저작자표시-비영리-변경금지 2.0 대한민국

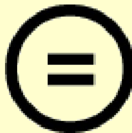
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

고성능 스토리지를 활용한 초고속 SAN 환경 구현

Towards Fast SAN Environment with High Performance Storage

2013년 2월

서울대학교 대학원

전기·컴퓨터 공학부

최 재 우

고성능 스토리지를 활용한 초고속 SAN 환경 구현

Towards Fast SAN Environment with High Performance Storage

지도교수 염 헌 영

이 논문을 공학석사학위논문으로 제출함

2013년 2월

서울대학교 대학원

전기·컴퓨터 공학부

최 재 우

최재우의 석사학위논문을 인준함

2013년 2월

위 원 장 민 상 렬 (인)

부위원장 염 헌 영 (인)

위 원 엄 현 상 (인)

초록

오늘날 서버 시스템 환경은 급속도로 증가하는 데이터를 효율적으로 처리 및 저장하기 위해 네트워크로 연결된 다수의 머신들로 clustering 이나, 분산시스템, 또는 SAN(Storage Area Network) 환경 등을 구성하여 보다 효율적인 서버 시스템을 구현하고 있다.

이러한 서버 시스템 환경에서의 performance bottleneck은 주로 스토리지 디바이스이며, 이를 극복하기 위해 고성능 스토리지 시스템에 대한 요구가 증가하고 있다. 하지만 단순히 스토리지 장비를 교체하는 것만으로는 고성능 스토리지의 뛰어난 성능을 제대로 활용할 수 없으며, 그에 맞는 최적화 작업이 필요하다.

본 연구에서는 먼저 기존의 SAN환경에 고성능 스토리지 장비를 적용해 보았고, 초고속 네트워크 장비를 사용했음에도 불구하고 small size random I/O에 대해서는 빠른 디바이스 장비의 뛰어난 성능을 제대로 활용할 수 없음을 확인하였다.

이에 우리는 고성능 스토리지 장비의 성능을 최대한 활용할 수 있는 SAN solution 개발을 목적으로 기존의 SAN I/O path에서 존재하던 software overhead를 없애고, 고성능 스토리지의 bandwidth를 최대한 활용할 수 있도록 I/O 처리과정에서의 parallelism을 높였으며, RDMA 전송방식을 사용하는 초고속 네트워크 환경에서 small size random I/O의 성능 향상을 위해 temporal merge mechanism 을 적용한 새로운 data transfer protocol 을 제안하였다. 또한 이를 Prototype 형태로 구현하여 latency와 bandwidth 측면에서 높은 성능 향상을 확인하였다.

주요어 : SAN, Stroage Area Network, Fast Storage, RDMA
Data Transfer
학 번 : 2011-20946

목차

초록 iii

목차 v

표 목차 vii

그림 목차 viii

제 1 장 서론 1

제 2 장 배경 6

2.1 High Performance storages 6

2.2 Traditional SAN I/O Path 7

2.3 Existing SAN Solution with Infiniband 8

제 3 장 Design Exploration 10

3.1 SAN I/O Path 상의 소프트웨어 오버헤드 제거 10

3.2 Increase Parallelism 12

3.3 Temporal Merge 14

제 4 장 Implementation Details 18
4.1 Block RDMA Protocol (BRP) 18
4.2 BRP Target 21

제 5 장 실험 및 검증 25
5.1 Latency & Bandwidth 26
5.2 BRP 성능 분석 30
5.3 Real Workload Evaluation 35

제 6 장 관련 연구 37

제 6 장 향후 과제 38

제 7 장 결론 39

참고 문헌 41

ABSTRACT 45

표 목차

Table 1 Infiniband RDMA send latency 17
Table 2 Latency Comparison between SRP-SRPT I/O path and BRP-BRPT I/O path 26

그림 목차

Figure 1 Traditional SAN I/O path in the Linux Subsystem 8

Figure 2 New SAN I/O path which removed the SCSI Layer 11

Figure 3 Processing multiple I/O requests in parallel with Work-Threads 14

Figure 4 Comparison between existing RDMA transfer and Temporal Merge applied RDMA transfer 17

Figure 5 Block RDMA Protocol structure 20

Figure 6 Throughput comparison between SRP-SRPT and BRP-BRPT for a variety of I/O patterns 29

Figure 7 Performance analysis with different numbers of threads for random write pattern 33

Figure 8 Performance analysis for three optimizations in BRP 33

Figure 9 Performance comparison for TPC-C benchmark on PostgreSQL 36

제 1 장 서론

오늘날 서버 시스템 환경은 급속도로 증가하는 데이터를 효율적으로 처리 및 저장하기 위해 네트워크로 연결된 다수의 머신들로 clustering 이나, 분산시스템, 또는 SAN(Storage Area Network) 환경 등을 구성하여 보다 효율적인 서버 시스템을 구현하고 있다.

CPU core의 수가 점점 늘어나면서 computing capability가 높아지고 초고속 network의 성능이 빠르게 발전하고 있는데 비해 스토리지 시스템 성능의 발전 속도는 그에 훨씬 미치지 못하고 있다. 그로 인해 전체 시스템 환경에서 스토리지 시스템은 항상 성능상의 bottleneck이 되고 있으며, 이러한 문제를 해결하기 위해 고성능 스토리지 시스템에 대한 시장의 요구가 증가하고 있는 추세이다. HDD기반의 스토리지 시스템 은 기계적 움직임으로 동작하는 하드웨어의 속성상 성능상에 한계가 존재하며, 고성능 스토리지 시스템에 대한 시장의 요구를 만족 시킬 수 없다. 그 대안으로 flash 기반의 고성능 SSD나, DRAM기반의 SSD, 혹은 memory를 직접 스토리지로 활용하는 테크닉 등이 대안으로 떠오르고 있으며, PCRAM, FeRAM, MRAM등의 차세대 스토리지 디바이스에 대한 연구도 활발히 진행되고 있다[4,9].

하지만 단순히 기존의 HDD기반의 스토리지 시스템 환경에 고성능 스토리지를 교체하는것 만으로는 고성능 스토리지 시스템의 성능을 제대로 활용할 수 없으며, 그에 따른 최적화 과정이 필요할 것이다. 이미 이러한 고성능 스토리지 시스템에 대한 연구가 다양한 system layer 및 solution 들에 대해서 진행 되고 있다[5,6,20, 25].

SAN(Storage Area Network)은 host computer와 스토리지간의 data

bus를 high-speed network로 대체한 시스템으로, 스토리지 디바이스는 network에 directly attached 되며, standard network protocol을 통해서 여러 host computer와 interact한다. 일반적으로 SAN 환경에서의 host computer를 Initiator, 스토리지 서버를 Target 이라 부른다. 이러한 SAN은 보통 enterprise 와 소규모 혹은 중규모의 비즈니스 환경에서 효율적인 서버시스템 구축을 위해 널리 사용되고 있으며, Gigabit Ethernet, Fibre Channel, Infiniband 등의 초고속 네트워크를 이용하여 구축할 수 있다[11,14,15].

본 연구에서는 먼저 SAN환경에 단순히 고성능 스토리지 장비를 적용하여 기존 SAN solution이 고성능 스토리지장비의 뛰어난 성능을 제대로 활용할 수 없다는 사실을 확인하였다. 스토리지 장비는 PCI-E 인터페이스를 이용하는 DRAM기반의 SSD를 이용하였다. 이 장비는 latency와 bandwidth 측면에서 뛰어난 성능을 자랑하며, sequential, random I/O 모두 uniform한 성능을 보여 주고 있어, 본 연구의 목적에 아주 적합한 스토리지 장비라 할 수 있다[23]. 기존의 solution으로는 SAN 환경을 구축할 수 있도록 해주는 open program인 SCST를 사용했고[22], 네트워크는 RDMA를 이용하여 데이터 전송을 수행하는 초고속 네트워크 장비인 Infiniband를 사용하였다[16]. 다양한 I/O 타입에 대해 실험한 결과 small size random I/O 경우 read와 write 모두 초고속 네트워크 장비를 사용했음에도 불구하고 고성능 스토리지 장비에 대한 성능이 현저하게 떨어지는 것을 확인할 수 있었다.

그 첫 번째 원인은 기존의 Block I/O path가 가지는 software overhead이다. 현재의 Block device에 대한 I/O path는 HDD에 대한 최적화를 수행하며 발전해 왔으며, 기본적으로 disk는 느리다는 가정하에 다양한 최적화 기술들을 적용하고 있다. 문제는 이러한 disk assumption이 고성

능 스토리지 시스템에 그대로 적용하기에는 I/O path가 너무 길고 비효율적이다. 기존의 SAN Solution은 보통 SCSI level에서 스토리지 가상화를 수행한다[1]. SCST 역시 SCSI middle layer에서 수행되는 커널 프로그램이며, Infiniband의 경우 SRP(SCSI RDMA Protocol)을 이용하여 RDMA를 통한 데이터 전송을 수행하게 된다. 실제로 low latency를 자랑하는 고성능 스토리지 장비에 대해서 SCSI layer는 I/O path를 불필요하게 길게 만들어 그 자체로써 오버헤드가 될 수 있음을 증명하는 연구들이 존재한다[5]. 또한 Block layer의 I/O scheduler의 경우도 마찬가지로 Disk를 가정한 scheduling policy를 사용하거나, merge를 위해 I/O를 바로 처리하지 않고 delay하는 등, 고성능 스토리지 시스템 하에서 오버헤드로 작용할 수 있는 불필요하고 복잡한 작업들을 수행하고 있다.

두 번째 원인은 parallelism의 부재이다. 일반적으로 고성능 스토리지의 경우 디바이스 레벨에서 I/O에 대한 parallelism을 높이고 있다. 이들은 multiple I/O를 효율적으로 처리하여 뛰어난 bandwidth 성능을 구현한다. 하지만 기존의 SAN solution들은 이러한 고성능 스토리지의 특성을 잘 이용하지 못하고 있다. RDMA를 이용한 SAN 환경에서의 I/O path는, write의 경우, 먼저 Initiator로부터 command (RDMA 수행을 위한 데이터 정보)를 받고, 이 후 Target에서 RDMA를 수행하여 initiator의 데이터를 읽어온 후, 실제 저장될 스토리지에 device I/O를 수행하게 된다. 서로 다른 I/O 요청에 대해서 위의 일련의 과정들은 서로 거의 독립적이라고 볼 수 있으며 parallel하게 처리될 수 있는 작업들이다. 하지만 기존의 SAN solution을 분석해보면 이러한 처리 과정에서 serialized execution들이 존재했으며, 특히 device I/O의 경우 serial 하게 처리되는 경우가 많았다. 이러한 처리구조는 small sized random I/O의 경우 더욱 심각한 성능저하 현상을 초래하게 되며, 고성능 스토리지를 제대로

활용하지 못하는 아주 비효율적인 구조라고 볼 수 있다.

