노드 (Read1 node)
입력 E1 구조 배열의 입력 인덱스에 해당하는 값을 출력한다(주의해야 할 점은 counter 값이 0인 경우에만 해당 인덱스의 값이 출력된다는 것이다).

iii. 쓰기 노드 (Write1 node)
입력 E1 구조 배열의 입력 인덱스에 입력 값을 쓰고, counter 값을 하나 감소시키고 그 구조에 대한 포인터를 출력한다.

iv. Bounds 노드 (Bound1 node)
입력 E1 구조 배열의 가장 낮은 인덱스와 가장 높은 인덱스를 출력한다.

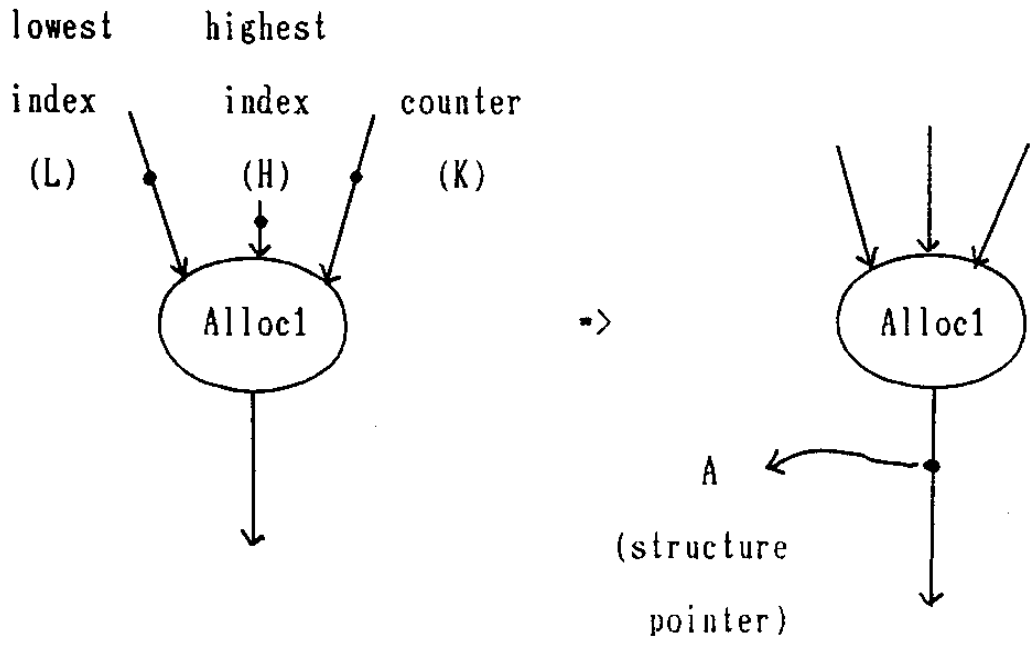
v. 크기 노드 (Size1 node)
입력 E1 구조 배열의 크기를 출력한다.

d. EI-structure-2 (Extended I-structure-2 or E2)

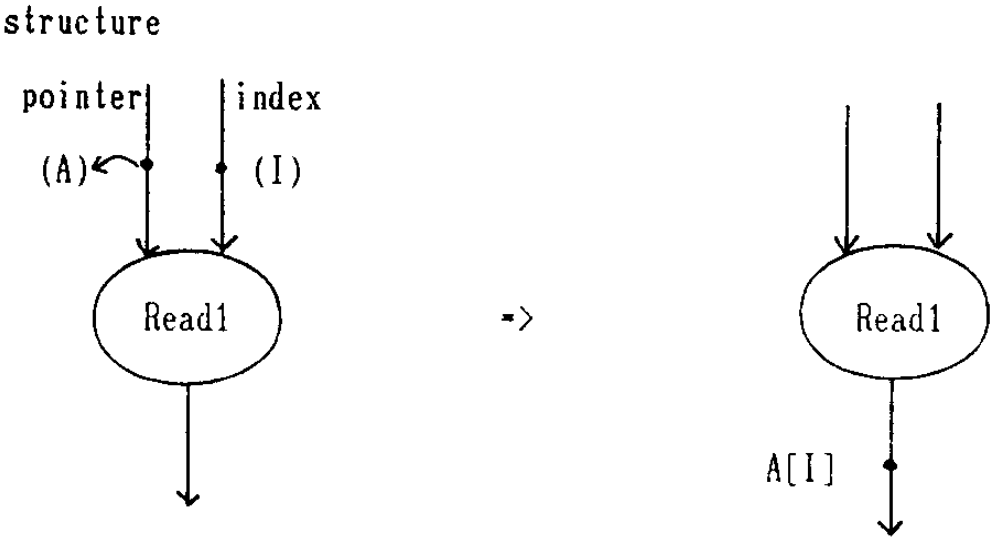
E2는 I-structure가 갖는 polynomial 문제를 해결하기 위하여 제안된 것으로 E2와 관련된 계산단위는 다음과 같고, 실행법칙은 그림 4.8에 제시되어 있다.

i. 할당 노드 (Allocation2 or Alloc2 node)
가장 낮은 인덱스가 L이고 가장 높은 인덱스가 H이며 각 원소 생성에 간여하는 계산단위가 K 개인 E2 구조의 배열을 할당하여 그것에 대한 포인터를 출력한다.

ii. 읽기 노드 (Read2 node)
입력 E2 구조 배열의 입력 인덱스에 해당하는 값을 출력한다(주의해야 할 점은 각 원소의 counter 값이 0인 경우에만 해당 인덱스의 값이 출력된다는 것이다).

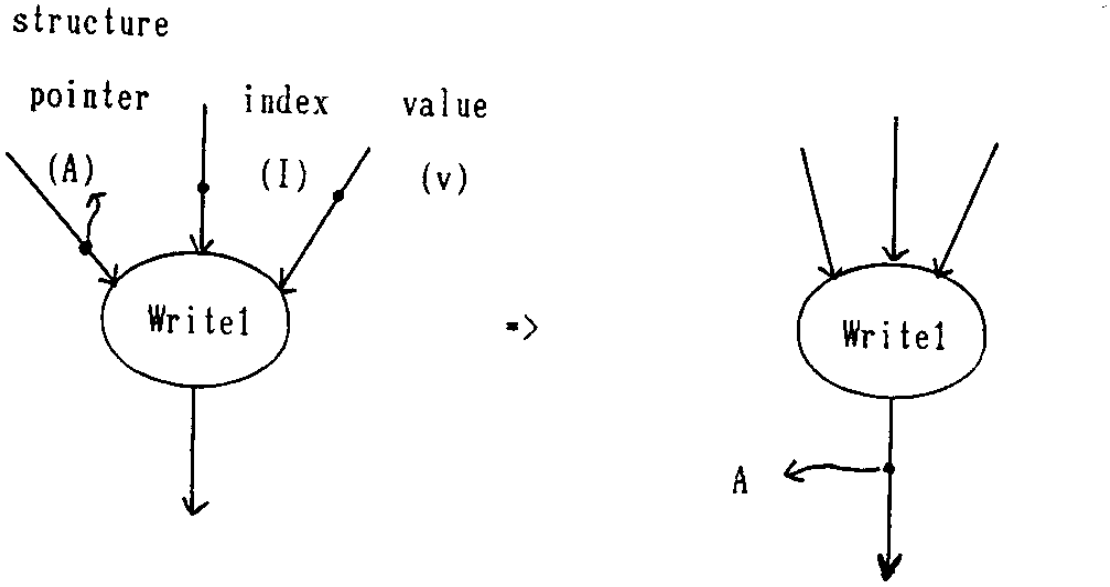


i. 할당 노드 (Allocation1 or Alloc1 node)

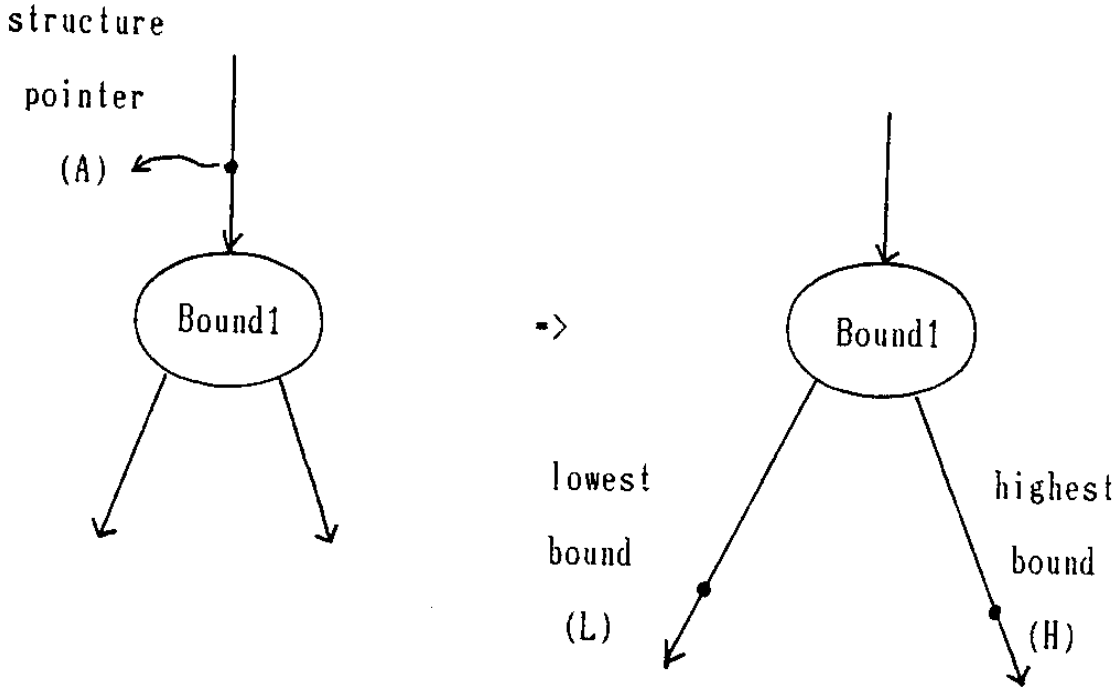


ii. 읽기 노드 (Read1 node)

그림 4.7 El-structure-1 계산 단위의 실행 법칙 (다음 장에 계속)

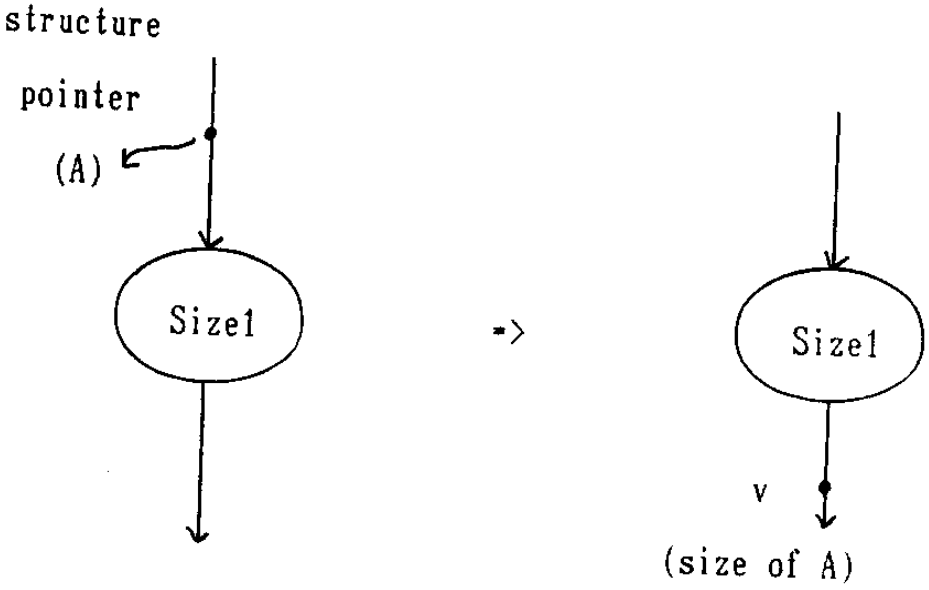


iii. 쓰기 노드 (Write1 node)



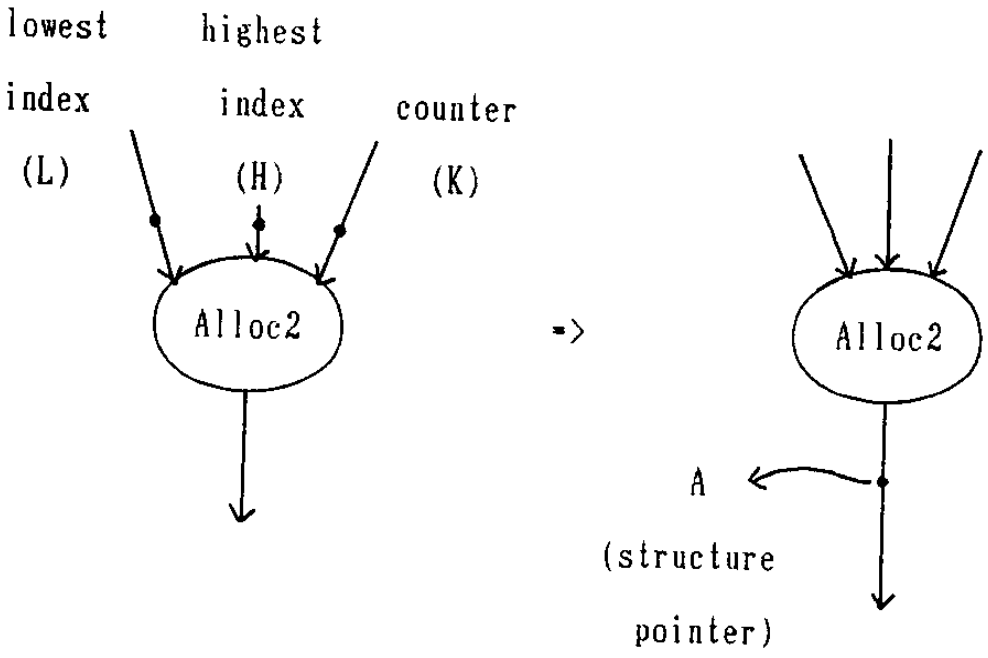
iv. Bounds 노드 (Bound1 node)

그림 4.7 EI-structure-1 계산 단위의 실행 법칙 (다음 장에 계속)

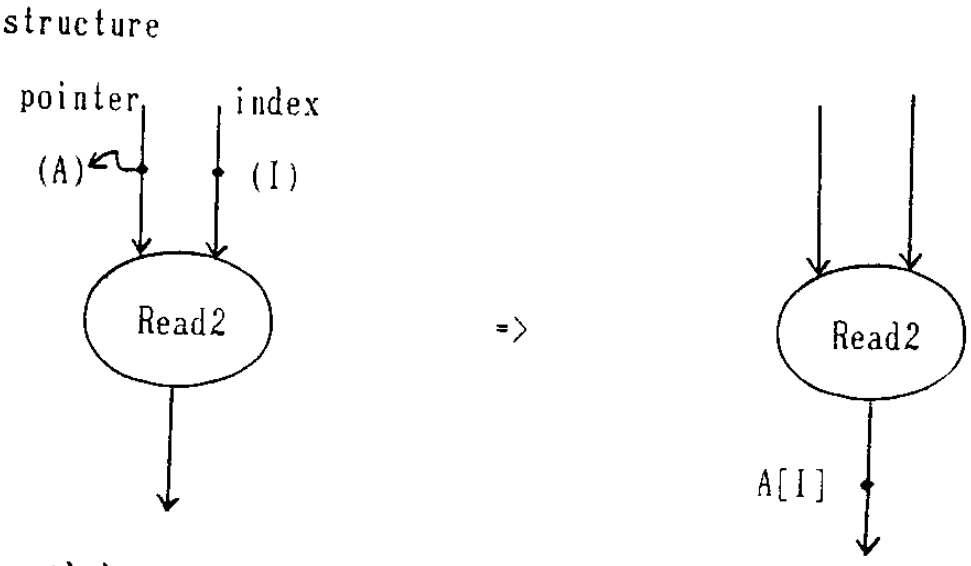


v. 크기 노드 (Size1 node)

그림 4.7 EI-structure-1 계산 단위의 실행 법칙

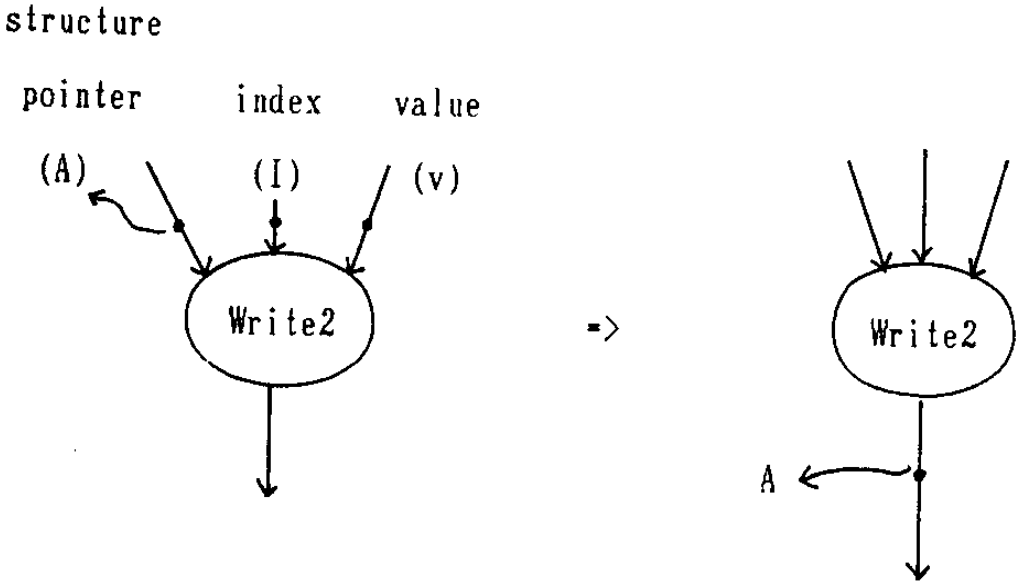


i. 할당 노드 (Allocation2 or Alloc2 node)

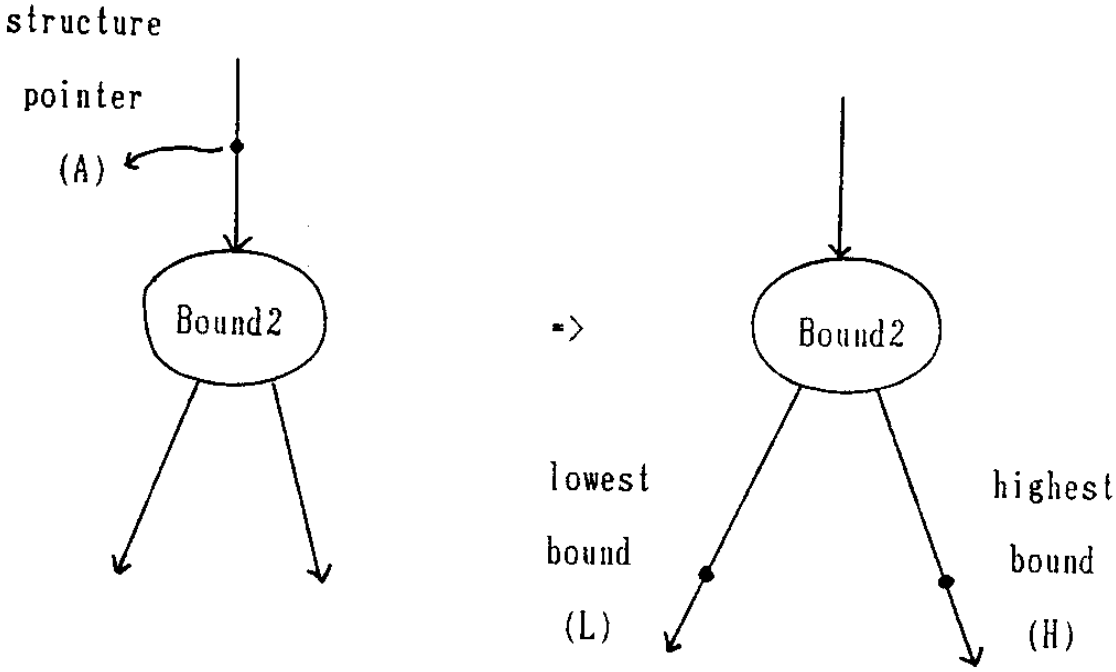


ii. 읽기 노드 (Read2 node)

그림 4.8 EI-structure-2 계산 단위의 실행 법칙 (다음 장에 계속)

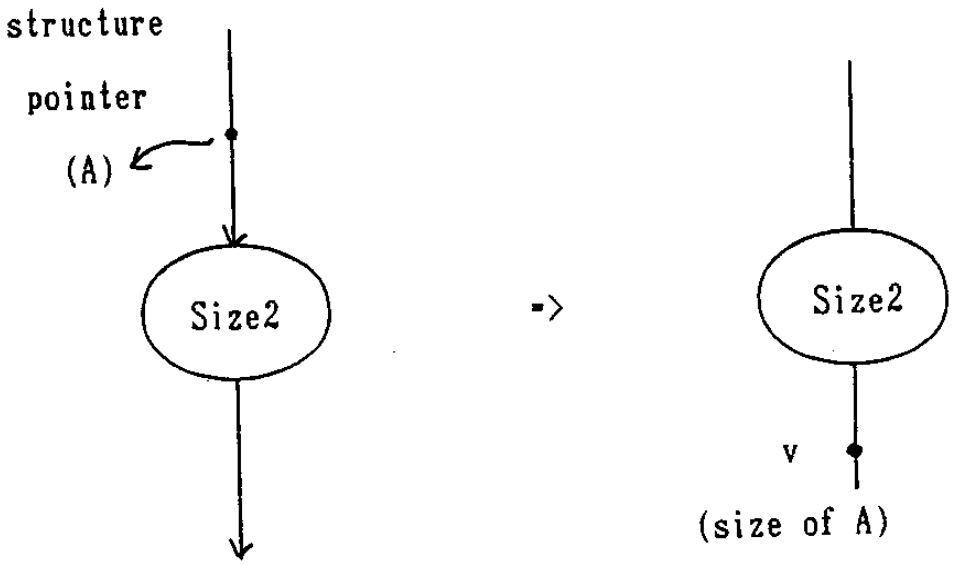


iii. 쓰기 노드 (Write2 node)



iv. Bounds 노드 (Bound2 node)

그림 4.8 EI-structure-2 계산 단위의 실행 법칙 (다음 장에 계속)



v. 크기 노드 (Size2 node)

그림 4.8 EI-structure-2 계산 단위의 실행 법칙

iii. 쓰기 노드 (Write2 node)

입력 E2 구조 배열의 입력 인덱스에 입력 값을 쓰고, counter 값을 하나 감소시키고 그 구조에 대한 포인터를 출력한다.

iv. Bounds 노드 (Bound2 node)

입력 E2 구조 배열의 가장 낮은 인덱스와 가장 높은 인덱스를 출력한다.

v. 크기 노드 (Size2 node)

입력 E2 구조 배열의 크기를 출력한다.

(2) 스트림/리스트 연산 (Stream/List Operations)

스트림이란 값의 순서화된 sequence로, 프로그램의 두 모듈(생산자와 소비자) 간에 순차적으로 생성/소모되는 데이타에 해당한다. 즉 데이타 플로우 프로그램에서 어떤 계산 단위의 집합이 일련의 토큰을 생성하면 다른 계산 단위의 집합이 같은 순서로 그 토큰들을 소모할 경우이다. 만일 데이타 토큰이 생성된 순서와 동일한 순서로 소모된다면 스트림은 구조적 자료를 대신할 수 있다. 주의할 사항은 스트림 내의 토큰의 시간적 순서가 의미가 있으므로 스트림은 history sensitive한 특별한 construct라는 점이다. 이러한 특성은 연산의 시간 의존성 때문에 부수효과(side-effect)가 발생하는 입출력의 경우에 있어 특히 유용하다. 이러한 이유로 base 언어에서도 스트림을 포함시켰다.

스트림은 구조적 자료의 특수한 경우로 볼 수 있지만 다음의 2가지 특성도 가지고 있다. 먼저 생성 순서와 소모 순서가 같다는 점이고 다음으로 구조의 크기가 indeterminate하여 수행 시간까지 알 수 없고 수행 시간때 End-of-Stream(EOS) 토큰에 의해 그 끝을 알 수 있다는 점이다.

토큰은 이러한 스트림 값을 가질 수 있는데 이 경우 스트림은 소모될 때까지는 임시로 특별한 기억 장치에 저장되어 있고 그 포인터 값을 토큰이 값으로 갖는다. 그림 4.9는 스트림의 예이다.

스트림은 nonstrict 리스트로 구현될 수 있으며, 스트림에 대한 연산자는 LISP에서 리스트 연산에 사용되는 연산자와 비슷하다. 먼저 []는 길이가 0인 스트림을 나

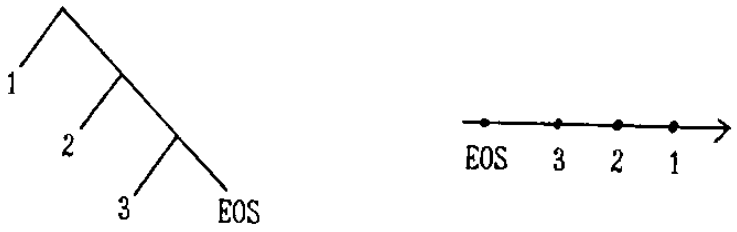


그림 4.9 스트림 [1, 2, 3] 의 표현

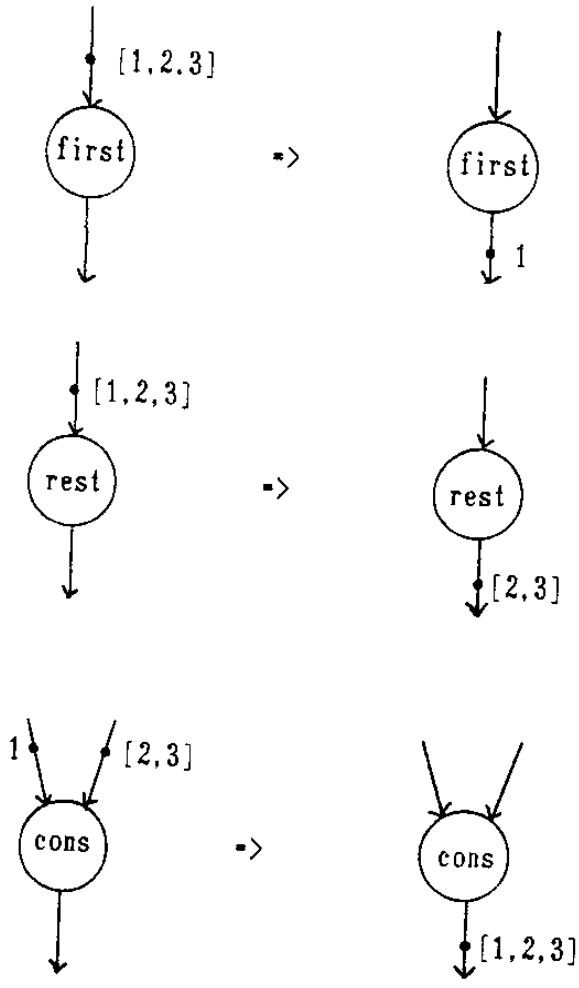


그림 4.10 스트림 연산자의 표시법과 예(다음 장에 계속)

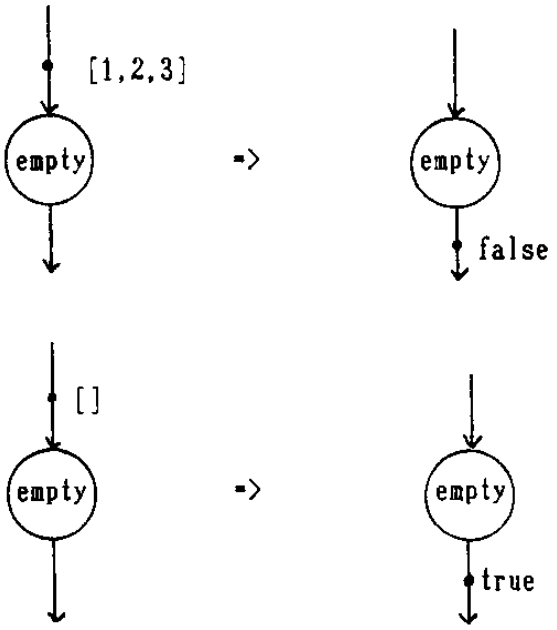


그림 4.10 스트림 연산자의 표시법과 예

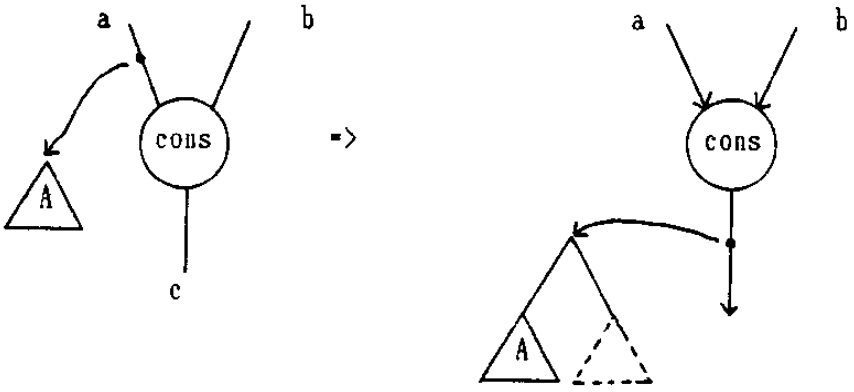


그림 4.11 nonstrict cons 연산자

타낸다. cons(c,s)는 새로운 스트림 s'를 만들어내는데 그 첫 원소가 c이고 나머지 원소는 스트림 s의 원소와 동일하다. first(s)는 스트림 s의 첫번째 원소를 결과로 갖는다. rest(s)는 스트림 s에서 첫번째 값을 제외한 나머지 스트림의 결과 값이다. cons와 first(rest) 연산 간의 관계는 다음과 같다.

```
first(cons(x,y)) ==> x
rest(cons(x,y))  ==> y
```

스트림에 대한 부울린 연산자로는 empty(s)가 있는데 s=[]이면 참 값을 결과로 내고, 이 외의 경우 false 값을 결과로 낸다. 그림 4.10은 스트림에 대한 연산자를 보여 주고 있다.

여기에서 주의해야 할 사항은 cons 연산자가 nonstrict하다는 사실이다. 즉, cons 연산자 cons(x,y)는 x와 y의 계산에 관계없이 결과값을 출력해야 하므로 cons의 결과값은 그림 4.11과 같은 불완전한 구조적 자료가 된다.

리스트에 관한 계산단위는 4.2.6 절에서 자세히 제시될 것이다.

4.2.4 데이타 플로우 모델에서의 nonstrict evaluation

Nonstrict evaluation은 lazy evaluation과 nondeterministic 계산을 실현하는데 중요하다. 데이타 플로우 모델에서는 3가지 경우의 nonstrict evaluation 계층이 있는데, 함수 적용, 수식, 기초적인 연산 계층이다 [AMHA 84].

(1) 함수 적용에서의 nonstrictness

함수 적용에서의 nonstrictness는 부분적 실행(partial computation)에 대한 함수 연결 메카니즘으로 구현된다. 함수 적용에 대한 메카니즘이 그림 4.12 에 나와 있다. 함수 적용과 관련된 계산단위에 대한 자세한 설명은 4.2.8 절에서 논의할 것이다.

그림 4.12로부터 함수 본체의 계산이 signal을 받아 시작되어 x,y,z의 값을 받을 때마다 진행됨을 알 수 있다. 따라서, 만일 함수 f가 z의 값이 정의되는 것과

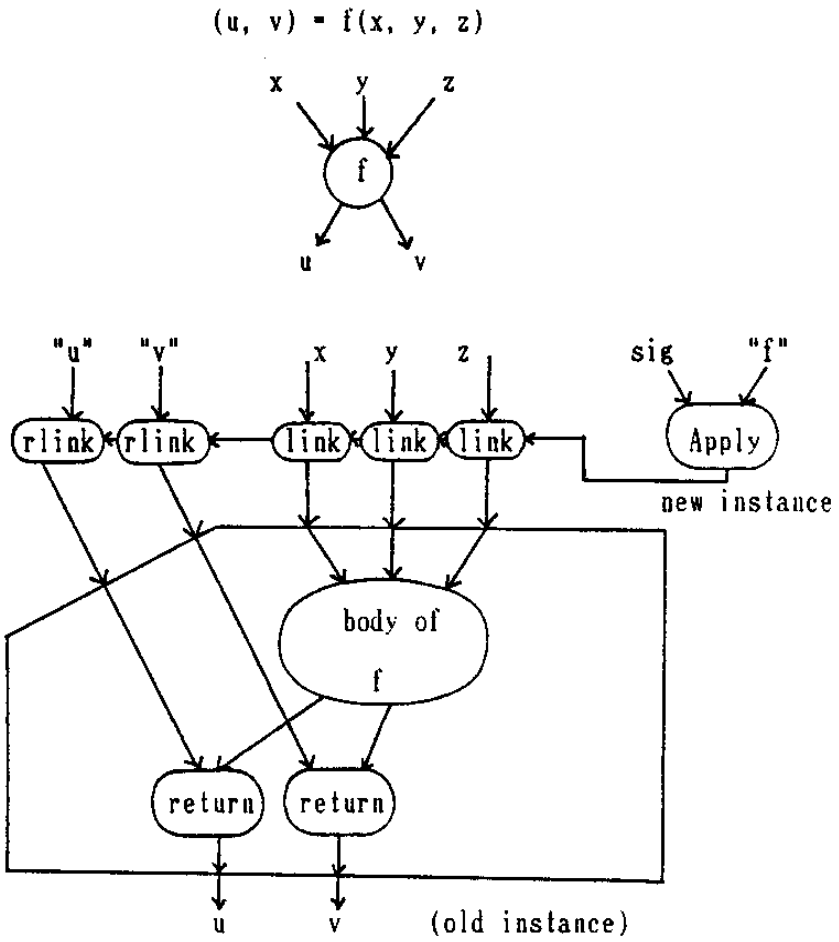


그림 4.12 함수 적용 메카니즘

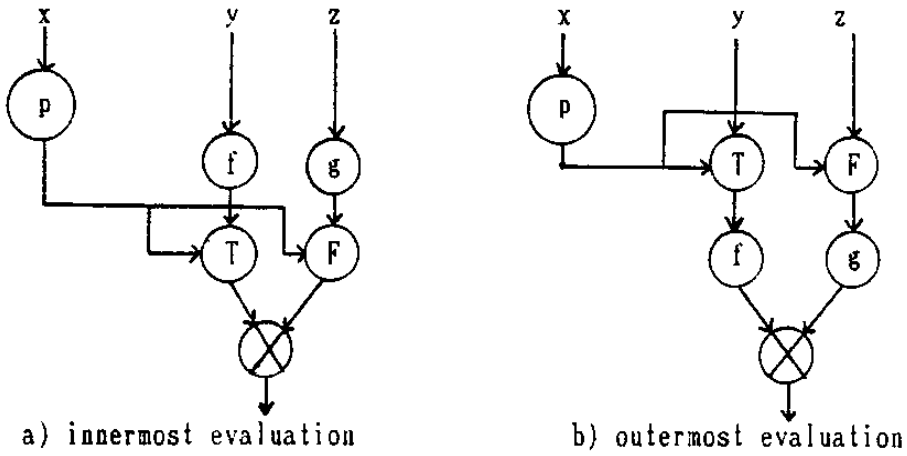


그림 4.13 if p(x) then f(y) else g(z) 에 대한 데이타 플로우 그래프

상관없이 u와 v의 값을 갖는다면, 계산 후에 그 값을 반환할 수 있다.

(2) 수식 계산에서의 nonstrictness

수식 계층에서의 nonstrict evaluation은 제어 토큰과 gate 노드(T 혹은 F 노드)를 사용하여 데이타 플로우 그래프에 직접 표현된다.

예를 들어 조건부 수식

```
if p(x) then f(y) else g(z)
```

에 대한 데이타 플로우 구현이 아래 그림 4.13에 나와 있다. 이 때 predicate 노드 p는 switch 연산을 제어하는 불린 토큰을 생성한다.

그림 4.13 a)는 nonstrict eager evaluation의 구현이고 그림 4.13 b)는 nonstrict lazy evaluation의 구현이다. 양쪽 모두 p(x)가 참(혹은 거짓)이면 g(z) 혹은 f(y)가 값을 갖는지 여부에 상관없이 결과는 f(y) 혹은 g(z)의 값이 된다(여기서 주의해야 할 점은 X 노드는 어느 한쪽의 입력 값을 출력 arc로 단순히 전달하는 역할을 수행하고, 실제의 데이타 플로우 그래프 상에는 존재하지 않는다는 것이다. 즉, X 노드의 입력 값들은 직접 X 노드의 출력 arc 상으로 보내지도록 되어 있다).

(3) Nonstrict한 기초적인 연산

구조적 자료에 대한 nonstrict eager/lazy evaluation과 비결정형(nondeterministic) 계산을 수행하기 위해서는 2개의 nonstrict한 기초적인 연산자가 필요하다.

첫번째 기초적인 연산자는 cons 연산자이다. 앞에서 설명한 바와 같이 cons의 정의는 다음과 같다.

```
car(cons(x,y)) => x
cdr(conds(x,y) => y
```

또 다른 nonstrict한 기초적인 연산자는 arbiter이다. 이 arbiter는 nonstrict-merge, nonstrict-or, nonstrict-and, guarded command 등과 같은 비결정형 계산을 구현하는데 있어서 필수적인 연산자이다.

4.2.5 Eager evaluation과 lazy evaluation의 효율적 구현

데이타 플로우 모델에서는 여러 종류의 데이타가 토큰에 지정된다. 가장 기본적인 데이타 플로우 모델은 정수, 실수, 부울린, 구조적 자료 등의 데이타 자체를 토큰에 지정한다. 이러한 계산 방식을 by-value 방식이라 한다. 보다 발전된 데이타 플로우 모델은 값을 지니고 있는 cell, 구조화 자료, 그리고 프로그램 코드에 대한 포인터를 토큰에 지정한다. 이러한 방식을 by-reference라 한다. 보다 편리한 계산 제어를 제공하는 훨씬 더 복잡한 데이타 플로우 모델은 by-value 방식과 by-reference 방식을 조합한 것이다 [AMHA 84].

Eager evaluation과 lazy evaluation을 가장 효율적으로 구현하는 방법은 by-value 방식과 by-reference 방식을 선택적으로 사용하는 것이다. Eager/lazy evaluation을 혼용해서 사용하는 경우에 eager/lazy evaluation할 부분을 결정해야 하는 문제가 있는데, 대체로 세 가지 방법이 있다.

(1) 프로그래밍 언어 시스템에서 고정적으로 결정되는 방법

우리는 흔히 lazy evaluation 방식으로 처리하는 것이 효율적인 곳을 어느 정도 알고 있고, 그러한 부분을 사용자와 시스템 간에 묵시적으로 약속할 수 있다. 예를 들면 if_then_else나 함수의 인자, 정의된 표현 등은 demand-driven 방식으로 계산을 행하는 것이 효율적일 수 있고, 그래서 프로그램 상의 모든 그러한 부분을 demand-driven 방식으로 계산을 행하고 그 외의 부분은 eager evaluation 방식으로 계산을 행하는 방법을 생각할 수 있다. 이 방법은 사용자가 lazy evaluation 방식의 계산을 표시해 줄 필요가 없고 컴파일러의 복잡도가 상당히 감소되나 필요 이상으로 lazy evaluation 방식의 계산을 요구함으로써 병렬 수행의 이득을 얻지 못할 수 있고 프로그램 수행시 동적인 수행 양식을 제어할 수 없는 단점이 있다. 그러나 이 방법은 가장 생각하기 쉽고 구현하기 쉬우며 뒤에서 언급될 다른 두 가지 방법의 구현에 기초가 되기 때문에 본 절에서는 우선 이 방법에 대해 살펴 보고자 한다.

Cell이 값의 정의(value definition)나 함수 인자에 의해 정의된 값의 이름에만

할당된다고 하자. 각 cell은 r과 c의 2개의 태그(tag)를 갖는데 이 태그들은 eager evaluation과 lazy evaluation 제어에 사용된다. 각 cell의 내용은 계산된 값이거나 recipe라 불리우는 프로그램 코드에 대한 포인터이다. 태그 r이 1인 경우 cell은 값을 포함하고 있으며 read 액세스가 허용된다. 태그 c가 1인 경우 cell의 내용은 recipe이다. 다음 5가지 기초적인 연산자가 cell 액세스 제어에 사용된다.

Gcell(s) : 새로운 cell을 만들어 태그 r과 c에 0 값을 지정하고 cell을 반환한다 (이 연산자는 signal 토큰 s에 의해 시작된다).

Wrcd(x,l) : 프로그램 코드 엔트리(노드 이름) l을 cell x에 write하고 태그 c 값을 1로 한 후 cell을 반환한다.

Bind(x,v) : 값 v를 cell x에 write하고 태그 r의 값을 1로 한 후 cell을 반환한다.

Repl(x,v) : cell x를 값 v로 대체한 후 cell을 반환한다.

Eval(x,s) : cell x의 값을 읽어들이는데 만일 r = 0이면 값이 write 되어지기를 기다리고, 만일 c = 1이면 recipe를 계산하라는 요구를 트리거(trigger)하고 값을 기다린다 (이 연산자는 signal 토큰 s에 의해 시작된다).

간단한 예가 그림 4.14에 나와 있는데, 이 그래프에서는 by-reference 방식이 값 이름 x에 대해 사용되었다. x의 값이 eval 연산자에 의해 요구되었을 때 요구(demand) 신호가 cell x로부터 gate 노드까지 데이타 a,b,c를 그래프에 전달하기 위해 트리거된다. 즉 수식 (a+b)*(b-c)가 by-value 방식에 의해 계산된다. 그림 4.14에서 G(gate) 노드는 제어 입력 arc 상에 signal이 올 때까지 입력 arc 상의 값이 그것을 입력으로 하는 노드로 전달되는 것을 억지시켜 주는 역할을 한다.

또 다른 예로 (u,v) = f(g(x),y,h(z))에 대한 함수 적용 구조가 아래에 있는데, 그림 4.15에서는 값 u 또는 v의 eval 연산으로부터 demand가 전송되었을 때 값 x,y,z에 대한 eval 연산이 시작된다.

한편 cell이 recipe를 포함하지 않고, 각 cell이 도착할 때 각각의 값에 대한

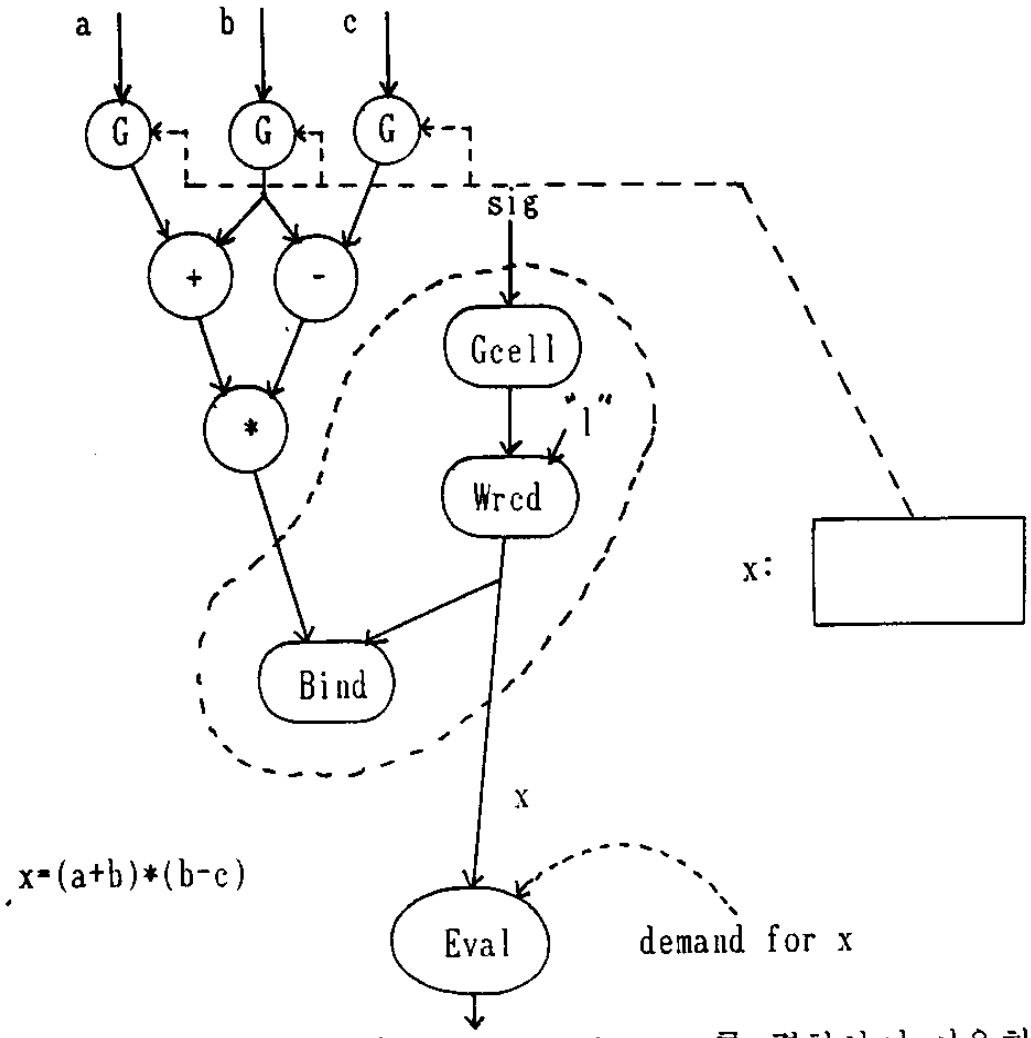


그림 4.14 call-by-value와 call-by-reference를 결합하여 사용한 예
(정의된 표현의 lazy evaluation에 대한 프로그램 그래프)

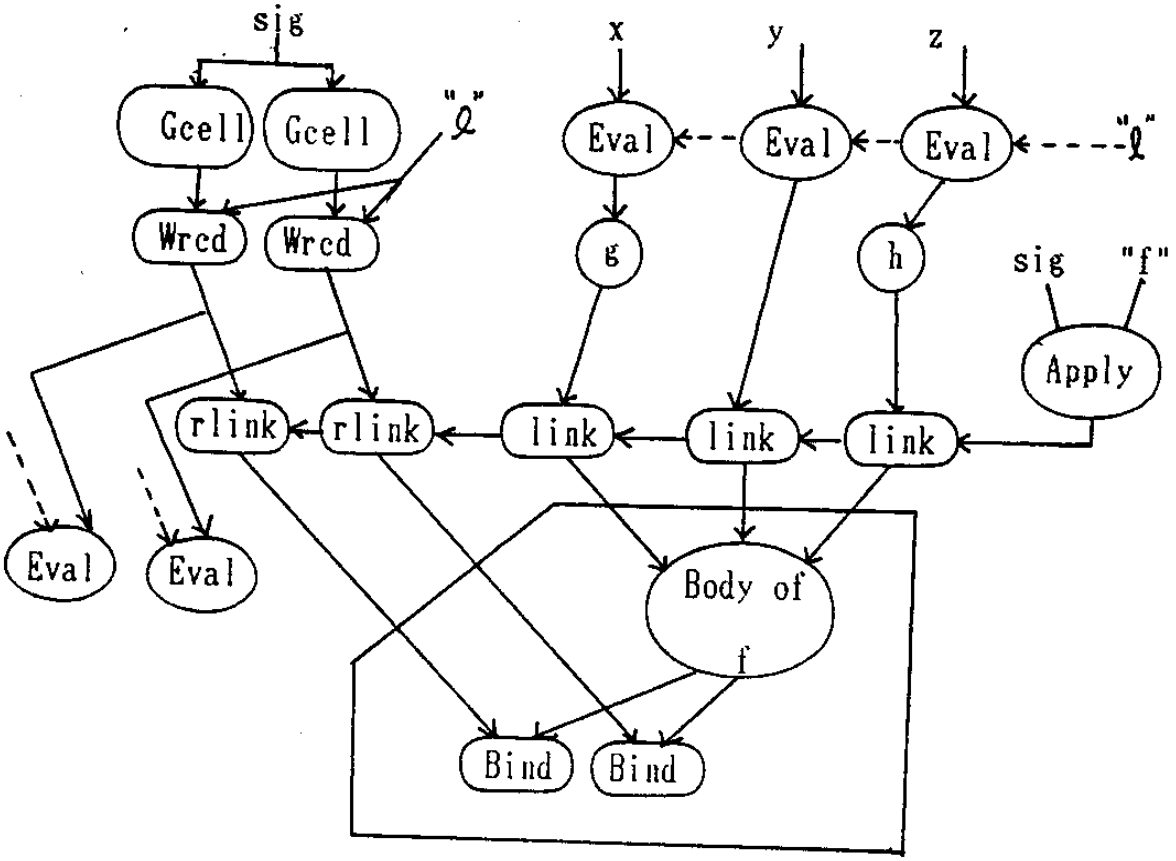


그림 4.15 call-by-value와 call-by reference를 복합한 함수 적용 메카니즘

eval 연산자가 곧 시작되도록 요구 신호가 사용된다면, 이것은 eager evaluation이 구현이 된다.

(2) 사용자가 프로그램 상에 명백하게 표시하여 주는 방법

우리는 lazy evaluation을 할 부분들을 사용자가 프로그램 상에 명백하게 표시하여 주면 그렇게 표현된 부분만 lazy evaluation하고 다른 부분들은 eager evaluation하는 방법을 생각할 수 있다. 이 방법은 사용자가 lazy evaluation할 부분을 일일히 표시하여야 하는 부담이 있으며 절에서 언급한 바와 같이 eager evaluation에서 발생하는 여러 문제가 발생할 수 있다. 따라서 사용자가 문제와 언어, 그리고 시스템에 익숙할 것을 요구한다. 앞서 언급한 프로그래밍 시스템에서 고정적으로 결정되는 방법과 뒤에 말할 컴파일러가 적절하게 결정하여 주는 방법들은 프로그램 수행시 자원의 제약이 있을 때 이를 대처하는 방안(병렬성의 제어)이 모두 시스템에 맡겨졌다. 그러나 대체로 우리는 응용에 따라 폭발적인 병렬성이 일어날 곳을 예측하여, 그러한 부분에 lazy evaluation할 것을 표시하여 줌으로써 제한된 자원하에서 효율적으로 계산을 수행해 나갈 수 있을 것이다. 따라서 특히 이 방법은 데이타 플로우 언어로 데이타 플로우 머신의 시스템 프로그램을 작성할 시에 긴요하게 사용될 것이다.

지체연산(Delaying evaluation) 과정은 함수 언어에 지체 연산자(delay)를 도입함으로써 명시적으로 나타난다. 예를 들어 수식 E(x,y,z)의 지체 연산은 다음과 같이 표현된다.

