論文93-30B-2-3

공유 메모리 다중 프로세서 시스템에서 고속 입출력 처리 기법

(High-Speed I/O Processing for Shared Memory Multiprocessor Systems)

尹 龍 鎬*, 林 寅 七*,

(Yong Ho Yoon and In Chil Lim)

要 約

본 논문에서는 공유메모리 다중 프로세서 시스템에서의 고속 입출력 처리 기법을 제안한다. 다양하고 많은 입출력 주변장치을 접속시킬 수 있는 고속 입출력 프로세서와 이 프로세서와 입출력 처리를 위한 주 프로세서 간의 통신 프로토콜과 작업 스케쥴링 기법을 제시하고, 주 프로세서의 성능에 비하여 상대적으로 느린 입출력 장치를 보완하기 위해 디스크 캐쉬 기법을 제안한다. 이 기법들을 TICOM 시스템에 구현하였으며 성능 측정을 통해 제안된 고속 입출력 처리 기법의 통계 자료를 수집 분석하고, 그 결과를 토대로 제안된 기법들의 우수성을 보였다.

Abstract

This paper suggests the new high-speed input/output techniques in a shared memory multiprocessor system. The high-speed I/O processor which can connect the different kinds of large sized I/O periperal devices, the communication protocol to the main processing units for I/O operations, and the job scheduling scheme are addressed. This paper also introduces the disk cache technique which supports the slow I/O devices comparing with the main processing units. These techniques were implemented in the TICOM system. The performance evaluation statistics were collected and analyzed for the suggested high-speed I/O processing techniques. These statistics show the superiority of the suggested techniques

I. 서 론

반도체 소자 기술의 급속한 발전으로 값싸고 강력한 마이크로프로세서와 메모리 소자가 공급되고, 소프트웨어 기술과 시스템 통합 기술에 힘입어 공유 메모리를 갖는 다중프로세서 구조의 상용 컴퓨터 시스템이 보편화되는 추세이다. 이렇게 값싸고 강력한 구성 요소를 갖춘 다중프로세서 시스템의 출현에도 불구하고 시스템에 장착된 프로세서의 수만큼 선형적인 성능 증가를 이루지 못하는 이유가 어디에 있는가? 프로세서 간의 통신 부하도 한 원인이 될 수 있지만 무엇보다 시스템을 구성하는 각 요소의 성능이 균형을 이루지 못하기 때문이다. 이와 관련하여 Amdahl [1] 은 시스템 성능과 각 요소들과의 관계를 수식 (1)로 표현하고 있다.

$$\text{Speedup} = 1/\{(1-Fr) + (Fr/Sp)\} \qquad (1)$$

* 正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)
接受日字 1993年 1月 15日

여기에서

Fr:개선 요소가 전체 시스템 사용에서 차지하는 비율
Sp:개선 요소의 성능개선 비율

이 수식의 의미는 시스템을 구성하는 요소들 중에 어느 한 요소의 처리 능력을 개선하였을때, 이 요소의 성능 개선이 전체 시스템의 성능개선에 반영되는 비율은 시스템내에서 이 요소가 사용되는 비율에 의해 제한을 받는다. 즉, 어떤 시스템의 사용 분야가 프로세서의 처리능력보다는 입출력의 처리능력이 보다 중요시 된다고 할 때, 이 시스템의 성능을 향상시키기 위해 단순히 프로세서의 성능을 향상시키는 것은 시스템 성능 향상에 기여할 수 없다. 따라서 공유 메모리와 버스를 기반으로 하는 다중프로세서 시스템에서 입출력의 성능개선 없이는 시스템 전체의 성능 향상을 기대할 수 없다.

시스템 버스를 기반으로 한 다중 프로세서 시스템에서는 버스의 대역폭으로 인하여 슬롯의 수가 제한된다.[2,3] 그리고 대부분의 슬롯들이 다중 CPU와 공유 메모리에 할당되고 제한된 몇개의 슬롯만 남게 된다. 이러한 환경에서 다량의 자료 저장장치, 많은 단말기와 통신장치를 직접 접속시켜 사용하는 것은 불가능하다. 각 주변장치로 부터 발생되는 빈번한 인터럽트 및 데이타 전송으로 공유 시스템 버스는 포화상태에 달해 시스템 성능에 심각한 타격을 줄 수 있다. 이러한 이유로 대부분의 공유메모리 다중 프로세서 시스템에서는 입출력을 전담하는 입출력 프로세서를 채택한다.[4,12] 입출력 프로세서(I/O Processor : IOP)는 많은 입출력 장치를 수용할 수 있는 장점은 있지만 다수의 입출력 요청자가 제한된 입출력 프로세서에 동시에 입출력을 요청할때 병목 현상(bottleneck)이 발생된다.[5,6] 따라서 입출력 프로세서에 쇄도하는 입출력 요청에 대하여 내부 자원을 균등하게 분배하여 사용하는 방법과 자료 지역성(data locality)을 위한 기법들이 시스템의 성능에 영향을 미치는 중요한 요소이다.[7,8]

본 논문은 공유 메모리 다중 프로세서 시스템에서 다양하고 대규모의 입출력 주변장치를 접속시켜 운용시킬 수 있는 고속 입출력 프로세서(High-speed I/O Processor : HIOP)를 제안하고, 이 프로세서에서 순차화(serialization)를 탈피한 인터페이스의 통신 규약을 정의하고, HIOP 커널 및 지역성(locality)을 향상하는 방안으로 디스크 캐쉬 기법을 제안한다. 그리고 TICOM 상에 이러한 기법을 구현하여 입출력 요청 빈도에 따른 IOP 내부의 통계 자료를 구하여 입출력 병목 현상이 근본적으로 어디에 있는가를 찾아 성능의 개선 방향을 제시한다.

본 논문은 서론에 이어 2장에서는 일반적인 공유 메모리 다중 프로세서 시스템에서 구성 가능한 입출력 구조를 소개한다. 3장에서는 프로세서 간의 통신을 위해 다중 채널 다중 메세지에 기반을 둔 메세지 전송 프로토콜(Message Passing Protocol : MPP)을 제시하고, HIOP 커널의 입출력 프로세스 스케쥴링 기법과 디스크 캐쉬 관리 기법에 대해서 제안한다. 4장에서는 이들 기법을 TICOM에 구현함에 있어 구현 환경, 측정 방법 및 측정 그리고 결과 분석을 기술하고, 5장에서는 4장의 결과 분석을 토대로 문제점과 방안을 제시한다.

Ⅱ. 다중 프로세서 시스템의 입출력 구조

일반적인 컴퓨터 시스템을 처리 기능별로 크게 분류하면 주 처리부(Main Processing Unit : MPU), 기억 장치부(Main Memory Unit : MMU), 입출력 처리부(Input/Output Processor : IOP) 그리고 정보 전송부(System Bus)등으로 나눌수 있다.

이러한 각 부분은 시스템 버스에 결합되어 시스템 구조를 형성한다. 다중 프로세서 시스템은 시스템 버스에 하나 이상의 주 처리부가 결합되어 있는 형태이다. 최상의 성능을 얻는 균형적인 시스템의 구조가 되기 위해 주 처리부에서는 처리능력 향상만큼 기억장치부와 입출력 처리부 그리고 정보 전송부도 함께 처리능력이 향상되어야 한다. 즉, 기억 장치부는 여러개의 메모리 모듈을 메모리 인터리빙(interleaving) 방식을 사용하여 다중 프로세서의 동일 메모리 모듈의 동시 접근에 대한 집중 현상(contention)을 방지하고 병렬(parallel) 또는 동시(concurrent) 처리를 가능하게 한다. 시스템 버스는 파이프라인 전송 방식을 사용하여 자료 전송율(data transfer rate)을 높여 정보의 고속 전송 요구에 대처할 수 있어야 한다. 그런데 입출력 처리부는 매체(media)가 가지는 특성으로 인해 성능 개선에 한계가 있고, 그리고 비교적 가격이 비싸기 때문에 다중 프로세서 시스템에서 입출력 구조에 많은 제약을 가진다.