이에 우리는 위에서 분석한 기존 SAN solution의 문제점을 바탕으로, 이를 극복할 수 있는 새로운 SAN solution 디자인을 제안한다. 본 연구에서는 먼저 SCSI layer로 인해 I/O path가 길어지는 문제를 해결하기 위해 SCSI layer를 완전히 걷어내는 작업을 수행하였다. 이를 통해 SCSI layer에서 발생하는 overhead를 제거하고 latency 측면에서의 성능 향상을 도모하였다. 그리고 기존의 I/O scheduler를 버리고 simple한 형태의 새로운 I/O scheduler를 구현하였다. 이는 기존의 I/O scheduler에서 사용했던 delay를 이용한 scheduling 기법들을 없애고, 가능한 한 빠르게 I/O request를 Target으로 전송할 수 있도록 하였다. 이를 통해 disk assumption을 제거하고 고성능 스토리지에 맞는 효율적이고 simple한 I/O path를 구현할 수 있었다. 또한 Target쪽에서 I/O 요청을 처리하는 일련의 모든 과정들을 thread pool을 이용한 multi thread 환경에서 처리되도록 하여 parallelism을 높였으며, 특히 실제 고성능 스토리지에 수행되는 device I/O가 multiple 하게 수행되어 디바이스의 bandwidth를 최대한 잘 활용할 수 있도록 구현하였다.

마지막으로 우리는 RDMA를 이용한 Initiator와 Target간의 데이터 전송 방식을 최적화한다. RDMA를 이용한 데이터 전송 protocol은 분명 뛰어난 성능을 보여준다. 하지만 small sized random I/O의 경우 기존의 SAN solution에서 사용되는 RDMA전송 방식은 분명 비효율 적인 측면이 존재하고 있었다. 상대적으로 느린 스토리지장비를 사용했을 경우 보이지 않았던 small size I/O에 대한 processing overhead 들이 고성능 스토리지 장비를 사용하면서 드러나는 것이다. 이에 우리는 small sized random I/O 의 성능을 높이기 위한 temporal merge 테크닉을 제안한다. Temporal merge는 특정 시간 동안 들어오는 I/O 요청들을 data의

공간적 연속성에 상관없이 전송하는 방법이다. 실제로 Temporal merge는 고성능 스토리지의 성능을 최대한 이용할 수 있는 기술로 소개된바 있으며, 우리는 이 기술을 두 머신 간의 data전송 protocol에 적용하였다. 실제로 Initiator와 Target간의 데이터 전송은 실제 backend 스토리지에 access하는 것이 아니므로, 공간적 연속성을 배제하더라도 개념적으로 아무런 문제가 없다. 또한 구현적인 측면에서 볼 때, RDMA에서는 scatter/gather DMA 기능을 제공하기 때문에 불연속적인 address 공간에 존재하는 서로 다른 data들을 한번의 RDMA요청으로 전송할 수 있다. 따라서 우리는 다수의 small sized random I/O에 대해 temporal merge를 적용하여 실제로 수행되는 데이터 전송의 횟수를 줄이고 그에 따른 processing 과정을 최적화하여 processing overhead를 최소화 하였다.

위와 같이 우리는 SAN 환경에서 고성능 스토리지 시스템을 위해 최적화된 새로운 solution을 제안 하였다. 또한 이를 prototype형태로 구현하였으며, 실험을 통해 높은 성능 향상을 확인할 수 있었다.

이 후 section2에서는 오늘날 사용되고 있는 고성능 스토리지와 차세대 스토리지에 대해서 살펴보고, 기존의 Block I/O path 와 Infiniband를 활용한 기존의 SAN solution에 대해서 간단히 설명할 것이다. 다음 section3에서는 새로운 SAN solution에서 적용된 기술들에 대해서 자세히 살펴볼 것이며, section 4에서는 구현의 관점에서 Initiator와 Target을 나누어 자세히 설명할 것이다. Section 5에서는 새로운 SAN solution상에서 I/O benchmark를 사용한 실험을 통해 성능이 향상되었음을 보일 것이다.

제 2 장 배경

2.1 High Performance Storages

오늘날 고성능 스토리지 시스템에 대한 요구가 증가하고 있지만, 기존의 스토리지 시스템 환경의 주류를 이루는 HDD 기반의 장비들은 이러한 시장의 요구를 만족시키지 못한다. 그 대안으로 flash 기반의 SSD나 DRAM 기반의 솔루션들이 고려되고 있다.

Flash 기반의 SSD 중 Fusion-io 사의 ioDrive 시리즈는 대표적인 고성능 스토리지 이다[10]. 이는 flash의 non-volatile 속성을 가지면서, 뛰어난 latency 및 bandwidth 성능을 자랑한다. 하지만 flash 자체가 가지는 문제점인 life time 이슈라든지 write성능 저하 현상 등은 여전히 존재하고 있다.

DRAM 기반의 Solution 중에는 먼저 main memory 자체를 스토리지로 사용하는 방식이 존재한다. 이 방식은 수천개의 일반 서버 머신들을 초고속 네트워크로 연결하여 각 머신들의 main memory에 모든 데이터 정보를 저장하는 방법과[13,19]한 머신에 대량의 DRAM을 꽂아 사용하는 방법으로[21] 다시 나뉘게 된다. 전자의 경우 보통의 머신들만으로 구현할 수 있다는 장점이 있지만 다수의 머신들로 인한 공간적 낭비와 network overhead에 대한 문제가 존재할 수 있다. 후자의 방식은 대량의 DRAM을 main memory로 사용하기 위한 하드웨어적인 기술이 필요하다. 두 방식 모두 DRAM의 물리적인 특성상 power failure가 발생했을 시 data loss 가 발생할 수 있다는 치명적인 약점을 가지고 있다. 따라서 이러한 문제를 해결하기 위해 머신들 간의 replication 이나 data backup 기술 등이 이용되고 있다.

DRAM을 이용한 또 다른 Solution 중 하나는 본 연구에서 사용하는 방식으로 DRAM 기반의 SSD이다.(JSM) 이는 PCI-E Channel을 통해 host computer에 연결되고, battery-backed 시스템으로 power failure 발생시 추가적인 battery의 전력을 이용해 데이터를 non-volatile 스토리지에 저장할 수 있도록 하여, DRAM의 구조적 문제를 해결하고 있다. 또한 뛰어난 bandwidth와 latency를 보여 주고 있으며, sequential / random pattern에 상관없이 uniform한 I/O 성능을 유지하기 때문에, 고성능 스토리지 환경에 대한 최적화를 수행하는 본 연구에 있어서 아주 적합한 스토리지 장비이다.

2.2 Traditional SAN I/O Path

모든 I/O 요청들은 스토리지 device에 I/O를 수행하기 위해 필요한 필수적인 정보들인 I/O type, sector address, transfer size들을 가지고 있다. Linux I/O subsystem에서 각 path를 지날 때마다 I/O를 수행하는 단위 structure는 바뀔 수도 있지만, 이러한 필수적인 정보들을 계속 유지하고 있다는 점은 변함이 없다. Figure 1은 일반적인 SAN환경에서의 I/O path를 추상적으로 나타낸 그림이다. 이 그림에서 알 수 있듯이 SAN 환경의 Initiator에서 수행되는 I/O 는 일반적인 Local SCSI 스토리지에 대한 Linux I/O subsystem path와 상당히 유사하다. Application 단에서 시작된 I/O 요청은 File System을 거쳐 Block Layer로 전송된다. File System 에서 Block Layer로의 I/O 수행 요청은 bio structure 단위로 전송된다. 이 bio 는 elevator algorithm 등을 수행하여 이전에 이미 들어와 대기하고 있는 다른 I/O 요청과 merge 될 수 있다. Merge가 되든 안 되든 bio는 request structure를 구성하게 되고 request queue에 insert되어 I/O scheduler에 의해 처리되기를 기다리게 된다. 이 후 I/O

scheduler에 의해 fetch된 request는 SCSI layer로 전송되고, SCSI Mid-level에서는 SCSI 디바이스에 대한 I/O 수행을 위해 request structure를 SCSI command로 변형 시킨다. 이 후 일반적인 local SCSI 디바이스에 대한 I/O path라면 구성된 SCSI command를 해당 SCSI 디바이스의 Host Bus Adapter(HBA) driver에 전송할 것이다. SAN I/O path 의 경우, SCSI command는 network를 통해 Target으로의 데이터 전송을 수행할 Front-End Initiator driver로 전달된다. 이 후 SCSI command를 기반으로 하는 데이터 전송 protocol을 따라 command, response 또는 data 전송 등이 수행 된다.

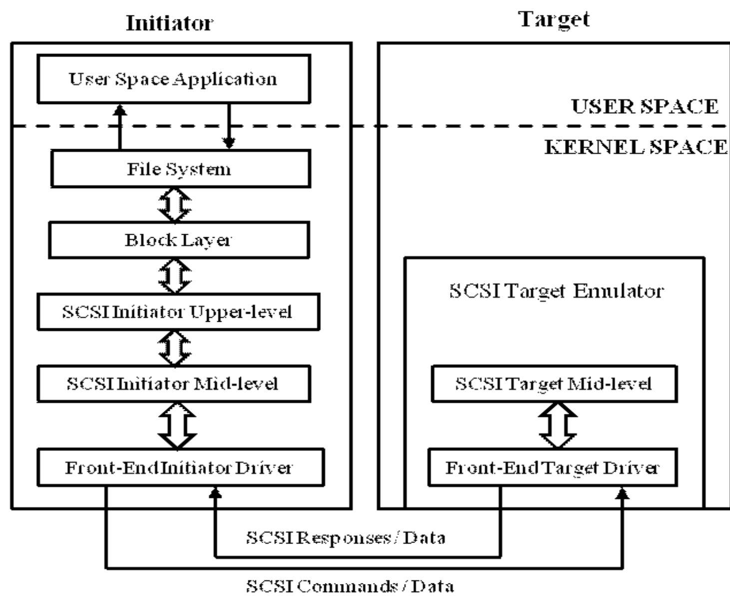


Figure 1: Traditional SAN I/O path in the Linux Subsystem[1]

2.3 Existing SAN Solution with Infiniband

현재 Linux 환경에서 Infiniband의 RDMA전송 기능을 활용해 간단히 SAN을 구현할 수 있는 방법은 두 가지 이다. 첫 번째는 SCSI RDMA

Protocol (SRP)를 이용하는 것이고[22], 다른 하나는 iSCSI extensions for RDMA (iSER) protocol을 이용하는 것이다[17]. 이 둘은 모두 Infiniband를 지원하고 SCSI layer에서 RDMA 데이터 전송 방식을 이용하여 SAN 환경을 구축할 수 있다는 점에서는 동일하지만, 실제로 어떤 solution을 사용할 지 결정하기에 앞서 이 둘 간의 tradeoff를 고려해야 할 것이다.

우선 SRP를 통한 SAN 구현은 SCST를 통해서 가능하며, iSER의 경우 tgt projet에 의해서 구현 가능하다. SRP는 iSER에 비해 SAN 환경을 구축하기 수월한 편이다. 또한 SCST의 SRP Target의 경우 kernel 영역에 구현되어 있어 user 영역에 구현되어 있는 iSER target 보다 높은 bandwidth와 짧은 latency를 보여준다. 하지만 iSER은 iSCSI 인터페이스를 사용하기 때문에 target-discovery를 수행하여 특정 Target에 log in한다거나 password 기반의 사용권한을 준다거나 하는 다양한 management 기능들을 이용할 수 있다[17,22].

본 연구에서는 일단 Performance에 초점을 맞추고 있기 때문에 SRP를 선택하였으며, 새로운 SAN solution을 구현하는데 있어서도 iSCSI 인터페이스에 의존하는 iSER 을 참조하는 것은 적합하지 않다고 판단되어 SRP를 기반으로 연구를 진행하였다.