E'(delay E(x,y,z)) , 수식일 때

또는 E'(u) where u = delay E(x,y,z) , 값 정의일 때

연산의 지체와 요구에에 대한 데이타 플로우 구현이 다음에 나와 있다. 그림 4.16에서와 같이 원시(source) 언어에 표현된 지체 연산자는 컴파일된 코드인 데이타 플로우 그래프에서는 입력 값에 대한 gate(혹은 eval) 노드로 나타난다. 만일 x,y,z가 연기가 안되었다면, E(x,y,z)의 계산은 gate를 통해 데이타 x,y,z를 전달한다. 만일 x,y, 또는 z가 연기되었다면 gate 연산자 대신 eval 연산자가 사용된다. 만일

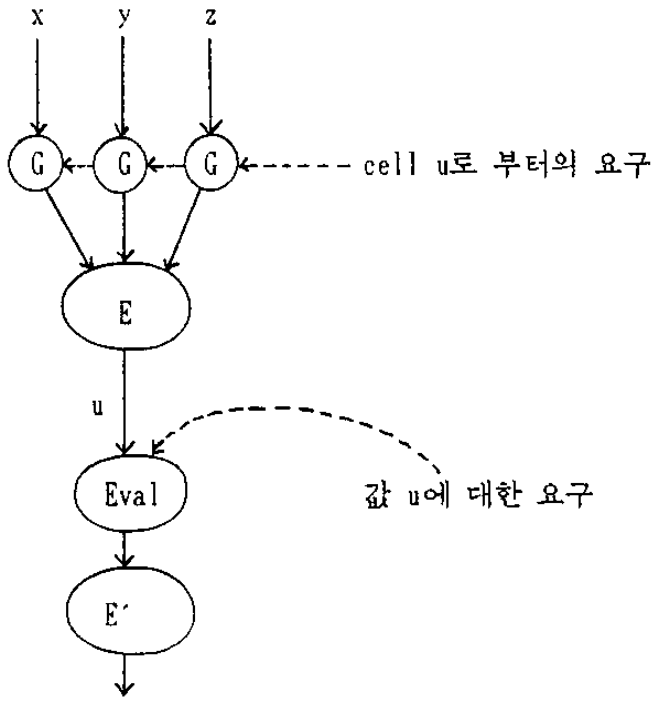


그림 4.16 E'(u)에 대한 데이타 플로우 그래프(u*delay E(x,y,z))

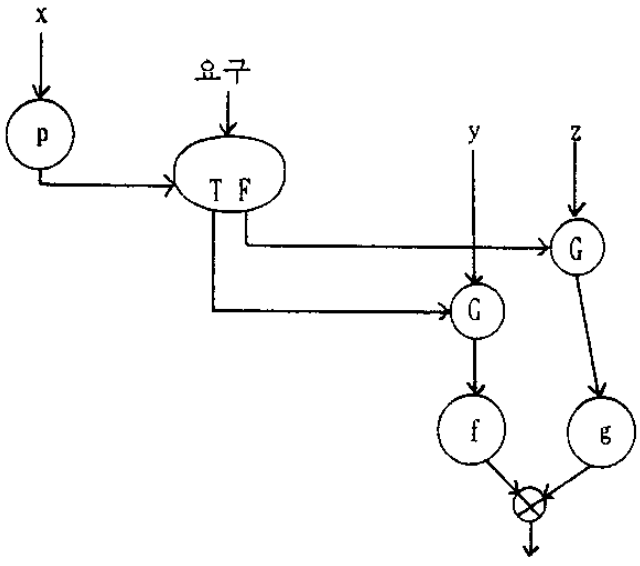


그림 4.17 if p(x) then delay f(y) else delay g(z)에 대한 데이타 플로우 그래프

지체된 연산이 중첩되었다면 gate를 여는 요구가 안쪽 값의 이름으로 전달된다.

이러한 전형적인 예가 조건부 수식의 계산이다.

```
if p(x) then delay f(y) else delay g(z)
```

는 그림 4.17과 같은 데이타 플로우 그래프로 변형된다. Call-by-need 또한 지체 연산자를 이용하여 나타낸다. 예를 들어 함수 F가 다음과 같이 정의되었다면

```
function F(u) = E'(u) ,
```

call-by-need는 함수 정의의 다음과 같은 형식 인자

```
function F(delay u) = E'(u)
```

와 함수 호출의 실인자 F(delay E(x,y,z))로 나타난다. 컴파일러는 실인자에 자동적으로 지체 연산자를 삽입한다.

만일 조건부 수식이 함수 적용 ff(p(x),f(y),g(z))에 대해 다음과 같이 함수 ff로 정의되었다면

```
function ff(x, delay y, delay z) = if x then y else z
```

수식 f(y)와 g(z)의 계산은 함수 본체 ff의 조건부 수식 계산 도중에 요구되어 질 때까지 연기된다. 함수 ff에 대한 데이타 플로우 그래프가 call-by-need 구현의 예로써 그림 4.18에 나와 있다.

(3) 컴파일러가 적절하게 결정하여 주는 방법

사용자가 프로그램 상에 명백하게 표현해 주는 방법은 사용자가 문제와 언어에 능숙하고, lazy evaluation할 부분을 정확하게 표시해야 하는 부담이 있고, 프로그래밍 언어 시스템에서 고정적으로 결정되는 방법은 lazy evaluation을 과다하게 요구함으로써 병렬성을 상실할 수 있기 때문에 우리는 컴파일러가 프로그램의 분석을 통해 lazy evaluation할 부분을 찾아 내고, 불필요한 요구 전달과 recipe의 생성을 제거하는 방법을 생각할 수 있다. 그러나 이 방법은 프로그램을 컴파일링하는 것이 앞선 방법보다 복잡하고 프로그램의 정적인 분석만을 행하기 때문에 그 외에 프로그램 수행시에 병렬성을 제어하는 것은 시스템에 맡김으로써 제한된 자원하에서의 병렬성을 제어하는 데에는 미흡하다. 그러나 이 방법은 앞선 방법과의 결합을 통해

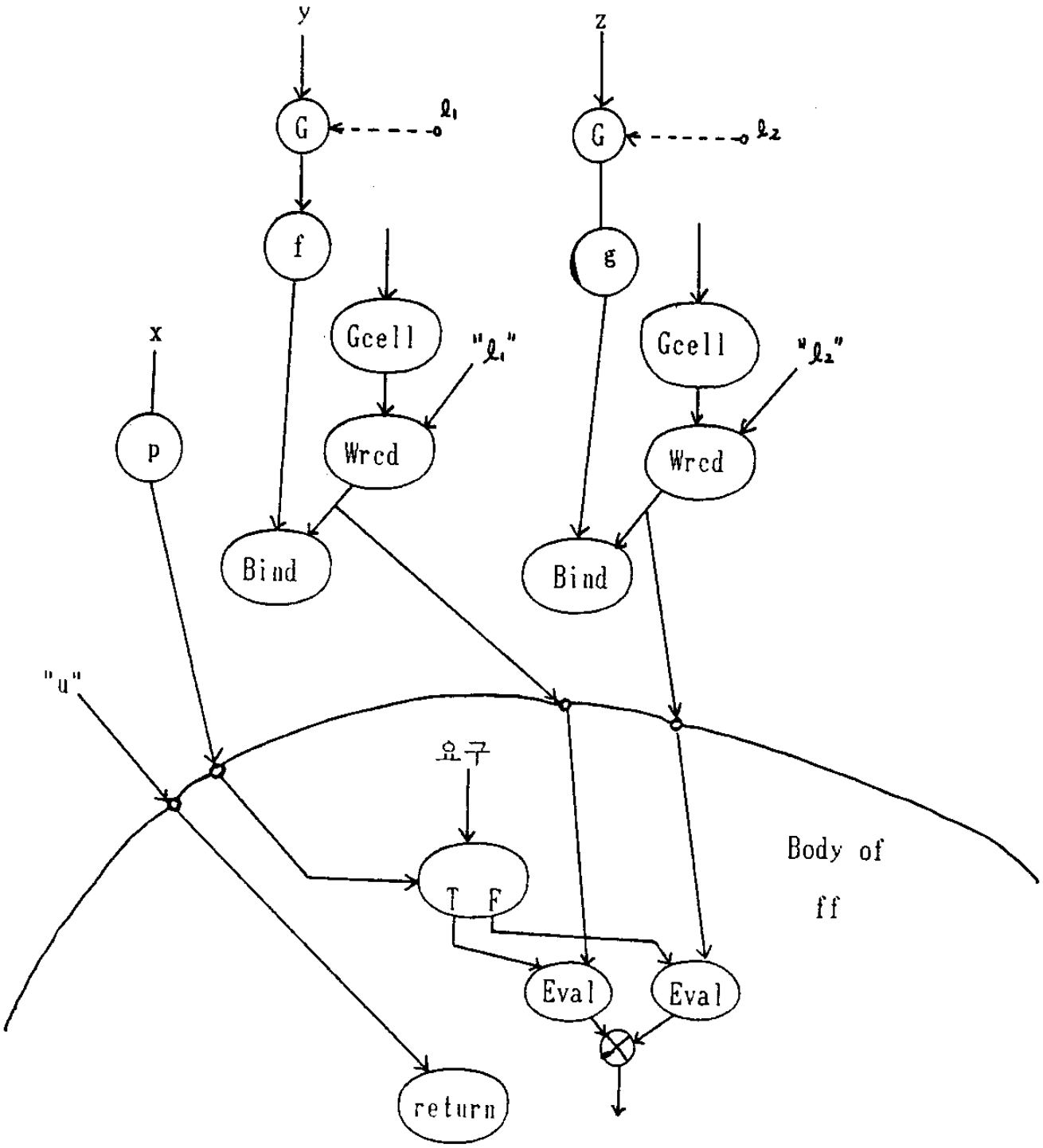


그림 4.18 함수 ff 내에서 call-by-need에 의한 데이타 플로우 구현

보다 효율적인 eager/lazy evaluation의 구현을 가능하게 할 수 있을 것이다.

Eager/lazy evaluation의 구현을 위해 제시된 세 가지의 방법은 각기 단점 및 한계가 있다. 본 연구에서는 이러한 세 가지 방법의 장점을 살리고 단점을 제거하는 방법을 제시한다. 제시된 방법은 아래와 같이 두 단계를 수행한다.

단계 1) 사용자는 병렬성의 제어와 효율적인 계산을 충분히 감안하여 프로그램상에 lazy evaluation할 부분을 표시한다.

단계 2) 컴파일러는 global data flow 분석이나 strictness 분석을 통해서 lazy evaluation할 부분들을 결정하고 불필요한 요구의 전달과 recipe의 생성을 제거해서 최적의 코드를 생성한다. 컴파일시에 컴파일러는 사용자가 프로그램상에 명시한 부분을 우선적으로 처리한다.

제시된 방법을 사용하게 되면 제한된 자원하에서 적절히 병렬성을 제어할 수 있게 되며, non-strict 연산과 같이 결과의 생성에 꼭 필요하지 않을 가능성이 있는 경우나 무한자료구조를 처리하는 경우에는 lazy evaluation을 행하도록 하고 또한 불필요한 요구의 전달 및 recipe의 생성을 피함으로써 고도의 병렬성과 안전한 계산을 보장 받을 수 있다.

4.2.6 Lenient cons와 lazy cons의 구현

구조적 자료의 조작에 by-value 모델과 by-reference 모델이 고려될 수 있지만, 리스트가 직접 토큰에 지정되는 by-value 모델은 실제적으로 부적합하다.

By-reference 구조에서는 리스트가 structure 메모리에 저장되어 있고 데이타에 대한 포인터가 토큰으로 전달된다. 그러면 cons 연산자는 structure 메모리에 cons cell을 생성하고, 그 cell을 가리키는 토큰을 출력한다. Car(또는 cdr) 연산자는 피연산자 토큰이 가리키는 cell의 car(또는 cdr) field를 읽어 들인다. 이 모델은 nonstrict cons 연산을 구현할 수 있는데, 각 cons 연산자가, cell이 car(또는 cdr) 값을 가지는지 여부에 상관없이, 결과 값으로 cons cell을 출력할 수 있기 때문이다.

By-reference 구조는 또한 리스트 처리의 2가지 구현을 가능하게 하는데, lenient cons라 불리우는 innermost eager evaluation과 lazy cons라 불리우는 outermost lazy evaluation이 그것이다.

Lenient cons에서, cons 연산자는 시작 신호의 도착에 의해 작동이 시작되어, car(또는 cdr) 값의 계산이 진행되는 동안, 결과 cons cell이 출력된다. 한편 lazy cons에서는 car(또는 cdr) 값이 car(또는 cdr) 연산자에 의해 요구되어 질 때만 계산되어 진다. read와 write 액세스를 제어하기 위해, 이 두 가지 구현에서도 r과 c의 두 태그가 사용된다. 전형적인 lazy cons 구현이 그림 4.19에 나와 있다. 만일 x와 y가 계산되어져야 한다면, eval 노드는 gate노드로 대체될 것이다. Lenient cons 구현은 이 그래프에서 Wcarcode,Wcdrcode와 2개의 gate 노드가 없는 것이다.

4.2.7 Nondeterminate 계산의 구현

Nondeterminate 프로그램을 구현하는 데는 nonstrict한 기초적 연산자 arbiter를 이용한다. 예를 들어 nonstrict merge 연산은 다음과 같이 적용된다.

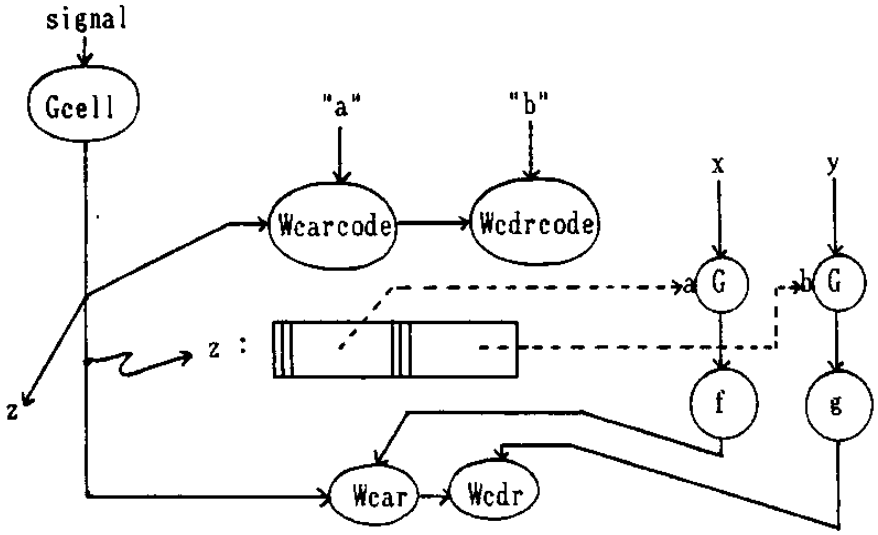
```
function merge(x,y) = cons(a,merge(v,w))
            where {(u,v) = Arb(x,y), a = car(u), w = cdr(u)}
```

이 merge 함수는 2개의 스트림을 하나로 merge한다. 이 함수은 nondeterminate 수행을 나타내는데, 스트림 원소가 x에서 오든지 y에서 오든지 상관없이, 먼저 도착한 스트림 원소가 나중에 도착한 스트림 원소보다 앞에 놓이기 때문이다.

Nondeterminate 계산의 또 다른 예는 guarded command의 구현이다. 다음의 case 수식은,

```
case { Gd1(y1) -> E1(x1),
       Gd2(y2) -> E2(x2),
         ...
       Gdn(yn) -> En(xn) }
```

guarded condition Gdi(yi), i=1,2,...,n를 만족시키는, 즉 Gdi(yi)가 계산되어 참값을 갖는 guarded 수식, Gdi(yi) -> Ei(x) 중의 하나가 선택되어, Ei(x)의 값이 결



z = lazycons(f(x),g(y))

cell : | r | c | car field | | r | c | cdr field |

태그 r : 값을 적용할 수 있는 가를 표시

태그 c : recipe 인가를 표시

그림 4.19 lazycons 구현

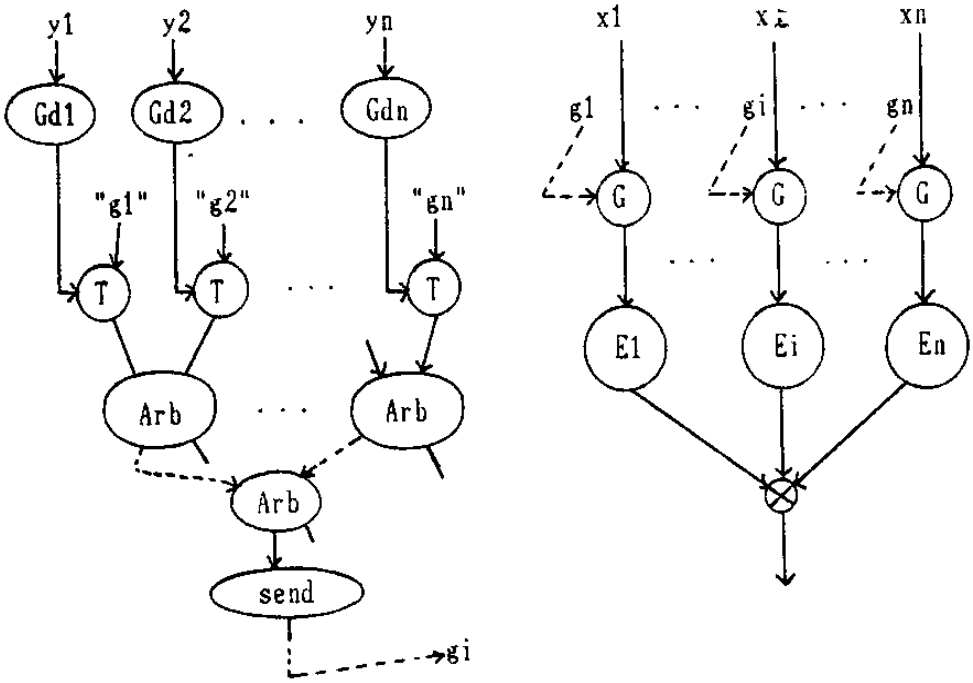


그림 4.20 guard의 구현

과로 반환됨을 의미한다. 이러한 수식의 데이타 플로우 구현이 그림 4.20에 나와 있다. 그래프에서 send 노드는 그 피연산자가 지정하는 노드에 signal을 보낸다.

4.2.8 함수의 정의와 적용 (Function Definition and Application)

(1) 함수 정의 (Function Definition)

토큰은 함수 정의를 값으로 가질 수 있는데 이 경우도 앞의 구조화 자료처럼 그에 대한 포인터를 값으로 토큰에 지정하는 것이다. 두 수 중에 최소값을 구하는 함수는 다음과 같이 정의될 수 있다.

```
min =  [(x,y) | if x<y  then  x  else  y]
```

위의 식은 함수 정의의 예인데, 여기서 min은 함수 명칭이고 식의 오른쪽 부분을 함수라 한다. x와 y는 형식 인자(formal parameters) 이고, if 부분을 함수 본체라 한다. 함수의 계산은 함수 적용과 단순화(simplification)로 이루어진다. 함수 적용은 함수 명칭을 함수 본체로 대체하고, 함수 본체 내의 형식 인자를 실인자로 대체하는 것으로, beta reduction 법칙에 해당한다. 어떤 함수는 기초적이며 공리(axiom)로 정의된다. 기초적인 함수 적용을 그 결과값으로 대체하는 것을 단순화라 한다. 데이타 플로우 그래프에서 단순화 계산은 연산자 노드로 나타나며 함수 적용의 역할을 하는 계산 단위가 apply 계산 단위이다.

그림 4.21은 apply 계산 단위의 개념적인 모습이며 실제로 데이타 플로우 그래프에서 단순화를 행할 때의 수행 환경을 번화시켜 주어야 한다. 이에 필요한 계산 단위에 대한 자세한 설명은 다음 절에서 다룰 것이다(주의하여야 할 점은 그림 상의 arg는 4.2.5에서와 같이 몇 개의 개별적인 arc를 통해서 표현될 수도 있으며 (부분적 함수 적용), 하나의 구조화 자료로 표현될 수도 있다(exportable 함수 적용)는 것이다).

(2) 함수 적용 (Function Application)

본 절에서는 exportable 함수 적용과 부분적인 함수 적용에 관련된 계산단위들에 대해 논의한다.

a. exportable 함수 적용 (exportable function application)

이 함수 적용은 일반적인 데이타 플로우 모델에서 채택하고 있는 표준 실행 법칙을 준수하는 함수 적용으로서, 자체의 처리요소는 이미 자원을 상당히 사용하고 있고 시스템 상에서 유용한 다른 처리요소가 있는 경우에 유용한 다른 처리요소에 함수 적용을 떠넘김으로써 또 다른 수준의 병렬성을 꾀하고자 할 때에 사용되는 것으로 그림 4.21에서 arg에 해당되는 인자들이 Heap 구조를 갖는 배열로 구성되이 (함수 코드와 함께) 인자 배열을 유용한 다른 처리요소로 넘기고 그 처리요소가 결과를 생성하면 그 결과를 회수하여 계산을 수행하는 형태를 취하게 된다.

b. 부분적 함수 적용(Partial function application)

일반적인 데이타 플로우 수행 방식에 의거하면 함수 적용에 있어서 실제로 함수에 대한 실행은 그 함수가 필요로 하는 모든 입력 자료들이 도달한 후에야 비로서 실행이 가능한데 이경우에 함수가 실제로 수행할 때에 몇 개의 입력 자료는 사용하지 않거나 혹은 꼭 필요한 입력자료를 소비하는 계산단위가 복잡하여 시간을 많이 소비할 가능성이 있는 상황에서는 4.2.4 절에서 언급한 바와 같이 함수 적용에 있어서 한 개 혹은 몇 개의 입력 자료만 도착 하여도 부분적으로 함수를 적용함으로써 보다 높은 병렬성을 이용해서 계산 속도를 올리는 것이 가능하다. 부분적 함수 적용은 주로 처리요소 내의 함수 적용에 사용되며 그것의 구현은 다음과 같다.

함수 적용이 이루어지기 위해서는 수행 환경(execution environment)의 변화가 있어야 하는데 이것을 가능하게 해 주는 것이 토큰의 color이다. 토큰의 color는 토큰의 수행 환경을 나타내며, parameter passing과 결과값의 반환에 사용되는 경우, applicative 프로그램을 순차적으로 계산하는 스택 지향 수식 계산의 제어 체인을 구현하는 것으로 간주할 수도 있다. 또한 서로 다른 color를 가진 토큰 사이에는 아무런 상호 작용이 없으므로, 그러한 토큰 사이에는 함수의 데이타 플로우 그래프의 공유가 가능하다. 함수 적용은 다음 순서로 진행된다.

가) 먼저 fire color pool로부터 새로운 color를 얻는다.

나) 함수의 실인자에 위의 color를 지정한 후 함수 본체로 인자를 보낸다.

다) 위의 color 내에서 함수을 수행한다.

라) 결과에는 적용이 수행될 때의 color를 지정한다.

가)에서 얻어진 새로운 color는 함수 적용 수행에 대한 수행 환경에 사용되고 있다. 결과값이 얻어지면 함수 적용이 일어난 원래의 환경으로 그 결과값을 되돌려 보낸다. 그림 4.22는 함수 적용에 사용된 계산 단위를 나타내고 있다. "apply" 계산 단위는 새로운 color를 제공해 준다(그림 4.23에서 call로 표현되어 있다). "link" 계산 단위는 새로 얻어진 color에 대해 수행 환경을 변화시켜 준다. "rlink" 계산 단위는 결과값이 반환될 장소에 대한 정보를 제공해 준다. 편의상 입력 arc의 토큰을 arg1, arg2, 등으로, 또 출력 arc의 토큰을 result라 하겠다.

그림 4.22에서 "a"와 "b"는 color를 나타내며 dest와 dest1은 destination 노드 name을 나타낸다. color와 destination name 쌍을 activated object name이라 정의한다. 토큰의 value field가 activated object name을 가질 수 있다고 가정할 때, 그림 4.23은 activated object name을 나타내는 값과 color "a"를 갖는 토큰이 dest로 명시된 목적지로 보내짐을 의미한다. activated object name 값은 color "b"와 destination 노드 name "dest1"의 쌍이다. "tname"이 토큰 name을 나타낸다면 다음과 같이 토큰 field를 표시할 수 있다.

```
tname.value.color = b
tname.value.name = dest1
tname.color = a
tname.destInation = dest
```

함수 적용에 대한 데이타 플로우 그래프가 그림 4.24에 나와 있다. 이 그림에서 "fname"은 토큰 [* |f]a를 받아 들이는데, 여기서 *는 "don't care" 조건을 나타낸다. x1,...,xm은 함수 f의 실인자이고 y1,...,yn은 결과 토큰에 대한 destination 노드 name이다.

4.2.9 루프(loop)와 관련된 계산단위

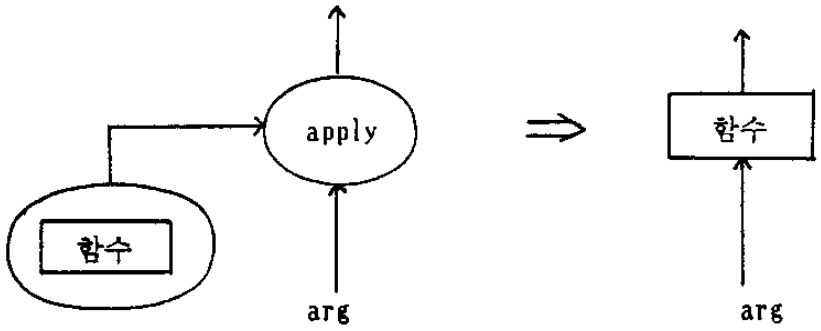
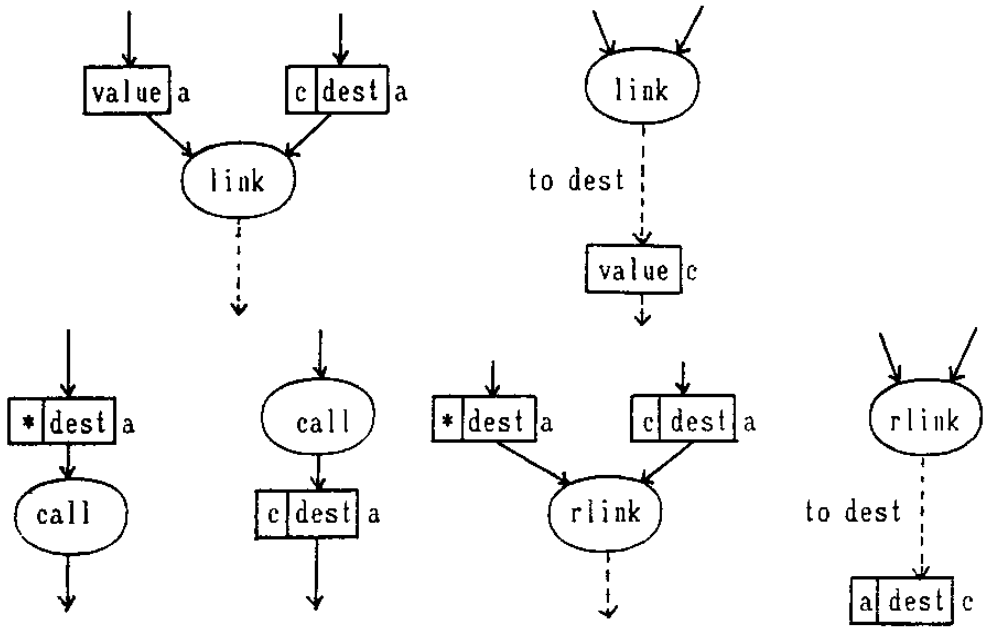


그림 4.21 APPLY 계산 단위



----- static arc

- - - dynamic arc

* don't care

그림 4.22 함수 적용에 대한 계산 단위

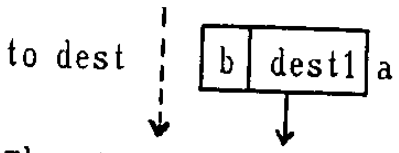


그림 4.23 activated object

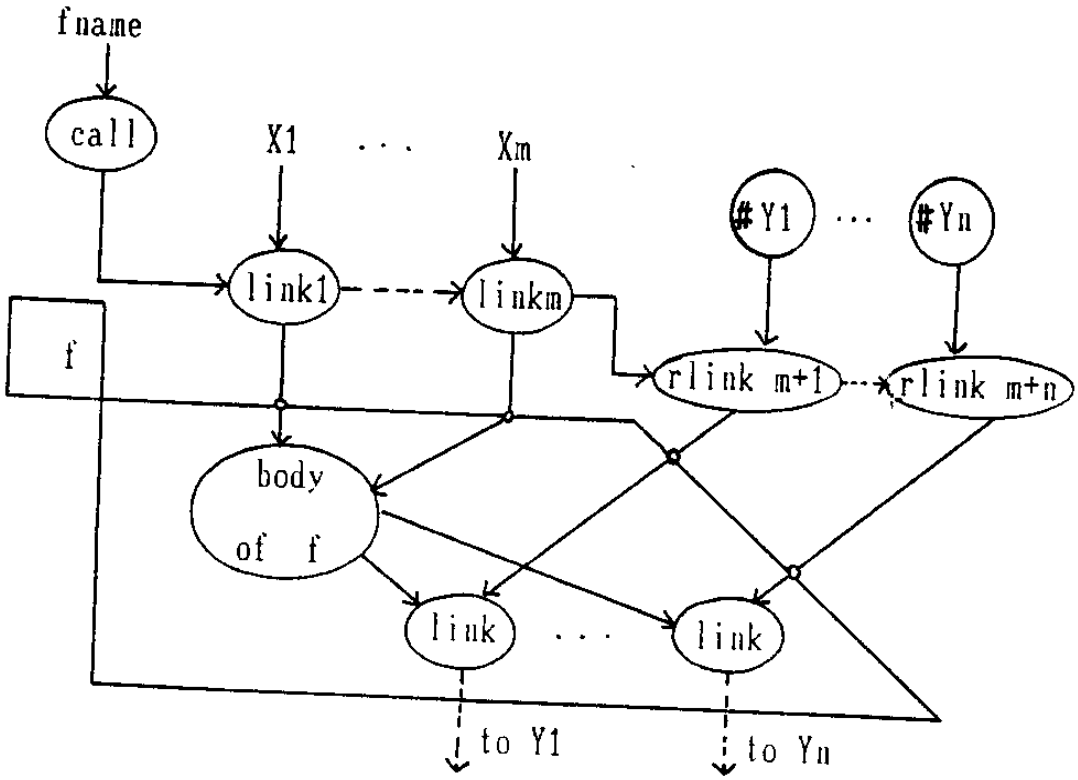


그림 4.24 함수 적용에 대한 데이타 플로우 그래프

루프와 관련된 계산단위로는 루프의 context를 바꾸어 주는 L^+, L^- 계산단위와 루프에서 반복사용되는 변수 (계산단위와 계산단위를 연결하는 arc의 이름)에 대해 입력 tag의 iteration (혹은 initiation) 항목을 증감시켜 주는 D^+, D^- 계산단위가 있으며, 그것에 관한 자세한 내용은 [ARGO 82]에 제시되어 있다.

4.3 데이타 플로우 프로그래밍 언어 (DDFPL: Divide-and-conquer Data Flow Programming Language)

본 절에서는 1차년도에서 제시한 HYPERDAC의 기본원칙을 지원하고, 4.2절에서 제안한 데이타 플로우 베이스 언어로 변환이 용이하며, 광범위한 응용범위를 가지고 문제의 병렬성을 용이하게 표현하고 제어할 수 있는, 함수언어에 기초한 데이타 플로우 프로그래밍 언어를 제시한다. 언어의 제시에 앞서 병렬언어의 요구사항과 제안된 언어의 특징을 요약하면 다음과 같다.

병렬언어는 일반적으로 프로그래밍 언어에 대해 요구하는 여러 사항(명확한 구문과 의미, 신뢰성, 신속한 번역, 효율적인 코드 등)에 부가적으로 다음과 같은 성격을 갖고 있어야 한다.

1) 문제에 내재한 가능한 모든 수준과 형태의 병렬성을 (내재적 혹은 명시적으로) 표현할 수 있는 언어가 바람직하다.
2) 프로그램에 내재된 병렬성을 탐지하기가 용이하여야 한다.
3) 프로그램의 정확한 작성과 증명이 용이하여야 한다.

본 연구에서는 앞서 언급한 병렬언어의 요구사항, HYPERDAC이 채택한 기본원칙의 지원, 데이타 플로우 베이스 언어와의 관계를 고려하여 다음과 같은 특징을 갖는 언어를 설계하였다.

1) 수치계산 (numerical computation) 및 기호 처리 (symbolic processing) 모두를 지원하드로써 보다 광범위한 응용분야에 사용이 가능토록 하였다.
2) 병렬성의 자연스러운 표현 및 탐지가 용이하고, 명확한 의미를 갖고있는 함수 언어를 기초로 한다.
3) 부가적으로 병렬성의 표현 및 탐지가 용이하도록 명시적 병렬구문(parallel constructs)을 제공한다 (예를 들면 Forall, For_each_while 등).
4) 제한된 자원 하에서 효과적인 자원의 사용을 위해 지체 연산 (delayed evaluation)을 제공하므로써 병렬성의 제어가 가능토록 하였다.
5) 다중프로세서 환경하에서 성능을 향상시키기 위하여 다른 처리요소에 작업을 전가시킴으로써 부가적인 병렬성의 이용이 가능토록 하였다(exportable 함수).

6) 강한 수형(strongly-typed)의 언어이기 때문에 수형 검색 및 디버깅(debugging)이 용이하다.

4.3.1 구문의 표현과 프로그램 요소

본 보고서에서는 언어의 구문을 표현하는데 확장된 BNF를 사용하며, 확장된 내용은 다음과 같다.

{s}는 s가 한번 혹은 한번도 나타나지 않는 것을 표현한다.

$\{s\}^*$는 s가 임의의 갯수만큼 나타나는 것을 표현한다.

$\{s\}^+$는 s가 한번이상 임의의 갯수만큼 나타나는 것을 표현한다.

ε는 빈 스트링을 나타낸다.

{s}'는 s {, s}*를 나타낸다.

프로그램을 구성하는 요소에는 연산자와 구둣점 기호, 불린 값, 실수, 정수, 문자 스트링, reserved word, 그리고 식별자(identifier)가 있다.

- 연산자와 구둣점 기호 (operation and punctuation symbols)

  + - * / < <= = >= <> ;

- <number> ::= <integer> | <real>

  <integer> ::= <sign> {<digit>}$^+$

  <real> ::= <sign> {<digit>}$^+$.{<digit>}$^+$ <scale factor>}

  <sign> ::= + | - | ε

  <scale factor> ::= E <sign> {<digit>}$^+$ | ε

- <character string> ::= {<character>}$^+$

- <identifier> ::= <letter> {<alphanumeric>}*

  <alphanumeric> ::= <letter> | <digit>

- <character> ::= elements of ASCII character set

- <letter> ::= A | B | C | D | ... | Z | a | b | c | ... | z

  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.3.2 값과 값의 수형 (Values and Types)

값의 영역(value domain)은 아래서와 같이 두개의 분리된 부영역 (subdomains)으로 나뉜다.

(1) 단순한 수형 (Simple Types): 정수, 실수, 불린, 문자

(2) 복합 (구조화) 수형 (Compound Types): 배열, 스트림, 리스트

수형의 선언, 정의, 명세에 관한 구문은 다음과 같다.

```
<type declaration> ::= type {<type definition>}* | ε
<type definition> ::= {<identifier>}' : <type specification>} ;
<type specification> ::= <simple type> | <compound type>
<simple type> ::= <basic type> | <scalar type>
<basic type> ::= integer | real | boolean | char
<scalar type> ::= <enumerated type> | <subrange type> | <type identifier>
<enumerated type> ::= ( <identifier> {<, <identifier>}* )
<subrange type> ::= <constant> .. <constant>
<constant> ::= <number> | <character string> | false | true | EOS |
               <identifier> | "<character string>" |
               undef [ <type specification>} ] |
               miss_elt [ <type specification> ]
<type identifier> ::= <identifier>
<compound type> ::= <array type> | <stream type> |
                    <list type>
<array type> ::= array [ <type specification> ] |
                 array [ <index spec> ] of <component type>
<component type> ::= <type specification>
<index spec> ::= {<scalar type>}'
<field spec> ::= <field list> {; <field list>}*
<field list> := {<field identifier>}' : <type specification>
```

<stream type> ::= stream of <component type>

<list type> ::= <list class> list of <basic type>

<list class> ::= lenient | lazy | ε

단순한 수형의 명세는 단순히 그 수형의 이름을 써주면 된다.

구조화 수형의 명세는 구성원소의 수형에 관한 정보를 제공해주는 수형생성자 (type constructors)를 명시하여 준다.

(예)

배열 수형 생성자

array[identifier]

array[array[real]]

4.3.3 값의 영역과 관련된 연산들

병렬처리 환경하에서 예외적인 현상 (exceptional cases)을 다루는 문제는 계산이 동시에 여러 곳에서 이루어지고, 잘못된 계산을 비롯하여 다른 계산의 종료가 어려운 관계로, 적어도 행해진 결과값에 예외적인 현상의 원인에 관한 정보를 계속 유지하여 파급시킴으로써 계산결과의 신뢰성을 올리고 문제의 원인을 찾아내는데 도움을 주는 방법을 사용하는 것은, 병렬처리 환경하에서의 완전한 예외적 현상에 대한 해결책이 제시되기 까지는, 바람직하다고 생각된다. 따라서 본 연구에서는 각 데이타 타입별로 적절한 원소(proper elements)와 오류 원소(error elements)를 포함하도록 함으로써 예외적인 현상을 부분적으로 다루는 것이 가능토록 하였다 (오류 원소는 그 수형의 적절한 값의 계산이 불가능할 때 그 계산의 결과로써 생성되는 값이다).

아래의 두개의 오류값은 모든 데이타 타입의 원소이다.

- undef[type]:

  연산의 피연산자 값이 그 연산의 피연산자 영역에 속하지 않을 때 발생하는 오류 (예를 들면 배열연산에서 참조하는 인덱스가 그 배열의 인덱스 범위를

벗어날 때).

- miss_elt[type]:

  배열연산에서 참조하는 인덱스가 배열의 인덱스 범위에 속하지만 그 인덱스 값이 존재하지 않을 때에 발생하는 오류.

이외에도 overflow나 underflow, zero_divide 등과 같이 컴퓨터를 사용한 계산 수행시에 흔히 발생할 수 있는 오류도 오류값에 포함된다.

(1) 값의 영역

a. 기본적인 수형

- 정수 수형 (integer type)

  . 적절한 원소:

  32 비트의 정수 표현을 가정할 때, -2^{31} -- (2^{31-1}) 내의 정수

  . 오류 원소:

  undef[integer], miss_elt[integer], pos_over[integer], neg_over[integer], zero_divide[integer], unknown[integer]

  (여기서 pos_over와 neg_over는 각각 양수와 음수의 overflow를 나타내며, unknown 오류값은 구현상의 영역에는 속하지 않을 가능성이 높으나 그것의 정확한 분류가 모호할 경우(예를 들면 undef[integer] - 1)에 발생되는 오류를 나타내고, zero_divide는 0으로 나누는 계산에서 발생하는 오류를 나타낸다).

- 실수 수형 (real type)

  . 적절한 원소:

  32 비트 혹은 64 비트의 유동소숫점 표현방법으로 표현할 수 있는 범위 내의 실수 (어떤 언어에서는 64 비트 실수를 double real로 수형선언을 하기도 하나 본 연구에서는 이것을 구분하지 않았다).

  . 오류 원소:

  undef[real], miss_elt[real], pos_over[real], neg_over[real], pos_under[real], neg_undef[real], unknown[real], zero_divide[real]

(여기서 pos_under나 neg_uncef는 각각 양수와 음수의 underflow 발생 시의 오류값을 나타낸다).

- 불린 수형 (Boolean type)
  . 적절한 원소:
  true, false
  . 오류 원소:
  undef[boolean], miss_elt[boolean]
- 문자 수형 (character type)
  . 적절한 원소:
  ASCII 문자 세트의 128 문자
  . 오류 원소:
  undef[char], miss_elt[char]

b. 구조화 수형 (Compound Type)

- 배열 수형 (array type)
  . 적절한 원소:
  구성원소가 적절한 원소로 구성된 배열
  . 오류 원소:
  undef[array[type]], miss_elt[array[type]], dbl_write[array[type]], dbl1_write[array[type]], dbl2_write[array[type]]
  (여기서 dbl_write, dbl1_write, dbl2_write는 각각 I-structure, EI-structure-1, EI-structure-2 배열의 연산시에 중복 write를 하거나 counter 이상으로 write하는 경우에 발생하는 오류이다).
- 스트림 수형 (stream type)
  . 적절한 원소:
  구성원소가 적절한 원소로 구성된 스트림
  . 오류 원소:
  undef[stream[type]], miss_elt[stream[type]]
- 리스트 수형 (list type)

· 적절한 원소:

구성원소가 적절한 원소로 구성된 리스트

· 오류 원소:

undef[list[type]], miss_elt[list[type]]

(2) 연산

a. 오류 테스트 (error tests)

isundef(v) : any -> bool (boolean value)

ismiss_elt(v) : any -> bool

iserror(v) : any -> bool

b. 불린 연산 (Boolean operations)

연산	기호	함수 표현
and	P	
or	P | Q	
equal	P = Q	
not equal	P ~= Q	
not	~ P	bool -> bool

c. 정수 연산 (integer operations)

연산	기호	함수 표현
덧셈, 뺄셈	J + K, J - K	Int, Int -> Int
곱셈, 나눗셈	J * K, J / K	
modulus	mod(J,K)	
지수 계산	exp(J,K)	
최대값, 최소값	max(J,K), min(J,K)	
equal, not equal	J = K, J ~= K	Int, Int -> bool

greater, less	J > K, J < K	
greater/equal	J >= K	
less/equal	J <= K	
negation	-J	Int -> Int
절대값	abs(J)	

d. 실수 연산 (real operations)

연산	기호	함수 표현
덧셈, 뺄셈	X + Y, X - Y	real, real -> real
곱셈, 나눗셈	X * Y, X / Y	
지수 계산	exp(X,Y)	
최대값, 최소값	max(X,Y), min(X,Y)	
equal, not equal	X = Y, X ~= Y	real, real -> bool
greater, less	X > Y, X < Y	
greater/equal	X >= Y	
less/equal	X <= Y	
negation	-X	real -> real
절대값	abs(X)	

e. 문자 연산 (character operations)

연산	기호	함수 표현
equal	C = D	char, char -> bool
not equal	C ~= D	
greater, less	C > D, C < D	
greater/equal	C >= D	
less/equal	C <= D	

f. 수형 변환 연산 (type conversion operations)

IntR : real -> Int

Float : Int -> real

IntC : char -> Int

char : Int -> char

g. 배열 연산 (array operations)

연산	기호	함수 표현
create	array type_name[]	-> array[T]
append	A[I = V] 혹은 A[I] = V	array, array -> array
select	A[I]	array, Int -> T
Concatenate	A || B 혹은 Conc(A,B)	array, array -> array
Join	Join(A,B)	array, array -> array
Index	Index_h(A), Index_l(A)	array -> Int
create/fill	Crefil(L,H,V)	Int, Int, T -> array[T]
Number	Num(A)	array -> Int
Set bounds	SB(A,L,H)	array, Int, Int -> array
Allocation	array(L,H)	Int, Int -> array
	array1(L,H,C)	Int, Int, Int -> array
	array2(L,H,C)	Int, Int, Int -> array
Read	A[I]	array[T], Int -> T
Write	A[I] = V	array, Int -> T
Bounds	Bounds(A)	array -> Int, Int
Size	Size(A)	array -> Int

h. 스트림/리스트 연산 (stream/list operations)

연산	기호	함수 표현
create	stream type_name[]	-> stream[T]
insert_rear	consl(G,V)	stream[T], T -> stream[T]
car	first(G)	stream[T] -> T
	car(L)	list[T] -> T
cdr	rest(G)	stream[T] -> stream[T]
	cdr(L)	list[T] -> list[T]
cons	cons(A,L)	T, list[T] -> list[T]
append	append(L1,L2)	list[T], list[T] -> list[T]
test for empty	empty(G)	list[T] -> bool
stream append	G || H	stream[T], stream[T] -> stream[T]
size of stream	ssize(G)	stream[T] -> Int

4.3.4 값의 이름과 수식 (Value Names and Expressions)

(1) 값의 이름 (value names)

값의 이름은 특정 수형의 계산값을 지칭하는 이름으로서, 함수 정의의 머리부분이나 let 블럭과 for (forall) 블럭 내에 나타난다.

(예)

```
Function F (X:integer returns real)
   <expression>
endfun

let
   X : real := 3.0;
```

```
    <다른 결합 (binding) 수식들>;
        ...
    in <expression>
  endlet
```

(2) 수식 (expressions)

수식은 보다 작은 수식과 연산 기호에 의해 형성되며, 프로그램 작성과 이해가 쉽도록 여러 arity를 갖는 수식도 허용한다. 또한 지체 계산할 수식에 대해서는 그 수식의 앞에 'delay' 표식을 명시할 수 있도록 하므로써 제한된 자원하에서 사용자가 보다 효과적인 자원의 이용방안을 프로그램할 수 있도록 하였다.

(예)
```
A
true
3.7 E -02
3 * (X + Y)
delay func(3 + X, Y)
product(X, delay Y)
R.X.Y.ZZ
if P then 4 else 5 endif
P,Q,R = TRIPPLE(X,Y,Z)
```

수식에 대한 구문은 다음과 같다.

```
<expression> ::= <simple expression> |
        <several construct expressions> |
        <simple expression> <relational_op> <simple expression> |
        delay <expression> | d_construct ( <list_of_blk_exp> ); |
        force <expression> | # <expression> | <blk_array_com>
<simple expression> ::= <term> |
        <simple expression> <relational_op> <simple expression>
```

```
<list_of_blk_exp> ::=
        {[ <blk_exp> ] : <array_name> % <com_exp> %}'
<blk_exp> ::= {( <com_blk_exp> )}'
<com_blk_exp> ::= {<exp>}'
<blk_array_com> ::= <array_name> % <com_exp> % from <array_ref>
<term> ::= <factor> | <term> <multiplying_op> <factor>
<factor> ::= <primary> | <unary_op> <primary>
<primary> ::= <constant> | <identifier> | ( <expression> ) |
        <function invocation> | <prefix_op> |
        <array_ref> | <array_generator> | <error_test> |
        <stream_ref> | <stream_generator> |
        <list_ref> | <list_generator>
<unary_op> ::= + | - | not
<adding_op> ::= + | - | or
<multiplying_op> ::= * | / | and | ||
<relational_op> ::= < | <= | > | >= | = | <>
<prefix_op> ::= error(<ex>) | min(<ex>,<ex>) | max(<ex>,<ex>)
        abs(<ex>) | intr(<ex>) | float(<ex>) | intc(<ex>) |
        char(<ex>) | mod(<ex>) | exp(<ex>) |
        first(<ex>) | rest(<ex>) | empty(<ex>) |
        ssize(<ex>) | consl(<ex>) |
        car(<ex>) | cdr(<ex>) | cons(<ex>,<ex>)
<ex> ::= <expression>
<function invocation> ::= <function name> (<actual para_list>)
<function name> ::= <identifier>
<actual para_list> ::= {<actual parameter>}'
<actual parameter> ::= <expression>
```

4.3.5 프로그램 구조 (Program Structures)

본 절에서는 프로그램의 구조에 대해 살펴보고, 함수 프로그램의 기본적인 구조인 제어구조 (conditional constructs), 블럭구조 (block constructs), 루프구조 (loop constructs), 함수적용 (function application)에 대해 살펴보고자 한다 (함수 적용에 관한 내용은 4.3.4의 수식에서 논의하였기 때문에 이 절에서는 설명을 생략한다).

본 연구에서 제안한 언어는 프로그램 구소상 함수들의 정의의 집합으로 구성되며, 함수 정의 내에 함수 정의를 허용하지 않으며 대신에 그 함수 정의에서 사용하는 함수들에 대해서는 IMPORT 선언을 통해 표시하여주고, 또한 그 함수의 수행시에 다른 처리요소로 함수적용을 전가시켜서 수행할 수 있는 함수들은 EXPORT 선언을 통해 표시함으로써 앞서 제시한 두가지 형태의 함수 적용을 사용자 선에서도 가능토록 하였다(프로그램 실행시에 수행될 함수는 프로그램에서 해당되는 함수를 실인자를 갖고 호출할 때 수행되며, 그 경우 실제의 DAC 프로그램은 다른 어떤 언어(예를 들면 C, PASCAL, Concurrent Pascal 등)로 작성될 것이고, 그 프로그램과 DAC 프로그램과의 연결은 운영체제가 담당한다).

