



Red Hat Enterprise Linux 6

성능 조정 가이드

Red Hat Enterprise Linux 6에서 서브시스템 처리량을 최적화

위험 4.0

Red Hat Enterprise Linux 6 성능 조정 가이드

Red Hat Enterprise Linux 6에서 서브시스템 처리량을 최적화
위험 4.0

Red Hat 내용 전문가

위험이

Don Domingo

Laura Bailey

법적 공지

Copyright © 2011 Red Hat, Inc. and others.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

초록

성능 조정 가이드에서는 Red Hat Enterprise Linux 6를 실행하고 있는 시스템의 성능을 최적화하는 방법에 대해 설명합니다. 또한 Red Hat Enterprise Linux 6의 성능 관련 업그레이드도 소개하고 있습니다. 이 가이드에는 현장에서의 시험 및 검증된 절차가 들어 있지만 Red Hat은 이를 프로덕션 환경에 적용하기 전 테스트 환경에서 계획된 모든 설정을 정확하게 테스트할 것을 권장합니다. 또한 모든 데이터 및 튜닝 전 설정을 백업해 두어야 합니다.

차례

1장. 개요	4
1.1. 대상	4
1.2. 수평적 확장성	5
1.2.1. 병렬 컴퓨팅	5
1.3. 분산 시스템	6
1.3.1. 통신	6
1.3.2. 스토리지	7
1.3.3. 통합 네트워크	8
2장. RED HAT ENTERPRISE LINUX 6 성능 특징	10
2.1. 64 비트 지원	10
2.2. TICKET SPINLOCKS	10
2.3. 동적 목록 구조	11
2.4. 텍리스 커널	11
2.5. 컨트롤 그룹	12
2.6. 스토리지 및 파일 시스템 개선 사항	13
3장. 시스템 성능 모니터링 및 분석	15
3.1. PROC 파일 시스템	15
3.2. GNOME 및 KDE 시스템 모니터	15
3.3. 내장된 명령행 모니터링 도구	16
3.4. TUNED 및 KTUNE	17
3.5. 애플리케이션 프로파일러	18
3.5.1. SystemTap	18
3.5.2. OProfile	19
3.5.3. Valgrind	19
3.5.4. Perf	20
3.6. RED HAT ENTERPRISE MRG	20
4장. CPU	22
토폴로지	22
스레드	22
인터럽트	22
4.1. CPU 토폴로지	22
4.1.1. CPU 및 NUMA 토폴로지	22
4.1.2. CPU 성능 튜닝	24
4.1.2.1. taskset으로 CPU 친화도 설정	26
4.1.2.2. numactl로 NUMA 정책 제어	26
4.1.3. numastat	28
4.1.4. NUMA 친화도 관리 데몬 (numad)	29
4.1.4.1. numad의 장점	30
4.1.4.2. 운영 모드	30
4.1.4.2.1. numad를 서비스로 사용	30
4.1.4.2.2. numad를 실행 파일로 사용	30
4.2. CPU 스케줄링	31
4.2.1. 실시간 스케줄링 정책	31
4.2.2. 일반 스케줄링 정책	32
4.2.3. 정책 선택	33
4.3. 인터럽트 및 IRQ 튜닝	33
4.4. RED HAT ENTERPRISE LINUX 6에서 NUMA 기능 강화	34
4.4.1. 베어 메탈 및 확장성 최적화	34
4.4.1.1. 토폴로지 인식 기능 향상	34

4.4.1.2. 멀티 프로세서 동기화 기능 개선	35
4.4.2. 가상화 최적화	35
5장. 메모리	36
5.1. HUGETLB (HUGE TRANSLATION LOOKASIDE BUFFER)	36
5.2. HUGE PAGES 및 TRANSPARENT HUGE PAGES	36
5.3. 프로파일 메모리 사용에 VALGRIND 사용	37
5.3.1. Memcheck로 메모리 사용량 프로파일링	37
5.3.2. Cachegrind로 캐시 사용량 프로파일링	38
5.3.3. Massif를 사용하여 힙 및 스택 영역 프로파일링	39
5.4. 용량 튜닝	40
5.5. 가상 메모리 튜닝	43
6장. 입/출력	45
6.1. 특징	45
6.2. 분석	45
6.3. 도구	46
6.4. 설정	50
6.4.1. CFQ (Completely Fair Queuing)	50
6.4.2. 데드라인 I/O 스케줄러	52
6.4.3. Noop	53
7장. 파일 시스템	55
7.1. 파일 시스템 튜닝 시 고려 사항	55
7.1.1. 포맷 옵션	55
7.1.2. 마운트 옵션	55
7.1.3. 파일 시스템 유지 보수	56
7.1.4. 애플리케이션 유의 사항	57
7.2. 파일 시스템 성능 프로파일	57
7.3. 파일 시스템	58
7.3.1. Ext4 파일 시스템	58
7.3.2. XFS 파일 시스템	59
7.3.2.1. XFS의 기본적 튜닝	59
7.3.2.2. XFS의 고급 튜닝	59
7.4. 클러스터링	61
7.4.1. GFS 2 (Global File System 2)	62
8장. 네트워킹	64
8.1. 네트워크 성능 개선	64
8.1.1. RPS (Receive Packet Steering)	64
8.1.2. RFS (Receive Flow Steering)	64
8.1.3. TCP thin-stream의 getsockopt 지원	64
8.1.4. 투명 프록시 (TProxy) 지원	65
8.2. 네트워크 설정 최적화	65
8.2.1. 소켓 수신 버퍼 크기	66
8.3. 패킷 수신 개요	67
8.3.1. CPU/캐시 친화도	68
8.4. 일반적인 큐/프레임 손실 문제 해결	68
8.4.1. NIC 하드웨어 버퍼	68
8.4.2. 소켓 큐	69
8.5. 멀티캐스트에 있어서 고려할 사항	69
부록 A. 고친 과정	71

1장. 개요

성능 조정 가이드는 Red Hat Enterprise Linux의 설정 및 최적화에 있어 포괄적인 참조 문서입니다. 이 릴리즈에는 Red Hat Enterprise Linux 5 성능 기능에 대한 정보도 들어 있지만 여기에 있는 모든 지시 사항은 Red Hat Enterprise Linux 6에 특정된 것입니다.

이 문서는 Red Hat Enterprise Linux의 특정 서브 시스템에 대해 설명하는 장으로 나뉘어져 있습니다. 성능 조정 가이드는 서브 시스템마다 다음과 같은 세 가지 주요 주제에 초점을 맞추고 있습니다:

특징

각 서브시스템 장에서는 Red Hat Enterprise Linux 6에서 고유의 (또는 다른 방법으로 구현된) 성능 기능에 대해 설명합니다. 또한 Red Hat Enterprise Linux 5를 통해 특정 서브 시스템의 성능이 크게 개선된 Red Hat Enterprise Linux 6 업데이트에 대해서도 설명합니다.

분석

이 문서에서는 각각의 특정 서브시스템 별 성능 지표도 소개합니다. 이러한 지표의 일반적인 값은 특정 서비스의 컨텍스트에 설명되어 있어서 실제 제품 시스템에서의 중요도의 이해를 돕습니다.

또한 성능 조정 가이드는 서브시스템의 성능 데이터 (즉 프로파일링)의 다른 검색 방법도 소개합니다. 여기서 보여드리는 프로파일링 도구 중 일부는 다른 곳에서 보다 자세하게 설명되어 있습니다.

설정

이 문서에서 가장 중요한 정보는 Red Hat Enterprise Linux 6의 특정 서브시스템의 성능을 조정하는 방법에 대한 지시 사항이라고 생각됩니다. 성능 조정 가이드에서는 특정 서비스에 대해 Red Hat Enterprise Linux 6 서브시스템을 세밀히 조정하는 방법에 대해 설명합니다.

특정 서브 시스템의 성능을 조정하면 다른 서브시스템의 성능에 악영향을 줄 수 있다는 것을 염두해 두십시오. Red Hat Enterprise Linux 6의 기본 설정은 중간 부하에서 실행되고 있는 대부분의 서비스에 대해 최적화되어 있습니다.

성능 조정 가이드에서 나열된 절차는 랩과 현장 모두에서 Red Hat 엔지니어가 철저히 테스트했지만 Red Hat은 프로덕션 서버에 적용하기 전 안전한 테스트 환경에서 계획하는 모든 설정을 정확하게 테스트할 것을 권장합니다. 또한 시스템을 튜닝하기 전 모든 데이터 및 설정 정보를 백업해야 합니다.

1.1. 대상

이 문서는 다음과 같은 두 가지 유형의 독자를 대상으로 하고 있습니다:

시스템/비즈니스 분석가

이 문서에서는 Red Hat Enterprise Linux 6 성능 기능을 높은 수준에서 설명하고 (기본값 및 최적화되었을 때 모두에서) 특정 작업부하의 서브 시스템을 수행하는 방법에 대해 충분한 정보를 제공합니다. Red Hat Enterprise Linux 6 성능 기능에 대한 상세한 설명을 통해 잠재적 고객 및 세일즈 엔지니어는 허용 가능한 수준에서 리소스 집약적 서비스를 제공하기 위해 이 플랫폼의 적합성을 이해할 수 있게 합니다.

또한 성능 조정 가이드는 가능한 각 기능에 대한 보다 상세한 문서로의 링크를 제공합니다. 세부적으로 사용자는 성능 기능을 충분히 이해하여 Red Hat Enterprise Linux 6의 배포 및 최적화에 있어서 높은 수준의 전략을 형성할 수 있습니다. 따라서 사용자는 인프라 제안을 개발 및 평가할 수 있습니다.

이는 기능에 초점을 둔 문서이므로 Linux 서브 시스템과 기업 수준의 네트워크를 고도로 이해할 수 있는 사용자를 위한 것입니다.

시스템 관리자

이 문서에 나열된 절차는 RHCE^[1] 기술 수준 (또는 이에 상응하는 즉 Linux 배포 및 관리경험 3-5년)을 갖는 시스템 관리자를 위한 것입니다. 성능 조정 가이드에서는 각 설정의 효과에 대해 최대한 상세하게 설명하고 있습니다. 즉 발생할 수 있는 성능 장단점을 설명하고 있습니다.

성능 튜닝에 있어서 기본 기술은 서버 시스템을 분석 및 튜닝하는 방법을 알고 있는 것에 있지 않습니다. 오히려 성능 튜닝에 익숙한 시스템 관리자는 특정 목적을 위해 Red Hat Enterprise Linux 6 시스템을 최적화하고 균형을 갖게 하는 방법을 알고 있습니다. 이는 특정 서브시스템의 성능을 개선하기 위한 설정을 구현할 때 장단점 및 성능 저하에 대해서도 알고 있다는 것을 의미합니다.

1.2. 수평적 확장성

Red Hat Enterprise Linux 6의 성능 개선에 있어서 Red Hat은 확장성에 초점을 두고 있습니다. 성능 개선 기능은 워크로드의 범위에서 다른 영역으로 — 즉 고립된 웹 서버에서 서버팜의 메인 프레임에 이르기까지 플랫폼의 성능에 영향을 미치는 방식에 따라 평가되고 있습니다.

확장성에 초점을 두어 Red Hat Enterprise Linux는 다른 유형의 워크로드와 목적에 적합한 다양성을 유지할 수 있습니다. 동시에 비즈니스가 성장하고 워크로드가 확대됨에 따라 서버 환경을 재구성하는 것은 (비용 및 공정 측면에서) 적은 비용으로 보다 더 인지 가능하게 됨을 의미합니다.

Red Hat은 수평적 확장성 및 수직적 확장성 모두에서 Red Hat Enterprise Linux를 개선해 왔습니다. 하지만 수평적 확장성이 더 일반적으로 적용 가능합니다. 수평적 확장성의 개념의 기반이 되고 있는 것은 여러 표준 컴퓨터를 사용하여 과도한 워크로드를 분산하고 성능 및 안정성을 개선하는 것입니다.

일반적인 서버 팜에서 이러한 표준 컴퓨터는 1U 랙 마운트 서버 및 블레이드 서버 형태를 취합니다. 각각의 표준 컴퓨터는 간단한 2 소켓 시스템과 같은 소형이 될 수 있습니다. 하지만 일부 서버 팜은 소켓 수가 많은 대형 시스템을 사용하기도 합니다. 일부 엔터프라이즈급 네트워크는 대형 및 소형 시스템을 결합하기도 합니다. 이러한 경우 대형 시스템은 고성능 서버 (예: 데이터베이스 서버)이고 소형 시스템은 전용 애플리케이션 서버 (예: 웹 또는 메일 서버)입니다.

이러한 유형의 확장성은 IT 인프라의 확장을 단순화합니다. 적당한 로드를 갖는 중소 기업에서는 피자 박스 서버 2 대로 모든 요구를 충족시킬 수 있습니다. 직원 수 증가, 운영 확장, 매출 크기 증가 등으로 IT 요구 사항은 볼륨 및 복잡성 모두에서 증가하게 됩니다. 수평적 확장성을 사용하면 기존 시스템과 (거의) 동일한 설정으로 추가 시스템을 간단하게 배포할 수 있습니다.

요약하면 수평적 확장성은 시스템 하드웨어 관리를 간소화하는 추상적 레이어를 추가합니다. Red Hat Enterprise Linux 플랫폼을 수평으로 확장하도록 개발하여 IT 서비스의 기능 및 성능 향상은 새롭고 쉽게 설정되는 시스템을 추가하는 것과 같이 간단한 것이 될 수 있게 합니다.

1.2.1. 병렬 컴퓨터

Red Hat Enterprise Linux의 수평 확장성에서 혜택을 받는 사용자는 시스템 하드웨어 관리를 간소화할 뿐만 아니라 현재 하드웨어 발달의 동향에 있어서 수평적 확장성이 적절한 개발 철학이 됩니다.

다음 사항을 생각해 봅시다. 대부분의 복잡한 엔터프라이즈 애플리케이션은 작업 간의 다른 조정 방법으로 동시에 수행해야 하는 수천 개의 작업이 있습니다. 초기 컴퓨터는 이러한 모든 작업을 처리하는 단일 코어 프로세서를 사용했지만 사실상 현재 사용 가능한 모든 프로세서는 멀티 코어를 사용합니다. 실질적으로 현대 컴퓨터는 멀티 코어를 단일 소켓에 두고 단일 소켓 데스크탑이나 노트북을 멀티 프로세서 시스템으로 만듭니다.

2010년에는 표준 Intel 및 AMD 프로세서는 2-16개의 코어로 사용할 수 있었습니다. 이러한 프로세서는 피자 박스 또는 블레이드 서버가 일반적으로 되어 현재 40 개 보다 많은 코어를 포함할 수 있습니다. 이러한 저비용의 고성능 시스템은 대형 시스템 기능과 특징을 메인 스트림으로 제공합니다.

시스템의 최상의 성능과 활용도를 달성하려면 각 코어는 항상 작업을 실행해야 합니다. 즉 이는 32 코어 블레이드 서버를 활용하려면 32개의 개별적 작업을 실행해야 한다는 것을 의미합니다. 하나의 블레이드

새시에 이러한 32 코어 블레이드가 10개 들어 있을 경우 전체 구성은 최소 320개의 작업을 동시에 처리할 수 있습니다. 이러한 작업이 단일 작업의 일부일 경우 이를 조정해야 합니다.

Red Hat Enterprise Linux는 하드웨어 개발 동향에 잘 적응하고 여기서 기업이 완전한 혜택을 누릴 수 있도록 하기 위해 개발되었습니다. 1.3절. “분산 시스템”에서는 Red Hat Enterprise Linux의 수평적 확장성을 가능하게 하는 기술에 대해 상세하게 설명합니다.

1.3. 분산 시스템

수평적 확장성을 완전히 실현하기 위해 Red Hat Enterprise Linux는 분산 컴퓨팅의 여러 구성 요소를 사용합니다. 분산 컴퓨팅을 구성하는 기술은 다음과 같은 세 개의 층으로 구분됩니다:

통신

수평적 확장성에서는 많은 작업이 (병렬로) 동시에 실행되어야 합니다. 따라서 이러한 작업은 프로세스 간 통신에서 작업을 조정해야 합니다. 또한 수평적 확장성의 플랫폼은 여러 시스템에 걸쳐 작업을 공유할 수 있어야 합니다.

스토리지

로컬 디스크를 통한 스토리지는 수평적 확장성의 요구 사항을 충족하기에 충분하지 않습니다. 일부 분산 또는 공유 형태의 스토리지는 단일 스토리지 볼륨의 용량이 새로운 스토리지 하드웨어를 추가하여 원활하게 확장할 수 있는 추상화 계층을 제공하는 것이 필요합니다.

관리

분산 컴퓨팅에서 가장 중요한 임무는 관리 계층입니다. 이러한 관리 계층은 모든 소프트웨어 및 하드웨어 구성요소를 조정하고 통신, 스토리지, 공유 리소스의 사용을 효율적으로 관리합니다.

다음 부분에서는 각각의 계층에 있는 기술에 대해 보다 상세하게 설명합니다.

1.3.1. 통신

통신 계층은 데이터 전송을 확인하며 다음과 같은 두 부분으로 구성되어 있습니다:

- 하드웨어
- 소프트웨어

여러 시스템이 통신하는 가장 간단한 (그리고 가장 신속한) 방법은 공유 메모리를 사용하는 것입니다. 이는 일반적인 메모리 읽기/쓰기 작업 사용을 필요로 합니다. 공유 메모리는 높은 대역폭, 짧은 대기 시간, 일반적인 메모리 읽기/쓰기 작업의 낮은 오버 헤드를 갖습니다.

이더넷

컴퓨터 간에 가장 일반적인 통신 방법은 이더넷을 사용하는 것입니다. 현재 시스템에서 GbE (Gigabit Ethernet)가 기본적으로 제공되며 대부분의 서버에는 기가바이트 이더넷 포트가 2-4개 있습니다. GbE는 우수한 대역폭 및 대기 시간을 제공합니다. 이는 현재 사용되는 대부분의 분산 시스템의 기초가 되고 있습니다. 시스템에 고속 네트워크 하드웨어가 있어도 전용 관리 인터페이스는 일반적으로 GbE를 사용합니다.

10GbE

10GbE (10 Gigabit Ethernet)은 하이 엔드 및 미드 레인지 서버에서 급속히 확대하고 있습니다. 10GbE는 GbE의 10 배의 대역폭을 제공합니다. 주요 장점 중 하나는 최신 멀티 코어 프로세서와 함께 사용하면 통신과 컴퓨팅 간의 균형을 회복하는 것입니다. 단일 코어 시스템에서 GbE를 사용하는 경우와 8 코어 시스템

템에서 10GbE를 사용하는 것을 비교해 보면 차이를 잘 알 수 있습니다. 이 방법의 사용에서 전체적인 시스템 성능을 유지하고 통신 병목 현상을 해소하기 위해 10GbE가 특히 유용합니다.