제 3 장 Design Exploration

3.1 SAN I/O Path 상의 소프트웨어 오버헤드 제거

앞에서 언급한 바와 같이 SCSI layer는 고성능 스토리지 시스템에는 overhead로 작용한다. 하지만 기존의 SAN solution은 보통 SCSI layer에서 스토리지 가상화를 수행하고 있다. 이는 Infiniband를 사용하는 SAN 역시 마찬가지이며, file system으로부터 전달된 I/O 요청은 block layer에서의 bio structure 부터 시작해서 merge 등의 과정을 거쳐 request structure로 구현 되어 request queue에 저장되며, 이후 스케쥴링 과정에서 fetch된 request는 다시 SCSI command로 변형되어 SRP driver에 전달되고 network 전송 protocol을 통해 Target으로의 command 전송이 수행된다. Target쪽에서는 Initiator로부터 전달받은 Command를 분석하여 RDMA를 통한 데이터 전송과 실제 스토리지 에 대한 device I/O 등을 수행하게 될 것이다. Initiator로부터 전달받은 SCSI 기반의 command와 데이터를 바탕으로 새로운 bio structure를 구성하여 block I/O를 수행하는 것이다. 이러한 일련의 I/O과정을 살펴볼 때, 고성능 스토리지를 사용하는 SAN환경에서 SCSI command 기반의 network 전송 protocol로 얻을 수 있는 이득이 전혀 없으며, 오히려 Initiator 단의 I/O path를 불필요하게 길어지게 만들고 Target 쪽에서의 processing overhead만 더 커지게 되어, 결국 I/O 수행 latency가 길어지게 된다.

따라서 우리는 이러한 disk의 잔재인 SCSI layer를 걷어내고 Figure 2와 같이 block layer 최상단에서 스토리지 가상화를 구현하였다. I/O 요청에 대한 command 구현은 block layer의 bio structure를 기반으로 하며,

데이터 전송은 기존의 SRP와 같이 scatter / gather RDMA 방식을 사용한다. Target쪽에서는 기존보다 적은 processing overhead로 bio structure를 구성하여 block I/O를 수행할 수 있게 된다. 이와 같이 SCSI layer를 제거함으로써 SAN I/O path를 간소화 하여 I/O 수행 latency를 줄일 수 있었다.

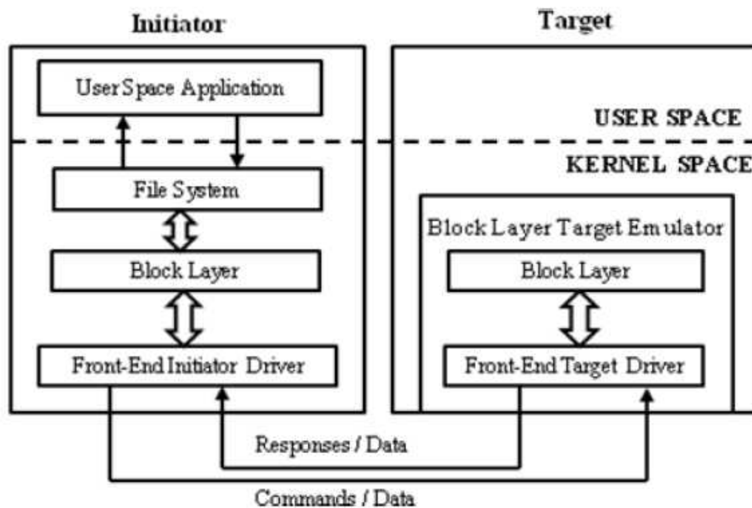


Figure 2: New SAN I/O path which removed the SCSI Layer

Linux I/O subsystem의 I/O scheduler는 기본적으로 Disk 를 가정한 scheduling policy를 사용하거나, merge를 위해 I/O를 바로 처리하지 않고 delay하는 등, 고성능 스토리지 시스템 하에서 오버헤드로 작용할 수 있는 불필요하고 복잡한 작업들을 수행하고 있다. 예를 들어 linux 의 기본 scheduler policy인 CFQ는 가장 대표적인 Disk assumption 중 하나이다. 각 프로세스마다 작업큐를 가지고, time slice를 할당 받으며, time slice 안에 작업을 모두 끝내더라도 바로 끝내지 않고 특정 시간동안 혹시나 있을 I/O요청을 기다린 후 결국 없으면 다음 프로세스 큐로 이동하는 방식이다. 이는 고성능 스토리지 에서는 불필요한 delay를 발

생시켜 치명적인 overhead로 작용한다. scheduler의 정책을 NOOP으로 변경함으로써 overhead를 어느 정도 감소시킬 수 있다. 하지만 여전히 merge를 위한 불필요한 대기를 수행하거나 복잡한 스케쥴링 과정을 거치는 등, 고성능 스토리지에 적합하지 않은 작업들을 수행하고 있다[8].

이에 본 연구에서는 기존의 I/O scheduler를 사용하지 않는다. 이는 기존 I/O scheduler에서 발생하는 overhead를 없애기 위해서 뿐만 아니라, 위에서 언급했듯이 bio structure를 기반으로 하는 전송 protocol을 사용하므로, request structure 기반에서 동작하는 I/O scheduler를 사용하는 것은 그 자체로 overhead가 되기 때문이다. 우리는 아주 simple한 형태의 요청 큐를 구현하고 이 후에 설명할 temporal merge 테크닉을 이용한 fetch과정을 수행함으로써 최대한 빨리 I/O를 처리할 수 있도록 구현하였다.

3.2 Increase Parallelism

고성능 스토리지 시스템을 위한 SAN solution에 적용 될 두 번째 최적화 기술은 Target에서 I/O 요청을 처리하는 과정에 대해서 parallelism을 증가 시키는 것이다. 일반적으로 고성능 스토리지의 경우 디바이스 레벨에서 I/O에 대한 parallelism을 높이고 있다. 이는 multiple I/O를 효율적으로 처리하여 뛰어난 bandwidth 성능을 구현할 수 있도록 한다. 하지만 기존의 SAN solution들은 이러한 고성능 스토리지의 특성을 잘 이용하지 못하고 있다. RDMA를 전송방식을 이용하는 SAN 환경에서 Target이 수행하는 I/O 요청에 대한 처리 과정은 다음과 같이 크게 4가지로 분류할 수 있다. 1) RDMA 수행, 2) Device I/O 수행, 3) Initiator에게 response 전송, 4) I/O요청 완료작업 수행. 그리고 서로 다른 I/O 요청에 대해서 수행하는 이러한 일련의 처리 과정들은 서로 거의 독립적

이라고 볼 수 있으며 parallel하게 처리될 수 있는 작업들이다. 특히 디바이스 I/O가 parallel 하게 처리될 경우 고성능 스토리지의 뛰어난 multiple I/O 처리 능력을 활용할 수 있어 성능상에 큰 이점을 얻을 수 있다. 하지만 기존의 SAN solution을 분석해보면 device I/O의 경우 serial 하게 처리되는 경우가 많았으며, 이와 같이 스토리지의 bandwidth를 제대로 활용하지 못하는 처리구조는 small sized random I/O의 경우 더욱 심각한 성능저하 현상을 초래하게 된다. 느린 HDD를 사용하는 기존의 SAN 환경에서는 page cache를 사용하는 File-I/O 방식을 통해서 asynchronous한 I/O를 수행하기 때문에, memory 와 스토리지간의 엄청난 성능 차이로 인해 굳이 device I/O 과정을 parallel하게 수행할 필요가 없었다. 하지만 low latency 스토리지 장비의 경우 page cache를 활용하는 것이 오히려 추가적인 copy overhead를 유발시킬 수 있다. 또한 memory size에 따라 성능차이가 많이 발생하게 되고, 다수의 Initiator들에 의해 intensive한 I/O 요청이 들어올 경우 memory contention이 발생하여 page cache를 효율적으로 활용하지 못해 전체 시스템 성능이 저하되는 경우가 발생할 수 있다.

따라서 본 논문에서 제안하는 고성능 스토리지 시스템을 위한 SAN solution에서는 Figure 3과 같이 Target에서 수행되는 일련의 모든 I/O 처리 과정들을 thread-pool내의 work-thread들에 의해 처리될 수 있도록 구현하였고, 이를 통해 multiple device I/O 가 parallel하게 수행될 수 있는 처리환경이 구축되어 고성능 스토리지 장비의 뛰어난 bandwidth를 잘 활용할 수 있게 되었다.

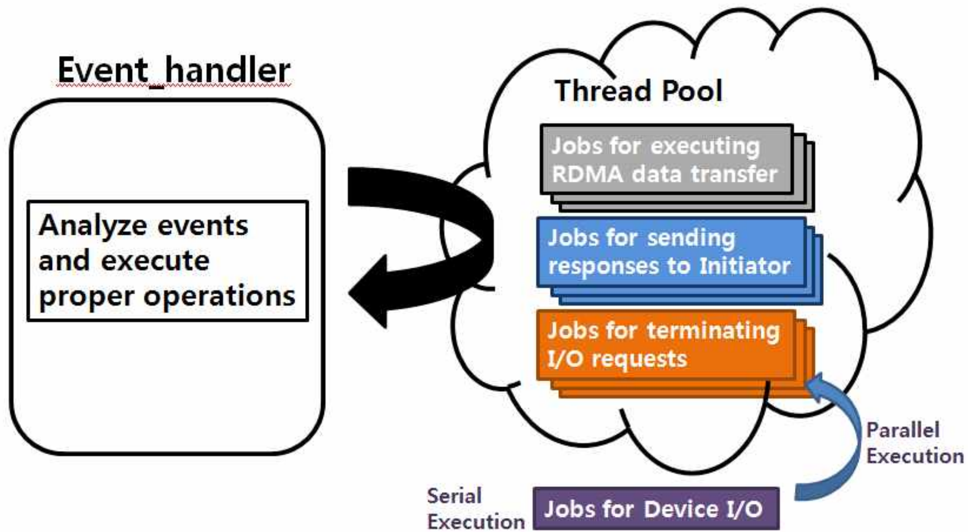


Figure 3: Processing multiple I/O requests in parallel with Work-Threads

3.3 Temporal Merge

마지막으로 우리는 RDMA를 이용한 Initiator와 Target간의 데이터 전송 방식을 최적화한다. RDMA를 이용한 데이터 전송 protocol은 분명 뛰어난 성능을 보여준다. 하지만 small sized random I/O의 경우 기존의 SAN solution에서 사용되는 RDMA전송 방식은 분명 비효율 적인 측면이 존재하고 있다.

RDMA 전송 프로토콜을 포함한 모든 네트워크 프로토콜에는 데이터 전송 이전의 pre-processing 과정과 데이터 전송 이후의 post-processing 과정을 거친다. 우리는 RDMA 전송 프로토콜을 기반으로 하는 SAN환경에서 RDMA 데이터 전송을 수행하기 위해 준비하는 과정을 pre-processing 과정이라 하고, RDMA 데이터 전송완료에 대한 처리과정을 post-processing이라 하겠다. I/O 요청의 타입(Read/Write) 따라 처리

하는 processing 루틴이 조금 다르긴 하지만, 공통적으로 RDMA 데이터 전송을 수행하는 주체는 Target이며, 일반적으로 pre-processing 과정에서는 RDMA전송을 위한 메모리영역을 확보하고 DMA mapping을 수행한다. Post-processing의 경우 RDMA completion event를 처리한다거나, DMA mapping 해제, 또는 RDMA를 수행하기 위해 할당했던 자원들을 해제하는 작업등을 수행하게 된다.

