대용량의 InfiniBand 기반 DVSM 시스템 구현을 위한 성능 요구 분석

조 명 진[†] · 김 선 욱[††]

요 약

지난 수년간 저가의 공유메모리(Shared Memory) 시스템을 개발하기 위한 방법으로 빠른 상호 연결 네트워크를 이용한 DVSM (Distributed Virtual Shared Memory) 시스템의 구조에 관한 연구가 활발하게 진행되어 왔다. 그러나 DVSM은 소프트웨어 적으로 메모리 일관성을 유지하기 때문에 분산 처리 노드간의 많은 데이터 및 제어 신호 통신이 요구되며 이러한 통신 과부하(overhead)가 전체 성능 향상을 결정짓는 요인으로 작용한다. 일반적으로 프로세싱 노드의 수가 증가하면 통신 과부하도 따라서 증가하기 때문에 통신 과부하는 대용량(large-scale)의 DVSM을 구현하는데 매우 중요한 성능 요인이다. 이 논문에서는 차세대 상호 연결 기술 중 하나인 InfiniBand를 기반으로 대용량 DVSM 시스템을 구현하기 위한 성능 확장성을 정량적 및 정성적으로 연구하였다. 또한 이 연구를 바탕으로 성능 확장성이 뛰어난 DVSM 시스템을 개발하기 위한 차세대 상호 연결 네트워크의 요구 성능을 분석하였다.

키워드 : 분산 가상 공유 메모리, 인피니밴드, FESI 프로토콜, 성능분석

Analysis of Performance Requirement for Large-Scale InfiniBand-based DVSM System

Myeong-jin Cho[†] · Seon Wook Kim[††]

ABSTRACT

For past years, many distributed virtual shared-memory(DVSM) systems have been studied in order to develop a low-cost shared memory system with a fast interconnection network. But the DVSM needs a lot of data and control communication between distributed processing nodes in order to provide memory consistency in software, and this communication overhead significantly dominates the overall performance. In general, the communication overhead also increases as the number of processing nodes increase, so communication overhead is a very important performance factor for developing a large-scale DVSM system. In this paper, we study the performance scalability quantitatively and qualitatively for developing a large-scale DVSM system based on the next generation interconnection network, called the InfiniBand. Based on the study, we analyze a performance requirement of the next-coming interconnection network to be used for developing a performance-scalable DVSM system in the future.

Key Words : Distributed virtual shared memory, InfiniBand, FESI protocol, Performance analysis

1. 서 론

저가의 고성능 컴퓨터 시스템을 구성하기 위한 장래성 있는 방법 중 하나는 몇 개의 독립적 프로세싱 노드를 고속의 네트워크로 연결하는 것이다. 이러한 시스템을 분산 메모리 시스템이라고 한다. 그 대표격인 클러스터는 단지 1/50 내지 1/10 정도의 비용만으로 동일한 성능의 슈퍼컴퓨터를 구현할 수 있다[1]. 그러나 이러한 시스템에서는 프로그램을 개발하기가 쉽지 않다. 왜냐하면 프로그래머가 프로세싱 노드간의 데이터를 언제, 누구와 어떤 데이터를 공유해야 하는지를 프로그램 상에서 하나하나 명확히 표현해야 하기 때문이다. 게다가 분산 메모리 시스템에서는 프로그램을 개발하는 것뿐만 아니라 프로그램을 최적화 하거나 디버깅하는 것 역시 쉽지 않다.

이러한 문제점을 해결하기 위하여 하드웨어로 메모리 일

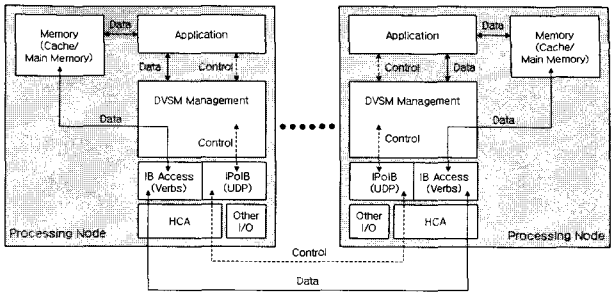
※ 이 논문은 2004년도 한국학술진흥재단의 지원에 의하여 연구되었음 (KRF-2004-003-D00304).
이 논문의 선행 연구 논문은 한국정보과학회 HPC 연구회 추계학술발표대회 제17권 2호에 게재되었음.
† 정 회 원 : 고려대학교 전자전기공학과 석사과정
†† 종신회원 : 고려대학교 전기전자전파공학과 부교수
논문접수 : 2007년 4월 13일, 심사완료 : 2007년 5월 3일

관성(memory consistency)을 제공하는 공유 메모리 시스템(shared-memory system)이 제안되었다. 이러한 시스템에서는 OpenMP[2]나 Pthread와 같은 강력하면서도 손쉬운 병렬 프로그래밍 환경을 사용할 수 있다. 그러나 하드웨어로 공유메모리를 구현하는 경우는 확장성에 제한이 있으며 가격이 매우 비싸다는 단점이 있다. 고가의 개발 비용이 소요되는 단점을 극복하기 위하여 소프트웨어적으로 분산 메모리 시스템에서 공유 메모리 프로토콜을 지원하는 DVSM (Distributed Virtual Shared Memory) 시스템이 제안되고 있다[3,4,5]. 그러나 DVSM 시스템에서 소프트웨어적으로 메모리 일관성을 유지하기 위하여 많은 통신 과부하를 발생시키며 이러한 통신 성능이 전체 성능향상을 결정짓는 요인으로 작용한다. 이러한 사실은 저가의 대용량 공유 메모리 시스템을 개발하는데 큰 장애가 되고 있다.

Ethernet과 같은 전통적인 통신 방식은 데이터 통신을 할 때 데이터 요청 프로세스가 데이터 제공 프로세스에게 인터럽트를 발생시킨다. 일반적으로 이러한 인터럽트는 프로세스간의 전체 성능을 저하시키는 부하 불균형(load imbalance)을 일으키기 때문에 이러한 문제를 해결하기 위하여 InfiniBand 구조(IBA)[6]와 같은 차세대 상호 연결 기술에 대한 요구가 크게 증가하고 있다. IBA의 경우 RDMA(Remote Direct Memory Access)를 지원하며 RDMA를 통해 소프트웨어의 간섭이 없는 메모리 트랜잭션(RDMA read/write 및 atomic 연산)이 가능하다. DVSM 시스템에서 IBA의 RDMA를 사용하면 프로세스 간의 통신을 위한 프로세스 인터럽트를 하지 않고도 다른 프로세스의 메모리 영역을 접근할 수 있으므로 메모리 일관성을 위한 많은 과부하를 줄일 수 있다. 또한 RDMA와 atomic 연산은 공유메모리 시스템에서의 프로그래밍 개념과도 매우 일치한다.

이 논문에서는 다양한 DVSM 과부하 분석을 통하여 대용량의 InfiniBand 기반 DVSM 시스템의 성능 확장성(performance scalability)을 정량적 및 정성적으로 연구하였다. 그리고 본 연구를 위하여 단순한 분석적 모델을 소개하고 상태 정보 교환 기법과 페이지 Prefetch[7]와 같은 몇 가지 최적화 기술을 적용하여 그 결과를 고찰해 보았다. 또한 이러한 연구를 바탕으로 추후 성능 확장성이 뛰어난 DVSM(performance-scalable DVSM) 시스템을 개발하기 위해 사용될 차세대 상호 연결 네트워크의 요구 성능을 분석하였다. 본 연구진은 DVSM 시스템 개발 및 성능 분석을 위하여 FESI라는 InfiniBand 기반의 공유 메모리 프로토콜을 개발하였다. FESI 프로토콜은 공유 메모리 일관성 프로토콜(shared-memory coherence protocol)과 유사하며 컨트롤 신호와 데이터 트랜잭션을 위해 각각 UDP와 RDMA read/write를 사용한다.