불행히도 10GbE은 비용이 높습니다. 10GbE NIC 비용은 하락한 반면 상호 연결 (특히 광섬유)의 가격은 높은 상태로 남아있고 10GbE 네트워크 스위치 비용은 매우 높습니다. 장기적으로 이러한 가격은 하락할 것이라 예상하지만 현재 10GbE는 서버룸 백본 및 성능 크리티컬 애플리케이션에서 가장 많이 사용되고 있습니다.

Infiniband

Infiniband는 10GbE 보다 더 높은 성능을 제공합니다. 이더넷과 함께 사용되는 TCP/IP 및 UDP 네트워크 연결 이외에 Infiniband는 공유 메모리 통신을 지원합니다. 이는 Infiniband가 RDMA (remote direct memory access)를 통해 시스템 간의 작업을 가능하게 합니다.

Infiniband는 RDMA를 사용하여 TCP/IP 오버헤드 또는 소켓 연결없이 데이터를 시스템간에 직접 이동할 수 있습니다. 결과적으로 이는 일부 애플리케이션에서 중요한 대기 시간을 감소시킬 수 있습니다.

Infiniband는 높은 대역폭과 낮은 대기 시간, 낮은 오버헤드를 필요로 하는 HPTC (High Performance Technical Computing) 애플리케이션에서 가장 많이 사용되고 있습니다. 많은 슈퍼 컴퓨팅 애플리케이션이 이에서 혜택을 받고 더 빠른 프로세서 또는 더 많은 메모리가 아닌 Infiniband에 투자하여 성능 개선을 하는 것이 가장 좋은 방법으로 지적되고 있습니다.

RoCCE

RoCCE (RDMA over Ethernet)는 10GbE 인프라를 통해 Infiniband 스타일 통신 (RDMA 포함)을 구현합니다. 10GbE 제품의 볼륨 확대와 관련하여 비용 향상을 감안할 때 다양한 시스템 및 애플리케이션에서 RDMA 및 RoCCE 사용 증가가 예상됩니다.

Red Hat은 각각의 이러한 통신 방법과 Red Hat Enterprise Linux 6와의 사용을 완전하게 지원합니다.

1.3.2. 스토리지

분산된 컴퓨팅을 사용하는 환경에서는 공유 스토리지의 여러 인스턴스가 사용됩니다. 이는 다음 중 하나를 의미합니다:

- 단일 위치에서 여러 시스템이 데이터를 저장하고 있는 경우
- 여러 스토리지 어플라이언스로 구성된 스토리지 장치 (예: 볼륨)

가장 보편화된 스토리지의 예에는 시스템에 마운트된 로컬 디스크 드라이브입니다. 모든 애플리케이션이 하나의 호스트 또는 소수의 호스트에서 호스트되고 있는 경우 이는 IT 운영에 적절합니다. 하지만 인프라가 수십 또는 수백개의 시스템으로 확장되면 여러 로컬 스토리지 디스크로 관리하는 것은 어렵고 복잡해 집니다.

분산 스토리지는 비즈니스가 확대됨에 따라 스토리지 하드웨어 관리를 쉽고 자동화하기 위해 계층을 추가합니다. 여러 시스템이 유용한 스토리지 인스턴스를 공유하게 하여 관리자가 관리해야 하는 장치 수를 줄일 수 있습니다.

여러 스토리지 어플라이언스의 스토리지 기능을 하나의 볼륨으로 통합하여 사용자와 관리자 모두에게 도움이 되게 합니다. 이러한 유형의 분산 스토리지는 스토리지 풀에 추상적 계층을 제공합니다. 사용자는 스토리지의 단일 단위를 볼 수 있는 반면 관리자는 더 많은 하드웨어를 추가하여 이를 쉽게 확장할 수 있습니다. 분산 스토리지를 가능하게 하는 일부 기술은 파일 오버 및 멀티패스와 같은 추가된 혜택도 제공합니다.

NFS

NFS (Network File System)는 여러 대의 서버나 사용자가 **TCP** 또는 **UDP**를 통해 원격 스토리지의 동일한 인스턴스를 마운트 및 사용하는 것을 허용합니다. 일반적으로 **NFS**는 여러 애플리케이션에 의해 공유되는 데이터를 보유하고 있습니다. 또한 대량 데이터의 대량 저장에 적합합니다.

SAN

SAN (Storage Area Networks)는 파이버 채널이나 **iSCSI** 프로토콜을 사용하여 스토리지에 원격 액세스를 제공합니다. 파이버 채널 인프라 (파이버 채널 호스트 버스 어댑터, 스위치, 스토리지 어레이 등)는 고성능, 높은 대역폭, 대량 스토리지를 결합합니다. **SAN**은 프로세스에서 스토리지를 분리함으로써 시스템 디자인의 유연성이 매우 높아집니다.

SAN의 또 다른 장점은 주요 스토리지 하드웨어 관리 작업을 수행하기 위해 관리 환경을 제공한다는 것입니다. 이러한 작업에는 다음이 포함됩니다:

- 스토리지로의 액세스 제어
- 대량 데이터 관리
- 시스템 구축
- 데이터 백업 및 복제
- 스냅샷 만들기
- 시스템 장애 복구 지원
- 데이터 무결성 보장
- 데이터 마이그레이션

GFS2

Red Hat GFS2 (Global File System 2) 파일 시스템은 몇몇 특별한 기능을 제공합니다. **GFS2**의 기본 기능은 동시에 읽기/쓰기 액세스, 여러 클러스터 멤버 간 공유를 포함하여 단일 파일 시스템을 제공합니다. 즉 각 클러스터 멤버는 **GFS2** 파일 시스템의 "디스크 상"에서 완전히 동일한 데이터를 볼 수 있습니다.

GFS2는 모든 시스템이 동시에 "디스크"에 액세스하는 것을 허용합니다. 데이터 무결성을 유지하려면 **GFS2**는 **DLM (Distributed Lock Manager)**을 사용하여 하나의 시스템이 한 번에 특정 위치에 쓸 수 있게 합니다.

GFS2는 스토리지에서 고가용성을 필요로 하는 폐일오버 애플리케이션에 아주 적합합니다.

GFS2에 대한 보다 자세한 내용은 **GFS (Global File System) 2**에서 참조하십시오. 일반적인 스토리지에 대한 보다 자세한 내용은 **스토리지 관리 가이드**에서 참조하십시오. 이 두가지 모두는 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있습니다.

1.3.3. 통합 네트워크

네트워크를 통한 통신은 일반적으로 이더넷을 통해 이루어지며 스토리지 트래픽과 함께 전용 파이버 채널 **SAN** 환경을 사용합니다. 일반적으로 시스템 관리를 위해 전용 네트워크 또는 직렬 링크를 갖으며 **heartbeat^[2]**가 있을 수도 있습니다. 결과적으로 단일 서버는 일반적으로 여러 네트워크에 있게 됩니다.

각 서버에서 여러 연결을 제공하면 비용이 비싸지고, 거대해지며 관리가 복잡하게 됩니다. 따라서 모든 연결을 하나로 통합하는 방법이 필요하게 됩니다. **FCoE (Fibre Channel over Ethernet)** 및 **iSCSI (Internet SCSI)**가 이러한 요구에 부응합니다.

FCoE

FCoE로 표준 파이버 채널 명령 및 데이터 패킷은 단일 CNA (converged network card)를 통해 10GbE 물리적 인프라로 이동합니다. 표준 TCP/IP 이더넷 트래픽과 파이버 채널 스토리지 작업도 동일한 링크를 통해 이동할 수 있습니다. FCoE는 여러 논리 네트워크/스토리지 연결을 위해 하나의 물리적 네트워크 인터페이스 카드 (하나의 케이블)를 사용합니다.

FCoE는 다음과 같은 장점을 제공합니다:

연결 수 감소

FCoE를 사용하면 서버로의 네트워크 연결 수를 절반으로 줄일 수 있습니다. 성능이나 가용성을 목적으로 여러 연결을 선택할 수 있지만 단일 연결은 스토리지와 네트워크 연결을 모두 제공합니다. 이는 특히 피자 박스 서버와 블레이드 서버의 경우 구성 요소의 공간이 한정되어 있기 때문에 유용할 것입니다.

낮은 비용

연결 수를 줄이면 바로 케이블, 스위치 및 기타 네트워크 장비 수가 줄어들게 됩니다. 또한 이더넷의 기록에 있어서 경제적 규모를 특징으로 합니다. 시장에서의 디바이스 수가 수백만에서 수십억으로 늘어나면 100Mb 이더넷 및 기가 바이트 이더넷 장치의 가격 하락에서 볼 수 있듯이 네트워크 비용은 현저하게 감소합니다.

이와 유사하게 10GbE 사용을 채택하는 기업이 늘어남에 따라 이 가격은 더 저렴해 지게 됩니다. 또한 CNA 하드웨어가 단일 칩으로 통합되고 사용 확대는 시장에서 볼륨을 증가시키고 결과적으로 시간이 지남에 따라 가격이 대폭 하락하게 됩니다.

iSCSI

iSCSI (Internet SCSI)는 다른 유형의 통합된 네트워크 프로토콜로 FCoE의 대안입니다. 파이버 채널과 같이 iSCSI는 네트워크를 통해 블록 레벨 스토리지를 제공합니다. 하지만 iSCSI는 전체 관리 환경을 제공하지 않습니다. FCoE를 통한 iSCSI의 주요 장점은 iSCSI는 파이버 채널의 대부분의 기능과 유연성을 더 낮은 비용으로 제공할 수 있다는 점입니다.

[1] Red Hat 인증 엔지니어. 자세한 내용은 <http://www.redhat.com/training/certifications/rhce/>에서 참조하십시오.

[2] *Heartbeat*는 시스템 간에 메시지를 교환하여 각 시스템이 작동하고 있는지를 확인합니다. 시스템이 "heartbeat 없음"일 경우 시스템이 실패하여 종료되었다고 간주하고 다른 시스템이 이를 인수합니다.

2장. RED HAT ENTERPRISE LINUX 6 성능 특징

2.1. 64 비트 지원

Red Hat Enterprise Linux 6는 64 비트 프로세서를 지원합니다. 이 프로세서는 이론적으로 최대 16 엑사 바이트 메모리를 사용할 수 있습니다. GA (general availability)에서 Red Hat Enterprise Linux 6는 최대 8TB의 물리적 메모리 지원을 테스트 및 검증하고 있습니다.

Red Hat은 더 큰 메모리 블록을 사용 가능하게 하는 여러 기능을 지속적으로 도입 및 개선하고 있기 때문에 Red Hat Enterprise Linux 6가 지원하는 메모리 크기는 여러 마이너 업데이트를 통해 증대할 예정입니다. 개선 사항에 대한 예 (Red Hat Enterprise Linux 6 GA 시점에서)는 다음과 같습니다:

- Huge pages 및 transparent huge pages
- NUMA (Non-Uniform Memory Access) 개선

이러한 개선 사항은 다음 부분에서 자세히 설명합니다.

Huge pages 및 transparent huge pages

Red Hat Enterprise Linux 6에서 *huge pages*의 구현으로 다른 메모리 작업 동안 메모리 사용을 효과적으로 관리할 수 있게 되었습니다. Huge pages는 표준 4 KB 페이지 크기와 비교하여 동적으로 2 MB 페이지를 이용하므로 애플리케이션은 기가바이트나 테라바이트 메모리 처리에서 충분히 확장 가능합니다.

Huge pages는 수동으로 생성, 관리, 사용하기 어렵습니다. 이러한 문제에 대처하기 위해 Red Hat Enterprise 6에서는 *THP* (transparent huge pages)를 사용하는 기능이 있습니다. THP는 huge pages 사용에 관한 여러 복잡한 사항을 자동으로 관리합니다.

huge pages 및 THP에 대한 보다 자세한 내용은 5.2절. “Huge Pages 및 Transparent Huge Pages”에서 참조하십시오.

NUMA 개선

현재 새로운 시스템의 대부분은 *NUMA* (Non-Uniform Memory Access)를 지원하고 있습니다. NUMA는 대규모 시스템의 하드웨어 설계 및 생성을 간소화합니다. 하지만 이는 애플리케이션 개발에 다른 복잡성을 추가합니다. 예를 들어 NUMA는 로컬 및 원격 메모리를 구현하고 여기서 원격 메모리는 로컬 메모리 보다 몇 배의 액세스 시간이 소요됩니다. 이 기능은 (다른 기능과 함께) 배포되는 운영 체제, 애플리케이션, 시스템 설정 등 많은 성능에 영향을 미칠 수 있습니다.

Red Hat Enterprise Linux 6는 NUMA 시스템에서 애플리케이션 및 사용자 관리에 유용한 몇 가지 추가 기능에 의해 NUMA 사용에 대해 최적화되어 있습니다. 이러한 기능에는 CPU 친화도, CPU 핀 설정 (cpusets), numactl, 제어 그룹 등이 포함되며, 이는 프로세스 (affinity) 또는 애플리케이션 (핀 설정)을 특정 CPU나 CPU 세트에 "바인딩"할 수 있게 합니다.

Red Hat Enterprise Linux 6에서 NUMA 지원에 대한 자세한 내용은 4.1.1절. “CPU 및 NUMA 토폴로지”에서 참조하십시오.

2.2. TICKET SPINLOCKS

시스템 설계의 요점은 프로세스가 다른 프로세스에 의해 사용되는 메모리를 변경하지 않도록 확인하는 것입니다. 메모리에서 제어되지 않는 데이터 변경은 데이터 손상이나 시스템 충돌을 초래할 수 있습니다. 이를 방지하기 위해 운영 체제는 프로세스가 메모리의 일부를 잠그고 작업을 실행한 후 메모리를 잠금 해제하거나 "해제"할 수 있게 합니다.

메모리 잠금의 일반적인 구현은 *spin locks*을 통한 것으로 이는 프로세스가 잠금을 사용할 수 있는지에 대한 여부를 지속적으로 확인하고 잠금을 사용할 수 있을 경우 즉시 잠금을 실행할 수 있게 합니다. 동일한

잠금에 대해 경합하는 여러 프로세스가 있을 경우 잠금 해제된 후 잠금을 요청한 첫 번째 프로세스가 이를 획득하게 됩니다. 모든 프로세스가 메모리에 동일한 액세스를 가질 경우 이 방법은 "공정 (fair)"에서 잘 작동하게 됩니다.

불행히도 NUMA 시스템에서 모든 프로세스가 잠금에 동일한 액세스를 갖지 않습니다. 잠금과 같은 동일한 NUMA 노드에 있는 프로세스는 잠금을 취득하는데 불공정한 장점이 있습니다. 원격 NUMA 모드에서의 프로세스는 잠금 부족 상태에 빠져 성능이 저하됩니다.

이 문제를 해결하기 위해 Red Hat Enterprise Linux는 *ticket spinlocks*를 구현했습니다. 이 기능은 잠금에 예약 대기열 메커니즘을 추가하고 모든 프로세스가 요청한 순서로 잠금을 취득할 수 있게 합니다. 이는 잠금 요청에 있어서의 타이밍 문제와 불공정한 이점을 제거합니다.

ticket spinlock 오버헤드는 일반적인 *spinlock* 보다 다소 크지만 확장성이 뛰어나 NUMA 시스템에서 보다 나은 성능을 제공합니다.

2.3. 동적 목록 구조

운영 체제는 시스템의 각 프로세서에서 정보 세트를 필요로 합니다. Red Hat Enterprise Linux 5에서 이러한 정보 세트는 메모리에 있는 고정 크기 배열에 할당되었습니다. 각각의 개별적 프로세서에 있는 정보는 이 배열로 인덱스화하여 검색되었습니다. 이는 비교적 적은 프로세서가 들어있는 시스템의 경우 빠르고 쉽고 간단한 방법이었습니다.

하지만 시스템의 프로세서 수가 증가함에 따라 이 방법은 상당한 오버 헤드를 산출합니다. 메모리의 고정 크기 배열은 하나의 공유 리소스이기 때문에 많은 프로세서가 동시에 액세스하려고 하면 병목 현상이 발생할 수 있습니다.

이 문제를 해결하기 위해 Red Hat Enterprise Linux 6는 프로세서 정보의 동적 목록 구조를 사용합니다. 이는 프로세서 정보에 사용된 배열을 동적으로 할당할 수 있게 합니다. 시스템에 8 개의 프로세서만 있는 경우 8 개의 항목만 목록에 생성됩니다. 2048 개의 프로세서가 있는 경우 2048 개의 항목만이 생성됩니다.

동적 목록 구조로 보다 더 세밀한 잠금이 가능합니다. 예를 들어, 프로세서 6, 72, 183, 657, 931, 1546에서 동시에 정보 업데이트가 필요한 경우 더 많은 병렬로 실행할 수 있습니다. 이러한 상황은 소형 시스템에서 보다 대형의 고성능 시스템에서 더 많이 발생합니다.

2.4. 틱리스 커널

Red Hat Enterprise Linux의 이전 버전에서 커널은 지속적으로 시스템 인터럽트를 생성하는 타이머 기반 메커니즘을 사용했습니다. 각 인터럽트 동안 시스템은 폴링되었습니다. 즉, 이는 실행해야 할 작업이 있는지를 확인했습니다.

설정에 따라 시스템 인터럽트 또는 *타이머 틱*은 초당 수백 번 또는 수천 번 발생할 수 있습니다. 이는 시스템의 작업 부하에 관계없이 초당 발생하는 것입니다. 로드가 적은 시스템에서 프로세스는 효과적으로 슬립 상태를 사용할 수 없으며 이는 *전력 소비*에 영향을 미칩니다. 시스템이 슬립 상태에 있을 경우 시스템은 전력 소비를 최소화할 수 있습니다.

시스템을 가장 전력 효율적인 방법으로 실행하려면 가능한 작업을 신속하게 완료하고 가능한 슬립 상태에 길게 들어가도록 하는 것입니다. 이를 구현하기 위해 Red Hat Enterprise Linux 6는 *틱리스 커널 (tickless kernel)*을 사용합니다. 이를 사용하면 인터럽트 타이머가 유휴 루프에서 제거되어 Red Hat Enterprise Linux 6를 완전히 인터럽트 구동 환경으로 변환하게 됩니다.

틱리스 커널은 시스템이 유휴 시간 동안 깊은 슬립 상태에 들어갈 수 있게 하여 완료해야 할 작업이 있을 경우 즉시 반응합니다.

보다 자세한 내용은 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *전력 관리 가이드*에서 참조하십시오.

2.5. 컨트롤 그룹

Red Hat Enterprise Linux는 성능 튜닝을 위해 여러 유용한 옵션을 제공합니다. 수 백개의 프로세서까지 확장하는 대형 시스템을 튜닝하여 뛰어난 성능을 제공할 수 있습니다. 하지만 이러한 시스템의 튜닝에는 상당한 전문 지식과 명확하게 정의된 워크로드가 필요합니다. 비용이 비싸고 소수인 대형 시스템의 경우 이를 특별히 다루어도 상관없었습니다. 현재 이러한 시스템은 메인 스트림이 되어 보다 효과적인 도구가 필요합니다.