프로그램 구조에 대한 구문은 다음과 같다.

```
<program> ::= {<function definition>}*
<function definition> ::=
    DAC DEF {EXP}<function name> <function heading>
            IMPORT <list of import functions>
            EXPORT <list of exportable functions>
            <type declaration>
            <function body>
    endDACfun
<function name> ::= <identifier>
<function heading> ::=
            ( <formal list> returns <result types> ) ;
```

```
<formal list> ::= <name definition> {; <name definition>}*
<result types> ::= {<type specification>}'
<function body> ::= <expression>
<name definition> ::= {<name>}' : <type specification>
<name> ::= <entire name> | <component name>
<entire name> ::= <identifier>
<component name> ::= <indexed name>
<indexed name> ::= <array name> [ {<expression>}' ]
<array name> ::= <name>

<several construct expressions> ::= <conditional constructs> |
          <block constructs> | <loop constructs>
```

(1) 제어 구조 (control or conditional constructs)

제어 구조는 제어 입력값에 따라 자료의 흐름을 제어하는 구조로 베이스 언어상에서 분배, 선택, T, F 노드 등으로 구현되며, 언어상에는 if-then-else, case 수식 형태로 표현된다. 제어구조에 대한 구문은 다음과 같다.

```
<conditional constructs> ::=
          ( <initial part> <if-then-else construct> ) |
          <if-then-else construct> | <case expression>

<if-then-else construct> ::=
          if <expression> then <true expression>
                         else <false expression>
<initial part> ::= <type declaration> |
          <binding definitions> | ε
<true expression> ::= {<expression>}' | do_nothing
<false expression> ::= {<expression>}' | do_nothing
<binding definitions> ::= {<binding>}+
```

```
<binding> ::= {<name>}' = {<expression>}' ; |
              <several construct bindings>

<case expression> ::= ( <initial part> <case construct> ) |
              <case construct>
<case construct> ::= case <expression> of <case lists> endcase
<case lists> ::= <case list> {; <case list>}*
<case list> ::= <case label> : {<exression>}'
<case label> ::= {<identifier>}'

<several construct bindings> ::=
              <conditional bindings> | <loop bindings>

<conditional bindings> ::=
              ( <initial part> <if-then-else binding> ) |
              <if-then-else binding> | <case binding>
<if-then-else binding> ::=
              if <expression> then <true binding>
                                   else <false binding>
<true binding> ::= <binding> | do_nothing
<false binding> ::= <binding> | do_nothing

<loop bindings> ::= <for binding> | <while binding>
<for binding> ::=
              ( <initial part> for <head expression>
                               do {<binding>}* )
<while binding> ::=
              ( <initial part> while <boolean expression>
                               do {<binding>}* )
```

(2) 블럭 구조 (block constructs)

블럭 구조는 수식과 같은 효과를 나타내며, 여러개의 수식 정의 문장과 결과 수식으로 구성된다. 일반적으로 블럭구조는 복잡한 수식을 표현할 때 사용되며, 그것에 대한 구문은 다음과 같다.

```
<block constructs> ::=
        ( <initial part> {<expression>}' ) | <let block>
<let block> ::= ( let <initial part> in {<expression>}' )
```

(3) 루프 구조 (loop constructs)

루프는 각 루프의 instance 간에 데이타 종속관계가 있는 순환형 루프 (cyclic loop)와 데이타 종속관계가 없는 비순환형 루프 (acyclic loop)로 분류할 수 있으며, 순환형 루프의 경우는 루프의 각 instance 간의 파이프 라인 수행을 통해 병렬처리가 가능하며, 비순환형 루프의 경우는 각 instance 별로 펼쳐서 (unfolding) 계산을 수행하므로써 병렬처리할 수 있다. 또한 병렬성의 탐지가 용이하도록 하기위해 제안한 언어에서는 병렬 구조인 Forall과 For_each_while 등의 구조를 부가적으로 제공하였으며, 루프 구조에 관한 구문은 다음과 같다.

```
<loop constructs> ::= <for construct> | <while construct>
        <for_each construct> | <for_each_while construct> |
        <forall construct>
<for construct> ::=
        ( <initial part> for <head expression>
                do <expression> )
<while construct> ::=
        ( <initial part> while <boolean expression>
                do <expression> )
<boolean expression> ::= <expression>
<head expression> ::= <boolean expression> |
        <identifier> = <start> , <final> {, <increment>}
```

```
<start> ::= <identifier> | <number>
<final> ::= <identifier> | <number>
<increment> ::= <number>

<for_each construct> ::=
          ( <initial part>
          for each <identifier> in <stream name>
                  {; <identifier> in <stream name>}*
          do <expression> )

<for_each_while construct> ::=
          ( <initial part>
          for each <identifier> in <stream name>
                  {; <identifier> in <stream name>}*
          while <expression> do <expression> )

<forall construct> ::=
          forall <identifier> in [ <forall range> ]
                  {; <identifier> in [ <forall range> ]}*
                  <type declaration>
                  <binding definitions>
                  {<forall body>}+
          endall
<forall range> ::= <expression> , <expression>
<forall body> ::=
          <name> ::= construct <expression> ; |
          <name> ::= eval <forall_op> <expression> ;
<forall_op> ::= plus | times | min | max | or | and
```

4.3.6 프로그램 예

다음은 divide_and_conquer 형태로 matrix multiplication을 행하는 프로그램을 DDFPL로 작성한 예이다.

```
Function DAC_mmt(A;B:array[array[integer]], N:integer;
                                        returns array[array[integer]])
(* This program takes two N x N matrices A and B as inputs and returns N
x N matrix C which is the product of A by B. *)
(* It can be solved according to the divide-and-conquer schema by
dividing A, B and C into four square matrices each as follows.

   | A11  A12 |   | B11  B12 |   | C11  C12 |
   |          | x |          | = |          |
   | A21  A22 |   | B21  B22 |   | C21  C22 |

   and computing the four components of C by eight independent
multiplications, followed by four additions : Cij = Ai1 x B1j + Ai2 x
'B2j for i,j ε {1,2}. The eight multiplications are performed by eight
recursive calls that are executed in parallel. *)
(* We assume that N is even, to simplify programming task. *)
{C = array (1,N) (1,N) ;
  {if N = 2
    then {forall I from 1 to 2 do
            {forall J form 1 to 2 do
               C[I,J] = A[I,1] * B[1,J] + A[I,2] * B[2,J] }}
    else {C= d_construct([(1,N/2),(1,N/2)] : C%1,1%,
                         [(1,N/2),((N/2 +1),N)] : C%1,2%,
                         [((N/2 +1),N),(1,N/2)] : C%2,1%,
                         [((N/2 +1),N),((N/2 +1),N)] : C%2,2% ) ;
(* P%x,y% is a component(x row and y column) block matrix of matrix P.
```

For instance, P can be represented by block matrices as following:

$$P = \begin{vmatrix} P\%1,1\% & P\%1,2\% \\ P\%2,1\% & P\%2,2\% \end{vmatrix}$$

"d_construct" is an operation which constructs a matrix, consisting of several block matrices. *)

```
        A%1,1% from A[(1,N/2),(1,N/2)] ;
        A%1,2% from A[(1,N/2),((N/2 +1),N)] ;
        A%2,1% from A[((N/2 +1),N),(1,N/2)] ;
        A%2,2% from A[((N/2 +1),N),((N/2 +1),N)] ;

        B%1,1% from B[(1,N/2),(1,N/2)] ;
        B%1,2% from B[(1,N/2),((N/2 +1),N)] ;
        B%2,1% from B[((N/2 +1),N),(1,N/2)] ;
        B%2,2% from B[((N/2 +1),N),((N/2 +1),N)] ;
```

(* "from" construct is used to represent a block matrix in a matrix. For instance, "A%1,1% from A[(1,N/2),(1,N/2)]" means that a block matrix A%1,1% of A consists of rows from 1 to N/2 and columns from 1 to N/2, of matrix A. *)

```
    { forall I from 1 to 2 do
        { forall J from 1 to 2 do
            C%I,J% = DAC_mmt(A%I,1%,B%1,J%,N/2) +
                     DAC_mmt(A%I,2%,B%2,J%,N/2)  }}}}
  in
    C }
```

4.3.7 분석 및 컴파일러에 관한 고려사항

DDFPL의 문제점 및 컴파일러의 설계시 고려되어야 할 점들은 다음과 같다.

(1) delayed evaluation과 exporatable function에 대한 프로그래머의 명백한 표

기의 문제점

. 부분적으로 프로그래머가 명시하고,

. 나머지 부분은 부가적으로 컴파일러가 담당한다.

(2) 프로그램의 적절한 분할 및 할당 문제

. 지역성 이용 측면

. 병렬성 이용 측면

. 통신 overhead 감소 측면

(3) 배열의 수형 결정 문제

. exportable 함수의 인자 --> heap 구조 배열.

. PE내의 함수 적용 --> I-structure.

. EI-structure-1과 EI-structure-2 --> 프로그래머의 명백한 표기

. 프로그래머의 표기가 우선한다.

(4) forall construct의 효과적인 구현 문제

. 파이프라인 처리

. 배열 처리

4.4 HYPERDAC(HYPER-Divide-And-Conquer multiprocessor system)의 개념적 설계

Divide-And-Conquer(DAC) 알고리즘을 효과적으로 수행하면서 일반적인 다른 알고리즘도 무리없이 수행할 수 있도록 제안된 고속의 다중처리 시스템인 HYPERDAC은 DAC 알고리즘의 병렬성을 고도로 이용하기 위하여 여러 수준의 병렬성을 이용할 수 있는 데이타 플로우 모델을 계산 모델로 채택하고, DAC 알고리즘의 논리적 계층 구조를 활용하고 일반적인 다른 알고리즘에의 적용도 용이하도록 Hypertree를 상호연결망으로 채택하고, 각 처리요소에 부과된 부하를 균등하게 함으로써 전체 시스템의 이용도와 성능을 올리는 분산 부하 균형 정책으로 Gradient 부하 균형 정책을 채택하였으며, 제한된 자원하에서 과도한 병렬성을 억지하고 계산의 지역성을 이용함으로써 시스템 자원의 효율적 사용을 고양하는 throttling 기법을 채택한다.

본 절에서는 HYPERDAC의 기본적인 원칙을 제시하고, HYPERDAC의 구조에 대해 개념 설계를 행하며, 시스템의 작동 시나리오와 성능 분석에 대해 차례로 논의한다.

4.4.1 HYPERDAC의 원칙

본 연구에서 제안한 HYPERDAC은 DAC 알고리즘을 매우 효율적으로 수행할 뿐만 아니라, 일반적인 여러 알고리즘도 효율적으로 수행할 수 있는(범용에 가까우면서도 특히 DAC 알고리즘을 고속으로 처리할 수 있는) 고속의 다중처리 시스템이다.

그러므로 HYPERDAC은 DAC 알고리즘의 특징인 recursiveness, 관련된 자료의 동적 할당, 그리고 부분들간의 독립성을 효과적으로 이용할 수 있어야 하며 비정규적(irregular)이고 임의의 크기의 계산 구조(Computation Structure)를 갖는 여러 알고리즘도 비교적 큰 부담없이 실행되어야 할 것이며, 또한 다중처리 시스템에서 필수적으로 요구되는 확장성(Expandability), 결함 허용성 (Fault-tolerancy), 그리고

VLSI 기술에의 적합성 등을 가져야 할 것이다.

본 연구에서 HYPERDAC에 대해 채택한 원칙은 다음과 같다.

- 데이타 플로우 계산 모델
- Hypertree 상호연결망
- Gradient 부하 균형 정책
- Throttling

(1) 데이타 플로우 계산 모델

HYPERDAC은 계산 모델로 데이타 플로우 계산 모델을 채택한다. 그러나 기존의 다른 데이타 플로우 계산 모델의 구현 방법과는 달리 태스크(사용자가 정의한 함수) 수준과 기계어 수준의 병렬성을 공히 이용하는 구현 방법을 채택하였으며, 또한 순수한 data-driven 방식의 계산 모델에서 문제시되는 과도한 병렬성의 제어를 위해 지체 연산(Delayed evaluation)을 첨가하였다.

a. 계산 이동(분산) 단위로서 태스크와 functional pipeline

HYPERDAC에서는 상당한 병렬성이 존재해서 계산의 이동을 통한 분산 수행이 보다 효과적인 경우에 그러한 사실을 사용자가 프로그램상에 표현하거나, 컴파일러가 탐지함으로써, 혹은 run-time 시에 시스템이 부하 균형에 적절히 대응함으로써, 관련된 계산을 유용한(available) 노드로 이동(migration)하여 병렬 수행함으로써 성능의 향상을 꾀하며, 아울러 각 태스크 내의 병렬성은 기존의 데이타 플로우 머신이 채택하는 functional pipeline을 통해 이용함으로써 최대한의 병렬성을 이용하고 있다.

예를 들면, DAC 알고리즘의 수행시에 해당되는 DAC procedure에 사용자가 언어상에 병렬수행의 표기를 행하고, 다른 태스크에는 그러한 표기를 하지 않는 경우에, 컴파일러는 그에 상응하는 적절한 코드 변환을 행하고, run-time 시에 시스템은 표기된 procedure들은 일단 그 노드에서 직접 수행하지 않고 Apply 큐에 삽입한 다음 시스템 내에 유용한 노드가 있고 그 노드로의 태스크 전달 오버헤드가 병

렬 수행을 통한 이득을 퇴색하게 하지 않는다면, 그 태스크를 해당 노드로 이동하고, 결과를 전달받음으로써 병렬 수행을 행할 수 있으며, 만약 그러한 노드가 없다면, 그러한 태스크를 이동하지 않고 자체에서 처리하게 된다. 다시 말해서 시스템의 모든 노드의 이용도(utilization)를 올리면서 가능한 통신의 오버헤드가 병렬 수행을 통한 이득을 상쇄시키지 않도록 함으로써 전체 시스템의 성능 향상을 꾀하고 있다. 또한 각 태스크 내의 병렬성을 이용하기 위하여 데이타 플로우 계산 모델의 병렬성을 이용하는 functional pipeline을 사용함으로써 전 시스템의 성능을 무리없이 자연스럽게 고양시키는 방법을 채택하고 있다.

b. 동적 태깅 모델(Dynamic Tagging Model)

HYPERDAC에서는 데이타 플로우 계산의 구현 모델로 U-interpreter에서 사용하는 동적 태깅 모델을 채택하였다 [ARCU 83a].

c. Eager/Delayed evaluation

HYPERDAC에서는 최대한의 병렬성 이용과 과도한 병렬성의 제어를 위해 eager/delayed evaluation 기법을 모두 허용한다. 또한 이러한 eager/delayed evaluation을 사용자 수준에서의 명백한 표기 혹은 컴파일러 탐지를 통해 행함으로써 보다 강력한 병렬성의 programmability를 제공한다.

예를 들어 사용자 수준에서의 eager/delayed evaluation 표기를 보면 다음과 같다.

i. Eager evaluation

```
x = f(y, z, # p)
```

위의 표현은 f 함수에 y, z, p 값을 인수로 적용한 결과 값이 x 라는 것을 의미한다. 여기서 주의해야 할 점은 일반적으로 데이타 플로우 계산 모델의 standard firing rule에 따르면 함수의 계산이 모든 인수(arguments)의 값이 계산되어 도달된 후에 수행되지만, 위와 같이 # 표시 뒤의 인수에 대해서는 그 값이 계산, 도달되

지 않아도 y, z 값이 계산되어 도달되면 일단 함수 계산을 수행한다는 것이고, 이 값을 요구하는 계산 단위는 이값이 계산된 후에 그 사실을 통보받고 실행하게 되어서 (non-standard firing rule), 최대한의 병렬성을 얻을 수 있다. 그러나 본 연구에서는 아주 기초적인 연산자를 제외한 대부분의 계산이 nonstrict 하고 eager 하게 수행될 수 있도록 하였기 때문에

eager evaluation은 사용자의 특별한 표기없이도 자동적으로 구현된다(자세한 내용은 4.2 절에 제시되어 있다).

ii. Delayed evaluation

```
x = delay f(y, z, p)
```

위의 표현에서 delay 표현이 없는 경우는 일반적인 데이타 플로우 semantics에 의거해 실행되겠지만, 앞에 delay가 붙은 경우에는 y, z, p 값이 모두 계산되어 도달되어도 함수 f의 실행을 일단 제지할 것을 시스템에 요구하고 그후 force x를 만날 때에만 그 실행을 재개하도록 한다.

이러한 delayed evaluation의 구현은 두 가지로 볼 수 있다.

- 함수(사용자 정의 함수 혹은 시스템 정의 함수)의 delayed evaluation은, 그 계산 단위를 만나게 되면 시스템은 그 함수의 실행을 연기하다가, force를 만나게 되면, 실행을 재개하는 방식으로 수행한다(이것의 구현 방법은 4.2절에 제시되어 있다).
- 일반적인 수식의 delayed evaluation 함수를 제외한 다른 표현에도 delayed evaluation 기법을 적용할 수 있으며 그것의 구현 방법은 4.2 절에 제시되어 있다.

(2) 상호연결망(Interconnection Network)

다중처리 시스템에서 가장 이상적인 경우는 문제의 계산 구조(Computation Structure)와 실제의 시스템의 구조가 일치하는 경우(즉, 문제의 통신 구조와 granule이 실제 구조상의 통신 구조와 granule과 일치하는 경우)이다.

우리가 어떤 특정의 문제만을 효율적으로 수행하는 전용 컴퓨터를 목표로 한다면, 그것은 구조상에 비교적 용이하게 구현될 수 있으나, 과연 그러한 전용 컴퓨터(Special-purpose computer)의 응용이 그것의 설계 및 구현에 사용된 비용을 정당화할 수 있을 것인가는 몇몇 응용분야(예를 들면 image processing 등)를 제외하고는 아직 정확히 밝혀지지 않았고 또한 논란의 대상이 되고 있다[SNJA 85].

따라서 우리가 본 연구에서 HYPERDAC에 채택한 접근 방법은 가능한 DAC 알고리즘을 가장 효율적으로 수행하고, 기타의 여러 일반적인 알고리즘도 별무리없이 수행할 수 있는 컴퓨터를 설계하는 것이었다.

HYPERDAC은 DAC 알고리즘의 특성인 계산의 논리적 계층구조를 실제 구조상에서 가능한 유지하고, 여타의 알고리즘을 무리없이 수행할 수 있고, 확장성, 결함허용성 및 VLSI 기술의 이용을 허용하기 위하여 Hypertree를 기본 상호연결 네트워크의 위상(Topology)으로 채택하였다.

Hypertree의 특징은 다음과 같다[GOSE 81].

a. 트리(tree) 구조

이것은 DAC 알고리즘의 논리적 계층 구조를 가능한 구조상에 유지하는데 적절한 구조이다.

b. 비교적 풍부한 상호연결 링크(links)

이것은 결함허용성과 일반적인 알고리즘의 통신 구조에서 요구하는 통신을 무리없이 제공하고, 부하 균형을 행할 시에 평균적으로 짧은 거리를 제공하기 위해 필요한 성질이다(Hypertree에서는 일반적인 트리 구조와는 달리 각 수준별로 cube connection을 제공하고 있으며, 이러한 부가적인 cube connection은 FFT 및 기타의 알고리즘과 결함허용을 위해 제시되었다[FOOT 84, GOSE 81]).

c. 노드당 고정된 핀(pin) 수와 VLSI 기술에의 적용성

통신선(communication line)을 drive 하는데 소요되는 power와 pin 수에 대한 제약을 현실적으로 고려하기 위해 Despain[DEPA 79]은 노드당 일정한 통신 bandwidth를 가정하였고, 그 가정에 의하면 가능한 적은 핀 수와 노드당 고정된 핀 수를 사용하면 확장성이 좋아지고, 각 링크의 통신 bandwidth가 증가하고, 그리고 VLSI 기술의 높은 logic/pin 비율 원칙에 부합된다.

d. 확장성

Hypertree에서는 시스템의 작동에 무관하게 각 노드를 확장시킬 수 있으며, 그러한 확장이 시스템의 성능을 증가시켜 주면서도, 별도의 오버헤드를 요구하지 않는다(즉, 노드의 증가에 따라 switch 링크의 수가 증가한다거나, 하나의 노드의 증가가 과도한 노드의 동반 증가를 요구한다거나(Hypercube의 경우) routing 및 addressing에 관련된 정보가 크게 변동되지 않는다).

e. routing 및 기타의 Hypertree에 관한 자세한 내용은 [GOSE 81]에 제시되어 있다.

(3) Diffusion에 의한 태스크의 이동(Gradient Load Balancing scheme)

DAC 알고리즘의 성격상 문제(의 데이타)에 따라 분할되는 부문제들의 수가 상이하고, 그러한 부문제에 할당되는 자료의 양 또한 상이하기 때문에 단순한 태스크 분산은 시스템 전체의 부하 및 이용도의 불균형으로 인하여 성능이 저하될 것이다. 따라서 run-time 시에 시스템의 부하에 따라 적절히 능동적으로 대처하는 부하 균형 정책이 필요하다.

그러나 부하 균형을 중앙집중식으로 처리한다면 그러한 일을 행하는 노드의 교통 적체 및 고장에 따른 시스템의 성능 저하는 피할 수 없게 된다. 그래서 다중처리 환경하에서 효율적인 분산 부하 균형 정책으로 여러 방법이 제시되었다[LIKE 86].

HYPERDAC에서는 그러한 분산 부하 균형 방법중 gradient 모델 [LIKE 86]을 채택하였다. Gradient 모델은 지역적 부하 균형을 통한 전 시스템의 부하 균형을

피하는, 비교적 구현이 간단하고, 불필요한 복잡성 및 제어 정보의 전달을 억지하는, 요구 지향(demand-driven) 부하 균형 정책으로서 다중처리 환경하에서 매우 적합한 것으로 알려져 있다[KELI 84a,LIKE 86]. Gradient 부하 균형 정책을 간략하게 설명하면 다음과 같다.

a. 작업의 분산을 요구하는 노드는 주변의 노드에 그러한 사실을 전달한다.

즉 노드의 부하 상태가 바뀔 때나 혹은 일정한 시간 후에 매번 부하에 관한 제어 정보를 전달하는 것이 아니라, 그 노드가 다른 작업을 전달 받아서 행할 여력이 있을 때에만 제어 정보를 전달함으로써 불필요한 제어 정보의 전달을 회피한다.

b. 부하 정보 및 통신 오버헤드를 고려하여 분산을 결정한다.

각 노드는 자신의 부하와 주변의 부하 정보, 그리고 태스크 이동에 따른 통신 오버헤드를 고려하여, 태스크를 이동할 것인지, 아니면 자체에서 처리할 것인지를 결정한다.

이와 같이 Gradient 모델은 요구 지향(demand-driven) 형식을 취하므로 불필요한 제어 정보의 전달을 피하고, 주변 노드들의 상태를 통해 전 시스템의 부하 균형을 행하는 분산 부하 균형 정책이기 때문에 다중처리 환경에 적절한 부하 균형 정책이다. Gradient 모델에 관한 자세한 설명은 [LIKE 86]에 제시되어 있다.

현재 [KELL 85,LIKE 86]에서 제시된 모델에서는 각 노드의 부하 상태를 결정하는 요인으로 Apply 큐 내의 패킷의 수, 사용되는 메모리의 양 등을 고려하고 있으나, 본 연구에서는 그 외에 구조화 메모리의 사용량, WM의 사용량 등을 고려하고 있지만 관련된 식의 형태와 인자에 관해서는 보다 많은 응용분야에 대한 경험과 연구가 필요하다.

(4) Throttling

일반적으로 시스템 내의 사용 가능한 자원에 비해 상당한 병렬성이 있는 경우, 있는 그대로 모든 병렬성을 실현하려는 것은 무모하고, 최악의 경우 자원의 소진으

로 인해 교착상태와 같은 회복하기 어려운 상황까지 유발할 수 있다. 따라서 그러한 환경하에서는 사용가능한 시스템 자원에 적합하도록 병렬성을 억제(Throttling)하는 것이 필요하다.

그러한 throttling의 필요성 및 관련된 여러 기법이 [RUSA 87]]에 제시되어 있다. 그 논문에 의하면 throttling의 기법을 다음과 같이 두 가지로 분류하고 있다.

. 시스템 상에 처리할 activity의 수를 제한한다.

. 지역성(locality)를 강화한다.

HYPERDAC에서는 다음과 같은 방법으로 위의 두 가지 기법 모두를 이용하고 있다.

a. 시스템 상에 처리할 activity 수의 제한

HYPERDAC에서는 이러한 효과를 다음과 같은 두 가지 방법으로 구현하고 있다.

i. delayed evaluation은 해당되는 계산을 지체시킴으로써 시스템상에 처리할 작업을 당분간(force를 수행할 때까지 혹은 시스템내에 유용한 자원이 있을 때까지) 제한한다.

ii. 시스템의 부하가 포화상태에 도달하거나 태스크의 분산(이동) 계산을 통한 이득이 통신 오버헤드로 상쇄될 우려가 있는 경우에 각 노드는 태스크의 이동을 억제 하고, 단순히 Apply 큐에 삽입함으로써 노드의 지역적 처리를 기다리게 되어, 처리(분산 처리 혹은 자체 처리)할 작업을 제한한다.

b. 지역성(locality)을 강화한다.

HYPERDAC에서는 이러한 효과를 다음과 같은 두 가지 방법으로 구현하고 있다.

i. 각 노드당 작업 이동 단위는 태스크이고 분산 계산으로 명시되지 않은 태스크는 그 노드내에서 처리된다. 태스크 내의 계산은 다른 노드에 위치한 태

스크간의 통신과 관련된 계산 외의 다른 계산은 노드 내에서 파이프라인을 통해 처리된다. 즉 태스크 내의 대부분의 연산이 여러 노드를 거치는 연산 없이 그 노드 자체에서 처리된다.

ii. 시스템의 부하가 포화된 상태에서는 더이상의 작업 이동은 행하지 않고, 노드 자체 내에서 Apply 큐에서 가능한 확장의 기회가 적은 패킷부터 가져와서 처리한다. 이것의 구현은 다음과 같다.

시스템이 포화상태에 도달하기 전에는 각 노드는 Apply 큐의 앞에 있는 태스크부터 먼저 이동시킨다. 즉 아직 시스템이 포화상태에 도달하지 않았기 때문에 보다 병렬 처리의 기회가 많은 태스크를 이동한다(즉 recursion level이 적은 것이 보다 이동의 우선순위가 높다).

그러나 시스템이 포화상태에 도달하면 태스크의 이동으로 인한 이득이 없고 오히려 상태를 악화시켜서 성능을 저하시키기 때문에 각 노드는 Apply 큐에서 가장 병렬처리의 기회가 적은 태스크(Apply 큐의 뒤 쪽의 태스크)를 먼저 처리한다.

위와 같은 처리 방법은 시스템이 포화상태에 이르기 전에는 breadth-first로 계산을 행하고, 시스템이 포화상태에 도달하면 depth-first로 계산을 수행하는 것과 유사하다[KELL 85].

4.4.2 HYPERDAC의 구조

시스템의 전반적인 구조는 그림 4.25과 같이 Hypertree 구조를 갖는다. 주의해야 할 점은 일반적인 트리구조와는 달리 각 수준별로 별도의 cube connection을 갖는다는 것이다. 그림 4.26은 각 노드가 스위치(switch)와 처리요소(PE: Processing Elements)로 구성된 것을 보여 주고, 그림 4.27은 노드 구조의 블럭 다이아그램이다.

그림에서 보는 바와 같이 처리요소는 비동기적으로 수행하는 8개의 부시스템(Subsystems or Functional units)으로 구성되어 있으며, 각 부시스템 간에는 고정된

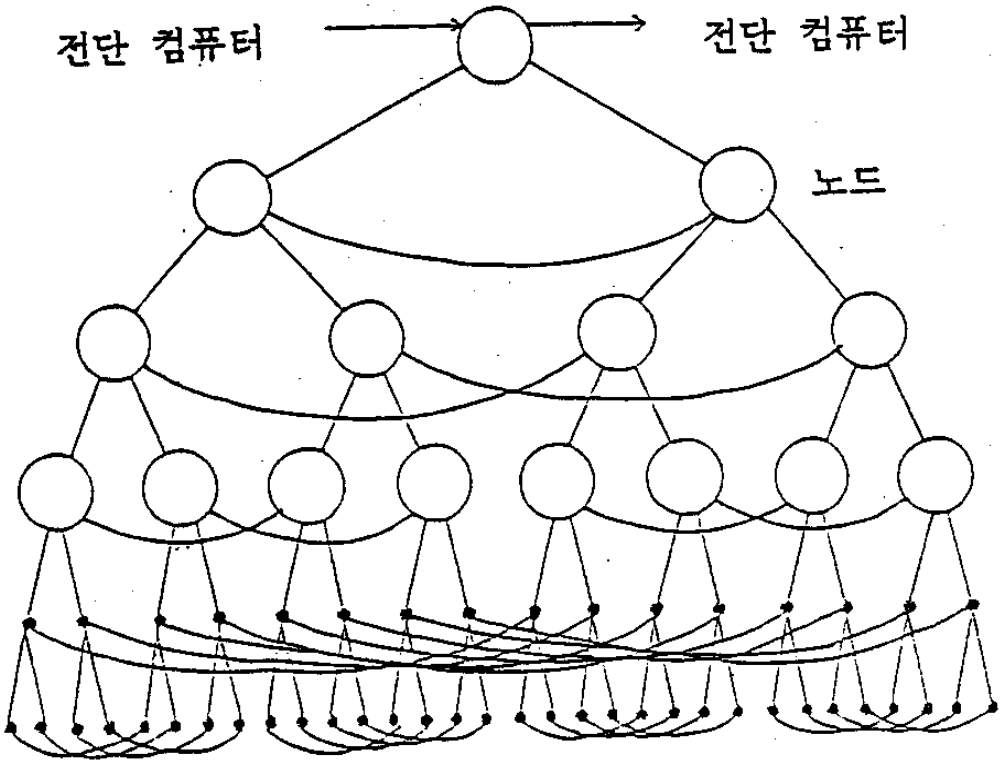


그림 4.25 HYPERDAC의 전반적인 구조

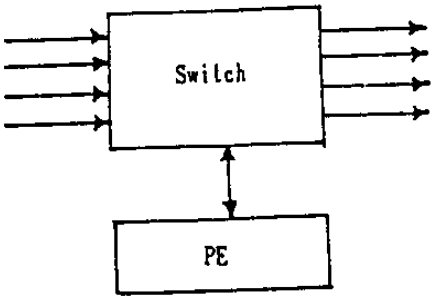


그림 4.26 노드 구조

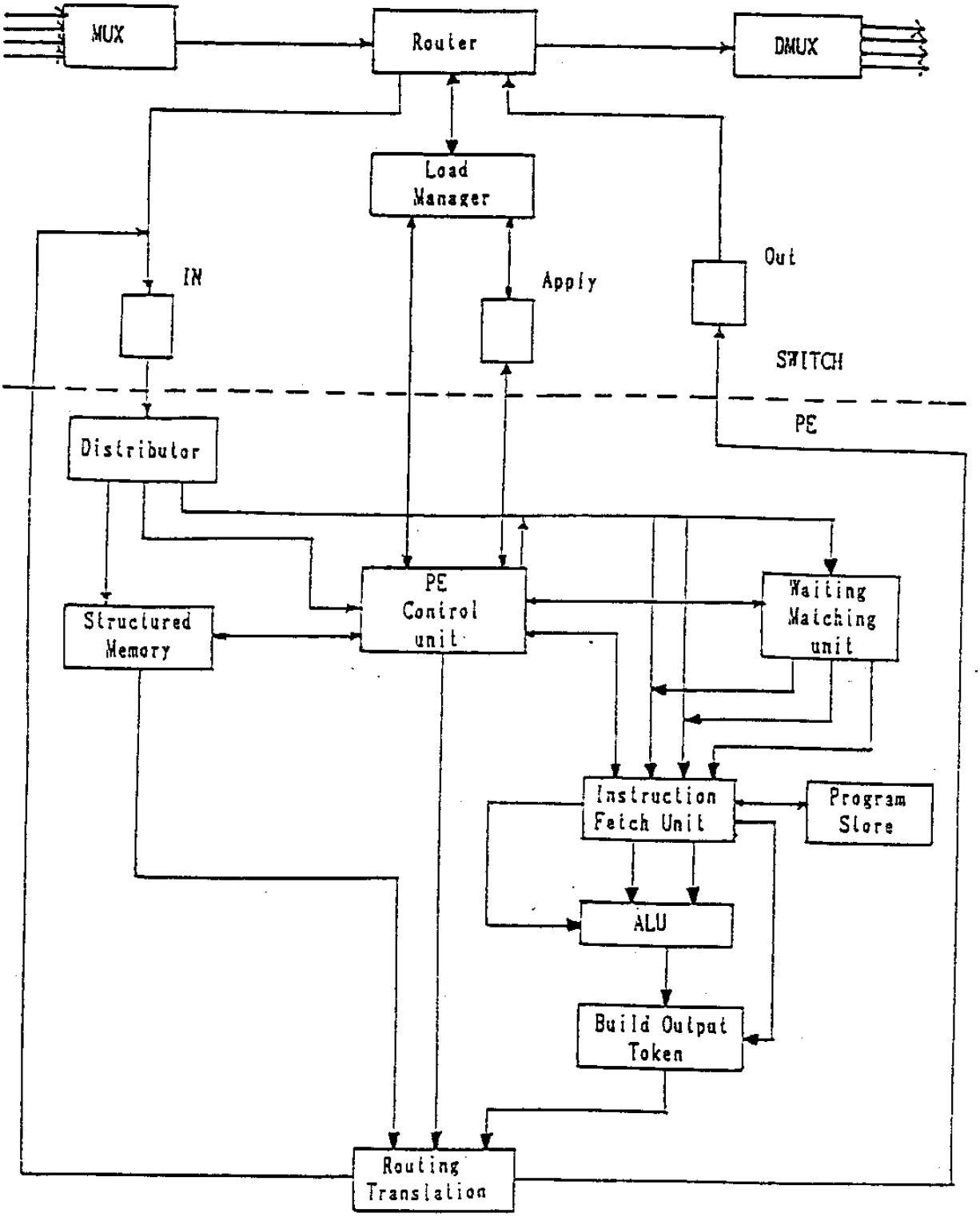


그림 4.27 노드 구조의 블럭다이아그램

크기의 버퍼(buffer)로 연결되어 각 부시스템 간의 속도 차이를 완충시켜 주며, send-acknowledge protocol로 교신한다. 또한 스위치는 단순히 패킷의 경로 설정과 전달을 행하는 router와 각 노드의 부하 균형을 담당하는 Load Manager. 그리고 다른 노드로 전달하거나 전달받은 태스크를 유지하는 Apply 큐로 구성되어 있다.

본 절에서는 HYPERDAC에서의 데이타 플로우 프로그램의 표현 방법과, 실제 노드 내에서 이동되는 데이타 토큰의 표현과 그러한 토큰이 노드에서 처리되는 과정에 대해 살펴본 후, HYPERDAC에 대한 간단한 성능 분석을 행한다.

(1) 프로그램의 표현

데이타 플로우 모델에서 프로그램은 데이타 플로우 그래프 형태로 표현되고, 채택한 모델이 기계어 수준의 병렬성을 이용하기 때문에, 함수 적용 계산단위를 제외하고 각 계산단위는 일반적인 컴퓨터 시스템에서 사용되는 기계어 형태가 된다. 그러나 데이타 플로우 모델에서는 해당 입력(피연산자)의 존재가 직접 기계어 실행의 기준이 되기 때문에, 대부분의 컴퓨터 시스템의 기계어 형태에서 존재하는 연산 코드, 피연산자 항목에, 부가적으로 해당 기계어 연산의 수행 후에 그 결과 값을 필요로 하는 계산단위에 관한 정보도 포함해야 하며, 이것은 기계어를 노드로 표현하는 그래프 상에서 노드와 노드를 연결하는 arc에 관한 정보를 나타내는 것과 동일하다.

다음은 프로그램 메모리(PS: Program Store) 내의 기계어에 대한 표현이다.

<Header>
<OPcode>
<Token-1-disposition>
<Token-2-disposition>
<Constant-disposition>
<Constant-source>
<Destination-list-flag>
<Constant-specification>

<Data-type>

<Data length>

<Data class>

<Data vlaue>

<Destination>

<Number-of-token-to-enable-instruction>

<Destination-instruction-port-number>

<Destination-virtual-address>

<Destination-list-flag>

<OPcode>는 연산 코드를 의미하고, <Token-1-disposition>과 <Token-2-disposition>은 이 기계어의 몇번째의 피연산자 인지를 표시해 주는 항목이고, <Constant-disposition>은 기계어가 상수를 피연산자로 사용하는 지의 표시이고, <Constant-source>는 상수 값을 기계어 표현 내에 포함하는지 아니면 IFU 내의 상수 영역에 상수 값을 저장하고 상수 영역에 대한 포인터를 기계어에 포함하는 지를 나타내고, <Data value>는 실제의 상수 값을 나타내며, <Constant-specification>에서 <Data-type> 부분은 해당 상수에 관한 내용을 포함하고, <Destination-list-flag>는 이 기계어 계산의 결과를 필요로 하는 기계어의 유무를 나타내고 만약 1이면 계산 값을 전달할 목표 기계어가 있다는 것을 나타내며, 목표 기계어에 대한 정보는 <Destination> 부분에서 제공되며, <Number-of-tokens-to-enable-instruction>은 목표 기계어의 실행에 필요한 피연산자 의 갯수를 나타내며, <Destination-instruction-port-number>는 해당 기계어의 계산 값이 목표 기계어의 몇번째 피연산자로 사용되는 지를 나타내며, <Destination-virtual-address>는 목표 기계어가 위치하는 주소를 나타내며, <Destination-list-flag>는 해당 기계어의 계산 값을 필요로 하는 목표 계산단위가 하나 이상 있을 때에는 1로 표시되고 또 다른 목표 기계어에 대한 <Destination> 부분이 명시되며, 목표 기계어가 하나이면 0으로 표시되고 기계어의 표현이 완성된다.

(2) 데이타 토큰(패킷)의 표현

동적 데이타 플로우 모델을 구현하는데 있어서, 데이타 플로우 프로그램 그래프를 흐르는 데이타 토큰은 실제의 값뿐만 아니라, 값이 전달되는 계산단위의 주소와 recursion 이나 반복 계산의 경우에 발생하는 코드 공유 문제 해결을 위한 부가적인 정보 (계산의 context, 반복수)도 포함한다.

구조화 메모리에 대한 요구 패킷과 처리요소의 제어장치의 서비스를 요구하는 시스템 패킷을 제외한 일반적인 데이타 토큰의 형태는 다음과 같다.

```
<Token-type>
<PE-number>
<Tag>
   <Color>
   <Instruction-address>
   <Initiation-number>
<Number-of-tokens-to-enable-instruction>
<Port-number>
<Data>
   <Data-type>
     <Data-length>
     <Data-class>
   <Data vlaue>
```

<Token-type>은 토큰의 형태를 나타내는 항목으로 구조화 메모리 요구 패킷, 시스템 패킷, 일반적인 데이타 토큰을 표시한다. <PE-number>와 <Tag> 부분의 <Instruction-address> 항목은 해당 토큰이 전달되는 목표 기계어의 위치를 나타내고, <Port-number>는 해당 토큰의 목표 기계어의 몇번째 피연산자가 되는 지를 나타낸다. <Number-of-tokens-to-enable-instruction> 항목은 해당 토큰이 전달되는 목표 기계어가 필요로 하는 피연산자의 갯수를 나타내며, 이 항목은 해당 토큰이 Waiting Matching unit을 거쳐서 IFU로 가야 하는지 아니면 곧장 IFU로 갈 수

있는지를 표시해 준다. <Tag> 부분의 <Color>는 목표 기계어 실행 context를 (즉, 목표 기계어가 속하는 코드 (혹은 태스크) 블럭의 context를 나타낸다) 표시하고, <Initiation-number>는 목표 기계어가 몇번째의 반복 실행에 속하는 가를 표시한다. <Data> 부분은 실제로 전달되는 데이타 값을 나타낸다.

(3) 노드 구조

그림 4.27의 노드 구조 블럭 다이아그램의 각 부시스템의 기능에 대한 설명은 다음과 같다.

a. Router

Router가 입력으로 받는 패킷은 다음의 3 가지 종류이다.

- 다른 노드로부터 받는 패킷
- LM(Load Manager)으로부터 받는 패킷
- RTU(Routing Translation Unit)로부터 받는 패킷

Router는 위에서 열거한 3 가지 종류의 패킷을 입력으로 받고, 각 패킷에 포함된 패킷 목적지를 확인한 후에 그러한 목적지에 도달하기 위해 적절한 경로를 선택하고 그것을 이웃하는 다른 노드들에 전송하거나, 그 패킷이 그 노드를 목적지(destination)로 하는 패킷일 경우에 제어(control) 패킷과 Apply 패킷은 LM으로 전달하고, 그외의 패킷은 IN 큐(Queue)에 저장한다.

입력으로 받는 패킷의 종류에 따라 Router가 취하는 행동은 다음과 같다.

i. 다른 노드로부터 받은 패킷

이 패킷은 크게 제어 패킷, Apply 패킷, 그리고 그외의 패킷으로 분류할 수 있다.

- 제어 패킷은 다른 노드에서 보내온 각 노드의 부하에 대한 정보 및 부하 이동에 관한 요구와 같은 제어 목적을 위한 패킷으로, Router는 그러한 제어

패킷을 LM에 전달하고, LM가 제어 정보를 처리하도록 한다.

- Apply 패킷은 다른 노드에서 그 태스크를 분산 처리함으로써 성능을 올릴 목적으로 전송한 패킷으로서, Router는 그 패킷의 목적지가 이 노드이면 그것을 LM에 보내서 그것을 Apply 큐에 저장하도록 하고, 그렇지 않으면 해당되는 노드에 대한 적절한 경로를 선택하여 주변 노드로 전송한다.
- 그 외의 패킷은 구조화 자료(Structured Data)에 대한 요구 패킷, PE 제어에 관한 패킷, 계산된 값을 갖는 패킷 등이며, Router는 그러한 패킷을 분산자(distributor)로 보내서, 분산자가 각 경우에 따라 적절히 배분하도록 한다.

ii. LM으로부터 받은 패킷

LM은 Router에 두 가지 종류의 패킷을 보낸다. 두 가지 종류의 패킷은 그 노드의 부하에 관한 정보 및 태스크 요구에 관한 제어 패킷과 분산 처리할 태스크를 포함 하는 Apply 패킷이다. Router는 제어 패킷을 주변 노드로 broadcasting 하고 Apply 패킷은 목적 노드에 적절한 경로를 선택하여 그것을 해당 주변 노드로 전송한다.

iii. RTU로부터 받은 패킷

이 패킷은 그 노드에서 처리한 결과를 요구하는 노드로 보내기 위한 패킷으로 Router는 목적 노드에 적절한 경로를 선택하여 그것을 해당 노드로 전송한다.

b. 분산자(distributor)

이것은 IN 큐에서 패킷을 취한 후 패킷의 종류에 따라 해당 장치로 단순히 분배하는 역할을 수행한다. 패킷의 종류는 구조화 메모리 (Structured Memory)에 대한 요구 패킷, 시스템 매니저(manager)나 I/O에 대한 요구를 포함하는 PE 제어 패킷, 결과 값을 갖는 패킷이고, 분산자는 그것들을 각각 구조화 메모리, PE 제어 장치(PE Control Unit), 그리고 WM(Waiting Matching unit)과 IFU(Instruction Fetch Unit)로 보낸다.

c. LM(Load Manager)

이것은 Gradient 부하 균형 방법을 관장하는 장치로서 PE 제어 장치를 통한 그 노드에 관한 부하 정보와 Router를 통한 다른 노드의 부하에 관한 정보를 토대로 Apply 큐에 있는 태스크를 분산처리 할 것인지를 결정하고, 분산 처리를 할 경우에는 목적 노드를 결정하여 Apply 큐에서 전송할 태스크를 선택하여 그것을 (Apply 패킷) Router에 보내고, 또한 그 노드로 향한 Apply 패킷은 Apply 큐에 삽입을 한다(여기서 주의해야 할 점은 LM이 시스템이 set_up된 상태 이후로 그 노드에 관한 부하 정보를 계속 유지하여야 한다는 점이다).

d. PE 제어 장치(Control Unit)

이것은 각 노드가 적절히 작동하도록 그 노드 내의 여러 장치를 제어하는 장치로서 특히 I/O와 매니저(color manager 등)와 관련된 작업 및 diagnostic 정보를 관리하는 작업을 수행하고 클럭(Clock)을 생성하고, 그 노드의 프로그램 메모리에 로드(load)된 태스크들과 그 태스크의 프로그램 메모리 상의 위치에 관한 정보도 관리,유지한다.

또한 IFU로부터 전달된 exportable 함수의 적용에 대해 그 함수의 인자로 쓰이는 구조화 자료에 대한 포인터와 함수의 이름을 Apply 큐에 삽입하고, 후에 LM으로부터 exportable 함수를 다른 노드로 전달하라는 신호를 받으면, exportable 함수에 대한 Apply 패킷을 생성하여, 그것을 Router를 통해 해당 노드로 전달하게 된다. 그리고 자기 노드에서의 수행을 위해 다른 노드로부터 전달된 Apply 패킷은 그 패킷에 포함된 함수의 인자를 구조화 메모리에 할당.기록하고, 그것에 대한 포인터와 함수 이름 Apply 큐에 삽입하고(만약 해당 함수가 노드에 로드되어 있지 않으면, 그노드의 입출력 장치를 통해 해당 함수를 프로그램 메모리에 로드하고 관련된 정보를 관리,유지하며, 프로그램 상의 주소를 IFU에 전달하여 IFU도 그 함수의 시작 주소를 관리,유지토록 한다), 시스템이 포화상태에 도달하여 throttling 이 시작되거나, 노드상에 처리할 작업이 어느 기준이하로 줄어들면, Apply 큐에서 태스크(함수)를 가져와서, 그 태스크의 인자에 대한 구조화 메모리 상의 포인터를 WM

에 전달하여 처리한다.

그리고 PE 제어장치는 노드내의 여러 부시스템에 관한 정보를 LM에 수시로 제공하여 LM가 노드의 부하에 대해 현실적인 판단을 행할 수 있도록 도와 준다.

e. 구조화 메모리(Structured Memory)

이것은 4.1 절에서 제시한 구조화 자료를 저장하고 조작하는 장치로, 그림 4.28에서 보는 바와 같이 구조화 메모리 제어 장치, 지체 쓰기 큐(DRR: Deferred Read Requests), 실제의 저장 장치인 RAM으로 구성된다.

구조화 메모리 제어장치는 배열 및 리스트의 할당과 조작을 총괄하여 관리하고, 구조화 메모리의 사용상태 등을 PE 제어장치를 통해 LM에 전달하여 LM이 구조화 메모리에 관한 현실적인 정보를 유지하여 정확한 부하 균형 정책을 수행토록 도와 준다. I-structure, E1, E2 구조 배열에 대한 읽기 요구는 해당 원소가 아직 쓰여지지 않은 경우에는 그 요구를 DRR 큐에 저장하고 관련된 포인터 조작 연산을 수행하고, 메모리에 이에 대한 포이터를 유지시키며(이것은 처음의 DRR의 경우에 해당하고 그 뒤의 DRR은 DRR 큐의 조작만 한다), 이미 해당 원소가 쓰여진 경우에는 그값을 취한 후 해당 목적 계산단위에 전달한다. I-structure, E1, E2 구조 배열에 대한 쓰기 연산은, 쓸 값을 해당 원소에 쓰고, 이미 그 원소에 대한 읽기 요구가 있는 경우에는, 그것들을 DRR 큐에서 제거하여 그값을 해당 목적 계산단위들에 전달한다. 그리고 Heap 구조 배열과 리스트에 대해서는 해당 구조의 할당과 reference counter 조작, garbage collection 작업 등을 수행한다.

f. WM(Waiting Matching unit)

데이타 플로우 모델에서 계산의 수행은 (입력) 데이타의 이용가능성에 (availability of data) 의해 결정된다. 따라서 데이타의 이용가능성을 탐지하는 작업은 매우 중요한 문제이다. 또한 recursion이나 loop 등을 코드 공유(code sharing)를 통해 처리하는 동적 태깅 모델(dynamic tagging model)에서는 이러한 탐지 작업이 보다 복잡해진다.

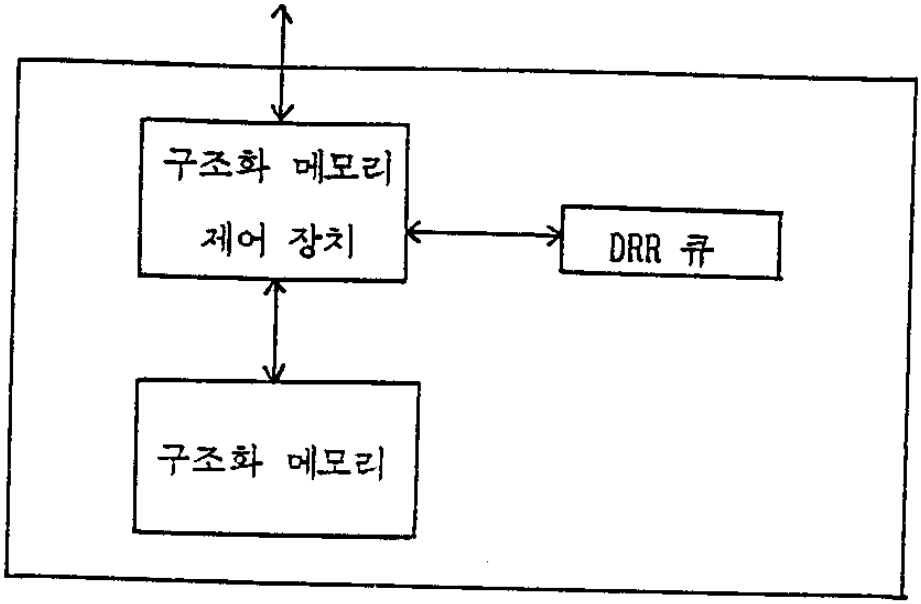


그림 4.28 구조화 메모리

데이타의 이용가능성을 탐지하는 작업을 2개의 입력 데이타를 요하는 계산단위를 기준으로 설명하면 다음과 같다.

. 임의의 계산단위(구체적으로 말하면 문제의 계산단위에 값을 전달하는 입력 계산단위)로 부터 생성된 값이 waiting matching unit에 도달된다.
. 도착한 값은 현재 waiting matching unit에 그값과 동일한 태그를 갖는 다른 값이 있나를 검토한 후, 만약 있으면 그값과 이미 waiting matching unit에 저장된 값을 함께 Instruction Fetching Unit에 보내고, 만약 없으면 그값은 waiting matching unit에 저장된다.

이와 같이 탐지하는 작업은 태그에 대해 행해지기 때문에 waiting matching unit은 CAM(Content-Addressable Memory)의 기능을 갖추어야 할 것이다.

그러나 CAM은 그것의 낮은 집적도와 비싼 비용 문제로 인해 현실적으로 waiting matching unit으로 구현하기가 어려운 실정이다. 따라서 본 연구에서는 Manchester 대학에서 채택한 hardware hashing 기법 [GUWA 80]을 사용하고 RAM으로 구현된 pseudo-associative store를 사용한다.

g. IFU(Instruction Fetch Unit)

이것은 WM에서 전달되거나(피연산자가 2 개인 연산의 경우), 직접 분산자에서 전달된(피연산자가 1 개인 연산의 경우) 데이타를 입력으로 받고 그 데이타가 사용하는 연산을(데이타 플로우 모델에서 데이타가 수행의 active 한 요인이기 때문에 이런 표현을 하였다) 프로그램 메모리에서 fetch 한 후 (해당 연산을 프로그램 메모리에서 용이하게 찾을수 있도록 각 함수에 대한 프로그램 메모리 상의 위치를 유지하는 코드 베이스 레지스터들도 각 함수별로 유지한다), 해당 연산과 피연산자를 ALU (Arithmetic Logic Unit)에 전달한다(일반적인 산술, 논리 연산은 해당 연산과 피연산자를 ALU에 전달하나, Apply 연산은 PE 제어 장치에 중재를 요구한다). 또한 해당 연산의 수행 결과 값을 보낼 연산에 대한 포인터는 BOT(Build Output Token) 장치로 보내서, ALU에서 생성된 결과 값이 그것을 요구하는 적절한 연산으로 보낼 수 있도록 한다.

또한 by-reference 메카니즘을 통한 지체연산(delayed evaluation)을 효과적으로 처리하기 위하여 IFU 내에 cell memory를 프로그램 메모리와 별도로 유지 하고, Instruction 내에 포함된 상수를 효과적으로 처리하기 위하여 별도의 상수 메모리와 각 태스크와 관련된 상수 영역에 관한 정보를 포함하는 상수 베이스 레지스터들을 둔다.

h. 프로그램 메모리(PS: Program Store or Memory)

이것은 프로그램을 저장하는 장치이다. 여기서 프로그램은 데이타 플로우 프로그램 그래프(dataflow program graph) 형태가 되며 각 태스크 별로 메모리상의 위치는 IFU가 코드 베이스 레지스타들을 통해 관리 유지하고 있으며 (이 정보는 컴파일시에 추출되어 시스템 수행 전에 Download/Initialization 단계에서 PE 제어 장치 및 IFU에 제공되거나 실행시에 필요에 따라 동적으로 로드되어, 관련된 정보가 PE 제어장치 및 IFU에서 관리,유지되기도 한다). 프로그램 메모리 내의 각 기계어에 대한 표현은 4.4.2의 (1)에서 자세히 제시되어 있다.

i. BOT(Build Output Token unit)

이것은 ALU로부터의 결과값과 결과값을 보내야 될 연산의 포인터를 IFU로부터 입력으로 받아 데이타 패킷을 형성하고 그것을 RTU로 보낸다.

j. RTU(Routing Translation Unit)

이것은 PE 제어 장치, 구조화 메모리, BOT로부터 입력 패킷을 받아 그 패킷의 주소 부분을 참조하여 적절한 노드 번호(number)를 결정하고 그 내용을 포함시킨 후, 해당 노드가 자신의 노드이면 그것을 IN 큐에 삽입 하고, 그렇지 않은 경우는 그것을 Router로 보낸다.

k. IN 큐

Router와 분산자간의 완충 역할을 한다.

l. Apply 큐

이것은 분산 처리할 태스크에 대한 정보를 포함하고 있는 큐로서 그노드 내에서 분산 처리를 요청한 태스크와 외부에서 그 노드에서의 처리를 요청한 태스크를 갖고 있으며, 그것의 제거(delete) 연산은 LM의 통제하에 PE 제어장치를 통해 이루어진다.

m. Output 큐

RTU와 Router 간의 완충 역할을 한다.

n. 기타의 큐

각 장치간의 작동 속도 격차를 완화시키기 위하여 여러 큐를 유지한다.