본 논문은 다음과 같이 구성되어 있다. 제 2장에서는 본 연구진이 개발한 실험적인 DVSM 구조에 대하여 소개하고, 제 3장에서는 DVSM 시스템의 성능을 정량적 및 정성적으로 분석한다. 또한 차세대 대용량 DVSM 시스템을 위한 성능 요구조건을 분석하고 마지막으로 제 4장에서 결론을 맺고자 한다.



(그림 1) 본 연구를 위한 InfiniBand DVSM의 구조

2. 실험적인 InfiniBand 기반 DVSM

2.1 전체 구조

(그림 1)은 본 연구진이 개발한 실험적인 InfiniBand 기반 DVSM 시스템의 전체적인 모습을 보여주고 있다. 트랜잭션은 컨트롤 트랜잭션(control transaction)과 데이터 트랜잭션(data transaction)으로 나뉘는데, 이렇게 트랜잭션을 분리한 것은 우리가 제안한 DVSM 프로토콜에 존재하는 과부하를 각각 자세하게 연구 할 수 있게 해주기 때문이다. Verb 레이어는 데이터 트랜잭션을 위한 RDMA의 접근을 위해 사용하고, IPoIB 레이어(UDP)는 컨트롤 트랜잭션을 위해 사용되며 두 레이어는 모두 IBA에 의해 제공된다.

[8]과 달리 메모리는 어플리케이션에서 사용하는 메인 메모리와 각 프로세싱 노드간의 데이터 전송 에 사용되는 InfiniBand 하드웨어의 RDMA 메모리 두 가지로 구분된다. 어플리케이션을 실행하기위하여 각 프로세싱 노드의 메인 메모리를 우선적으로 사용하였고 다른 프로세스와 통신이 필요한 경우에 한해 임시적으로 RDMA 메모리에 데이터를 복사하여 사용 하였다. 이와 같은 방식은 RDMA의 메모리를 어플리케이션용 메모리로 사용하였을 경우 RDMA 메모리 크기보다 더 큰 공유 메모리를 필요로 하는 경우의 문제점을 해결하기 위해서이다[9].

2.2 메모리 전송 프로토콜

이 절에서는 하드웨어 공유 메모리 시스템 상의 캐시 일관성 프로토콜과 비슷한 LRC(Lazy Release Consistency)[15] 기반 FESI DVSM 프로토콜을 소개한다[10]. 본 연구에서 소개하는 FESI 프로토콜의 상태 전이 다이어그램은 (그림 2)와 같다.

2.2.1 프로세스간의 프로토콜

FESI 프로토콜에서 각 페이지의 상태는 Invalid(I), Shared(S), Exclusive(E), Protected Shared(PS) 그리고 Floating(F)의 다섯 가지 상태로 정의되며, 그 의미는 다음과 같다.

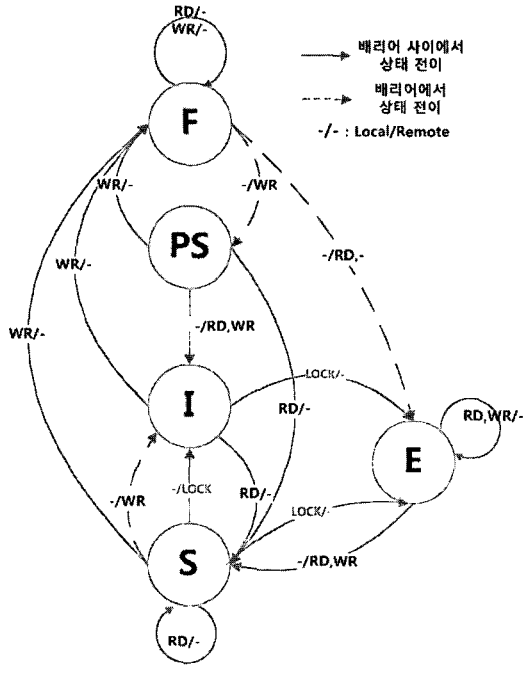
Invalid (I) : 해당 페이지의 정보가 더 이상 유효하지 않는 상태이다. 이 경우 프로세스가 페이지에 접근할 때 반드시 다른 프로세스로부터 유효한 페이지를 획득해야만 한다.

Shared (S) : 해당 페이지는 유효하며 페이지가 여러 프로세스와 공유 되는 상태이다. 페이지에 쓰기를 시도할 경우 현재의 페이지 정보를 기준으로 수정한 데이터(diff)를 기록하기 위하여 메모리 쓰기 보호가 필요하다. 만약 자신의 프로세스가 이 상태를 유지하고 있으나 다른 프로세스가 해당 페이지를 수정하였다면 배리어 시점에서 Invalid 상태가 된다.

Protected Shared (PS) : 여러 프로세스가 같은 시간대에 같은 페이지에 쓰기를 하였을 경우 발생하는 상태이다. 이 상태에서는 수정의 기준이 되는 페이지와 해당 페이지의 수정된 데이터 즉, diff를 가지고 있다. PS 상태가 존재하는 경우 유효한 페이지를 얻기 위해서는 모든 PS 상태의 프로세스로부터 diff 정보를 받아서 최신의 페이지를 작성하여야만 한다.

Exclusive (E) : 하나의 프로세스가 유일하게 유효한 페이지를 가지고 있고 또한 독점적으로 사용 될 때 발생한다. Exclusive 상태가 존재하면 다른 모든 프로세스의 상태는 Invalid 상태가 된다. 이 상태는 혼자서 페이지를 독점하므로 페이지 접근이 무한히 자유롭다. 단, 다른 프로세스가 해당 페이지를 접근하게 되면 자신이 작성하고 있던 페이지를 즉시 Shared 상태로 만들어 다른 프로세스와 공유하게 된다.

Floating (F) : 동기화(Synchronization) 사이에서만 존재하는 임시적인 상태이다. 프로세스가 페이지 정보를 수정할 때 임시적으로 Floating 상태가 된다. 배리어 시점에서 원격 프로세스의 행동에 따라 다음과 같이 상태가 변화한다. 만약 다른 프로세스에서도 페이지 정보를 수정하였다면 Protected Shared 상태로 전이되고 그렇지 않다면 Exclusive 상태로 전이된다.



(그림 2) FESI 프로토콜의 상태 전이 다이어그램

위와 같은 다섯 가지의 메모리 상태 외에도 두개의 원격 신호가 정의되어 있다.

Page Request : 유효한 페이지를 원하는 프로세스는 다른 프로세스들에게 최신 페이지를 요청한다. 페이지를 요청할 때 어느 프로세싱 노드가 어떤 페이지에 대한 최신 정보를 요구하는지를 페이지 요청(Page Request) 신호에 실어서 다른 모든 노드에 보낸다. 이때 쓰기를 통해 발생한 요구는 PAGE_WRITE, 읽기를 통해 발생한 페이지 요구는 PAGE_READ로 각각 구분하여 보낸다. 이에 페이지 요청에 대한 응답(Page Grant)이 도착하면 RDMA를 통해 유효한 페이지와 필요한 경우 diff 정보를 가져와 최신 페이지를 조합한다.

Page Grant : 페이지 요청에 대한 응답으로 Invalid 상태의 프로세스는 PAGE_INVALID 신호를 보내 자신에게 유효한 페이지가 없음을 알린다. 반면 유효한 페이지를 가지고 있는 경우에는 일단 유효한 페이지 정보를 자신의 RDMA 메모리에 복사를 하고 RDMA 주소 정보를 PAGE_GRANT 신호에 실어 보낸다. Protected Shared 상태인 경우에는 diff 정보를 가지고 있기 때문에 diff 정보도 페이지와 함께 보내어 diff를 통해 유효한 페이지를 조합할 수 있도록 한다.