복잡성에 더하여 서비스 통합을 위해 현재 보다 강력한 시스템이 사용되고 있습니다. 이전의 4에서 8 대의 서버에서 실행되고 있던 작업은 현재 하나의 서버로 대체되었습니다. 1.2.1절. “병렬 컴퓨팅”에서 설명했듯이 여러 중급 시스템에는 현재 예전의 고성능 시스템보다 더 많은 코어가 들어 있습니다.

현재 많은 애플리케이션은 병렬 처리 용으로 설계되어 성능 향상을 위해 여러 스레드 또는 프로세스를 사용합니다. 하지만 일부 애플리케이션은 8 개 이상의 스레드를 보다 효과적으로 사용할 수 있습니다. 따라서 일반적으로 여러 애플리케이션은 기능을 극대화하기 위해 32-CPU 시스템에 설치해야 합니다.

다음과 같은 상황을 생각해 봅시다: 소형의 저렴한 메인 스트림 시스템이 현재 이전의 비싼 고성능 시스템의 성능을 갖는 패리티입니다. 저렴한 고성능 시스템은 시스템 설계자에게 더 적은 수의 시스템에 더 많은 서비스를 통합할 수 있는 기능을 부여합니다.

하지만 일부 리소스 (I/O 및 네트워크 통신 등)는 공유되고 CPU 수 만큼 빠르게 확장하지 않습니다. 따라서 여러 애플리케이션을 포함하는 시스템은 하나의 애플리케이션이 단일 리소스를 독점하면 전체 성능이 저하될 수 있습니다.

이 문제를 해결하기 위해 현재 Red Hat Enterprise Linux 6는 *cgroups* (control groups)를 지원합니다. Cgroups를 통해 관리자는 필요에 따라 리소스를 특정 작업에 할당할 수 있습니다. 예를 들어 4 개의 CPU 중 80%, 60GB 메모리, 디스크 I/O의 40%를 데이터베이스 애플리케이션에 할당할 수 있습니다. 동일한 시스템에서 실행 중인 웹 애플리케이션에는 두 개의 CPU, 2GB 메모리, 사용 가능한 네트워크 대역폭의 50%를 할당할 수 있습니다.

결과적으로 데이터베이스와 웹 애플리케이션 모두는 시스템 리소스를 과도하게 사용하지 않고 우수한 성능을 제공할 수 있습니다. 또한 *cgroups*의 여러 측면인 *자체 튜닝* (*self-tuning*)으로 인해 시스템은 워크로드 변경에 따라 대응할 수 있습니다.

cgroup에는 다음과 같은 두 가지 주요 구성 요소가 있습니다:

- cgroup에 할당된 작업 목록
- 이러한 작업에 할당된 리소스

cgroup에 할당된 작업은 cgroup 내에서 실행됩니다. 해당 작업의 자식 작업도 cgroup 내에서 실행됩니다. 이를 통해 관리자는 전체 애플리케이션을 단일 단위로 관리할 수 있습니다. 또한 관리자는 다음과 같은 리소스의 할당을 설정할 수 있습니다:

- CPUsets
- 메모리
- I/O
- 네트워크 (대역폭)

CPUsets 내에서 cgroups를 통해 관리자는 CPU 수, 특정 CPU 또는 노드의 친화도 [3] 및 작업 세트에서 사용하는 CPU 시간을 설정할 수 있습니다. CPUsets를 설정하기 위해 cgroups를 사용하는 것은 전반적인 성능 향상을 보장하고 애플리케이션이 CPU 시간 동안 작동되지 않는지를 확인하는 동시에 다른 작업의 비용에서 애플리케이션이 리소스를 과도하게 사용하지 않도록 하기 위해 필수적입니다.

I/O 대역폭과 네트워크 대역폭은 다른 리소스 컨트롤러에 의해 관리됩니다. 리소스 컨트롤러를 통해 **cgroup**의 작업이 소비하는 대역폭 양을 결정할 수 있으며 **cgroup**의 작업이 리소스를 과잉 소비하거나 리소스 부족이 되지 않게 할 수 있습니다.

Cgroups를 통해 관리자는 높은 수준에서 다양한 애플리케이션을 필요로 하고 소비하는 시스템 리소스를 정의 및 분배할 수 있습니다. 그 후 시스템은 자동으로 이러한 다양한 애플리케이션을 관리하고 균형을 유지하며 예측 가능한 우수한 성능을 제공하고 전체 시스템 성능을 최적화합니다.

컨트롤 그룹 사용 방법에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *리소스 관리 가이드*에서 참조하십시오.

2.6. 스토리지 및 파일 시스템 개선 사항

Red Hat Enterprise Linux 6는 스토리지 및 파일 시스템 관리를 위해 여러 개선된 기능을 갖추고 있습니다. 이 버전에서 가장 주목할 만한 두 가지 기능은 **ext4** 및 **XFS** 지원입니다. 스토리지 및 파일 시스템 관련 성능 개선에 대한 전체적인 설명은 **7장. 파일 시스템**에서 참조하십시오.

Ext4

Ext4는 Red Hat Enterprise Linux 6의 기본 파일 시스템입니다. 이는 **EXT** 파일 시스템 제품군의 4세대로 이론적으로 최대 1 엑사바이트 파일 시스템과 16TB 단일 파일을 지원합니다. Red Hat Enterprise Linux 6는 최대 16TB의 파일 시스템과 16TB의 단일 파일을 지원합니다. 더 큰 스토리지 기능 이외에 **ext4**에는 다음과 같은 다른 여러 새로운 기능이 포함되어 있습니다.

- 익스텐트 기반의 메타데이터
- 지연 할당
- 저널 체크섬 (Journal check-summing)

ext4 파일 시스템에 대한 보다 자세한 내용은 **7.3.1절. “Ext4 파일 시스템 ”**에서 참조하십시오.

XFS

XFS는 강력하고 안정된 64 비트 저널링 파일 시스템으로 단일 호스트에서 매우 큰 파일 및 파일 시스템을 지원합니다. 이러한 파일 시스템은 **SGI**에 의해 개발된 것으로 매우 큰 서버 및 스토리지 어레이에서 장기간 실행된 기록이 있습니다. XFS 기능은 다음과 같습니다:

- 지연 할당
- 동적으로 할당된 inode
- 여유 공간 관리의 확장성을 위한 B 트리 인덱싱
- 온라인 조각 모음 및 파일 시스템 확장
- 정교한 메타데이터 미리 읽기 알고리즘

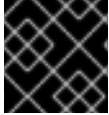
XFS는 엑사바이트로 확대된 반면 Red Hat이 지원하는 최대 XFS 파일 시스템 크기는 100TB입니다. XFS에 대한 보다 자세한 내용은 **7.3.2절. “XFS 파일 시스템 ”**에서 참조하십시오.

대형 부트 드라이브

전형적인 BIOS는 최대 2.2TB의 디스크 크기를 지원합니다. BIOS를 사용하는 Red Hat Enterprise Linux 6 시스템은 **GPT** (Global Partition Table)라는 새로운 디스크 구조를 사용하여 2.2TB 이상의 디스크 크기를 지원할 수 있습니다. GPT는 데이터 디스크 용으로만 사용할 수 있으며 BIOS와 함께 부팅 드라이브 용으

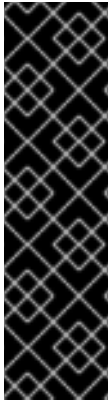
로는 사용할 수 없으므로 부팅 드라이브의 최대 크기는 2.2TB만 될 수 있습니다. 원래 BIOS는 IBM PC 용으로 생성되었습니다. BIOS가 최신 하드웨어에 적응할 수 있도록 진화한 반면 **UEFI** (Unified Extensible Firmware Interface)는 새로운 최신 하드웨어를 지원하도록 설계되어 있습니다.

Red Hat Enterprise Linux 6는 UEFI를 지원합니다. 이는 BIOS (아직 지원됨)를 대체하기 위해 사용될 수 있습니다. Red Hat Enterprise Linux 6를 실행하고 있는 UEFI를 갖춘 시스템에서는 부팅 파티션 및 데이터 파티션 모두의 경우 GPT 및 2.2TB (또는 그 이상) 파티션을 사용할 수 있습니다.



중요

Red Hat Enterprise Linux 6는 32 비트 x86 시스템 용 UEFI를 지원하지 않습니다.



중요

UEFI와 BIOS를 위한 부트 설정은 서로 매우 다르다는 것을 명심하십시오. 따라서, 설치된 시스템은 설치 중 사용했던 부팅방법에서 사용한 펌웨어와 동일한 방법으로 부팅해야만 합니다. BIOS를 사용하는 시스템에 운영 체제를 설치한 다음에, 그 설치를 이용해 UEFI를 사용하는 시스템에서 부팅할 수 없습니다.

Red Hat Enterprise Linux 6는 UEFI 사양 버전 2.2를 지원합니다. UEFI 사양 버전 2.3 이상을 지원하는 하드웨어는 Red Hat Enterprise Linux 6와 시작 및 실행해야 하지만 이러한 최신 사양에 의해 정의된 추가 기능을 사용할 수 없게 됩니다. UEFI 사양은 <http://www.uefi.org/specs/agreement/>에서 살펴볼 수 있습니다.

[3] 일반적으로 노드는 소켓 내에 있는 CPU 또는 코어 세트로 정의됩니다.

3장. 시스템 성능 모니터링 및 분석

다음 부분에서는 시스템 및 애플리케이션 성능을 모니터링하고 분석하는데 사용할 수 있는 도구에 대해 간단하게 소개하고 각 도구를 가장 효율적으로 사용할 수 있는 상황을 제시합니다. 각 도구에 의해 수집되는 데이터는 최적의 성능 이하의 원인이 되는 병목 현상 및 기타 시스템 문제를 밝혀낼 수 있습니다.

3.1. PROC 파일 시스템

proc "파일 시스템"은 Linux 커널의 현재 상태를 나타내는 파일의 계층 구조가 들어 있는 디렉토리입니다. 이를 통해 애플리케이션 및 사용자는 시스템의 커널 보기를 할 수 있습니다.

proc 디렉토리에는 시스템의 하드웨어 및 현재 실행 중인 프로세스에 대한 정보가 들어 있습니다. 이 파일의 대부분은 읽기 전용이지만 일부 파일 (주로 **/proc/sys**에 있는 파일)은 커널에 설정 변경을 전달하기 위해 사용자 및 애플리케이션에 의해 조작될 수 있습니다.

proc 디렉토리에 있는 파일을 확인 및 편집에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *운용 가이드*에서 참조하십시오.

3.2. GNOME 및 KDE 시스템 모니터

GNOME 및 KDE 데스크탑 환경 모두에는 시스템 동작을 모니터링하고 수정하는데 도움이 되는 그래픽 도구가 있습니다.

GNOME 시스템 모니터

GNOME 시스템 모니터는 기본 시스템 정보를 표시하고 시스템 프로세스, 리소스, 파일 시스템 사용량을 모니터링할 수 있습니다. **Terminal**에서 **gnome-system-monitor** 명령으로 이를 오픈하거나 **응용 프로그램** 메뉴를 클릭하여 **시스템 도구 > 시스템 모니터**를 선택합니다.

GNOME 시스템 모니터에는 다음과 같은 네 개의 탭이 있습니다:

시스템

컴퓨터의 하드웨어 및 소프트웨어에 대한 기본 정보를 표시합니다.

프로세스

활성 프로세스 및 해당 프로세스 간의 관계 뿐 만 아니라 각 프로세스에 대한 자세한 정보를 보여줍니다. 또한 표시된 프로세스를 필터링하고 이러한 프로세스에 특정 작업 (시작, 중지, 중단, 우선 순위 변경 등)을 수행할 수 있게 합니다.

리소스

현재 CPU 시간 사용량, 메모리, 스왑 공간 사용량, 네트워크 사용량을 표시합니다.

파일 시스템

파일 시스템 유형, 마운트 지점, 메모리 사용량과 같은 각각에 대한 몇 가지 기본적인 정보와 함께 마운트된 모든 파일 시스템을 나열합니다.

GNOME 시스템 모니터에 대한 보다 자세한 내용은 애플리케이션에 있는 **도움말** 메뉴나

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있는 *운용 가이드*에서 참조하십시오.

KDE 시스템 가드

KDE 시스템 가드를 통해 현재 실행되고 있는 프로세스 및 현재 시스템 로드를 모니터링할 수 있습니다. 터미널에서 **ksysguard** 명령으로 이를 오픈하거나 **키크오프 애플리케이션 실행**을 클릭한 후 **응용 프로그램 > 시스템 > 시스템 모니터**를 선택합니다.

KDE 시스템 가드에는 두 개의 탭이 있습니다:

프로세스 표

기본값으로 실행 중인 모든 프로세스의 목록을 알파벳 순으로 표시합니다. 또한 총 **CPU** 사용량, 물리적 또는 공유 메모리 사용량, 소유자, 우선 순위 등과 같은 다른 속성에 따라 프로세스를 정렬할 수 있습니다. 가시적 결과를 필터링하거나, 특정 프로세스를 검색 또는 프로세스에서 특정 작업을 수행할 수 있습니다.

시스템 로드

CPU 사용량, 메모리, 스왑 공간 사용량, 네트워크 사용량의 그래픽 기록을 표시합니다. 자세한 분석 및 그래프 키를 위해 그래프에 마우스 오버합니다.

KDE 시스템 가드에 대한 보다 자세한 내용은 응용 프로그램에 있는 **도움말** 메뉴에서 참조하십시오.

3.3. 내장된 명령행 모니터링 도구

그래픽 모니터링 도구 이외에 **Red Hat Enterprise Linux**는 명령행에서 시스템을 모니터링하는데 사용할 수 있는 몇 가지 도구를 제공합니다. 이러한 도구의 장점은 런레벨 5 이상에서 사용할 수 있다는 것입니다. 다음 부분에서는 각 도구에 대해 간략하게 설명하고 가장 적합하게 사용될 수 있는 용도에 대해 제안합니다.

top

top 도구는 실행 중인 시스템에서 프로세스의 동적인 실시간 뷰를 제공합니다. 이는 시스템 요약, **Linux** 커널에 의해 관리되는 작업 등과 같은 다양한 정보를 표시할 수 있습니다. 또한 프로세스를 조작할 수 있는 제한된 기능을 가지고 있습니다. 이러한 동작 및 표시되는 정보는 모두 설정 가능하고 설정 세부 사항은 다시 시작하기를 통해 영구적으로 만들 수 있습니다.

기본값으로 프로세스는 **CPU** 사용량의 비율로 정렬되어 표시되며 대부분의 리소스를 소비하는 프로세스를 쉽게 확인할 수 있습니다.

top 사용에 대한 보다 자세한 내용은 **man top** **man** 페이지에서 참조하십시오.

ps

ps 도구는 활성 프로세스의 선택 그룹의 스냅샷을 찍습니다. 기본값으로 이 그룹은 현재 사용자가 소유하고 있고 동일한 터미널과 관련된 프로세스로 제한됩니다.

이는 **top** 보다 더 자세한 프로세스 정보를 제공하지만 동적이지 않습니다.

ps 사용에 대한 보다 자세한 내용은 **man ps** **man** 페이지에서 참조하십시오.

vmstat

vmstat (Virtual Memory Statistics)는 시스템의 프로세스, 메모리, 페이징, I/O 차단, 인터럽트, **CPU** 동작에 대한 즉각적인 보고서를 출력합니다.

이는 **top**과 같이 동적이지 않지만 거의 실시간으로 시스템 동작을 관찰할 수 있는 샘플링 간격을 지정할 수 있습니다.

vmstat 사용에 대한 보다 자세한 내용은 **man vmstat** **man** 페이지에서 참조하십시오.

sar

sar (System Activity Reporter)는 지금까지 현재의 시스템 동작에 대한 정보를 수집 및 보고합니다. 기본값 출력은 하루의 시작에서 10분 간격으로 현재의 CPU 사용률을 포함합니다:

```
12:00:01 AM      CPU      %user      %nice      %system      %iowait      %steal
%idle
12:10:01 AM      all        0.10        0.00        0.15        2.96        0.00
96.79
12:20:01 AM      all        0.09        0.00        0.13        3.16        0.00
96.61
12:30:01 AM      all        0.09        0.00        0.14        2.11        0.00
97.66
...
```

이 도구는 **top** 또는 유사한 도구를 통해 시스템 동작에 대한 보고서를 주기적으로 작성하기에 유용한 대안입니다.

sar 사용에 대한 보다 자세한 내용은 **man sar** **man** 페이지에서 참조하십시오.

3.4. TUNED 및 KTUNE

Tuned는 다양한 시스템 구성 요소의 사용에 대한 데이터를 모니터링하고 수집하여 필요에 따라 동적으로 시스템 설정을 조정하기 위해 이러한 정보를 사용하는 데몬입니다. 이는 CPU 및 네트워크 사용 변경 사항에 반응하고 활성 장치에 있는 성능을 향상시키거나 비활성 장치의 전력 소비를 줄이기 위해 설정을 조정할 수 있습니다.

부수적인 **ktune**은 **tuned-adm** 도구와 함께 사전 설정된 튜닝 프로파일을 제공하여 수많은 특정 사용 사례에 있어서 성능을 향상시키고 전원 소비를 감소시킵니다. 이 프로파일을 편집하거나 새 프로파일을 생성하여 사용자 환경에 적합한 성능 솔루션을 생성합니다.

tuned-adm의 일부로 제공되는 프로파일에는 다음과 같은 것이 포함됩니다:

default

기본적인 절전 프로파일입니다. 이는 가장 기본적인 절전 프로파일입니다. 디스크 및 CPU 플러그인만 활성화합니다. 이는 **tuned-adm**을 튜닝 해제하는 것과 동일하지 않습니다. 이 경우 **tuned** 및 **ktune** 모두는 비활성화됩니다.

latency-performance

전형적인 지연 성능 튜닝 용 서버 프로파일입니다. 이는 **tuned** 및 **ktune** 절전 메커니즘을 비활성화합니다. **cpuspeed** 모드는 **performance**로 변경됩니다. 각 장치의 I/O 엘리베이터는 **deadline**으로 변경됩니다. 서비스의 전원 관리 품질의 경우 **cpu_dma_latency** 요구 사항 값 **0**이 등록됩니다.

throughput-performance

전형적인 처리량 성능 튜닝 용 서버 프로파일입니다. 시스템이 엔터프라이즈급 스토리지가 없는 경우 이 프로파일이 권장됩니다. 이는 **latency-performance**와 동일하지만 다음과 같은 차이점이 있습니다:

- **kernel.sched_min_granularity_ns** (scheduler minimal preemption granularity)는 **10** 밀리초로 설정됩니다.
- **kernel.sched_wakeup_granularity_ns** (scheduler wake-up granularity)는 **15** 밀리초로 설정됩니다.