실제로 Infiniband의 RDMA 전송을 이용한 SAN 환경은 latency와 bandwidth 측면에서 뛰어난 성능을 가진다. 하지만 small size random I/O pattern의 경우, 기존의 환경에서 발견할 수 없었던 숨은 overhead가 고성능 스토리지 시스템을 만나면서 드러나게 된다. Figure 4의 첫 번째 그림은 Wrirte 요청에 대한 기존의 RDMA 전송 방식을 도식화한 것이다. Target에서는 Initiator로부터 전달받은 command 정보를 바탕으로 pre-processing과정을 거쳐 RDMA 데이터 전송을 수행하고, 이후 RDMA 전송이 완료 되면 post-processing 과정을 거친다. Large size I/O의 경우 Network transfer time이 I/O latency를 dominate하기 때문에 이러한 processing overhead가 충분히 가려지게 되지만, small size I/O의 경우 request 당 걸리는 network transfer time이 상대적으로 작기 때문에 processing overhead가 더욱 두드러지게 된다. 이러한 processing overhead가 기존의 느린 HDD의 경우 초고속 네트워크 장비의 latency나 bandwidth 보다 한참 떨어지기 때문에 이로 인한 성능저하가 발생하지 않았다. 하지만 low latency 스토리지의 경우 이러한 overhead가 전체 성능에 치명적으로 작용한다.

이에 우리는 고성능 스토리지를 위한 SAN solution에 적용될 마지막 optimization 테크닉으로 temporal merge를 제안한다. temporal merge는 특정 시간 동안 들어오는 I/O 요청들을 공간적 연속성에 상관없이

merge하여 전송하는 방법으로, small I/O pattern에 대해서 low latency 스토리지 장비의 성능을 최대한 활용할 수 있는 기법으로 소개된 바 있다[25]. 우리는 이 기술을 Initiator와 Target간의 RDMA 전송 protocol에 적용하고자 한다. 실제로 Initiator와 Target간의 데이터 전송은 실제 스토리지에 access하는 것이 아니기 때문에 이러한 공간적 연속성을 배제한 merge 테크닉이 개념적으로나 구현적으로 아무런 문제가 되지 않는다. Figure 4의 두 번째 그림은 temporal merge 기법을 적용한 RDMA 전송 방법을 도식화한 것이다. 그림에서처럼 Initiator 쪽에서는 Target으로 전송하기 위한 command 구성할 시, 서로 다른 I/O 요청들을 공간적 연속성에 상관없이 하나로 merge하여 jumbo command를 생성해 전송한다. 이 후 jumbo command에 통합되어 저장된 다수의 I/O 데이터 정보를 바탕으로 pre-processing을 수행한다. 또한 RDMA에서는 scatter / gather DMA를 지원하고 있으며, 이 방식을 통해 서로 연속적이지 않은 메모리 영역에 존재하는 데이터들을 한번의 RDMA 수행으로 전송할 수 있다. RDMA 전송이 완료되면 그에 대한 post processing 과정을 수행하게 된다. 이 후 디바이스 I/O를 수행하게 되고, 이에 대한 completion response는 완료되는 순서대로 먼저 Initiator에 전송하여 I/O 요청에 대한 응답성을 높인다.

Table 1은 데이터 크기의 변화에 따른 RDMA 전송 시간을 측정한 것이다. Table 1에서 보는 바와 같이 동일한 크기의 데이터를 각각 여러 번 나누어 보내는 것 보다, 한번에 merge 해서 전송하는 것이 latency 측면에서 훨씬 효율적인 것을 확인할 수 있다.

이와 같이 우리는 RDMA 전송 protocol에 temporal merge 테크닉을 적용함으로써 small size random I/O workload 수행 시, command 및 RDMA 데이터 전송 횟수를 줄이고 그에 따른 processing 과정을 효율적

으로 처리하도록 하여 전체 Performance를 향상시킬 수 있었다.

Bytes	min[usec]	max[usec]	typical[usec]
64	4.69	12.64	4.73
128	4.87	12.72	4.91
256	5.24	11.03	5.28
512	6.51	12.42	6.56
1024	8.27	15.84	8.32

Table 1: Infiniband RDMA send latency

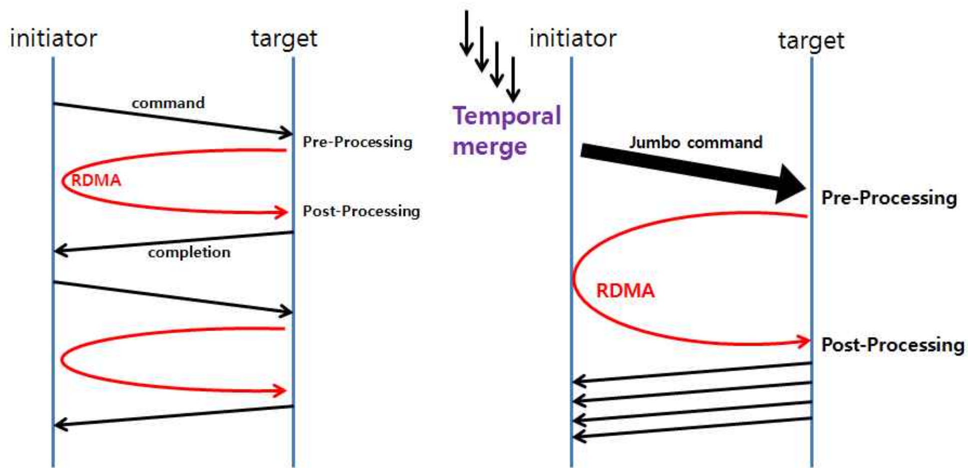


Figure 4: Comparison between existing RDMA transfer and Temporal Merge applied RDMA transfer

제 4 장 Implementation Details

4.1 Block RDMA Protocol

앞에서 설명했던 바와 같이 우리는 기존 SAN solution에서 사용하던 SCSI layer와 SCSI command 기반의 전송 protocol을 버리고, 보다 빠른 I/O path를 구현하기 위해 block layer의 bio structure 기반의 전송 protocol을 구현 하였다. 이름은 기존의 SCSI RDMA Protocol (SRP) 을 참조하여 구현하였기 때문에 Block RDMA Protocol (BRP)라 정하였다.

Figure 5에서는 BRP structure를 보여주고 있다. bio strucure의 정보들을 바탕으로 brp_cmd_jumbo structure 를 구성하여 Target에 전송한다. brp_cmd_jumbo의 구성을 살펴보면, 먼저 Target 쪽에서 새로운 bio structure를 생성하는데 필요한 필수적인 rw(read/write), size, page_count 등의 정보들을 가지고 있다. 또한 brp_cmd_jumbo에서는 Target 쪽에서 RDMA 데이터 전송을 수행하기 위해 필요한 정보들을 가져야 한다. 이는 add_data멤버에 저장되며 RDMA 수행을 위한 dma address와 remote key 등을 담고 있는 brp_info structure를 add_data에 flexible array member 방식으로 유지한다. temporal merge를 통해 구현된 jumbo command 는 서로 다른 I/O요청에 대한 정보들을 하나의 brp_cmd_jumbo에 저장해야 한다. jumbo_count 멤버 변수는 brp_cmd_jumbo에 몇 개의 서로 다른 I/O 요청정보가 merge되었는지를 나타내며, merge된 bio의 수라고 볼 수 있다. 이 경우 그 개수만큼 brp_info structure가 add_data 멤버에 저장되고 각각은 brp_info strucure의 tag변수에 의해 구분되어진다. 이 후 Target 쪽에서의 I/O 수

행이 완료되고, I/O 요청에 대한 response는 각각 따로 받게 된다. 이때 Target 으로부터 전달받은 response에 저장된 tag정보를 확인하여 그에 matching 되는 bio를 completion 시킬 수 있다.

temporal merge는 기본적으로 small size random I/O에 대한 성능향상을 위해 적용된 최적화 기술이다. large size data의 경우 merge를 수행하지 않아도 network bandwidth를 충분히 활용할 수 있다. 따라서 우리는 small size I/O에 대해서만 temporal merge를 수행하고 있으며, 여기서 말하는 small size I/O는 하나의 page size 크기를 가지는 데이터에 한정한다.

본 연구에서 구현한 temporal merge 처리 과정은 read 와 write사이에 약간의 차이가 존재한다. File System으로부터 내려온 bio structure들은 먼저 wait-queue에 저장된다. wait-queue는 read와 write에 각각 존재하며, write의 경우 worker thread를 깨워 wait-queue에 있는 다수의 write 요청들을 merge하여 하나의 brp_cmd_jumbo를 구성하여 전송한다. read의 경우 동시에 수행되는 다수의 thread 중 하나의 winner thread 만이 brp_cmd_jumbo 를 구성하여 전송한다.

temporal merge 구현 시 고려해야 할 중요한 요소 중 하나는 temporal merge를 수행하는 시점이다. temporal merge의 경우 intensive 하게 내려오는 small I/O들을 처리하는데 효과적이다. 하지만 sparse 하게 내려오는 small I/O 들에 대해서 temporal merge를 수행할 경우, Target 시스템 내에서의 I/O 요청에 대한 처리과정 중 parallelism을 크게 떨어트리게 되고, 이는 오히려 전체 I/O 성능이 떨어지게 만드는 요인이 된다. 따라서 우리는 현재 I/O 처리 상황이 busy한지를 판단할 수 있는 I/O counter 를 두고, Target에게 I/O 요청을 수행할 때마다 counter를 증가

시키고, Target으로부터 I/O 요청이 완료되었다는 response를 받을 때마다 counter를 감소 시킨다. I/O counter가 특정 threshold 값 이상 일 경우 intensive한 I/O가 수행되고 있는 상황이라 여기고 temporal merge를 수행하고, threshold 값 아래일 경우는 merge없이 바로 I/O요청을 Target에게 전송한다.

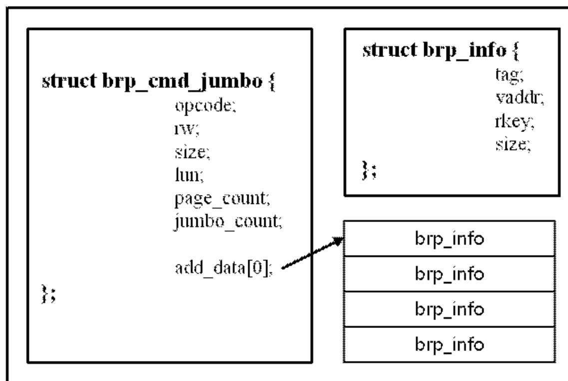


Figure 5: Block RDMA Protocol structure

temporal merge 구현 시 고려해야 할 한 가지 추가적 요소는 merge 될 수 있는 개수의 최대값 이다. intensive하게 수행되는 I/O의 경우 wait-queue에 많은 I/O 요청들이 대기하는 상황이 발생하게 될 것이다. 하지만 무작정 많은 수의 요청들을 merge하여 수행하는 것은 오히려 Target쪽에서의 parallelism을 떨어트려 성능 저하를 유발할 수 있다. 따라서 적정한 크기의 최대 merge size가 필요한데 이는 일종을 heuristic 이며, 우리는 실험을 통해 최대 merge size를 8로 정하였다. 8 이상 증가 시켰을 경우 점차적으로 성능 저하가 발생했으며, 8이하의 경우 temporal merge로 부터 얻는 성능상의 이득이 크지 않았다.