본 장에서는 다중 프로세서 시스템에서 가장 적합한 입출력 구조를 제안하기 위해 여러가지 구성 형태을 소개한다.

1. 직접 제어 구조

다중 프로세서 시스템에서 가장 쉽게 생각할 수 있

는 입출력 구조는 시스템 버스에 입출력 제어기을 직접 결합하는 방식이다. 이 구조에서 1개의 프로세서가 N개의 입출력 장치를 처리할 능력을 가진다면 선형적인 성능 증가를 가상한 M개의 프로세서로 구성된 다중 프로세서 시스템은 N x M개의 입출력 장치를 처리할 능력을 가진다고 볼 수 있다. 물론 프로세서 간의 통신 부하, 시스템 버스의 자료 전송율, 기억 장치의 엑세스 속도 등 주요한 요인들을 배제한 경우이다. 그러나 단일 버스를 이용한 다중 프로세서 시스템에서는 제한된 슬롯(slot)으로 인해 많은 입출력 장치를 시스템 버스에 결합한다는 것은 많은 제약이 따른다. 또한 시스템 버스가 표준 버스가 아니면 입출력 장치들이 버스와 접속될 수 있도록 독자적으로 설계되어야 한다. 이러한 문제점들로 인해 특수한 목적외에는 거의 채택되지 않는 구조이다.

2. 아답터(adapter) 제어 구조

직접 제어 구조에서 입출력 장치는 버스에 의존한다. 즉, 입출력 제어기는 버스의 프로토콜을 따르는 인터페이스 로직을 가져야 한다. 이러한 제한으로 직접 제어 구조는 특수한 목적으로 사용된다. 아답터 구조는 입출력 버스로서 표준화된 버스와 시스템 버스간의 정합 기능을 갖는 구조로서 직접 제어 구조와 함께 주 처리부 관점에서는 각 입출력 장치를 직접 제어하는 형태이다. 이러한 구조는 아답터 하부에 많은 입출력 장치를 수용할 수 있고 비교적 좋은 자료 전송율과 다중 프로세서 시스템에서 각 자원의 이용율을 높일수 있는 구조이다. 그러나 주변장치와의 통신 즉, 요청 및 서비스가 시스템 버스를 전송 통로로 하기 때문에 시스템 버스가 포화(saturation)될 가능성이 있다. 문자 장치(character device)인 경우 한 문자의 입출력 마다 인터럽트가 발생되고, 번지가 지시되고, 자료가 전송된다. 문자 입출력이 시스템 성능과 직결되는 블록 입출력에 영향을 주지 않아야 하므로 다중 프로세서 시스템에서 일괄적인 아답터 제어 구조보다는 부분적으로 아답터 구조를 채택할 때 좋은 성능을 기대할 수 있다.

3. 입출력 프로세서 구조

직접 제어 구조나 아답터 제어 구조는 입출력 요청기로 부터 직접 입출력을 제어하는 구조이지만 입출력 프로세서 구조는 요청기의 요구에 대해서 자신의 제어 방식에 의해 입출력을 제어한다. 입출력 프로세서는 프로세서, 로컬 메모리, 로컬 버스, 다양한 주변장치로 구성된 일종의 독립된 컴퓨터 시스템이다.

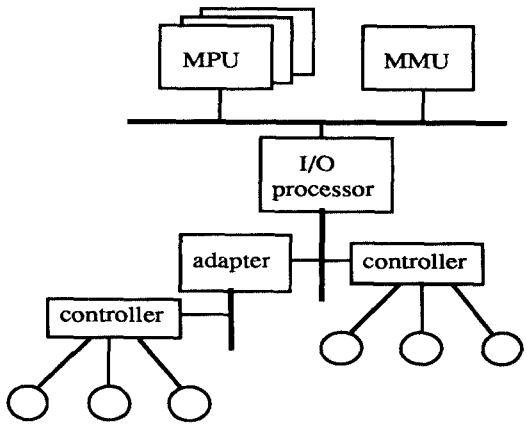

그림 1. 전용 입출력 프로세서 제어 구조
Fig. 1. Exclusive I/O Processor.

입출력 프로세서는 그림 1과 같이 아답터 제어 구조와 구조적으로 유사하지만 큰 차이는 내부 버퍼링과 디스크 캐쉬를 통해 물리적 입출력을 통하지 않고서 데이타 지역성(data locality)을 높일 수 있는 방법이 제공된다. 즉, 입출력 프로세서에 인터페이스하는 방법과 성능을 향상시킬 수 있는 지능성이 부여된다면 직접 제어구조 혹은 아답터 제어구조가 갖는 단점을 보완하게 된다.

본 논문에서는 고속의 입출력을 가능하게 지원해주는 고속 입출력 프로세서를 제안한다.

Ⅲ. 고속 입출력 프로세서(High-Speed I/O Processor)

고속의 입출력을 위한 입출력 프로세서의 외적인 요인은 하드웨어적인 것으로 시스템 버스의 자료 전송율과 기타 주변 장치의 엑세스 속도 등이 있다. 본 논문은 주어진 외적인 요인을 기반으로 입출력 프로세서의 내적인 요인의 향상 방안을 다음과 같이 제시한다.

첫째, 프로세서 간의 통신 방식으로 메세지를 이용한 다중 채널 다중 메세지 전송 방식을 정의한다.

둘째, 입출력 요청에 대해 프로세스 단위의 스케쥴링 기법을 제시한다.

셋째, 입출력 자료의 지역성을 높이기 위해 디스크 캐쉬 기법을 제시한다.

1. 메세지 송수신 방법

시스템 버스 상의 각 프로세서 사이의 정보 전달 방식으로 MPP(Message Passing Protocol)를 사용한다. 이 방식은 프로세서들 간에 공유하는 같은 번지 영역내의 공유 메모리와 하드웨어 인터럽트를 이용하는 메세지에 기반을 둔 전송 방식이다. 이것은 프로세서 간에 전달 정보량이 많을 때 유용하며 특히 입출력 프로세서와의 효율적인 통신을 위해 사용된다.

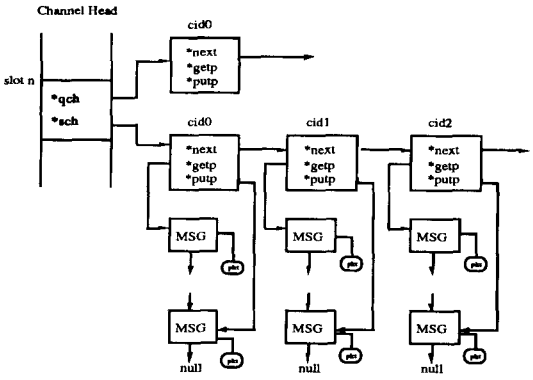

그림 2. 슬롯 n에 대한 MPP 구조도
Fig. 2. MPP Structure for Slot n.

입출력 프로세서는 많은 종류의 입출력 장치가 장착될 수 있다. 이러한 장치들은 처리 속도가 서로 다르므로 처리 순서를 달리하는 방법이 필요하다. 입출력 요청시에 독립적인 채널을 사용하여 서로 다른 벡터 값과 인터럽트 레벨을 가지면 입출력 커널에서 효과적인 자원 운용이 가능하다. 요청 프로세서들의 빈번한 입출력 요청시에는 일정 시간 또는 일정한 메세지 갯수 등을 이용하여 다중 메세지를 하나의 인터럽트로 입출력 프로세서에 전송함으로서 병목 현상을 완화시킬 수 있다.