2.2.2 프로세스간의 프로토콜 : 메인 메모리와 RDMA 메모리 사이의 일관성

[5, 8]과 달리 본 연구진이 구현한 DVSM 시스템은 어플리케이션을 실행하기 위하여 각 노드의 메인 메모리를 작업 공간(working space)으로 사용하고 다른 프로세스와 데이터를 교환할 경우에만 HCA의 내부 메모리, 즉 RDMA 메모리를 사용한다. 이러한 경우, 메인 메모리는 항상 최신의 페이지 정보를 유지하지만 RDMA 메모리는 데이터를 교환할 때만 사용되므로 항상 유효한 상태를 유지하지는 못한다. 그런데 외부 프로세스는 RDMA 메커니즘을 통하여 메인 메모리가 아닌 RDMA 메모리만 접근이 가능하므로 메인 메모리와 RDMA 메모리간의 일관성을 유지하기 위한 메커니즘이 필요하다.

일관성 메커니즘은 다음과 같다. 만약 메인 메모리에 RDMA 메모리보다 더 최신의 페이지를 가지고 있고 다른 프로세스로부터 페이지 요청을 받은 경우, 프로세스는 RDMA 메모리에 유효한 페이지를 복사한다. 이와 달리 메인 메모리와 RDMA 메모리가 같은 정보를 가지고 있다면 일관성을 유지하기 위한 트랜잭션은 필요하지 않다.

3. 성능 확장성 분석

본 연구에서는 Intel 2.0GHz Xeon Processor와 512MB의 메인 메모리, 그리고 133MHz 4X PCI-X 128MB InfiniBand HCA 카드[11]를 가진 8개의 프로세싱 노드를 이용하여 DVSM 시스템 성능을 측정하였다. 성능 측정에는 SPEC OpenMP 벤치마크[12] 프로그램을 사용하였다.

분석적 모델의 speedup을 측정하기 위하여 수식 (1)과 같은 Amdahl의 법칙[13]을 사용하였다. 수식 (1)의 각 요소를 살펴보면, T_{serial}과 T_{par}은 응용 프로그램의 직렬영역(serial region)과 병렬영역(parallel region)을 처리하는데 소요되는 시간이고 T_{comm}은 메모리 일관성을 위한 통신 과부하이다. 또한 $T_{barrier}$는 프로세스간의 동기화를 위한 배리어 시간이며 N_{proc}은 프로세싱 노드의 수를 나타낸다.

$$Speedup = \frac{T_{serial} + T_{par}}{\frac{T_{par}}{N_{proc}} + T_{serial} + T_{comm} + T_{barrier}} \quad (1)$$

T_{comm}은 페이지의 상태를 올바르게 유지하기 위한 처리 시간으로 상태 변화량과 상태 변화 시에 소요되는 시간으로 표현될 수 있다. T_{comm}을 수식화하면 수식 (2)와 같다.

$$\begin{aligned} T_{comm} = {} & T_{(I\to S)} \times N_{(I\to S)} + T_{(I\to F)} \times N_{(I\to F)} + T_{(S\to I)} \times N_{(S\to I)} \\ & + T_{(S\to F)} \times N_{(S\to F)} + T_{(PS\to I)} \times N_{(PS\to I)} + T_{(PS\to F)} \times N_{(PS\to F)} \\ & + T_{(PS\to S)} \times N_{(PS\to S)} + T_{(F\to PS)} \times N_{(F\to PS)} + T_{(F\to E)} \times N_{(F\to E)} \\ & + T_{(E\to S)} \times N_{(E\to S)} \end{aligned} \quad (2)$$

수식 (2)에서 $T_{(X\to Y)}$와 $N_{(X\to Y)}$는 각각 X 상태에서 Y 상태로 변할 때 소요되는 시간과 총 상태 변화량을 나타낸다. 그런데 $S\to F$, $F\to PS$, $F\to E$, $E\to S$, $S\to I$, $PS\to I$와 같은 상태 전이는 자신이 유효한 페이지를 갖고 있거나 유효하지 않은 상태로 전이되므로 프로세스 간의 통신이 필요 없게 된다. 그러므로 위 6가지 경우의 처리 시간은 이상적으로 0이라 할 수 있으며, 이를 통해 수식 (2)를 좀 더 단순화하면 다음과 같은 수식 (3)을 얻을 수 있다.

$$\begin{aligned} T_{comm} = {} & T_{(I\to S)} \times N_{(I\to S)} + T_{(I\to F)} \times N_{(I\to F)} \\ & + T_{(PS\to F)} \times N_{(PS\to F)} + T_{(PS\to S)} \times N_{(PS\to S)} \end{aligned} \quad (3)$$

DVSM 프로토콜은 소프트웨어적으로 구현되어 있기 때문에 통신 과부하는 전체 성능을 결정짓는 요소로 작용하고 있다. 이러한 이유로 대용량 DVSM 시스템을 구축함에 있어서 통신 성능에 대한 요구 조건을 이해하는 것은 매우 중요하다. 다음 절에서 본 연구진이 개발한 세 가지 시나리오를 소개하고 FESI 프로토콜의 과부하를 정량적 및 정성적으로 상세히 분석하고자 한다. 첫 번째는 버스 기반의 snoop 프로토콜과 같이 다른 모든 프로세싱 노드에 데이터를 요청하는 방식이다. 두 번째는 어떤 프로세스가 유효한 페이지를 가지고 있는지를 알고 해당 노드에만 데이터를 요청하는 방식이다. 이 경우는 디렉터리 기반의 캐시 일관성 프로토콜과 유사하다. 마지막으로 어떤 페이지가 다른 프로세스에 의해 빈번하게 접근 되어질 때, 프로세스가 실제로 데이터를 필요로 하기 이전에 페이지를 미리 RDMA 메모리에 가져다 놓는 방식이다.

3.1 Case 1 : 모든 프로세싱 노드에 데이터를 요청하는 경우

Case 1에서는 프로세스가 유효한 페이지를 얻기 위해 자신을 제외한 모든 프로세싱 노드에 페이지 요청 신호를 보낸다. 이 경우 $T_{(X\to Y)}$는 다음과 같이 정의 할 수 있다.

$$T_{(X\to Y)} = T_{multicast} + (N_{proc} - 1) \times T_{ack} + T_{rdma} \quad (4)$$

여기서 $T_{multicast}$는 멀티캐스트로 페이지 요청을 보내는데 걸리는 시간이고 T_{ack}는 페이지 요청에 대한 응답이 되돌아오는데 걸리는 시간이다. 또한 T_{rdma}는 RDMA를 사용하여 유효한 데이터를 가져오는데 소요되는 시간이다. 그런데 이상적인 환경에서 멀티캐스트를 보내는 시간과 그에 대한 응답(ACK)을 받는 시간은 동일하다. 따라서 수식 (4)는 간단하게 수식 (5)와 같이 표현될 수 있다.

$$T_{(X\to Y)} = N_{proc} \times T_{ack} + T_{rdma} \quad (5)$$

수식 (5)를 수식 (3)에 대입하여 식을 정리하면 수식 (6)이 된다. 수식 (6)을 살펴보면 통신 과부하 T_{comm}은 $(N_{(I\to S)} + N_{(I\to F)} + N_{(P\to F)} + N_{(P\to S)}) \times (N_{proc} \times T_{ack})$으로 표현되는 컨트롤 신호를 다루기 위한 컨트롤 과부하(control overhead)와 $(N_{(I\to S)} + N_{(I\to F)} + N_{(P\to F)} + N_{(P\to S)}) \times T_{rdma}$로 표현되는 RDMA 트랜잭션을 위한 데이터 이동 과부하(data movement overhead)로 나눌 수 있다.