이 외에 infiniband와 관련하여 RDMA를 수행하기 위해 필요한 자원 할당 및 데이터 전송 수행 등은 관련 API들을 활용하였고[3], 기존

Solution 인 SRP의 source 코드를 분석 및 참조하여 구현하였다.

4.2 BRP Target

BRPT Target은 본 연구에서 제안하는 새로운 SAN 환경의 Target 시스템에서 수행되는 kernel module이다. BRP Target에서는 Initiator로부터 전달받은 command를 분석하여 RDMA 데이터 전송 및 디바이스 I/O를 수행하고 모든 I/O요청 처리가 완료되면 그에 대한 response를 Initiator에게 전달한다.

위에서 언급한 일련의 과정들은 section 3에서 보였던 바와 같이 brp_work라는 구조체 형태로 저장되며, 이러한 brp_work들은 wait-list에 insert되어 이 후 thread-pool형태로 관리되는 work-thread에 의해 fetch되어 처리된다. brp_work는 총 3가지 WORK_TYPE을 가지며, 각각의 type에서 수행되는 operation은 read/write type에 따라 조금씩 차이가 있다.

첫 번째 type은 HANDLE_RECV 이며, 이는 일련의 I/O 처리 과정 중 가장 먼저 수행되는 work이며, Initiator로부터 전달받은command를 분석하여 그에 맞는 처리 루틴을 수행하는 것이다. Read의 경우 디바이스 I/O를 먼저 수행하게 될것이고 Write의 경우 RDMA 데이터 전송을 먼저 수행하게 된다. 두 번째 type은 HANDLE_SEND 이다. 이는 두 가지 루틴으로 나뉘게 되는데 RDMA 전송이 완료되었음을 나타내는 completion event와 Initiator로의 response 전송이 완료되었음을 알리는 event이다. 전자는 오직 Write요청과 관련된 루틴으로 Initiator로부터 RDMA 전송을 통해 데이터를 읽어왔음을 알리는 것이며, 이 후 디바이스 I/O를 수행하게 된다. 후자의 경우 Read/Write 모두에 해당되는 것으로 I/O요청 완료에 대한 response를 전송이 제대로 수행되었음을 알

리는 것으로, 해당 I/O요청을 수행하기 위해 할당했던 자원들을 해제하는 루틴을 수행한다.

마지막 세 번째 type은 HANDLE_IO 이다. 이는 실제 backend 스토리지에대한 device I/O 수행이 완료되었을 때 수행 되는 work이다. 이 역시 read냐 write냐에 따라 수행하는 operation이 조금 다르다. Write의 경우 디바이스 I/O가 전체 I/O 처리과정에서 가장 마지막이므로 곧 바로 I/O 요청 완료에 대한 response를 Initiator에게 전송한다. 하지만 Read의 경우 Device I/O 수행 후 Initiator에게 RDMA를 통한 데이터 전송을 수행해야하며, 이후 I/O 요청 완료에 대한 response를 전송한다.

위에서 소개된 3가지 type의 work들은 thread-pool에서 관리되는 다수의 work-thread들에 의해 parallel하게 수행되며, 해당 work-thread가 wake-up되어 수행되는 과정은 leader-follower model 방식을 따라 수행한다[7]. 이러한 thread-pool 환경에서 sparse 한 I/O 상황일 경우 적은 수의 work-thread가 active하게 수행 되고, intensive한 I/O 환경에서는 많은 수의 work-thread들이 수행될 것이다. 또한 thread-pool에서 관리되는 최대 work-thread 수는 intensive한 I/O 상황을 충분히 처리 할 수 있을 만큼의 수가 되어야 한다. 이 값은 일종의 heuristic으로 볼 수 있으며, 수행되는 시스템 환경의 core 수에 영향을 많이 받을 것이다. 본 연구에서 사용한 머신에서는 실험을 통해 확인한 결과 16개의 thread면 intensive한 환경에서도 충분히 효율적으로 처리가 가능했다.

기본적으로 BRP Target에서 RDMA 데이터 전송을 수행하기 위한 준비과정은 기존 Solution인 SRP Target 모듈과 크게 다르지 않다. Initiator로부터 받은 command를 바탕으로 RDMA 데이터 전송을 위한 버퍼공간을 확보하고 scatter / gather DMA를 위한 scatterlist를 할당한 후

DMA 맵핑을 수행한다. 이후 RDMA 데이터 전송을 수행하는 Infiniband 관련 API를 활용하는 것이다. 하지만 기존의 SAN Solution과 가장 큰 차이점은 바로 temporal merge에 대한 처리이다. 하나의 jumbo command에 다수의 I/O 요청들에 대한 정보가 merge되어 저장되어있다. 이를 기존의 처리 방식을 그대로 적용한다면 각각에 대한 I/O 요청이 단순히 serial 하게 처리되는 것으로 프로세싱 과정에서 별다른 이득이 없다. 오히려 parallelism을 떨어트려 전체 성능을 떨어트리는 요인으로 작용할 수도 있다. 이에 우리는 이러한 jumbo command에 대한 처리 과정을 효율적으로 수행하기 위해 pre-allocation 을 최대한 활용하였다. 앞에서 설명했던 바와 같이 temporal merge는 small size I/O요청에 대해서만 수행되며, 이들은 모두 한 개의 page size 데이터 크기를 가지는 I/O 요청들이며, 이들에 대한 자원 할당과정은 완전히 예측 가능하다. 따라서 page size크기를 가지는 bio 및 scatterlist 를 포함하여 RDMA 및 디바이스 I/O를 수행하는데 필요한 자원들 중 가능한 모든 자원들을 pre-allocation하여 관리하고 있다가, temporal merge된 jumbo command에 저장된 다수의 I/O요청을 처리할 때 할당하는 식으로 구현하여, processing과정에서 드는 비용을 최소화하였다.

또한 jumbo command를 처리하는데 있어서 read / write요청에 대해 대처하는 방식에 차이가 있다. 두 경우 모두 하나의 jumbo command에 다수의 I/O 요청에 대한정보가 저장되어 전송된 것은 동일하다. 하지만 read의 경우 device I/O를 먼저 수행하게 되고, write의 경우 RDMA 데이터 전송을 먼저 수행한다. 따라서 Read 의 경우 고성능 디바이스의 bandwidth를 최대한 활용하기 위해 곧바로 처리과정을 parallel화 시킨다. 다시 말하면, merge된 I/O의 수만큼 brp_work structure를 구성하고, 그 만큼의 work thread 들을 wake up 시켜 multiple한 I/O를 수행

할 수 있도록 한다. Write의 경우 RDMA 데이터 전송을 먼저 수행해야 하며, jumbo command에 저장된 다수의 I/O요청에 대해 한 번의 RDMA 수행으로 처리 되기를 원한다. 따라서 RDMA 수행하기 전까지의 준비과정이 merge된 I/O요청의 수만큼 serial하게 처리될 수밖에 없다. 이를 해결하기 위해 앞에서 언급했던 pre-allocation을 최대한 활용하여 overhead를 최소화 하겠다는 것이다. 이 후 RDMA 요청이 끝나고 디바이스 I/O를 수행하게 될 때는 Read와 같이 multiple I/O를 수행하도록 하여 고성능 스토리지의 성능을 최대한 활용하도록 한다.

제 5 장 실험 및 검증

우리는 총 두 개의 머신을 사용하여 Initiator와 Target을 구성해 실험을 진행하였다. 두 머신 모두 두 개의 Intel Xeon E5630 2.53GHz quad core CPUs (총 8 core)를 가지고 각각 8GB, 16GB size의 main memory를 장착하고 있으며, 모든 실험은 Linux 2.6.32 kernel에서 수행하였다. Target에는 64GB(= 8GB of DDR2*8) capacity를 가지는 DRAM-Based SSD가 존재하며, 이 장비를 본 연구의 고성능 스토리지 시스템으로 활용한다. SAN환경 구축을 위한 초고속 네트워크 장비로는 Mellanox사의 ConnectX-2 Infiniband 장비인 MHQH18B-XTR을 사용했으며, 최대 40Gb/s 의 bandwidth의 성능을 가진다.

우리는 먼저 기존 Solution과의 성능을 비교하고, 앞에서 제안했던 3가지 optimization에 대한 성능을 분석하는 실험들을 진행하였다. 각각의 optimization들을 적용한 정도에 따라 BRP-1, BRP-2, BRP-3로 구분 짓는다. BRP-1은 SCSI layer를 걷어내고 I/O scheduler를 수정한 최적화만을 적용 한 것이고, BRP-2는 BRP-1에 추가적으로 Target에서 수행되는 I/O 처리과정에서의 parallelism을 높인 경우 이며, 마지막 BRP-3는 이에 더해서 temporal merge를 적용한 것으로 intensive한 I/O를 판단하는 최적의 threashold값이 반영된 최종 버전이 되겠다. numbering 없이 BRP라고 표현한 것은 모든 optimization이 적용된 BRP-3이다.

우리는 FIO와 같은 benchmark tool을 이용하여[12], 다양한 I/O pattern에 대한 실험을 하였다. 먼저 기존의 SAN solution인 SRP-SRPT 드라이버에서 수행한 실험결과와 본 연구에서 제안한 BRP-BRPT 드라이버에서 수행한 결과를 latency와 bandwidth 측면에서 비교 분석 할 것이다. 또한 고성능 스토리지 에 대한 local I/O 수행 결과를 바탕으로,

local I/O 대비 얼마만큼의 성능이 나오는지를 살펴보겠다. 이 후 본 연구에서 제안하는 3가지 optimization에 대한 성능을 분석하고, 마지막으로 real-workload에 대한 실험으로 Database환경을 구축하여 TPC-C를 수행하고 두 SAN solution간의 성능을 비교 분석 할 것이다.

5.1 Latency & Bandwidth

우리는 앞에서 설명했던 바와 같이 SAN I/O path의 길이를 줄여 I/O 수행 latency를 낮추기 위해 SCSI layer를 제거하고 scheduler를 최적화하여, BRP-BRPT 환경을 구현하였다. Table 2는 기존의 SAN solution인 SRP-SRPT와 본 연구에서 구현한 SAN solution 각각에 대해서 read/write I/O 수행하고, 그에 대한 latency 결과값이다. 실험은 4KB의 I/O 요청 크기에 대해 dd test 를 direct option을 적용하여 10000번의 direct I/O를 수행하였고, 그에 대한 I/O latency 평균을 계산한 결과값이다. 괄호 안의 수치는 전체 latency에서 디바이스 I/O의 latency를 뺀 값으로, 디바이스 I/O를 제외한 나머지 SAN I/O path에서의 latency를 비교할 수 있으며, 디바이스에 걸리는 read/write latency는 각각 12usec, 13usec이었다. 실험결과 Read의 경우 약 31.7(39.2)%의 latency를 줄였고, write는 약 28(33.8)%정도 줄였다. 이는 SCSI layer의 제거와 scheduler 간소화 작업을 통해 I/O path의 길이를 크게 줄일 수 있음을 보여준다.