(4) 시스템 작동 시나리오 (System Operation Scenario)

앞서 제시한(4.3 절) DDFPL로 작성된 프로그램이 실제로 HYPERDAC에서 실행되어 결과가 도출될 때까지의 과정을 살펴보면 다음과 같다.

a. 컴파일 단계(Compilation stage)

DDFPL로 작성된 프로그램은 각 함수별로 컴파일되고, 함수 정의의 앞 부분에서 제시된 IMPORT 리스트의 함수들은 컴파일 단계에서 가능한 한 노드에 할당되는 것을 원칙으로 하고, 경우에 따라서 함수의 크기, 구조화 자료의 예상 크기, 예상되는 병렬도 등이 큰 함수는 주변의 다른 노드에 할당하는 형식으로 로드 모듈을 구성한다. 컴파일 단계에서 이러한 정보는 어느 정도 추출될 수 있다. 예를 들면 각 함수가 실행하는데 필요한 프로그램 메모리의 양은 그 함수 코드의 크기에 IMPORT 리스트 내의 함수들의 양의 합으로 표현할 수 있으며(또한 구조화 자료의 크기 및 병렬도는 프로그램 내의 병렬 구문의 분석을 통하거나, 필요에 따라 컴파일러가 그런 정보를 프로그램 사용자와의 대화를 통해 어느 정도 얻을 수 있다), 그 합이 어느 수준을 초과하면, 그중 큰 함수들부터 차례로 다른 노드로 할당하는 것을 고려할 수 있으며, 이러한 분석 과정을 계속하면 문제의 프로그램을

HYPERDAC 상에 할당할 수 있다(만약 전 프로그램의 크기가 시스템 내의 프로그램 메모리 양을 초과한다면, 각 노드가 일반적인 상용 컴퓨터에서처럼 가상 기억 장치 조작 기법을 사용할 수도 있겠으나, 그러한 극단적인 경우는 본 연구에서 고려하지 않았으며, 이러한 문제를 해결하고 보다 효과적인 컴파일러 기법에 대해서는 보다 연구가 수행되어야 할 것이다).

이렇게 구성된 프로그램의 로드 모듈은 고속 분산 처리 시스템의 전단 컴퓨터(FEC)에 이동되어 관리,유지되고(또한 이러한 정보가 각 노드의 입출력 장치에 중복,유지되어서 후에 실행할 때 필요에 따라 PE 제어 장치의 제어 하에 프로그램 메모리의 적절한 영역으로 로드되고, 그러한 정보가 PE 제어 장치 및 IFU에서 관리,유지된다), 그 로드 모듈의 입력 자료가 위치될 노드에 관한 정보도 컴파일 단계에서 추출되어 전단 컴퓨터에 이동되어 관리된다.

b. 로드 모듈의 시스템 적재 및 초기화 단계(Download/Initialization stage)

전단 컴퓨터에 이동된 로드 모듈은, 고속 분산 처리 시스템에서 실행되는 프로그램이, HYPERDAC에서 그 로드 모듈의 실행을 해당 입력 자료(실행 결과가 저장될 위치를 포함하여)와 함께 요구하면, 전단 컴퓨터는 그 로드모듈에 대한 프로그램과 데이타 패킷을 형성하여 HYPERDAC에 전달하고(또한 후에 그 로드 모듈의 실행 결과를 정확한 위치에 반환하기 위하여 HYPERDAC의 실행 결과가 반환될 주소를 할당하고 그 정보를 HYPERDAC에 프로그램과 데이타 패킷에 부가하여 전달한다), HYPERDAC의 각 PE 제어 장치는 컴파일러가 제공한 정보를 근거로 로드 모듈을 HYPERDAC 상에 적절히 로드하고, 해당 입력 자료는 그 입력 자료가 위치할 노드의 구조화 메모리에 할당, 저장하고 동시에 실행결과 값을 전단 컴퓨터에 반환할 시에 필요한 반환 주소에 관한 정보를 PE 제어 장치(대부분이 루트 노드의 PE 제어 장치가 된다)가 유지하며, 구조화 자료에 대한 포인터가 최초의 실행 시작 함수를 구동하기 위하여 WM으로 전달됨으로써 실행의 준비가 완료된다.

c. 실행 및 결과 반환 단계(Execution and Result-return stage)

위에서 말한 바와 같이 일단 최초로 수행을 시작할 함수에 대한 입력 자료가

준비되어 해당 목적 계산단위로 전달되면, 그 때부터 HYPERDAC은 실제로 프로그램 수행을 시작하고, 수행이 끝난 뒤에, 결과를 반환할 주소를 갖고 있는 노드의 PE 제어 장치가 수행 결과를 루트 노드를 통해 전단 컴퓨터에 반환함으로써 한 프로그램에 대한 작업을 종료한다.

4.4.3 성능 분석 (Performance Analysis)

본 절에서는 DAC 알고리즘의 특징을 분석하고 그 특징을 기초로, DAC 알고리즘의 실행시간에 대한 모델을 설정한 후, 순차 컴퓨터와 HYPERDAC에서의 DAC 알고리즘 실행시간에 대해 분석을 행하고, 문제점들을 살펴본다.

(1) DAC 알고리즘의 특징

DAC 알고리즘은 일반적으로 널리 알려진 문제 해결 방식(Problem Solving Technique) 중의 하나로, 그것은 복잡하고 큰 문제를 여러 개의 (비교적) 독립적인 부문제들로 분할(decomposition)한 후 각 부문제의 풀이들을 결합(composition)하여 문제를 푸는 방식을 취한다.

DAC 알고리즘은 일반적인 문제 해결 방식의 하나이므로 광범위한 응용 부분을 가지며, 알고리즘 성격 자체가 여러 부문제의 병렬 처리를 통한 성능 향상의 기회를 자연스럽게 제공하고 있다.

DAC 알고리즘의 일반적인 프로그램의 형태는 다음과 같다 [HOZO 83].

```
Procedure Dac(I:Structure; O:Structure)
   local S1,S2,...,Sk : Structure;
   local T1,T2,...,Tk : Structure;
     if SMALL(I)
        then  return(ANSWER(I;O));
        else
              Begin
```

```
            SPLITINPUT(I;S1,S2,...,Sk);
            cobegin
               Dac(S1;T1); Dac(S2;T2); ... ; Dac(Sk;Tk);
            coend;
            COMBINE(T1,T2,...,Tk,I;O);
        End
    end procedure Dac
```

위에서 보는 바와 같이 DAC 알고리즘의 성격은 recursive 하고, 수행시에 부문제의 수(분기의 수) 및 부문제와 관련된 자료의 양이 결정되고, 부문제들 간에는 거의 통신이 없다(독립적이다)는 점이다.

앞서 설명한 DAC 알고리즘의 일반적인 프로그램 형태에서 볼 수 있듯이, DAC 프로그램의 실행은 입력 I로부터 k개의 독립적인 구조 S1,S2,...Sk를 생성한다. S1,S2,...,Sk는 I의 변경되지 않는 copy의 k-decomposition으로 구성될 수도 있으며, I로부터 계산된 새로운 구조의 k-decomposition으로 구성될 수도 있다. 특히 후자의 경우, 컴파일 시간에 어떤 프로세서가 어떤 데이타를 access할 것인가를 결정할 수 없으며, 따라서 데이타를 특정 프로세서에 실행전에 미리 할당할 수 없다. 중요한 점은 partition이 입력 자료에 의존하기 때문에 일반적으로 그러한 자료의 선할당(preassignment)이 불가능하다는 것이다.

DAC 알고리즘의 실행시간에 대한 분석은 다음과 같다.

1. 입력 데이타의 복사와 실행 준비 시간 (T_{init})
   입력 자료의 크기에 따라 DAC 알고리즘은 두가지 실행형태를 갖는다.

   a. 입력 자료가 직접 계산할 수 있을 정도로 적은 경우:
      2. 단순한 직접계산 (T_{simp})
      3. 결과 반환 (T_{rct})

   b. 입력자료가 직접 계산하기에 큰 경우:

2. 분할 작업
   - 여러 부문제로 분할하는 계산 (T_{sp})
   - 부문제의 계산 (T_{call} or T_{com}, T_{sd})
     .순차적인 경우는 모든 부문제의 계산이 순차적으로 완료될 것을 가정하고,
     .병렬처리할 경우는 작업을 이동하여 계산하는 시간을 나타낸다. 따라서 통신에 소요되는 시간이 추가되며, 반면에 각 부문제의 계산은 동시 병렬처리가 가능한 경우에 가장 긴 시간이 걸린 부문제의 계산시간이 실제의 계산시간이 된다.
3. 부문제들의 결과를 결합하는 작업 (T_{comb})
4. 결합된 결과를 반환하는 작업 (T_{ret})

각 작업 설명의 오른쪽 괄호안의 기호는 각각 해당 작업에 소요되는 시간을 표시한다.

그림 4.29는 DAC 알고리즘의 실행 형태를 표시해 주는 그림이다.

위의 k-ary tree의 level을 L로 표시하고, 분지 갯수를 k로 했을 때 트리의 내부 노드의 갯수는 $(k^L - 1) / (k - 1)$ 이고, 말단 노드의 갯수는 k^L이고, 전체 노드의 갯수는 $(k^{L+1} - 1) / (k - 1)$ 이다.

(2) 순차 컴퓨터와 HYPERDAC의 성능 비교

현재 HYPERDAC의 구조에 대한 구체적인 설계가 완성되지 않았고, DDFPL에 대한 컴파일러도 구현이 안된 상태이기 때문에, 각 부시스템의 delay, 상호연결망의 capacity, 부하 균형의 영향, HYPERDAC의 패킷 생성율, 자료의 분할 양 (정도), 프로그램에 내재한 병렬도, 프로그램의 베이스 언어로의 정확한 번역 등에 관한 사항을 정확히 알 수 없으며, 따라서 본 분석에서는 이러한 것들에 대해 가정을 행한 후에 성능 모델을 제시하고, matrix multiplication을 입력 크기, HYPERDAC 상의 파이프 라인 stage 수, 함수 적용의 자체노드에서의 처리율, 하나의 피연산자를 갖

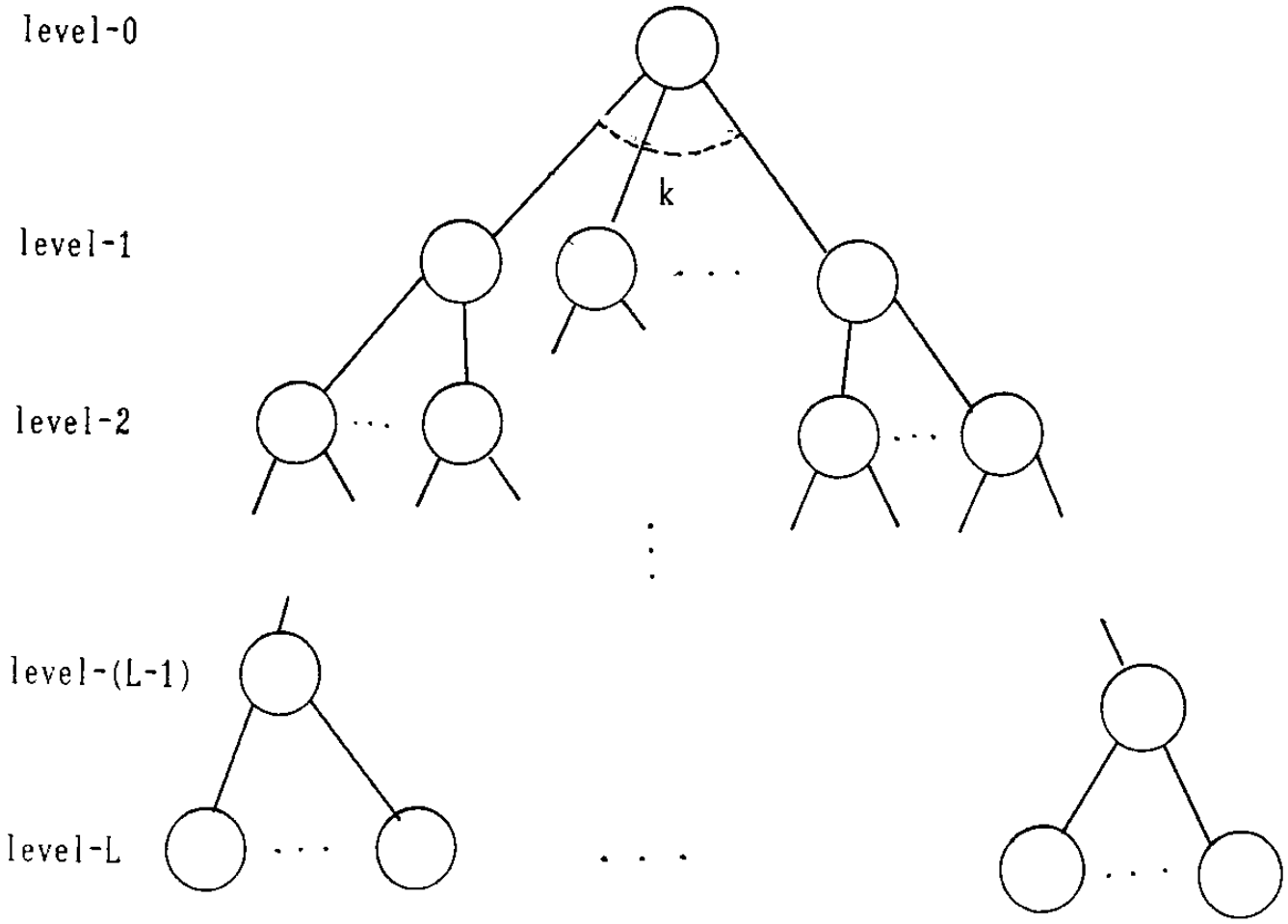


그림 4.29 DAC 알고리즘의 실행 형태

는 명령어의 확률 등의 관점에서 성능 분석을 행한다.

a. 가정

i. 프로그램의 실행시에 동적으로 행해지는 부하 균형 효과는 편의상 무시하고, DAC 알고리즘의 각 수준에서 함수적용을 다른 노드로 전달하여 처리할 확률은 P_f라 한다.

ii. HYPERDAC의 경우에 프로그램상에서 발생하는 모든 함수적용이 각 처리요소에서 처리될 만큼 처리요소의 갯수는 충분하다
(즉, 처리요소의 수 > $[(k - 1)^{L+1} - 1] / (k - 1)$).

iii. DAC 알고리즘의 각 수준별로 분할되는 자료의 양은 동일하다.

iv. 그외의 가정은 성능 비교를 논하면서 다룬다.

b. 순차 컴퓨터

DAC 알고리즘을 순차 컴퓨터에서 순차적으로 실행되는 경우에 실행 시간은 아래의 형태로 표현할 수 있다.

$$
\begin{aligned}
& T_{init} + T_{term} + \\
& T_{sp,0} + k\,T_{call,0} + T_{comb,0} + \\
& k\,(T_{sp,1} + k\,T_{call,1} + T_{comb,1} + T_{ret,1}) + \\
& k^2(T_{sp,2} + k\,T_{call,2} + T_{comb,2} + T_{ret,2}) + \\
& \qquad \cdots \\
& k^{L-1}(T_{sp,L-1} + k\,T_{call,L-1} + T_{comb,L-1} + T_{ret,L-1}) + \\
& k^{L}(T_{simp} + T_{ret,L})
\end{aligned}
$$

각 항목에 대한 설명은 다음과 같다.

T_{init} : DAC 알고리즘 실행의 초기화에 소요되는 시간
T_{term} : DAC 알고리즘 실행 결과를 반환하는데 소요되는 시간
$T_{sp,i}$: i-level에서 분할 계산에 소요되는 시간
$T_{call,i}$: i-level에서 부문제들에 대한 호출에 소요되는 시간

$T_{comb,i}$: i-level에서 부문제들의 실행결과를 결합하는데 소요되는 시간

$T_{ret,i}$: i-level에서 (i - 1)-level로 실행결과를 반환하는데 소요되는 시간

T_{simp} : L-level (말단 노드들)에서 수행되는 단순한 (직접) 계산에 소요되는 시간

L : DAC 알고리즘의 깊이

k : DAC 알고리즘의 분기의 수

c. HYPERDAC

본 절에서는 HYPERDAC에서 DAC 알고리즘을 수행할 시에 최악의 실행시간과 각 level 당 일정한 함수 적용율을 사용하는 경우의 실행시간에 대해 분석한다.

i. 최악의 경우

DAC 알고리즘을 HYPERDAC에서 모든 함수적용을 유용한 다른 처리요소로 전가하여 병렬로 처리할 경우에 실행에 소요되는 시간은 다음과 표현할 수 있다 (이 경우 통신과 관련된 overhead와 각 단계별로 최악의 실행시간이 계속되는 것을 가정하는 최악의 경우 분석이 된다).

$$
\begin{aligned}
& T_{start} + T_{term} + \\
& T_{sp,0} + T_{sd,0} + T_{com,0} + T_{comb,0} + \\
& T_{init,1} + T_{sp,1} + T_{sd,1} + T_{com,1} + T_{comb,1} + T_{ret,1} + \\
& T_{init,2} + T_{sp,2} + T_{sd,2} + T_{com,2} + T_{comb,2} + T_{ret,2} + \\
& \qquad\qquad \cdots \\
& T_{init,L-1} + T_{sp,L-1} + T_{sd,L-1} + T_{com,L-1} + T_{comb,L-1} + T_{ret,L-1} + \\
& T_{init,L} + T_{simp} + T_{ret,L}
\end{aligned}
$$

각 항목에 대한 설명은 다음과 같다.

T_{start} : DAC 알고리즘 실행을 위해 전단 컴퓨터로부터 프로그램과 데이타 패킷을 받아 HYPERDAC 상에 적절한 위치에 로드하고 실행준비를 완료하는데 소요되는 시간

T_{term} : HYPERDAC에서 실행된 결과를 전단 컴퓨터로 반환하는데 소요되는 시간

$T_{init,i}$: i-level에서 함수적용을 실제로 수행하기 위해서 해당 입력자료를 i-level 처리요소의 구조화 메모리 상에 할당하여 실행이 준비되는데 소요되는 시간

$T_{sp,i}$: i-level에서 분할계산에 소요되는 시간

$T_{sd,i}$: i-level에서 (i + 1)-level로 자료를 이동시킬 때 실제로 전달될 때까지 i-level에서 지체되는 시간

$T_{com,i}$: i-level에서 (i + 1)-level로 부문제에 대한 자료를 전달하는데 소요되는 시간

$T_{comb,i}$: (i + 1)-level에서 반환된 값을 i-level에서 결합하는데 소요되는 시간

$T_{ret,i}$: i-level에서 부문제의 실행결과를 (i - 1)-level로 반환하는데 소요되는 시간

$T_{simp,i}$: L-level에서 수행되는 단순한 (직접) 계산에 소모되는 시간

각 항목에 대한 근사치는 다음과 같다 (자세한 유도 과정은 평이하기 때문에 생략한다).

$$T_{init} = (M / (k - 1))\ t_{di}$$

$$T_{sp} = (kM / (k - 1))\ (t_{ap} / (P \text{ or } D_s))$$

$$T_{sd} = C_3(kL / l)\ t_{qda}$$

$$T_{com} = (M / (k - 1))\ t_{fo} + (M / (k - 1))\ (l / C)\ AD\ t'_{sd}$$

$$T_{comb} = (kM / (k - 1))\ (t_{ap} / P\ (\text{or } D_c)) + (M / (k - 1))\ t_{fi}$$

$$T_{ret} = (M / (k - 1))\ t_{fo} + (M / (k - 1))\ (l / C)\ AD\ t'_{sd}$$

$$T_{simp} = (n_p / P\ (\text{or } D_p))\ t_{ap}$$

각 기본적인 시간 항목에 대한 설명은 다음과 같다.

t_{di} : 단위 입력 자료를 PE 제어 장치를 통해 구조화 메모리에 기록하는데 소요되는 시간

t_{fo} : i-level에서 (i + 1)-level로 이동할 자료들에 대한 단위 자료당 패킷 형

성 시간

t'_{sd} : 출발 노드에서 목표 노드로 가면서 각 노드의 switch에서 소요되는 시간

t_{fi} : 부문제로부터의 결과값이 단위자료당 할당된 구조화 메모리의 영역으로 채워지는데 소요되는 시간

t_{qda} : Apply 큐에 힘수 적용 패킷을 저장하고 제거하는데 소요되는시간

t_{ap} : 평균적으로 하나의 계산단위 (기계어)를 처리하는데 소요되는 시간

그 외의 항목에 대한 설명은 다음과 같다.

D_s : 분할 계산의 병렬도

D_c : 결합 계산의 병렬도

D_p : 단순한 직접 계산의 병렬도

n_p : 단순한 직접 계산과 관련된 기계어의 수

C_3 : T_{sd}의 표현에 필요한 상수

C : 노드 당 일정한 bandwidth를 가정할 때, 링크 수와 링크 당 bandwidth의 곱

l : 노드 당 링크의 수

AD : 시스템내에서 노드와 노드간의 평균 거리

ii. 각 level 당 일정한 함수 적용률을 사용하는 경우

간단히 계산의 지역성 이용 측면과 통신 비용을 감안하기 위하여, DAC 알고리즘 수행시에 각 level 당 다른 노드에 함수 적용을 전가하여 실행하는 비율(P_f)을 사용하는 경우에 실행시간은 다음과 같이 표현된다 (지역적으로 처리하는 계산이 실제의 최장시간에 간여되고, 자체노드의 지역적 계산과 다른 노드에 전달되어 수행되는 계산이 중첩되어 실행되는 것을 가정한다).

$$T_{init} + T_{term} +$$
$$T_{sp,0} + (1-P_f)\ k\ T_{call,0} + T_{comb,0} +$$
$$(1-P_f)\ k\ (T_{sp,1} + (1-P_f)\ k\ T_{call,1} + T_{comb,1} + T_{ret,1}) +$$
$$\{(1-P_f)\ k\}^2(T_{sp,2} + (1-P_f)\ k\ T_{call,2} + T_{comb,2} + T_{ret,2}) +$$

...

$\{(1-P_f)\ k\}^{L-1}(T_{sp,L-1} + (1-P_f)\ K\ T_{call,L-1} + T_{comb,L-1} + T_{ret,L-1}) +$

$\{(1-P_f)\ k\}^{L}(T'_{simp} + T_{ret,L})$

각 시간 항목에 대한 구체적인 내용은 다음과 같다.

$T_{sp,i} = (M / k^{i+1})\ (t_{ap} / P)$

$T_{call,i} = t_{call\ overhead}$

$T_{comb,i} = (M / k^{i+1})\ (t_{ap}/P)$

$T_{ret,i} = t_{return\ overhead}$

$T'_{simp} = 8\ k\ t_{ap} / P$

이 경우에 각 항목에 대한 합은 다음과 같다.

$T_{sp} = (M/k)\ (t_{ap}/P)[1+(1-P_f)\{1-(1-P_f)^{L-1}\}/P_f]$

$T_{call} = [(1-P_f)k]\{[(1-P_f)k]^{L}-1\}/[(1-P_f)k-1]t_{call\ overhead}$

$T_{comb} = T_{sp}$

$T_{ret} = [(1-P_f)k]\{[(1-P_f)k]^{L}-1\}/[(1-P_f)k-1]t_{return\ overhead}$

$T_{simp} = [(1-P_f)k]^{L}(8k)t_{ap}/P$

d. 성능 분석

4.3절에서 제시된 matrix multiplication 프로그램에 대해 그것의 크기, HYPERDAC의 파이프라인 stage 수, 함수 적용의 다른 노드로의 전가율 (Pf) 등에 대해 순차 컴퓨터와 HYPERDAC의 성능을 분석한 결과는 다음과 같다.

HYPERDAC 에서 최악의 경우를 가정한 경우에 문제의 입력 자료 크기에 관계없이 일정하게 5배 정도의 계산속도의 향상이 있었으나, 입력크기에 따라 (DAC 알고리즘의 깊이에 따라) 늘어나는 처리요소의 증가에 대한 보상이 전혀 없으며 따라서 efficiency는 과도하게 저하된다. 이것이 계산의 지역성 이용과 과도한 함수 적용으로 인한 통신비용의 낭비를 줄이는 것이 필요하며 아울러 위의 두가지 문제를 실행시에 적절하게 조화시킬 수 있는 부하 균형 방법과 함수 적용시 인자들의 과도

한 복사를 방지할 수 있는 기법이 필요하고, 또한 문제의 계산 구조를 시스템에 보다 효과적으로 사상할수 있는 컴파일러 기법이 필요하다는 사실을 말해 준다.

각 level 당 일정한 함수 적용 전가율을 사용한 경우에 앞의 경우보다는 계산속도가 두드러지게 향상 되었으나, 계산속도의 증가율이 처리요소의 증가율을 따라잡지 못해서, 이 경우도 역시 level 이 깊어질수록 efficiency 가 급격히 저하되고 있는데, 이 경우는 본 분석에서 제시한 level 당 일정한 함수 적용 전가율의 가정이 실제의 실행시에 별로 도움을 주지 못하며, 시스템 내의 처리 요소의 이용도가 저하되기 때문이다. 따라서 보다 유연성 있는 granularity의 설정과 시스템 부하에 따라 동적으로 부하 균형을 행하는 효과적인 분산 부하 균형 정책이 필요함을 말해준다. 최악의 경우와 각 level당 일정한 함수 적용 전가율을 사용한 경우의 분석은 우리가 채택한 계산방법의 특징인 계산의 지역성 이용 측면과 병렬성의 제어 측면을 고려하지 않은 아주 거칠은 분석이어서, 우리의 계산 방법이 앞서 분석한 결과보다 성능이 우월할 것이라고 확신할 수 있으나, 채택한 계산방법이 실행시에 동적으로 적용되는 측면이기 때문에, 상세한 구조와 컴파일러가 구현이 되지 않은 현 상황에서 위의 두가지 경우에 대한 분석을 통해 HYPERDAC의 성능을 개선할수 있는 여러 요인들을 발견할 수 있었다.

4.5 결론 및 향후 연구 계획

Divide-And-Conquer (DAC) 알고리즘을 효과적으로 수행하면서 일반적인 다른 알고리즘도 무리없이 수행할 수 있도록 제안된 고속의 다중처리 시스템인 HYPERDAC은 DAC 알고리즘의 병렬성을 고도로 이용하기 위하여 여러 수준의 병렬성을 이용할 수 있는 데이타 플로우 모델을 계산 모델로 채택하고, DAC 알고리즘의 논리적 계층 구조를 활용하고 일반적인 다른 알고리즘에의 적용도 용이하도록 Hypertree를 상호연결망으로 채택하고, 각 처리요소에 부과된 부하를 균등하게 함으로써 전체 시스템의 이용도와 성능을 올리는 분산 부하 균형 정책으로 Gradient 부하 균형 정책을 채택하였으며, 제한된 자원하에서 과도한 병렬성을 억지하고 계산의 지역성을 이용함으로써 시스템 자원의 효율적 사용을 고양하는 throttling 기법을 채택한다.

본 연구에서는 이러한 특징을 갖는 HYPERDAC 시스템을 실제로 구현하는데 초석이 될 베이스 언어와 프로그래밍 언어를 설계하고, 간단한 성능 분석을 통하여 향후 연구에 대한 문제점들을 추출하였다. 따라서 앞으로는 이러한 연구를 토대로 다음과 같은 연구가 보강되어야 할 것이다.

- 순수한 데이타 구동형 모델의 단점을 제거하고 리덕션 수행 모델 및 폰 노이만 계산 모델의 장점을 취합하는 새로운 통합 병렬 모델에 대한 연구
- 계산단위의 granule과 상호 연결망과의 관계를 고려한, 유연하고 효과적인 granule과 상호 연결망의 설정
- 프로그램의 효과적인 분할 및 사상 기법에 관한 병렬 컴파일러에 관한 기법
- 시스템의 유용도를 효과적으로 향상시킬 수 있는 분산 부하 균형 정책 및 부하상태의 결정에 관련된 인자의 규명 및 그들의 관계 설정
- 제한된 자원 하에서 병렬성을 하드웨어나 소프트웨어 기법을 이용하여 효과적으로 제어할 수 있는 throttling 기법
- 시스템 원형 및 시스템 소프트웨어의 구현

5. 고속 고신뢰 상호 결합망

5.1 상호 결합망

5.1.1 개관

고속 분산 처리 시스템은 다수의 범용 컴퓨터와 전용 컴퓨터 그리고 전용 컴퓨터의 통신을 지원하기 위한 전단 컴퓨터 들로 구성된다. 고속으로 수행되는 math package 전용 컴퓨터와 DAC 전용 컴퓨터의 기능을 범용 컴퓨터가 효과적으로 이용하기 위해서는 보다 빠르고 신뢰성이 높은 상호 결합망이 본 고속 분산 처리 시스템에서 구성되어야 한다. 따라서 데이타 전송률이 50 Mbps인 고속 근거리 통신망을 이중으로 구성함으로써 컴퓨터들 간의 통신속도를 향상시켜 처리량을 증가 시킬 수 있도록 한다. 또한 한개의 통신망에 결함이 발생한 경우에도 나머지 한개의 통신망으로 통신이 원활히 수행될 수 있도록 함으로써 신뢰성이 높은 상호 결합망을 구성한다(그림 5.1 참조).

5.1.2 구조

(1) 고속 근거리 통신망

본 고속 분산 시스템의 상호 결합망으로써 구성되는 고속 근거리 통신망의 일반적 특성은 다음과 같다.

ㄱ) 높은 데이타 전송률

10 Mbps가 제공되는 일반 근거리 통신망과는 달리 30-70 Mbps정도의 데이타 전송률이 제공된다[WILL 87].

ㄴ) 고속의 인터페이스 장치

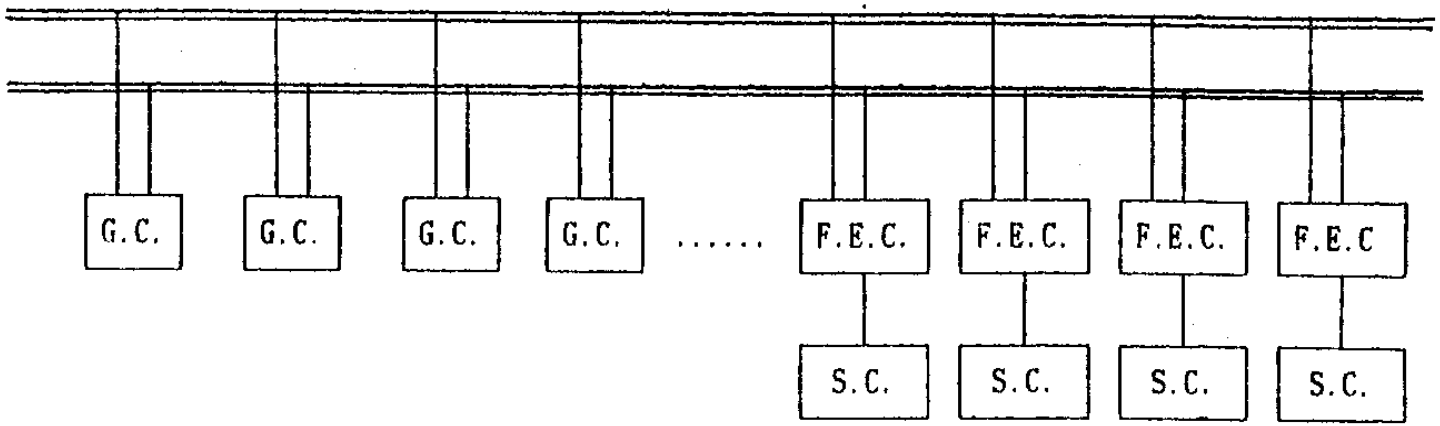
호스트 컴퓨터와 통신망의 접속시 고속의 물리적 회선과 인터페이스 장치가 필요하다.

ㄷ) 분산 액세스

근거리 통신망의 경우와 같이 분산 엑세스 제어 방법을 사용함으로써 신뢰성과 효율이 향상된다.

ㄹ) 적용 거리 제한

통신의 효율을 향상시키기 위해 적용 범위는 건물의 한 층 혹은 한 개의 연구실로 제한된다.



G.C. : 범용 컴퓨터

F.E.C. : 진단 컴퓨터

S.C. : 전용 컴퓨터

그림 5.1 고속 고신뢰 상호 결합망

(2) 동축 케이블 근거리 통신망

통신망의 전송 매체로는 트위스트 페어, 광섬유, 동축 케이블 등이 주로 사용되고 있다. 트위스트 페어 회선은 고속의 통신 수행에 적당하지 못한 반면 광섬유는 100-300 Mbps 정도의 데이타 전송률이 보장된다. 그러나 높은 데이타 전송률의 특성을 충분이 활용할 수 있는 통신 인터페이스 장치와 매체 제어 프로토콜이 개발 단계이며 가격대 성능면에서도 다른 매체에 비해 유리하지 못하므로 본 보고서에서는 일반적으로 많이 사용되고 있는 동축 케이블을 대상으로 통신망을 설계하였다. 1990년대에는 광섬유를 이용한 FDDI(Fiber Distributed Data Interface) 기술이 정착되어 널리 보급되리라 예상된다. 동축 케이블을 사용하는 기존의 근거리 통신망의 경우 10 Mbps의 데이타 전송률이 제공된다. 그러나 고속 근거리 통신망(HSLN; High Speed Local Network)에서는 50 Mbps의 데이타 전송률이 보장된다. 물론 데이타 전송률이 10 Mbps인 경우와 50 Mbps인 경우의 동축 케이블의 기계적, 전기적인 특성은 각각 50 ohm 과 75 ohm으로써 상이하다. 현재 사용되는 고속 근거리 통신망은 대부분 동축 케이블 버스 구조이며 버스 구조의 근거리 통신망과 유사한 점이 많다. 특히 패킷 스위칭과 다중 액세스 매체를 위한 매체 액세스 제어(medium access control) 기법이 보편적으로 사용된다. 물론 근거리 통신망과 고속 근거리 통신망의 가장 큰 차이점은 데이타 전송률에 있다. 그리고 고속 근거리 통신망에 있어서도 브로드 밴드와 베이스 밴드의 적용이 가능하다. 본 고속 고신뢰 상호 결합망에서는 베이스 밴드 방식을 사용하며 그 주요 특성을 다음과 같이 기술한다.

ㄱ) 50 Mbps의 데이타 전송률

전송속도가 빠르기 때문에 고속 근거리 통신망의 설치 비용은 일반 근거리 통신망보다 많이 든다.

ㄴ) 적용 최대 거리 1 km 정도

높은 데이타 전송률을 효율적으로 사용하기 위해 실제 적용되는 거리는 1 km 내외로 한다.

ㄷ) 접속되는 호스트 컴퓨터 수의 제한

통신의 효율을 위해 접속되는 호스트의 수는 수십개(10-30개) 이하로 한다.

ㄹ) 다수의 케이블을 지원

2 개 이상의 동축 케이블을 사용할 수 있으므로 처리량과 신뢰도를 증가시킬 수 있다.

근거리 통신망이 범용의 통신망으로 사용되는 반면 고속 근거리 통신망은 특수 목적용 통신 시스템에 적합하다. 높은 데이타 전송률이 제공되는 고속 근거리 통신망을 적은 범위 내에서 구성되는 본 고속 분산 시스템에 적용함으로써 범용 컴퓨터와 전용 컴퓨터 간의 통신은 물론 범용 컴퓨터 간의 통신에도 고속의 통신이 수행되도록 한다. 뿐만 아니라 이중의 고속 근거리 통신망을 사용함으로써 처리량의 증가는 물론 신뢰도가 더욱 향상되도록 한다[JAME 80].

5.1.3. 통신 기법

(1) 충돌 회피 CSMA

이더네트에서는 CSMA(Carrier Sense Multiple Access)/CD(Collision Detection) 기법이 사용되고 있다. 그러나 고속 근거리 통신망의 경우에는 ANSI의 표준에 명시되어 있는 충돌회피 CSMA 기법을 사용한다[WILL 83]. 이 프로토콜은 CSMA 를 기본으로 하며, 따라서 매체를 통하여 전송하려는 컴퓨터는 먼저 다른 통신이 있는지를 살피고 전송하는데, 여기서는 매체가 쉬고 있을 때 충돌을 방지하는 알고리즘을 추가시킨다. 이 방법을 따르면 각 컴퓨터(또는 포트) 들은 물리적 순서와는 관계없이 순서를 갖게 된다[port(1), port(2), ... , port(N)]. 한 컴퓨터에서 전송을 한 후에는 매번 이러한 순서가 정해지며, 이러한 초기화 작업 후에 각 컴퓨터는 자기의 순서를 기다려서 차례가 오면 전송한다. 그러므로 port(I+1) 은 port(I)가 전송할 기회를 잡은 다음에야 전송이 가능하다. 이 때의 기다리는 시간은 다음 3 가지의 합으로 구성된다.

- o port(I)가 전송을 시작할 수 있기까지의 시간 [port(I-1)에 대한 전송 기회에 입각하여]
- o port(I)가 전송 기회를 갖는 동안의 포트 지연 시간
- o port(I)와 port(I+1) 사이의 전파 지연 시간

이를 보면 개념이 아주 간단해 보이나, 몇 번 수정을 거치면 복잡한 시스템이 된다. 한 전송이 이루어진 후에 port(1)은 전송을 할 수 있는 권리를 갖는다. 그런데 특정 시간 동안 전송하지 않으면, 이번에는 port(2)가 기회를 잡는 등 이런 식으로 계속 진행되다가 어떤 port(I)에서 전송이 이루어지면 다시 순서를 정한다(재초기화). 그리고 port(I)가 port(J) ($I \neq J$)로 여러 개의 프레임을 연속적으로 전송하고자 하

는 경우에 port(I)에 우선 순위를 부여함으로써 채널 사용시 계속적인 통신이 가능하도록 할 수 있다. 즉 범용 컴퓨터가 전용 컴퓨터로 대량의 데이타를 전송하고자 하는 경우 여러 개의 프레임으로 구성한 후 통신망을 계속 사용하여 전송할 수 있다. 따라서 처리할 데이타를 모두 수신한 전용 컴퓨터는 즉시 연산을 수행할 수 있게 된다.

(2) 충돌 회피 알고리즘

ANS X3T9.5 에서 제정한 매체 액세스 제어 프로토콜은 CSMA를 기본으로 하는 충돌 회피 알고리즘을 사용하고 있다. 이 방법외에도 여러가지 알고리즘이 사용되긴 하나 본 고속 고신뢰 상호 결합망에서는 성능이 개선된 ANS X3T9.5의 충돌 회피 알고리즘을 이용한다. 이 방법은 모든 컴퓨터들이 보다 공정하게 통신망을 사용하도록 하고 있다. 즉 방금 전송한 컴퓨터는 다른 컴퓨터들이 모두 기회를 갖기 전에는 전송 기회를 잡지 못하도록 한다. 알고리즘을 보다 간략하게 나타내기 위해 다음과 같이 용어를 정의한다.

o 우선 순위 액세스 기회 (priority access opportunity) : 한 프레임을 수신한 후 포트에 승인되는 기간. 이것은 acknowledge 프레임이나 다중 프레임 (multiframe)에 사용된다.
o PAT (priority access timer) : 우선 순위 액세스 기회의 시간을 측정하는 데 사용된다. 즉 64 비트 시간.
o 중재 액세스 기회 (arbitrated access opportunity) : 각 포트에 차례로 부여되는 전송 가능 시간. 즉 16 비트 시간이며 충돌을 방지하도록 할당된다.
o AAT (arbitrated access timer) : 각 포트에게 액세스 기회가 겹쳐지지 않도록 제공하는데 사용된다.
o RT (resynchronization timer) : 가장 늦은 전송이 도달하기 까지의 다른 모든 타이머의 활동을 저지하는 역할을 한다.
o WF (arbiter wait flag) : 공정성을 기하기 위하여 사용된다. 하나의 포트가 전송할 때 그의 WF는 설정되어 다른 모든 포트들이 전송기회를 가질 때까지 전송하지 않도록 한다.

그림 5.2에서 타이머의 대문자는 변수를 나타내며, 소문자는 특정한 값을 나타낸다. 타이머가 규정된 최대의 값에 도달했을 때 종료되었다고 한다. 이 기법은 다중 프

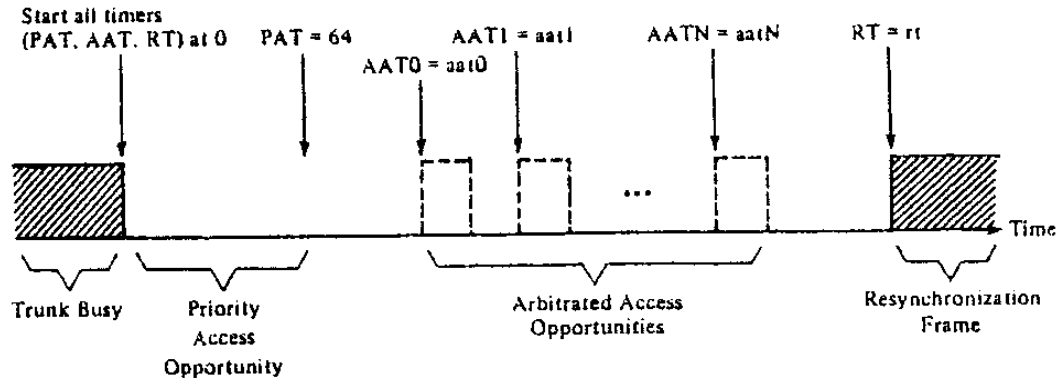


그림 5.2 ANS X3T9.5 매체 액세스 프로토콜의 동작

레임을 사용하여 큰 화일들을 빠르게 전송할 수 있다. 또한 적은 수의 컴퓨터들이 접속되어 있으므로 라운드-로빈 방식으로도 길지않은 지연 시간안에 처리할 수 있다.

(3) 충돌 회피 알고리즘의 동작

전송 매체상의 각 포트는 각각 세개의 타이머(PAT, AAT, RT)를 가지고 있다. 매체가 active 상태인 경우에는 모든 포트의 타이머들이 0으로 초기화되며 매체가 idle 상태인 경우에 세 개의 타이머가 작동되기 시작한다. 이 때 프레임을 최종적으로 수신한 포트는 우선 순위 액세스 기회를 가지게 되며 PAT 시간이 종료되기 전까지 전송을 시작할 수 있다. 물론 이 경우에 다른 포트가 전송을 시작한다면 모든 포트의 타이머들은 리세트 상태가 되고 처음부터 다시 작동이 된다. 각 포트는 중재 액세스 기회의 시작을 나타내기 위하여 AAT 값이 할당되어 있다. 이 값은 포트(0)부터 시작하여 포트의 순서에 따라 aat0 < aat1 < ... < aat N 의 순서로 값의 크기가 결정된다. 포트(0)의 시간이 종료되기 전까지 WF 신호가 설정되어 있지 않고 전송할 프레임이 있다면 포트(0)는 전송을 시작할 수 있다. 이 때 전송 준비 시간은 16 비트 시간이 된다. 그리고 포트(0) 는 전송 개시 후 WF 신호를 1로 설정한다. 이 신호는 포트의 RT 시간이 종료될 때까지 해제되지 않는다. 즉 다른 모든 포트들이 한번씩 전송 기회를 갖기 전에는 전송을 개시했던 포트가 중재 액세스 기회 동안 다시 전송할 기회가 주어지지 않게 됨을 의미한다. 그러나 프레임을 전송받은 때의 우선 순위 액세스 기회의 경우에는 전송을 시작할 수 있다. 포트(0)는 중재 액세스 기회 동안 전송할 프레임이 없거나 WF 신호가 1로 설정되어 있는 경우에는 포트(1)로 기회가 넘어간다. 포트(1)은 포트(0)의 경우와 동일한 조건으로 AAT1 시간 (aat1)이 종료되기 전까지 전송을 시작할 수 있다. 모든 포트들이 해당 종료 시간 동안 프레임을 전송하지 않거나 각 WF가 1로 설정된 경우에는 RT에 의해 재초기화 작업이 수행된다. 즉 특정 포트의 RT가 종료될 때 WF와 AAT 시간을 리세트 상태로 만든다. 그리고 그 포트의 AAT 시간이 종료될 때 프레임을 전송하여 모든 포트의 타이머들을 리세트 상태로 만든다. 각 포트에 있는 세 개의 타이머들의 종료 시간은 전파 지연 시간을 고려하여 설정되며 그에 대한 일반적인 전략은 다음과 같다.

o pat는 모든 포트에 대하여 동일하다 - 64 비트 시간
o aat0는 임의의 포트로 부터 우선 순위 전송이 포트(0)에 도착할 때 기다리게 되

는 시간이 설정된다.

o aat I(I > 0)는 포트(I-1)로 부터 중재 액세스 전송이 포트(I)에 도달할 때 까지 기다리게 되는 시간이 설정된다.

o rt는 모든 포트에 대하여 동일하며 포트(N)으로 부터의 중재 액세스 전송이 모든 포트에 도달하게 될 때 까지 기다리게 되는 시간이다.

위의 조건들을 만족시키기 위한 단순한 기법으로는 각 타이머에 2T + DP 시간 만큼을 증가하여 할당하는 것이다. 이 때 T는 양단간(end-to-end) 전파 지연 시간이며 DP는 포트가 전송 준비 작업을 하는데 걸리는 시간이다. 그러나 이 개념은 최적의 기법이 아니므로 ANSI 표준에서는 임의의 두 포트 간의 실제 전파 지연 시간에 의거한 기법을 제안하였다. 통신망에 있는 포트의 물리적인 순서에 따라 전파 지연 시간의 차이가 생기므로 본 보고서에서는 대기 시간을 최소화하기 위하여 포트의 실제적인 순서와 통신망 상에서의 위치가 일치하도록 한다(그림 5.3 참조). 따라서 이 경우에는 다음과 같은 공식이 유도된다.

pat = 64 단위 : 비트 타임

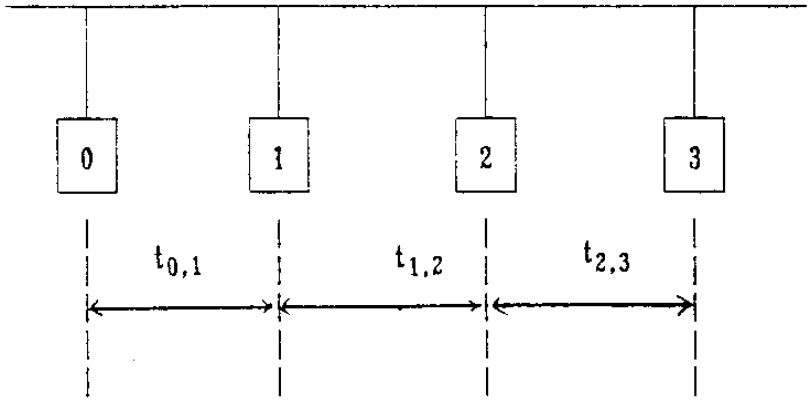
aat0 = 64 + $2t_{0,f}$

aatI = aat(I-1) + $2t_{I,I-1}$ + DP, i <= I <= N

rt = aatN + $2t_{N,f}$

(단 $t_{I,f}$는 포트(I)와 그 포트로 부터 가장 넓리 멀리 떨어진 포트 간의 전파 지연 시간임)

이와 같이 ANS의 매체 액세스 제어 기법에서는 충돌을 회피할 수 있고 우선 순위 액세스 기회를 이용하여 용량이 큰 화일도 전송할 수 있는 장점이 있다. 반면에 CSMA/CD 방식보다 포트의 대기 시간이 지연될 수 있다. 즉 포트(I)의 경우 앞의 포트들이 전송을 모두 하지않은 경우에는 aatI 시간이 불필요하게 지연되었다고 간주할 수 있기 때문이다. 그러나 고속 근거리 통신망에서는 접속된 포트의 수가 적고 지연 시간의 상한값이 정해질 수 있기 때문에 지연 시간으로 인한 오버헤드는 크지 않다고 볼 수 있다.



PAT = 64 비트 타임
DP = Port Delay = 16 비트 타임
AAT0 = PAT + $2t_{0,3}$
AAT1 = AAT0 + $2t_{0,1}$ + DP
AAT2 = AAT1 + $2t_{1,2}$ + DP
AAT3 = AAT2 + $2t_{2,3}$ + DP
RT = AAT3 + $2t_{2,3}$ + DP

그림 5.3 포트의 배열과 AAT 시간 설정

5.2 통신 인터페이스 설계

5.2.1 논리적 인터페이스

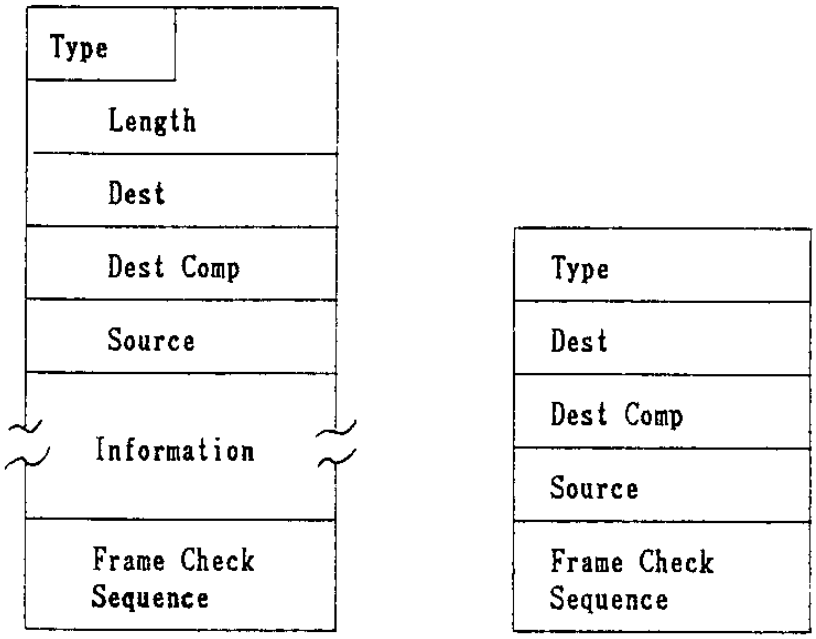
(1) 범용 컴퓨터와 전단 컴퓨터간의 통신

범용 컴퓨터와 전단 컴퓨터 간의 연결은 전용 컴퓨터마다 전단 컴퓨터를 직접 연결시켜 통신에 관한 업무를 전담하도록 한다. 즉 전용 컴퓨터로 데이타를 전송하고자하는 범용 컴퓨터는 전단 컴퓨터로 먼저 전송하게 된다. 고속 고신뢰 상호결합망에서는 두 개의 동축 케이블이 사용되기 때문에 두 개의 통신망 전용 제어기가 필요하며 이들은 통신 전용 CPU에 의해 각각 독립적으로 작동한다. 통신망 전용 제어기는 동축 케이블을 제어하는 직렬 인터페이스 장치와 직접 연결되며 충돌회피 CSMA 기능을 수행하는 논리 회로가 포함되어 있다. 그리고 통신 전용 CPU와 두 개의 통신망 전용 제어기는 범용 컴퓨터 내부의 시스템 버스에 모두 연결된다. 따라서 범용 컴퓨터의 CPU는 데이타를 프레임 형식으로 구성하여 통신 전용 CPU로 전송하게 된다. 즉 통신망 전용 제어기는 범용 컴퓨터의 CPU를 통하는 대신 통신 전용 CPU의 제어에 의해 프레임을 송수신하게 된다. 프레임의 형식적 구성은 그림 5.4와 같으며 전송될 데이타의 길이는 제한된 크기 내에서 가변적으로 유지되도록 한다.