〈표 1〉 벤치마크별 프로세스 수의 변화에 따른 state 변화량

(단위 : 회)

Benchmark	# of Procs	I→S	I→F	PS→S	PS→F	Total
SWIM	2	20939	71577	54	89	92659
	4	31448	95217	54	89	126808
	8	36643	107042	54	89	143828
WUPWISE	2	8587	17701	183	1	26472
	4	8598	15013	183	1	23795
	8	8594	13669	183	1	22447
APPLU	2	33432	4905	392	5086	43815
	4	44111	107754	815	54315	206995
	8	27151	33946	740	62781	124618
MGRID	2	30284	8086	2216	447	41033
	4	41346	4359	2018	509	48232
	8	46354	2661	1590	379	50984

〈표 2〉 벤치마크별 프로세스 수의 변화에 따른 T_{comm} 값

(단위 : μsec)

Benchmark	# of Procs	I→S	I→F	PS→S	PS→F	Total
SWIM	2	2512680	8589240	6480	10680	11119080
	4	6918560	20947740	11880	19580	27897760
	8	15390060	44957640	22680	37380	60407760
WUPWISE	2	1030440	2124120	21960	120	3176640
	4	1891560	3302860	40260	220	5234900
	8	3609480	5740980	76860	420	9427740
APPLU	2	4011840	588600	47040	610320	5257800
	4	9704420	23705880	179300	11949300	45538900
	8	11403420	14257320	310800	26368020	52339560
MGRID	2	3634080	970320	265920	53640	5519947
	4	9096120	958980	443960	111980	11663143
	8	19468680	1117620	667800	159180	23284972

$$T_{comm} = (N_{(I\to S)} + N_{(I\to F)} + N_{(PS\to F)} + N_{(PS\to S)}) \times (N_{proc} \times T_{ack}) + (N_{(I\to S)} + N_{(I\to F)} + N_{(PS\to F)} + N_{(PS\to S)}) \times T_{rdma} \quad (6)$$

여기에서 T_{rdma}는 InfiniBand 하드웨어 장비에 의해 결정되며 데이터 크기에 따른 시간 변화가 거의 없는 상수 값으로 본 연구에서 사용된 장비에서는 약 20μsec의 시간이 소요된다. 또한 T_{ack}의 경우에도 실제로는 네트워크 상황에 따라 변하는 변수이지만 하나의 네트워크 장비에 연결된 클러스터 환경이므로 대략적으로 일정한 시간이 소요된다고 할 수 있다. 이 논문에서는 이 값을 50μsec로 가정한다. 따라서 수식 (6)에서 변수는 $N_{(X\to Y)}$와 N_{proc} 만이 남게 된다. 이에 프로세스 수의 변화에 따른 상태의 변화량을 측정하고 측정된 값을 통해 T_{comm}을 계산한 결과는 각각 <표 1>, <표 2>와 같다.

DVSM 시스템의 speedup을 구하기 위해서는 T_{comm} 뿐만이 아니라 T_{serial}, T_{par}, $T_{barrier}$와 같은 변수의 값도 알아야 한다. 이에 T_{serial}, T_{par}를 구하기 위해 각 벤치마크의 병렬화 정도(Parallelism)[14]를 구해보면 <표 3>과 같고 $T_{barrier}$는 <표 4>와 같다.

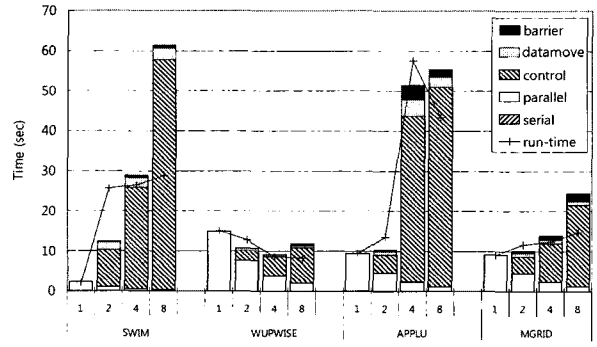
〈표 3〉 각 벤치마크 별 병렬화 정도

Benchmark	병렬영역	직렬영역
SWIM	99.5 %	0.05 %
WUPWISE	99.8 %	0.02 %
APPLU	99.9 %	0.01 %
MGRID	99.9 %	0.01 %

〈표 4〉 각 벤치마크 별 프로세싱 노드 수의 변화에 따른 멀티캐스트 방식의 $T_{barrier}$ 값

(단위 : μsec)

Benchmark	P=2	P=4	P=8
SWIM	327958	524650	686560
WUPWISE	111665	205313	586158
APPLU	401468	3595419	1957550
MGRID	589584	1028960	1863937



(그림 3) 전체 프로세스에 데이터를 요청하는 경우의 실행 시간

(그림 3)은 Case 1의 각 벤치마크의 분석적인 실행 시간과 실제 측정 시간을 나타낸다. 범례에서 "barrier"는 배리어 처리시간, "datamove"는 데이터 이동 과부하, "control"은 컨트롤 과부하, "serial"과 "parallel"은 각각 직렬 영역과 병렬 영역을 처리하는데 소요되는 시간이다. 마지막으로 "runtime"은 실제 실행 시간을 의미한다. 그래프에서 볼 수 있듯이 대부분의 벤치마크에서 컨트롤 과부하가 전체 실행 시간을 결정짓는다. 따라서 확장 가능한 DVSM 시스템을 개발하기 위해서는 반드시 컨트롤 과부하를 줄여야 한다는 것을 잘 보여준다. 또한 본 연구진이 구현한 분석적 모델이 실제 측정된 값과 비교하여 충분히 합당함을 보여준다.

3.2 Case 2 : 특정 프로세싱 노드에 데이터를 요청하는 경우

앞서 Case 1에서 데이터 요청 프로세스가 불필요한 데이터 요청 신호를 보내므로 해서 심각한 성능 저하가 발생하였다. 이러한 문제를 해결하기 위하여 유효한 페이지를 가지고 있는 프로세스에만 데이터를 요청하는 메커니즘이 필요하며 Case 2에서는 배리어 시점에서 상태 정보를 공유하는 방법으로 이를 구현하였다. 이와 같은 방법은 상태 정보를 공유한다는 점을 제외하고는 디렉터리 기반의 캐시 일관성 프로토콜과 유사하다.

Case 2에서 T_{comm}은 다음과 같이 정의 될 수 있다.

$$T_{comm} = T_{ack} \times \sum_{k}(k+1)N_k + T_{rdma} \times \sum_{k} N_k \quad (7)$$

여기서 N_k는 데이터를 요청할 때 유효한 데이터를 가진 노드가 k개인 경우의 수이다. 또한 앞에서 언급했듯이 데이터 요청 신호와 그 응답은 같은 시간이 걸리므로 $(k+1)\times N_k$로 계산하였다. 이에 N_k를 구해보면 <표 5, 6, 7, 8>과 같다.