I/O Type	SRP-SRPT (usec)	BRP-BRPT (usec)	Latency Reduction
Read	63 (51)	43 (31)	-31.7 (-39.2)%
Write	75 (62)	54 (41)	-28 (-33.8)%

Table 2: Latency Comparison between SRP-SRPT I/O path and BRP-BRPT I/O path

이 후 우리는 FIO micro-benchmark를 통해서 다양한 I/O pattern에 대해 실험을 수행하였다. 먼저 Figure 6은 기존의 solution인 SRP-SRPT와 본 논문에서 제안한 BRP-BRPT와의 성능 비교 결과이다. 실험 결과에 존재하는 bar들은 각각 SRP의 CFQ policy, SRP의 NOOP policy, local I/O 성능, 그리고 본 논문에서 제안한 BRP 성능의 결과를 비교한 것이다. 실험은 16개의 thread를 수행하였으며, 각각 3GB의 workload로 총 48GB의 data size를 가진다. I/O 는 sequential write, random write, sequential read, random read 총 4가지 type에 대해 수행했으며, request size 는 small(4KB), large(1MB)를 구분하여 실험을 수행하였다.

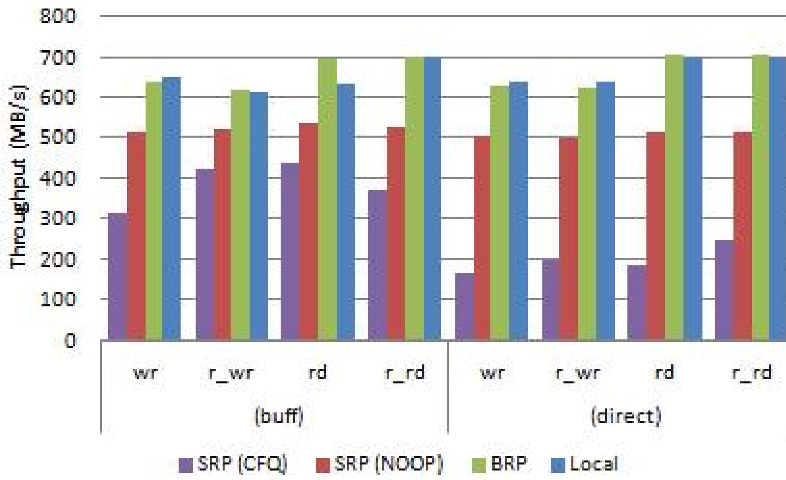
본 논문에서 사용하는 DRAM based SSD의 경우 driver단에서 이미 충분한 최적화가 구현되어있다. 실험결과에서 알 수 있듯이 local I/O subsystem에서 수행할 때의 성능은 large / small, read / write 에 크게 상관없이 uniform 하고 뛰어난 performance를 보여준다.

이러한 고성능 스토리지 환경에서 SRP-SRPT를 이용한 SAN환경 구축시, Linux 시스템의 default I/O scheduler인 CFQ policy가 큰 overhead로 작용하여 심각한 성능 저하가 발생하게 되며, 이는 실험 결과에서 잘 확인할 수 있다. buffered I/O인 경우 page cache에 의한 효과로 인해 아주 큰 성능 저하가 발생하지 않지만 page cache의 혜택을 크게 받지 못하는 random I/O pattern이나 page cache를 사용하지 않는 direct I/O의 경우 심각한 성능 저하현상이 발생하였으며, 이는 small size인 경우 더욱 심각하다. 따라서 우리는 기존의 SAN solution에서의 I/O scheduler policy를 NOOP으로 변경하여 I/O scheduler의 overhead를 최소화 하였고 실험을 통해 대부분의 I/O pattern에 대해서 큰 성능 향상을 확인할 수 있었다. 하지만 여전히 small size random I/O pattern의 경우 기존 I/O stack이 가지는 overhead로 인해 디바이스의 성능을

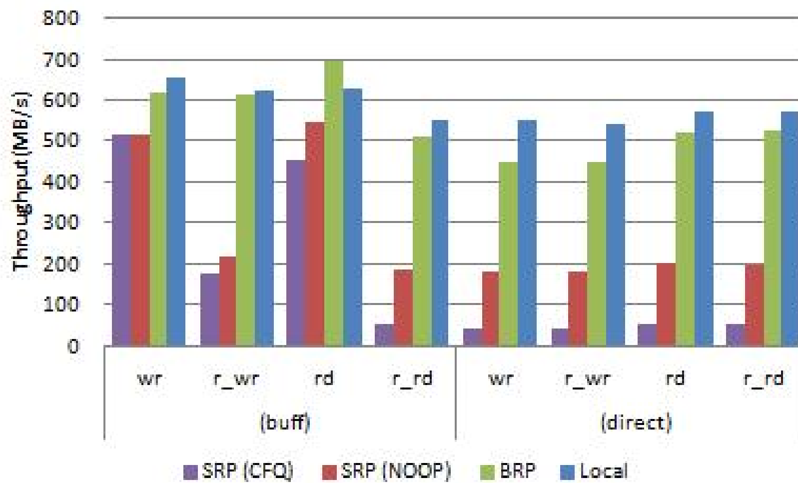
제대로 활용하지 못하고 있으며 Local I/O의 수행 결과 대비 큰 performance gap이 존재하는 것을 확인할 수 있다. 이에 우리는 앞서 제안한 최적화 방안들을 바탕으로 SAN I/O path를 최적화 하여 local I/O에 근접한 성능을 보여주는 새로운 SAN solution을 구현 하였다.

Figure 6 (a)는 large size data에 대한 I/O 성능 결과이다. 일반적으로 large size I/O의 경우 I/O 채널의 bandwidth를 잘 활용하고 있으며, 기존의 solution인 SRP에서도 충분히 나쁘지 않은 성능을 보여주고 있다. 하지만 BRP의 경우 거의 local I/O와 동일한 성능을 보여주고 있다. large size data의 경우 Initiator단에서 temporal merge를 수행하지 않으므로, 이는 I/O path 상의 software overhead의 제거와 parallelism에 대한 효과로 볼 수 있다. large size read의 경우 간혹 local에서 수행한 I/O 성능보다 더 뛰어날 때가 간혹 존재하게 되는데 이는 본 연구에서 사용한 디바이스의 I/O 처리 루틴이 성능 최적화를 위해 인터럽트 방식이 아닌 polling 방식으로 구현되어 있기 때문이다. 실제로 low latency 장비의 경우 polling 방식이 여러 면에서 더 나은 선택이 될 수 있지만, CPU와 같은 computing 자원을 더 많이 소모하게 된다. SAN환경을 구축할 경우 computing 자원의 소모가 분산되는 효과가 존재하게 되므로 이로 인한 성능저하가 발생하지 않는다.

Figure 6(b)는 small size data에 대한 I/O 성능 결과이다. 실제로 본 논문에서 진행한 연구는 이러한 small size I/O에 대한 기존 SAN I/O path의 문제점을 해결하는 것이었으며, 실험 결과에도 확인할 수 있듯이 큰 성능 향상이 있었고, local I/O 와 비교해도 나쁘지 않은 throughput을 달성하는데 성공하였다. 먼저 buffered I/O의 sequential write/read의 경우 기존 path에서의 성능도 나쁘지 않은 편이다.



(a) Large size I/O



(b) Small size I/O

Figure 6: Throughput comparison between SRP-SRPT and BRP-BRPT for a variety of I/O patterns

이는 각각 spatial merge와 pre-fetch등을 통해서 large size의 I/O를 수

행하기 때문이며, I/O 채널의 bandwidth를 잘 활용할 수 있기 때문이다. 문제는 merge나 pre-fetch등이 거의 수행되지 않는 random pattern small size I/O 와 small size direct I/O 이었으며, 이 경우 local I/O 대비 심각한 성능저하가 발생하였다. 이에 우리는 위에서 언급했던 3가지 I/O path 최적화를 적용한 새로운 SAN solution을 개발하였고, 최대 260%의 성능향상을 확인할 수 있었다. temporal merge의 경우 intensive하게 small size I/O가 수행될 경우 효과를 발휘 할 수 있으며, buffered I/O 방식의 random write이 가장 큰 혜택을 볼 수 있다. 이는 실험결과에도 잘 나타나고 있으며, 거의 local I/O에 근접하는 성능을 확인할 수 있다. 이 실험은 총 16개의 thread를 수행하여 얻은 결과로써 direct I/O의 경우 temporal merge가 동작하지 않는다. 따라서 이 경우 발생하는 성능향상은 SAN I/O path 상의 software overhead 제거와 parallelism 증대의 효과로 볼 수 있다.

5.2 BRP Performance Analysis

이번 section에서는 BRP에 적용했던 optimization들에 대한 성능 분석을 할 것이다. Figure 7은 thread 수 에 따른 performance 변화를 측정하고, 그에 따른 temporal merge의 효과를 분석하기 위한 그래프이다. 실험은 small sized random write pattern에 대해 진행하였으며, 정확성을 위해 memory의 buffer에 의한 이득을 반영하지 않는, direct I/O 방식으로 수행하였다. 여기서 나오는 BRP-3T는 추가적인 고려사항 없이 항상 temporal merge를 수행하도록 구현된 버전으로, 최적의 상황에서 temporal merge를 수행하는 BRP-3와 구분된다.

SRP와 BRP-1의 경우 앞서 언급했던 parallelism의 부족으로 인해 thread 수 가 증가하더라도 성능 상의 큰 이득이 없었다. 실험 결과에서

확인할 수 있듯이 thread 수가 16개 이상부터는 더 이상 throughput이 증가하지 않는다. 반면 parallelism을 고려한 BRP-2의 경우 thread 수가 증가함에 따라 throughput도 함께 증가하는 것을 확인할 수 있다. 이 후 temporal merge의 효율성과 성능을 분석하기 위해 BRP-2와 BRP-2기반에서 항상 temporal merge를 수행하도록 구현한 BRP-3T, 두 버전에 대해서 실험을 진행하였다. 실험 결과에서 알 수 있듯이 temporal merge는 intensive 한 I/O가 수행되지 않는다면, 오히려 temporal merge를 수행하지 않는 것 보다 성능 떨어 질 수 있다. 이는 temporal merge에 의해 merge된 여러 request들이 Target쪽에서 serial하게 처리되어 parallelism을 떨어트리기 때문이다. 아주 간단하게 예를 들자면, 동시에 수행 되는 4개의 I/O를 temporal merge를 수행하지 않는 경우 각각 서로 다른 work thread에서 parallel하게 처리될 수 있지만, 4개의 I/O요청이 하나로 merge되어 Target으로 전송된다면, 이는 하나의 work thread에서 serial하게 처리될 것이다. 이 경우 processing과정에서 serial하게 처리되는 overhead가 한번의 RDMA 수행으로 여러 I/O요청들의 데이터를 전송하는 이득보다 더 크기 때문에 성능저하가 발생하게 된다. 하지만 I/O가 intensive하게 수행되는 상황에서는 다수의 work thread에서 parallel하게 merge된 요청들을 처리할 수 있으므로, parallelism을 떨어트리지 않는다. 따라서 실험 결과에서도 알 수 있듯이 thread 수가 일정 수 이상을 넘어가게 되면 temporal merge가 더 뛰어난 throughput을 보여 주고 있으며, 이는 앞서 설명했던 바와 같이 local I/O 성능에 근접한 수치이다. buffered I/O에서 처리되는 write의 경우 asynchronous하게 처리되는 write-back동작에 의해 굉장히 intensive한 I/O가 내려오게 되며, 이 경우 temporal merge에 대한 효과가 더해져 더욱 뛰어난 성능을 나타내는 것이다. 따라서 우리는 이러한 I/O intensive한 상황을

판단할 수 있는 기준을 정할 필요가 있었고, 현재 Active하게 처리되고 있는 request의 수를 counting하여 일정 값 이상일 경우 temporal merge를 수행하도록 구현하였다. 이와 같이 기준이 되는 적정한 threshold값은 일종의 heuristic으로 수 차례 반복된 실험을 통해 그 값을 취할 수 있었고 이러한 최적의 threshold 값을 반영한 버전이 BRP-3이다.