- **vm.dirty_ratio** (virtual machine dirty ratio)는 40%로 설정됩니다.
- transparent huge pages가 활성화됩니다.

enterprise-storage

이 프로파일은 배터리 백업 컨트롤러 캐시 보호 및 디스크 내장 캐시 관리 등 엔터프라이즈급 스토리지로 엔터프라이즈 크기 서버 설정에 권장됩니다. 이는 **throughput-performance** 프로파일과 동일하지만 파일 시스템은 **barrier=0**로 다시 마운트됩니다.

virtual-guest

이 프로파일은 배터리 백업 컨트롤러 캐시 보호 및 디스크 내장 캐시 관리 등 엔터프라이즈급 스토리지로 엔터프라이즈 크기 서버 설정에 권장됩니다. 이는 **throughput-performance** 프로파일과 동일하지만 다음과 같은 차이점이 있습니다:

- **readahead** 값은 **4x**로 설정됩니다.
- root/boot 파일 시스템 이외의 파일 시스템은 **barrier=0**으로 다시 마운트됩니다.

virtual-host

enterprise-storage 프로파일에 따라 **virtual-host**도 가상 메모리의 swappiness를 줄이고 더 티 페이지의 보다 적극적인 쓰기 저장을 가능하게 합니다. 이 프로파일은 Red Hat Enterprise Linux 6.3 이상 버전에서 사용 가능하며 KVM 및 Red Hat Enterprise Virtualization 호스트를 포함하여 가상화 호스트의 프로파일에 권장됩니다.

tuned 및 **ktune**에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있는 Red Hat Enterprise Linux 6 전력 관리 가이드에서 참조하십시오.

3.5. 애플리케이션 프로파일러

프로파일링은 이를 실행하면서 프로그램의 동작에 대한 정보를 수집하는 프로세스입니다. 메모리 사용을 줄이고 프로그램의 전반적인 속도를 높이기 위해 최적화할 수 있는 프로그램 영역을 결정하기 위한 애플리케이션을 프로파일링합니다. 애플리케이션 프로파일링 도구는 이러한 과정을 단순화하는데 유용합니다.

Red Hat Enterprise Linux 6와 함께 사용하기 위해 지원되는 세 가지 프로파일링 도구 **SystemTap**, **OProfile**, **Valgrind**가 있습니다. 이러한 프로파일링 도구를 문서화하는 것은 이 가이드의 범위를 벗어나는 것이지만 다음 부분에서는 각 프로파일러에 적합한 작업에 대한 간략한 개요 및 상세 정보가 있는 링크를 제공합니다.

3.5.1. SystemTap

SystemTap은 사용자가 운영 체제 활동 (특히 커널 활동)을 모니터링하고 분석할 수 있는 추적 및 측정 도구입니다. 이는 **netstat**, **top**, **ps**, **iostat**와 같은 도구의 출력과 유사한 정보를 제공합니다. 하지만 수집된 정보에 대해 보다 상세히 필터링하고 분석할 수 있는 옵션이 포함되어 있습니다.

SystemTap은 보다 깊이 있고 정확하게 시스템 활동 및 애플리케이션 동작을 분석하여 시스템 및 애플리케이션 병목 현상을 정확하게 지적할 수 있게 합니다.

Eclipse 용 Function Callgraph 플러그인은 백엔드로 **SystemTap**을 사용하여 함수 호출, 반환, 시간 및 사용자 공간 변수를 포함 프로그램 상태를 완전히 모니터링하고 쉽게 최적화하기 위해 시각적으로 정보를 표시합니다.

SystemTap에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *SystemTap* 초보자 가이드에서 참조하십시오.

3.5.2. OProfile

OProfile (**oprofile**)은 시스템 전역 성능 모니터링 도구입니다. 이는 프로세서의 전용 성능 모니터링 하드웨어를 사용하여 메모리 참조 시 L2 캐시 요청 수, 전송된 하드웨어 인터럽트 수와 같은 커널 및 시스템 실행에 대한 정보를 검색합니다. 또한 프로세서 사용량 및 가장 많이 사용되는 애플리케이션 및 서비스를 지정하는데 사용될 수 있습니다.

OProfile은 Eclipse OProfile 플러그인을 통해 Eclipse와 함께 사용될 수 있습니다. 이 플러그인은 사용자에게 자신의 코드에서 가장 시간이 많이 소요되는 영역을 쉽게 결정하게 하고 결과의 시각화로 OProfile의 모든 명령행 기능을 수행할 수 있게 합니다.

하지만 사용자는 여러 OProfile 제한 사항을 알고 있어야 합니다:

- 성능 모니터링 샘플이 정확하지 않을 수 있습니다 - 프로세서가 잘못된 지시 사항을 실행할 수 있기 때문에 샘플이 인터럽트를 발생시킨 지시 사항 대신 인근 지시 사항에서 기록될 수 있습니다.
- OProfile은 시스템 전역의 것으로 프로세스가 여러번 시작 및 중지될 수 있으며 여러 실행에서의 샘플이 축적 허용됩니다. 즉 이는 이전 실행에서 데이터 샘플을 삭제해야 함을 의미합니다.
- CPU 제한 프로세서의 문제를 확인하는 것에 초점을 두고 있으므로 다른 이벤트에 대해 잠금 상태에서 기다리는 동안 수면 상태에 있는 프로세스를 인식하지는 않습니다.

OProfile 사용에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *운용 가이드*에서나 `/usr/share/doc/oprofile-<version>`에 있는 시스템의 **oprofile** 문서에서 참조하십시오.

3.5.3. Valgrind

Valgrind는 애플리케이션의 성능과 정확성 개선에 도움이 될 검색 및 프로파일링 도구를 제공합니다. 이러한 도구는 힙, 스택, 배열 오버런 이외에 메모리 및 스레드 관련 오류를 감지할 수 있으므로 애플리케이션 코드에서 오류를 확인하고 쉽게 수정할 수 있습니다. 캐시, 힙, 분기 예측을 프로파일하여 애플리케이션 속도를 높이고 애플리케이션 메모리 사용을 최소화할 수 있는 요소를 식별할 수 있습니다.

Valgrind는 애플리케이션을 통합 CPU에서 실행하고 기존 애플리케이션 코드를 실행하는 동안 계측하여 애플리케이션을 분석합니다. 다음에 "코멘트"를 붙여 애플리케이션 실행에 관련된 각각의 프로세스를 특정 사용자 파일 설명자, 파일, 네트워크 소켓으로 명확하게 식별합니다. 계측의 수준은 사용하는 Valgrind 도구 및 설정에 따라 다르지만 계측된 코드의 실행은 일반적 실행 보다 4-50배의 시간이 걸릴 수 있다는 것을 염두해 두십시오.

Valgrind는 다시 컴파일하지 않고 그대로 애플리케이션에서 사용할 수 있습니다. 하지만 Valgrind는 코드의 문제를 식별하기 위해 디버그 정보를 사용하므로 애플리케이션 및 지원 라이브러리가 유효한 디버깅 정보로 컴파일되지 않은 경우 이러한 정보를 포함하도록 다시 컴파일할 것을 강력하게 권장합니다.

Red Hat Enterprise Linux 6.4에서 Valgrind는 gdb (GNU Project Debugger)로 통합되어 디버깅 효율성을 높입니다.

Valgrind에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *개발자 가이드*에서 참조하거나 valgrind 패키지가 설치되어 있을 경우 **man valgrind** 명령을 사용하여 참조할 수 있습니다. 기타 다른 문서는 다음에서 확인하실 수 있습니다:

- `/usr/share/doc/valgrind-<version>/valgrind_manual.pdf`

- `/usr/share/doc/valgrind-<version>/html/index.html`

시스템 메모리를 프로파일하기 위해 Valgrind 사용하는 방법에 대한 자세한 내용은 5.3절. “프로파일 메모리 사용에 Valgrind 사용”에서 참조하십시오.

3.5.4. Perf

perf 도구는 여러 유용한 성능 카운터를 제공하여 사용자가 시스템에서 다른 명령의 효과를 평가할 수 있게 합니다:

perf stat

이 명령은 실행된 지시 사항 및 소비된 클럭 사이클을 포함하여 일반적인 성능 이벤트의 전체 통계를 제공합니다. 옵션 플래그를 사용하여 기본 측정 이벤트 이외에 이벤트의 통계를 수집할 수 있습니다. Red Hat Enterprise Linux 6.4에서 **perf stat**를 사용하여 하나 이상의 지정된 컨트롤 그룹 (cgroups)에 기반하여 모니터링을 필터링할 수 있습니다. 보다 자세한 내용은 **man perf-stat** man 페이지에서 참조하십시오.

perf record

이 명령은 성능 데이터를 **perf report**를 사용하여 나중에 분석할 수 있는 파일에 기록합니다. 보다 자세한 내용은 **man perf-record** man 페이지에서 참조하십시오.

perf report

이 명령은 파일에서 성능 데이터를 읽고 기록된 데이터를 분석합니다. 보다 자세한 내용은 **man perf-report** man 페이지에서 참조하십시오.

perf list

이 명령은 특정 컴퓨터에서 사용할 수 있는 이벤트를 나열합니다. 이러한 이벤트는 성능 모니터링 하드웨어 및 시스템의 소프트웨어 설정에 따라 다릅니다. 보다 자세한 내용은 **man perf-list** man 페이지에서 참조하십시오.

perf top

이 명령은 **top** 도구와 유사한 기능을 수행합니다. 실시간으로 성능 카운트 프로파일을 생성 및 표시합니다. 보다 자세한 내용은 **man perf-top** man 페이지에서 참조하십시오.

perf에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있는 Red Hat Enterprise Linux 개발자 가이드에서 참조하십시오.

3.6. RED HAT ENTERPRISE MRG

Red Hat Enterprise MRG의 실시간 구성 요소에는 **Tuna**가 포함되어 있습니다. 이는 사용자가 시스템을 튜닝할 수 있는 값을 조정하고 이러한 변경 사항의 결과를 확인할 수 있는 도구입니다. 이는 실시간 구성 요소를 사용하기 위해 개발되었지만 표준 Red Hat Enterprise Linux 시스템을 튜닝하는데 사용할 수 있습니다.

Tuna로 다음과 같은 불필요한 시스템 동작을 조정 또는 비활성화할 수 있습니다:

- 전원 관리, 오류 감지, 시스템 관리 인터럽트와 관련된 BIOS 매개 변수;
- 인터럽트 합병, TCP 사용과 같은 네트워크 설정;

- 파일 시스템 저널링에서 동작 저널링;
- 시스템 로깅;
- 특정 CPU 또는 CPU 범위에 의해 인터럽트와 사용자 프로세스가 처리되는지에 대한 여부;
- 스왑 공간이 사용되는지에 대한 여부;
- 메모리 부족 예외를 처리하는 방법

Tuna 인터페이스로 Red Hat Enterprise MRG 튜닝에 대한 보다 자세한 개념적인 내용은 *실시간 튜닝 가이드*의 "일반적인 시스템 튜닝" 장에서 참조하십시오. Tuna 인터페이스 사용에 대한 자세한 내용은 *Tuna 사용자 가이드*에서 참조하십시오. 두 가이드 모두는 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_MRG/에서 확인하실 수 있습니다.

4장. CPU

CPU 즉 *central processing unit*은 대부분의 시스템에 대해 잘못된 명칭입니다. 왜냐하면 *중앙화* (*central*)는 *하나* (*single*)를 의미하지만 대부분의 현대적인 시스템에는 하나 이상의 처리 장치 또는 코어가 있기 때문입니다. 물리적으로 CPU는 소켓에 있는 마더보드에 설치된 패키지에 들어 있습니다. 마더보드에 있는 각 소켓에는 다른 CPU 소켓, 메모리 컨트롤러, 인터럽트 컨트롤러, 기타 주변 장치와 같은 다양한 연결 장치가 있습니다. 운영 체제의 소켓은 관련된 리소스 및 CPU의 논리적 그룹입니다. 이러한 개념은 대부분의 CPU 튜닝 논의에서 중심이 됩니다.

Red Hat Enterprise Linux는 시스템 CPU 이벤트에 대한 통계를 보관합니다. 이러한 통계는 CPU 성능 향상을 위해 튜닝 전략을 계획할 때 유용합니다. 4.1.2절. “CPU 성능 튜닝”에서는 보다 더 유용한 통계의 일부와 이러한 통계의 위치, 성능 튜닝을 위해 이를 분석하는 방법에 대해 설명합니다.

토폴로지

오래된 컴퓨터에는 시스템 당 CPU 수가 비교적 적어 **SMP** (Symmetric Multi-Processor)라는 아키텍처를 허용했습니다. 즉 이는 시스템에 있는 각각의 CPU는 사용 가능한 메모리로 유사하게 (또는 대칭) 액세스할 수 있음을 의미합니다. 최근에는 소켓 당 CPU 수가 많아졌기 때문에 시스템에 있는 모든 RAM에 대해 대칭 액세스를 제공하는 것은 고가의 비용이 들게 됩니다. CPU 수가 많은 시스템의 대부분에서는 SMP 대신 **NUMA** (Non-Uniform Memory Access)라는 아키텍처가 사용되고 있습니다.

AMD 프로세서는 **HT** (Hyper Transport) 상호 연결과 함께 이러한 유형의 아키텍처를 사용하고 있는 반면 Intel은 **QPI** (Quick Path Interconnect) 디자인에서 NUMA를 구현하기 시작했습니다. 애플리케이션에 리소스를 할당할 때 시스템의 토폴로지를 설명할 필요가 있으므로 NUMA 및 SMP는 다르게 튜닝됩니다.

스레드

Linux 운영 체제에서 실행 단위는 **스레드**라고 합니다. 스레드에는 레지스터 컨텍스트, 스택, CPU에서 실행되는 실행 가능한 코드의 세그먼트가 있습니다. 사용 가능한 CPU에서 이러한 스레드의 스케줄을 관리하는 것은 운영 체제의 일입니다.

OS는 사용 가능한 코어에 걸쳐 스레드 작업 부하의 균형을 유지하여 CPU 사용률을 극대화합니다. OS는 주로 CPU를 지속적으로 작동하게 하는 것이므로 애플리케이션 성능에 관해서는 최적의 판단을 내리지 않을 수 있습니다. 애플리케이션 스레드를 다른 소켓에 있는 CPU로 이동하는 것은 메모리 액세스 작업이 소켓에 걸쳐 크게 저하될 수 있기 때문에 간단하게 현재의 CPU를 사용할 수 있을 때까지 대기하는 것보다 성능이 더 악화될 수 있습니다. 고성능 애플리케이션의 경우 일반적으로 스레드를 배치할 위치는 설계자가 결정하는 것이 더 좋습니다. 4.2절. “CPU 스케줄링”에서는 애플리케이션 스레드를 최고로 실행하기 위해 최선의 CPU 및 메모리 할당 방법에 대해 설명합니다.

인터럽트

애플리케이션 성능에 영향을 미치는 시스템 이벤트 중에서 다소 명확하지 않은 (하지만 중요한) 것 중 하나는 **인터럽트** (Linux에서 **IRQ**라고 함)입니다. 이러한 이벤트는 운영 체제가 처리하고 데이터 도착 또는 네트워크 쓰기나 타이머 이벤트와 같은 작업의 완료를 알리기 위해 주변 장치를 사용합니다.

애플리케이션 코드를 실행하는 OS 또는 CPU가 인터럽트를 처리하는 방법은 애플리케이션의 기능에 영향을 주지 않습니다. 하지만 이는 애플리케이션의 성능에 영향을 미칠 수 있습니다. 다음 부분에서는 인터럽트가 애플리케이션 성능에 부정적인 영향을 주지 않도록 하는 방법에 대해 설명합니다.

4.1. CPU 토폴로지

4.1.1. CPU 및 NUMA 토폴로지

최초의 컴퓨터 프로세서는 **단일 프로세서** (*uniprocessors*)로 시스템에는 하나의 CPU만 있었습니다. 운영 체제가 단일 CPU를 하나의 실행 (프로세스) 스레드에서 다른 스레드로 신속하게 전환함으로써 프로세스의 병렬 처리라는 환상이 이루어졌습니다. 시스템 성능을 향상시키기 위해 설계자는 지시 사항을 빠르게 실행하기 위해 클럭 속도를 높여도 어느 지점 (주로 현재의 기술로 안정적인 클럭 파형을 생성할 수 있는

한계) 까지만 작동함을 인지했습니다. 전체적인 시스템 성능을 향상시키기 위해 설계자는 다른 CPU를 시스템에 추가하여 두 개의 병렬 스트림에서 실행할 수 있게 했습니다. 이러한 프로세서를 추가하는 경향은 오랫동안 계속될 것입니다.

대부분의 초기 멀티 프로세서 시스템에서 각각의 CPU는 각각의 메모리 위치에 동일한 논리적 경로를 가지고 있도록 설계되었습니다 (일반적으로 병렬 버스). 이를 통해 각각의 CPU는 시스템에 있는 다른 CPU와 동일한 시간에 모든 메모리 위치에 액세스할 수 있습니다. 이러한 유형의 아키텍처는 SMP (Symmetric Multi-Processor) 시스템이라고 합니다. SMP는 CPU 수가 적은 경우 잘 작동하지만 CPU 수가 일정 수 (8 또는 16)를 초과하면 메모리에 동일한 액세스를 허용하는데 필요한 병렬 추적 번호가 사용 가능한 보드 공간을 너무 많이 사용하여 주변 장치에 대한 공간이 부족해 집니다.

시스템에서 다수의 CPU를 허용하기 위해 두 가지 새로운 개념이 결합되었습니다:

1. 직렬 버스

2. NUMA 토폴로지

직렬 버스는 패킷 버스트로 데이터를 전송하는 매우 높은 클럭 속도를 갖는 단일 와이어 통신 경로입니다. 하드웨어 설계자는 직렬 버스를 CPU 간이나 CPU와 메모리 컨트롤러 및 기타 주변 장치 사이에서 고속의 상호 연결로 사용하기 시작했습니다. 즉, 각각의 CPU에서 메모리 서브 시스템 보드에 32에서 64의 추적을 필요로 하는 대신 하나의 추적을 필요로 하기 때문에 보드에서 필요한 공간이 크게 감소되었습니다.

동시에 하드웨어 설계자는 다이 크기를 줄임으로써 동일한 공간에 더 많은 트랜지스터를 정리했습니다. 메인 보드에 개별 CPU를 두지 않고 멀티 코어 프로세서로 프로세서 패키지에 이를 정리하기 시작했습니다. 그 후 각 프로세서 패키지에서 메모리에 동일한 액세스를 제공하는 것이 아니라 설계자는 NUMA (Non-Uniform Memory Access) 전략에 의지하였습니다. 즉 각 패키지/소켓의 조합으로 고속 액세스의 전용 메모리 영역이 하나 이상 있는 것입니다. 각 소켓은 다른 소켓 메모리로 느리게 액세스하기 위해 다른 소켓에 대해 상호 연결을 갖고 있습니다.