<표 5, 6, 7, 8>의 결과를 보면 하나 혹은 두 개의 프로세스가 유효한 페이지를 가지는 경우가 대부분임을 알 수

〈표 5〉 SWIM에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	20939	0	20939	0	0	0	0	0	0
	I→F	71577	12195	59382	0	0	0	0	0	0
	PS→S	54	0	54	0	0	0	0	0	0
	PS→F	89	0	89	0	0	0	0	0	0
	Total	92659	12195	80464	0	0	0	0	0	0
4	I→S	31448	0	31386	62	0	0	0	0	0
	I→F	95217	6099	88940	178	0	0	0	0	0
	PS→S	54	0	54	0	0	0	0	0	0
	PS→F	89	0	89	0	0	0	0	0	0
	Total	126808	6099	120469	240	0	0	0	0	0
8	I→S	36643	0	36457	186	0	0	0	0	0
	I→F	107042	3051	103457	534	0	0	0	0	0
	PS→S	54	0	54	0	0	0	0	0	0
	PS→F	89	0	89	0	0	0	0	0	0
	Total	143828	3051	140057	720	0	0	0	0	0

〈표 6〉 WUPWISE에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	8587	0	8587	0	0	0	0	0	0
	I→F	17701	17669	32	0	0	0	0	0	0
	PS→S	183	0	183	0	0	0	0	0	0
	PS→F	1	0	1	0	0	0	0	0	0
	Total	26472	17669	8803	0	0	0	0	0	0
4	I→S	8598	0	8598	0	0	0	0	0	0
	I→F	15013	14981	32	0	0	0	0	0	0
	PS→S	183	0	183	0	0	0	0	0	0
	PS→F	1	0	1	0	0	0	0	0	0
	Total	23795	14981	8814	0	0	0	0	0	0
8	I→S	8594	0	8594	0	0	0	0	0	0
	I→F	13669	13637	32	0	0	0	0	0	0
	PS→S	183	0	183	0	0	0	0	0	0
	PS→F	1	0	1	0	0	0	0	0	0
	Total	22447	13637	8810	0	0	0	0	0	0

〈표 7〉 APPLU에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	33432	0	33432	0	0	0	0	0	0
	I→F	4905	4483	422	0	0	0	0	0	0
	PS→S	392	0	392	0	0	0	0	0	0
	PS→F	5086	0	5086	0	0	0	0	0	0
	Total	43815	4483	39332	0	0	0	0	0	0
4	I→S	44111	0	37949	5129	1033	0	0	0	0
	I→F	107754	2255	105459	40	0	0	0	0	0
	PS→S	815	0	812	3	0	0	0	0	0
	PS→F	54315	0	53195	560	560	0	0	0	0
	Total	206995	2255	197415	5732	1593	0	0	0	0
8	I→S	27151	0	22023	3621	1066	411	30	0	0
	I→F	33946	1085	32828	33	0	0	0	0	0
	PS→S	740	0	733	7	0	0	0	0	0
	PS→F	62781	0	61661	1120	0	0	0	0	0
	Total	124618	1085	117245	4781	1066	411	30	0	0

〈표 8〉 MGRID에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

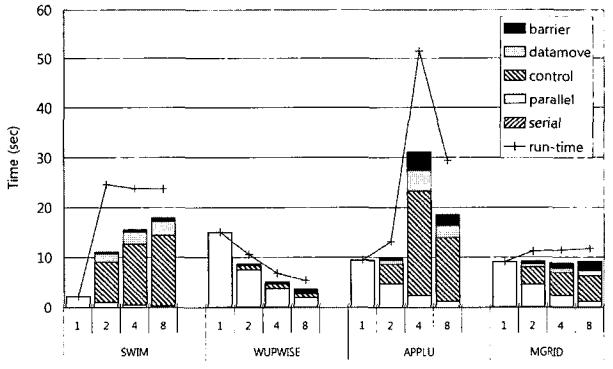
(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	30284	0	30284	0	0	0	0	0	0
	I→F	8086	7098	988	0	0	0	0	0	0
	PS→S	2216	0	2216	0	0	0	0	0	0
	PS→F	447	0	447	0	0	0	0	0	0
	Total	41033	7098	33935	0	0	0	0	0	0
4	I→S	41346	0	40448	738	160	0	0	0	0
	I→F	4359	3517	834	8	0	0	0	0	0
	PS→S	2018	0	1136	352	530	0	0	0	0
	PS→F	509	0	167	194	148	0	0	0	0
	Total	48232	3517	42585	1292	838	0	0	0	0
8	I→S	46354	0	44701	1275	153	100	75	0	50
	I→F	2661	1763	850	48	0	0	0	0	0
	PS→S	1590	0	1118	0	47	175	100	50	100
	PS→F	379	0	161	169	0	0	0	49	0
	Total	50984	1763	46830	1492	200	275	175	99	150

있다. 이 결과를 통해 벤치마크별 T_{comm} 값을 구해보면 <표 9>와 같다.

Speedup을 계산하기 위해서는 직렬 영역 처리 시간과 병렬 영역 처리시간은 Case 1에서 이미 계산하였기 때문에 $T_{barrier}$만 측정하면 된다. Case 2의 $T_{barrier}$는 Case 1에서 상태 정보를 공유하는 만큼의 과부하가 추가된 값이다.

측정 및 계산된 값을 바탕으로 Case 2에서 각 벤치마크의 분석적인 실행 시간과 실제 측정 시간을 나타내보면 (그림 4)와 같다.



(그림 4) 특정 프로세스에만 데이터를 요청하는 경우 실행 시간

〈표 10〉 각 벤치마크 별 State 테이블 방식의 $T_{barrier}$ 값

benchmark	P=2	P=4	P=8
SWIM	334077	534744	704200
WUPWISE	143443	260416	679863
APPLU	486790	3740975	2194895
MGRID	595987	1052103	1871692

<표 11>을 살펴보면 특정 프로세스에만 데이터를 요청함으로써 컨트롤 과부하가 눈에 띄게 감소하였음을 알 수 있다. 그러나 (그림 4)에 나타나듯이 Case 2 또한 여전히 많은 컨트롤 과부하가 존재한다.

〈표 11〉 각 벤치마크별 T_{comm} 절감 효과

노드의 수	SWIM	WUPWISE	APPLU	MGRID
2	13%	67%	10%	17%
4	48%	80%	45%	48%
8	72%	89%	71%	71%

3.3 Case 3 : 페이지 Prefetch 기법을 이용하는 경우

앞서 Case 2에서 특정 노드에만 데이터를 요청하는 방법으로 $T_{(X \to Y)}$ 값을 줄이고 이를 통해 성능 향상을 기대 할 수

〈표 9〉 벤치마크별 프로세스 수의 변화에 따른 T_{comm} 값

(단위 : μsec)

Benchmark	# of Procs	I→S	I→F	PS→S	PS→F	Total
SWIM	2	2512680	7125840	6480	10680	9655680
	4	3776860	10703060	6480	10680	14497080
	8	4406460	12505620	6480	10680	16929240
WUPWISE	2	1030440	3840	21960	120	1056360
	4	1031760	3840	21960	120	1057680
	8	1031280	3840	21960	120	1057200
APPLU	2	4011840	50640	47040	610320	4719840
	4	5653070	12661880	97950	6601800	25014700
	8	3613420	3944970	89150	7589720	15237260
MGRID	2	3634080	118560	265920	53640	4072200
	4	5014420	101440	312760	85580	5514200
	8	5686530	110160	284250	66180	6147120

있었다. 만약 $N_{(X\rightarrow Y)}$ 값도 함께 줄일 수 있다면 보다 많은 성능 향상을 기대할 수 있을 것이다. Case 3에서는 페이지 Prefetch 기법[7]을 사용하여 $N_{(X\rightarrow Y)}$ 값을 줄여보려고 한다. 여기서 페이지 Prefetch 기법은 빈번히 요청 되는 페이지를 프로세스가 필요로 하기 전에 미리 RDMA 메모리에 페이지를 가져다 놓는 방법이다.