그림 3과 같이 시스템 버스 위에 있는 각 프로세서는 공유메모리에 고유의 채널 헤드를 가지며 두개의 채널 즉, 요구 채널(reQuest CHannel : qch)과 서비스 채널(Service CHannel : sch)을 지시한다. 채널은 두개의 포인터(putp & getp)를 가지며 메세지를 단방향 접속 구조로 연결하여 채널을 형성한다. 메세지 삽입과 삭제를 용이하게 하기위해 getp는 SP(Service Processor)에 의해 메세지 삭제 포인터로만 사용되고, putp는 RP(Request Processor)에 의해 삽입 포인터로만 사용된다. 각 채널은 연결 포인터(next)에 의해 다중 채널을 형성하고, 채널 헤드로 부터 연결된 순서대로 채널 인식자(cid)를 부여한다. 채널 헤드의 첫번째 채널의 채널 인식자(cid0)는 슬롯간의 통신을 위해 예약되며 나머지는 해당 프로세서 내부 자원에 대한 다중 채널로서 고유의 인식자를 가진다. 예를 들어 cid1는 디스크 장치, cid2는 마그네틱 테이프 장치 등으로 부여하며 각 장치의 종류(device class)를 위한 정의값(definition value)도 채널 인식자 번호와 같게 함으로서 사용중 채널의 직접 사상(direct mapping)을 가능하게 한다.

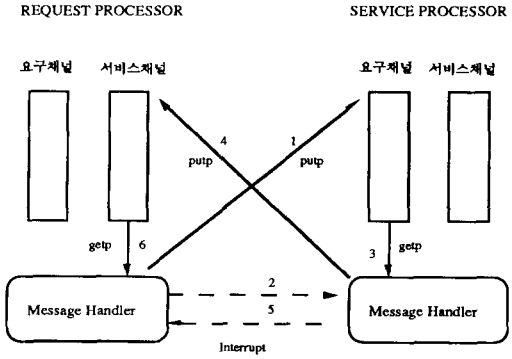

그림 3. 두 프로세서 간의 MPP 통신 절차
Fig. 3. MPP Communication Procedure between two Processors.

원천 슬롯(source slot)과 목표 슬롯(destination slot) 간의 통신 방식은 채널을 통해 메세지와 패킷의 전송으로 이루어진다. 메세지는 프로세서 사이의 교환 정보이고 패킷은 상대 프로세서의 하부 자원에 대한 요구 정보이다. 메세지는 패킷에 대한 포인터를 갖고 있으며 메세지와 패킷은 통신을 위한 기본 단위이다. 그림 3 은 두 슬롯 간의 통신 순서를 나타낸다. 각 프로세서는 메세지 핸들러를 가지고 있으며 목표 슬롯에 대한 요청이 있으면 1) 먼저 목표 슬롯 채널 헤드의 요구 채널에 메세지를 삽입하고, 2) 목표 슬롯의 프로세서에 인터럽트를 야기한다. 3) 인터럽트를 당한 목표 슬롯 프로세서의 메세지 핸들러는 자신의 채널 헤드의 채널에서 메세지를 가져와 서비스를 실시하며, 4) 서비스 완료후 원천 슬롯 채널 헤드의 서비스 채널에 서비스 완료 메세지를 삽입하고, 5) 원천 슬롯 프로세서에 인터럽트로 통보하면, 6) 원천 슬롯 프로세서의 메세지 핸들러는 서비스 완료된 메세지를 자신의 서비스 채널에서 가져와 결과를 조사하여 메세지를 복귀하는 기본 주기를 가진다. 각 채널은 여러 프로세서의 동시 조작에 의한 무결성을 보장받기 위해 채널 포인터 조작 과정을 임계 영역(critical section)으로 하는 보호 기법(locking mechanism)을 적용한다. [10,11]

1) 채널 헤드

시스템 버스 상의 각 프로세서 간의 통신은 채널 헤드(channel head)로 부터 시작된다. 각 프로세서가 자신이 가진 자원에 대한 응답기(Responder)가 없는 상태를 가정할때 프로세서 사이의 통신은 공유 메모리를 이용할 수 밖에 없다. 공유 메모리에서 운영체제가 관리하지 않는 일부를 할당받아 하나의 프로세서당 하나의 채널 헤드 구조체를 설정한다.

채널 헤드는 기본적으로 프로세서의 상태 정보(status)와 자신이 RP 또는 SP로 동작할 수 있기 때문에 요구 채널과 서비스 채널의 포인터 및 각각의 벡터 번호를 포함한다.

2) 채널

채널(channel)은 메세지 전송 통로이며 채널 헤드에서 지시된다. 채널은 각 프로세서의 내부 자원에 대한 독립된 전송 경로를 제공하므로 다중 채널로 구성된다. 이때 각 채널은 서로 다른 인터럽트 레벨과 벡터 값을 가질 수 있다. 이러한 구조는 입출력 프로세서와 같이 처리 속도가 다른 많은 내부 자원의 요구에 대한 순차화(serialization) 현상을 방지할 수 있다.

채널은 다수 프로세서에 의해 공유되기 때문에 상호 배타적인 사용(mutual exclusion)을 보장받기 위해 보호 필드(lock)를 가지고, 채널의 상태 정보와 메세지의 삽입과 삭제 포인터를 포함한다. 메세지의 삽입과 삭제 포인터는 서로 대응되는 프로세서가 각각 취급하므로서 FIFO 구조를 형성한다. 다중 채널은 채널의 단방향 접속 구조로 형성되며 접속 순서에 의해 채널 인식자와 벡터가 부여된다. 각 프로세서는 초기화시에 자신의 로컬 영역에 채널 인식자별로 채널 번지를 기록한다. 다중 채널의 접속 구조를 접근할때는 채널 번지를 통해 직접 지시함으로서 시스템 버스의 통신량을 줄인다.

3) 메세지

메세지(message)는 각 프로세서들 간의 공유 자원으로 사용 가능한 풀(pool)형태로 관리된다. 메세지 풀을 관리하는 메세지 헤드가 있는데, 이 헤드로 부터 메세지는 단방향 접속 구조로 연결된다. 헤드 구조에는 동시에 여러 프로세서로 부터 상호 배타적으로 할당 및 회수가 이루어져 무결성(integrity)을 보장하는 보호(lock) 변수를 가진다. 메세지는 RP 또는 SP가 처리할 제어 및 상태 정보를 포함한다. RP는 메세지를 할당받아 SP에 메세지를 전송하기 전에 메세지를 회수할 방법을 결정한다. 즉, 동기와 비동기 메세지 타입(type)이 있는데, 동기는 RP의 요청 프로세스가 메세지를 회수하기 위해 세마포 연산(semaphore operation)을 통해 동기화(synchronization)를 수행한다. 반면에 비동기는 RP의 요청 프로세스가 SP에 메세지를 전송하고 곧바로 회귀하여 다른 일을 수행할 수 있고, 나중에 메세지 핸들러가 돌아온 메세지를 메세지 풀에 회수해 주는 방식이다. 채널 형성을 위한 메세지 포인터가 있고 RP와 SP에 대한 인식자 필드는 채널 인식자와 함께 SP가 서비스 완료후 RP의 서비스 채널을 구하는데 이용된다. SP 내부 자원의 분류를 위해 필드가 있는데, SP가 입출력 프로세서인 경우 이 필드는 각각의 입출력 장치를 의미하므로 해당 장치 구동기에 접근하도록 해준다. SP가 RP의 요청에 따라 일을 수행하는 중에 비정상적인 상태가 감지되면 주의 메세지(attention message)를 반송함으로서 RP에게 통보한다.

4) 패킷

패킷(packet)은 RP가 SP 내부의 자원에게 전달할 내용이 수록되는 통신 매체로서 다양한 구조체를 가진다. 예를들어 입력 패킷 또는 제어용 패킷 등 사용자 정의로서 다양한 형태로 구성된다.