(2) 전단 컴퓨터와 전용 컴퓨터 간의 통신

전단 컴퓨터와 전용 컴퓨터는 고속의 병렬 인터페이스 장치가 직접 연결되어 있다. 그리고 전용 컴퓨터와의 통신을 위한 입출력 장치가 별도로 구성되어 있기 때문에 이를 제어하는 전단 컴퓨터의 운영 체제에 따라 상호 간의 통신이 수행된다. 전단 컴퓨터는 고속 근거리 통신망을 통하여 전송받은 데이타와 프로그램을 일단 자신의 기억 장치에 저장한 후 연산 수행을 위하여 연결된 전용 컴퓨터로 전송한다. 이 때 전송 컴퓨터와의 인터페이스 역할을 하는 외부 레지스터는 전단 컴퓨터 내에 구성한다. 따라서 전단 컴퓨터는 해당 레지스터를 주소로 하여 계속적인 데이타 전송을 수행함으로써 전용 컴퓨터와 통신을 하게 된다.

그리고 전용 컴퓨터의 연산 수행이 완료되면 외부 레지스터와 그의 주소를 이용하여 다시 전단 컴퓨터의 기억장치로 연산 결과가 전송된다. 따라서 연산 결과를 전송받은 전단 컴퓨터는 데이타와 프로그램을 원래 송신했던 범용 컴퓨터의 주소를 목적지 주소로 하여 프레임을 구성한 후 고속 근거리 통신망을 통하여 연산 결과를 재전송하게 된다.



Type(4 비트) : 프레임의 종류 구분
Length(12 비트) : information frame 길이
Dest(8 비트) : 목적지 스테이션의 주소
Dest Comp(8 비트) : 목적지 주소 발생시 오류 비트 점검
Source(8 비트) : 근원지 스테이션의 주소
Information : 최대 1534 바이트 제공
Frame Check Sequence(32 비트) : 32 비트 cyclic redundancy check

그림 5.4 프레임의 형식적 구성

5.2.2 통신망 하드웨어 인터페이스

(1) 범용 컴퓨터의 통신 프로세서 서버(server)

고속 분산 처리 시스템내에서 구성되는 범용 컴퓨터는 통신망 업무이외에 부동 소숫점 연산, 그래픽 등 일반적 작업까지도 수행하게 된다. 결과적으로 범용 컴퓨터의 CPU는 많은 작업 부담이 생기게 되므로 범용 컴퓨터내에 통신망 작업을 전담할 수 있는 별도의 통신 프로세서 서버가 구현되어야 한다. 이하 통신 프로세서 서버내의 CPU는 통신 전용 CPU로 그리고 범용 컴퓨터내의 CPU는 편의상 호스트 CPU로 언급한다. 통신 전용 CPU는 사실상 호스트 CPU와는 별도로 동작할 수 있으며 호스트 CPU로부터의 요구를 메시지 형태로 넘겨받아 처리 한다. 즉 호스트 CPU는 사용자와 프로세서 서버와의 연결만을 담당한다. 범용 컴퓨터의 CPU가 처리해야 하는 통신 프로토콜의 부담을 통신 프로세서 서버가 담당하게됨에 따라 실제적으로 호스트 CPU는 통신 요구의 객체에 해당되는 메세지만을 서버에게 전달한다. 따라서 범용 컴퓨터의 CPU는 통신에 대한 부담을 덜고 시스템내의 다른 프로세스를 처리함으로써 전체 시스템의 성능 향상을 기대할 수 있게 된다.

a. 하드웨어의 구성과 동작

2개의 통신망이 사용됨에 따라 통신 프로세서 서버내에는 통신망 전용 제어기와 직렬 인터페이스 장치가 각각 2개씩 구성된다. 즉 범용 컴퓨터의 내부 구성도는 그림 5.5와 같다. 시스템 버스는 범용 컴퓨터의 카드 슬롯에 해당하며 로칼 버스는 통신 프로세서 서버내의 버스를 의미한다. 그리고 시스템 버스와 로칼 버스는 이중 포트 기억 장치를 통해 연결된다. 이와 같은 구조를 사용함으로써 통신 프로세서 서버와 호스트 시스템의 버스를 분리시킴과 동시에 버스 경쟁에 의한 성능 저하를 방지할 수 있다.

i. 이중 포트 기억 장치

호스트 CPU와 통신 프로세서 서버상의 통신 전용 CPU 간의 통신은 상호 인터럽트를 통한 신호(signalling)기법과 이중 포트 기억 장치를 통한 데이타 전달로 이루어진다. 이중 포트 기억 장치는 호스트가 통신 프로세서 서버의 국지 기억 장치로 전송하는 프로그램, 제어명령, 데이타 등을 전달하기 위한 메세지 버퍼로 동작한다. 즉 메세지의 도착이나 명령 수행의 종료 시에는 두 프로세서가 서로 인터럽트를 걸어 유효한 데이타가 있음을 알린다. 그리고 호스트 CPU와 통신 전용 CPU가

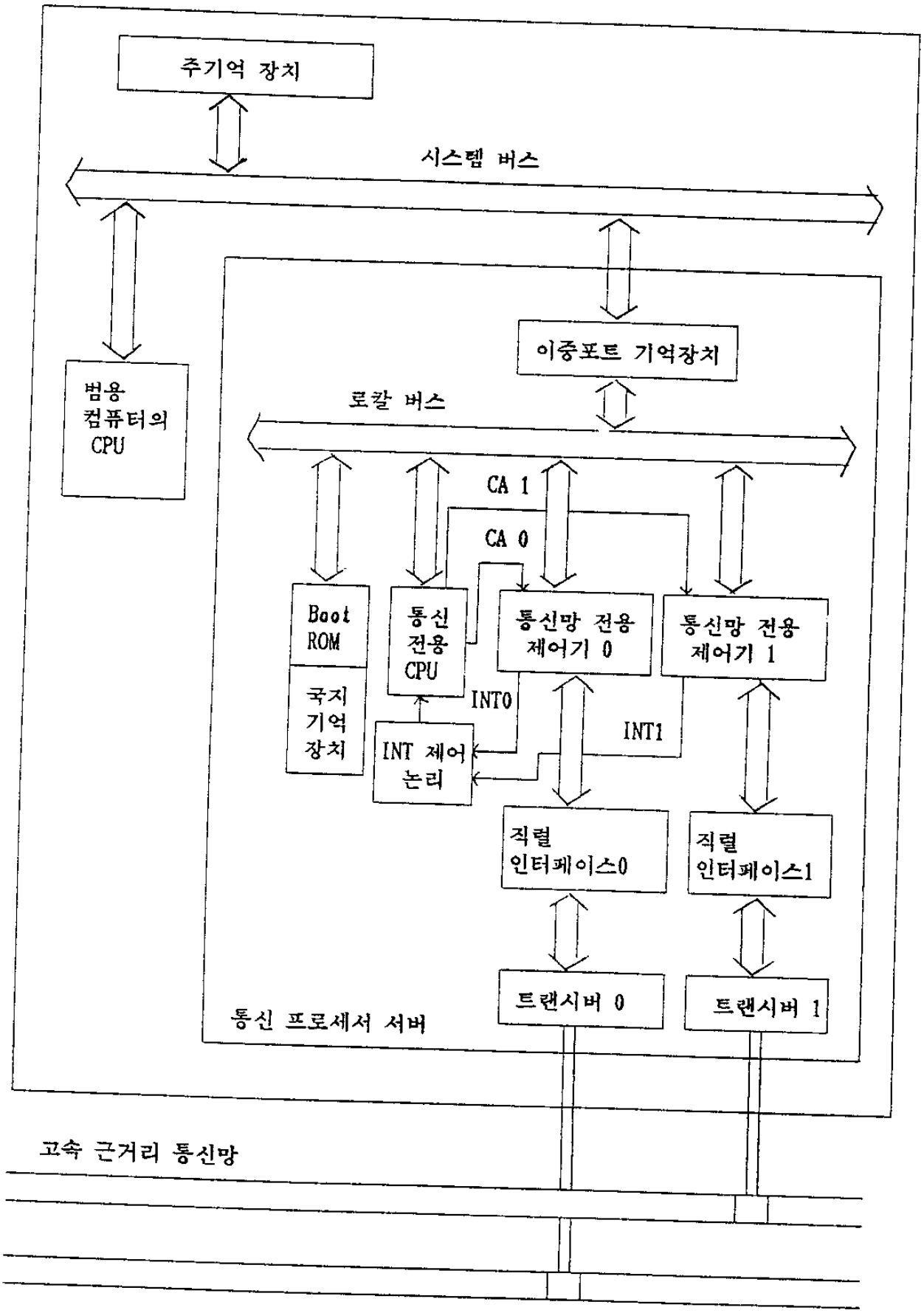


그림 5.5 통신 프로세서 서버가 설치된 범용 컴퓨터

동시에 이중 포트 기억 장치를 액세스하고자 하는 경우에는 우선 순위 제어 로직에 의해 어느 한 CPU의 액세스가 먼저 처리되고 다른 프로세서는 그 뒤에 액세스를 시작하게된다.

ii. 통신 전용 CPU와 통신망 전용 제어기

통신 프로세서 서버내에는 통신망 전용 제어기 2개와 이를 제어하는 통신 전용 CPU가 구성된다. 통신 전용 CPU는 실제로 구입 가능한 범용 CPU를 이용한다. 각 통신망 전용 제어기는 통신망의 데이타 링크 제어 전용 프로세서로 순차적인 명령이 아닌 특별한 데이타 구조에 의해 정의된 제어 명령들을 수행한다. 통신 전용 CPU에는 호스트와 두개의 통신망 전용 제어기로 부터 오는 세개의 외부 인터럽트 입력이 있다. 두 신호 중에는 호스트의 인터럽트가 우선 순위 로직에 의해 우선적으로 처리되고 통신망 전용 제어기로 부터의 인터럽트는 그 다음에 처리된다. 이 때 두개의 통신망 전용 제어기로 부터 동시에 인터럽트가 도착하는 경우에는 LRU (least recently used) 제어 로직에 의해 한 개의 인터럽트가 먼저 처리된다. 그리고 통신 전용 CPU의 여러가지 Exception 처리는 Boot ROM에 있는 벡터 테이블을 참조하여 수행된다. 벡터 테이블 중 ROM에서 처리하여야 하는 것들을 제외한 나머지 즉 다운 로드되는 프로그램에서 처리하여야 하는 것들은 벡터가 국지 기억 장치(RAM)상의 jump 테이블을 가리키고 그곳을 거쳐 실제 인터럽트 서비스 루틴으로 진입한다. 물론 RAM의 jump 테이블은 프로그램에 따라 다르므로 다운 로드시에 초기화된다.

iii. 통신망 전용 제어기와의 연결

고속 근거리 통신망의 데이타 링크 계층을 제어하는 각 통신망 전용 제어기는 직렬 인터페이스 장치 그리고 트랜시버와 연결된다. 통신망 전용 제어기는 프레임을 송수신하기 위한 데이타 구조를 독자적으로 운영하며 통신 전용 CPU와 상호 통신을 한다. 또한 통신 전용 CPU와 로칼 버스 경쟁을 통해 국지 기억 장치의 데이타 구조를 액세스한다. 통신망 전용 제어기와 통신 전용 CPU 간의 통신은 통신망 전용 제어기에 의해 하드웨어 적으로 정의되고 통신 전용 CPU가 초기화한 데이타 구조를 이용하여 이루어진다. 물론 이 데이타 구조에는 송수신 데이타 제어 영역, 상태 정보, 명령 구조 등이 포함된다. 통신 전용 CPU는 이 데이타 구조에

명령과 데이타를 적어 놓고 각 통신망 전용 제어기의 CA(channel attention) 단자에 신호를 보낸다. 이 CA에 의해 통신망 전용 제어기는 구동되어 작업을 수행하게 된다. 작업의 수행 후 혹은 한 프레임의 수신이 완료된 때 통신망 전용 제어기에는 해당 데이타 구조를 변경하고 통신 전용 CPU에 인터럽트를 발생시킴으로써 상호간에 통신을 한다. 그리고 프레임을 송신하는 경우에는 두 개의 통신망 전용 제어기에 모두 CA 신호를 전송한다. 각 통신망 전용 제어기는 통신망의 상태와 설정된 타이머 장치 그리고 WF 신호에 따라 통신망의 사용 여부가 탐지되면 통신전용 CPU로 다시 인터럽트 신호를 보낸다. 이 경우 통신망 전용 제어기들로 부터 두개의 인터럽트가 동시에 발생하게 된 경우에는 통신 전용 CPU의 우선 순위 로직에 의해 한 개의 인터럽트 만이 처리된다. 즉 송신 과정에서는 한 개의 통신망만을 사용하므로 나머지 한 개의 인터럽트는 처리하지 않는다. 그러나 두 개의 통신망 전용 제어기로 부터 일정 시간내에 통신망의 사용 가능성을 알리는 인터럽트가 발생하지 않게 되면 그 통신망은 사용이 불가능한 것으로 간주하고 그 시간 이후에는 정상적으로 작동하는 통신망만을 이용하게 된다. 즉 두 개의 통신망 중 한 개의 통신망만이 사용되는 것이다. 그리고 프레임을 수신하는 경우에는 먼저 수신받은 통신망 전용 제어기가 통신 전용 CPU로 인터럽트 신호를 보내게 됨에 따라 두 개의 인터럽트는 순차적으로 처리된다.

iv. 통신망 전용 제어기

고속 근거리 통신망을 위한 통신망 전용 제어기는 프로세서(통신 전용 CPU) 인터페이스 모듈, FIFO 모듈 그리고 직렬 인터페이스 장치와 연결되는 채널 인터페이스 모듈로 구성되며(그림 5.6 참조) 각 모듈에서 수행되는 기능을 다음과 같이 간략히 기술한다.

가) 프로세서 인터페이스 모듈

ㄱ) 명령어 장치

이중 포트 기억 장치로 부터 명령어를 액세스하고 실행시키기 위하여 통신 전용 CPU 인터페이스 모듈의 다른 장치들을 제어한다. 또한 통신 전용 CPU의 CA 신호에 대한 응답을 하며 초기화 과정을 관리한다.

ㄴ) 수신 장치

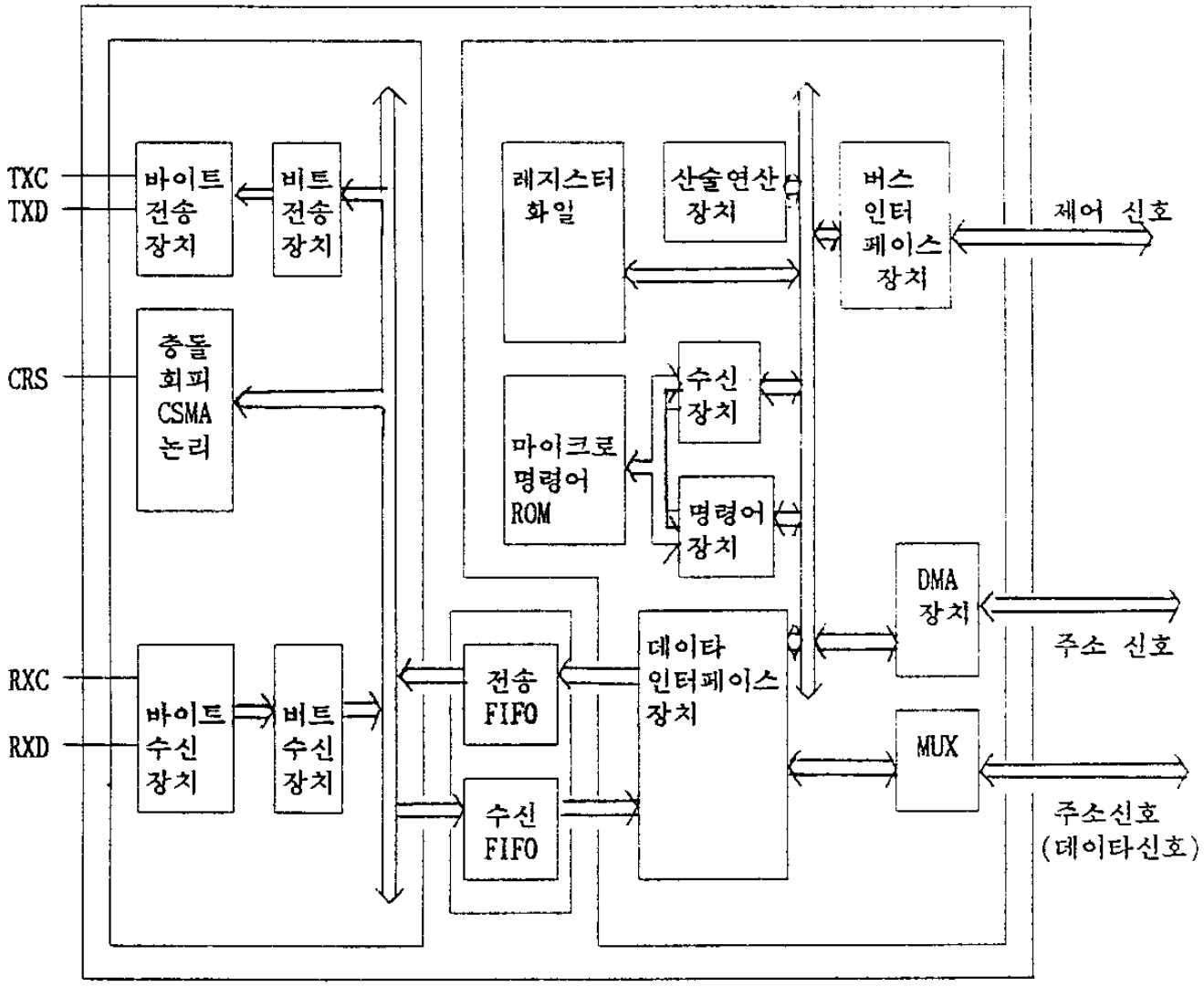


그림 5.6 통신망 전용 제어기의 블럭 구성도

버퍼의 이용 가능성, 크기, 위치에 관한 정보를 이중 포트 기억 장치로 부터 액세스하고 버퍼로의 데이타 전송을 관리한다.

ㄷ) 마이크로 명령어 ROM

명령어 장치와 수신 장치를 제어하는 마이크로프로그램이 내장된다.

ㄹ) 산술 연산 장치

마이크로 순차기에 의하여 간단한 산술 논리 연산을 수행한다.

ㅁ) 레지스터 화일

마이크로프로세서를 위한 내부용 데이타 기억 장치를 구성한다.

ㅂ) DMA 장치

통신 전용 제어기와 이중 포트 기억 장치 간의 정보 전송을 위하여 마이크로 순차기에서 제공되는 시작 주소와 바이트를 기점으로 기억 장치의 주소를 발생한다.

ㅅ) 버스 인터페이스 장치

CA, INT 신호이외에 통신 전용 CPU와 통신 전용 제어기에서 전송되는 모든 송수신 신호를 관리한다.

ㅇ) 데이타 인터페이스 장치

요구된 입력과 출력을 상호 연결함으로써 데이타와 상태 신호를 적당한 목적지로 배정하는 스위칭 장치이다. 또한 데이타의 packing 작업과 unpacking 작업도 수행한다.

나. 채널 인터페이스 모듈

채널 인터페이스 모듈은 네트워크 인터페이스에 대한 송신 부분과 수신 부분으로 구분된다.

ㄱ) 송신 부분

프레임용 전송 바이트장치, 비트 전송 장치, 충돌 회피 CSMA 논리 회로 들이 구성되어 있다. 즉 PAT, AAT, RT 타이머들이 내부에 구현되어 있다.

ㄴ) 수신 부분

프레임용 바이트 수신 장치, 비트 수신 장치 들이다.

다. FIFO 모듈

프로세서 인터페이스 모듈과 채널 인터페이스 모듈사이에 구성된 두 개의 FIFO 기억 장치이다.

ㄱ) 송신 FIFO

전송 방향으로 데이타를 송신한다.

ㄴ) 수신 FIFO

수신 방향으로 데이타를 수신한다.

v. 직렬 인터페이스 장치

고속 근거리 통신망용 직렬 인터페이스 장치에서 수행되는 기능은 다음과 같다.

- 송수신 프레임의 엔코딩/디코딩
- 트랜시버 케이블과의 전기적 접속
- loopback 기능 - 직렬 인터페이스 장치와 통신망 전용 제어기의 결함 탐지

이러한 기능들을 수행하는 직렬 인터페이스 장치의 블럭 구성도는 그림 5.7과 같다. 입출력 신호는 통신망 전용 제어기 부분과 트랜시버 케이블 인터페이스 부분으로 분리된다.

(2) 소프트웨어의 구성

a. Boot ROM과 다운 로드 프로그램

Boot ROM에는 통신 전용 CPU와 통신망 전용 제어기의 리세트 매개 변수, 인터럽트 벡터 테이블, 통신 프로세서 서버 하드웨어 진단 프로그램, 통신 전용 CPU exception 처리 루틴, 호스트 CPU로 부터 프로그램을 다운 로드받는 루틴이 포함된다. 통신 프로세서 서버는 호스트 시스템에 전원이 투입되는 순간에 리세트되어 POST(Power On Self Test)를 수행한다. POST 결과는 이중 포트 기억 장치에 기록되어 호스트가 통신 프로세서 서버의 상태를 파악할 수 있도록 한다. 통신 프로세서 서버는 POST 결과 이상이 없을 때에 다운 로드 대기 상태에 들어간 후 다운 로드를 수행하게 된다. 다운 로드가 끝나면 프로그램의 엔트리 포인트로 점프하여 그 프로그램을 실제 수행하게 된다. 따라서 호스트 시스템에서는 프로그램의 다운 로드와 이중 포트 기억 장치 제어를 위해 운영 체제내에 별도의 디바이스 드라이버 루틴이 구성되어야 한다.

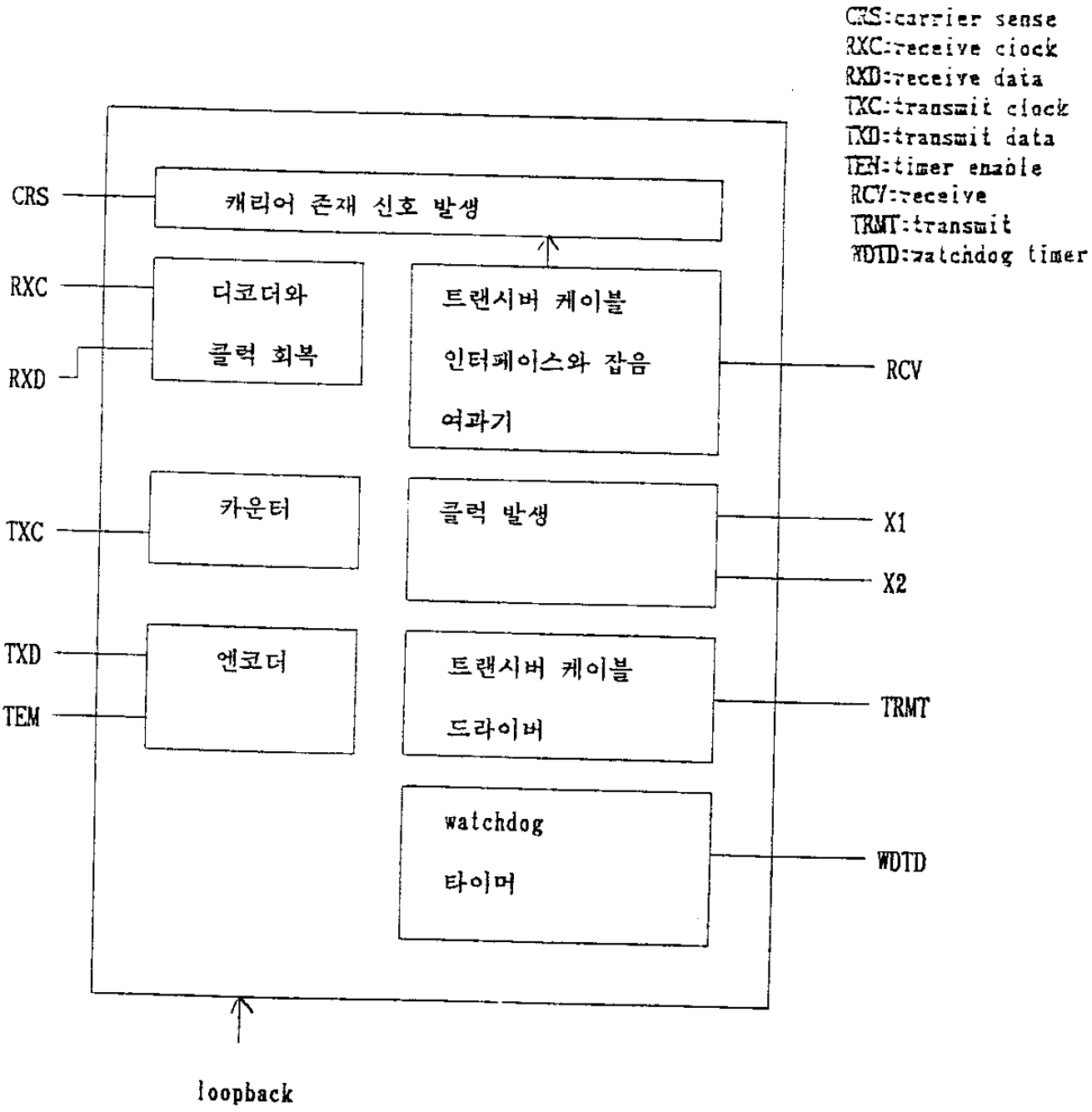


그림 5.7 직렬 인터페이스 장치의 블럭 구성도

b. 통신망 제어를 위한 프로그램

통신망 전용 제어기는 자체적으로 데이타 구조를 운영할 수 있는 능력을 가지고 있다. 통신 프로세서 서버의 국지 기억 장치 상에 위치하는 프로그램은 호스트 CPU와의 통신, 통신망 전용 제어기 자체의 초기화, 통신망 전용 제어기 데이타 구조에 따라 호스트로부터의 통신 요구를 구조화 하며, 통신망 전용 제어기 인터럽트 처리 등의 작업을 수행한다. 통신망 전용 제어기가 하드웨어적으로 정의하여 사용하는 데이타 구조의 예는 그림 5.8과 같다. 통신망 전용 제어기는 ROM 상의 SCP(System Configuration Pointer)를 참조하여 절대 번지에 있는 ISCP(Intermediate System Configurtion Pointer)를 읽어 SCB(System Configurtion Block)의 주소를 결정한다[INTE 84].

SCB에는 데이타 링크의 수행 통계 데이타와 통신망 전용 제어기가 수행할 명령 리스트 포인터, 수신 데이타 저장을 위한 데이타 구조의 포인터가 들어 있다. 호스트로부터 데이타 송신 요구나 상태 정보 보고 요청이 있으면 통신 전용 CPU는 그 내용을 CB(Command Block)으로 만들어 CBL(Command Block List)의 끝에 매달아 통신망 전용 처리기 중 한 개가 처리 하도록 한다. 통신망 전용 제어기는 각 명령이 처리될 때마다 통신 전용 CPU에 인터럽트를 걸어 결과를 알리고 통신 전용 CPU는 해당 CB를 CBL에서 떼어내 통신 전용 CPU의 Free CB List에 넣는다. 통신 전용 CPU가 프레임 송신 CB를 만들 때는 동시에 TBD(Transmit Buffer Descriptir)와 송신 프레임 버퍼가 프레임 크기에 맞추어 하나 또는 그 이상 만들어져 그 포인터가 CB에 기록되는 것이다. 통신망 전용 제어기가 프레임을 수신하면 통신 전용 CPU가 통신망 전용 제어기의 수신 유니트를 이네이블시키기 전에 먼저 초기화 해놓은 RFA(Receive Frame Area)에 있는 FD(Frame Descriptor)와 그에 딸린 RBD(Receive Buffer Descriptor), 수신 버퍼에 데이타를 저장하고 통신 전용 CPU에 인터럽트를 걸어 프레임 수신을 알린다. 통신 전용 CPU는 FD와 RBD, 버퍼로부터 프레임을 읽어 이중 포트 기억 장치를 통해 호스트 CPU에 전송하거나 별도의 버퍼에 저장한 뒤 FD와 RBD, 수신 프레임 버퍼를 다시 RFA에 돌려 준다. 통신망을 제어하기 위해서 범용 컴퓨터 시스템에서는 운영 체제 내에 별도의 루틴을 구성하여 통신망을 이용할 수 있도록 한다.

5.2.2 전단 컴퓨터

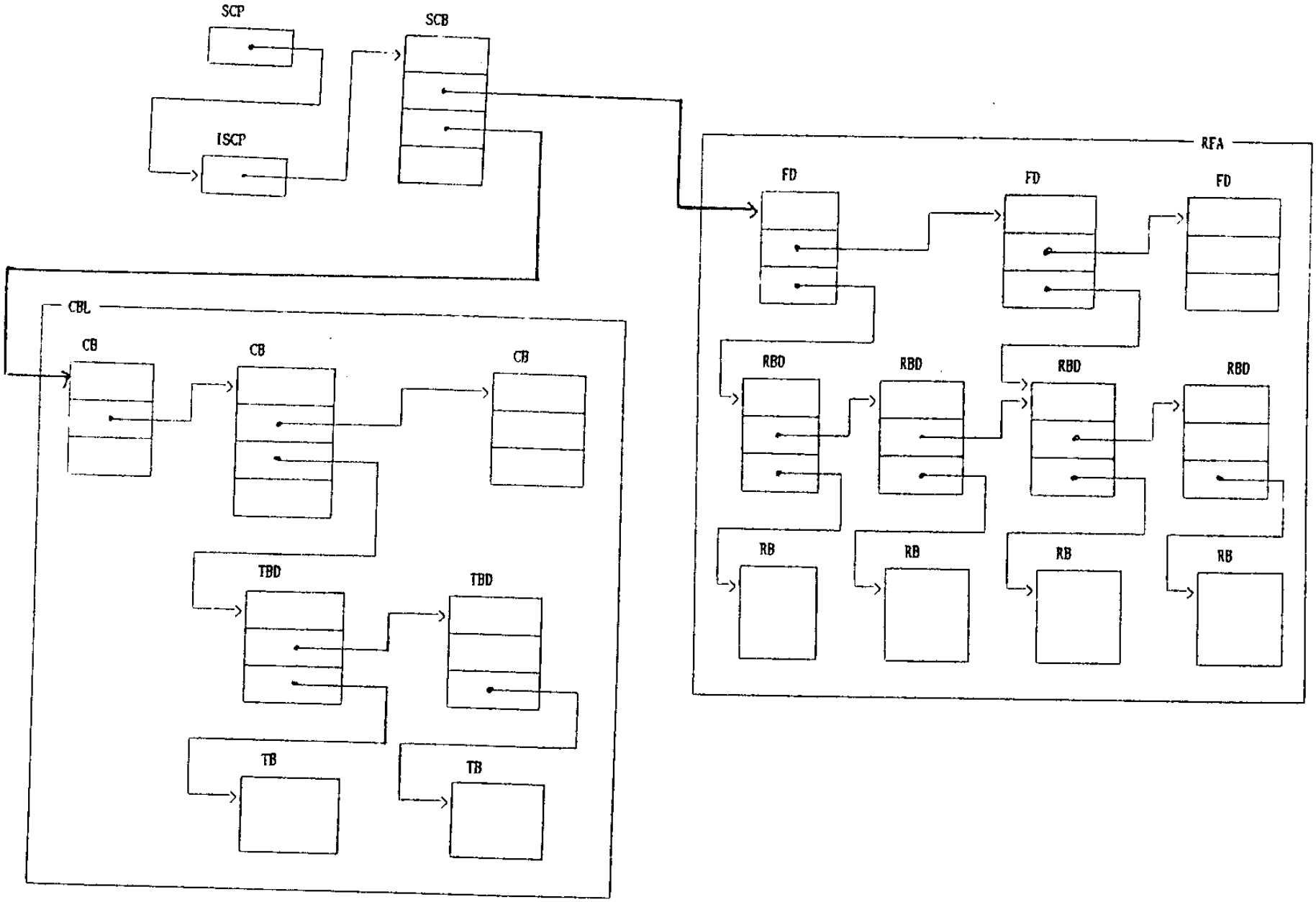


그림 5.8 통신망 전용 제어기의 데이타 구조

(1) 전단 컴퓨터의 통신 기능

math package 전용 컴퓨터와 DAC 전용 컴퓨터는 통신망을 직접 제어할 수 있는 기능이 구비되어 있지 않다. 따라서 전단 컴퓨터의 통신을 전담하기 위한 전단 컴퓨터(FEC; Front End Computer)가 고속 고신뢰 상호 결합망내에 구성된다. 전용 컴퓨터로 데이타를 전송하고자하는 범용 컴퓨터는 전용 컴퓨터에 연결된 전단 컴퓨터로 전송하게 된다. 전단 컴퓨터는 도착된 모든 데이타를 자신의 기억 장치에 일단 저장시킨 다음 전용 컴퓨터와의 통신 장치를 통하여 전용 컴퓨터내의 기억 장치로 데이타를 다시 전송한다. 이 때 전단 컴퓨터는 전용 컴퓨터와의 통신을 위한 외부 레지스터(CPU내의 레지스터와 구분하기 위한 표현)와 그 주소에 대한 디코딩 로직을 구성한다. 따라서 전단 컴퓨터는 전용 컴퓨터에서 실행 가능한 패킷 형식의 데이타를 특정 주소와 그 주소에 대한 디코딩 로직을 이용하여 외부 레지스터로 계속 전송하게 된다. 즉 전단 컴퓨터의 기억 장소와 전용 컴퓨터의 기억 장소 간에 외부 레지스터를 이용한 간접적인 데이타 전송이 수행되는 것이다. 전용 컴퓨터는 프로그램과 데이타를 처리한 후 전단 컴퓨터로 실행 종료를 알리는 인터럽트 신호를 발생시킨다. 그러면 전단 컴퓨터의 CPU는 실행이 종료되었음을 알고 외부 레지스터를 이용하여 다시 자신의 기억 장치로 저장하게 된다. 연후에 전단 컴퓨터는 결과를 프레임으로 구성하여 통신망을 액세스함으로써 원래의 범용 컴퓨터로 전송하게 된다. 범용 컴퓨터는 통신망 작업이외에 일반적인 프로세스를 수행하여야 함과는 달리 전단 컴퓨터에서는 전용 컴퓨터의 통신망 사용을 위한 기능만이 주로 강조된다. 따라서 범용 컴퓨터의 경우처럼 통신 전용 CPU를 별도로 구성할 필요가 없으며 통신 프로세서 서버내에는 두 개의 통신망 전용 제어기가 전단 컴퓨터의 시스템 버스와 직접 연결된다(그림 5.9 참조).

(2) 통신 프로세서 서버

범용 컴퓨터의 경우와 마찬가지로 두 개의 동축 케이블이 사용되기 때문에 두 개의 통신망 전용 제어기가 필요하며 이들은 각각 독립적으로 작동한다. 통신망 전용 제어기는 동축 케이블을 제어하는 직렬 인터페이스 장치와 직접 연결되며 충돌 회피 CSMA 기능을 수행하는 회로가 포함되어 있다. 그리고 호스트 CPU와 통신망 전용 제어기는 범용 컴퓨터 내부의 시스템 버스에 사실상 직접 연결된다. 따

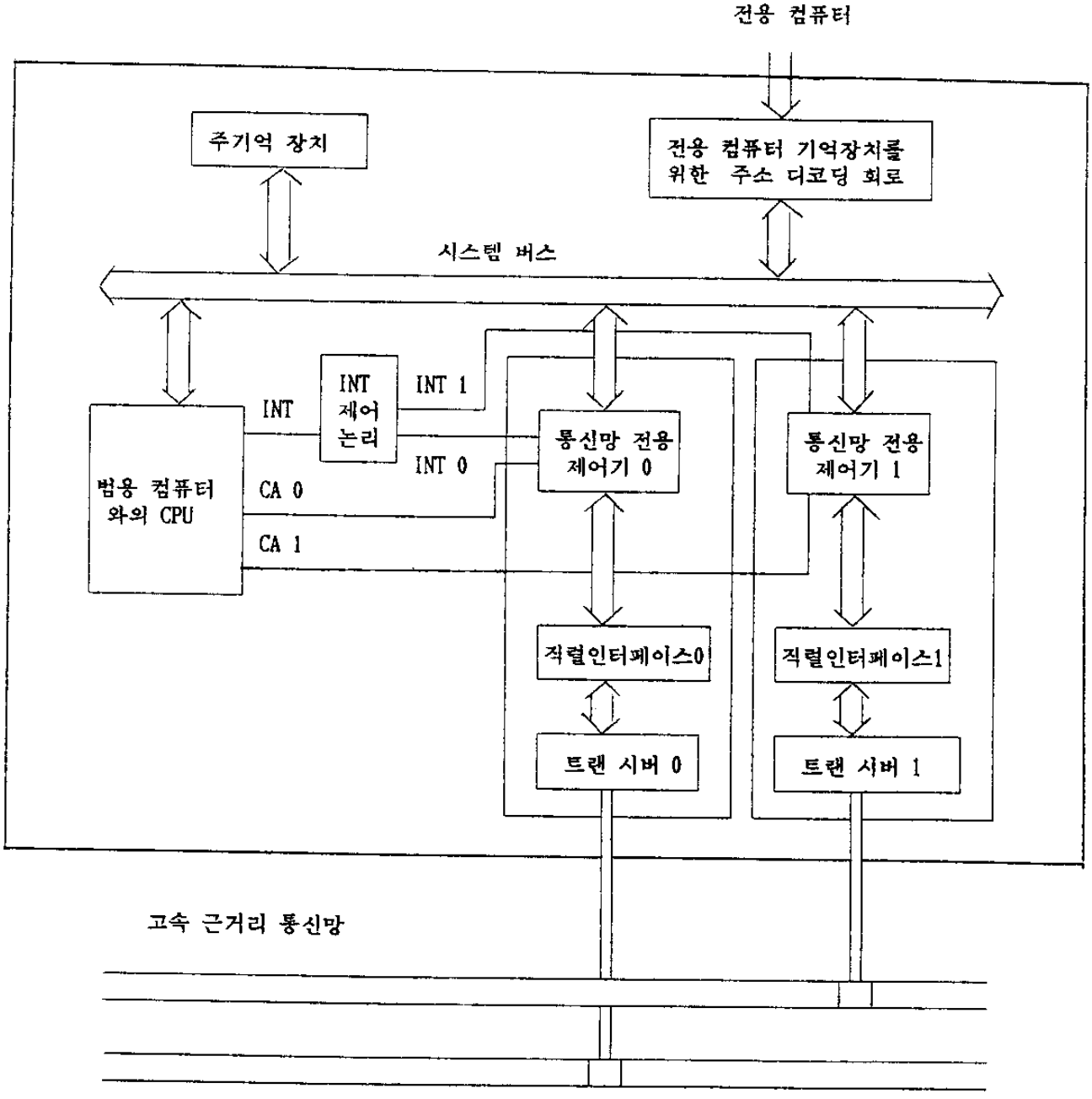


그림 5.9 전단 컴퓨터 내부적 구성

5.3 통신 소프트웨어

5.3.1 OSI 계층 구조

OSI(Open System Interconnection)의 표준 7 계층 구조와 비교한 고속 분산 시스템의 네트워크의 계층적 기능 구조는 그림 5.10과 같다. 동축 케이블은 직렬 인터페이스 장치에 의하여 제어가 되며 데이타 링크 계층의 기능은 통신망 전용 제어기에 의하여 수행된다. 그리고 통신망 전용 제어기를 제어하며 네트워크 계층과 트랜스포트 계층의 기능을 수행하는 통신 프로토콜은 OSI에서 제정한 프로토콜을 사용한다. 컴퓨터 시스템 사이의 상호 연결을 위하여 OSI 참조 모델을 따르는 프로토콜 구조는 국제적인 추세이고 향후에는 이 구조가 전반적으로 상용화될 가능성이 높고, 현재도 이에 대한 활발한 연구가 진행 중이다[TOLH 88]. 또한 이미 부분별로 실현되어 사용되고 있기때문에 본 고속 분산 처리 시스템의 통신 기능은 이에 맞추어 OSI 구조를 기본으로 한다. 따라서 각 계층의 수행되는 기능을 간략히 정의하고 구현될 통신 프로토콜의 프리미티프를 기술하며 구체적인 통신 사항은 생략한다. 그리고 OSI 표준 7 계층 구조에 따르는 통신 프로토콜은 범용 컴퓨터와 전단 컴퓨터에 동일하게 적용된다.

5.3.2 데이타 링크 계층

근거리 통신망에서는 통신 매체를 사용하기 위한 2 가지 방법(CSMA/CD, Token Ring)이 제공되나 고속 분산 처리 시스템의 상호 결합망에서는 고속 근거리 통신망 (HSLN)을 사용하므로 이미 언급된 충돌 회피 CSMA 방식을 매체 제어 방법(medium access control)으로 채택한다. 즉 그림 5.10의 데이타 링크 계층내의 물리적 데이타 링크 부분이 이에 해당한다. 그리고 상위 계층인 네트워크 계층과의 접속을 위한 공통 부분인 LLC(Logical Link Control)를 개발한다(그림 5.11 참조). LLC에는 표준안으로 3 가지 형태의 기능을 갖는 프로토콜이 존재하는데 본 고속 분산 처리 시스템의 전체적인 프로토콜 구조와 상위 계층의 프로토콜과 연관하여 살펴보면 구현될 LLC는 type 1 형태의 unacknowledgement connectionless service 방법을 사용한다. 이 서비스는 단순히 프레임들을 주고 받게 하며 점대점 (point-to-point), 멀티포인트, 브로드캐스트 방식들을 보조한다.

5.3.3 네트워크 계층

인터럽트가 일정한 시간동안 발생하지 않는 경우에는 통신망에 이상이 생긴것으로 간주한다. 따라서 그 시간 이후에는 정상적으로 작동하는 통신망을 이용하여 프레임을 송수신하게 된다.

b. 프레임의 수신

독립적으로 동작하는 두 개의 통신망 전용 제어기는 직렬 인터페이스 장치를 통해 수신된 프레임의 주소를 확인한다. 목적지 주소가 자기의 주소와 일치하는 경우에는 CPU에 INT(interrupt) 신호를 전송한다. 이 때 두 개의 INT(INT0, INT1)신호가 동시에 발생하면 통신망 인터럽트 제어기(전단 컴퓨터내의 인터럽트 처리장치 이용)에 의하여 하나의 INT 신호만이 CPU로 전송된다. 인터럽트 제어기에서는 LRU(least recently used) 기법으로 INT 신호의 충돌 문제를 해결한다. 전송된 INT 신호를 확인한 CPU는 수신(receive) 명령어와 CA(channel attention) 신호를 해당 통신망 전용 제어기로 송신한다. 그러면 그 통신망 전용 제어기는 시스템 버스를 획득하여 수신한 프레임을 주 기억 장치에 저장한다. 그리고 나서 다른 통신망 전용 제어기는 수신 프레임을 역시 주 기억 장치에 차례로 저장한다.

(4) 통신망 제어를 위한 프로그램

범용 컴퓨터와는 달리 전단 컴퓨터의 통신 프로세서 서버에는 별도의 국지 기억 장치와 이중 포트 기억 장치가 구성되어 있지 않다. 따라서 통신망 전용 제어기는 전단 컴퓨터의 주 기억 장치 중 ROM 상의 SCP(System Configuration Pointer)를 참조하여 절대 번지에 있는 ISCP(Intermediate System Configuration Pointer)를 직접 읽어 SCB(System Configuration Block)의 주소를 결정한다. 그리고 통신망 전용 제어기가 하드웨어적으로 정의하여 사용하는 데이타 구조(그림 5.8 참조)를 비롯하여 프레임의 송수신시 사용되는 실제 프로그램은 범용 컴퓨터의 경우와 동일하므로 중복되는 설명은 생략하기로 한다.

라서 통신망 전용 제어기는 시스템 버스의 사용권을 획득한 후 데이타를 프레임 형식으로 구성하여 전송하게 된다. 통신망 전용 제어기는 프레임의 송수신시 CPU를 통하지 않고 전단 컴퓨터의 주 기억 장치를 직접 이용할 수 있다. 프레임의 형식적 구성은 그림 5.4와 같으며 전송될 데이타의 길이는 역시 가변적으로 유지될 수 있도록 한다. 그리고 전용 컴퓨터로 전송되는 데이타의 형식은 통신망에서는 이용되는 프레임과는 달리 전용 컴퓨터에서 실행 가능한 여러 가지 형태의 패킷으로 구성된다. 패킷 형식의 데이타는 CPU에 의해 기억 장치로 부터 읽혀진 다음 외부 레지스터를 통하여 전용 컴퓨터의 기억 장치로 전송된다. 즉 전단 컴퓨터 내에는 전용 컴퓨터와의 통신을 위한 외부 레지스터와 그에 대한 주소 디코딩 로직이 하드웨어로 구성된다.

(2) 프레임의 송수신 과정

전단 컴퓨터의 프레임 송수신 과정은 범용 컴퓨터의 경우와 거의 동일하다. 단 전단 컴퓨터에는 통신 전용 CPU가 별도로 구성되지 않으므로 호스트 CPU가 통신 기능을 수행하게 된다. 그리고 통신 프로세서 서버내에 이중 포트 기억 장치를 별도로 구성하는 대신 주 기억 장치를 직접 이용하여 송수신되도록 한다. 통신망 전용 제어기가 하드웨어적으로 정의하여 사용하는 데이타 구조도 그림 5.8과 같다.

a. 프레임의 송신

CPU는 전송(transmit) 명령어를 두 개의 통신망 전용 제어기에 모두 전송한다. 전송 명령 신호를 수신한 통신망 전용 제어기는 충돌 회피 CSMA 방식으로 동축 케이블의 사용 가능성 여부를 탐지하게 된다. 사용 가능성이 확인된 경우에는 INT 신호를 CPU로 전송한다. 수신한 INT 신호에 따라 CPU는 CA 신호를 해당 통신망 전용 제어기로 송신하게 된다. 연후에 그 통신망 전용 제어기는 송신할 프레임을 주 기억 장치로 부터 가져와 목적지로 전송한다. 두 개의 통신망 제어기로 부터 인터럽트가 동시에 발생한 경우에는 LRU 기법에 의해 처리된다. 즉 CPU는 두 개의 통신망 중 사용 가능한 통신망을 탐지하여 프레임을 전송시키게 되는 것이다. 이 때 동일한 프레임을 두 번 전송할 필요는 없으므로 나머지 한개의 인터럽트는 확인만 한 후 실제적인 처리는 하지 않는다. 그리고 통신망 전용 제어기로 부터의

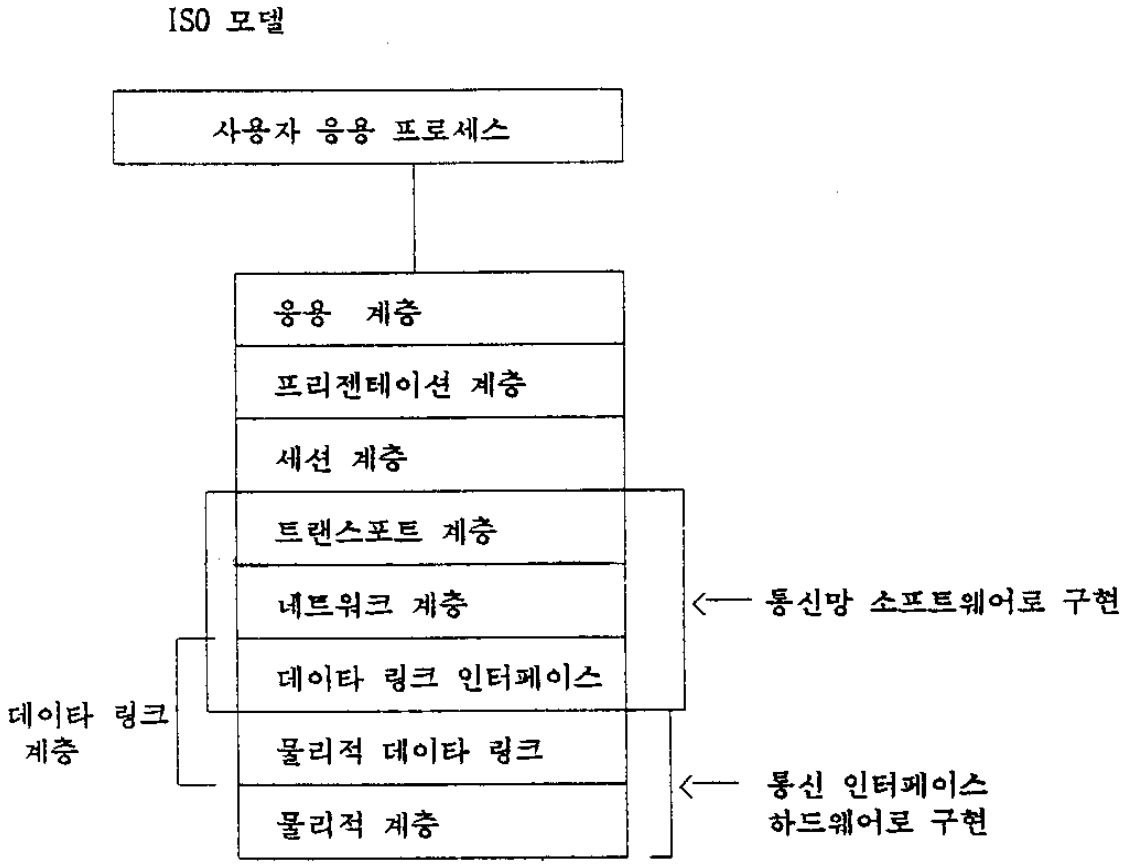


그림 5.10 통신 인터페이스 구현

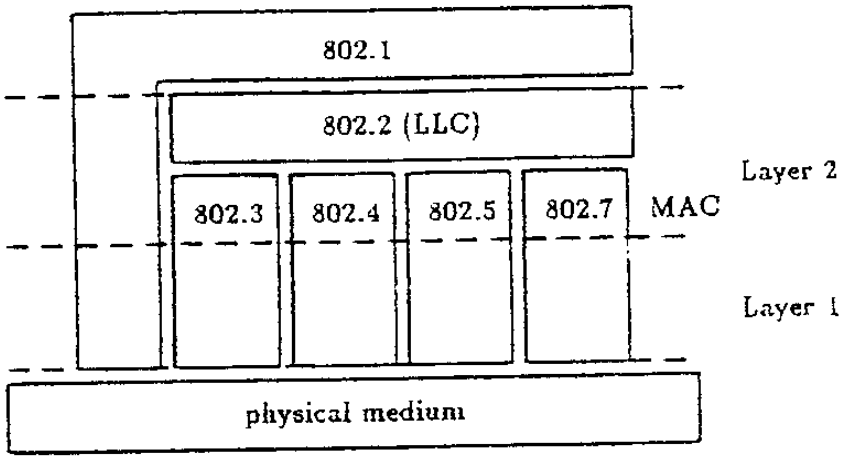


그림 5.11 데이타 링크 계층의 표준안

네트워크 계층(Network Layer)의 프로토콜은 장차 게이트웨이 역할을 하는 범용 컴퓨터를 통한 인터 네트워킹과 근거리 통신의 성능을 고려하여 무연결(connectionless) 방식을 사용한다. ISO CLNP(Connectionless Network Protocol)은 상위 계층에서 보내진 Transport Protocol Data Unit(TPDU)를 Network Protocol Data Unit(NPDU)로 구성하여 하위계층에 전달한다. 데이타가 목적지에 정확하게 도착하기 위하여 NPDU내에 모든 정보를 포함하고 있으며 만일 동일 네트워크가 아닐 경우에는 게이트웨이 역할을 하는 범용 컴퓨터를 통하여 목적지에 데이타가 전달 된다. 그리고 CLNP의 주요 기능은 다음과 같다. 표 5.1에서는 네트워크 계층의 프리미티브를 요약하고 있다.