간단한 NUMA의 예로 소켓이 2개 있는 마더보드가 있다고 가정합니다. 여기서 각 소켓은 쿼드 코어 패키지로 채워져 있습니다. 즉, 시스템에 총 8 개의 CPU가 있고 각 소켓에 4개씩 있는 것입니다. 각 소켓은 4GB RAM 메모리 뱅크가 부착되어 있어 시스템의 전체 메모리는 8GB가 됩니다. 이러한 예의 경우 CPU 0-3은 소켓 0에 있고 CPU 4-7은 소켓 1에 있습니다. 예에서 각 소켓은 NUMA 노드에 해당합니다.

CPU 0의 경우 뱅크 0에서 메모리에 액세스하려면 3 클럭 사이클이 필요할 수 있습니다. 메모리 컨트롤러에 주소를 표시하는 사이클, 메모리 위치에 대한 액세스를 설정하는 사이클, 위치를 읽기 또는 쓰기하는 사이클입니다. 하지만 CPU 4의 경우 동일한 위치에서 메모리에 액세스하려면 6 클럭 사이클이 필요할 수 있습니다. 이는 다른 소켓에 있으므로 소켓 1의 로컬 메모리 컨트롤러와 소켓 0에 있는 원격 메모리 컨트롤러라는 두 개의 메모리 컨트롤러를 통과해야 하기 때문입니다. 메모리가 해당 위치에서 경쟁하는 경우 (즉, 두 개 이상의 CPU가 동일한 위치의 메모리에 동시에 액세스하려고 하는 경우) 메모리 컨트롤러는 메모리에 대한 액세스를 중재하고 직렬화해야 하므로 메모리 액세스 시간이 더 오래 걸릴 수 있습니다. 캐시 일관성을 추가하면 (로컬 CPU 캐시에 동일한 메모리 위치에 대해 동일한 데이터가 포함되어 있는지 확인하는) 프로세스는 더욱 복잡해 집니다.

Intel (Xeon) 및 AMD (Opteron)의 최신 하이엔드 프로세서에는 NUMA 토폴로지가 있습니다. AMD 프로세서는 HyperTransport 또는 HT라는 상호 연결을 사용하며 반면 Intel은 QuickPath Interconnect 또는 QPI라는 상호 연결을 사용합니다. 상호 연결은 다른 상호 연결, 메모리, 주변 장치로의 물리적 연결 방법에 차이가 있지만 실질적으로 다른 연결된 장치에서 하나의 연결된 장치로 투명한 액세스를 허용하는 스위치입니다. 이 경우 '투명성'은 "비용 없음" 옵션이 아니라 상호 연결을 사용하기 위해 특별한 프로그래밍 API가 필요하지 않다는 것을 의미합니다.

시스템 아키텍처는 매우 다양하므로 비로컬 메모리에 액세스하여 부과되는 성능 패널티를 구체적으로 특정화하는 것은 현실적이지 않습니다. 상호 연결의 각 홉이 홉 당 적어도 비교적 일정한 성능 패널티를 부과하고 있다는 것은 말할 수 있습니다. 현재 CPU에서 두 개의 상호 연결의 메모리 위치를 참조하면 적어도 $2N + \text{메모리 사이클 시간}$ 단위를 액세스 시간에 부과합니다. 여기서 N은 홉 당 패널티입니다.

성능 패널티를 고려하면 성능에 민감한 애플리케이션은 NUMA 토폴로지 시스템에 있는 원격 메모리에 정기적으로 액세스하지 않도록 해야 합니다. 이러한 애플리케이션은 특정 노드에서 유지하고 해당 노드에서 메모리를 할당하도록 설정해야 합니다.

이를 위해 애플리케이션은 다음과 같은 사항을 알고 있어야 합니다:

1. 시스템의 토폴로지는 무엇입니까?
2. 현재 애플리케이션은 어디에서 실행되고 있습니까?
3. 가장 가까운 메모리 뱅크는 어디에 있습니까?

4.1.2. CPU 성능 튜닝

다음 부분에서는 CPU 성능 튜닝 방법의 이해를 돕고 프로세스에 도움이 되는 여러 도구에 대해 소개합니다.

원래 NUMA는 단일 프로세서를 여러 메모리 뱅크에 연결하는데 사용되었습니다. CPU 제조 업체가 프로세서를 정교하게 하고 die 크기가 축소되면서 여러 CPU 코어는 하나의 패키지에 포함될 수 있었습니다. 이러한 CPU 코어는 클러스터되고 각각은 로컬 메모리 뱅크에 대한 액세스 시간이 동일하게 되어 코어 간에 캐시를 공유할 수 있었습니다. 하지만 코어, 메모리, 캐시 간의 상호 연결을 통한 각각의 '홉(hop)'에는 약간의 성능 패널티가 발생했습니다.

그림 4.1. “NUMA 토폴로지에서의 로컬 및 원격 메모리 액세스”에 있는 예시 시스템에는 두 개의 NUMA 노드가 있습니다. 각 노드에는 네 개의 CPU, 메모리 뱅크, 메모리 컨트롤러가 있습니다. 노드에 있는 CPU는 해당 노드의 메모리 뱅크에 직접 액세스할 수 있습니다. 노드 1에 있는 화살표를 따라 가면 다음과 같은 단계가 있습니다:

1. CPU (0-3 중 하나)는 로컬 메모리 컨트롤러로의 메모리 주소를 표시합니다.
2. 메모리 컨트롤러는 메모리 주소로의 액세스를 설정합니다.
3. CPU는 해당 메모리 주소에서 읽기/쓰기 작업을 수행합니다.

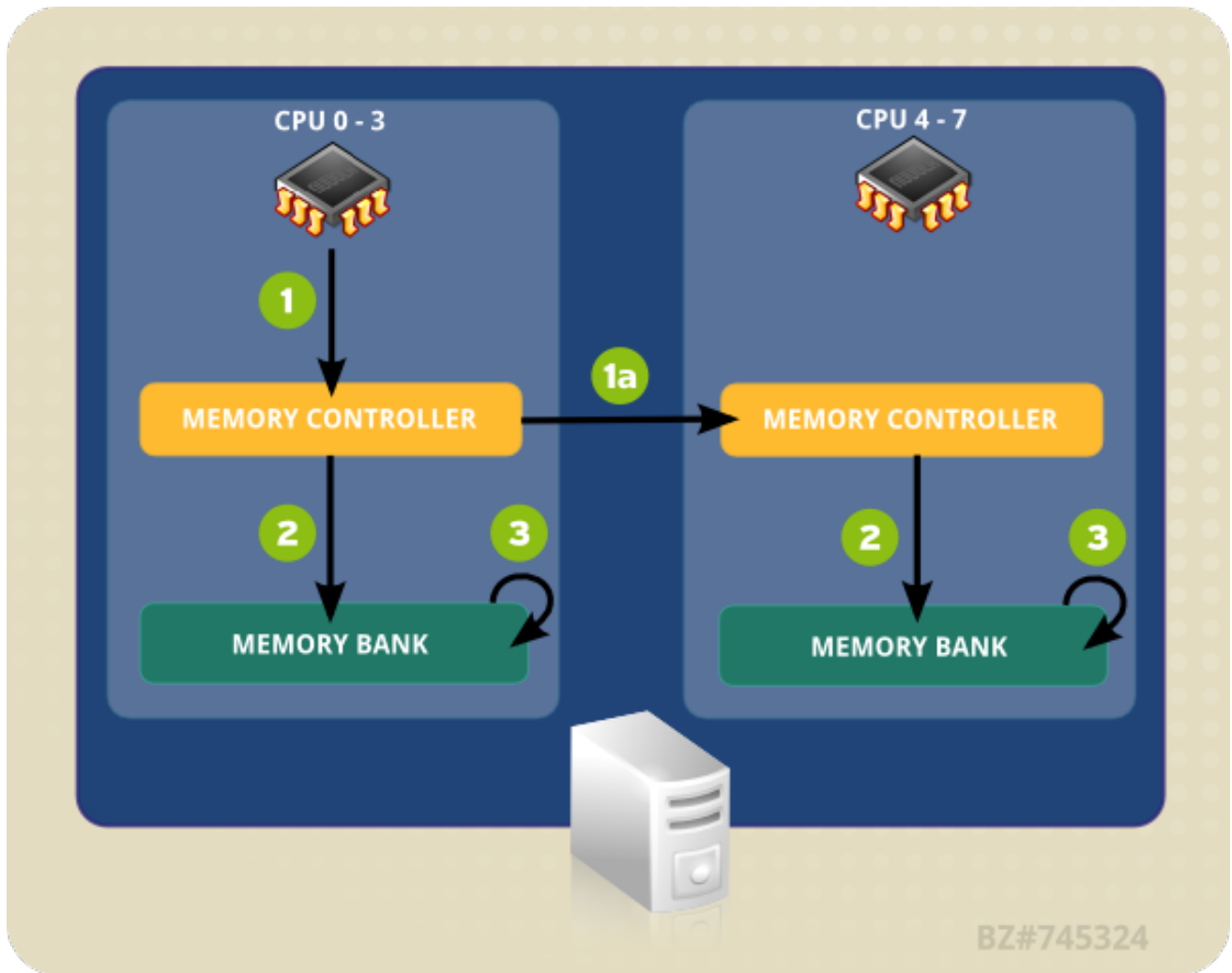


그림 4.1. NUMA 토폴로지에서 로컬 및 원격 메모리 액세스

하지만 하나의 노드에 있는 CPU가 다른 NUMA 노드의 메모리 뱅크에 있는 코드에 액세스해야 하는 경우 직접적인 경로가 아닙니다:

1. CPU (0-3 중 하나)는 로컬 메모리 컨트롤러에 원격 메모리 액세스를 표시합니다.
 1. 원격 메모리 주소의 CPU 요청은 메모리 주소가 있는 노드로의 로컬, 원격 메모리 컨트롤러에 전달됩니다.
2. 원격 메모리 컨트롤러는 원격 메모리 주소로의 액세스를 설정합니다.
3. CPU는 해당 원격 메모리 주소에서 읽기 또는 쓰기 작업을 수행합니다.

원격 메모리 주소를 액세스하려고 할 때 모든 작업은 여러 메모리 컨트롤러를 통과해야 하기 때문에 액세스에 두 배 이상의 시간이 걸릴 수 있습니다. 따라서 멀티 코어 시스템에서 성능의 주요 관심사는 정보가 가장 짧거나 빠른 경로로 가능한 효율적으로 이동할 수 있도록 하는 것입니다.

최적의 CPU 성능을 위해 애플리케이션을 설정하려면 다음과 같은 사항을 알고 있어야 합니다:

- 시스템의 토폴로지 (구성 요소가 연결된 방법)
- 애플리케이션을 실행하는 코어
- 가장 가까운 메모리 뱅크 위치

Red Hat Enterprise Linux 6에는 이러한 정보를 검색하고 이에 따라 시스템을 튜닝하는데 도움이 되는 여러 도구가 탑재되어 있습니다. 다음 부분에서는 CPU 성능 튜닝에 유용한 도구에 대한 개요를 설명합니다.

4.1.2.1. taskset으로 CPU 친화도 설정

taskset은 실행 중인 프로세스 (프로세스 ID에 따라)의 CPU 친화도를 검색 및 설정합니다. 이는 주어진 CPU 친화도를 사용하여 프로세스를 시작하는데 사용될 수 있으며, 이 경우 특정 프로세스를 특정 CPU 또는 CPU 모음에 연결합니다. 하지만 **taskset**은 로컬 메모리 할당을 보장하지 않습니다. 로컬 메모리 할당의 추가적 성능 향상이 필요한 경우 **taskset** 대신 **numactl**을 사용하는 것이 좋습니다. 보다 자세한 내용은 4.1.2.2절. “**numactl**로 NUMA 정책 제어”에서 참조하십시오.

CPU 친화도는 비트 마스크로 표시됩니다. 가장 낮은 순서의 비트는 첫 번째 논리 CPU에 해당하고 가장 높은 순서의 비트는 마지막 논리 CPU에 해당합니다. 이러한 마스크는 일반적으로 16 진수로 제공되어 **0x00000001**은 프로세서 0을 **0x00000003**은 프로세서 0 및 1을 나타냅니다.

실행 중인 프로세스의 CPU 친화도를 설정하려면 다음과 같은 명령을 실행합니다. **mask**는 프로세스를 연결하려는 프로세서 또는 프로세서의 마스크로 대체하고 **pid**는 변경하려는 친화도의 프로세스의 프로세스 ID로 대체합니다.

```
# taskset -p mask pid
```

주어진 친화도로 프로세스를 시작하려면 다음과 같은 명령을 실행합니다. 여기서 **mask**는 프로세스를 연결하려는 프로세서 또는 프로세서 마스크로 대체하고 **program**은 실행하려는 프로그램의 프로그램, 옵션, 인수로 대체합니다.

```
# taskset mask -- program
```

프로세서를 비트 마스크로 지정하는 대신 **-c** 옵션을 사용하여 별도의 프로세서를 콤마로 구분한 목록 또는 프로세서 범위를 제공할 수 있습니다. 예:

```
# taskset -c 0,5,7-9 -- myprogram
```

taskset에 관한 보다 자세한 내용은 **man taskset** man 페이지에서 참조하십시오.

4.1.2.2. numactl로 NUMA 정책 제어

numactl은 지정된 스케줄링 또는 메모리 배치 정책으로 프로세스를 실행합니다. 선택된 정책은 해당 프로세스 및 모든 자식 프로세스에 설정됩니다. **numactl**은 공유 메모리 세그먼트 또는 파일 용 영구 정책을 설정할 수 있고 프로세스의 CPU 친화도 및 메모리 친화도를 설정할 수 있습니다. 이는 **/sys** 파일 시스템을 사용하여 시스템 토폴로지를 지정합니다.

/sys 파일 시스템에는 CPU, 메모리, 주변 장치가 NUMA 상호 연결을 통해 연결된 방법에 관한 정보가 들어 있습니다. 특히 **/sys/devices/system/cpu** 디렉토리에는 시스템의 CPU가 각각 연결된 방법에 대한 정보가 들어 있습니다. **/sys/devices/system/node** 디렉토리에는 시스템의 NUMA 노드와 노드 간의 상대 거리에 대한 정보가 들어 있습니다.

NUMA 시스템에서 프로세서와 메모리 뱅크 간의 거리가 길수록 메모리 뱅크로의 프로세서의 액세스는 느려집니다. 따라서 성능 의존형 애플리케이션은 가장 가까운 메모리 뱅크에서 메모리를 할당하도록 설정해야 합니다.

성능 의존형 애플리케이션은 코어 설정 수에서 실행하도록 설정해야 합니다. 멀티 스레드 애플리케이션의 경우 특히 그러합니다. 첫 번째 레벨 캐시는 일반적으로 적기 때문에 여러 스레드가 하나의 코어에서 실행되는 경우 각 스레드는 이전 스레드가 액세스한 캐시된 데이터를 제거할 수 있습니다. 운영 체제가 이러한 스레드 사이에서 멀티 태스킹을 시도할 때 스레드가 서로의 캐시된 데이터를 지속적으로 제거하면 실행

시간 중 많은 부분이 캐시 라인 교체에 소요되어 버립니다. 이러한 문제는 *캐시 스레싱 (cache thrashing)*이라고 합니다. 따라서 멀티 스레드 애플리케이션을 단일 코어가 아닌 노드에 연결하는 것이 좋습니다. 이는 스레드가 여러 레벨 (첫번째, 두번째, 마지막 레벨 캐시)에서 캐시 라인을 공유하는 것을 허용하고 캐시 충돌 조작의 필요성을 최소화하기 때문입니다. 하지만 모든 스레드가 동일하게 캐시된 데이터에 액세스하는 경우 애플리케이션을 단일 코어에 바인딩하면 성능이 향상될 수 있습니다.

numactl을 사용하여 애플리케이션을 특정 코어 또는 NUMA 노드에 바인딩하여 코어 또는 코어 세트에 관련된 메모리를 할당할 수 있습니다. **numactl**이 제공하는 유용한 옵션은 다음과 같습니다:

--show

현재 프로세스의 NUMA 정책 설정을 표시합니다. 이 매개 변수는 추가 매개 변수를 필요로 하지 않으며 다음과 같이 사용할 수 있습니다: **numactl --show**

--hardware

시스템에서 사용 가능한 노드의 인벤토리를 표시합니다.

--membind

지정된 노드에서 메모리만 할당합니다. 노드가 사용 중이고 이러한 노드에 있는 메모리가 충분하지 않을 경우 할당은 실패하게 됩니다. 이러한 매개 변수의 사용 방법은 **numactl --membind=nodes program**입니다. 여기서 *nodes*는 메모리를 할당하고자 하는 노드 목록이고, *program*은 메모리 요구 사항을 해당 노드에서 할당해야 하는 프로그램입니다. 노드 번호는 콤마로 구분된 목록, 범위, 또는 이 두 가지 조합으로 지정할 수 있습니다. 보다 자세한 내용은 **numactl man** 페이지인 **man numactl**에서 참조하십시오.

--cpunodebind

지정된 노드에 속한 CPU에 있는 명령 (및 자식 프로세스)만을 실행합니다. 이 매개 변수의 사용 방법은 **numactl --cpunodebind=nodes program**입니다. 여기서 *nodes*는 지정된 프로그램 (*program*)이 바인딩되어야 하는 CPU의 노드 목록입니다. 노드 번호는 쉼표로 구분된 목록 또는 범위 또는 이 두 가지 조합으로 지정할 수 있습니다. 보다 자세한 내용은 **numactl man** 페이지인 **man numactl**에서 참조하십시오.

--physcpubind

지정된 CPU에 있는 명령 (및 자식 프로세스)만을 수행합니다. 이 매개 변수의 사용 방법은 **numactl --physcpubind=cpu program**입니다. 여기서 *cpu*는 **/proc/cpuinfo**의 프로세서 필드에 표시된 물리적 CPU 번호를 콤마로 구분한 목록이고 *program*은 이러한 CPU에서만 실행해야 하는 프로그램입니다. CPU는 현재 **cpuset**에 상대적으로 지정될 수 있습니다. 보다 자세한 내용은 **numactl man** 페이지인 **man numactl**에서 참조하십시오.

--localalloc

현재 노드에 항상 할당되어야 하는 메모리를 지정합니다.