5) 송수신 절차 및 프리미티브

각 프로세서는 메세지 인터페이스를 위한 루틴을 가진다. 메세지 또는 패킷의 관리 주체는 항상 RP로서 SP는 RP로 부터 받은 메세지 혹은 패킷을 반송하는 역할만 한다. 물론 어느 프로세서이든지 RP 또는 SP가 될 수 있다. 메세지 핸들러는 일종의 인터럽트 처리 루틴이다. RP의 프로세스는 메세지에 대해 동기 또는 비동기 모드를 지정할 수 있다. 동기 모드는 SP의 결과에 따라 RP 프로세스의 진행 과정을 결정할 때 사용한다.

채널은 요구 채널과 서비스 채널이 있기 때문에 RP와 SP의 채널 결정 방법이 다르다. RP 프로세스는 SP의 요구 채널이고, RP의 메세지 핸들러는 자신의 서비스 채널을 결정한다. 그리고 SP의 프로세스는 RP의 서비스 채널이고, 메세지 핸들러는 자신의 요구 채널을 결정한다. 채널에 메세지의 삽입과 삭제는 독립된 포인터 변수를 사용한다. RP의 메세지 송신은 SP의 요구 채널에 메세지를 삽입하고 메세지 카운터를 증가하는 과정을 임계 영역으로 하며 인터럽트 전송은 적당한 메세지 갯수 혹은 타이머 구동에 의해 일정한 시점 후에 인터럽트를 전송하도록 한다. 이것은 메세지 전송 요구가 쇄도할 때 하나의 인터럽트 전송으로 여러개의 메세지를 한번에 전송하는 효과를 갖고 다중 프로세서 시스템에서 인터럽트 버스의 포화를 예방하는데 도움을 준다. SP의 메세지 수신은 메세지 핸들러가 요구 채널의 삭제 포인터(getp)로 부터 첫번째 메세지를 tmp(임시 메세지 헤

드)로 가져오면 그 채널에 연결된 모든 메세지를 가져올 수 있기 때문에 보호 입상도(lock granularity)를 줄여준다. tmp의 모든 메세지에 대해 전처리(preprocessing)를 한 다음 그 메세지를 명령큐(command queue)에 연결하고 명령큐를 기다리는 프로세스를 깨워준다. (wakeup)

o 송신 프리미티브

- 적당한 메세지 갯수에 의한 인터럽트

```
{
    채널 헤드 결정;
    인터럽트 마스크;
    채널 식별자에 의한 채널 결정;
    채널 잠금;
    채널에 메세지 삽입;
    채널의 메세지 카운터 증가;
    채널 잠금 해제;
    인터럽트 레벨 복귀;
    if (전송 가능 메세지 갯수) {
        if (타이머 구동중) {
            타이머 구동 해제;
        }
        인터럽트 전송;
    } else if (타이머 구동중이 아니면) {
        타이머 구동;
    }
}
```

- 타이머 구동기에 의한 인터럽트

```
{
    채널 잠금;
    if (채널에 메세지) {
        채널 잠금 해제;
        인터럽트 전송;
        if (타이머 구동중이 아니면)
            타이머 구동;
    } else 채널 잠금 해제;
}
```

o 수신 프리미티브

```
{
    채널 헤드 결정;
    인터럽트 마스크;
    채널 식별자에 의한 채널 결정;
    채널 잠금;
    채널로 부터 메세지를 tmp로 받음;
    채널의 메세지 카운터를 지움;
    채널 잠금 해제;
    인터럽트 레벨 복귀;
    for (tmp의 모든 메세지에 대해) {
        메세지 전처리;
        명령큐에 메세지 삽입;
    }
}
```

2. 입출력 제어 흐름

입출력 프로세서는 입출력 목적을 위해서 항상 목표 슬롯으로 동작한다. 그러나 입출력 프로세서에 의해 주의 메세지 전송시는 원천 슬롯이 될 수 있다.

입출력 프로세서는 외부 인터페이스에 메세지 핸들러가 있으며 메세지 핸들러에 의해 접수된 메세지는 내부 명령큐에 큐잉된다. 입출력 프로세서는 기본적으로 명령큐를 처리하는 하나의 프로세스가 존재하지만 어떤 순간 여러개의 프로세스가 존재할 수 있다. 처음 초기화를 주도한 프로세스 0는 디스크 캐쉬의 write-back 정책을 위한 데몬(daemon)으로 전환과 함께 sleep하면서 하나의 프로세스를 생성한다. sleep은 곧 명령큐 처리 프로세스의 소멸을 의미하기 때문에 sleep시에 새로운 명령큐 처리 프로세스를 하나 만들어 주어야 한다. fork에 의해 생성된 프로세스는 항상 동일한 텍스트를 가지는데, 이 텍스트는 명령큐에서 메세지를 떼어서 해당 구동기를 호출하는 절차를 가진다.

주 프로세서에서 입출력 프로세서에 입출력을 요구하였을때, 입출력 프로세서에서 입출력 처리를 위한 프로세스 수명은 그림 4와 같다.

각 노드(node)는 프로세스의 동작을 나타낸다. 처음 명령큐가 empty이면 프로세스는 sleep 한다. 그렇지 않을 경우 장치 구동기는 먼저 디스크 캐쉬에 원하는 데이타 블록이 있는지를 보아 있으면 (cache hit) RP에 응답하고 다음 명령큐를 반복 수행한다. 디스크 캐쉬에서 적중하지 않았으면 (cache miss) 물리적인 입출력을 수행한다. 입출력 장치를 요구한 후 프로세스는 sleep 한다. 이때의 sleep이 명령 처리의 첫번째 sleep이면 명령큐 처리를 위한 새로운 프로세스를 생성하기 위해 프로세스 fork를 수행 고 sleep 한다. 이때 idle 프로세스의 고갈로 fork 실패

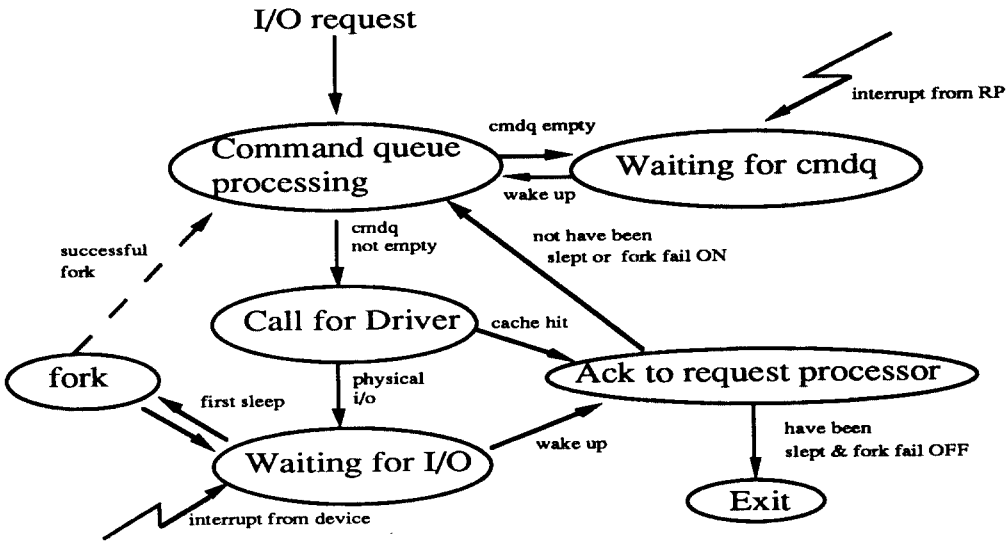

그림 4. 입출력 처리를 위한 IOP 프로세스 수명
Fig. 4. IOP Process life cycle for I/O Processing.

시 fork fail을 ON 시켜 fork 실패를 나타낸다. 이 프로세스는 입출력 장치의 완료와 함께 전달된 인터럽트 핸들러가 wakeup 시켜주면, RP에 응답하고 조건에 따라 exit 하거나 다음 명령큐를 위해 계속한다. 명령큐를 처리하는 프로세스는 명령큐가 empty 시 sleep하는데, RP의 입출력 요청을 통보 받은 인터럽트 핸들러는 명령큐에 메세지를 달고, 그 프로세스를 wakeup 시켜준다. 이로써 여러개의 명령이 동시에 처리되나, 명령큐에는 항상 오직 한개 만의 프로세스가 대기한다.