- Connectionless Network Protocol(CLNP, ISO DIS 8473)
  . PDU(Protocol Data Unit) Composition and Decomposition
  . Header Format Analysis
  . PDU lifetime Control
  . Route PDU
  . Segmentation and Reassembly
  . Discard PDU
  . Error Reporting
  . PDU Header Error Reporting
  . Padding
  . Record Route
  . Quality of Service Maintenance

5.3.4 트랜스포트 계층

ISO 트랜스포트 계층(transport layer)중에서 컴퓨터 시스템에 구현될 프로토콜은 Class 4로서 연결관리(Connection Management), 데이타 전송(Data Transfer)에 관련하여 하위 계층들에서 제공하는 최소한의 서비스에 대하여 충분히 보상할 수 있는 기능을 수행하여 정확하게 데이타를 목적지에 전달하여 준다. 즉, 데이타의 번호 부여와 acknowledgement를 통하여 데이타의 상실, 중복되는 것을 방지한다. 또한 Check-sum 기능을 가지고 있어 손상된 데이타를 검출하고 Window나 Credit를

표 5.1 네트워크 계층의 프리미티브

PRIMITIVE	PARAMETERS	TYPE
N-CONNECT	CALLED ADDRESS CALLING ADDRESS RESPONDING ADDRESS RECEIPT CONFIRMATION SELECTION EXPEDITED DATA SELECTION QOS PARAMETER SET USER DATA	C
N-DATA	CONFIRMATION REQUEST USER DATA	U
N-DATA-ACKNOWLEDGE		U
N-EXPEDITED-DATA	USER DATA	U
N-RESET	ORIGINATOR REASON	C
N-DISCONNECT	ORIGINATOR RESPONDING ADDRESS REASON USER DATA	U/P

TYPE U : unconfirmed
C : confirmed
P : provider - initiated

사용함으로써 통신하는 사용자 사이의 흐름을 제어한다. 그리고, 수행되는 주요 기능들은 다음과 같으며, 이에 대한 프리미티브들은 표 5.2에 기술되어 있다.

- Connection-oreiented Transport Protocol, Class 4 (TP4, ISO 8073)
  - TPDU(Transport Protocol Data Unit) transfer
  - Segmenting and Reassemblying
  - Concatenation and Seperating
  - Connection Establishment and Connection Refusal
  - Normal Release
  - Association of TPDUs with Transport Connection
  - TPDU numbering
  - Expedited Data Transfer
  - Retension until acknowledgement of TPDUs
  - Explicit Flow Control
  - Checksum
  - Frozen References
  - Retransmission on Time-out
  - Resequencing
  - Inactivity Control
  - Treatment of Protocol Errors

5.3.5 세션 계층

세션 계층(session layer)의 프로토콜은 Session service 사용자 사이의 논리적인 연결(connection)을 설정하고, 대화(dialogue)내에 동기화 지점(synchronization point)을 설정하여 오류 발생시에 합의된 동기화 지점(synchronization point)으로 부터 대화를 재개하고, Activity를 관리하며, 설정된 연결을 순차적인 방법으로 해제한다. Session connection 설정시, 각 기능 단위 들을 협상하고 토큰의 위치를 결정한다. 그리고, 세션 계층내의 프리미티브들은 표 5.3과 같으며 수행될 주요 기능들은 다음과 같다.

표 5.2 트랜스포트 계층의 프리미티브

PRIMITIVE	PARAMETERS	TYPE	TPDUs
T-CONNECT	CALLED ADDRESS	C	CR
	CALLING ADDRESS		CC
	EXPEDITED DATA OPTION		
	QUALITY OF SERVICE		
	USER DATA (32 octets)	U	DT
	RESPONDING ADDRESS		
T-DATA	USER DATA (unlimited)	U	DT
			AK
			RJ
T-EXPEDITED-DATA	USER DATA (16 octets)	U	ED
			EA
T-DISCONNECT	DISCONNECT REASON	U/P	DR
	USER DATA (64 octets)		DC

표 5.3 세션 계층의 프리미티브

PRIMITIVE	PARAMETERS	TYPE	SPDUs
S-CONNTECT	SESSION CONNTECT ID CALLING SSAP CALLED SSAP RESULT QUALITY OF SERVICE SESSION REQS INITIAL SYNC NUMBER INITIAL TOKENS USER DATA (512 octets)	C	CN AC RF
S-DATA	USER DATA (unlimited)	U	DT
S-EXPEDITED-DATA	USER DATA (14 octets)	U	EX
S-TYPED-DATA	USER DATA (unlimited)	U	TD
S-CAPABILITY-DATA	USER DATA (512 octets)	C	CD CDA
S-TOKEN-PLEASE	TOKENS USER DATA (512 octets)	U U	PT GT
S-TOKEN-GIVE	TOKENS	U	GTC
S-CONTROL-GIVE	NONE		GTA
S-SYNC-MINOR	TYPE SYNC NUMBER USER DATA (512 octets)	C	MIP MSA
S-SYNC-MAJOR	SYNC NUMBER USER DATA (512 octets)	C	MAP MAA
S-RESYNCHRONIZE	RESYNC TYPE SYNC NUMBER RESYNC TOKENS USER DATA (512 octetes)	C	RS RA
S-P-EXCEPTION-REPORT	REASON	P	ER
S-U-EXCEPTION-REPORT	REASON USER DATA (512 octets)	U	ED
S-ACTIVITY-START	ACTIVITY ID USER DATA (512 octets)	U	AS
S-ACTIVITY-INTERRUPT	REASON	C	AI AIA
S-ACTIVITY-RESUME	ACTIVITY ID OLD ACTIVITY ID SYNC NUMBER OLD SESSION CONNECT ID USER DATA (512 octets)	U	AR
S-ACTIVITY-DISCARD	REASON	C	AD ADA
S-ACTIVITY-END	SYNC NUMBER USER DATA (512 octets)	C	AE AEA
S-RELEASE	RESULT USER DATA (512 octets)	C	DN FN
S-U-ABORT	USER DATA (9 octets)	U	AB AA
S-P-ABORT	REASON	P	AB

- Connection-oriented Session Protocol (ISO DIS 8327)
  . Connection Establishment
  . Normal Release of Connection
  . Abnormal Release of Connection
  . Data Transfer (Normal, Expedited, Typed, Capability)
  . Token Handling (Givem Please, Control Give)
  . Synchronization (Major, Minor) and Resynchronization
  . Exception reporting
  . Activity Management(Resume, Interrupt, Discard, End)
  . Use of Transport Service

5.3.6 프레젠테이션 계층

프레젠테이션 계층(presentation layer)에 대한 개발은 기능상 세션 계층의 기능과 큰 차이가 없고 동일 시스템 사이의 표현 방법의 차이가 없기 때문에 프레젠테이션 프로토콜의 기능을 구현하지는 않고 단지 상위 계층의 서비스 요청에 따라 세션 계층의 서비스를 연결하여 준다. 표 5.4에는 프레젠테이션 계층의 프리미티브가 정의되어 있으며 이 계층에서 수행되는 기능들은 다음과 같다.

- Basic Encoding Rules for Abstract Syntax Notaion One(ASN.1 ISO DIS 8825)
  . Encoding of a Boolean Value
  . Encoding of a Bitstring Value
  . Encoding of a Octetstring Value
  . Encoding of a Null Value
  . Encoding of a Sequence(-of) Value
  . Encoding of a Set(-of) Value
  . Encoding of a Choice Value
  . Encoding of a Selection Value
  . Encoding of a Tagged Value
  . Encoding for Values of Character Set String Types

표 5.4 프레젠테이션 계층의 프리미티브

PRIMITIVE	PARAMATERS	TYPE	PPDUs
P-CONNECT	CALLING PSAP	C	CP
	CALLED PSAP		CPA
	RESPONDING PSAP		CPR
	MULTIPLE CONTEXTS		
	CONTEXT DEF LIST		
	DEFAULT CONTEXT		
	QUALITY OF SERVICE		
	PRESENTATION REQS		
	SESSION REQS		
	INITIAL SYNC NUMBER		
	INITIAL TOKENS		
	SESSION CONNECT ID		
	RESULT		
	USER DATA		
P-DATA	USER DATA	U	TD
P-EXPEDITED-DATA	USER DATA	U	TE
P-TYPED-DATA	USER DATA	U	TTD
P-CAPABILITY-DATA	USER DATA	C	TC
			TCC
P-ALTER-CONTEXT	CONTEXT DEF LIST	C	AC
	CONTEXT DEF LIST		ACA
	USER DATA		
P-TOKEN-PLEASE	TOKEN ITEM	U	
P-TOKEN-GIVE	TOKEN ITEM	U	
P-CONTROL-GIVE	NONE	U	
P-SYNC-MINOR	TYPE	C	
	SYNC NUMBER		
	USER DATA		
P-SYNC-MAJOR	SYNC NUMBER	C	
	USER DATA		
P-RESYNCHRONIZE	TYPE	C	RS
	SYNC NUMBER		RSA
	TOKEN ITEM		
	CONTEXT ID LIST		
	USER DATA		
P-P-EXCEPTION-REPORT	REASON	P	
P-U-EXCEPTION-REPORT	REASON	U	
	USER DATA		
P-ACTIVITY-START	ACTIVITY ID	U	
	USER DATA		
P-ACTIVITY-INTERRUPT	REASON	C	
P-ACTIVITY-RESUME	ACTIVITY ID	U	
	OLD ACTIVITY ID		
	SYNC NUMBER		
	OLD SESSION CONNECT ID		
	USER DATA		
P-ACTIVITY-DISCARD	REASON	C	
P-ACTIVITY-END	SYNC NUMBER	C	
	USER DATA		
P-RELEASE	USER DATA	U	
	ABORT DATA		

5.3.7 응용 계층

응용 계층(Application layer)의 FTAM(File Transfer Access and Management) 프로토콜은 가상 화일 스토어 개념을 사용해서 상이한 화일 시스템을 가지는 시스템들 사이에서도 화일 전송등의 화일 서비스를 제공할 수 있다. 다음에 열거되는 기능들은 기본적으로 ISO 권고안의 필수 기능 부분만 포함하는 것을 원칙으로 하였지만, 본 고속 분산 시스템의 운영체계(O.S.)가 가지는 제약으로 인해 구현이 어려운 부분은 제외될 수도 있다. FTAM 프로토콜은 화일 서비스를 지원하기 위해서 ACSE(Association Control Service Element)를 통한 결합 설정 및 기본적인 화일접근과 관리 서비스를 제공하기 위해 하위 계층들의 서비스를 이용한다. 그리고, Message Handling System(MHS)과 Virtual Terminal Protocol(VTP)에 대한 규격은 생략한다.

응용 계층에서 수행되는 주요 기능은 다음과 같다.

-Association Control Service Element(ACSE, ISO DP 865012)
- . Association Establishment
- . Normal Release of an Association
- . Abnormal Release of an Association

-File Transfer, Access and Management(FTAM, ISO DIS 8571)
- . FTAM Regime Establishment(Orderly)
- . FTAM Regime Termination(Abrupt)
- . File Regime Control
- . File Management
- . Grouped Operations
- . File Access
- . Read/Write
- . Sending Data
- . Canceling Transfer
- . Terminating Transfer

5.4 성능 평가

5.4.1 전파 지연 시간과 전송률

long-haul 네트워크와 근거리 네트워크의 차이점은 전송률(R)과 네트워크의 거리(d)에 의해 구분된다. 그리고 R x d의 결과가 바로 근거리 네트워크의 특성을 결정하게 된다. 단적인 예를 들면 거리가 1 km이고 전송률이 100 Mbps인 네트워크는 10 km이며 10 Mbps인 네트워크와 성능면에서 동일하다고 볼 수 있다. R x d는 통신 매체의 전파 속도 V로 나뉘어 지며 이 Rd/V는 임의의 두 노드 사이에 전송될 수 있는 비트 수, 즉 매체상에서 전송되는 비트의 길이를 의미하게 된다. 근거리 통신망과 고속 근거리 통신망의 성능 평가시 동일한 매개 변수가 적용되므로 본 고속 고신뢰 상호 결합망에서도 다음과 같은 a 값을 사용한다[WILL 87].

a = length of data path(in bits) / length of frame (1)

즉, a는 전송 매체를 비트 길이로 나타낸 값을 전형적인 프레임의 크기로 나눈 값이다. 따라서 다음 식이 성립된다.

$$a = Rd/VL \quad (2)$$

L : 프레임의 길이
d/V : 전송 매체상의 전파 지연 시간
L/R : 프레임의 전송 시간

결국 a 는 다음과 같이 표현된다.

a = 전파 지연 시간(propagation time) / 전송 시간(transmission time) (3)

고속 근거리 통신망의 경우 a는 0.01 - 1 범위 사이의 값을 갖게 된다.

5.4.2 성능 평가 변수

본 성능 분석에서는 네트워크의 스테이션(범용 컴퓨터와 전단 컴퓨터) 수, 스테이션들 간의 전송 구분 간격 시간(inter-transmission interval), 전송되는 프레임의 평균 길이 들을 통신망의 부하 함수로써 구성하여 통신망의 처리량(throughput)과

프레임의 평균 전송 지연 시간을 측정하고자 한다[MACD 87]. 처리량은 단위 시간당 전송된 프레임수(혹은 바이트수, 비트수)를 의미하며 실제적으로는 데이타의 성공적 전송을 위해 사용된 채널의 이용도를 나타낸다. 그리고 처리량은 변수 S로써 표시한다. 프레임의 평균 전송 지연 시간 T_d는 한 프레임의 전송 의도가 시작된 시간 부터 전송이 완료된 때까지의 시간을 나타낸다. 따라서 T_d에는 채널의 사용을 위한 대기 시간과 프레임의 실제 전송 시간이 포함된다. 그리고 이 T_d는 정규화된 평균 전송 지연 시간(normalized mean delay) D에 의해 의미를 갖는다. 따라서 채널의 사용을 위한 대기 시간이 고려되지 않는 경우의 프레임 전송 시간을 T_f라 한다면 D는 다음과 같이 나타낼 수 있다.

$$D = T_d / T_f \qquad (4)$$

본 고속 고신뢰 상호 결합망에서의 T_f는 프레임의 비트 길이와 데이타 전송률 50 Mbps를 곱한 결과가 그 값이 결정된다.

통신망상의 부하는 주로 채널의 경쟁 효과를 배제한 경우의 입력 부하(offered load)로써 표시하게 된다. 한 스테이션이 한 프레임을 성공적으로 전송한 후 다음 프레임의 전송을 시작할 때까지의 시간 T_i 즉 프레임의 평균 전송 간격과 스테이션 수 N을 변수로 한 고속 고신뢰 상호 결합망의 입력 부하 G는 다음과 같다.

$$G = (N * T_f)/(T_i + T_f) \qquad (5)$$

그리고 시뮬레이션을 위해 설정된 각 성능 평가 매개 변수의 값은 다음과 같이 정의된다.

$N = 32$
$T_f = 0.25$ ms
$G = 0.25 - 2.0$
$T_i = (N/G - 1) * T_f$
$a = 0.02$

5.4.3 시뮬레이션 모델

본 고속 고신뢰 상호 결합망에서는 두 개의 고속 근거리 통신망이 사용된다.

그리고 각 고속 근거리 통신망은 충돌 회피 CSMA 방식에 의하여 통신이 수행된다. 충돌 회피 CSMA 알고리즘에서는 각 스테이션에 세 개의 타이머 PAT, AAT, RT가 장치되어 있음을 가정하며 이 알고리즘의 구체적 내용은 앞 절에서 이미 언급된 바 있음으로 내용의 구체적 기술은 피하기로 한다. 그리고 N개의 스테이션은 물리적인 순서와 스테이션의 포트 순서가 일치하도록 배열하였다(그림 5.12 참조). 범용 컴퓨터와 전단 컴퓨터간의 통신은 물론 범용 컴퓨터간의 통신도 상호 결합망 내에서 수행되는 것을 가정하였으므로 전용 컴퓨터를 위한 전단 컴퓨터의 수는 N개의 범위 내에서 가변적으로 조정될 수 있다. 그리고 스테이션 간에는 1개의 프레임 단위가 전송됨을 원칙으로 한다. 그러나 PAT를 이용하는 다중 프레임(multiframe)의 전송도 가능하다. 충돌 회피 CSMA 방식을 사용하여 프레임을 전송하는 시뮬레이션 과정은 그림 5.13과 같다.

5.4.4 성능 분석

입력 변수 G는 0.25 부터 2.0 까지의 값으로 변화된다. 그리고 처리량 S와 정규화된 평균 전송 지연 시간(normalized mean delay) D는 통신망이 1 개인 경우와 2 개인 경우에 대해 다음과 같이 각각 나타낸다.

	S	D
통신망이 1 개인 경우 :	S_1	D_1
통신망이 2 개인 경우 :	S_2	D_2

시뮬레이션을 통한 D와 S의 각 분석 결과는 그림 5.14와 그림 5.15에 있다.

a. 정규화된 평균 전송 지연 시간

그림 5.14에서는 입력 부하 G가 증가함에 따라 D_1과 D_2가 증가하는 결과를 비교하고 있다. G가 0.25 부터 0.5 사이의 범위에 있는 경우에는 D_1과 D_2가 큰 차이가 없다. 그러나 G가 0.75 이상인 경우 부터 D_1은 D_2에 비해 지수 함수적으로 급격히 증가하고 있다. 즉 입력 부하가 커질수록 전송되는 프레임의 평균 전송 시간은 매우 지연되어 충돌이 없는 경우의 프레임 전송 시간인 T_f와 비교한 D_1의 값이 역시 매우 증가됨을 알 수 있다. 그러나 통신망이 2 개인 경우에는 프레임의 평균 전송 시간이 입력 부하 G의 증가에 따라 지연되긴 하나 D_1에 비하여 급격한

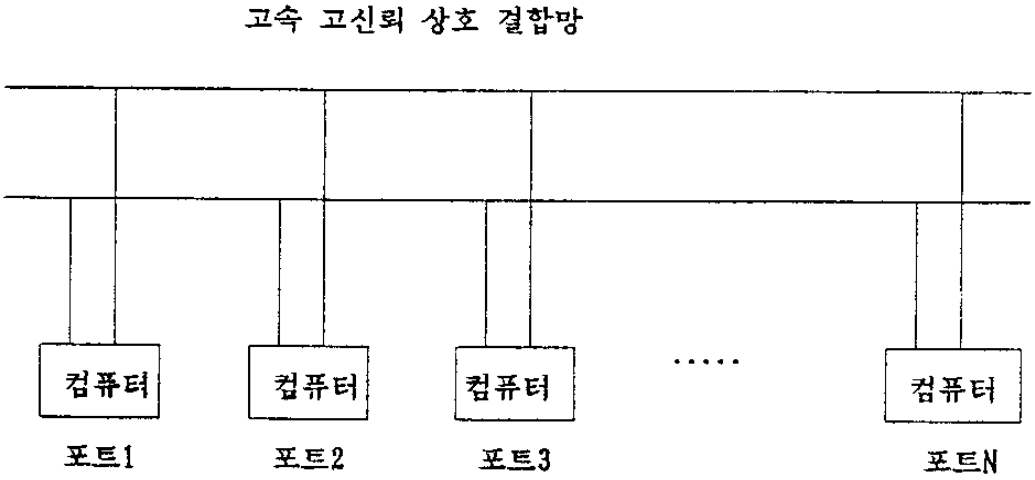


그림 5.12 스테이션의 물리적인 순서와 포트 순서

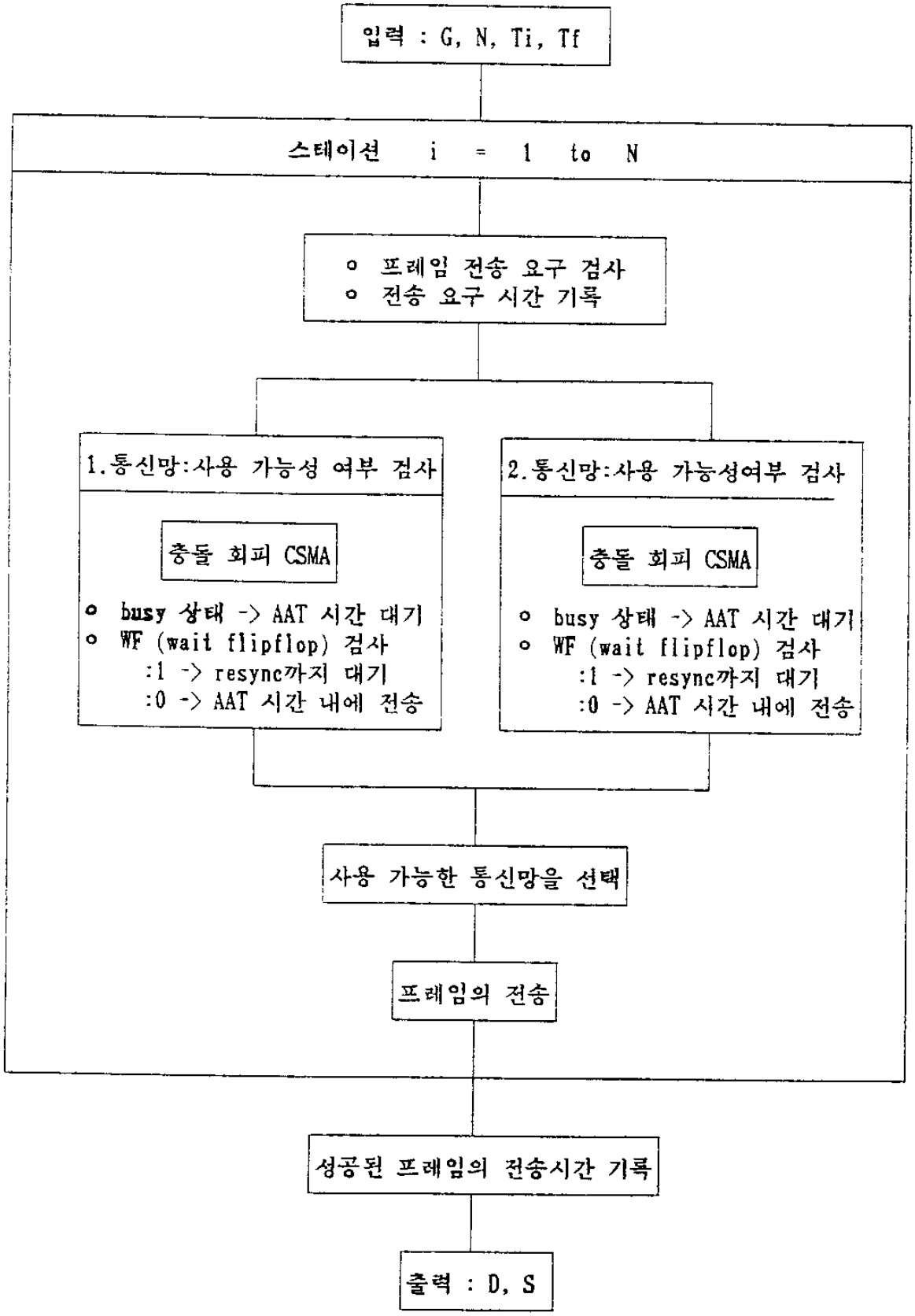


그림 5.13 시뮬레이션 모델의 유통도

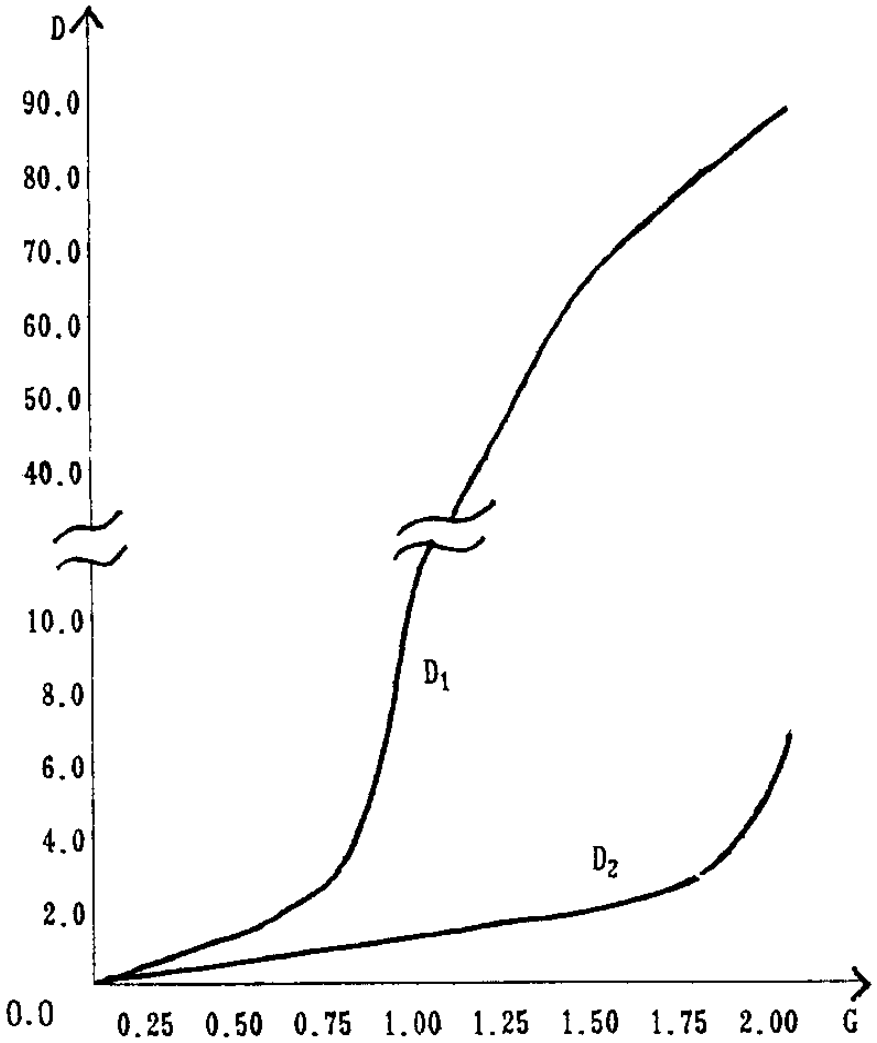


그림 5.14 프레임의 정규화된 평균 전송 지연 시간 비교(D_1, D_2)

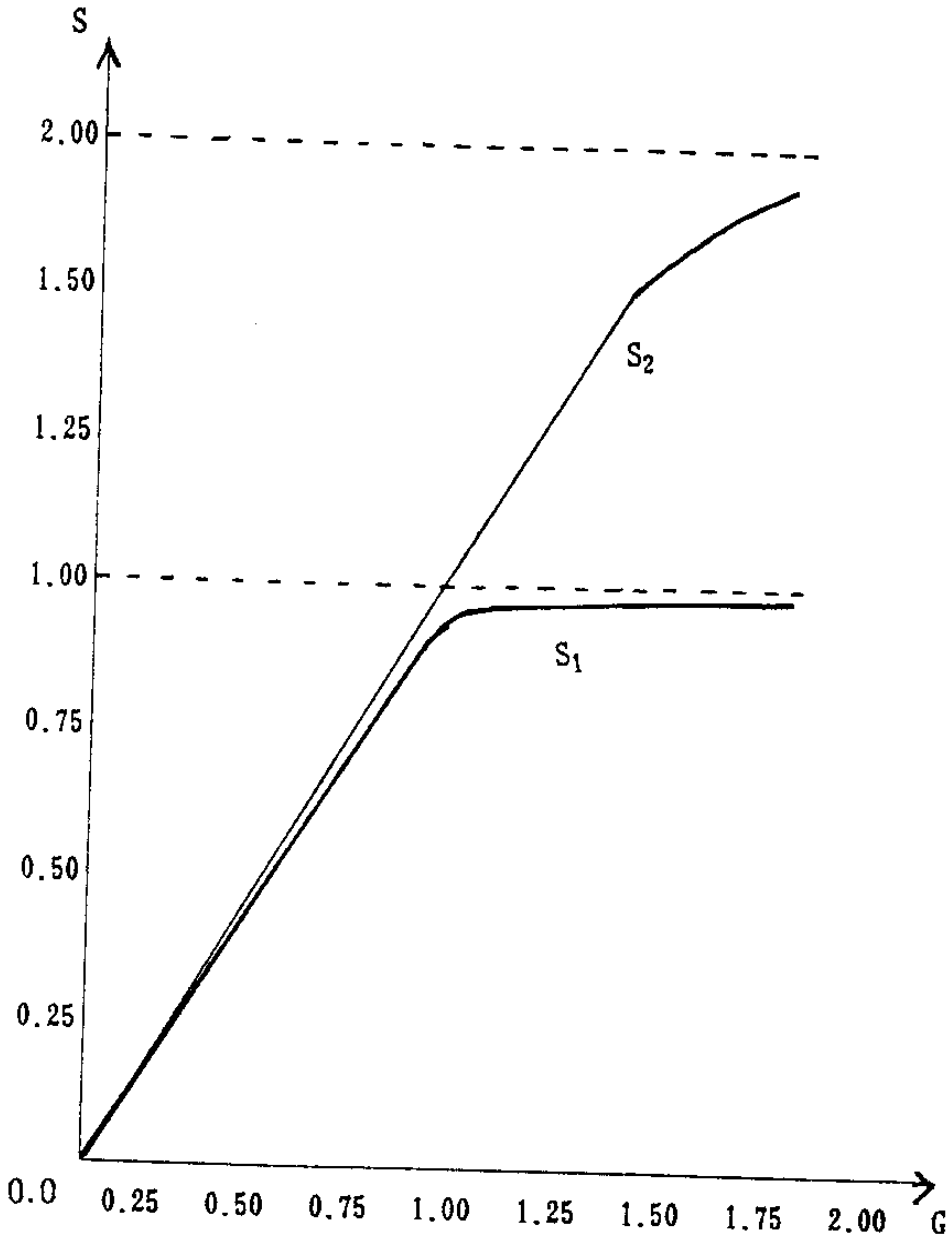


그림 5.15 처리량의 비교(S_1, S_2)

증가 현상이 나타나고 있지는 않다. 통신망이 이중 구조로 구성되어 있으므로 각 노드(범용 컴퓨터 혹은 전용 컴퓨터)는 idle 상태인 통신망을 선택하여 프레임을 전송할 수 있으며 전송되기 까지 대기되는 지연 시간을 감소시킬 수 있게 된다. 따라서 통신망이 이중 구조로 구성되는 경우는 단일 구조로 구성되는 경우보다 프레임의 평균 전송 지연 시간을 감소시킬 수 있다.

b. 처리량

그림 5.15에서는 입력 부하의 증가에 따라 처리량 S_1과 S_2이 증가되는 결과를 비교하고 있다. G가 1.0 이하인 경우에는 S_1과 S_2는 큰 차이가 없음을 알 수 있다. 즉 일정 시간 동안 프레임의 성공적 전송을 위하여 사용된 각 채널의 이용도는 실제 비슷한 것으로 간주할 수 있다. 그러나 G가 1.0 이상인 경우에는 S_1의 변화가 거의 없다. 입력 부하가 계속 증가함에도 불구하고 처리량은 증가되지 않고 거의 일정한 값을 유지하므로 채널의 이용도가 한계값에 도달한 것임을 알 수 있다. 반면에 S_2는 통신망이 이중 구조로 구성되어 있어 입력 부하가 2.0 까지 증가해도 한계값에 도달하지 않고 계속적으로 채널의 이용도가 증가하는 현상을 나타내고 있다. 물론 G가 2.0보다 더 커지는 경우에는 통신망이 이중 구조로 구성된다 하더라도 일정한 입력 부하의 값을 기점으로 역시 채널의 이용도는 한계값에 도달하게 된다. 따라서 이중 구조로 구성된 통신망 구조는 한 개로 구성된 통신망 구조에 비해 입력 부하가 2.0 이하인 경우 처리량이 2 배 가까이 증가됨을 알 수 있다.

이와 같이 고속 근거리 통신망이 이중으로 구성된 본 고속 고신뢰 상호 결합망에서 프레임의 평균 전송 지연 시간이 감소되고 처리량이 증가되는 결과를 시뮬레이션을 통하여 분석하였다. 물론 이중 구조로 구성되므로 단일 구조로 구성되는 경우에 비하여 구현시 설치 가격이 2 배 이상이 되는 단점도 있다. 그러나 한 개의 통신망에 결함이 발생하여 사용이 불가능한 경우 나머지 한 개의 통신망으로도 계속적인 통신을 수행할 수 있으므로 이중 구조로 구성되는 고속 고신뢰 상호 결합망의 경우 성능의 개선은 물론 신뢰성도 향상시킬 수 있다.

6. 자원 공용 분산 운영체제

6.1 운영체제의 구조

고속 분산 처리 시스템은 기존의 워크스테이션들과 새로운 실행모델을 기반으로 설계된 전용 컴퓨터들이 고속 고신뢰의 상호 결합망과 전단 컴퓨터를 통하여 연결된 분산 처리 시스템이다. 기존의 워크스테이션은 사용자 인터페이스를 담당하며, 전용 컴퓨터는 워크스테이션의 계산서버로 사용된다. 자원 공용 분산 운영체제는 고속 분산 처리 시스템의 각 자원을 통합적으로 운영하는 분산 운영체제이다. 이기종 전용 컴퓨터들을 통합적으로 운영하기 위해서 운영 체제는 다음과 같은 요구조건을 만족해야 한다.

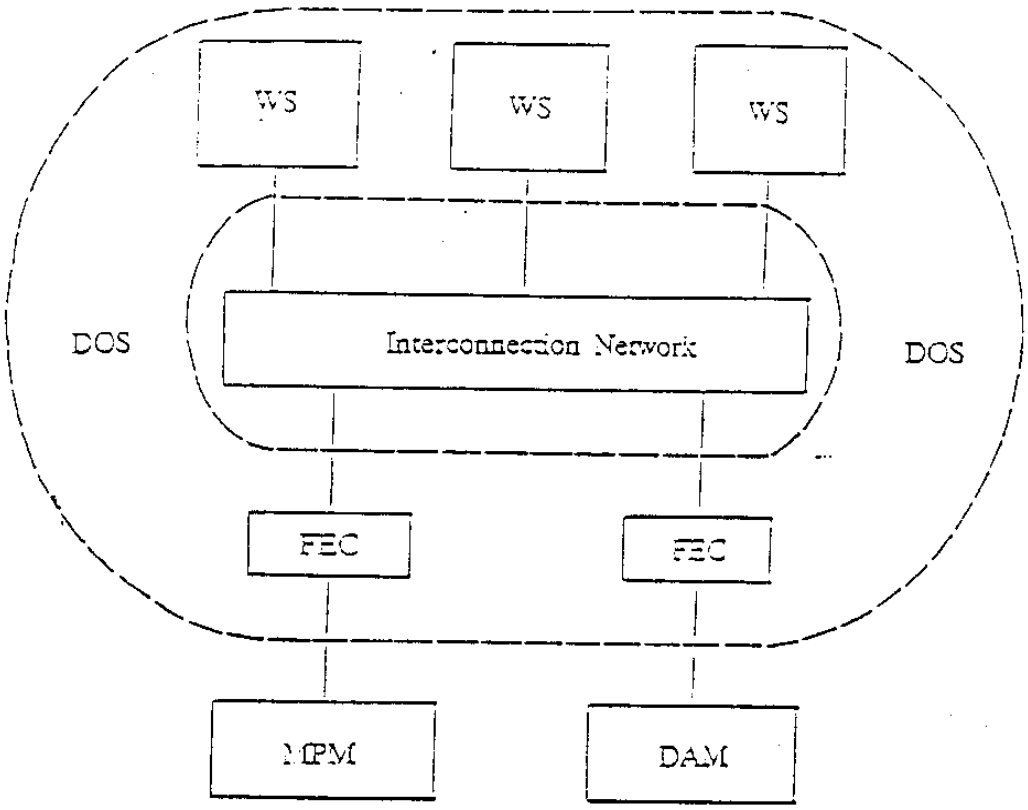
(1) 프로세서의 분산실행을 효과적으로 지원해야 한다.

본 시스템은 하나의 프로세서를 프로시듀어(procedure) 단위로 다수의 컴퓨터에 분산시켜 실행시키는 기능을 제공하고 있다. 이를 위해서는 두가지 요구가 만족되어야 한다. 첫째, 간편하고 효과적인 사용자 인터페이스가 제공되어야 한다. 프로시듀어의 분산실행 호출및 동기화를 위한 간편하고 효과적인 사용자 인터페이스를 제공함으로써, 사용자가 구체적인 실행절차를 모르더라도 효과적으로 프로그램을 작성할 수 있도록 해야 한다. 둘째, 고속의 통신을 지원해야 한다. 본 시스템의 성능은 프로세서의 분산 실행의 성능에 크게 좌우된다고 할수 있다. 따라서 운영체제에서는 효율적인 통신 규약(protocol)을 사용하는 등, 고속의 통신을 지원해야 한다.

(2) 공용 자원에 대한 투명한(transparent) 사용자 인터페이스를 지원해야 한다.

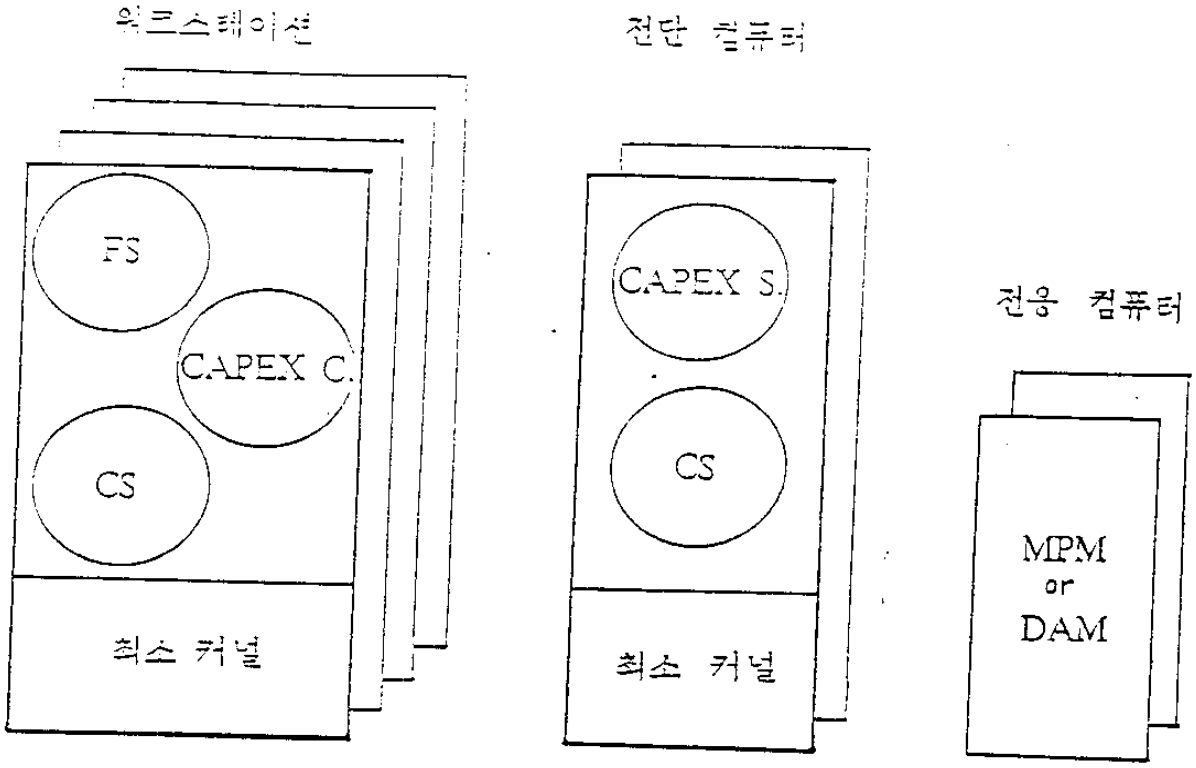
본 시스템의 전용 컴퓨터는 계산 서버로 작동한다. 따라서 운영체제에서는 전용 컴퓨터를 일종의 공용 자원으로 간주할수 있다. 기존의 공용 자원(화일 시스템)과 함께 전용 컴퓨터에 대한 네트워크 투명한 사용자 인터페이스를 제공해야 한다.

고속 분산 처리 시스템에서의 자원 공용 분산 운영체제의 실행환경을 그림 6.1.1이



WS : Workstation
FEC : Front End Computer
MPM : Math Package Machine
DAM : Divide and conquer Algorithm Machine
DOS : Distributed Operating System

그림 6.1.1 자원 공용 분산 운영체제의 실행 환경



FS : 화일 서버 (File Server)

CS : 통신 서버 (Communication Server)

CAPEX C. : 분산 실행 클라이언트

(Cross Architecture Procedure Execution Client)

CAPEX S. : 분산 실행 서버

(Cross Architecture Procedure Execution Server)

MPM (Math Package Machine)

DAM (Divide and conquer Algorithm Machine)

그림 6.1.2 자원 공용 분산 운영체제의 구조

보이고 있다. 사용자와의 인터페이스를 담당하는 범용 컴퓨터로서의 워크스테이션과 범용 컴퓨터와 계산 서버인 전용 컴퓨터간의 통신 인터페이스를 담당하는 전단 컴퓨터 위에서 자원 공용 분산 운영 체제가 수행된다.

자원 공용 분산 운영 체제의 구조는 그림 6.1.2와 같다. 본 연구에서는 분산 운영체제의 신뢰성과 융통성을 높이고, 시스템 소프트웨어의 제작을 용이하게 하고, 이기종의 전용 컴퓨터로 구성된 분산 시스템의 구조에 적합하게 하기 위하여, 자원 공용 분산 운영체제는 최소 커널과 각종 서버로 구성하였다. 기존의 UNIX 운영체제(System V)의 커널을 기반으로 본 연구에 맞게 확장한 커널은 그 기능을 가급적 최소화시켜 프로세서 관리(process management), 메모리 관리(memory management)와 프로세스간 통신(inter_process communication) 으로 구성하였고, 그외의 기능은 각종 서버가 제공하도록 한다. 화일 서버나 통신 서버는 사용자 공간에서 수행되며, 특별히 사용자 프로그램의 분산 실행을 지원하기 위해 분산 실행 시스템을 이루는 CAPEX 클라이언트와 서버가 사용자 공간에서 제공된다.

6.2절에서는 최소 커널에 대한 상세 설계로, 중심 개념과 이들의 primitive들을 pseudo 코드 수준으로 기술한다. 본 분산 처리 시스템의 가장 큰 특징은 워크스테이션의 사용자 프로그램이 일부 Mach Package 전용 컴퓨터나 Divide and Conquer 전용 컴퓨터에서 수행되어 그 결과를 돌려받도록 하는 것이다. 이러한 메카니즘을 효율적으로 수행하도록 설계된 분산 실행 시스템이 6.3절에서 기술된다.

6.2 최소 커널의 상세 설계

본 운영체제는 최소 커널과 각종 서버들로 구성된다. 최소 커널은 기존 운영체제(System V)의 커널중 프로세스 관리, 메모리 관리, 프로세스간 통신이 선택되었으며 이들을 본 시스템의 분산 처리 환경으로 확장하였다. 한편 분산 실행 시스템과 같은 기능은 서버로 최소커널 위의 프로세스로 존재하게 된다. 이렇게 되면 이러한 서버 프로세스와 사용자 프로세스는 구별되지 않는다. 커널 외부의 프로세스는 항상 커널을 통해서 통신해야 한다. 그리고 프로세스들은 다른 프로세스와 메시지 전달을 통해 통신하게 되는데, 이를 위해 각 프로세스는 메시지 큐(message queue)를 갖게 된다. 이와 같이 커널을 최소화 하는 방법은 다음과 같은 장점을 가지고 있다. 첫째, 신뢰성이 높다. 커널 이외의 프로세스들은 항상 커널을 통해서 통신해야 한다. 따라서 최소 커널이외의 프로세스들(기존 운영체제의 커널 프로세스일 수도 있다)의 테스팅(testing)과 에러 고립(isolation)이 용이하므로, 한 프로세스의 파괴가 전체 운영체제의 파괴를 가져오지 않는다. 둘째, 융통성이 높다. 이와 같은 방법은 커널 외부의 프로세스들간의 인터페이스를 정확하고 간편하게 정의하고 있으므로 운영체제의 융통성이 높다. 예를 들어 여러개의 화일 시스템중 특정 화일 시스템의 화일 서버 프로세스와 통신하여 화일 시스템을 엑세스하게 된다. 세째, 시스템 소프트웨어(system software)의 제작이 용이하다. 시스템 소프트웨어가 커널 외부의 프로세스로 제작된다면, 응용 프로그램의 경우와 동일 하므로 시스템 소프트웨어 제작이 용이하다. 네째, 분산 시스템의 구조에 적합하다. 각 프로세스들의 분산 실행이 용이하므로, 분산 시스템의 구조에 알맞다.

다음의 각 절에서는 본 연구에서 설계한 최소 커널의 구성 요소들인 프로세스 관리, 메모리 관리, IPC 등에 관해 기술한다. 각 부분들은 기존 UNIX에서 확장 수정될 부분들을 pseudo 코드 수준으로 설계된다.

6.2.1 프로세스 관리

본 연구의 운영체제는 시스템으로 들어오는 모든 작업을 프로세스형태로 수행하는 프로세스 모델로 한다. 기존 운영체제에서 사용하는 프로세스 모델은 수행고자하는 작업에 병렬성이 존재할 때, 동일한 내용을 포함한 여러 프로세스(HWP: Heavy Weight process)를 생성하여 병렬처리를 한다. 이러한 경우는 프로세스 생성에서 많은 자원할당에 의한 생성기간의 지연과 프로세스의 유지 관리에 오버헤드가 따르게 된다.

이러한 문제점이 근본적으로 기존의 프로세스가 유지해야 할 정보가 많은데서 기인하므로 본 시스템에서는 관련된 상태정보를 최소화한 프로세스(LWP: Light Weight Process) 개념을 도입하여 HWP와 병행하여 사용가능하게 한다. LWP란 생성한 프로세스의 주소공간을 공유하는 프로세스로 공유되지 않는 상태정보만 유지한다.

LWP에 대한 프리미티브를 제공함에 있어 기본적인 성격을 지닌 최소의 인터페이스만 제공하여 LWP의 통제를 사용자 수준에서 하게 함으로써 여러가지 응용이 쉽게 구현되도록 한다. 병렬성이 존재하는 작업 부분은 여러 LWP로 나누어 각각을 여러 가용 CPU로 이동시켜 실행시키고 그동안 이동된 LWP들과 병렬성이 있는 부분까지 원래의 작업을 계속 실행할 수 있게 하여 최대의 병렬성을 얻게함으로써 전 시스템의 효율을 높인다.

(1) 사용자 인터페이스

프로세스 관리를 사용자 측면에서 쉽게 이해할 수 있게 프로세스의 생성, 종료 및 기타 시스템 호출을 기술한다.

a. 프로세스의 생성

본 운영체제는 두가지 형의 프로세스 사용을 지원하므로 프로세스 생성 시스템 호출을 다음 두가지로 정의한다.

- create (filename, argv, envp)

create는 새로운 child 프로세스를 생성한다. 먼저 가용 상태정보 리스트에서 한 노드를 선택하여 생성할 child 프로세스의 상태정보를 저장하여 사용 상태정보 리스트에 삽입한다. 이때 주소공간의 텍스트와 데이타 영역은 프로그램 화일 filename의 내용으로 설정되고, 사용자 스택 영역은 적당한 크기로 설정되어 초기화된다. 시스템 호출의 결과는 child 프로세스의 번호가 된다. argv는 실행 프로그램의 문자 스트링 파라메터의 배열이고, envp는 생성되는 프로세스의 환경을 나타내는 문자 스트링의 배열이다.

- lcreate (sysname)

lcreate는 호출한 프로세스의 주소공간(텍스트 및 데이타 영역)을 공유하는 lwp를 생성한다. 이때 사용자 스택 영역의 주소공간은 호출한 프로세스의 스택환경을 복사한다. sysname이 0이면 호출된 시스템내에서 생성되고, sysname이 주어지면 원격실행 관리자에 의해 원격 시스템 sysname에 생성된다. 새로이 생성된 lwp 프로세스내에서는 lfork가 0의 값을 복귀하고, 호출한 프로세스내에서는 생성된 lwp의 프로세스 번호를 복귀한다.

b. 프로세스의 종료 - exit (status)

exit는 기다고 있는 parent 프로세스에게 정상종료 또는 비정상 종료를 나타내는 status를 보고하고, 호출한 프로세스를 종료시킨다. 커널은 내부의 시그널 응답으로 해당 프로세스를 종료시킬 수 있다. hwp 프로세스의 종료는 주소공간을 모두를 회수하지만 lwp는 사용자 스택 영역의 주소공간만 회수한다.

c. 프로세스의 동기화

- suspend (lwp_pid)

한 hwp가 생성한 lwp 프로세스간에 실행을 동기화한다. 인수로 지정된 lwp_pid에 해당하는 프로세스가 resume 시스템 호출이 이루어질 때까지 대기상태가 된다.

- resume (lwp_pid)

한 hwp가 생성한 lwp 프로세스간에 실행을 동기화한다. 인수로 지정된 suspend된 프로세스를 실행상태로 한다.

- signal (sig, function)

signal 시스템 호출은 호출한 프로세스가 시그널을 받았을 때 함수 function에 정의된 내용으로 시그널 처리를 하게 한다. signal 시스템 호출에 의한 시그널 처리가 명시되지 않은 프로세스가 시그널을 받는 경우는 프로세스를 종료하면서 parent 프로세스에게 시그널 번호를 넘겨준다. sig의 값은 'death of lwp' 시그널을 제외하고는 기존 UNIX 운영체제와 동일하다.