--preferred

가능한 경우, 메모리는 지정된 노드에 할당됩니다. 메모리를 지정된 노드에 할당할 수 없는 경우 다른 노드로 대체합니다. 이 옵션은 **numactl --preferred=node**과 같이 하나의 노드 번호만을 사용합니다. 보다 자세한 내용은 **numactl man** 페이지인 **man numactl**에서 참조하십시오.

numactl 패키지에 포함된 **libnuma** 라이브러리는 커널이 지원하는 NUMA 정책에 간단한 프로그래밍 인터페이스를 제공합니다. 이는 **numactl** 보다 정교한 튜닝에 유용합니다. 보다 자세한 내용은 **man numa(3)** **man** 페이지에서 참조하십시오.

4.1.3. numastat



중요

이전에 **numastat** 도구는 Andi Kleen에 의해 작성된 Perl 스크립트였습니다. 이는 Red Hat Enterprise Linux 6.4 용으로 다시 작성되었습니다.

기본 명령 (**numastat**, 옵션이나 매개 변수 없이)은 도구의 이전 버전과 엄격하게 호환성을 유지하고 있지만 이 명령에 추가되는 옵션이나 매개 변수는 출력 내용과 포맷 모두가 변경됨에 유의합니다.

numastat는 프로세스 및 운영 체제에 대해 시스템의 메모리 통계 (할당 수 및 누락 등)를 NUMA 노드 당으로 표시합니다. 기본적으로 **numastat**를 실행하면 각 노드에 대해 다음과 같이 이벤트 카테고리에 의해 점유된 메모리 페이지 수를 표시합니다.

numa_miss 및 **numa_foreign** 값이 낮으면 CPU 성능이 최적화되어 있음을 나타냅니다.

numastat의 업데이트된 버전도 프로세스 메모리가 시스템에 걸쳐 확산되어 있는지 또는 **numactl**을 사용하여 특정 노드에 중앙 집중화되어 있는지를 표시합니다.

메모리가 할당된 노드와 동일한 노드에서 프로세스 스레드가 실행되고 있는지를 확인하려면 CPU 당 **top** 출력을 사용하여 **numastat** 출력을 상호 참조합니다.

기본 추적 카테고리

numa_hit

해당 노드에 할당을 시도하여 성공한 수입니다.

numa_miss

다른 노드에 할당 시도한 것으로 원래 의도된 노드에 메모리가 부족하여 해당 노드에 할당된 수입니다. 각각의 **numa_miss** 이벤트는 해당하는 **numa_foreign** 이벤트가 다른 노드에 있습니다.

numa_foreign

처음에는 해당 노드에 할당 의도한 것으로 대신 다른 노드에 할당된 수입니다. 각각의 **numa_foreign** 이벤트는 해당하는 **numa_miss** 이벤트가 다른 노드에 있습니다.

interleave_hit

해당 노드에 인터리브 정책 할당을 시도하여 성공한 수입니다.

local_node

해당 노드의 프로세스가 해당 노드에 있는 메모리 할당에 성공한 횟수입니다.

other_node

다른 노드의 프로세스가 해당 노드에 있는 메모리를 할당한 횟수입니다.

다음 옵션 중 하나를 적용하면 메모리의 표시 단위가 메가바이트로 변경됩니다. (소수점 두 자리 까지 반올림됨) 다음에서 설명하고 있듯이 다른 특정 **numastat** 동작을 변경합니다.

-c

표시된 정보 테이블을 가로로 축소합니다. 이는 NUMA 노드 수가 많은 시스템에서 유용하지만 열의 폭과 열 사이의 간격은 예측 불가능합니다. 이 옵션을 사용하면 메모리 양은 가장 가까운 메가바이트로 반올림됩니다.

-m

노드 당 시스템 전체 메모리 사용량을 표시합니다. 이는 `/proc/meminfo`에 있는 정보와 유사합니다.

-n

원래 측정 단위로 메가 바이트를 사용하여 업데이트된 형식으로 **numastat** 명령 (`numa_hit`, `numa_miss`, `numa_foreign`, `interleave_hit`, `local_node`, and `other_node`)과 동일한 정보가 표시됩니다.

-p *pattern*

지정된 패턴의 노드 당 메모리 정보를 표시합니다. *pattern*의 값이 숫자로 구성되면 **numastat**는 이를 숫자 프로세스 식별자로 간주합니다. 그렇지 않을 경우 **numastat**는 지정된 패턴의 프로세스 명령행을 검색합니다.

-p 옵션 값 다음에 입력되는 명령행 인수는 필터링 추가 패턴으로 간주됩니다. 추가 패턴은 필터를 좁히는 것이 아니라 확장합니다.

-s

표시된 데이터를 내림 차순으로 정렬하므로 (**total** 란에 따라) 메모리 사용량이 많은 것이 먼저 나열됩니다.

옵션으로 *node*를 지정하면 표는 *node* 란에 따라 정렬됩니다. 이 옵션을 사용할 때 다음과 같이 *node* 값은 **-s** 옵션 바로 뒤에 와야 합니다.

```
numastat -s2
```

옵션과 값 사이에 공백을 넣지 마십시오.

-v

자세한 정보를 표시합니다. 즉 여러 프로세스의 프로세스 정보는 각 프로세스에 대해 자세한 정보를 표시합니다.

-V

numastat 버전 정보를 표시합니다.

-Z

표시된 정보에서 제로 값을 갖는 표의 행과 열을 생략합니다. 표시 목적으로 제로로 반올림된 제로에 가까운 값은 표시된 출력에서 생략되지 않음에 유의합니다.

4.1.4. NUMA 친화도 관리 데몬 (numad)

numad는 자동 NUMA 친화도 관리 데몬입니다. 이는 시스템의 NUMA 토폴로지 및 리소스 사용량을 모니터링하여 동적으로 NUMA 리소스 할당 및 관리 (즉 시스템 성능)를 향상시킵니다.

시스템 작업 부하에 따라 **numad**는 벤치 마크 성능을 최대 50% 까지 개선할 수 있습니다. 이러한 성능을 개선하기 위해 **numad**는 정기적으로 `/proc` 파일 시스템에서 정보에 액세스하여 노드 당 사용 가능한 시스템 리소스를 모니터링합니다. 그 후 데몬은 충분히 정렬된 메모리와 최적의 NUMA 성능을 위한 CPU 리

소스를 갖는 NUMA 노드에 중요한 프로세스를 배치합니다. 프로세스 관리의 최신 임계값은 하나의 CPU의 최소 50%와 300 MB 메모리입니다. **numad**는 리소스 사용량 수준을 유지하고 필요시 NUMA 노드 간 프로세스를 이동하여 할당 균형을 다시 조정합니다.

numad는 다양한 작업 관리 시스템을 쿼리할 수 있는 사전 배포 컨설팅 서비스도 제공하고 있으며 프로세스의 CPU 및 메모리 리소스의 초기 바인딩으로 도움을 제공합니다. 이러한 사전 배포 컨설팅 서비스는 시스템에서 **numad**가 데몬으로 실행되고 있는지에 대한 여부와 상관 없이 사용 가능합니다. 사전 배포 컨설팅 서비스의 **-w** 옵션을 사용하는 방법에 대한 보다 자세한 내용은 **man numad man** 페이지에서 참조하십시오.

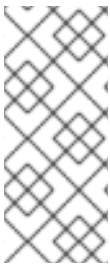
4.1.4.1. numad의 장점

numad는 주로 상당한 리소스 양을 소비하는 장기 실행 프로세스를 갖는 시스템에 이점을 제공합니다. 이러한 프로세스가 전체 시스템 리소스의 하위 집합에 포함되는 경우 특히 그러합니다.

numad는 여러 NUMA 노드의 리소스를 사용하는 애플리케이션에 유용합니다. 하지만 시스템에서 사용되는 리소스의 비율이 증가함에 따라 **numad**가 제공하는 혜택은 감소합니다.

프로세스의 실행 시간이 몇 분이거나 많은 리소스를 소비하지 않는 경우 **numad**는 성능을 개선하지 않을 가능성이 높습니다. 대형 메모리 데이터베이스와 같은 지속적으로 예상치 못한 메모리 액세스 패턴을 갖는 시스템도 **numad**의 사용으로 혜택을 얻을 가능성이 낮습니다.

4.1.4.2. 운영 모드



참고

커널 메모리 계산 통계는 대량의 병합 후 서로 상충하는 경우가 있습니다. 따라서 KSM 데몬이 대량의 메모리를 병합하면 **numad**에 혼란을 일으킬 수 있습니다. 차후 릴리즈에서 KSM 데몬은 보다 더 NUMA를 인식하게 됩니다. 하지만 현재는 시스템의 사용하지 않는 메모리의 양이 많은 경우 KSM 데몬을 튜닝 해제 및 비활성화하여 성능을 향상시킬 수 있습니다.

numad는 두 가지 방법으로 사용할 수 있습니다:

- 서비스로 사용
- 실행 파일로 사용

4.1.4.2.1. numad를 서비스로 사용

numad 서비스를 실행하는 동안 작업 부하에 따라 시스템을 동적으로 튜닝하려 합니다.

서비스를 시작하려면 다음 명령을 실행합니다:

```
# service numad start
```

재부팅 후에도 서비스를 유지하려면 다음 명령을 실행합니다:

```
# chkconfig numad on
```

4.1.4.2.2. numad를 실행 파일로 사용

numad를 실행 파일로 사용하려면 다음 명령을 실행합니다:

```
# numad
```

numad가 중지될 때 까지 계속 실행됩니다. 실행되는 동안 실행 동작은 **/var/log/numad.log**에 기록됩니다.

numad 관리를 특정 프로세스로 제한하려면 다음과 같은 옵션을 사용하여 시작합니다.

```
# numad -S 0 -p pid
```

-p pid

지정된 **pid**를 명시적 대상 목록에 추가합니다. 지정된 프로세스는 **numad** 프로세스 중요도 임계값에 도달할 때 까지 관리되지 않습니다.

-S mode

-S 매개 변수는 프로세스 스캔 유형을 지정합니다. 다음에서 볼 수 있듯이 이를 **0**으로 설정하면 **numad** 관리를 명시적으로 추가 프로세스에 한정합니다.

numad를 중지하려면 다음 명령을 실행합니다:

```
# numad -i 0
```

numad를 중지해도 NUMA 친화도를 개선하기 위해 변경한 내용이 삭제되지 않습니다. 시스템 사용이 크게 변경되는 경우 **numad**를 다시 실행하여 친화도를 조정하고 새로운 조건 하에서 성능이 향상됩니다.

사용 가능한 **numad** 옵션에 대한 보다 자세한 내용은 **numad man** 페이지 **man numad**에서 참조하십시오.

4.2. CPU 스케줄링

스케줄러는 시스템의 CPU를 지속적으로 가동시키는 역할을 합니다. Linux 스케줄러는 여러 스케줄링 정책을 구현하고 특정 CPU 코어에서 스레드를 언제 얼마나 오랫동안 실행할 지를 결정합니다.

스케줄링 정책은 다음과 같은 두 가지 주요 범주로 나눌 수 있습니다:

1. 실시간 정책

- SCHED_FIFO
- SCHED_RR

2. 일반 정책

- SCHED_OTHER
- SCHED_BATCH
- SCHED_IDLE

4.2.1. 실시간 스케줄링 정책

실시간 스레드가 먼저 스케줄되고 모든 실시간 스레드가 스케줄 된 후 일반 스레드가 스케줄됩니다.

실시간 정책은 인터럽트없이 완료해야 하는 시간이 중요한 작업에 사용됩니다.

SCHED_FIFO

이 정책은 **정적 우선 순위 스케줄링**이라고도 합니다. 이는 각 스레드의 고정 우선 순위 (1에서 99 사이)를 정의하기 때문입니다. 이 스케줄러는 **SCHED_FIFO** 스레드 목록을 우선 순위 순서로 스캔하여 실행할 준비가 된 가장 우선 순위가 높은 스레드를 스케줄합니다. 이 스레드는 차단, 종료 또는 실행할 준비가 된 보다 높은 우선 순위를 갖는 스레드로 대체될 때 까지 실행됩니다.

우선 순위가 가장 낮은 실시간 스레드도 비 실시간 정책 스레드 보다 먼저 스케줄됩니다. 하나의 실시간 스레드만이 존재할 경우 **SCHED_FIFO** 우선 순위 값은 중요하지 않습니다.

SCHED_RR

SCHED_FIFO 정책의 라운드 로빈 변형입니다. **SCHED_RR** 스레드에는 고정 우선 순위 (1에서 99 사이)가 주어집니다. 하지만 동일한 우선 순위를 갖는 스레드는 특정 양 또는 시간 범위 내에서 라운드 로빈 스타일이 스케줄됩니다. **sched_rr_get_interval(2)** 시스템 호출은 시간 범위 값을 반환하지만 시간 범위 길이를 사용자가 설정할 수 없습니다. 이 정책은 동일한 우선 순위로 여러 스레드를 실행해야 하는 경우 유용합니다.

실시간 스케줄링 정책의 정의된 의미에 대한 보다 자세한 내용은 시스템 인터페이스 — 실시간의 **IEEE 1003.1 POSIX** 표준에서 참조하십시오. 이는 http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_08.html에서 확인할 수 있습니다.

스레드 우선 순위 점의에 있어서 모범 사례는 낮은 순위에서 시작하여 정당한 지연이 확인된 경우에만 우선 순위를 증가시키는 것입니다. 실시간 스레드는 일반 스레드 처럼 기간이 있지 않습니다. **SCHED_FIFO** 스레드는 차단, 종료, 더 높은 우선 순위를 갖는 스레드로 대체될 때 까지 실행됩니다. 따라서 우선 순위를 99로 설정하는 것은 권장되지 않습니다. 이는 프로세스를 마이그레이션 및 위치독 스레드와 동일한 우선 순위 레벨로 배치하는 것이 됩니다. 이러한 스레드는 연산 루프에 들어갔기 때문에 이러한 스레드가 차단 되면 실행 불가능하게 됩니다. 이러한 상황에서 단일 프로세서 시스템은 궁극적으로 잠금되어 버립니다.

Linux 커널에서 **SCHED_FIFO** 정책에는 대역폭 제한 메커니즘이 포함되어 있습니다. 이는 CPU를 독점할 수 있는 실시간 작업에서 실시간 애플리케이션 프로그래머를 보호합니다. 이러한 메커니즘은 다음의 **/proc** 파일 시스템 매개 변수를 통해 조정할 수 있습니다:

/proc/sys/kernel/sched_rt_period_us

CPU 대역폭의 100% 가능한 시간을 마이크로초 단위로 정의합니다. ('us'는 일반 텍스트 'µs'에 가장 가까운 것에 해당됨) 기본값은 1000000µs 또는 1 초입니다.

/proc/sys/kernel/sched_rt_runtime_us

실시간 스레드 실행에 할당할 시간을 마이크로초 단위로 정의합니다 ('us'는 일반 텍스트 'µs'에 가장 가까운 것에 해당됨). 기본 값은 950000µs 또는 0.95 초입니다.

4.2.2. 일반 스케줄링 정책

일반 스케줄링 정책에는 **SCHED_OTHER**, **SCHED_BATCH**, **SCHED_IDLE**의 3 가지가 있습니다. 하지만 **SCHED_BATCH** 및 **SCHED_IDLE** 정책은 우선 순위가 매우 낮은 작업을 대상으로 하고 있어서 성능 조정 가이드에서는 관심이 낮아지고 있습니다.

SCHED_OTHER, 또는 SCHED_NORMAL

기본 스케줄링 정책입니다. 이 정책은 CFS (Completely Fair Scheduler)를 사용하여 이 정책을 사용하는 모든 스레드에 대해 공정한 액세스 기간을 제공합니다. CFS는 각 프로세스 스레드의 **niceness** 값

에 따라 부분적으로 동적 우선 순위 목록을 설정합니다. (이 매개 변수 및 **/proc** 파일 시스템에 대한 보다 자세한 내용은 *운용 가이드*에서 참조하십시오.) 이는 프로세스 우선 순위를 통해 사용자에게 간접적인 제어 수준을 제공하지만 동적 우선 순위 목록은 **CFS**를 통해 직접적으로 변경할 수 있습니다.

4.2.3. 정책 선택

애플리케이션의 스레드에 대해 올바른 스케줄링 정책을 선택하는 것은 항상 쉬운 작업이 아닙니다. 일반적으로 실시간 정책은 시간적으로 중요하거나 중요한 작업에 대해 신속하게 스케줄되어야 하고 장기간 실행되지 않는 작업에 대해 사용되어야 합니다. 일반 정책은 일반적으로 실시간 정책보다 더 나은 데이터 처리량 결과를 제공합니다. 이는 스케줄러가 보다 효율적으로 스레드를 실행하게 할 수 있기 때문입니다. (즉, 선점 (pre-emption)을 위해 너무 자주 재스케줄할 필요가 없습니다.)

대량의 스레드를 관리하고 있고 주로 데이터 처리량 (초 당 네트워크 패킷, 디스크에 쓰기 등)에 관련되어 있는 경우 **SCHED_OTHER**를 사용하여 시스템이 CPU 사용을 관리하게 합니다.

이벤트 반응 시간 (지연 시간)에 관련되어 있는 경우 **SCHED_FIFO**를 사용합니다. 스레드 수가 적은 경우 CPU 소켓을 고립시키고 스레드를 소켓의 코어로 이동시켜 코어에서 다른 스레드와 시간 경합을 하지 않도록 합니다.

4.3. 인터럽트 및 IRQ 튜닝

인터럽트 요청 (IRQ)은 하드웨어 레벨에서 전송되는 서비스에 대한 요청입니다. 인터럽트는 전용 하드웨어 라인이나 정보 패킷 (메시지 인터럽트 신호 또는 **MSI**)과 같은 하드웨어 버스를 통해 전송될 수 있습니다.

인터럽트가 활성화되면 IRQ 수신은 인터럽트 컨텍스트로의 전환을 묻는 메시지를 표시합니다. 커널 인터럽트 디스패치 코드는 IRQ 번호와 관련된 등록된 **ISR (Interrupt Service Routines)**의 목록을 검색하고 각 **ISR**을 차례로 호출합니다. **ISR**은 인터럽트를 승인하고 동일한 **ISR**에서 중복 인터럽트를 무시한 후 인터럽트 처리 완료 및 향후 인터럽트 무시에서 **ISR**을 중지시키기 위해 보류 처리기를 대기열에 넣습니다.

/proc/interrupts 파일은 I/O 장치 당 CPU 당 인터럽트 수를 나열합니다. 이는 IRQ 번호, 각 CPU 코어에 의해 처리되는 인터럽트 수, 인터럽트 유형, 인터럽트를 수신하도록 등록된 코마로 구분된 드라이버 목록을 표시합니다. (보다 자세한 내용은 **man 5 proc** proc(5) **man** 페이지에서 참조하십시오.)