3.3.1 완벽한 페이지 Prefetch

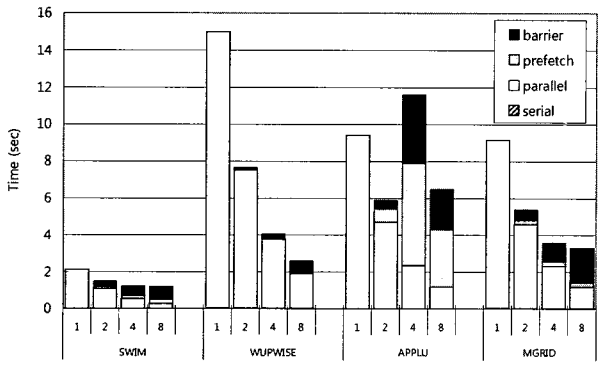
완벽한 페이지 Prefetch는 프로세스가 최신의 페이지를 필요로 할 때 RDMA 메모리로부터 언제나 유효한 페이지를 얻을 수 있는 경우를 말한다. 이 경우 페이지를 요청하기 위하여 다른 프로세스와 더 이상 원격 신호를 주고받을 필요가 없으므로 T_{comm}은 급격히 감소한다. 따라서 이 경우 앞선 Case 1, 2의 수식 (6, 7)과 달리 T_{comm}은 단지 Prefetch를 실행하는데 걸리는 시간인 $T_{prefetch}$로 표현할 수 있으며 $T_{prefetch}$는 $T_{rdma}\times N_{prefetch}$가 된다. 여기서 $N_{prefetch}$는 Prefetch를 실행한 횟수이다. (그림 5)는 완벽한 페이지 Prefetch를 하는 경우 실행 시간을 나타내며 APPLU를 제외한 나머지 모든 벤치마크에서 성능 향상이 있음을 보여준다. 그러나 실제로 완벽한 페이지 Prefetch는 불가능하기 때문에 (그림 5)의 결과는 우리가 기대할 수 있는 성능의 최대값을 나타낸다고 볼 수 있다.

3.3.2 간단히 구현된 페이지 Prefetch

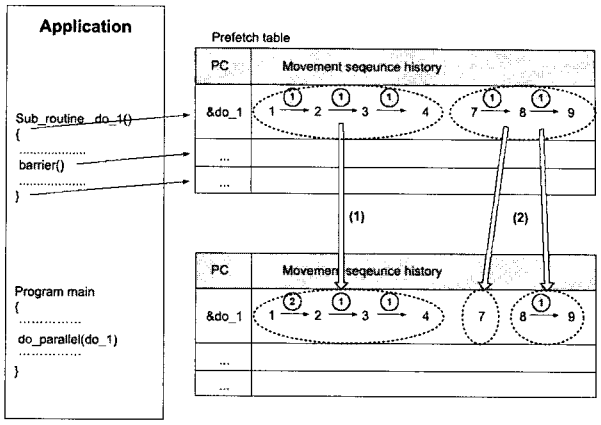
이에 다음과 같은 방법으로 간단히 페이지 Prefetch를 구현하여 보았다[8]. 동기화(synchronization) 사이마다 페이지 이동에 대한 순서 정보를 기록하고 있는 Prefetch 테이블을 유지한다. 페이지 이동이 발생하면 DVSM 시스템이 기록된 정보를 통하여 페이지 이동을 예상하여 페이지 요청에 의해 페이지 묶음을 가져온다.

(그림 6)을 통해 어떻게 Prefetch가 적용되는지 확인할 수 있다. Prefetch를 구현하기 위하여 각각의 병렬영역의 PC 값을 index로 하는 Prefetch 테이블을 구현한다.

1) 올바른 Prefetch의 경우 (그림 6의 (1)): 페이지 1에서 segv가 발생하였을 때, 페이지 1과 연결된 페이지 체인을 가져온다. 이때 원형안의 수로 표시된 Prefetch weight를



(그림 5) 완벽한 페이지 Prefetch를 사용한 경우의 실행 시간



(그림 6) 페이지 Prefetch 방법

고려하여 가져올 페이지 체인을 결정짓는다. 이 과정이 올바르게 동작을 하였을 경우에는 다음에 가져올 Prefetch의 weight를 증가시킨다.

2) 잘못된 Prefetch의 경우 (그림 6의 (2)) : 페이지 7에서 segv가 발생하였을 경우에, 1)에서와 마찬가지로 페이지 7과 연결된 페이지 체인을 가져온다. 그러나 확인 결과 Prefetch가 틀린 것으로 판단되면 다시 segv를 발생시킨다. 이때 페이지 체인의 weight를 감소시키거나 체인에서 분리시켜서 더 이상의 Prefetch 실패가 발생하지 않도록 한다.

일반적인 페이지 Prefetch의 경우에는 완벽한 페이지 Prefetch와 달리 Prefetch 실패가 발생하므로 유효한 데이터를 요청하기 위한 원격 신호의 전송이 필요하다. 따라서 T_{comm}을 정의하면 다음과 같다.

$$T_{comm} = T_{ack}\times\sum_{k}(k+1)\times N_k + T_{rdma}\times\sum_{k}N_k + T_{prefetch}\times N_{prefetch} \quad (8)$$

간단히 구현된 페이지 Prefetch에서는 $T_{prefetch}$가 대략적으로 $2(T_{ack}+T_{rdma})$로 나타내어진다. 여기서 두 번의 T_{ack}는 데이터를 요청하고 응답을 받는데 사용되고 한 번의 T_{rdma}는 페이지 묶음을 가져오는데 사용되며 마지막의 T_{rdma}는 Prefetch가 완료되었음을 데이터 제공자에게 알리는데 사용된다.

T_{comm}을 계산하기 위해서 $N_{prefetch}$와 N_k를 측정해보면 <표 12, 13, 14, 15, 16>과 같다.

〈표 12〉 각 벤치마크별 Prefetch 실행 횟수

(단위 : 회)

노드의 수	SWIM	WUPWISE	APPLU	MGRID
2	668	117	5022	2306
4	1063	128	39467	1915
8	1593	128	22250	1895

〈표 13〉 Prefetch를 적용하였을 때 SWIM에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	2719	0	2719	0	0	0	0	0	0
	I→F	19540	12195	7345	0	0	0	0	0	0
	PS→S	13	0	13	0	0	0	0	0	0
	PS→F	11	0	11	0	0	0	0	0	0
	Total	22283	12195	10088	0	0	0	0	0	0
4	I→S	4162	0	4100	62	0	0	0	0	0
	I→F	17388	6099	11111	178	0	0	0	0	0
	PS→S	14	0	14	0	0	0	0	0	0
	PS→F	11	0	11	0	0	0	0	0	0
	Total	21575	6099	15236	240	0	0	0	0	0
8	I→S	5003	0	4817	186	0	0	0	0	0
	I→F	16723	3051	13138	534	0	0	0	0	0
	PS→S	14	0	14	0	0	0	0	0	0
	PS→F	11	0	11	0	0	0	0	0	0
	Total	21751	3051	17980	720	0	0	0	0	0

〈표 14〉 Prefetch를 적용하였을 때 WUPWISE에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	1229	0	1229	0	0	0	0	0	0
	I→F	17682	17669	13	0	0	0	0	0	0
	PS→S	103	0	103	0	0	0	0	0	0
	PS→F	0	0	0	0	0	0	0	0	0
	Total	19014	17669	1345	0	0	0	0	0	0
4	I→S	1260	0	1260	0	0	0	0	0	0
	I→F	14994	14981	13	0	0	0	0	0	0
	PS→S	125	0	125	0	0	0	0	0	0
	PS→F	0	0	0	0	0	0	0	0	0
	Total	16379	14981	1398	0	0	0	0	0	0
8	I→S	1259	0	1259	0	0	0	0	0	0
	I→F	13650	13637	13	0	0	0	0	0	0
	PS→S	125	0	125	0	0	0	0	0	0
	PS→F	0	0	0	0	0	0	0	0	0
	Total	15034	13637	1397	0	0	0	0	0	0