3. 입출력 프로세스 스케쥴링

IOP의 입출력 요구에 대한 스케쥴링 기법은 입출력 요구로 정의할 수 있는 메세지를 프로세스에 대응시켜 CPU 할당과 회수를 위한 프로세스 단위의 관리 기법으로 정의할 수 있다. 프로세스를 CPU에 할당하고 또는 회수하는 정책으로 가장 간단한 방법은 정해진 순서대로 처리하는 FIFO 방식일 것이다. 이러한 정책 결정은 job의 특성과도 관련이 있다. 입출력 job은 입출력 한계 (I/O bound) 특성으로 인해 CPU 점유 시간이 비교적 짧기 때문에 스케쥴링 부하 즉, 문맥 교환 (context switching)을 최소화할 필요가 있다. 입출력 프로세서에서 프로세스는 text, data 그리고 stack으로 구성되며 프로세스 0가 디스크 캐쉬의 일관성 유지를 위해 데몬으로 동작하는 동안 common text를 공유하는 여러개의 프로세스에 의한 multi-tasking 개념으로 동작한다. 프로세스는 프로세스 테이블에서 할당받은 이후에는 수행큐 또는 sleep 큐에 위치하고 exit 할때 다시 프로세스 테이블에 idle 상태로 존재한다. (그림 5)

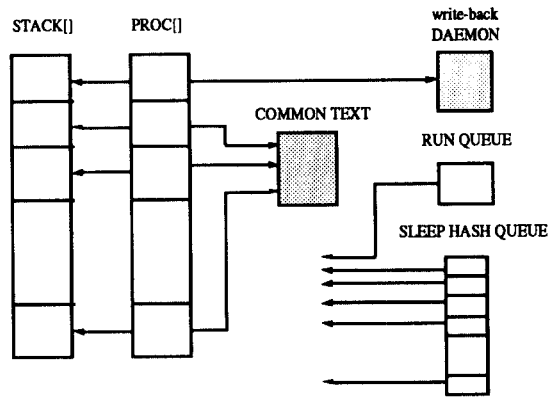

그림 5. 프로세스 관리 구조도
Fig. 5. Process Management.

입출력 job은 대부분 입출력 대기 상태로 시간을 보낸다. 이러한 대기 시간동안 다음 입출력의 준비를

위해 CPU를 할당할 간단한 스케쥴링이 필요하다. 그러나 입출력 job의 특성이 동일하지는 않다. 즉, 어떤 job은 대기 시간이 길거나 짧을 수 있고, 주변 장치로 부터 데이타 전송 단위가 다르고, 또한 입출력 대기없이 디스크 캐쉬 또는 문자 버퍼에서 만족하는 경우와 같이 프로세스의 상이한 동작으로 인한 처리 방법을 달리할 필요성도 대두된다. 따라서 사용자가 임의로 우선 순위를 부여하여 스케쥴링시 주변장치의 특성을 CPU 할당 순서로 정하는 요소로 활용한다.

1) 프로세스 구조

프로세스의 구조를 간략화시켜 입출력 제어에 필요한 기본 요소로 구성하였다. 문맥(context)을 저장할 영역과 프로세스 상태 필드, 우선순위 필드를 두었는데, 우선 순위는 각 job 마다 고유의 우선순위 값을 가진다. sleep 하는 이유를 나타내는 사건 (event) 값을 기록하는 필드는 wakeup시에 탐색 키(search key)로 사용된다. 프로세스는 단방향 큐로 연결되므로 수행큐(run queue) 또는 sleep큐 등을 구성하는 포인터 변수가 있다.

2) 프로세스 생성

프로세스의 생성은 fork에 의해 이루어진다. 수행중인 프로세스가 첫번째 sleep에 빠지기 전에 새로운 프로세스를 fork를 통해 생성한다. 이것은 명령큐에서 하나의 job을 할당받은 프로세스가 sleep함으로서 다음 명령큐의 처리를 위한 프로세스가 소멸되는 것을 방지하기 위함이다. fork의 수행 절차는 다음과 같다.

o 프로세스 생성 : fork()

프로세스 테이블에서 NULL인 엔트리를 찾는다;
if (모두 사용중이면)
생성 실패 플래그 설정;
회귀;

우선순위 변수에 기본 우선 순위 값을 기록;
콘텍스트 저장 영역을 지움;
콘텍스트 저장 영역의 PC 위치에 common text 번지를 설정;
콘텍스트 저장 영역의 SP 위치에 대응되는 스택의 번지를 설정;
수행큐에 연결;
회귀;

3) 프로세스 상태 변환

입출력 프로세서의 초기화 과정을 종료하면서 프로세스 0는 디스크 캐쉬의 write-back 정책을 위한 데몬으로 전환되고 최초로 sleep을 수행한다. 프로세스의 상태 천이는 3가지가 있다. RUNNING 상태는 CPU를 점유하고 수행중인 상태이고, SLEEP 상태는 RUNNING 중인 프로세스가 어떤 자원 또는 event로 인하여 계속 수행할 수 없을때 CPU를 runable한 다른 프로세스 중에 우선순위(priority)가 가장 높은 프로세스를 선택하여 제어를 넘겨주고, 자신은 sleep queue에 연결하여 BLOCKING 상태로 들어간다. READY TO RUN 상태는 sleep중인 프로세스가 wakeup되거나 새로운 프로세스가 생성될때 RUNNING을 위해 준비중인 수행큐에 연결된 상태를 나타낸다. 그림 6에서 프로세스의 생성인 fork가 일어나는 시기는입출력 대기를 위한 최초의 sleep을 수행할때 항상 새로운 프로세스를 fork하여 명령큐를 조사하는 하나의 프로세스를 유지하도록 한다. 프로세스의 종료는 proc 테이블이 full되어 fork 실패 현상이 일어나지 않아야 하고 입출력 대기를 한적이 있는 프로세스는 즉시 종료하여 proc 테이블의 한 entry를 비워준다.

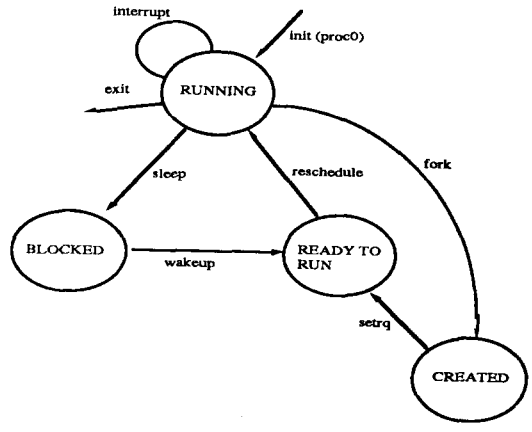

그림 6. 프로세스 상태 천이
Fig. 6. Process State Transition.

exit를 하지 않는 경우는 fork fail로 fork fail flag가 ON 상태이거나 또는 입출력 대기과정을 거치지 않아 명령큐 처리를 위한 프로세스를 fork하지 못한 경우로서 자신이 exit하지 않고 다음 명령큐를 RUNNING 상태에서 계속한다. RUNNING 중인 상태에서 인터럽트가 발생하면 현재 수행중인 프로세스의 문맥을 스택에 저장한 후 CPU는 인터럽트 핸들러를 수행한다. 서비스 완료 후 스택의 문맥을 레지스타에 restore한 후 프로세스의 RUNNING 상태

가 계속된다.