IRQ에는 관련 "친화도" 속성 **smp_affinity**가 있어 해당 IRQ의 **ISR** 실행을 허용하는 CPU 코어를 정의합니다. 이러한 속성은 하나 이상의 특정 CPU 코어에 인터럽트 친화도 및 애플리케이션의 스레드 친화도를 할당하여 애플리케이션의 성능을 향상시키는데 사용될 수 있습니다. 이를 통해 지정된 인터럽트와 애플리케이션 스레드 간의 캐시 라인을 공유할 수 있습니다.

특정 IRQ 번호의 인터럽트 친화도 값은 관련 **/proc/irq/IRQ_NUMBER/smp_affinity** 파일에 저장되어 있으며 이는 **root** 사용자로 수정 및 확인할 수 있습니다. 이 파일에 저장된 값은 시스템에 있는 모든 CPU 코어를 나타내는 16 진수 비트 마스크입니다.

예를 들어 네 개의 CPU 코어를 갖는 서버 상의 이더넷 드라이버에 대한 인터럽트 친화도를 설정하려면 이더넷 드라이버와 관련된 IRQ 번호를 지정해야 합니다:

```
# grep eth0 /proc/interrupts
32: 0      140      45      850264      PCI-MSI-edge      eth0
```

해당 **smp_affinity** 파일을 배치하려면 IRQ 번호를 사용합니다:

```
# cat /proc/irq/32/smp_affinity
f
```

`smp_affinity`의 기본값인 **f**는 IRQ가 시스템에 있는 모든 CPU에서 서비스할 수 있다는 것을 의미합니다. 다음과 같이 이 값을 **1**로 설정하면 CPU 0 만이 이 인터럽트를 서비스할 수 있다는 것을 의미합니다:

```
# echo 1 >/proc/irq/32/smp_affinity
# cat /proc/irq/32/smp_affinity
1
```

코마를 사용하여 분리된 32 비트 그룹의 **smp_affinity** 값을 구분하는데 사용할 수 있습니다. 이는 32 개 이상의 코어를 갖는 시스템에 필요합니다. 예를 들어 다음의 예에서는 IRQ 40이 64 코어 시스템의 모든 코어에서 서비스되는 것을 보여줍니다:

```
# cat /proc/irq/40/smp_affinity
ffffffff,ffffffff
```

64 코어 시스템의 상위 32 코어에 있는 IRQ 40을 서비스하려면 다음을 수행해야 합니다:

```
# echo 0xffffffff,00000000 > /proc/irq/40/smp_affinity
# cat /proc/irq/40/smp_affinity
ffffffff,00000000
```



참고

인터럽트 스티어링을 지원하는 시스템에서 IRQ의 **smp_affinity**를 수정하여 하드웨어를 설정하고 인터럽트를 특정 CPU에서 실행하는 결정을 커널에서의 간섭없이 하드웨어 수준에서 수행할 수 있습니다.

4.4. RED HAT ENTERPRISE LINUX 6에서 NUMA 기능 강화

Red Hat Enterprise Linux 6에는 오늘날의 고도로 확장 가능한 하드웨어의 잠재력을 사용하기 위해 여러 향상된 기능이 포함되어 있습니다. 다음 부분에서는 Red Hat Enterprise Linux 6에서 제공하는 가장 중요한 NUMA-관련 성능 개선에 대한 높은 수준의 개요를 설명합니다.

4.4.1. 베어 메탈 및 확장성 최적화

4.4.1.1. 토폴로지 인식 기능 향상

다음과 같이 향상된 기능을 통해 Red Hat Enterprise Linux는 낮은 수준의 하드웨어 및 아키텍처 정보를 검색할 수 있으며 시스템의 프로세스를 자동으로 최적화하는 기능을 개선합니다.

토폴로지 검색 강화

이는 운영 체제가 낮은 수준의 하드웨어 정보 (논리 CPU 및 하이퍼 스레드, 코어, 소켓, NUMA 노드, 노드 간 액세스 시간 등) 부팅 시 검색할 수 있게 하며 시스템에서 프로세스를 최적화합니다.

완전 공정 스케줄러 (CFS, Completely Fair Scheduler)

이러한 새로운 스케줄링 모드는 런타임이 유효한 프로세스 간에 동등하게 공유되고 있는지 확인합니다. 이를 토폴로지 검색과 결합하여 프로세스가 동일한 소켓에 있는 CPU로 스케줄되어 고가의 원격 메모리 액세스에 대한 필요를 방지하고 가능한 캐시 내용이 보존되는지를 확인할 수 있습니다.

malloc

malloc은 프로세스에 할당된 메모리 영역이 프로세스가 실행되고 있는 코어에 물리적으로 가능한 가깝도록 최적화되어 있습니다. 이를 통해 메모리 액세스 속도가 높아집니다.

skbuff I/O 버퍼 할당

malloc과 유사하게 이는 장치 인터럽트와 같은 I/O 작업을 처리하는 CPU에 물리적으로 가까운 메모리를 사용하도록 최적화되어 있습니다.

장치 인터럽트 친화도

어떤 CPU가 어떤 인터럽트를 처리하는가에 관한 장치 드라이브에 의해 기록된 정보는 캐시 친화도를 보존하고 소켓 사이의 높은 볼륨간 소켓 통신을 제한하며, 동일한 물리적 소켓 내에 있는 CPU에 인터럽트 처리를 제한하는데 사용될 수 있습니다.

4.4.1.2. 멀티 프로세서 동기화 기능 개선

멀티 프로세서 간 작업을 조정하는 것은 병렬로 실행되고 있는 프로세서가 데이터 무결성을 손상하지 않도록 하기 위해 자주, 시간이 많이 걸리는 작업이 필요합니다. Red Hat Enterprise Linux에는 이러한 분야에서 성능을 향상시키기 위해 다음과 같은 개선된 기능이 포함되어 있습니다:

RCU (Read-Copy-Update) 잠금

일반적으로 잠금의 90%는 읽기 전용 목적으로 취득됩니다. RCU 잠금은 액세스되는 데이터가 수정되지 않을 때 단독 액세스 잠금을 획득할 필요성을 제거합니다. 이러한 잠금 모드는 현재 페이지 캐시 메모리 할당에 사용됩니다. 현재 잠금은 할당 또는 할당 취소 작업에만 사용됩니다.

CPU 당 및 소켓 당 알고리즘

대부분의 알고리즘은 보다 세밀한 잠금을 허용하도록 동일한 소켓에 있는 협력 CPU 간 잠금 조정을 수행하도록 업데이트되었습니다. 수많은 글로벌 스핀락 (spinlock)은 소켓 당 잠금 방법으로 대체되어 업데이트된 메모리 할당자 영역과 관련 메모리 페이지 목록은 작업 할당 또는 할당 취소 작업을 수행할 때 메모리 할당 논리가 보다 효율적으로 메모리 매핑 데이터 구조의 하위 집합을 통과할 수 있습니다.

4.4.2. 가상화 최적화

KVM은 커널 기능을 활용하고 있기 때문에 KVM 기반 가상화 게스트는 모든 베어 메탈 최적화에서 바로 혜택을 받을 수 있습니다. Red Hat Enterprise Linux에는 가상화 게스트가 베어메탈 시스템의 성능 수준에 접근할 수 있도록 여러 향상된 기능이 포함되어 있습니다. 이러한 향상된 기능은 스토리지 및 네트워크 액세스로의 I/O 경로에 초점을 두고 데이터 베이스 및 파일 서비스와 같은 집중적 워크로드가 가상화 배포를 사용할 수 있도록 하고 있습니다. 가상화 시스템의 성능을 개선하는 NUMA 고유의 기능 개선 사항은 다음과 같습니다:

CPU 핀 설정

가상 게스트가 로컬 캐시 사용을 최적화하고 고가의 소켓 간의 통신 및 원격 메모리 액세스에 대한 필요성을 제거하기 위해 특정 소켓에서 실행하도록 바인딩할 수 있습니다.

THP (transparent hugepages)

THP를 활성화하면 시스템은 연속적인 대량의 메모리에 대해 자동으로 NUMA 인식 메모리 할당 요청을 수행하고, 잠금 경합 및 TLB (translation lookaside buffer) 메모리 관리 작업 횟수가 감소되며, 가상화 게스트에서 최대 20% 까지 성능이 향상됩니다.

커널 기반 I/O 구현

가상 게스트의 I/O 서브시스템이 커널에 구현되어 대량의 컨텍스트 스위칭 및 동기화, 통신 오버 헤드를 방지하여 노드간 통신 및 메모리 액세스 비용이 현저하게 감소됩니다.

5장. 메모리

다음 부분에서는 Red Hat Enterprise Linux에서 사용 가능한 메모리 관리 기능의 개요와 이러한 관리 기능을 사용하여 시스템에서 메모리 사용을 최적화하는 방법에 대해 설명합니다.

5.1. HUGETLB (HUGE TRANSLATION LOOKASIDE BUFFER)

실제 메모리 주소는 메모리 관리의 일환으로 가상 메모리 주소로 변환됩니다. 물리적 주소에서 가상 주소로의 매핑 관계는 페이지 테이블이라는 데이터 구조에 저장됩니다. 모든 주소 매핑의 페이지 테이블을 읽는 데 시간 및 리소스를 소비하게 되므로 최근 사용된 주소에 대한 캐시가 있습니다. 이러한 캐시를 TLB (Translation Lookaside Buffer)라고 합니다.

하지만 TLB는 많은 주소 매핑만을 캐시할 수 있습니다. 요청된 주소 매핑이 TLB에 없는 경우, 가상 주소 매핑에 물리 주소를 결정하기 위해 여전히 페이지 테이블을 읽을 수 있어야 합니다. 이는 "TLB 미스"라고 합니다. 대용량 메모리 요구 사항을 갖는 애플리케이션은 최소 메모리 요구 사항을 갖는 애플리케이션보다 TLB 미스에 의해 영향을 받을 가능성이 더 높습니다. 이는 메모리 요구 사항 및 TLB의 캐싱 주소 매핑에 사용되는 페이지의 크기 사이의 관계 때문입니다. 각 미스에는 페이지 테이블을 읽어오는 것이 포함되므로 가능한 이러한 미스를 피하는 것이 중요합니다.

HugeTLB (Huge Translation Lookaside Buffer)는 매우 큰 세그먼트에서 메모리 관리를 가능하게 하는 것으로 한 번에 보다 많은 주소 매핑을 캐시할 수 있습니다. 이는 TLB 미스의 가능성을 감소시키고 그 결과 대용량 메모리 요구 사항을 갖는 애플리케이션의 성능이 향상됩니다.

HugeTLB 설정에 대한 내용은 커널 문서 `/usr/share/doc/kernel-doc-version/Documentation/vm/hugetlbpage.txt`에서 참조하십시오.

5.2. HUGE PAGES 및 TRANSPARENT HUGE PAGES

메모리는 *페이지 (pages)*라는 블록에서 관리됩니다. 한 페이지는 4096 바이트입니다. 1MB 메모리는 256 페이지에 상응하며 1GB 메모리는 256,000 페이지에 상응합니다. CPU에는 이러한 페이지 목록이 포함된 내장 메모리 관리 장치가 있고 각 페이지는 *페이지 테이블 엔트리*를 통해 참조됩니다.

대량의 메모리를 관리하기 위해 시스템을 활성화할 수 있는 다음과 같은 두 가지 방법이 있습니다:

- 하드웨어 메모리 관리 장치에서 페이지 테이블 엔트리 수를 늘림
- 페이지 크기를 확대함

현재 프로세서에 있는 하드웨어 메모리 관리 장치는 수백에서 수천 페이지 테이블 엔트리만을 지원하므로 첫 번째 방법은 비용이 많이 듭니다. 또한 수천 페이지 (메가바이트 메모리)에서 잘 작동하는 하드웨어 및 메모리 관리 알고리즘은 수백만 (심지어 수십억) 페이지에서는 실행에 어려움이 있을 수 있습니다. 이러한 경우 성능 문제를 일으킬 수 있습니다. 애플리케이션이 메모리 관리 장치가 지원하는 것 보다 많은 메모리 페이지를 사용해야 할 경우 시스템은 더 느린 소프트웨어 기반 메모리 관리로 폴백하여 시스템 전체가 더 느리게 실행됩니다.

Red Hat Enterprise Linux 6에서는 *huge pages*를 사용하여 두 번째 방법을 구현하고 있습니다.

간단하게 말하면 *huge pages*는 2MB 및 1GB 크기의 메모리 블록입니다. 2MB 페이지를 사용하는 페이지 테이블은 여러 기가 바이트의 메모리 관리에 적합한 반면 1GB 페이지의 페이지 테이블은 테라바이트 메모리까지의 확장에 가장 적합합니다.

Huge pages는 부팅시 할당해야 합니다. 이는 수동으로 관리하기 어려우므로 효과적으로 사용하기 위해 코드를 크게 변경해야 할 경우가 많습니다. 이를 위해 Red Hat Enterprise Linux 6에서는 *THP (transparent huge pages)*의 사용을 구현하고 있습니다. THP는 *huge pages*의 생성, 관리, 사용의 대부분을 자동화하는 추상화 계층입니다.

THP는 huge pages 사용에 있어서 복잡성을 시스템 관리자 및 개발자로 부터 제거합니다. THP의 목적은 성능 개선에 있으므로 개발자 (커뮤니티 및 Red Hat 모두)는 다양한 시스템, 설정, 애플리케이션, 워크로드에 걸쳐 THP를 테스트 및 최적화합니다. 이는 THP의 기본 설정이 대부분의 시스템 설정의 성능을 향상시킬 수 있게 합니다.

현재 THP는 힙과 스택 영역 등과 같은 익명의 메모리 영역에서만 매핑할 수 있음에 유의합니다.

5.3. 프로파일 메모리 사용에 VALGRIND 사용

Valgrind는 사용자 공간 바이너리에 대한 계측을 제공하는 프레임워크입니다. 이는 프로그램 성능을 프로파일링하고 분석하는데 사용할 수 있는 여러 도구와 함께 제공됩니다. 다음 부분에서 소개되는 도구는 메모리 사용 및 잘못된 메모리 할당 및 메모리 할당 해제와 같은 메모리 오류를 감지하는데 도움이 될 수 있는 분석을 제공합니다. 이러한 모든 것은 valgrind에 포함되어 다음과 같은 명령을 사용하여 실행할 수 있습니다:

```
valgrind --tool=toolname program
```

toolname을 사용하고자 하는 도구 이름 (메모리 프로파일링의 경우 memcheck, massif, cachegrind)으로 변경하고 program을 Valgrind으로 프로파일링하고자 하는 프로그램으로 변경합니다. Valgrind의 계측을 사용하면 프로그램이 평소보다 더 느리게 실행되는 원인이 될 수 있음에 유의합니다.

Valgrind 기능에 대한 개요는 3.5.3절. “Valgrind”에 설명되어 있습니다. Eclipse 용으로 사용 가능한 플러그인에 대한 정보를 포함한 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있는 개발자 가이드에서 확인할 수 있습니다. 부수적인 문서는 valgrind 패키지 설치 시 `man valgrind` 명령으로 확인하거나 다음의 위치에서 확인할 수 있습니다:

- `/usr/share/doc/valgrind-version/valgrind_manual.pdf`,
- `/usr/share/doc/valgrind-version/html/index.html`.

5.3.1. Memcheck로 메모리 사용량 프로파일링

Memcheck는 기본 Valgrind 도구로 `--tool=memcheck`를 지정하지 않고 `valgrind program`으로 실행될 수 있습니다. 발생해서는 안되는 메모리 액세스, 정의되지 않거나 초기화되지 않은 값의 사용, 올바르게 해제된 힙 메모리, 중복 포인터, 메모리 누수와 같은 감지 및 진단이 어려운 여러 메모리 오류를 감지하고 보고합니다. Memcheck를 사용하면 프로그램은 일반적으로 실행되는 것 보다 10-30 배 느리게 실행됩니다.

Memcheck는 감지된 문제의 종류에 따라 특정 오류를 반환합니다. 이러한 오류는 `/usr/share/doc/valgrind-version/valgrind_manual.pdf`에 포함된 Valgrind 문서에 자세히 설명되어 있습니다.

Memcheck는 이러한 오류를 보고만 할 수 있을 뿐 — 오류 발생을 방지할 수 없음에 유의합니다. 일반적으로 세그먼트 오류를 발생시킬 수 있는 방식으로 프로그램이 메모리에 액세스하는 경우 세그먼트 오류는 여전히 발생합니다. 하지만 Memcheck는 오류 직전에 오류 메시지를 기록합니다.

Memcheck는 검사 과정에 초점을 두는데 사용할 수 있는 명령행 옵션을 제공합니다. 이러한 옵션에는 다음과 같은 것이 있습니다:

--leak-check

활성화할 경우 Memcheck는 클라이언트 프로그램이 완료되면 메모리 누수를 검색합니다. 기본값은 **summary**로 발견된 누수 수를 출력합니다. 사용 가능한 다른 값은 **yes** 및 **full**로 모두 개별적 누수 세부 정보를 제공하며 **no**는 메모리 누수 검사를 비활성화합니다.

--undef-value-errors

활성화할 경우 (**yes**로 설정할 경우) Memcheck는 정의되지 않은 값이 사용되고 있을 때 오류를 보고합니다. 비활성화할 경우 (**no**로 설정할 경우) 정의되지 않은 값 오류는 보고되지 않습니다. 이는 기본 값으로 활성화되어 있습니다. 이를 비활성화하면 Memcheck 속도가 약간 빨라집니다.

--ignore-ranges

적용 가능성을 검사할 때 사용자는 Memcheck가 무시해야 하는 하나 이상의 범위를 지정할 수 있습니다. 여러 범위는 다음과 같이 콤마로 구분합니다. 예: **--ignore-ranges=0xPP-0xQQ, 0xRR-0xSS**

옵션의 전체 목록은 `/usr/share/doc/valgrind-version/valgrind_manual.pdf`에 있는 문서에서 참조하십시오.

5.3.2. Cachegrind로 캐시 사용량 프로파일링

Cachegrind는 시스템의 캐시 계층 및 (옵션으로) 분기 예측과 프로그램의 상호 작용을 시뮬레이션합니다. 이는 시뮬레이션된 첫 번째 레벨의 지시 사항 및 데이터 캐시 사용을 추적하여 이러한 캐시 레벨과 상호 작용한 불충분한 코드를 감지하고 마지막 레벨 캐시 (이것이 두 번째 또는 세 번째 레벨 캐시이던지 간에) 에서 메인 메모리로의 액세스를 추적합니다. 결과적으로 Cachegrind와 함께 실행되는 프로그램은 일반적으로 실행되는 것 보다 20-100배 더 느리게 실행됩니다.