〈표 15〉 Prefetch를 적용하였을 때 APPLU에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	5036	0	5036	0	0	0	0	0	0
	I→F	4828	4483	345	0	0	0	0	0	0
	PS→S	103	0	103	0	0	0	0	0	0
	PS→F	22	0	22	0	0	0	0	0	0
	Total	9989	4483	5506	0	0	0	0	0	0
4	I→S	11517	0	5352	5132	1033	0	0	0	0
	I→F	3275	2255	980	40	0	0	0	0	0
	PS→S	77	0	74	3	0	0	0	0	0
	PS→F	1132	0	12	560	560	0	0	0	0
	Total	16001	2255	6418	5735	1593	0	0	0	0
8	I→S	8332	0	3204	3621	1066	411	30	0	0
	I→F	1615	1085	497	33	0	0	0	0	0
	PS→S	71	0	64	7	0	0	0	0	0
	PS→F	1134	0	14	1120	0	0	0	0	0
	Total	11152	1085	3779	4781	1066	411	30	0	0

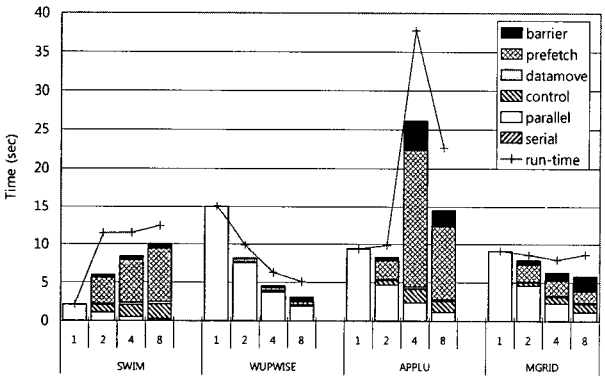
〈표 16〉 Prefetch를 적용하였을 때 MGRID에서 state 변화 시 유효한 데이터를 얻기 위해 필요한 노드의 수

(단위 : 회)

			유효한 데이터를 얻기 위해 필요한 노드의 수							
노드 수	상태변화	변화량	0	1	2	3	4	5	6	7
2	I→S	2692	0	2692	0	0	0	0	0	0
	I→F	7272	7098	174	0	0	0	0	0	0
	PS→S	1283	0	1283	0	0	0	0	0	0
	PS→F	86	0	86	0	0	0	0	0	0
	Total	11333	7098	4235	0	0	0	0	0	0
4	I→S	5035	0	4066	744	225	0	0	0	0
	I→F	3730	3517	205	8	0	0	0	0	0
	PS→S	1400	0	589	327	484	0	0	0	0
	PS→F	506	0	164	194	148	0	0	0	0
	Total	10671	3517	5024	1273	857	0	0	0	0
8	I→S	5035	0	4066	744	225	0	0	0	0
	I→F	3730	3517	205	8	0	0	0	0	0
	PS→S	1400	0	589	327	484	0	0	0	0
	PS→F	506	0	164	194	148	0	0	0	0
	Total	10671	3517	5024	1273	857	0	0	0	0

<표 13, 14, 15, 16>의 결과를 보면 Case 2의 경우와 마찬가지로 하나 혹은 두 개의 프로세스가 유효한 페이지를 가지는 경우가 대부분이다. 하지만 Case 2의 결과에 비해서 변화량이 현저히 줄어들었음을 알 수 있다. 이 결과를 통해 벤치마크별 T_{comm} 값을 구해보면 다음과 같다.

위의 측정 및 계산된 결과를 바탕으로 실행 시간 분석 그래프를 그려보면 (그림 7)과 같다.



(그림 7) 간단히 구현된 Prefetch를 사용하였을 경우 실행 시간

(그림 7)을 통해 알 수 있듯이 WUPWISE와 MGRID에서 성능향상이 있음을 알 수 있다. 그러나 비록 컨트롤 과부하는 현저히 줄어들었으나 여전히 많이 존재하며 무엇보다 Prefetch를 하기위한 과부하가 많이 증가하여 완벽한 페이지 Prefetch에는 못 미치는 성능을 나타내고 있다. 그 이유는 각 벤치마크별 Prefetch 적중률이 성능을 결정짓기 때문이다. 자세히 살펴보면 SWIM, WUPWISE, APPLU 그리고 MGRID에서 각각 30%, 90%, 10% 그리고 80%의 적중률을 보였으며 비교적 적중률이 높은 WUPWISE와 MGRID에서는 성능향상이 있었으나 반면 낮은 적중률을 보이는 SWIM과 APPLU에서는 성능향상을 얻지 못하였다. 따라서 더 나은 Prefetch 기법을 사용한다면 완벽한 페이지 Prefetch에 보다 가까운 성능 향상을 얻을 수 있으리라 기대한다. 또한 이 실험을 통해 페이지 Prefetch가 DVSM 시스템의 과부하를 줄이는데 매력적인 해결책임을 알 수 있다.

3.4 토의

본 연구에서 기준이 되는 네 개의 벤치마크는 모두 병렬화가 99% 이상으로 이론적으로는 <표 18>의 Ideal 경우와 같은 큰 성능향상을 얻을 수 있다. 그러나 실제로 구현된 DVSM 시스템에서는 많은 과부하가 존재하여 이상적인 성

〈표 17〉 벤치마크별 프로세스 수의 변화에 따른 T_{comm} 값

(단위 : μsec)

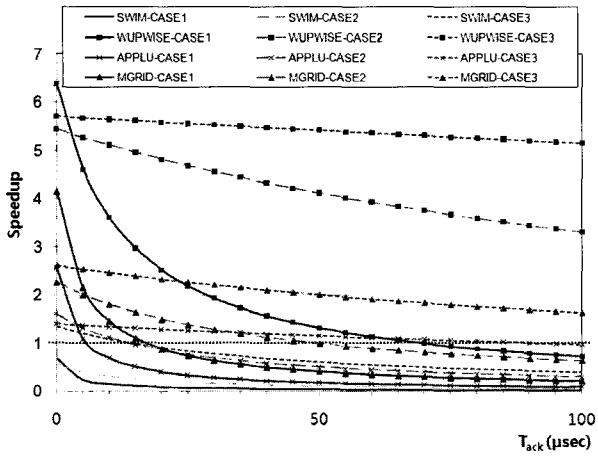
Benchmark	# of Procs	I→S	I→F	PS→S	PS→F	Total
SWIM	2	326280	881400	1560	1320	1210560
	4	502540	1363580	1680	1320	1869120
	8	609660	1667340	1680	1320	2280000
WUPWISE	2	147480	1560	12360	0	161400
	4	151200	1560	15000	0	167760
	8	151080	1560	15000	0	167640
APPLU	2	604320	41400	12360	2640	660720
	4	1741940	124400	9390	219840	2095570
	8	1355140	65250	8870	192080	1621340
MGRID	2	323040	20880	153960	10320	508200
	4	663900	25960	232750	85220	1007830
	8	881770	44040	202610	94630	1223050

〈표 18〉 세 가지 경우에서의 성능향상 비교

Benchmarks	Procs	Ideal	Case 1	Case 2	Case 3	
					Perfect	Simple
SWIM	1	1	1	1	1	1
	2	1.99	0.084	0.087	1.42	0.189
	4	3.94	0.081	0.09	1.736	0.188
	8	7.729	0.074	0.091	1.765	0.174
WUPWISE	1	1	1	1	1	1
	2	1.996	1.178	1.424	1.954	1.522
	4	3.976	1.747	2.207	3.703	2.396
	8	7.889	1.846	2.793	5.774	2.949
APPLU	1	1	1	1	1	1
	2	1.998	0.696	0.714	1.593	0.955
	4	3.988	0.163	0.183	0.808	0.249
	8	7.944	0.217	0.32	1.446	0.415
MGRID	1	1	1	1	1	1
	2	1.998	0.783	0.817	1.696	1.062
	4	3.988	0.736	0.801	2.549	1.147
	8	7.944	0.62	0.779	2.775	1.059

능향상을 얻지 못하고 있다.