4) 문맥 교환

문맥 교환 (context switching) 오버헤드를 최소화하고, 가능한 문맥 교환이 발생하지 않도록 고려하였다. CPU의 처리 시간이 길때 강제적인 문맥 교환이 CPU 사용의 균등한 스케쥴링 측면에서 더 효과적일 수 있다. 그러나 본 논문에서는 문맥 교환이 일어나는 시간은 현재 수행중인 프로세스가 sleep에 빠질때로 국한한다. 이와 같은 관점은 입출력 job의 특성에 연류되는데, 한개의 프로세스를 CPU가 수행하는 시간보다는 입출력의 대기 시간이 상대적으로 많이 걸리기 때문에 이 sleep 기간동안 다른 입출력 요청 처리를 위해 생성된 프로세스 혹은 sleep에서 깨어난 다른 프로세스의 문맥을 교환해 주므로서 짧은 CPU 사용을 원하는 많은 프로세스에게 우선순위 원칙에 따라 CPU 사용권을 넘겨주기 위함이다.

4. 디스크 캐쉬 관리 기법

일반적으로 프로세서와 디스크 장치의 물리적 입출력에 대한 속도 차이로 인한 병목 현상을 완화하고자 시스템 메모리에 디스크 블록을 캐쉬한 기법이 사용되고 있다. 이러한 소프트웨어적인 캐쉬를 버퍼 캐쉬라고 하며 입출력 요청시 캐쉬에서 실패하면 물리적인 입출력을 야기한다. 본 논문에서 제안된 고속 입출력 프로세서는 지능형으로 하부 입출력 장치의 제어을 위해 흐름 제어를 위한 버퍼링과 시스템 전체 성능에 가장 큰 영향을 미치는 디스크 입출력 성능 향상을 위해 디스크 캐쉬 기법을 제시한다.

디스크 캐쉬는 개념적으로 디스크 장치와 입출력 프로세서 사이에 위치하여 디스크의 일부 블록의 복사본을 적절한 기법으로 유지하며 입출력 요구시에 디스크 캐쉬에서 만족하는 비율인 적중률(hit ratio)을 높여준다. 캐쉬의 동작 원리는 unix 운영체제의 버퍼 캐쉬 관리 방법과 유사하다. 본 논문에서는 디스크로 부터 가장 최적화된 입출력 단위로 캐쉬 라인을 구성하고, 다시 한 라인은 buffered i/o의 입출력 단위인 블록으로 나누어 상태에 따라 운용된다. 다음은 제시된 캐쉬의 구성과 동작 원리에 대하여 설명한다.

1) 캐쉬 구성

디스크 캐쉬의 구성은 입출력 프로세서의 DBRAM(Data Buffer RAM)에 캐쉬 라인을 구성하고 각 캐쉬 헤드가 이 라인을 지시하므로서 하나의 캐쉬 세그먼트를 구성한다. 한 라인은 8개의 블록(512 bytes * 8)으로 구성되는데, 이것은 디스크로부터 한번에 가장 효율적인 입출력 단위로 정해진다. RP의 buffered 입출력의 단위인 논리 블록은 512 bytes에서 4K bytes까지 효과적으로 지원한다. 여러개의 캐쉬 세그먼트는 해쉬큐(hash queue)에 의해 전체적인 디스크 캐쉬를 구성하며 block number가 태그(tag)로 사용되고 hash function에 의해 해쉬 인덱스가 구해진다. 교체 알고리듬(replacement algorithm)은 LRU(Least Recently Used) 방식이고, dirty 블록의 디스크 갱신 방식은 write-back 정책으로 입출력 프로세서에 write-back 전담 데몬이 있다.

캐쉬 헤드는 캐쉬 라인과 블록 각각에 대하여 4가지 상태 값을 가지는데, 그 의미는 다음과 같다.

VALID : 디스크의 해당 블록과 동일한 블록으로 유효한 값을 가짐.

INVALID : 무효화된 블록으로서 태그 미스와 같은 의미로 취급.

PVALID : 캐쉬 라인의 여러 블록중에 VALID와 INVALID가 공존할때, 일부 유효한 블록을 가짐.

DIRTY : 유일하게 유효한 블록을 가지며 디스크의 내용과는 일치하지 않음.

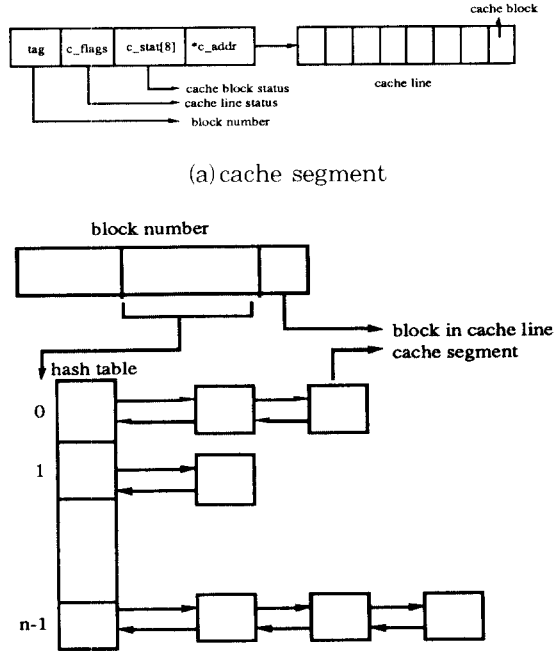

그림 7. 디스크 캐쉬 관리 구조도
Fig. 7. Disk Cache Structure.

그림 7(a)는 하나의 캐쉬 세그먼트를 보이는데, 태

그와 캐쉬 라인 및 각 블록의 상태 변수와 8개의 512 bytes로 구성된 캐쉬 라인을 보인다. 그림 7(b)는 캐쉬 관리 구조로서 각 블록 번호를 이용하여 구해진 해쉬 키에 의해 캐쉬 세그먼트가 해쉬된다. 각 캐쉬 세그먼트의 태그 필드는 캐쉬 라인의 첫번째 블록 번호가 된다.

2) 캐쉬 동작

본 논문에서는 버퍼 캐쉬 관리 기법으로 unix 운영체제에서 관리하는 방법을 도입하여 캐쉬 세그먼트의 상태 천이에 대해서 설명한다. 하나의 캐쉬 세그먼트는 8개의 데이타 블록을 포함한다. 각 블록은 입출력 요청에 대한 데이타의 전송 단위이고, 한 라인은 디스크로 부터 가장 효율적인 입출력 단위이다. 그림 8은 raw 입출력과 buffered 입출력에 대한 캐쉬 세그먼트의 상태 천이를 보이고 있다. 먼저 캐쉬 라인의 상태 필드에 대한 각 블록의 상태 필드와의 관계를 보면 다음과 같다.

VALID 8개의 블록이 모두 VALID일때
INVALID 8개의 블록이 모두 INVALID일때
PVALID VALID와 INVALID가 공존할때
DIRTY 8개의 블록중 하나라도 DIRTY일때

raw 입출력에서 읽기 동작은 캐쉬 라인의 각 블록의 상태 필드를 조사하여 DIRTY이면 상태 변화없이 캐쉬 블록으로 부터 읽어오고, INVALID이면 라인 상태값을 VALID 혹은 PVALID로 바꾸고 디스크로 부터 캐쉬 블록으로 읽어온다. raw 입출력의 쓰기 동작은 디스크에 먼저 데이타를 write한 다음 캐쉬의 해당 블록에 데이타를 복사하고 라인 전체 블록이 복사되었으면 이전 상태에 상관없이 VALID로 바꾸고, 나머지 블록이 INVALID이면 PVALID로 바뀌고 DIRTY이면 라인 상태는 변화가 없다.

buffered 입출력에서 read operation은 캐쉬에 먼저 접근하여 조사한다. VALID이면 상태의 변화 없이 캐쉬에서 읽어가고, INVALID이면 디스크에서 블록을 캐쉬로 먼저 읽어와서 상태 필드를 VALID로 바꾼다음 캐쉬로 부터 읽어간다. PVALID인 경우 INVALID인 블록은 디스크로 부터 읽어와서 VALID로 상태를 바꾼다음 캐쉬에서 읽어가고 DIRTY인 경우 여러가지 상태 변화가 예상된다. 먼저 DIRTY 블록은 디스크에 플러쉬(flush)하고 VALID로 바꾼다음 라인 상태를 VALID 혹은 PVALID로 바꾸고, 원하는 블록이 DIRTY가 아니면 나머지 블록에 상관없이 VALID 블록에 DIRTY 라인이 된다. buffered 입출력에서 쓰기 동작은 캐쉬에 복사한 다음 라인과 해당 블록을 DIRTY로 마크한다.