- kill (pid, sig)

kill 시스템 호출은 pid에 의해 지정된 프로세스에게 sig 시그널을 보낸다. pid가 0 이면 호출한 프로세스의 parent 프로세스에 의해 생성된 모든 child 프로세스, lwp 프로세스에 시그널을 보낸다.

- wait (wait_stat)

wait 시스템 호출은 종료된 child 프로세스나 lwp 프로세스를 발견할 때까지 호출 프로세스를 대기상태로 한다. wait_stat는 종료된 child 프로세스 또는 lwp 프로세스가 넘겨주는 값의 주소를 지정한다.

d. 상태정보 검색 - getstat (status)

프로세스 테이블의 해당 엔트리에 있는 프로세스 상태정보를 검색하는 시스템 호출로 status는 복귀된 상태정보를 저장한 structure의 주소를 지정한다.

(2) 주요 자료구조 및 알고리즘

한 프로세스의 상태정보는 프로세스 테이블내의 한 엔트리와 실행상태를 저장한 블록에 저장된다. 프로세스 테이블의 엔트리는 아래의 필드로 구성된다.

· 프로세스 상태를 저장한 상태 필드
· 프로세스의 종류(hwp, lwp) 표시필드
· 현재 실행상태를 저장한 블록의 주소필드
· 사용자 id 필드
· 프로세스 id 리스트 필드
· 시그널을 처리하는 함수의 주소필드
· 스케줄링에 사용되는 자료필드
· 타이머 필드
· 기타

프로세스 id 리스트에는 hwp인 경우는 parent id, child id 및 lwp id를 포함하고, lwp인 경우는 lwp를 생성한 프로세스 id를 포함한다. 프로세스 id는 (sysname, pid) 쌍으로 정의되며, 로칼 시스템에 존재하는 프로세스 id의 sysname은 0이고 원격 시스템의 sysname은 분산 환경에서 정의된 시스템명이다. 현재 실행상태를 저장한 블록에는 실행시간에 사용되는 스택과 프로세스 주소공간 테이블이 포함되어 있다. 실행시간 스택에는 커널 모드에서 프로세스의 실행제어를 위한 실행 주소, 레지스터 상태가 보관된다. 프로세스 주소공간 테이블은 텍스트, 데이타, 사용자 스택의 가상주소를 저장한다.

a. create (filename, argv, envp)

입력 : 실행 화일명,
　　파라메터 리스트,
　　환경 변수 리스트
출력 : 호출된 프로세스에서는 생성된 child의 프로세스 번호,
　　생성된 child 프로세스에서는 0

{

　　분산 화일 시스템 서버에 실행화일 filename을 요청;
　　사용가능한 프로세스 테이블 엔트리와 실행상태 저장 블록을 할당;

```
        프로세스 번호를 지정;
        생성하는 프로세스의 상태를 "생성중"으로 표시;
        프로세스 테이블 엔트리를 초기화;
        환경변수 리스트로 실행상태 저장 블록을 초기화;
        실행화일에 명시된 주소공간 영역을 기억장치에 적재;
        주소공간의 사용자 스택을 초기화;
        if (실행프로세스가 parent 프로세스) {
                child 프로세스의 상태를 "준비상태"로 변경;
                return(child 프로세스 번호);
        } else {
                실행상태 저장 블록의 타이밍 필드를 초기화;
                return(0);
        }
}
```

b. lcreate (sysname)

```
입력 : 생성될 시스템명
출력 : 호출 프로세스에서는 lwp의 번호,
       생성된 lwp에서는 0

{
        if (프로세스가 원격 생성) {
                프로세스 상태정보를 포함하는 패킷 구성;
                구성된 패킷을 네트워크 요구 처리 루틴에 전달;
                결과 패킷을 받을 때까지 대기;
                return(시스템명, lwp 프로세스 번호);
        } else {
              사용가능한 프로세스 테이블의 엔트리와 실행상태 저장블록을 할당;
```

```
프로세스 번호를 지정;
생성하는 프로세스 상태를 "생성중"으로 표시;
if (네트워크 요구 처리 루틴으로부터의 호출)
        전달받은 패킷으로 생성하는 프로세스의 프로세스 테이블의
        엔트리와 실행정보 저장 블록을 구축;
else
        호출 프로세스의 상태정보로 생성하는 프로세스의 프로세스
        테이블의 엔트리와 실행정보 저장블록을 구축;
분산화일 서버에 호출 프로세스가 사용하는 프로세스가 공유됨을
메시지로 보냄;
텍스트와 데이타 영역의 영역 참조 카운터를 증가; /* 텍스트 영역과
데이타 영역이 공유됨 */
사용자 모드로 복귀할 때 실행환경이 생성되는 lwp 주소공간이 되도록
실행시간 스택을 수정;
switch (프로세스) {
        case 호출 프로세스 :
                lwp 상태를 "준비상태"로 변경;
                return(lwp 프로세스 번호);
        case 다른 노드에서 이전해 온 lwp 프로세스 :
                시스템명과 lwp 프로세스 번호를 포함하는 패킷을
                구성하여 네트워크 요구 처리 루틴에 전달;
                break;
        case lwp 프로세스 :
                실행상태 저장 블록의 타이밍 필드를 초기화;
                return(0);
```

c. exit (status)

입력 : return 코드

출력 : 없음

```
{
        모든 시그널은 무시;
        if (프로세스가 hwp) {
                if (제어 터미날과 연관된 프로세스 그룹의 리더) {
                        프로세스 그룹의 모든 멤버에게 hangup 시그널을 보냄;
                        모든 멤버의 프로세스 그룹은 0으로 설정;
                }
                생성된 모든 lwp 프로세스에게 kill 시그널을 보냄;
                프로세스와 관련된 영역들을 모두 회수;
        } else /* 프로세스가 lwp */ {
                실행정보 저장 블럭과 사용자 스택 회수;
                텍스트와 데이타 영역의 영역참조 카운터를 감소; /* 원격 주소
                공간인 경우는 네트워크 요구 처리 루틴을 호출 */
        }
        분산화일 서버에 화일 정리를 요청;
        프로세스 상태를 "zombie"로 함;
        모든 자식 프로세스의 부모 프로세스 번호를 init 프로세스 번호로 함;
        부모 프로세스에게 "death of child" 또는 "death of lwp"
        시그널을 보냄; /* 부모 프로세스가 원격 시스템에 있는 경우 네트워크
        요구 처리 루틴을 호출 */
        "context switch"  실행;
```

d. suspend (lwp_pid)

입력 : suspend할 프로세스 번호
출력 : 없음

```
{
        if (suspend할 lwp가 원격 프로세스) {
                suspend 패킷을 구성하여 네트워크 요구 처리 루틴에 전달;
                sleep(대기완료);
        } else {
                suspend될 lwp의 상태를 대기상태로 함;
                원격 요청이면 대기완료 패킷을 구성하여 네트워크 요구 처리 루틴에
                전달;
        }
}
```

e. resume (lpw_pid)

입력 : resume할 프로세스 번호
출력 : 없음

```
{
        if (resume할 lwp가 원격 프로세스)
                resume 패킷을 구성하여 네트워크 요구 처리 루틴에 전달;
        else
                resume할 lwp의 상태를 준비상태로 함;
}
```

f. signal (sig, function)

입력 : 시그널 번호, 시그널 처리 함수 주소

출력 : 없음

{

프로세스 테이블 엔트리의 시그널 리스트 필드의 해당 노드에
함수주소 삽입;

}

g. kill (pid, sig)

입력 : 프로세스 번호, 시그널 번호
출력 : 없음

{

if (pid가 양수)

프로세스 번호가 pid인 프로세스에게 kill 시그널을 보냄;
/* pid가 원격 프로세스이면 kill 패킷을 구성하여 네트워크 요구
처리 루틴에 전달 */

else /* pid가 0 */

호출 프로세스의 parent 프로세스 또는 호출 lwp를 생성한
프로세스가 생성한 모든 child, lwp 프로세스에게 kill 시그널을
보냄; /* 원격 프로세스가 포함되어 있으면 프로세스별로
kill 패킷을 구성하여 네트워크 요구 처리 루틴에 전달 */

}

h. wait (wait_stat)

입력 : 현재의 프로세스 상태를 저장하는 변수의 주소
출력 : zombie 프로세스 번호, exit 코드

{

if (기다리는 프로세스의 child 또는 lwp 프로세스가 없음)

```
                return(error);
        for (;;) {
                if (기다리는 프로세스가 zombie child 또는 lwp를 가짐) {
                        임의의 zombie 프로세스를 선택;
                        zombie 프로세스의 CPU 사용시간을 기록;
                        zombie 프로세스의 프로세스 테이블 엔트리를 회수;
                        return(zombie 프로세스 번호, exit 코드);
                }
                sleep(zombie 프로세스 생성);
        }
}
```

i. getstat (status)

입력 : 프로세스의 상태정보 주소
출력 : 없음

```
{
        프로세스 테이블 엔트리의 상태정보를 structure 변수에 보관;
        변수의 주소를 status 값으로 함;
}
```

6.2.2. 메모리 관리

범용 워크스테이션들을 Ethernet과 같은 근거리 통신망으로 상호 연결하여 구성한 분산 처리 시스템에서 작업의 효율을 높이고 값비싼 자원들을 최대한 활용하기 위하여 본 분산 운영체제에서는 최소화된 프로세스(Light Weight Process : LWP)의 원격 실행 기능을 제공하고 있다. 예를 들어, 한 노드에서 실행 중이던 프로세스가 새로 최소화된 프로세스(LWP)를 하나 생성하였다고 가정하자. 이 때 생성된 LWP가 보다 효율적으로 실행되어야 하거나 그 노드가 과부하 상태에 있을 경우, 커널은 네트워크상에서 유휴한 상태 혹은 비교적 부하가 적은 상태에 있는 노드를 찾아내어 그 곳으로 대상 프로세스에 관한 정보들을 이전한 다음, 이를 실행시킨 후 그 결과를 원래의 노드로 가져오게 된다.

이 절에서는 메모리 관리를 위해 필요한 자료구조와 관련 시스템 프로세스 및 응용 프로세스의 원격 실행 기능을 보다 효율적으로 지원하기 위하여 고안된 네트워크 요구 페이징(network demand paging) 기법에 관하여 설명하기로 한다. 네트워크 요구 페이징 기능을 제공하기 위하여 본 분산 운영체제에서는 커널 내부의 페이지 부재 오류 처리 루틴(page fault handler)과 네트워크 요구 처리 루틴(Network Request Handler : NRH)에 이 기능을 위한 코드를 포함하고 있다.

(1) 메모리 관리를 위한 자료구조

본 운영체제의 커널은 메모리 관리 및 페이징 기능을 위하여 다음과 같은 자료구조를 가지고 있다.

- 페이지 테이블 (page table)
- 디스크 블럭 지시자 (disk block discriptors)
- 페이지 프레임 테이블 (page frame table)
- 스왑 테이블 (swap table)

페이지 테이블의 각 엔트리는 해당 페이지의 물리적 주소와 그 페이지의 유효, 참조, 수정, 쓰기시-복사(copy-on-write), 페이지 원격 이전(mflag) 여부 및 나이, 보호 등에 관한 정보를 포함하고 있으면서 하나의 디스크 블럭 지시자와 연관되어 있다. 디스크 블럭 지시자는 그와 연관된 페이지 테이블 엔트리의 해당 페이지가 스왑 영

역(swap area) 혹은 화일 시스템 내에서 어디에 위치하는가를 알려주는 자료구조이다. 페이지 프레임 테이블은 메모리 내의 각 페이지에 해당하는 엔트리를 가지고 있는데, 커널은 이 테이블의 각 엔트리를 연결하는 두 개의 리스트를 유지하고 있다. 그 중 하나는 가용 페이지들을 연결해 두는 리스트(free list)이고 나머지 하나는 해쉬 리스트(hashed list)인데, 후자는 어떤 프로세스가 특정한 가상 주소에 대하여 페이지 부재 오류를 일으켰을 때 원하는 페이지가 메모리 내의 다른 곳에 이미 존재하는가를 알 수 있도록 하는 자료구조로서 보조 기억 장치로부터의 불필요한 읽음작업을 피할 수 있도록 해준다. 스왑 테이블의 각 엔트리는 스왑 영역에 존재하는 모든 페이지 각각에 대응하는 것으로서 그 페이지를 지칭하고 있는 페이지 테이블 엔트리의 갯수를 포함하고 있다.

(2) 페이지 스틸러(page stealer)

페이지 스틸러는 최근에 더이상 참조하지 않은 메모리 페이지를 스왑 디바이스(swap device)로 옮기는 시스템 프로세스로 항상 커널 모드에서 실행된다. 이 프로세스는 시스템 초기화시 커널에 의해 생성되어 그 시스템이 작동하는 동안 항상 존재한다. 커널은 시스템 내의 가용 메모리 양이 일정 수준 이하로 떨어졌을 경우 페이지 스틸러를 깨워 가용 메모리 양이 일정 수준 이상 확보될 때까지 선택된 페이지들을 스왑 디바이스로 옮긴다. 페이지 스틸러는 스왑 디바이스로 옮길 페이지를 선정하기 위하여 각 페이지들을 대상으로 에이징(aging) 기능을 추가로 수행하게 된다.

(3) 네트워크 요구 페이징 메카니즘

네트워크 요구 페이징이 수행되는 개략적인 메카니즘은 다음과 같다.

한 응용 프로세스가 실행 도중 페이지 부재 오류(page fault)를 발생시켰을 경우 제어가 메모리 관리 시스템 내의 페이지 부재 오류 처리 부분으로 이동한다. 페이지 부재 오류 처리 부분에서는 먼저 해당 페이지의 이전 여부에 관한 검사 과정을 거친 후 그 페이지가 위치하고 있는 노드의 네트워크 주소와 그 노드 내에서의 관련 프로세스 식별자 및 페이지 부재 오류를 발생시킨 가상 주소 등을 네트워크 요구

처리 루틴(NRH)에 전달해 준다. 그리고나서 페이지 부재 오류를 발생시켰던 응용 프로세스는 '네트워크 페이지 반입 완료'라는 사건(event)에 대하여 수면 상태(sleep state)에 들어가게 된다. NRH는 페이지 부재 오류 처리 부분에서 구성한 정보를 근거로 페이지를 반입할 대상 노드에 전송할 요청 패킷을 구성한 후 이를 네트워크 디바이스 구동 루틴을 통하여 대상 노드에 전송한다. 대상 노드의 네트워크 디바이스 구동 루틴은 수신한 패킷을 적절한 형태로 재구성한 다음, 그 노드에 있는 NRH로 전달해 준다. NRH에서는 재구성된 패킷의 내용을 검사한 후, 그 노드 내의 해당 프로세스와 관련된 페이지 테이블을 검사하여 원하는 페이지를 메모리 내의 특정 영역 혹은 디스크로 부터 읽어들여 자신의 주소 공간으로 복사한 다음, 이를 근거로 네트워크 페이지 반입 요청에 대한 응답 패킷을 구성하여 이를 다시 네트워크 디바이스 구동 루틴으로 전달한다. 응답 패킷을 대상 노드로 부터 전해 받은 네트워크 디바이스 구동 루틴은 앞에서 언급한 것처럼 수신한 응답 패킷에 대한 재구성 작업을 마친 후, 이를 NRH에 전달하고 NRH는 재구성된 패킷에 포함된 페이지 내용을 페이지 부재 오류 처리 부분에서 할당받은 새로운 페이지에 복사한 후, 페이지 부재 오류로 인해 실행이 일시 정지된 응용 프로세스와 관련된 프로세스 테이블 엔트리의 사건 플래그(event flag)에 '네트워크 페이지 반입 완료'를 표시한다. 한편, 일시 정지 상태에 있던 응용 프로세스는 기다리던 사건, 즉 '네트워크 페이지 반입 완료'가 발생함에 따라 그 프로세스가 진입해 있던 페이지 부재 오류 처리 부분의 특정한 지점, 다시 말해서 수면을 위한 시스템 호출(sleep system call)로 부터 복귀하게 된다. 그리고 나서 해당 프로세스와 관련된 페이지 테이블의 내용들을 조정한 다음 페이지 부재 오류 처리 부분으로 부터 복귀하여 다시 실행이 가능한 상태로 된다.

(4) 페이지 부재 오류 처리 부분의 설계

페이지 부재 오류 처리 루틴은 페이지 부재 오류로 인하여 발생하는 소프트웨어 인터럽트에 의해 구동되는 것으로서, 페이지 부재 오류를 일으킨 프로세스의 컨텍스트(context) 내부에서 인터럽트 처리 루틴의 형식으로 실행된다. 이 루틴은 네트워크 요구 페이징을 지원하는 부분을 포함하고 있는데, 특히 네트워크 상의 두 노

드, 즉 HWP가 존재하고 있는 노드와 그 HWP가 생성한 LWP가 이전해 간 노드가 특정한 페이지를 공유하게 되는 경우 어느 한 쪽에서만 그 페이지에 대한 참조나 수정을 가능케하여 페이지의 내용에 대한 일관성 문제를 해결하도록 하고 있다. 즉 다른 노드로 이전해 간 페이지에 대한 참조나 수정 요구가 발생하였을 때마다 그 노드에 대하여 네트워크 페이지 반입 요청을 하게 되는데, 이 때 페이지를 요구받고 이를 전송해 준 상대방 노드도 그 페이지를 다시 사용해야 할 경우 페이지 부재 오류 처리 부분을 통하여 네트워크 페이지 반입 요청을 하여야 한다. 이를 위하여 페이지 부재 오류 처리 루틴과 이 밖의 관련 루틴들은 페이지 부재 오류를 일으킨 프로세스의 성격 및 페이지 테이블 엔트리 내의 mflag 등의 정보를 이용한다.

페이지 부재 오류 처리 루틴의 가상 코드는 아래와 같다.

page_fault_handler (vaddr)

입력 : 페이지 부재 오류가 발생한 가상 주소 /* vaddr */
출력 : 없음
{

```
vaddr에 대응하는 메모리 관련 자료구조( 가상 영역 테이블, 페이지 테이블,
디스크 블럭 지시자 등 )의 엔트리를 찾고 해당 가상 영역을 lock;

if (vaddr이 접근이 허용되지 않는 가상 주소 공간에 위치) {
        segmentation violation 신호를 프로세스에 보냄;
        레이블 out으로 분기;
}

if (vaddr이 유효) /* 프로세스가 윗 부분에서 수면 상태에 있었을 수 있음 */
        레이블 out으로 분기;

if (mflag가 세트) {
        if (프로세스가 HWP)
```

```
                HWP에 의해 생성된 LWP가 이전해 간 노드의 주소와 그 노드
                내에서의 LWP 식별자 등을 알아낸 다음, 이를 vaddr과 함께
                network request handler에 전달;
        else if (프로세스가 LWP)
                LWP를 생성한 HWP가 있는 노드의 주소와 그 노드 내에서의
                HWP 식별자 등을 알아낸 다음, 이를 vaddr과 함께 network
                request handler에 전달;
        sleep(네트워크 페이지 반입이 완료되는 사건);
        레이블 fin으로 분기;
}

if (페이지가 캐쉬에 있음) {
        페이지를 캐쉬로 부터 제거;
        페이지 테이블 엔트리의 내용을 조정;
        /* 다른 프로세스가 vaddr에 대하여 먼저 페이지 부재 오류를 발생
           시켰을 수 있음 */
        while (페이지 내용이 유효)
                sleep(페이지 내용이 유효하게 되는 사건);
} else { /* 페이지가 캐쉬에 없음 */
        새로운 페이지를 가상 영역에 할당;
        새로운 페이지를 캐쉬에 넣고 페이지 프레임 테이블의 엔트리를 갱신;
        if (페이지가 이전에 메모리에 없고 'demand zero'로 표시) {
                할당된 페이지를 0으로 초기화;
        } else {
                페이지를 swap 디바이스나 실행 화일로 부터 읽어들임;
                sleep(I/O가 완료되는 사건);
        }
        페이지 내용이 유효해지는 사건을 기다리는 프로세스들을 깨움;
}
```

```
fin :   페이지 유효 비트를 세트;
        페이지 수정 비트, 나이 등을 초기화;
        프로세스의 우선 순위를 다시 계산;

out :   가상 영역을 unlock;
}
```

6.2.3. 분산 프로세스간 통신

자원 공용 분산 운영체제는 최소 커널의 한 부분으로서, 서로 다른 노드상의 프로세스들간의 통신을 제공하는 분산 IPC를 지원한다. 본 분산 IPC는 키의 지명 범위를 확장하여 네트워크상의 각 노드의 커널에 존재하는 메시지 큐를 지정할 수 있도록 하고, 서로 다른 노드상의 프로세스간 통신을 지원하기 위하여 원격 메시지 큐에 대한 네트워크 주소와 관련 정보를 지역적으로 유지하는 "채널"이라는 대상을 도입하고, 수신자 메시지 큐에 가장 최근에 전달받은 메시지의 송신자에 대한 통신 주소를 유지함으로써 응답 기능을 효과적으로 지원하며, 공동의 목적을 위하여 상호 협동하는 분산된 프로세스들의 논리적인 그룹을 위한 그룹 통신을 지원하는 특성을 가지고 있다.

(1) 사용자 인터페이스

여기서는 분산 IPC 기능이 원격 메시지 전달 및 그룹 통신을 지원하기 위하여, 추가된 시스템 호출 내용을 기술하고 있다.

a. 메시지 큐의 생성 및 접근 - msgget (key, flag)

사용자가 msgget 시스템 호출시 flag에 IPC_GLOBAL과 IPC_CREAT를 지정하면 전체 분산 시스템 내에서 유일한 키를 가진 메시지 큐가 생성된다. 시스템은 키의 유일성을 보장하기 위하여 네트워크를 통한 전역 검색을 실시하게 된다. 키의 유일성이 확인되면 그 키에 대한 메시지 큐가 생성되고 그에 대한 메시지 큐 식별자가 사용자에게 전달된다. 사용자는 이 메시지 큐 식별자를 사용하여 큐로부터 메시지를 수신할 수 있게 된다.

Msgget 시스템 호출시 IPC_CREAT를 지정하지 않으면 전체 네트워크 상에서 그 키를 가진 메시지 큐를 찾아 그에 대한 채널을 생성하고, 이 채널에 대한 메시지 큐 식별자가 사용자에게 전달된다. 사용자는 이 메시지 큐 식별자를 사용하여 원격 메시지 큐에 메시지를 전송할 수 있게 된다.

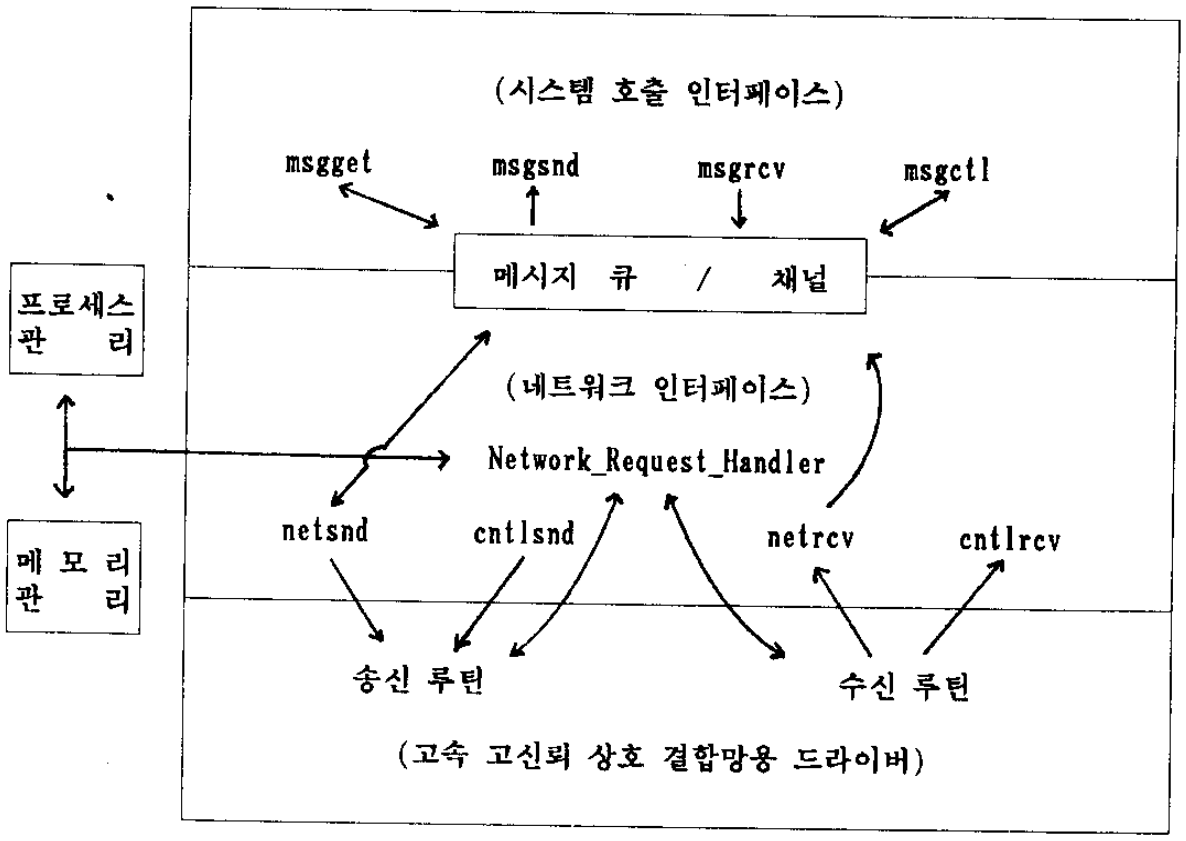


그림 6.5 분산 IPC내의 주요 자료 구조와 주요 루틴의 구성

Flag를 IPC_GROUP 또는 IPC_JOIN으로 지정하면 그룹을 생성하거나 그룹의 구성원(member)으로 가입되게 된다.

b. 메시지의 송신 - msgsnd (qid, msg, size, flag)

Qid가 지정하는 채널이 가리키고 있는 원격 메시지 큐에 대하여 메시지를 전송한다. Flag가 IPC_REPLY로 지정되면 가장 최근에 읽혀진 메시지의 송신자에게 응답 메시지를 전송한다. Flag를 IPC_GROUP으로 지정하면 qid가 지정하는 큐 또는 채널의 키와 같은 키를 갖는 시스템 내의 모든 메시지 큐에 메시지가 다중 전송(multicating)된다.

c. 메시지의 수신 - msgrcv (qid, msg, size, type, flag)

기존의 시스템 호출과 사용자 인터페이스(user interface)는 변함이 없다. 다만 응답을 위해 가장 최근에 수신한 메시지의 호스트 식별자와 큐 식별자가 메시지 큐에 기록됨으로써 응답을 용이하게 한다.

d. IPC 시스템의 제어 - msgctl (qid, cmd, statbuf)

Cmd가 IPC_NOTIFY 또는 IPC_SYSDOWN이면, 모든 노드에 로컬 노드가 정상적으로 작동되고 있다는 것 또는 곧 작동을 끝낼 것이라는 것을 각각 알려준다. Cmd가 IPC_RMID인 경우, 그 큐 또는 채널이 그룹에 속해 있다면, 그 그룹내 모든 노드에 큐 또는 채널의 제거 사실을 알려준다. Cmd가 IPC_SETADDR이면 사용자 공간으로부터 통신하고 있는 원격 노드 식별자와 원격 큐 식별자를 복사해 온다.

(2) 주요 자료 구조 및 주요 루틴

본 분산 IPC내의 주요 자료 구조와 주요 루틴의 구성은 그림 6.5와 같다.

분산 IPC의 주요 자료 구조로는 메시지 큐와 채널을 들 수 있다. 기존의 메시지 큐 구조로 채널까지 표현하고, 플래그를 통해 메시지 큐와 채널을 구별한다. 한편 하나의 그룹을 형성하고 있는 프로세스들의 메시지 큐와 채널은 그림 6.6과 {이 리스트 구조를 형성하고 있다. 그룹에 최초로 가입한 프로세스에게는 메시지 큐가 할당되고, 그 메시지 큐에는 그룹의 유지를 위한 정보가 저장된다. 만약 메시지 큐를 할당받은 프로세스가 그룹을 탈퇴하게 되어 해당 메시지 큐를 폐쇄해야 한다면, 메시지 큐 바로 다음에 연결된 채널(그림 6.6의 경우 채널1)이 메시지 큐로 변환되며, 그룹 유지 정보도 그 곳으로 옮겨진다. 이와 같이 메시지 큐의 폐쇄시 그룹 유지 정보를 효과적으로 유지하기 위하여, 메시지 큐와 채널이 리스트로 연결되어 있다.

분산 IPC의 주요 루틴은 사용자 인터페이스를 제공하는 시스템 호출 인터페이스 루틴과 그 밖의 네트워크 인터페이스를 담당하는 네트워크 인터페이스 루틴으로 구별된다. 네트워크 수신 루틴은 인터럽트 구동 방식으로 동작한다. 네트워크 요구 처리 루틴(NRH)은 노드 간 프로세스 및 메모리 관리와 관련되어 외부로 부터 들어오는 요청을 네트워크 디바이스 구동 루틴을 통하여 전달받아서 그 요청의 형태에 따라 적절한 작업을 수행하거나, 외부로 전송하고자 하는 요청을 네트워크 디바이스 구동 루틴으로 전달하는 커널 내부의 한 루틴이다.

다음은 시스템 호출 인터페이스 루틴에 대한 의사 코드를 기술한 것이다. (편의상 주요 내용과 거리가 먼 "에러 처리 사항"은 생략한다.) 이들은 그림 6.7과 같은 자료 구조를 대상으로 한다. "큐 헤더"는 메시지 큐와 채널을 나타내는 자료 구조로서, "큐 헤더 배열"의 한 원소이다. 각 큐 헤더(즉 메시지 큐 혹은 채널)에 전달된 메시지는, 커널 내부적으로 메시지 헤더와 메시지 헤더가 포인트하는 데이타 영역의 일부로 구성되며, 메시지 헤더들은 리스트 형태로 큐 헤더에 연결된다.

a. msgget (key, flag)

입력 : 키값
　　플래그

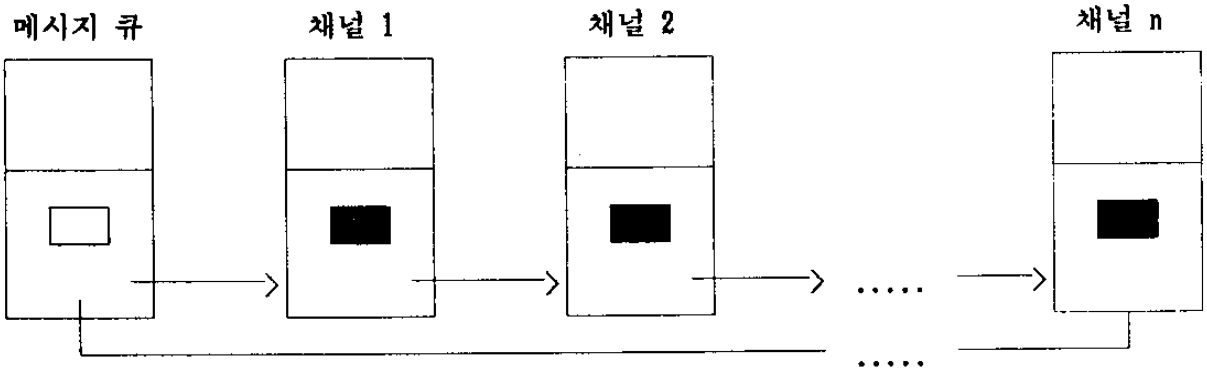


그림 6.6 그룹 유지를 위한 자료 구조

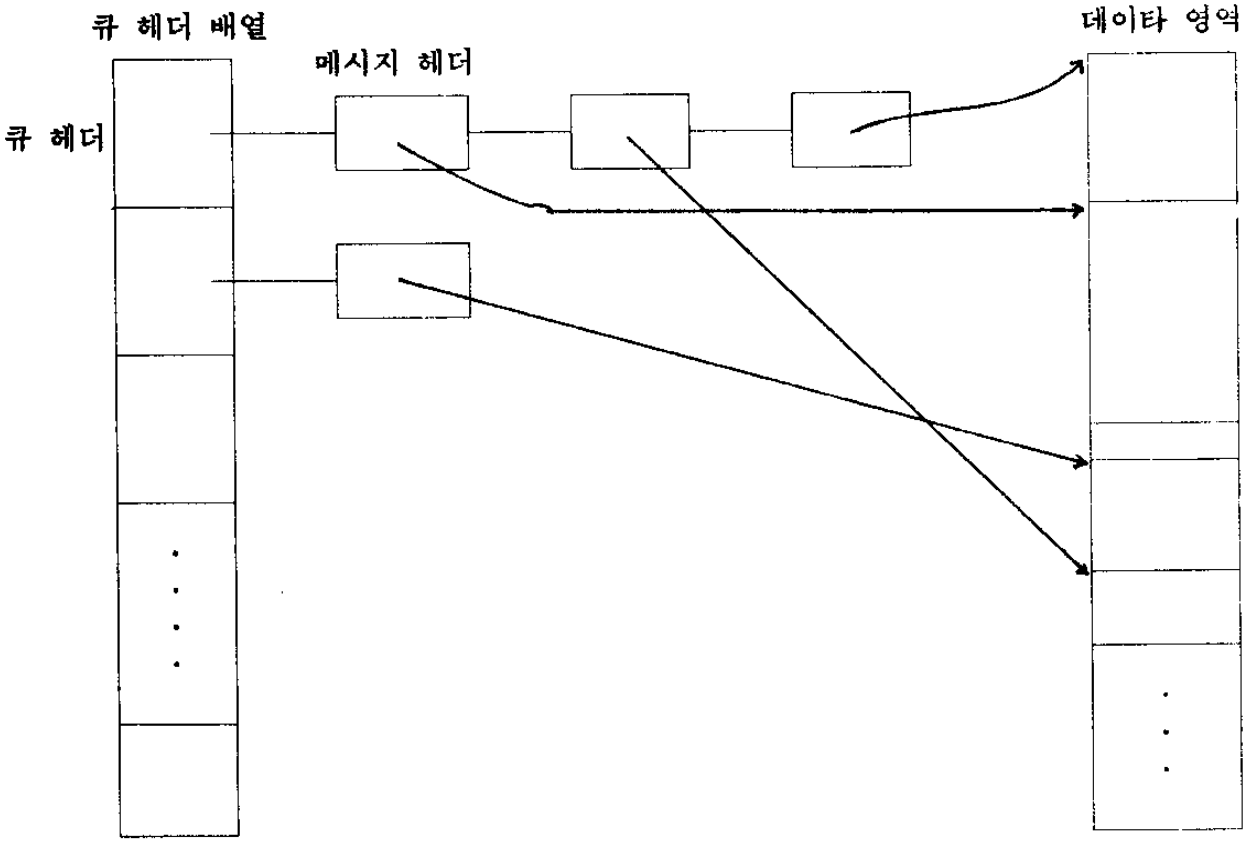


그림 6.7 메시지 전달을 위한 자료 구조

출력 : 큐 식별자

```
{
    if (key == IPC_PRIVATE)
    {
        큐 헤더 배열로 부터 가용 큐 헤더 할당;
        할당된 가용 큐 헤더를 메시지 큐로서 초기화;
        return (해당 메시지 큐의 식별자)
    }

    if (flag의 IPC_GLOBAL, IPC_GROUP 또는 IPC_JOIN가 세트)
        key를 키값으로 하는 기할당 채널 또는 기할당 메시지 큐를 로컬 노드에서
        검색;
    else
        key를 키값으로 하는 기할당 메시지 큐를 로컬 노드에서 검색;
    if (해당 메시지 큐 혹은 채널 존재)
    {
        if (flag의 IPC_CREAT와 IPC_EXCL이 세트)
            return (에러 코드);
        return (해당 메시지 큐 혹은 채널의 식별자);
    }

    if (flag의 IPC_GLOBAL, IPC_GROUP 또는 IPC_JOIN가 세트)
    {
        key를 키값으로 하는 기할당 메시지 큐를 원격 노드에서 검색;
        if (해당 메시지 큐 존재)
        {
            if (flag의 IPC_CREAT와 IPC_EXCL이 세트)
                return (에러 코드);
            큐 헤더 배열로 부터 가용 큐 헤더 할당;
```

```
            할당된 가용 큐 헤더를 채널로서 초기화;
            원격 메시지 큐로 부터 필요한 정보(access permission 등)를 채널로
            복사;
            if (flag의 IPC_GROUP 또는 IPC_JOIN이 세트)
                초기화된 채널을 그룹 리스트에 삽입;
            return (해당 채널의 식별자);
        }
        else
        {
            if (!(flag의 IPC_CREAT가 세트))
                return (에러 코드);
            큐 헤더 배열로 부터 가용 큐 헤더 할당;
            할당된 가용 큐 헤더를 메시지 큐로서 초기화;
            if (flag의 IPC_GROUP 또는 IPC_JOIN이 세트)
                해당 메시지 큐에 그룹 유지 정보를 초기화;
            return (해당 메시지 큐의 식별자);
        }
    }
    else /* 로컬 IPC를 위한 메시지 큐 생성 */
    {
        if (!(flag의 IPC_CREAT가 세트))
            return (에러 코드);
        큐 헤더 배열로 부터 가용 큐 헤더 할당;
        할당된 가용 큐 헤더를 메시지 큐로서 초기화;
        return (해당 메시지 큐의 식별자);
    }
}
```

b. msgsnd (qid, msgp, size, flag)

입력 : 큐 식별자
메시지 포인터
메시지 크기
플래그
출력 : 없음

```
{
    식별자의 타당성 및 엑세스 권한 검사;
    while (메시지를 저장할 공간의 불충분)
    {
        if (flag의 IPC_NOWAIT가 세트)
            return (에러 코드);
        sleep (공간이 반환될 때까지);
    }
    메시지 내용을 사용자 공간에서 커널 공간으로 복사;

    if (flag의 IPC_REPLY가 세트)
    {
        가장 최근의 송신자(즉 클라이언트)의 주소를 획득;
        if (클라이언트가 원격 노드에 존재)
        {
            netsnd를 통해 클라이언트가 존재하는 원격 노드에 메시지 전송;
            return (결과 코드);
        }
    }

    else if (flag의 IPC_GROUP가 세트)
    {
```

```
        netsnd를 통해 그룹에 속한 프로세스가 존재하는 원격 노드에 메시지를
        다중 전송;      /* 이때 메시지 큐에 있는 그룹 식별자를 함께 전달 */
        return (결과 코드);
    }

    else if (qid 가 지명하는 큐 헤더가 채널)
    {
        채널로 부터 원격 메시지 큐의 주소 획득;
        netsnd를 통해 원격 노드에 메시지 전송;
        return (결과 코드);
    }

    else
    {
        가용 메시지 헤더 할당;
        메시지 헤더 삽입 및 기타 자료 구조 내용 변경;
        메시지 수신을 위해 sleep 중인 프로세스들을 wake up;
    }
```

c. msgrcv (id, msgp, size, type, flag)

입력 : 큐 식별자
        메시지 포인터
        메시지의 수신할 수 있는 최대 크기
        메시지 타입
        플래그
출력 : 실제 수신한 메시지 크기

메시지 송신이 성공적으로 수행된 경우, 메시지 송신자의 주소를 메시지 큐에 기록하는 것을 제외하면, 기존의 UNIX System V IPC와 동일하다.

d. netsnd (sqid, rhost, rqid, msgp, size, flag)

입력 : 송신측 큐 식별자
수신측 노드 식별자
수신측 큐 식별자
메시지 포인터
메시지 크기
플래그
출력 : 없음

```
{
    (네트워크) 메시지 패킹;
        /* (네트워크) 메시지 포맷 : 그룹 통신 여부를 표시하는 플래그,
                                    송신 프로세스의 노드 식별자,
                                    송신 프로세스의 채널(또는 메시지 큐) 식별자,
                                    수신 프로세스의 메시지 큐(또는 채널) 식별자,
                                    (IPC) 메시지 크기,
                                    (IPC) 메시지 내용.
        */
    if (flag의 IPC_GROUP이 세트)
        메시지를 모든 노드에 브로드캐스팅;
        /* 고속 고신뢰 상호 결합망용 드라이버의 송신 루틴 사용 */
    else
        메시지를 rhost에 전송;
        /* 고속 고신뢰 상호 결합망용 드라이버의 송신 루틴 사용 */
}
```

e. netrcv (netmsgp, size)

```
/* 송신 노드의 netsnd에 의해 전송된 메시지가 고속 고신뢰 상호 결합망을 통해
   수신 노드측에 전달되면, 인터럽트 구동 방식으로 netrcv가 호출된다. */

입력 : 메시지 포인터
       메시지 크기
출력 : 없음

{
    (네트워크) 메시지 언패킹; /* (네트워크) 메시지 포맷 : "netsnd" 참조 */
    if (flag의 IPC_GROUP이 세트)
        for (해당 그룹 식별자를 가진 큐 헤더에 대해)
        {
            가용 메시지 헤더 할당;
            메시지 헤더 삽입 및 기타 자료 구조 내용 변경;
            메시지 수신을 위해 sleep 중인 프로세스들을 wake up;
        }
    else
    {
        /* "수신 프로세스의 메시지 큐 식별자"가 가르키는 큐 헤더에 대해 */
        가용 메시지 헤더 할당;
        메시지 헤더 삽입 및 기타 자료 구조 내용 변경;
        메시지 수신을 위해 sleep 중인 프로세스들을 wake up;
    }
}
```

f. network_request_handler (packet)

입력 : 네트워크 디바이스 구동 루틴이나 프로세스 관리 부분 혹은 페이지 부재 오류 처리 루틴으로 부터 전달 받은 패킷

출력 : 없음

{

switch (패킷의 type) {

/* 프로세스 관련 처리 부분 */

case 원격 생성될 프로세스가 이전해 간 노드에 보내는 상태 정보 :
case 프로세스를 원격 생성한 후 원래의 노드로 보내는 응답 패킷 :
case 프로세스의 원격 실행 후 이를 요청한 원래의 노드로 보내는 exit 패킷 :
case 원격 실행 중인 프로세스가 있는 노드로 보내는 suspend 패킷 :
case 원격 실행을 위해 옮겨간 프로세스가 있는 노드로 보내는 resume 패킷 :
case 원격 실행 중인 프로세스가 있는 노드로 보내는 kill 패킷 :
    구성된 패킷을 네트워크 디바이스 구동 루틴으로 전달;
    break;

case 원격 생성할 프로세스와 관련되어 원래의 노드로 부터 받은 상태 정보 :
case 다른 노드에서 프로세스를 원격 생성한 후, 이와 관련되어 받은 응답 패킷 :
case 프로세스가 원격 실행된 노드로 부터 받은 exit 패킷 :
case 프로세스의 원격 실행을 요청한 원래의 노드로 부터 받은 suspend 패킷 :
case 프로세스의 원격 실행을 요청한 원래의 노드로 부터 받은 suspend 패킷 :
case 프로세스의 원격 실행을 요청한 원래의 노드로 부터 받은 kill 패킷 :
    수신한 패킷에 따라 상위 레벨의 적절한 프로세스 관리 부분을 호출;
    break;

/* 메모리 관련 처리 부분 */

case 원격 실행을 위해 이전해 온 프로세스가 페이지 부재 오류를 발생
  시켰을 때 이에 대한 커널 내부의 처리 부분에서 구성한 패킷 :
    페이지 부재 오류 처리 부분에서 준비한 정보로 요청 패킷 구성;
    요청 패킷을 네트워크 디바이스 구동 루틴에 전달;

```
        break;

case 다른 노드로 이전해 간 프로세스로 부터 받은, 특정 페이지에 대한
     요청을 포함하고 있는 패킷 :
        해당 프로세스와 관련된 페이지 테이블을 검사하여 원하는
        페이지를 메모리 내의 영역 혹은 디스크로 부터 읽어들임;
        네트워크 페이지 반입 요청에 대한 응답 패킷 구성;
        페이지 테이블 엔트리의 mflag를 세트;
        요청 패킷을 네트워크 디바이스 구동 루틴에 전달;
        break;

case 원격 실행을 위해 이전해 온 프로세스가 원래의 노드에 요청하여 전송
     받은 패킷 :
        응답 패킷에 포함된 페이지 내용을 페이지 부재 오류 처리
        부분에서 할당 받은 새로운 페이지에 복사;
        페이지 테이블 엔트리의 mflag를 리세트;
        페이지 부재 오류로 인하여 실행이 일시 정지된 프로세스와
        관련된 프로세스 테이블 엔트리의 사건 플래그를 세트;
        /* 네트워크 페이지 반입 완료라는 사건에 대하여 수면 상태에
           있는 모든 프로세스를 깨움 */
        wakeup(네트워크 페이지 반입이 완료되는 사건);
        break;

default         :
        메시지 type이 무효;
        적절한 오류 처리;
}
```

6.3. 고속 분산 실행 시스템

고속 분산 실행 시스템(Cross Architecture Procedure EXecution System : 이후 CAPEX 시스템)은, 프로그래밍 기법이 서로 다른 워크스테이션과 전용 컴퓨터상에서, 하나의 분산 프로그램을 효과적으로 작성하고 번역하며, 효율적으로 실행시킬 수 있는 프로그래밍 환경을 지원한다. 즉 CAPEX 시스템은 사용자에게 모듈 인터페이스 기술 언어, 스터브 생성자 및 실행 시간 서버를 지원함으로써, 사용자가 워크스테이션과 전용 컴퓨터에서 실행될 (하나의 분산 프로그램의) 각 모듈이 가지게 되는 프로그래밍 기법상의 차이를 극복할 수 있도록 하였다.

분산 실행 시스템은, 분산 실행되는 모듈 사이의 인터페이스를 기술하고, 이 인터페이스 기술로 부터 스터브 모듈을 생성하는 기능과, 실행 시간시 분산 실행의 각 기능을 담당하는 각종 서버가 제공되어야 한다. CAPEX 시스템은 CAPEX 프로그래밍 시스템과 CAPEX 실행 시간 시스템에서 이를 지원하고 있다.

6.3.1. CAPEX 프로그래밍 시스템

앞서 밝힌 바와 같이, 고속 분산 처리 시스템의 워크스테이션과 전용 컴퓨터상에서 수행되는 분산 프로그램의 모듈은 수행될 컴퓨터의 종류에 따라 각각 서로 다른 언어로 작성되어야 한다. 프로그래머는 CAPEX 프로그래밍 시스템을 사용함으로써, 서로 다른 언어로 작성된 모듈 사이의 인터페이스 차이를 효과적으로 극복할 수 있다.

고속 분산 처리 시스템의 목적에 맞추어, 다중 언어 프로그래밍(multi-language programming)을 간편하고 효율적으로 지원하기 위하여, 모듈 인터페이스에 다음과 같은 제약 조건을 두었다. (기술의 편의상, C 언어로 작성된 모듈을 'C 모듈', 전용 언어로 작성된 모듈을 '전용 언어 모듈'이라고 호칭하겠다.)

> 첫째, CAPEX 프로그래밍 시스템을 통한 C 모듈과 전용 언어 모듈 사이의 호출은 C 모듈에서 전용 언어 모듈로의 단방향으로 제한된다. 단 전용 언어내에서 C 코드의 삽입을 허용하므로, 전용 언어 모듈에서 C 모듈의

호출이 가능하다. 그러나 이는 전용 언어에서 제공되는 기능이지, CAPEX 프로그래밍 시스템에서 제공되는 기능은 아니다.

둘째, C 모듈과 전용 언어 모듈의 모듈 인터페이스는 C 언어의 신택스 및 시맨틱스를 기준으로 한다. 모듈 인터페이스에는 모듈과 인자의 명칭, 인자의 수형을 들 수 있다.

세째, 전형적인 원격 프로시듀어 호출 시스템에서와 같이, 분산 실행되는 모듈 사이에는 공유되는 주소 공간이 제공되지 않는다. 즉 분산 실행되는 모듈 사이의 유일한 정보 교환 방법은 인자 및 결과의 교환이다.

CAPEX 프로그래밍 시스템의 가장 중요한 설계 원칙은 C 언어 컴파일러와 전용 언어 컴파일러를 전혀 수정하지 않고 다중 언어 프로그래밍을 지원하는 것이다. 따라서 CAPEX 프로그래밍 시스템은 C 언어와 전용 언어 이외의 언어들에 대한 다중 언어 프로그래밍의 지원에 있어서도 일반적인 모델이 될 수 있을 것이다.

CAPEX 프로그래밍 시스템과 각 컴파일러를 사용한 분산 프로그램의 번역 과정을 그림 6.8에 나타냈다. 그림 6.8에서 *로 표시된 MIDL(Module Interface Description Language)와 CSG(CAPEX Stub Generator)가 CAPEX 프로그래밍 시스템을 구성한다. 사용자는 굵은 선의 직사각형으로 표시된 내용, 즉 모듈 인터페이스 기술, C 언어 모듈들과 전용 언어 모듈들을 작성한다. 이후 사용자는 모듈 인터페이스 기술을 CSG를 통해 번역하여 C 코드의 스터브 모듈을 얻고, C 언어 모듈과 스터브 모듈을 C 컴파일러를 통해 번역하여 워크스테이션에서 수행될 목적 코드를 얻는다. 한편 전용 컴퓨터에서 수행될 목적 코드는, 전용 언어로 작성된 모듈들이 전용 언어 컴파일러에 의해 번역된 것이다.