연구진은 본 연구를 위하여 다음과 같은 세 종류의 DVSM 시스템을 구현하였다. 1) 모든 프로세스에 데이터 요청 신호를 보내는 시스템, 2) 배리어 시점에서 상태 정보를 교환하여 특정 프로세스에만 데이터를 요청하는 시스템, 3) 간단히 구현된 Prefetch를 통해서 유효한 페이지를 미리 RDMA 메모리에 가져다 놓는 시스템. 첫 번째 구현은 우리가 얻을 수 있는 최소한의 성능을 의미하며 특정 노드에만 데이터를 요청하여 불필요한 신호를 제거함으로써 두 번째 구현과 같은 성능 향상을 얻을 수 있었다. 그리고 우리가 얻을 수 있는 최대의 성능을 의미하는 완벽한 페이지 Prefetch와 이상적인 경우(Ideal)와의 성능 차이는 LRC에서의 페이지 diff의 조합으로 인한 과부하[8]와 완벽한 페이지 Prefetch를 위한 Prefetch 과부하로 인해 발생한 것임을 알 수 있다. 또한 완벽한 페이지 Prefetch와 간단히 구현된 Prefetch의 성능 차이는 컨트롤 과부하와 Prefetch 실패(miss)로 인하여 발생한 결과임을 알 수 있다.



(그림 8) 각 벤치마크별 T_{ack}의 변화에 따른 성능 확장성

각 벤치마크별로 살펴보면, SWIM의 경우는 완벽한 페이지 Prefetch의 경우에만 성능 향상을 보이고 있으며 Prefetch의 적중률이 성능향상을 결정짓고 있음을 알 수 있다. WUPWISE의 경우는 모든 경우에서 성능 향상을 이루고 있다. APPLU의 경우는 SWIM의 경우와 마찬가지로 완벽한 페이지 Prefetch에서만 성능향상을 보이고 있다. 그러나 APPLU는 동기화를 위한 과부하가 많기 때문에 다른 벤치마크와 달리 완벽한 페이지 Prefetch에서도 큰 성능 향상을 얻지는 못하고 있다. MGRID의 경우는 프로세싱 노드의 수가 증가하여도 성능의 차이가 거의 없는 것을 알 수 있다. 이는 프로세싱 노드의 수가 증가하여 생긴 병렬처리 이득과 과부하가 거의 비슷하기 때문이며 이는 최적화를 통해 성능향상을 얻을 수 있을 것이다.

마지막으로 우리의 분석적 모델을 사용하여 T_{ack}의 변화에 따른 확장성을 각 벤치마크별로 측정해 보았으며 그 결과는 (그림 8)과 같다. 앞에서 언급했듯이 우리는 클러스터 환경에서 T_{ack}를 50μsec로 가정하였다. (그림 8)에서 보이듯이 T_{ack}를 줄일 수 있다면 보다 많은 성능향상을 기대 할 수 있다. 그렇기 때문에 대용량 DVSM 시스템을 개발하기 위해서는 UDP를 사용하여 원격 프로세스에 데이터 요청 신호를 보내는 시간인 T_{ack}를 반드시 최적화해야만 한다. SWIM에서는 제어신호 뿐 아니라 데이터 트랜잭션도 고성능이어야 하며, 많은 경우에 있어 10μsec 정도의 성능이어야 전체 프로그램의 성능향상을 기대할 수 있다.

4. 결 론

이 논문에서 차세대 상호 연결 네트워크를 기반으로 하는 대용량의 DVSM 시스템의 성능 확장성(performance scalability)을 정량적 및 정성적으로 연구하였다. 본 연구를 위해서 FESI 프로토콜을 사용한 InfiniBand 기반의 DVSM 시스템을 구현하였으며 세 종류의 DVSM 시스템에 대하여

상세한 과부하 분석을 시행하였다. 이 연구를 통해 컨트롤 과부하가 대부분의 경우에서 성능을 결정짓는 요소로 작용하고 있음을 알 수 있었다. 또한 페이지 Prefetch 기법이 DVSM 시스템의 성능을 향상시키는 매력적인 방법 중 하나임을 확인 할 수 있었다. 마지막으로 우리의 분석적 모델을 통해 DVSM 시스템의 성능향상을 위한 컨트롤 신호의 성능 요구 조건을 계산할 수 있었다. 본 연구는 추후 성능 확장성이 뛰어난 DVSM 시스템을 개발하기 위해 유용하게 사용될 것이다.

참 고 문 헌

[1] http://www.ksc.re.kr/infor/infor_1.htm.

[2] http://www.openmp.org/

[3] A. L. Cox C. Amza, S. Dwarkadas, P. Keleher, H.Lu, R. Rajamony, and W. Zwaenepoel. "TreadMarks : Shared memory computing on networks of workstations," In IEEE Computer, number 2, pp.18-28, 1996.

[4] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michal, Cierniak, Srinivasan Parthasarthy, Wagner Meira, Sandhya Dwarkadas, and Michael Scott. "VM-based shared memory on low-latency, remote-memory access networks," ISCA-24, pp.157-169, 1997.

[5] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder P. Singh, "Home-based SVM protocols for SMP clusters: Design, simulations, implementation and performance," HPCA-4, pp.113-124, January 1998.

[6] http://www.infinibandta.org/

[7] Inho Park, Seon Wook Kim, "The distributed virtual shared-memory system based on the Infiniband architecture," Journal of Parallel and Distributed Computing, vol. 65, Issue 10, pp.1271-1280, 2005.

[8] Inho Park, Seon Wook Kim, "Study of OpenMP applications on the InfiniBand-based software distributed shared-memory system," Parallel Computing, vol. 31, Issues 10-12, pp.1099-1113, October 2005.

[9] T. Birk and L. Liss and A. Schuster, "Efficient Exploitation of Kernel Access to InfiniBand: a Software DSM Example," Hot Interconnects, pp.130-135, August 2003.

[10] 조명진, 김선욱, "InfiniBand 기반 분산 가상 공유 메모리 시스템에서의 프로토콜 과부하 분석," 한국정보과학회 HPC 연구회 추계학술발표대회, pp.19-26, 경희대학교, 2006.

[11] http://www.mellanox.com/

[12] http://www.spec.org/omp/

[13] Amdahl, G.M. "Validity of the single-processor approach to achieving large scale computing capabilities," In *AFIPS Conference Proceedings*, vol. 30, AFIPS Press, Reston, VA., pp.483 - 485, 1967.

[14] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones and B. Parady. "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance." In Workshop on OpenMP Applications and Tools, pp.1-10, July 2001.

[15] Keleher. P., Cox. A.L., Zwaenepoel. W., "Lazy Release Consistency for Software Distributed Shared Memory," ISCA, pp.13-21, 1992.

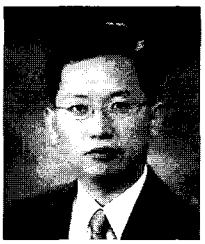
조 명 진

e-mail : linux@korea.ac.kr

2006년 고려대학교 전기전자전파공학부 졸업(학사)

2006년~현 재 고려대학교 전자전기공학과 석사과정

관심분야 : 병렬처리, 성능평가 등



김 선 욱

e-mail : seon@korea.ac.kr

1988년 고려대학교 전자전산공학과 졸업(학사)

1990년 Ohio State Univ. 전기공학과 졸업(이학석사)

2001년 미국 퍼듀대학 전기컴퓨터공학과 졸업(공학박사)

2001년~2002년 인텔 staff software Engineer

2002년~2005년 고려대학교 전기전자전파공학과 조교수

2005년~현 재 고려대학교 전기전자전파공학과 부교수

관심분야 : 컴파일러, 프로세서 및 SoC 디자인, 병렬처리, 성능평가 등