Ⅳ. 성능 측정

1. 측정 방법

디스크 캐쉬를 포함한 입출력 장치에서는 캐쉬에서의 적중, 실패율이 성능에 중요한 영향을 미치기 때문에 적중, 실패율을 측정할 필요가 있다 . 현재까지 수행된 디스크 캐쉬의 적중율 측정 방식은 크게 3가지로 분류된다.

첫째 방식은 하드웨어 모니터(Hardware Monitor)를 이용하는 방식으로서 캐쉬 관련 하드웨어에 하드웨어 모니터를 부착시켜서 캐쉬에서 발생하는 적중, 실패수를기록하는 방식이다.

둘째 방식은 프로세서에 내장된 트레이싱(tracing) 기능을 이용하는 것이다. 트레이스 비트(bit)가 설정되면 프로세서는 트랩(trap) 인터럽트 상태로 들어가며, 이때 인터럽트 처리 루틴내에서 캐쉬 태그(Tag)의 내용을 비교하여 요구된 블록의 적중, 실패를 알

Member	c_flags	c_stat[8]			Buffered I/O	
Meaning / Operation	cache line status	cache block status	READ	WRITE	READ	WRITE
STATUS	VALID	VALIDs	VALID or PVALID	VALID or PVALID or DIRTY	VALID	DIRTY
	INVALID	INVALIDs			VALID	
	PVALID	VALID & INVALID			VALID	
	DIRTY	DIRTY & any			DIRTY or VALID or PVALID	

그림 8. 캐쉬 상태 천이도
Fig. 8. Cache State Transition Table.

수 있게 한다. 트레이스 비트는 인스트럭션(Instruction)을 이용해 설정될 수 있으며 컴파일러의 도움을 받거나 측정자가 프로그램을 작성할 수도 있다.

셋째 방식은 소프트웨어 시뮬레이션 방식으로서 캐쉬를 포함한 대상 시스템을 모형화한 소프트웨어 시뮬레이터를 만들고 실제 시스템 혹은 수학적 모델로부터 구한 입력 트레이스(Trace)를 시뮬레이터에 제공하여 캐쉬에서의 적중율을 구하는 방식이다.

각각의 방식은 서로 장단점을 가지고 있다. 하드웨어 모니터 방식은 정확한 결과를 얻을 수 있으나 하드웨어를 구현해야 하는 비교적 비싼 비용을 부담해야 하고, 아울러 대상 시스템의 하드웨어 구조를 상세히 알아야 하드웨어 모니터를 설계할 수 있다는 어려움이 있다. 트레이싱 기능을 이용하는 방식은 비교적 데이타의 정확성은 유지하고 있으나, 캐쉬내의 데이타를 프로그램 상으로 접근할 수 있어야 한다는 제약과 실제 시스템 운용상황에서 없는 인터럽트 처리 루틴 처리 시간으로 인해 실제 시스템과 다른 상황의 데이타를 얻을 가능성이 있다는 단점이 있다. 소프트웨어 시뮬레이터를 이용하는 방식은 위의 두가지 방식에 비해 정확성은 상대적으로 떨어지지만 실제 시스템이 없는 상황에서도 소프트웨어를 이용하여 대상 시스템을 쉽고 값 싸게 모형화할 수 있다는 장점이 있으며, 시스템 구성을 바꿔 가면서 결과를 예측할 수 있는 장점이 있다. 시뮬레이터에 주어질 입력 트레이스는 실제 시스템에서 직접 구하여 사용하거나[8], 수학적 모델을 이용하기도 한다.

본 논문에서 제안한 기법에 대한 성능 측정 방식은 성격상 소프트웨어 시뮬레이터를 이용한 형태의 방식을 이용한다. 디스크 캐쉬는 일종의 시뮬레이터로 구성하고, 이 시뮬레이터를 이용하여 디스크 캐쉬의 적중, 실패율을 측정한다. 입력 트레이스는 벤치 마크 프로그램인 AIM SUITE II를 수행하여 구한다. 호스트(host)에서 입출력 요구율(메시지 전송율)을 변화시키면서 디스크 캐쉬의 유무에 대해 예상되는 병목 위치에 데이타를 구하는 기능을 추가한다. 측정 대상은 디스크 구동기에 대한 이용도 및 명령 큐, 디스크 큐, 수행 큐의 단위 시간(1 milli second)당 평균 큐의 길이 그리고 IOP가 접수한 메시지 갯수, 프로세스가 명령 큐에 대하여 응답하는 시간인 "command fetch time"에 대한 간격(interval) 등의 통계 자료와 호스트에서 IOP에 대한 응답 시간의 평균 값들이다.

2. 성능 측정 환경

고속 입출력 처리를 위해 제안된 기법들에 대한 성능을 측정하기 위해 TICOM 시스템에 제안된 기법들을 구현하였다. TICOM의 최대 규격은 20개 까지의 CPU와 4 IOP, 1 SCM, 512M bytes 메모리, 32G bytes 디스크 용량을 제공하는 공유 메모리 다중 프로세서 시스템이다.

본 논문에서 제안된 기법들을 TICOM에 구현하기 위해 TICOM 운영체제에 다중 채널 다중 메세지방식의 메세지 통신 규약을 적용하였고, IOP의 ROM에 메세지 통신 규약과 스케쥴링 기법 그리고 디스크 캐쉬 기법을 채택하여 구현하였다.

TICOM의 측정 환경은 주로 시스템 성능에 영향을 미치는 블록 입출력에 대한 제한된 구성으로서 2 CPU, 256M bytes 메모리, 1 IOP, 1 SCM 그리고 VME 버스에 SMD 인터페이스로된 1G bytes 용량의 한개의 디스크 장치를 가지는 시스템이다.

표 1. 3.5Mbytes Disk cache에서 측정 자료
Table 1. Measured Data in 3.5Mbytes Disk Cashe.

측정 종류 \ I/O 요청률	1ms	4ms	7ms	10ms	13ms	단위
디스크 구동기 이용률	66.215	34.619	29.967	28.023	26.820	%
CMD 큐 길이	0.832	0.025	0.009	0.005	0.003	개수/ms
디스크큐 길이	8.681	5.643	5.281	5.157	5.085	개수/ms
RUN 큐 길이	0.009	0.007	0.008	0.011	0.012	개수/ms
대기 메세지수	5.602	1.369	0.951	0.793	0.736	개수/ms
command fetch interval	0.771	2.195	3.666	5.168	6.682	ms
IOP 응답시간	5.103	3.447	3.132	2.885	3.015	ms

표 2. Non-Disk cache에서 측정 자료
Table 2. Measured Data in Non-Disk Cashe.

측정 종류 \ I/O 요청률	1ms	4ms	7ms	10ms	13ms	단위
디스크 구동기 이용률	98.407	98.411	95.696	82.171	68.259	%
CMD 큐 길이	0.013	0.011	0.009	0.005	0.003	개수/ms
디스크큐 길이	289.554	286.276	21.430	13.434	9.735	개수/ms
RUN 큐 길이	0.017	0.017	0.017	0.020	0.019	개수/ms
대기 메세지수	288.567	285.488	19.177	8.902	5.135	개수/ms
command fetch interval	2.955	3.008	3.686	5.199	6.760	ms
IOP 응답 시간	1712.815	1689.850	130.861	84.166	62.493	ms

3. 측정 결과

실제로 TICOM에서 AIM SUITE II의 표준 부하에 대하여 3.5M bytes의 디스크 캐쉬를 사용하였을 때 읽기 적중율(read hit ratio)은 80%를 얻었고, 쓰기 적중율(write hit ratio)은 95%를 얻었다.