(1) MIDL (Module Interface Description Language)

MIDL은 워크스테이션과 전용 컴퓨터상에서 분산 실행되는 모듈들 사이의 인터페이스, 즉 외부 모듈의 정의, 그 모듈의 인자와 결과의 수형 그리고 외부 모듈이 실행되는 전용 컴퓨터의 유형을 기술하는 언어이다. 사용자가 MIDL을 사용하여 작

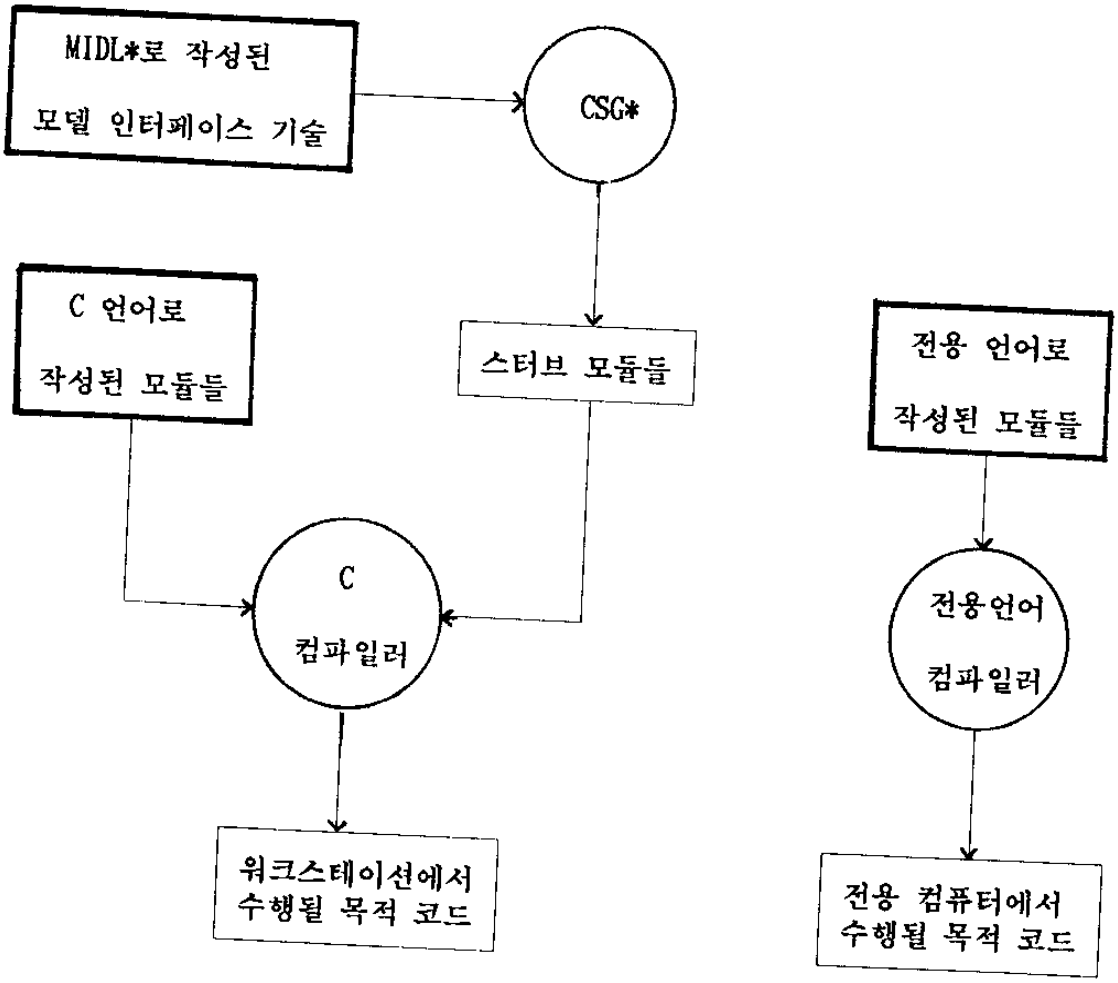


그림 6.8 분산 프로그램의 번역 과정

성한 모듈 인터페이스 기술은 CSG에 의해 번역되어, 각 외부 모듈에 대한 스터브 모듈이 생성된다. 스터브 모듈은 인자들을 패킹하여 "분산 실행 요청" 메시지를 생성하여, 이를 CAPEX 클라이언트에 전달하고, CAPEX 클라이언트로 부터 결과를 전달받아 이를 언패킹하여 원래의 모듈로 귀환한다.

워크스테이션상에서 실행되는 메인 모듈 및 기타 모듈은 C 언어로 작성되고, 전용 컴퓨터상에서 실행되는 모듈은 각 전용 언어로 작성된다. 앞에서 밝힌 바와 같이 C 언어로 기술된 모듈과 전용 언어로 기술된 모듈과의 모듈 인터페이스는 C 언어의 신택스와 시맨틱스를 기준으로 한다. 따라서 MIDL의 신택스는 대부분 C 언어의 신택스를 그대로 유지하였다. 다음은 MIDL의 구문을 표시한 것이다.

```
<midl>              ::= <type_defs> <modules>
<type_defs>         ::= <type_def> <type_defs> | <type_def>
<type_def>          ::= type_def <type> <ident> ;
<modules>           ::= <module> <modules> | <module>
<module>            ::= <type_of_computer> <file_name>
                                          <ret> <ident> ( <arguments> ) ;
<type_of_computer>  ::= MP | HDAC
<file_name>         ::= <fstring>
<fstring>           ::= <fchar> <fstring> | <fchar>
<fchar>             ::= <alphanume> | .
<ret>               ::= <type> |
<arguments>         ::= <argu> <arguments> |
<type>              ::= char | short | int | long | unsigned | float | double
                        | <array> | <struct_or_union>
<array>             ::= <type> <dimension>
<dimension>         ::= [ <const> ]
<struct_or_union>   ::= struct { <struct_decl_list> }
                        | struct <ident> { <struct_decl_list> }
```

```
                    | struct <ident>
                    | union { <struct_decl_list> }
                    | union <ident> { <struct_decl_list> }
                    | union <ident>
<struct_decl_list> ::= <struct_decl> <struct_decl_list> | <struct_decl>
<struct_decl>      ::= <type> ;
<ident>            ::= <apha> <aphanume>
<aphanume>         ::= <apha> <aphanume> | <digit> <aphanume> | _ <aphanume> |
<const>            ::= <digit> | <digit> <const>
<apha>             ::= a - z | A - Z
<digit>            ::= 0 - 9
```

(2) CSG (CAPEX Stub Generator)

CSG는 MIDL로 작성된 모듈 인터페이스 기술을 입력으로 받아서, 이를 번역하여 스터브 모듈을 생성하는 스터브 모듈 생성자이다. 스터브 모듈은 인자들을 패킹하여 "분산 실행 요청" 메시지를 생성하여, 이를 CAPEX 클라이언트에 전달하고, CAPEX 클라이언트로 부터 결과를 전달받아 이를 언패킹하여 원래의 모듈로 귀환한다. 그림 6.9의 (a)는 사용자가 MIDL로 작성한 모듈 인터페이스 기술이고, (b)는 CSG가 생성한 스터브 모듈이다.

```
MP "mpvector" int inner_product (int[100], int[100]);
```

(a) 모듈 인터페이스 기술

```
#include      <capex.h>

struct msgform {
```

```
        long    mtype;
        char    mtext[10000];
} cxmsg;
char  *cxmsgp;

            define MYKEY 5833
            define MACH MP    /* MP - Math Package 전용 컴퓨터 */
            define FNAME "mpvector"
            define MNAME "inner_product"

inner_product (cxarg1, cxarg2)
        int     cxarg1[], cxarg2[];
{
        int     cx1, cxaqid, cxcqid;

        /* "분산 실행 요청" 메시지를 생성한다 */
        cxmsgp = cxmsg.mtext;
        cxpack_int (CALL);        /* 명령 코드 패킹 : CALL - 분산 실행 요청 */
        cxaqid = msgget(MYKEY, 0777);
        cxpack_int (cxaqid);     /* 사용자 프로세스 큐 식별자 패킹 */
        cxpack_int (MACH);          /* 전용 컴퓨터 유형 패킹
        cxpack_str (FNAME);        /* 적재 화일명 패킹 */
        cxpack_str (MNAME);      /* 호출 모듈명 패킹 */
        for (cx1 = 0; cx1 < 100; cx1++)   /* 첫번째 인자 패킹 */
                cxpack_int (cxarg1[cx1]);
        for (cx1 = 0; cx1 < 100; cx1++)   /* 두번째 인자 패킹 */
                cxpack_int (cxarg2[cx1]);

        /* "분산 실행 요청" 메시지를 CAPEX 클라이언트에 전달한다 */
        cxcqid = msgget (CAPEX, 0777);
```

```
        cxmsg.mtype = 1;
        msgsnd (cxqid,

        /* "호출 결과" 메시지를 CAPEX 클라이언트로 부터 받는다 */
        msgrcv (cxaqid, cxmsg, MAXMSGSIZE, cxqid, 0);

        /* "호출 결과" 메시지를 언패킹하여 결과값을 추출하고 귀환한다 */
        return (cxunpack_int (cxmsg.mtext));
}
```

(b) 스터브 모듈

그림 6.9 CSG의 사용예

그림 6.9에서 몇가지 사항에 주목할 필요가 있다. 우선 스터브 모듈 내부에서 유지되는 변수 및 함수의 명칭에는 항상 "cx"가 붙는다. 이는 사용자가 작성한 부분의 전역 변수 및 함수의 명칭과의 중복을 될 수 있는 한 피하기 위한 것이다. 그리고 cxpack_int, cxpack_str과 cxunpack_int 등의 루틴은 라이브러리로서 존재한다. MP, CAPEX 그리고 MAXMSGSIZE 등은 이미 <capex.h>에 정의되어 있다.

6.3.2. CAPEX 실행 시간 시스템

CAPEX 실행 시간 시스템은 실행 시간에 워크스테이션과 전용 컴퓨터 사이의 분산 실행을 지원하는 서버들, 즉 CAPEX 클라이언트와 CAPEX 서버로 구성된다. 그림 6.10은 CAPEX 실행 시간 시스템의 구성을 보여주고 있다. 실행중 사용자 프로세스의 스터브 모듈로 부터 CAPEX 클라이언트에 "분산 실행 요청" 메시지(일종의 호출 패킷)가 전달되면, CAPEX 클라이언트는 실행 코드를 "분산 실행 요청" 메시지에 첨가하여, 실행 초기에 할당된 전용 컴퓨터의 전단 컴퓨터의 CAPEX

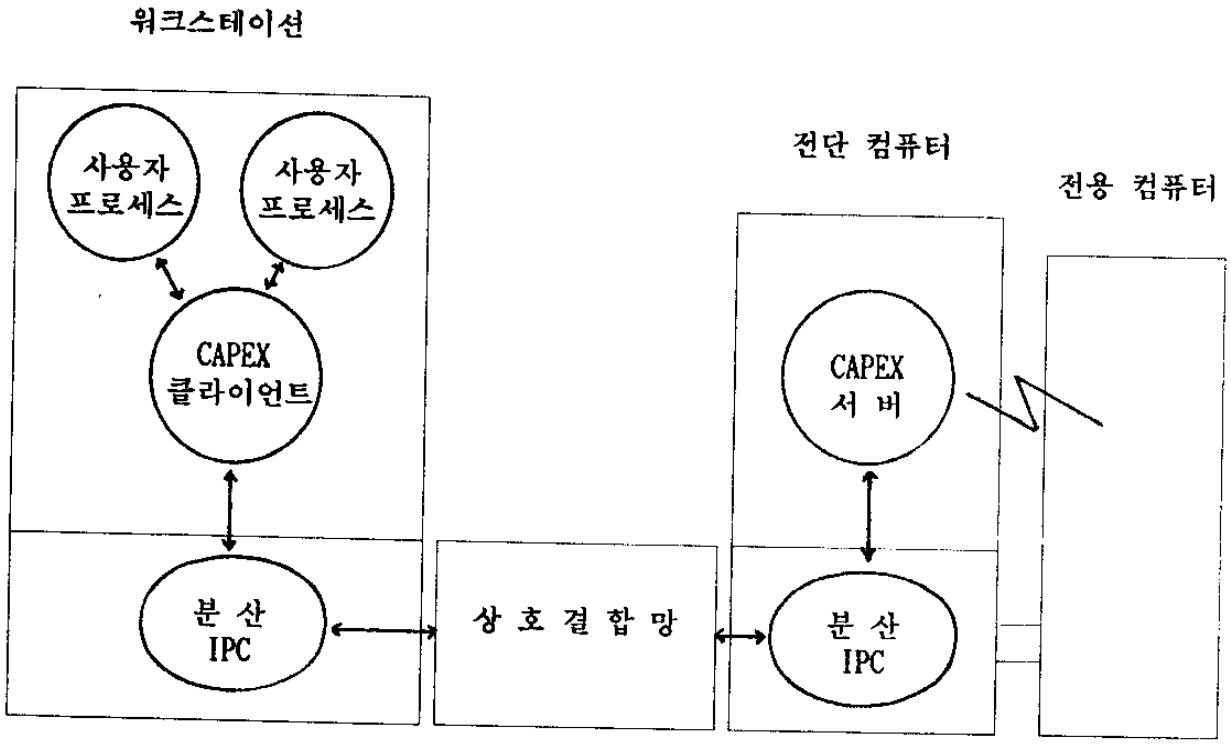


그림 6.10 CAPEX 실행 시간 시스템의 구성

서버에 전달한다. CAPEX 클라이언트로 부터 "분산 실행 요청" 메시지를 전달받은 CAPEX 서버는 적재 코드, 모듈명 그리고 인자 등을 전용 컴퓨터에 전달하고, 실행 개시를 지시한다. 전용 컴퓨터의 실행 결과는 CAPEX 서버와 CAPEX 클라이어트를 거쳐 사용자 프로세스에 전달된다.

(1) 사용자 인터페이스

본 자원 공용 분산 운영체제내의 프로세스들은 자원 공용 분산 운영체제가 제공하는 분산 IPC를 통해 서로 통신한다. 같은 방법으로 사용자 프로세스는 CAPEX 클라이언트에 일정한 양식의 메시지를 전달함으로써 필요한 서비스를 요청하게 되고, CAPEX 클라이언트도 그 결과를 일정한 양식의 메시지로 사용자 프로세스에 통보한다.

그림 6.11는 사용자 프로세스가 CAPEX 클라이언트에 전달하는 서비스 요청 메시지의 신택스를 나타낸 것이다.

```
<서비스 요청 메시지> ::= <서버 할당 요청> | <분산 실행 요청> | <실행 종료 통보>
<서버 할당 요청>     ::= ALLOC  사용자_큐_식별자  전용_컴퓨터_유형
<분산 실행 요청>     ::= CALL  사용자_큐_식별자  전용_컴퓨터_유형  적재_화일명
                         모듈명  인자_리스트
<실행 종료 통보>     ::= EXIT  사용자_큐_식별자  CAPEX_서버_큐_식별자_리스트
```

그림 6.11 서비스 요청 메시지의 신택스

사용자 프로세스의 서버 할당 요청과 실행 종료 요청에 대해서, CAPEX 클라이언트는 단순한 결과 코드만을 사용자 프로세스에 전달한다. 그리고 사용자 프로세스의 분산 실행 요청에 대해서는, CAPEX 클라이언트는 사용자 프로세스에 결과값

(결과값의 수형이 배열이나 레코드인 경우에는 값들의 리스트)이 전달되는데, 이 결과값은 스터브 모듈에 의해 언패킹되어 원래의 모듈로 귀환된다.

(2) 주요 자료 구조 및 주요 루틴

그림 6.12와 6.13은 CAPEX 클라이언트와 CAPEX 서버 사이에서 교환되는 메시지의 신택스이다.

```
<클라이언트-서버 메시지> ::= <분산 실행 요청>
<분산 실행 요청>         ::= CAPEX_클라이언트_큐_식별자  사용자_큐_식별자
                             적재_코드_크기  적재_코드  모듈명  인자_리스트
```

그림 6.12 CAPEX 클라이언트에서 CAPEX 서버로 전달되는 메시지의 신택스

```
<서버-클라이언트 메시지> ::= <분산 실행 회신>
<분산 실행 회신>         ::= RCALL  사용자_큐_식별자 실행_결과
```

그림 6.13 CAPEX 서버에서 CAPEX 클라이언트로 전달되는 메시지의 신택스

a. CAPEX 클라이언트

```
{
    while (1)
        for (사용자 프로세스 및 CAPEX 서버로 부터 전달되는 각 메시지에 대해)
            switch (메시지 유형)
            {
```

```
                case 서버 할당 요청 : /* 사용자 프로세스 */
                    해당 유형의 전용 컴퓨터 할당;
                    사용자 프로세스에게 할당된 전용 컴퓨터의 전단 컴퓨터의
                    CAPEX 서버의 메시지 큐 식별자를 송신;
                    break;
                case 분산 실행 요청 : /* 사용자 프로세스 */
                    사용자 프로세스로 부터 전달된 "분산 실행 요청" 메시지에
                    실행 코드 추가;
                    해당 CAPEX 서버에 "분산 실행 요청" 메시지 송신;
                    break;
                case 분산 실행 회신 : /* CAPEX 서버 */
                    결과값을 사용자 프로세스에 전달;
                    break;
                case 실행 종료 통보 : /* 사용자 프로세스 */
                    "CAPEX_서버_큐_식별자_리스트"에 포함되어 있는 CAPEX
                    서버를 미할당 상태로 전환;
                    break;
            }
}
```

b. CAPEX 서버

```
{
    while (1)
        for (CAPEX 클라이언트로 부터 전달되는 각 메시지에 대해)
        {
            "적재_코드_크기", "적재_코드", "모듈명" 그리고 "인자_리스트"를
            전용 컴퓨터의 메모리의 일정한 위치에 전달;
            전용 컴퓨터에 실행 지시;
```

```
        }
}
```

한편 CAPEX 서버의 지시에 의해 실행을 시작한 전용 컴퓨터는 실행을 끝마친 후 전단 컴퓨터의 CAPEX 서버에 인터럽트를 걸어 실행 결과를 전달한다. 그러면 CAPEX 서버는 결과값을 기록한 "분산 실행 회신" 메시지를 생성하여 CAPEX 클라이언트에 전달한다.

6.4. 자원 공용 분산 운영체제의 구현

자원 공용 분산 운영체제의 주요 임무는, 사용자가 분산 프로그램을 워크스테이션과 전용 컴퓨터상에서 효과적으로 개발하고 수행시킬 수 있도록 하는 것이다. 본 연구에서는 이를 입증하기 위하여, 커널 수준에서 분산 프로세스간 통신(분산 IPC)과 사용자 수준에서 분산 실행 시스템(CAPEX 시스템)을 각각 구현하였다. 구현한 소스 코드는 [조규찬89]에 제시되어 있다.

6.4.1. 구현 환경

고속 분산 처리 시스템을 구성하는, Math Package 전용 컴퓨터, DAC 알고리즘 전용 컴퓨터와 상호 결합망은 실제 하드웨어로 구현되지 않았으므로, IBM AT 호환 기종의 마이크로 컴퓨터들이 Ethernet으로 연결된 분산 시스템을 대상으로, 분산 IPC와 CAPEX 시스템을 구현하였다. 또한 분산 IPC와 CAPEX 시스템은 UNIX System V Version 2.0에서 각각 커널과 사용자 수준으로 구현되었다.

6.4.2. 다른 서브 시스템과의 인터페이스

설계된 자원 공용 분산 운영체제에서는 CAPEX 서버가 전용 컴퓨터의 메모리의 일정한 위치에 수행될 모듈의 코드 및 인자를 전달하고 수행 개시를 지시하며 실행이 끝난 후에는 인터럽트 구동 방식으로 전용 컴퓨터의 메모리의 일정한 위치에서 모듈의 결과값을 전달받게 된다. CAPEX 서버의 구현에서는, 이러한 CAPEX 서버와 전용 컴퓨터 사이의 인터페이스는 CAPEX 서버가 마이크로 컴퓨터상에서 모듈(실제로는 CAPEX 서버와 통신하고 모듈을 구동시키는 역할을 수행하는 메인 루틴이 포함된 하나의 프로그램)을 프로세스화하고, 인자와 결과를 분산 IPC를 통해 메시지로 교환하는 것으로 대치하였다.

전용 컴퓨터에서 수행되는 모듈은 전용 언어로 작성되나, 구현된 CAPEX 시스템에서는 메인 루틴과 같이 C 언어로 작성하였다. 이는 전용 언어에 대한 컴파일러가 구현되지 않았기 때문이다. 그리고 전용 컴퓨터에서 수행될 모듈이 전용 언어

대신 C 언어로 작성되더라도, C 언어와 전용 언어의 모듈 인터페이스는 C 언어의 모듈 인터페이스를 기준으로 하므로, CAPEX 시스템의 설계 내용을 축소하여 구현한 것은 아니다.

자원 공용 분산 운영체제와 상호 결합망과의 인터페이스는 상호 결합망이 자원 공용 분산 운영체제에 제공하는 상호 결합망용 드라이브 루틴으로서, 이는 분산 IPC에 의해 호출된다. 따라서 자원 공용 분산 운영체제와 상호 결합망과의 인터페이스로 Ethernet의 드라이브 루틴을 그대로 사용하였다.

6.4.3. 구현 내용

분산 IPC와 CAPEX 시스템에 대한 간략한 의사 코드는 이미 6.2.3.과 6.3.에 제시되어 있다. 여기서는 분산 IPC와 CAPEX 시스템의 구현에 대한 세부적인 사항을 밝히고자 한다.

UNIX System V IPC의 메시지 교환 방법에서는 "키"를 통해 메시지 큐(혹은 채널)을 지명한다. 분산 IPC에서는 기존의 단일 시스템과 동일한 사용자 인터페이스를 제공하기 위해서, "키"의 지명 범위을 분산 시스템 전체로 확장하였고, "키"를 그룹 식별자로도 사용하였다. 이러한 키는 사용자 수준의 논리적 주소와 시스템 수준의 물리적 주소로 구분하여 생각할 수 있다. 논리적 주소는 4 바이트로 구성되어 있는데 1 비트의 그룹 비트와 나머지의 키 부분으로 구성되어 있다. 이와 같은 논리적 주소는 전체 분산 시스템에 대하여 통신 대상의 존재 위치에 무관한 하나의 전역 통신 주소 공간을 형성하므로, 프로세스는 그 호스트의 위치에 상관없이 서로 통신할 수 있게 된다. 물리적 주소는 각각 2 바이트의 호스트 식별자와 큐 식별자로 구성되어 있다. 논리적 주소의 물리적 주소에 대한 사상은 통신 경로의 설정시 수행되어 시스템에 의하여 유지된다.

CAPEX 프로그래밍 시스템의 구성 요소인 MIDL과 CSG는 UNIX System V의 lex와 yacc를 사용하여 검증 혹은 구현하였다. 스터브 생성자인 CSG는 인자 및 결과의 수형으로 정수, 문자 그리고 배열을 지원하도록 구현되었고, 레코드 수형은 구

현 중이다. 단, 레코드 수형의 경우, MIDL을 사용하여 모듈 인터페이스를 기술할 때, 한 레코드의 필드로서 다른 레코드 수형을 사용하려는 경우, 내부에 삽입되는 레코드 수형은 그 이전에 미리 "typedef"문을 이용하여 정의하여야 한다는 제약을 두었다.

전용 컴퓨터(또는 그의 전단 컴퓨터, 또는 그의 전단 컴퓨터에서 수행되는 CAPEX 서버)의 할당을 위해, 시스템내의 여러 CAPEX 클라이언트 중 하나를 " 전용 컴퓨터 할당 관리자"로 선정하였다. 즉 어느 CAPEX 클라이언트가 전용 컴퓨터를 할당하고자 한다면, 그 CAPEX 클라이언트는 "전용 컴퓨터 할당 관리자"에 할당 요청을 하게 되고, "전용 컴퓨터 할당 관리자"는 당시의 전용 컴퓨터의 할당 상태를 참조하여, 그 요청에 대한 적절한 서비스를 수행한다. 이러한 전용 컴퓨터 할당 방법은 "전용 컴퓨터 할당 관리자"가 존재하는 노드에 부하 지나치게 집중될 위험이 있지만, 시스템 전체의 통신량을 줄일 수 있을 것으로 생각된다.

7. 결론

과거의 중앙 집중형 처리 시스템에 비해 성능, 자원의 공유성, 신뢰도, 확장성, 응용성등이 우수한 분산 처리 시스템을 구성하려는 노력이 최근의 컴퓨터 하드웨어와 고속 통신망의 급속한 발전으로 중소형 컴퓨터를 고속의 통신망을 통해 연결한 형태로 구체화 하고있다. 본 연구에서 제시한 고속 분산 처리 시스템은 이러한 시도의 하나로 제시된 시스템이다. 즉 Mach. Package 프로그램을 고속 병렬 처리 하는 전용 컴퓨터와 Divide and Conquer 형태의 프로그램을 고속 병렬 처리하는 전용 컴퓨터 및 각각의 전용 언어를 개발하고, 이들 전용 컴퓨터들을 계산 서버로 사용하도록 범용 워크스테이션에 고속 고신뢰 상호 결합망을 통해 연결하여, UNIX 호환 분산 운영체제로 전체 시스템을 제어함으로써, 고속 분산 시스템을 구축하였다. 본장에서는 제안된 고속 분산 처리 시스템을 이루는 각 부분들에 대하여 연구 수행 결론 및 앞으로의 연구과제에 대하여 기술한다.

작성된 Math. Package 프로그램을 고속 병렬로 수행하기 위한 전용 컴퓨터로서는 그래프 리덕션 방식의 리덕션 컴퓨터를 제안하였다. 이 시스템은 다수의 프로세서들을 X-Tree 형태의 상호연결망으로 연결하고 있다. X-링크를 통한 데이타 전송은 순수한 Tree 구조가 가지는 단점인 루트부근의 통신 병목현상을 어느 정도 피하게 해준다. 컴파일러가 프로그램을 함수단위로 번역하여 코드블럭들을 생성하면 이들 코드블럭들은 Math. Package 전용 컴퓨터의 여러 범용 프로세서에 저장된 후 상당히 적은 통신부하를 일으키며 병렬수행되어진다. 이는 본 시스템의 범용 프로세서가 생성된 코드블럭들을 기존의 폰노이만 방식으로 수행함으로써 지역참조성의 장점을 이용하기 때문이다. 한편 설계 제안된 전용언어는 medium/coarse granularity를 갖는 함수언어이면서도 프로세서 할당자라는 일종의 자원처리문을 제공하여 프로그래머의 전문지식을 프로그램 수행시에 반영하도록 해주고 있다. 예제 프로그램 n × n (Solving Linear System by Cramer's Rule)에 의하여 성능분석을 한 결과 통신부하의 overhead가 바람직하다는 전제 하에서 프로세서의 갯수에 비례하는 성능향상을 기대할 수 있었다. 이는 점차 발전하고 있는 통신매체에 힘입어 상당히 현실적인 결과로 나타날 것이다. 추후 전용언어의 컴파일러 및 Math. Package의 개발 그리고 설계 제안된 Math. Package 전용 컴퓨터의 구현 등에 관한 연구가 이루어져야 할 것이다.

Divide-And-Conquer (DAC) 형태의 프로그램은 HYPERDAC으로 불리는 전용 컴퓨터에서 수행된다.제안된 HYPERDAC은 Divide-And-Conquer 알고리즘을 효과적으로 수행하면서도 일반적인 다른 알고리즘도 무리없이 수행할 수 있도록 제안된 고속의 다중처리 시스템으로서, DAC 알고리즘의 병렬성을 고도로 이용하기 위하여 여러 수준의 병렬성을 이용할 수 있는 데이타 플로우 모델을 계산 모델로 채택하였다. 그리고 DAC 알고리즘의 논리적 계층 구조를 활용하고 일반적인 다른 알고리즘에의 적용도 용이하도록 Hypertree를 상호결합망으로 채택하였으며, 각 처리요소에 부과된 부하를 균등하게 함으로써 전체 시스템의 이용도와 성능을 올리는 분산 부하 균형 정책으로 Gradient 부하균형 정책을 채택하였다. 제한된 자원하에서의 과도한 병렬성을 억지하고 계산의 지역성을 이용함으로써 시스템 자원의 효율적 사용을 고양하는 throttling 기법을 채택하였다. 본 연구에서는 이러한 특징을 갖는 HYPERDAC 시스템을 실제로 구현하는데 초석이 될 베이스 언어와 프로그래밍 언어를 설계하고, 간단한 성능 분석을 통하여 향후 연구에 대한 문제점들을 추출하였다. 따라서 앞으로는 이러한 연구를 토대로 다음과 같은 연구가 보강되어야 할 것이다.

첫째, 순수한 데이타 구동형 모델의 단점을 제거하고 리덕션 수행 모델 및 폰 노이만 계산 모델의 장점을 취합하는 새로운 통합 병렬 모델에 대한 연구

둘째, 계산단위의 granule과 상호 결합망과의 관계를 고려한, 유연하고 효과적인 granule과 상호 결합망의 설정

세째, 프로그램의 효과적인 분할 및 사상 기법에 관한 병렬 컴파일러에 관한 기법 시스템의 유용도를 효과적으오 향상시킬 수 있는 분산 부하 균형 정책 및

네째, 부하 상태의 결정에 관련된 인자의 규명 및 그들의 관계 설정

다섯째, 제한된 자원 하에서 병렬성을 하드웨어나 소프트웨어 기법을 이용하여 효과적 으로 제어할 수 있는 throttling 기법

여섯째, 시스템의 원형 및 시스템 소프트웨어의 구현 등이다.

범용 컴퓨터와 전단 컴퓨터 간의 통신은 이중으로 구성된 고속 근거리 통신망을 사용한다. 두 개의 통신망이 사용됨에따라 이러한 통신 기능을 수행하는 통신망 전용 제어기와 직렬 인터페이스 장치도 각각 설계되어야 한다. 본 연구에서는 하드웨어로 구현될 통신망 전용제어기와 직렬 인터페이스 장치에서 수행될 주요 기능을

설정하고 그에 대한 블럭 구성도를 연구하였다. 또한 이러한 통신 인터페이스 장치를 제어하고 양단간의 신뢰성있는 통신을 보장하기 위한 통신망 소프트웨어의 기능적 요건을 ISO 표준 프로토콜을 기준으로 정의하였다. 그리고 이러한 통신 인터페이스가 설치된 고속 근거리 통신망을 시뮬레이션함으로써 고속 고신뢰 상호 결합망의 성능을 분석, 평가하였다. 즉, 두 개의 통신망을 사용함으로써 단위 시간당 처리량의 증가는 물론 프레임의 평균 전송 지연 시간이 감소되는 결과를 한 개의 통신망을 사용하는 경우와 비교 분석하였다. 또한 한개의 통신망에서 결함이 발생한 경우에도 나머지 한 개의 통신망으로 통신 수행이 가능하므로 통신망 시스템의 신뢰성을 향상시킬 수 있다. 이와 같이 고속 분산 처리 시스템에서는 고속 근거리 통신망을 이중으로 설치함으로써 컴퓨터 간의 고속 고신뢰 상호 결합망을 구성하였다. 고속 근거리 통신망은 데이타 전송률이 빠르기 때문에 하드웨어 인터페이스 장치도 매우 빠른 처리율을 지원하여야 한다. 따라서 데이타 링크 계층의 CSMA 프로토콜과 함께 물리적 계층의 기능이 대규모 집적 회로로써 구성되어야 한다. 현재 ANSI의 X3T9.5가 표준화된 모델로 제시되고 있지만 앞으로는 동축 케이블뿐 아니라 광섬유를 전송 매체로써 효율적으로 사용하기 위한 고속 근거리 통신망용 표준 프로토콜이 개발되어야 한다. 또한 고속 근거리 통신망은 일반 목적용 네트워크보다는 고속 분산 처리 시스템과 같은 특수 목적용 네트워크에 적용하기 위하여 보다 많은 연구가 수행되어야 할 것이다.

제안된 고속 분산 처리 시스템에서 기존의 워크스테이션은 사용자가 작성한 프로그램을 수행하고 그 결과를 사용자에게 돌려 주는 사용자 인터페이스를 담당하며, 전용 컴퓨터는 계산 서버로 사용된다. 자원 공용 분산 운영체제는 고속 분산 처리 시스템의 각 자원을 통합적으로 운영하는 분산 운영체제이다. 이기종 전용 컴퓨터들을 통합적으로 운영하기 위해 운영체제는 공용 자원에 대한 투명한 사용자 인터페이스를 지원해야 하며, 프로세스의 분산 실행을 효과적으로 지원해야 한다. 본 연구에서는 이러한 구조에 적합하도록 기존의 UNIX 운영체제의 커널을 기반으로 확장한 본 운영체제의 커널은 그 기능을 가급적 최소화시켜 프로세스 관리, 메모리 관리, 프로세스간 통신으로 구성하였고, 그외의 기능은 각종 서버가 제공하도록 하였다. 본 연구의 운영체제는 프로세스 모델을 사용하였다. 분산 실행을 효과적으로 지원하기 위해 기존 UNIX의 프로세스와, 또다른 관련된 상태 정보를 최소화한 프로세스(LWP) 개념을 도입하여 이를 프로세스 관리부에 확장하였다. 분산된 노드간의

프로세스 및 데이타의 이동을 효과적으로 지원할 수 있는 네트워크 범위의 요구 페이징 기법을 메모리 관리부가 제공하도록 하였으며, 분산된 프로세스간의 효율적 통신을 지원하기 위해 채널, 송수신 응답 트랜잭션과 그룹통신의 세가지 개념을 제시하였다. 그리고 사용자 프로그램의 전용 컴퓨터에서의 수행을 위해 CAPEX 클라이언트와 서버로 구성된 분산 실행 시스템을 구현하였다. 본 연구에서 커널은 pseudo 코드로 설계되었지만, 앞으로 완전한 코드로 구현하여 봄으로써 분산 운영체제의 타당성이나, 그 성능을 조사해야 할 것 이다. 또한 화일 서버와 같은 각종 서버들을 구현하여 하나의 완전한 기능을 하는 운영체제를 만들어야 할 것이다.

이상에서와 같이 기술된 연구 결과 및 앞으로의 연구 방향을 토대로 하여 본 연구에서 제시된 고속 분산 처리 시스템을 실제로 구현하는 연구가 추후 이루어져야 할 것이다. 이러한 구현 연구는 고속 분산 처리 시스템 및 병렬 처리 분야에 커다란 발전을 이룩할 것으로 믿는다.

참 고 문 헌

[김길용88] 조규찬, 김길용, 조유근, 분산 시스템을 위한 확장된 **UNIX System V IPC**, *1988* 춘계 학술 발표 논문집.

[이석호84] 이석호, 홍봉희, 한글 질의어 시스템의 설계및 구현, 한국 정보과학회 논문지, *Vol. 11, 1984.*

[조규찬89] 조규찬, 김홍근, 조유근, 이기종간의 분산 실행 시스템, 연구 보고, 서울대학교 컴퓨터공학과, *SNU-CE-SSR-02, 1989.*

[한전87a] 한상영, 전주식 외 6인, 데이타 플로우 컴퓨터에 관한 연구, 최종연구 보고서, 과학기술처, *1987.*

[한전87b] 한상영, 전주식 외 2인, 데이타 플로우 컴퓨터에 관한 연구, *'86* 특정연구 결과 발표회 논문집, *1987, pp. 141-145.*

[ACDE79] Ackerman,W.B. and J.B. Dennis, **VAL - a Value - oriented Algorithmic Language**, *Preliminary Reference Manual, Springer-Verlag, 1983, pp. 165-190.*

[AMON84] M. Amamiya, S. Ono, and N. Takahashi, **Partial Computation with a Data Flow Machine**, *Proc. 5th Conf. on Mathematical Methods in Software Science and Engineering, 1984, pp. 87-113.*

[AMHA83] M. Amamiya, R. Hasegawa, Y. Kiyoki, **Eager and Lazy Evaluation Mechanism in Data Flow Architecture and its application to Parallel Inference Machine**, *Proc. of Work Meeting for Computers, IEEE, Japan, Nov. 1983, pp. 41-51.*

[AMHA84] M. Amamiya, R. Hasegawa, **Dataflow Computing and Eager and Lazy Evaluation**, *New Generation Computing, Vol 2, No. 2, 1984,*

pp. 105-129.

[ANID86] Avadis Tevanian, Richard F.Rashid, **Mach - A Basis for Future Unix Development**, *EUUG Autumn 86.*

[ARGO82] Arvind and K.P. Gostelow, **The U-interpreter**, *IEEE Computer, Vol. 15, No. 2, February 1982, pp. 42-48.*

[ARIN83] Arvind and R.A. Inaucci, **A Critique of Multiprocessing von Neumann Style**, *Proc. of the 10th Ann. Symp. on Computer Architecture, June 1983, pp. 426-436.*

[ARCU83a] Arvind, D.E. Culler, R.A. Inaucci, V. Kathail, R.E. Thomas, **The Tagged Data Flow Architecture**, *Prelimlnary version for distribution in subject 6.83s, MIT, Aug. 1983.*

[ARCU83b] Arvind, D.E. Culler, **Why Data Flow Architecture ?**, *MIT/CGSM-229, Sep. 1983.*

[ARNP86] Arvind, R.S. Nikhil, and K.K. Pingali, **I-structure: Data Structures for Parallel Computing**, *LNCS, vol. 279, Oct. 1986, pp. 336-369*

[ARTH80] Arvind and R.E. Thomas, **I-srtuctures : An efficient data type for functional languages**, *Tech. Memo 210, Lab. for Computer Science, MIT, Cambridge, Mass.*

[ARVI80] Arvind, **I-structure: an efficient data structure for functional languages**, *MIT/LCS/TM-178, Sep. 1980.*

[ARVI83] Arvind, R.A. Iannucci, **Two fundamental issues in multiprocessing : the data flow solution**, *MIT/CSGM 226-2, July 1983.*

[ATHM86] M.Amamiya, M. Takesue, R. Hasegawa, and H. Mikami,

Implimentation and Evalution of List-Processing-Oriented Data Flow Machine, *13th Annual Int. Symp. on Computer Architecture, June 1986, pp. 10-19.*

[AVID87] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper and M. Young, **Mach Threads and the Unix Kernel: The Battle for Control**, *CMU Tech. Report CMU-CS-87-149, Aug. 1987.*

[BACK78] John Backus, **Can programming be liberated from the von Neumann style ? a functional style and its algebra of program**, *CACM, Vol.21, No.8, pp. 613-641, Aug. 1978.*

[BUEH87] R.Buehrer, K.Ekanadham, "Incorporating data flow ideas into von Neumann processors for parallel execution", IEEE Trans. on Computers, Vol. c-36.No.12, Dec. 1987, pp.1515-1520.

[CARS81] Carson, J., and E. Forman, **Analysis of Local Area Network Performance**, *Proceedings of the 1981 IEEE Computer Networking Conference*, 1981.

[CORN79] M. Cornish, D.W. Hogan, J.C. Jensen, **The Texas Instruments distibuted data processor**, *Proc. Louisioana Computer Exposition, Lafayette, La., pp. 189-193, Mar. 1979.*

[DARI81] J. Darlington, M. Reeve, **ALICE - multi-processor reduction machine for the parallel evaluation of applicative languages**, *Proc. ACM conf. Functional Language Computer architecture, pp. 65-75, 1981.*

[DAVI78] A.L. David, **The architecture and system model of DDM1 : a recursively structured data driven machine**, *Proc. 5th Ann. Symp. Computer Architecture, New York, pp. 210-215, 1978.*

[DEGA84] J.B. Dennis, G.R. Gao, and K.W. Todd, **Modeling the weather with a Data Flow Supercomputer**, *IEEE TOC, Vol. c-33, No. 7, July 1984, pp. 592-603.*

[DENN74] J.B. Dennis, **First Version of a Data Flow Procedure Language**, *Lecture Notes in Computer Science, Vol.19, Springer-verlag, 1974, pp. 362-376.*

[DENN80] J.B. Dennis, **Data Flow Supercomputers**, *IEEE Computer, Vol. 13, N0. 11, Nov. 1980, pp. 48-56.*

[DENN82] J.B. Dennis, Wiilie Y-P Lim, W.B.Ackermann, **The MIT data flow engineering model**, *MIT/CSGM, Nov. 1982.*

[DEPA79] A.M. Despain and D.A. Patterson, **The computer as a component**, *1979.*

[DIST79] **Distributed Processing Technical Committee Newsletter**, *Vol. 1, No. 3, 1979.*

[ENSL74] Enslow, P., **Multiprocessors and Parallel Processing**, *Wiley, New York, 1974.*

[ENSL77] Enslow, P., **Multiprocessor Organization - A Survey**, *ACM Computing Surveys, March 1977.*

[ERIK86] Erik Reeh Nielson, Soren Laucsen, Vilhelm Rosenqvist, **An Expandable Object-Based UNIX Kernel**, *Summer Conference Proceedings 1984 USENIX Association.*

[FOLT80] Folts, H., **X.25 Transaction Oriented Features - Datagram and Fast Select**, *IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp. 496-500.*

[FOOT84] G.C. Fox and S.W. Otto, **Algorithms for Concurrent Processors**, *Physics Today, May 1984, pp. 50-59.*

[GAPA82] D.D. Gajski, P.A. Padua, D.J. Kuck, R.H. Kuhn, **A Second Opinion on Data Flow Machines and Languages**, *IEEE Computer, Feb. 1982, pp. 58 - 69.*

[GAER82] J. Gaudiot, M.D. Ercegovac, **A Scheme for Handling Arrays in Data Flow Systems**, *Proc. of the 3rd Conf. on Distributed Computing Systems, Fort Lauderdale, FL., Oct. 1982, pp. 724 - 729.*

[GAUd86] J. Gaudiot, **Structure Handling in Data-Flow Systems**, *IEEE TOC Vol. C-35, No. 6, Jun. 1986, pp. 489 - 502.*

[GOSE81] J.R. Goodman and C.H. Sequin, **Hypertree : A Multiprocessor Interconnection Topology**, *IEEE TOC, Vol. c-30, N0. 12, Dec. 1981, pp. 923-933.*

[GUWA80] J. Gurd and J. Watson, **A data driven system for high speed parallel computing**, *comput. Design 9, 6 and 7(June) 1980, pp. 91-100 and 97-106.*

[GUKI85] J.R. Gurd, C.C. Kirkham, and I. Watson, **The Manchester prototype dataflow computer**, *CACM 28, 1(Jan) 1985, pp. 34-52.*

[GURD80] J. Gurd, I. Watson, **Data driven system for high speed parallel computing - part 2: hardware design**, *Computer Design, pp. 97-106, July 1980.*

[HWAN84] Kai Hwang, Fay A. Briggs, **Computer architecture and parallel Processing**, *McGraw-hill, pp. 1-51, 1984.*

[HEMO76] P. Henderson, J.H. Morris, **A Lazy Evaluator**, *Proc. of 3rd ACM Sympo. on Principles of Programming Languages, Atlanta, 1976, pp. 95 - 103.*

[HEND80] P. Henderson, **Functional Programming : Application and Implementation**, *Englewood Clifts, N.J.,Prentice-Hall, 1980.*

[HOZO83] E. Horowitz and A. Zorat, **Divide-and-Conquer for Parallel Processing**, *IEEE TOC, Vol. c-32, No. 6, June 1983, pp 582-585*

[HUDA85] P.Hudak, **Para-functional programming**, *IEEE COMPUTER, Aug. 1986, pp.60-69.*

[HUDA86] P.Hudak, B,Goldberg, **Alfalfa: distributed graph reduction on a hybrid multiprocessor**, *Proceeding of a Workshop, New Maxico, Sep. 1986, pp.94-113.*

[HWBR84] K. Hwang, F.A. Briges, **Computer Architecture and Parallel Processing**, *Mcgraw-Hill, 1984.*

[HIRA84] K. Hiraki et al., **A hardware design of SIGMA-I, a data flowcomputer for scientific computations**, *Proc. 1984 International Conf. Parallel Processing, Michigan, pp.524-531, Aug. 1984.*

[INTE84] **LAN Computers User's Manual**, *Intel Corporation, 1984.*

[JAAS84] R. Jagannathan, E.A. Ashcroft, **Eazyflow : A Hybrid Model for Parallel Processing**, *Proc. of International Conf. on Parallel Processing, 1984, pp 514 - 523*

[JAME80] James E. Thornton, **Back-End Networt Approaches**, *IEEE Computer, Feb. 1980, pp 10-17.*

[JARK85] M. Jarke, Y. Vassiliou, **A Framework for Choosing a Database Query Language,** *Computing Surveys, Vol. 17, 1985.*

[JONE87] S.L.P.Jones, **The implementation of functional programming languages,** *Prentice Hall, 1987, pp.24-25.*

[KELI79] R.M. Keller, G. Lindstrom, S. Patil, **A Loosely-Coupled Applicative Multiprocessing System,** *Proc. of the 1979 National Computer Conference, IFIPS, 1979, pp 613 - 622.*

[KELI84a] R.M. Keller, F.C.H. Lin, **Simulated Performance of a Reduction-Based Multiprocessor** , *IEEE Computer, Vol. 17, No. 7, July 1984, pp 70 - 82.*

[KELI84b] R.M. Keller, F.C.H. Lin, J. Tanaka, **Rediflow Multiprocessing,** *Proc. of COMPCON 1984, pp 410 - 417.*

[KELL79] R.M. Keller, G. Lindstrom, S. Patil, **A loosely-coupled applicative multiprocessing system,** *National Computer Conference, pp.613-622, 1979.*

[KELL80] R.M. Keller, **Semantics and Application of Function Graphs** , *Technical Report UUCS-80-112 Dept. of Computer Science, University of Utah, Oct. 1980.*

[KELL85] R.M. Keller, **Rediflow Architecture Prospectus,** *Technical Report No. UUCS-85-105, Dept. of Computer science, University of Utah, Aug. 1985.*

[KIEB85] R.B. Kieburtz, **The G-Machine: a fast, graph reduction evaluator,** *Functional Programming Languages and Computer Architecture, pp. 400-413, 1985.*

[KORT84] H. Korth, G. Kuper, et al., **SYSTEM/U : A Database System Based on the Universal Relation Assumption,** *ACM TODS, Vol. 9, 1984, 331-347.*

[LIEB80a] Liebowitz, B.H., and J.H. Carson, **Tutorial : Distributed Processing, 2nd ed.,** *IEEE EHO 127-1, IEEE Computer Society, Long Beach, Calif., 1980.*

[LIKE86] F.C.H. Lin and R.M. Keller, **Gradient Model : A Demand-Driven Load Balancing Scheme**, *The 5th International Conf. on Distributed Computing Systems, 1986, pp 329-336.*

[MACO87] M.H. MacDougall, **Simulating Computer Systems**, *The MIT Press, 1987.*

[MAGO84] G. Mago, D. Middletron, **The FFP machine - a progressive report,** *Proc. of the International Workshop on High-level Language Computer Architecture, pp.513-525, May 1984.*

[MAIE83] D. Maier, J. Ullman, **On the Foundation of the Universal Relation Model**, *ACM TODS, Vol. 9, 1984, 1283-308*

[MAIE84] D. Maier, J. Ullman, **Maximal Objects and the Semantics of Universal Relation Databases**, *ACM TODS, Vol. 8, 1983, 1-14.*

[MARO82] M.J.Maron, **Numerical analysis**, *Macmillan 1982, pp.97-116.*

[METC76] Metcalfe, R., and D. Boggs, **Ethernet : Distributed Packet Switching for Local Computer Networks**, *Communications of the ACM, Vol. 19, 1976.*

[MICH86] Michael B. Jones, Richard F. Rashid, **Mach and Matchmaker:**

Kernel and Language Support for Object-Oriented Distributed Systems, *CMU Tech. Report CMU-CS-87-150, Sep, 1986.*

[MICH87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D.Black and R.Baron, **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**, *Proceedings of the eleventh ACM Symposium on Operating Systems Principles, Nov, 1987.*

[MICH87] Michael Durr, **TNetworking IBM PC**, *Que Corp., 1987.*

[MIKE86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M.Young, **Mach: A New Kernel Foundation fo Unix Development**, *USENIX 86 Summer.*

[PAHW85] A. Pahwa, **A. Arora, Automatic Database Navigation : Towards a High Level User Interface**, *proc. of entity-relationship approach, 1985.*

[PLAS76] A. PLASet al., **LAU system architecture: a parallel data-driven processor based on single assignment**, *Proc. 1976 International Conf. Parallel Processing, pp.293-302.*

[PERR87] N.Perry, **HOPE$^+$**, *International research report ref.IC/FPR/LANG/2.5.1/7, Functional Programming Section, Department of Computing, Imperial College, \university of London.*

[RASH81] Richard F. Rashid, George G. Robertson, **Accent : A communication oriented network operating system kernel**, *Proceeding of the 8th ACM Symposium on Operating Systems Principles 1981 PP 64-75*

[RAYM87] Raymond Brooke Essick IV, **The Corss-Architecture Procedure Call,**

The dissertation of Doctor of Philosophy in Computer Science of the University of Illinois at Urbana-Champaign, 1987.

[RITC74] Ritchie, D.M., and K. Thompson, **The UNIX Operating System**, *Communications of the ACM, Vol. 17, No. 7, 1974.*

[RUSA87] C.A. Ruggiero and J. Sargeant, **Control of parallelism in the Manchester Dataflow Machine**, *Functional Programming Languages and Computer Architectures, Portland, Oregon, USA, Sep. 14-16, 1987, Proceedings, Springer-Verlag, pp 1-15*

[SNJA85] L. Snyder, L.H. Jamieson, D.B. Gannon, and H. J. Siegel, **Algorithmically Specialized Parallel Computers**, *Academic Press, 1985*

[SRIN86] V.P. Srini, **An Architectural Comparison of Dataflow Systems**, *IEEE Computer, Mar. 1986, pp 68 - 88.*

[STAL83] Stallings, W., **Local Area Networks : An Introduction**, *Macmillan, New York, 1983.*

[STON76] M. Stonebraker, E. Wang, **The Design and Implementation of INGRES**, *ACM TODS, Vol. 1, 1976*

[STUC83] Stuck, B.W., **Calculating the Maximum Mean Data Rate in Local Area Networks**, *IEEE Computer Magazines, May 1983, pp. 72-76.*

[TANE81] Tanenbaum, A., **Computer Networks**, *Prentice Hall, Englewood Cliffs, N.J., 1981.*

[THUR79a] Thurber, K.J., and G.M. Masson, **Distributed Processor Communication Architecture**, *Lexington Books, D.C. Heath, Lexington, Mass., 1979.*

[THUR79b] Thurber K.J., **Tutorial : Distributed Processor Communication Architecture**, *IEEE Computer Society, Long Beach, Calif., October 1979.*

[TOLH88] M.R. Tolhurst, **Open System Interconnection**, *Macimillan, 1988.*

[TOWN87] Paul Townsend, **Flagship hardware and implementation**, *ICL Technical Journal, pp.68-87, Mar. 1986.*

[TRBR82] P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins, **Data-Driven and Demand-Driven Computer Arichitecture**, *ACM Computing Surveys, Vol. 14, No. 1, Mar. 1982, pp 93 - 143.*

[TREL82] P.C. Treleaven et al., **Data-driven and demand-driven computer architecture**, *Computing Surveys, Vol.14, No.1, pp.93-142, Mar. 1982.*

[TURN85] D.A.Turner, **Miranda - a nonstrict functional language with polymorphic types**, *Proc. Conference on Functional Programming Languages and Computer Architecture, Nancy, 1-16., LCNS201, Springer Verlag.*

[VEEN86] A.H. Veen, **Dataflow Machine Architecture**, *ACM Computing Surveys, Vol. 18, No. 4, Dec. 1986, pp 365-396.*

[VEEN86] A.H. Veen, **Data flow machine architecture**, *ACM Computing Surveys, V.18, No.4, pp.363-396, Dec. 1986.*

[VEGD84] S.R. Vegdahl, **A survey of Proposed Architectures for the Execution of Functional Languages**, *IEEE TOC, Vol. 33, No.2, Dec. 1984, pp 1050 - 1071.*

[WEIT80] Weitzman, Cay., **Distributed Micro/Minicomputer Systems**, *Prentice Hall, 1980.*

[WENG80] K.S. Weng, **An Abstract Implementation for a Generalized Data Flow Language**, *MIT/LCS/TR-228, Feb. 1980.*

[WILK87] A.Wilkstroem, **Functional programming using standard ML**, *Prentice Hall, Jun. 1987.*

[WILL83] William E. Burr, **An Overview of the Proposal American National Standard for Local Distribution Data Interface**, *Communication of the ACM, Aug. 1983, pp. 554-561.*

[WILL87] William Stalling, **Local Networks**, *Macmillan, 1987.*

[ZIMM81] Zimmermann, J., et al. **Basic Concept for the support of distributed Systems: The Chorus Approach**, *Second International Conference on Distributed Computing Systems, Versailles, France, 1981.*

주 의

1. 이 보고서는 과학기술처에서 시행한 특정 연구개발 사업의 연구보고서이다.

2. 이 연구개발내용을 대외적으로 발표할 때에는 반드시 과학기술처에서 시행한 특정 연구개발사업의 연구결과임을 밝혀야 한다.