이 후 우리는 BRP 성능 분석을 위한 두 번째 실험으로 각각의 최적화 방안에 대한 효과를 확인해보았다. Figure 8 은 각 최적화 방안들의 성능을 비교한 것이다. 각 실험은 small size random I/O pattern에 대해서 16개의 thread로 수행하였고, 각각의 패턴에 대해서 local 환경에서 수행한 I/O throughput을 기준으로 normalized 된 performance를 계산한 결과값이다. 이를 통해 각각의 최적화를 적용하였을 경우 local I/O를 기준으로 얼마만큼 성능이 향상했는지에 대해 살펴볼 수 있다.

buffered I/O의 random write의 경우, BRP-1만 적용하더라도 큰 성능상의 이득이 존재한다. 이는 buffered I/O 방식에서는 단순히 I/O path를 줄여 latency를 줄이는 것만으로도 page cache의 활용도를 크게 증가시킬 수 있음을 뜻하며, 이는 local 대비 약 84% 정도의 성능으로 35%인 SRP보다 크게 증가한 수치이다. direct I/O에서의 BRP-1의 경우 단순히 latency가 줄어든 비율과 엇비슷한 성능향상을 확인할 수 있었으며, write와 read 각각은 local 대비 50%, 47%정도의 성능이며, 33%, 35%인 SRP에 비해 어느 정도 증가된 수치이다.

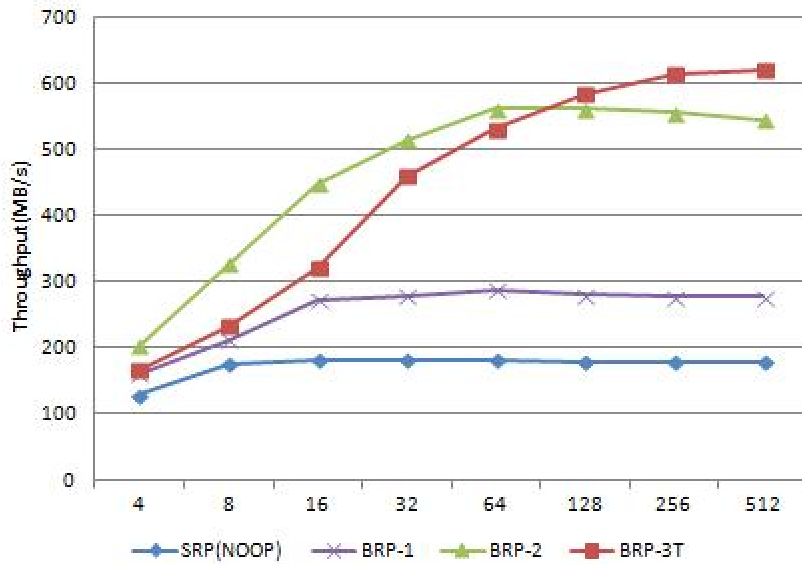


Figure 7: Performance analysis with different numbers of threads for random write pattern

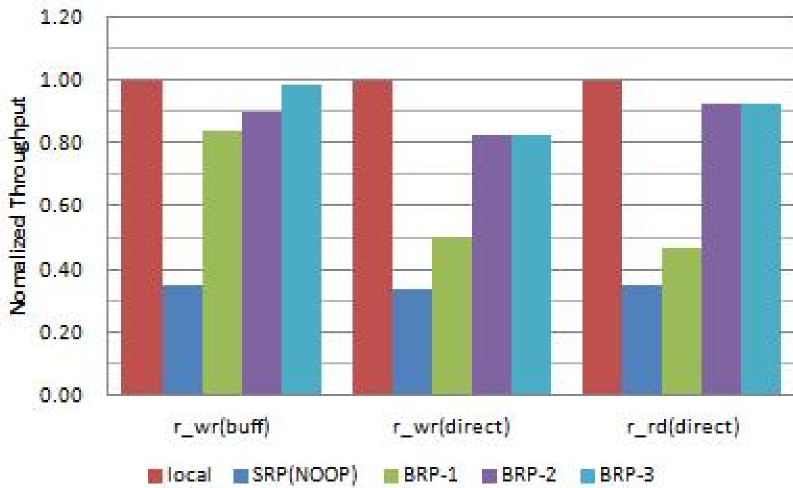


Figure 8: Performance analysis for three optimizations in BRP

Direct I/O의 경우 parallelism을 증가시킨 최적화인 BRP-2에서 큰 성능향상을 확인할 수 있다. 이는 parallel한 I/O수행을 통해 고성능 스토리지장비의 뛰어난 bandwidth를 제대로 활용할 수 있게 된 덕분이다. write와 read 각각의 수치는 local I/O 대비 약 82%, 92% 정도의 성능이며, Figure 7 에서 확인했듯이 thread 수를 증가시킴에 따라 디바이스 bandwidth의 활용도를 더 높일 수 있어 성능은 더욱 증가하게 된다. buffered write의 경우 BRP-2에 의해 약간의 성능향상을 확인할 수 있었으며, 이 값은 local I/O 대비 90% 정도의 성능이다. 언뜻 보기에는 BRP-2의 parallelism 최적화로 인한 성능 향상이 별로 없는 것처럼 보이겠지만, 사실 이 경우는 BRP-1에 의해 극대화된 메모리 효과 때문에 BRP-2로 인한 성능 향상 부분이 많이 가려진 것이다. 실제로 실험을 통해 확인한 결과 memory size를 크게 줄였을 경우 BRP-1의 성능은 많이 줄어들지만, BRP-2의 성능은 아주 약간만 줄어든 것을 확인할 수 있었다.

마지막으로 BRP-3는 temporal merge를 적용한 최적화이며, 앞서 설명했던 바와 같이 intensive한 I/O 상황에서 효과를 발휘한다. 16개의 thread에 의해 수행되는 direct I/O 환경은 intensive한 I/O상황을 발생시키지 않는다. 따라서 temporal merge를 수행하지 않으므로 성능은 BRP-2와 동일하게 될 것이다. 하지만 이 경우도 마찬가지로 Figure 7에서 확인할 수 있듯이 thread 수가 증가할 경우 temporal merge의 효과가 나타날 것이다. Buffered write의 경우 asynchronous 한 write-back 수행으로 인해 intensive 한 I/O가 발생하게 되므로 temporal merge 최적화의 가장 큰 수혜자라고 볼 수 있다. 이 경우의 성능은 local I/O 대비 약 98% 이며 local I/O 성능에 거의 근접한 수치라고 볼 수 있다.

5.3 Real Workload Evaluation

마지막으로 우리는 Real Workload에 대한 Benchmark로써 Database환경을 구축하여 실험을 수행하였다. 실험환경은 BenchmarkSQL tool을 이용하여 TPC-C benchmark를 수행하였고, 데이터베이스 서버 프로그램으로는 postgreSQL을 사용하였다[2,24,18]. Warehouse의 수는 320개로 약 32GB data 사이즈를 유지했으며, terminal 수를 증가시킴에 따라 평균 transactions per minute 를 측정하였다. Figure 9은 8-128까지 terminal의 수를 증가시킴에 따른 tpmC를 SRP와 BRP에 대해서 비교한 것이다. 실험 결과에서 확인할 수 있듯이 모든 경우에 대해서 BRP가 SRP보다 뛰어난 성능을 보여주고 있으며, 약 35%~40% 정도 성능이 향상되었다. 하지만 이러한 성능 향상 정도는 앞서 수행한 micro-benchmark 실험에서의 성능 향상에 훨씬 미치지 못하고 있다. 이 수치는 software overhead를 제거함으로써 latency를 줄인 효과와 유사한 비율이며, 실제로 trace를 분석한 결과 parallelism과 temporal merge의 효과는 크게 나타나지 않았음을 확인할 수 있었다. 일단 TPC-C의 경우 read intensive한 workload를 가지는 benchmark로써, temporal merge에 대한 이득이 많이 볼 수 없는 구조이며, 무엇보다 데이터 베이스 layer에서 고성능 스토리지환경에 불필요한 버퍼링, delayed write등의 최적화를 수행함으로써 intensive한 디바이스 I/O가 발생하지 않는 다는 점이 가장 큰 문제였다. 이는 어떻게 보면 본 연구에서 진행한 block layer 최적화의 한계를 보여주는 것이며, 고성능 스토리지의 성능을 최대한 활용하기 위한 환경을 구축하고 최적화를 진행하기 위해서는 block layer뿐만 아니라 file system이나 database와 같은 upper layer에서의 최적화도 함께 이루어 져야 한다는 것을 보여주는 것이다.

Figure 9: Performance comparison for TPC-C benchmark on PostgreSQL

제 6 장 관련 연구

NBD (Network Block Device)는 네트워크 블록 디바이스를 위한 솔루션 중 하나이며 [26], 본 연구에서 구현한 BRP와 유사하게 Block Layer 상에서 스토리지 가상화를 수행하고 있다. 하지만 NBD는 general purpose를 가지는 솔루션으로 고성능 디바이스에 대한 특별한 최적화를 고려하고 있지 않고 있다. 이는 기존의 block I/O path를 따르고 있으며, elevator merge나 plug-unplug mechanism 등 고성능 디바이스에 적합하지 않은 I/O 스케쥴러 동작들을 그대로 사용하고 있다. 또한 multiple I/O를 parallel하게 처리할 수 있는 핸들러도 따로 구현되어 있지 않고 있다. 그리고 NBD는 TCP/IP기반의 솔루션으로서 Infiniband와 같은 high speed network을 사용하기 위해서는 ip_over_ib 같은 translation layer가 필요한데 이 역시 추가적인 translation overhead를 야기 시킨다.

제 7 장 향후 과제

우리는 고성능 스토리지 장비의 성능을 최대한 활용할 수 있는 SAN solution 개발을 목적으로 본 연구를 진행하고 있으며, 앞에서 언급한 바와 같이 현재는 단순히 성능만을 고려한 prototype형태로 구현이 완료된 상황이다. 이 후 실제 상용화 가능하도록 SAN 환경 구성에 필수적인 reliability 관련 기술들과 management 측면을 고려하여 최종적인 SAN Solution을 개발 하려 한다. 또한 Linux 뿐만 아니라 Window 등의 다른 OS와의 호환이 가능하도록 그에 맞는 driver를 구현할 예정이다. 본 논문에서 제안한 BRP는 리눅스 block layer의 bio structure기반의 전송 프로토콜이지만 실제 전송 packet에 저장되는 내용은 OS를 불문하고 I/O 수행에 필수적인 general한 값들이다. 따라서 Linux가 아닌 다른 OS라 할지라도 호환성을 유지하는데는 큰 어려움이 없을 것이다. 그리고 본 논문에서 제안한 SAN optimization 관련 테크닉들을 SAN 뿐만 아니라 네트워크를 통한 block layer replication solution 과 같은 다른 solution 들에도 적용해볼 계획이다.