디스크 캐쉬의 유무에 따라 입출력 요청율을 변화하였을때 측정한 결과는 표 1과 표 2와 같다. 표 1은 3.5M bytes 디스크 캐쉬를 사용하였을 경우이고, 표 2는 디스크 캐쉬를 사용하지 않았을때 측정한 값이다. 입출력 요청율은 AIM에서 디스크 입출력이 차지하는 부분을 일정하게 감소하면서 구해진 값이다.

4. 결과 분석

측정 결과를 분석하기 전에 먼저 디스크 캐쉬의 일관성을 유지하기 위해 매초 간격으로 write-back 데몬이 dirty 블록을 찾아 디스크 블록을 갱신하는 오버헤드가 디스크 구동기 이용율, 디스크큐 길이, IOP 응답 시간 등에 나타난 결과치에 직접적인 영향을 미치고 있음을 전제한다.

디스크 구동기의 이용율은 실제 디스크로 물리적인 입출력을 야기함을 의미하는데 디스크 캐쉬가 있을 경우에도 입출력 빈도가 많으면 캐쉬 미스가 많다는 것을 보여준다. 명령큐, 디스크큐 그리고 수행큐는 메세지의 대기 행렬로서 큐의 길이는 병목 위치를 예측하게 한다. 입출력 요청이 IOP에 도착하여 여러 대기 행렬을 거치지만 어떤 순간 대부분이 디스크큐에 대기하며 이것은 물리적인 입출력에 의한 입출력 대기 시간의 지연으로 인한 병목 현상이 디스크큐에서 나타나고 있음을 알 수 있다. 대기 메세지 수는 요청 프로세스가 최대 300개 메세지를 보유하고 있을 때 IOP가 어떤 순간에 보유하고 있는 메세지 수로서 디스크 캐쉬가 없고 입출력 요청 빈도가 높을때 거의 서비스하지 못하고 물리적 입출력 대기 상태로 머물고 있음을 보여준다.

command fetch time의 간격은 물리적 입출력을 준비하는 시간 혹은 디스크 캐쉬에서 적중하여 자료 전송을 완료할 때까지 소요되는 시간으로 볼 수 있으며, 물리적인 입출력인 경우 프로세스가 입출력 sleep을 하면서 새로운 프로세스를 fork하고 문맥 교환 등을 함으로서 비교적 많은 시간이 소요된다. 이것은 디스크 캐쉬를 사용한 표 1에서 command fetch interval time이 입출력 요청율 보다 거의 2배 빠른 시간으로 대부분 프로세스가 명령큐를 기다리고 있다. 입출력 요청율을 변화시켰을 경우에도 일정하게 유지하는 반면, 디스크 캐쉬를 사용하지 않은 표 2에서는 물리적인 입출력이 많음으로 인하여 구동기 이용율이 높다. 입출력 요청율이 1ms 와 4ms 일때 command fetch interval time이 길게 나타나고 있다. 표 1과 표 2에서 나타난 결론적인 수치는 IOP 응답 시간에 의해 극명하게 나타내고 있다. 디스크 캐쉬의 유무에 따라 IOP 응답 시간의 차이가 크다는 것은 디스크 캐쉬가 성능에 큰 영향을 미친다는 것을 알 수 있고, 특히 입출력 요청 빈도가 높을 때 캐쉬가 있는 경우 단위 입출력 요청에 대해 상대적으로 훨씬 좋은 특성을 보여주고 있다.

V. 결 론

본 논문은 버스를 기반으로 하는 공유메모리 다중 프로세서 시스템에서의 입출력 제약 사항을 수용하는 고속 입출력 프로세서 구조를 제안하였다. 이 입출력 프로세서가 주 프로세서와 원활한 처리를 위한 통신

프로토콜인 MPP를 정의하였고, 여러 주 프로세서에서 전달되어 오는 대량의 입출력 명령들을 순차적이 아닌 multi-tasking 기법을 활용한 입출력 job scheduling 기법과 주 프로세서에 비해서 상대적으로 느린 입출력처리 장치들을 보완하기 위한 디스크 캐쉬 기법을 제시하였다.

그리고 다중 프로세서 시스템에서 고속 입출력 처리를 위한 입출력 프로세서를 제안하면서 일반적인 입출력 프로세서 커널로 채용 가능한 기법도 소개하였다. TICOM 위에 이러한 기법들을 구현하여 성능을 측정한 결과 제시된 기법이 입출력 성능을 크게 향상시킬 수 있음을 알 수 있었다.

특히 본 논문에서 제안한 다중 채널 다중 메세지 전송 방식은 일반적인 밀결합 다중 프로세서 시스템의 프로세서 간의 통신을 위해 가장 효과적으로 적용할 수 있다.

본 논문에서는 다중 프로세서 시스템에서 많은 입출력 자원을 제어하는 입출력 프로세서 구조의 예상되는 병목 현상을 다중 채널 다중 메세지 전송 방식으로 해결하였다. 프로세스 관리 및 디스크 캐쉬 기법으로 자원의 효율적인 운용과 병목 현상의 근원인 입출력 장치로 부터 물리적인 입출력을 줄여 주므로서 다중 프로세서의 처리 능력의 증가에 비례하여 시스템 성능을 높일 수 있다. 그러나 입출력 요청율이 높으면 결국 물리적 입출력 빈도가 많아져서 성능 저하의 원인이 되며 이것은입출력 시스템이 갖는 근본적인 문제점이라고 볼때 앞으로 이의 해결 [9] 을 위한 연구가 있어야 한다.

參 考 文 獻

[1] D.A. Patteson, and J.L. Hennessy, "Computer Architecture A Quantitative Approach," Morgan Kaufmann Publishers Inc., 1990.

[2] Janaki Akella, Daniel P.Siewiorek, "Modeling and Measurement of the Impact of Input/ Output on System Performance, "Computer Architecture Conference Proceeding, pp 390-399, 1992.

[3] A. L. Narasimba Reddy, "A Study of I/O System Organization," Computer Architecture Conference Proceeding, pp 308-317, 1992.

[4] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. 17th Ann. Int. Symp. on Computer Architecture*, May 1990.

[5] Ron Wilson, "Designers rescue superminicomputers from I/O bottleneck,"*Computer Design*, pp 61 71, Oct. 1987.

[6] Ron Wilson, "designers seek new approaches to open I/O bottlenecks, "*Computer Design*, pp 57-73, Oct. 1988.

[7] Randy H. Katz et al., "Disk System Architectures for High Performance Computing," *Proceeding of The IEEE*, vol. 77, no.12, pp.1842-1858, Dec. 1989.

[8] Alan J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations, "ACM Transactions on Computer Systems, vol.3, no.3, pp 161-203, August. 1985.

[9] Peter M. Chen, David Patterson, "Maximizing Performance in Striped Disk Array," *Proc. 17th Ann. Int. Symp.* on Computer Architecture, May 1990.

[10] Ed Gould, "Device Drivers in a Multiprocessor Environment," USENIX, 1985.

[11] Gary Graunke and Shreekant Thakkar, "Synchronization Algorithms performance Shared-Memory Multiprocessors," Computer, pp 60-69, June 1990.

[12] A. Moolenaar, "Transferring UNIX I/O to I/O Processors, "Delft Univ., Jul. 1985.

著 者 紹 介

尹 龍 鎬 (正會員)
1949年 10月 15日生. 1975年 2月 한양대학교 전자공학과 졸업(공학사). 1981年 2月 연세대학교 전자계산학과 졸업(공학석사). 1988年 9月 한양대학교 전자공학과 박사과정 수료. 1993年 2月 현재 한국전자통신연구소 책임연구원. 주관심분야는 컴퓨터구조, 병렬처리, ASIC 설계, CAD 등임.

林 寅 七 (正會員) 第29卷 A編 第8號 參照
현재 한양대학교 전자공학과 교수