또한 고성능 스토리지의 성능을 최대한 활용할 수 있는 시스템 구축을 위해서는 block layer뿐만 아니라 file system이나 application level에서의 최적화도 함께 진행될 필요가 있다. 실제로 우리는 모든 computer system layer에서 고성능 스토리지 시스템의 특성을 잘 이용할 수 있는 최적화 방안에 대해 계속 연구를 진행하고 있다.

마지막으로 이러한 최적화에 대한 연구를 더욱 발전시켜 이 후 등장할 차세대 스토리지에 대해서도 그대로 적용할 수 있도록 할 것이며 미래의 고성능 스토리지 에 대한 효율적인 환경을 구축하는데 기여할 것이다.

제 8장 결론

우리는 고성능 스토리지 시스템의 뛰어난 성능을 최대한 잘 활용할 수 있는 SAN solution에 대한 디자인 및 구현에 관한 연구를 진행하였다. 먼저 고성능 스토리지 시스템을 기존의 SAN solution에 그대로 적용해 보았으며, 초고속 네트워크 장비인 Infiniband를 사용했음에도 불구하고, 심각한 성능 저하 현상이 발생하는 것을 확인하였다. 이에 우리는 그에 대한 원인을 분석하고, 문제를 해결할 수 있는 새로운 SAN Solution을 제안 하였다.

고성능 스토리지 시스템을 기반으로 하는 SAN환경에서 필요한 첫 번째 optimization은 스토리지의 low-latency 특성에 맞는 짧은 I/O path 를 가져야 한다는 것이다. 이에 우리는 기존 SAN I/O path에 존재하는 Software overhead를 제거하여 전체 I/O path를 단축시키는 일을 수행하였다. 이를 위한 작업으로 먼저 기존 SAN I/O path의 SCSI layer를 제거하여 수행 코드의 길이를 줄였고, 기존의 scsi command 기반의 전송 프로토콜을 활용하는 것이 아니라 block layer에서의 bio structure를 기반으로 하는 RDMA 전송 프로토콜인 Block RDMA Protocol(BRP)을 구현하여 데이터 전송을 수행하도록 했다. 또한 기존 I/O scheduler에서 delay를 이용하여 I/O 최적화를 구현하는 동작들 역시 고성능 스토리지를 위한 빠른 I/O path구현에 방해가 된다. 이에 우리는 기존의 I/O scheduler를 과감히 포기하고 최대한 빠르게 데이터를 전송할 수 있는 simple한 I/O scheduler를 구현하였고, 이러한 최적화 과정들을 통해 SAN 환경에서의 I/O path를 상당히 단축 시켰으며, I/O latency를 크게 줄일 수 있었다.

두 번째로 꼭 필요한 최적화는 고성능 스토리지 장비의 internal

parallelism을 잘 인식하고 활용할 수 있는 I/O 처리 mechanism이다. 기존 SAN 환경에서 Target 쪽에서의 I/O 요청을 처리 과정을 분석했을 때 parallelism이 많이 부족한 것을 확인할 수 있었고, 특히 디바이스 I/O를 수행하는 부분에서 serial하게 처리되는 경향이 있어, 고성능 스토리지의 뛰어난 bandwidth를 제대로 활용하지 못하고 있었다. 이에 우리가 제안하는 SAN Solution에서는 target에서 I/O 요청을 처리하기 위한 일련의 모든 과정들을 thread-pool형태로 관리되는 work-thread에서 parallel하게 처리 할 수 있도록 구현하였다.

고성능 SAN환경에 필요한 마지막 최적화는 Initiator와 Target간의 효율적인 RDMA 데이터 전송 메커니즘이다. 디스크 기반의 기존 SAN환경에서는 디스크 I/O가 전체 I/O latency를 dominate하기 때문에 네트워크 overhead는 완전히 가려져 드러나지 않았다. 하지만 low-latency 장비에서는 high speed network장비를 쓰더라도 전체 시스템에 영향을 미치는 overhead가 될 수 있으며, 특히 small size data 전송의 경우 request당 network의 processing overhead 가 성능에 영향을 미치는 경우가 발생하게 된다. 이에 우리는 temporal merge 테크닉을 적용하여 서로 다른 small size I/O 요청들에 대해서 공간적 연속성에 상관없이 merge를 수행하여 jumbo command 를 구성해 RDMA를 수행하는 최적화를 구현하였다. 이러한 temporal merge 기술을 통해서 network overhead를 줄여 전체 I/O 성능이 향상될 수 있도록 하였다.

우리는 위에서 언급한 최적화 기법들을 prototype 형태로 구현해 실험을 진행하였으며, read, write 각각에 대해서 약 39.2%, 33.8%의 I/O latency를 단축시킬 수 있었고, bandwidth 측면에서는 small size I/O의 경우 read, write 각각 약 274%, 280% 정도의 성능향상을 확인할 수 있었다.

참고 문헌

[1] ASHISH PALEKAR, DESIGN AND IMPLEMENTATION OF ALINUX SCSI TARGET FOR STORAGE AREA NETWORKS, In ALS' 01

[2] BenchmarkSQL, http://sourceforge.net/projects/benchmarksql/

[3] Bob Woodruff, Introduction to the InfiniBand Core Software, Linux Symposium, Vol. 2, July 2005, p271-282

[4] Burr, G. W., Overview of candidate device technologies for storage-class memory, IBM Journal of Research and Development, Vol. 52 Issue 4.5, July 2008, p449-464

[5] CAULFIELD, A. M., ET AL. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In MICRO'10 (2010), pp. 385 – 395.

[6] Chaarawi., S. Using a shared Storage class memory device to improve the reliability of RAID arrays, In PDSW 2010, p1-5

[7] Cruz, J., Jr., Leader-follower strategies for multilevel systems, Automatic Control, IEEE Transactions on, Vol. 23, Issue 2, Apr 1978

p244 - 255

[8] DP Bovet, M Cesati, Understanding the Linux Kernel, 3rd Edition

[9] Freitas, R. F., Storage-class memory: The next Storage system technology, IBM Journal of Research and Development, Vol. 52 Issue 4.5, July 2008, p439-447

[10] Fusion-io, ioDrive, http://www.fusionio.com/products/iodrive/

[11] IBM, Introduction to Storage Area Networks, http://www.redbooks.ibm.com/redbooks/pdfs/sg245470.pdf

[12] Jens Axboe, Flexible IO Tester, http://git.kernel.dk/?p=fio.git

[13] John Ousterhout, The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM, ACM SIGOPS Operating Systems Review, Vol. 43 Issue 4, January 2010, p 92-105

[14] Mellanox, Building a Scalable Storage with InfiniBand, http://www.mellanox.com/relateddocs/whitepapers/WP_Scalable_Storage_InfiniBand_Final.pdf

[15] Mellanox, InfiniBand Storage, http://www.mellanox.com/pdf/whitepapers/I-nfiniBand_Storage_WP_0

50.pdf

[16] Mellanox, Introduction to InfiniBand,
http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[17] Mike Ko, Technical Overview of iSCSI Extensions for RDMA (iSER) & Datamover Architecture for iSCSI (DA), RDMA Consortium 2003

[18] PostgreSQL, http://www.postgresql.org/

[19] RAMCloud Project,
https://ramcloud.stanford.edu/wiki/display/ramcloud/RAMCloud

[20] Ru Fang, High Performance Database Logging using Storage Class Memory, In ICDE 2011, p1221-1231

[21] SAP, SAP HANA,
http://www.sap.com/solutions/technology/in-memory-computing-platform/index.epx

[22] SCST, Generic SCSI Target Subsystem for Linux,
http://scst.sourceforge.net/

[23] TAEJININFOTECH, HHA 3804, http://www.taejin.co.kr

[24] TPC-C benchmark, http://www.tpc.org/tpcc/

[25] Young Jin Yu, Exploiting Peak Device Throughput from Random Access Workload, In HotStorage'12

[26] P. Machek. Network Block Device (TCP version) http://nbd.sourcefor-ge.net

ABSTRACT

Today's server environments consist of many machines constructing clusters for distributed computing system or storage area networks (SAN) for effectively processing or saving enormous data. In these kinds of server environments, backend-storages are usually the bottleneck of the overall system. But it is not enough to simply replace the devices with better ones to exploit their performance benefits. In other words, proper optimizations are needed to fully utilize their performance gains. In this work, we first applied a high performance device as a backend-storage to the existing SAN solution, and found that it could not utilize the low latency and high bandwidth of the device, especially in case of small sized random I/O pattern even though a high speed network is used. So, we propose a new design that contains three optimizations: 1) removing disk legacies to lower I/O latency; 2) parallelism to utilize the high bandwidth of the device; 3) temporalmergemechanismto reduce network overhead. We implemented them as a prototype and found that our solution makes substantial performance improvements in terms of both the latency and bandwidth.

Keywords: SAN, Stroage Area Network, Fast Storage, RDMA Data Transfer

Student Number: 2011-20946

감사의 글

지난 2년 동안의 석사과정은 정말 좋은 분들과 함께 연구할 수 있었던 즐거운 시간이었습니다. 먼저 이렇게 좋은 환경에서 배움의 기회를 주신 두 분 교수님께 감사드립니다. 그리고 제 옆자리에서 항상 온화한 미소로 큰 가르침을 주신 유느님, 영진이형께 정말 감사드립니다. 제가 형 옆자리에 앉게 된 건 정말 신의 한 수였다고 생각합니다. 그리고 항상 즐거움을 주시는 연구실 마스코트 신규형, 연구실 생활하면서 정말 여러가지 많은 도움을 받았던 것 같습니다. 또 형 덕분에 정말 즐겁게 연구실 생활을 할 수 없었습니다. 그리고 태진에서 많은 가르침과 도움을 주신 동인이형께도 정말 감사드립니다. 덕분에 학교에서 배울 수 없었던 많은 지식들을 배울 수 있었고 졸업도 무난하게 할 수 있었던 거 같습니다. 그 밖에 많은 시간을 함께 하지는 못했지만 형석이형, 은성이형, 민규형, 용경이형, 임영이 누나, 인순이 누나께도 감사드리구요. 처음 연구실 들어왔을 때 잘 챙겨 주고 알려 주신 선배님들 신웅형, 완희형, 승미, 승민이, 운태 정말 감사합니다. 그리고 제가 정말 사랑하는 우리 동기님들, 국태형, 세훈이형, 찬호 윤희, 그리고 동기처럼 느껴지는 내영이 까지, 덕분에 연구실 생활이 정말 즐거웠습니다. 앞으로도 종종 연락하면서 친하게 지냈으면 좋겠습니다. 그리고 항상 열심히 하시는 우리 후배님들 설웅형, 성재형, 용석이, 동유, 민영이, 계신이, 문봉이, 시봉이, 한울이, 지웅이, 다혜에게도 감사드립니다. 마지막으로 언제나 저를 믿어주고 응원해주시는 부모님과 형 완전 사랑합니다.