Cachegrind를 실행하려면 다음과 같은 명령을 실행합니다. 여기서 *program*은 Cachegrind로 프로파일링 하고자 하는 프로그램으로 대체합니다.

```
# valgrind --tool=cachegrind program
```

Cachegrind는 전체 프로그램에서 그리고 프로그램에 있는 각 기능에 대해 다음과 같은 통계를 수집할 수 있습니다:

- 첫 번째 레벨 지시 캐시 읽기 (또는 실행된 명령) 및 읽기 미스, 마지막 레벨 지시 캐시 읽기 미스;
- 데이터 캐시 읽기 (또는 메모리 읽기), 읽기 미스, 마지막 레벨 캐시 데이터 읽기 미스;
- 데이터 캐시 쓰기 (또는 메모리 쓰기), 쓰기 미스 및 마지막 레벨 캐시 쓰기 미스
- 실행 및 잘못 예측된 조건 분기
- 실행 및 잘못 예측된 간접 분기

Cachegrind는 이러한 통계에 대한 요약 정보를 콘솔에 인쇄하고 파일에 보다 자세한 프로파일링 정보를 씁니다 (기본값으로 **cachegrind.out.pid**, 여기서 *pid*는 Cachegrind를 실행하는 프로그램의 프로세스 ID입니다). 이러한 파일은 다음과 같이 **cg_annotate** 도구로 추가 처리할 수 있습니다:

```
# cg_annotate cachegrind.out.pid
```



참고

cg_annotate는 경로의 길이에 따라 120 자 보다 긴 행을 출력할 수 있습니다. 출력을 명확하게 하고 읽기 쉽게하려면 위의 명령을 실행하기 전 창을 적어도 이 폭으로 확대하는 것이 좋습니다.

또한 `Cachegrind`에 의해 생성된 프로파일 파일을 비교하여 변경 전 후의 프로그램 성능을 쉽게 도표로 만들 수 있습니다. 이를 위해 `cg_diff` 명령을 사용합니다. 여기서 *first*는 초기 프로파일 출력 파일로 *second*는 후속 프로파일 출력 파일로 바꿉니다:

```
# cg_diff first second
```

이 명령은 결합된 출력 파일을 생성하며, 보다 자세한 내용은 `cg_annotate`를 사용하여 확인할 수 있습니다.

`Cachegrind`는 출력에 초점을 두는 옵션을 지원합니다. 사용 가능한 옵션 중 일부는 다음과 같습니다:

--I1

콤마로 구분하여 첫 번째 레벨 지시 캐시의 크기, 연결성, 행의 크기를 지정합니다: --
I1=size, associativity, line size.

--D1

콤마로 구분하여 첫 번째 레벨 데이터 캐시의 크기, 연결성, 행의 크기를 지정합니다: --
D1=size, associativity, line size.

--LL

콤마로 구분하여 마지막 레벨 캐시의 크기, 연결성, 행의 크기를 지정합니다: --
LL=size, associativity, line size.

--cache-sim

캐시 액세스 및 미스 카운트의 모음을 활성화 또는 비활성화합니다. 기본값은 **yes** (활성화)입니다.

이 옵션 및 **--branch-sim** 모두를 비활성화하면 `Cachegrind`에는 수집할 정보가 없게 됨에 유의합니다.

--branch-sim

분기 지시 및 예측 실패 수의 모음을 활성화 또는 비활성화합니다. 이는 `Cachegrind`를 약 25% 느리게 할 수 있으므로 기본값은 **no** (비활성화)로 설정되어 있습니다.

이 옵션 및 **--cache-sim** 모두를 비활성화하면 `Cachegrind`에는 수집할 정보가 없게 됨에 유의합니다.

옵션의 전체 목록은 `/usr/share/doc/valgrind-version/valgrind_manual.pdf`에 있는 문서에서 참조하십시오.

5.3.3. Massif를 사용하여 힙 및 스택 영역 프로파일링

`Massif`는 특정 프로그램이 사용하는 힙 영역을 측정합니다. 사용 가능한 공간 및 부기 및 조정 목적으로 할당된 추가 공간 모두에 해당합니다. 이는 프로그램에 의해 사용되는 메모리 양을 줄이는데 도움을 줄 수 있습니다. 이로 인해 프로그램 속도가 개선되고 프로그램을 실행하는 컴퓨터의 스왑 공간을 프로그램이 고갈할 가능성을 감소시킬 수 있습니다. `Massif`는 프로그램의 어떤 부분이 힙 메모리 할당에 대해 관여하는가에 대한 자세한 정보도 제공합니다. `Massif`와 함께 실행되는 프로그램은 정상적인 실행 속도보다 약 20 배 더 느리게 실행됩니다.

프로그램의 힙 사용량을 프로파일링하려면 사용하고자 하는 `Valgrind` 도구로 **massif**를 지정합니다:

```
# valgrind --tool=massif program
```

Massif가 수집한 프로파일링 데이터는 기본적으로 **massif.out.pid**라는 파일에 기록됩니다. 여기서 **pid**는 지정된 **program**의 프로세스 ID입니다.

이러한 프로파일링 데이터는 다음과 같이 **ms_print** 명령으로 그래프화할 수 있습니다:

```
# ms_print massif.out.pid
```

이는 프로그램의 실행에 대한 메모리 사용량을 표시하는 그래프를 생성하며 최고 메모리 할당 지점을 포함하는 프로그램에 있는 다양한 지점에서 할당을 담당하는 사이트에 대한 자세한 정보도 생성합니다.

Massif는 도구의 출력을 지시하는데 사용할 수 있는 명령행 옵션을 제공합니다. 사용 가능한 옵션 중 일부는 다음과 같습니다:

--heap

힙 프로파일링을 수행할 지에 대한 여부를 지정합니다. 기본값은 **yes**입니다. 이 옵션을 **no**로 설정하면 힙 프로파일링을 비활성화할 수 있습니다.

--heap-admin

힙 프로파일링을 활성화할 때 관리에 사용할 블록 당 바이트 수를 지정합니다. 기본값은 블록 당 **8** 바이트입니다.

--stacks

스택 프로파일링을 수행할 지에 대한 여부를 지정합니다. 기본값은 **no** (비활성화)입니다. 스택 프로파일링을 활성화하려면 이 옵션을 **yes**로 설정합니다. 하지만 이러한 설정으로 Massif가 현저히 느려지게 될 것임에 유의합니다. 또한 프로파일링된 프로그램이 제어하는 스택 부분의 크기를 보다 알기 쉽게 표시하기 위해 Massif는 메인 스택 크기가 시작 시 **0**이라고 가정하고 있다는 점에 유의합니다.

--time-unit

프로파일링에 사용되는 시간 단위를 지정합니다. 이 옵션에는 세 개의 유효한 값이 있습니다: 실행된 명령 (**i**), 기본값으로 대부분의 경우 유용합니다. 실시간 (**ms** 밀리초 단위), 특정 인스턴스에서 유용합니다. 힙 또는 스택에서 할당/할당해제된 바이트 (**B**), 이는 다른 시스템에서 가장 재생 가능해서 단기 실행 프로그램 및 테스트 목적으로 유용합니다. 이 옵션은 **ms_print**로 Massif 출력을 그래프화할 때 유용합니다.

옵션의 전체 목록은 **/usr/share/doc/valgrind-version/valgrind_manual.pdf**에 있는 문서에서 참조하십시오.

5.4. 용량 튜닝

다음 부분에서는 메모리, 커널, 파일 시스템 용량, 각각에 관련된 매개 변수, 이러한 매개 변수의 조정에서 절충 사항에 대한 개요를 설명합니다.

튜닝하는 도중 이러한 값을 일시적으로 설정하려면 **proc** 파일 시스템에서 해당 파일에 원하는 값을 **echo** 명령으로 실행합니다. 예를 들어, **overcommit_memory**를 일시적으로 **1**로 설정하려면 다음을 실행합니다:

```
# echo 1 > /proc/sys/vm/overcommit_memory
```

proc 파일 시스템에서 매개 변수로의 경로는 변경에 영향을 받는 시스템에 따라 다르다는 것에 유의합니다.

이 값을 영구적으로 설정하려면 **sysctl** 명령을 사용해야 합니다. 보다 자세한 내용은 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *운용 가이드*에서 참조하십시오.

튜닝 가능한 용량 관련 메모리

다음의 각 매개 변수는 **proc** 파일 시스템에 있는 **/proc/sys/vm/**에 있습니다.

overcommit_memory

대량 메모리 요청을 수락 또는 거부할 지에 대해 결정하는 조건을 지정합니다. 이러한 매개 변수에 대해 세 가지 사용 가능한 값이 있습니다:

- **0** — 기본 설정입니다. 커널은 사용 가능한 메모리 양을 추정하고 잘못된 요청을 실패시켜 휴리스틱 메모리 오버커밋 처리를 수행합니다. 불행히도 메모리는 정확한 알고리즘이 아닌 휴리스틱 알고리즘을 사용하여 할당되므로 이러한 설정은 시스템에서 사용 가능한 메모리가 오버로드되게 할 수 있습니다.
- **1** — 커널은 메모리 오버커밋 처리를 수행하지 않습니다. 이러한 설정에서 메모리 오버로드 가능성은 증가하므로 메모리 집약적 작업을 위한 성능입니다.
- **2** — 커널은 총 사용 가능한 스왑의 합계와 **overcommit_ratio**에 지정된 물리적 RAM의 백분율 보다 메모리가 크거나 동일한 경우 요청을 거부합니다. 메모리 오버커밋의 위험을 줄이고자 할 경우 이 설정이 가장 적합합니다.



참고

이 설정은 스왑 공간이 실제 메모리보다 큰 시스템의 경우에만 권장됩니다.

overcommit_ratio

overcommit_memory가 **2**로 설정되어 있는 경우 고려해야 할 물리적 RAM의 백분율을 지정합니다. 기본값은 **50**입니다.

max_map_count

프로세스가 사용할 수 있는 메모리 맵 영역의 최대 수를 지정합니다. 대부분의 경우 기본값으로 **65530**이 적절합니다. 애플리케이션에 이 파일 보다 많은 수를 매핑해야 하는 경우 이 값을 늘립니다.

nr_hugepages

커널에서 설정되는 hugepage 수를 지정합니다. 기본값은 **0**입니다. 시스템에 물리적으로 연속된 빈 페이지가 충분할 경우에만 hugepage를 할당 (또는 할당 해제)할 수 있습니다. 이러한 매개 변수에 의해 예약된 페이지를 다른 목적으로 사용할 수 없습니다. 보다 자세한 내용은 **/usr/share/doc/kernel-doc-kernel_version/Documentation/vm/hugetlbpage.txt**에 있는 설치된 문서에서 참조하십시오.

튜닝 가능한 용량 관련 커널

다음의 각 매개 변수는 **proc** 파일 시스템에 있는 **/proc/sys/kernel/**에 있습니다.

msgmax

메세지 큐에서 단일 메세지의 최대 크기를 바이트 단위로 지정합니다. 이 값은 큐의 크기 (**msgmnb**)를 초과해서는 안 됩니다. 기본값은 **65536**입니다.

msgmnb

단일 메시지 큐의 최대 크기를 바이트 단위로 지정합니다. 기본값은 **65536** 바이트입니다.

msgmni

메시지 큐 식별자의 최대 수를 지정합니다 (따라서 큐의 최대 수). 64 비트 아키텍처 시스템에서 기본값은 **1985**이고 32 비트 아키텍처에서 기본값은 **1736**입니다.

shmall

한 번에 시스템에서 사용할 수 있는 총 공유 메모리 양을 바이트 단위로 지정합니다. 64 비트 아키텍처 시스템에서 기본값은 **4294967296**이며 32 비트 아키텍처에서 기본값은 **268435456**입니다.

shmmax

커널이 허용하는 최대 공유 메모리 세그먼트를 바이트 단위로 정의합니다. 64 비트 아키텍처 시스템에서 기본값은 **68719476736**이고 32 비트 아키텍처의 경우 기본값은 **4294967295**입니다. 하지만 커널은 이 보다 더 큰 값을 지원함에 유의합니다.

shmmni

시스템 전체의 공유 메모리 세그먼트의 최대 수를 지정합니다. 64 비트 및 32 비트 아키텍처에서 기본값은 **4096**입니다.

threads-max

시스템 전체에서 커널이 한번에 사용할 스레드 (작업)의 최대 수를 지정합니다. 기본값은 커널 **max_threads** 값과 동일합니다. 사용되는 식은 다음과 같습니다:

```
max_threads = mempages / ( 8 * THREAD_SIZE / PAGE_SIZE )
```

threads-max의 최저 값은 **20**입니다.

튜닝 가능한 용량 관련 파일 시스템

다음의 각 매개 변수는 proc 파일 시스템에 있는 **/proc/sys/fs/**에 있습니다.

aio-max-nr

모든 활성 비동기 I/O 컨텍스트에서 허용되는 최대 이벤트 수를 지정합니다. 기본값은 **65536**입니다. 이 값을 변경해도 커널 데이터 구조를 미리 할당하거나 크기 변경이 되지 않음에 유의합니다.

file-max

커널이 할당된 최대 파일 처리 개수를 나열합니다. 기본값은 커널에서 **files_stat.max_files**의 값과 일치하고 이는 **(mempages * (PAGE_SIZE / 1024)) / 10**, 또는 **NR_FILE** (Red Hat Enterprise Linux에서는 8192) 중 가장 큰 값으로 설정됩니다. 이 값이 높을 수록 사용 가능한 파일 처리 부족으로 인한 오류를 해결할 수 있습니다.

튜닝 가능한 메모리 부족 종료

OOM (Out of Memory)는 스왑 공간을 포함한 모든 사용 가능한 메모리가 할당된 컴퓨터 상태를 말합니다. 기본적으로 이러한 사항은 시스템 패닉의 원인이 되어 예상대로 작동을 중지합니다. 하지만 **/proc/sys/vm/panic_on_oom** 매개 변수를 **0**으로 설정하면 OOM 발생 시 커널이 **oom_killer** 기능

을 호출합니다. 일반적으로 **oom_killer**는 악성 프로세스를 종료하고 시스템을 유지합니다.

다음과 같은 매개 변수는 프로세스 마다 설정할 수 있으며, **oom_killer** 기능에 의해 강제 종료되는 프로세스의 제어 능력을 증가시킵니다. 이는 **proc** 파일 시스템에 있는 **/proc/pid/**에 있으며 여기서 **pid**는 프로세스 ID 번호입니다.

oom_adj

값을 **-16**에서 **15**로 지정하면 프로세스의 **oom_score**를 결정하는데 도움이 됩니다. **oom_score** 값이 높으면 **oom_killer**에 의해 프로세스가 종료될 가능성이 높아집니다. **oom_adj** 값을 **-17**로 설정하면 프로세스의 **oom_killer**가 비활성화됩니다.



중요

조정된 프로세스에 의해 생성된 모든 프로세스는 해당 프로세스의 **oom_score**를 상속합니다. 예를 들어, **sshd** 프로세스가 **oom_killer** 기능에서 보호되고 있을 경우 SSH 세션에 의해 시작된 모든 프로세스도 모두 보호됩니다. 이는 OOM 발생시 시스템을 구제하기 위한 **oom_killer** 기능에 영향을 미칠 수 있습니다.

5.5. 가상 메모리 튜닝

가상 메모리는 프로세스, 파일 시스템 캐시, 커널에 의해 소모됩니다. 가상 메모리 사용률은 다음과 같은 매개 변수에 의해 영향을 받을 수 있는 여러 요소에 따라 달라집니다:

swappiness

0에서 100의 값으로 시스템이 스왑하는 정도를 제어합니다. 값이 높으면 시스템 성능에 우선 순위를 두고 물리적 메모리가 활성화되지 않은 경우 메모리에서 프로세스를 적극적으로 스왑합니다. 값이 낮으면 상호 작용에 우선 순위를 두고 가능한 오래동안 물리적 메모리에서 프로세스 스왑을 피하기 때문에 응답 지연을 감소시킵니다. 기본값은 **60**입니다.

min_free_kbytes

시스템 전체에 걸쳐 빈 공간으로 두는 최소 크기 (KB)입니다. 이 값은 각각의 낮은 메모리 영역에서 워터마크 값을 계산하는데 사용되며 그 후 그 크기에 비례하여 저장된 빈 페이지 수를 할당합니다.



주의

이 매개 변수를 설정할 때 주의하십시오. 값이 너무 낮거나 너무 높으면 시스템을 손상시킬 수 있습니다.

min_free_kbytes를 너무 낮게 설정하면 시스템이 메모리 회수를 실행하지 못할 수 있습니다. 이로 인해 시스템이 중단되고 메모리 부족으로 인해 여러 프로세스가 종료될 수 있습니다.

하지만 이러한 매개 변수를 너무 높은 값 (총 시스템 메모리의 **5-10%**)으로 설정하면 바로 시스템 메모리 부족을 초래하게 됩니다. Linux는 캐시 파일 시스템 데이터에 사용 가능한 모든 **RAM**을 사용하도록 설계되어 있습니다. **min_free_kbytes** 값을 높게 설정하면 시스템이 메모리를 회수하는데 너무 많은 시간을 소모하게 됩니다.

dirty_ratio

백분율 값을 정의합니다. 더티 데이터의 Writeout은 (**pdflush**를 통해) 더티 데이터가 총 시스템 메모리의 이러한 퍼센트를 차지하는 때에 시작됩니다. 기본값은 **20**입니다.

dirty_background_ratio

백분율 값을 정의합니다. 더티 데이터의 Writeout은 (**pdflush**를 통해) 더티 데이터가 총 메모리의 이러한 퍼센트를 차지하는 때에 백그라운드에서 시작됩니다. 기본값은 **10**입니다.

drop_caches

이 값을 **1**, **2**, **3** 중 하나로 설정하면 커널은 페이지 캐시와 슬랩 캐시의 다양한 조합을 드롭하게 됩니다.

1

시스템은 모든 페이지 캐시 메모리를 비활성화하여 해제합니다.

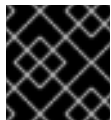
2

시스템은 사용되지 않는 모든 슬랩 캐시 메모리를 해제합니다.

3

시스템은 모든 페이지 캐시 및 슬랩 캐시 메모리를 해제합니다.

이는 비파괴적인 작업입니다. 더티 개체가 해제될 수 없으므로 이 매개 변수의 값을 설정하기 전 **sync**를 실행하는 것이 좋습니다.



중요

프로덕션 환경에서 **drop_caches**를 사용한 메모리 해제는 권장되지 않습니다.

튜닝하는 도중 이러한 값을 일시적으로 설정하려면 **proc** 파일 시스템에서 해당 파일에 원하는 값을 **echo** 명령으로 실행합니다. 예를 들어, **swappiness**를 일시적으로 **50**으로 설정하려면 다음을 실행합니다:

```
# echo 50 > /proc/sys/vm/swappiness
```

이 값을 영구적으로 설정하려면 **sysctl** 명령을 사용해야 합니다. 보다 자세한 내용은 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/의 *운용 가이드*에서 참조하십시오.

6장. 입/출력

6.1. 특징

Red Hat Enterprise Linux 6에서는 I/O 스택에서 여러 성능 개선 사항을 소개하고 있습니다.

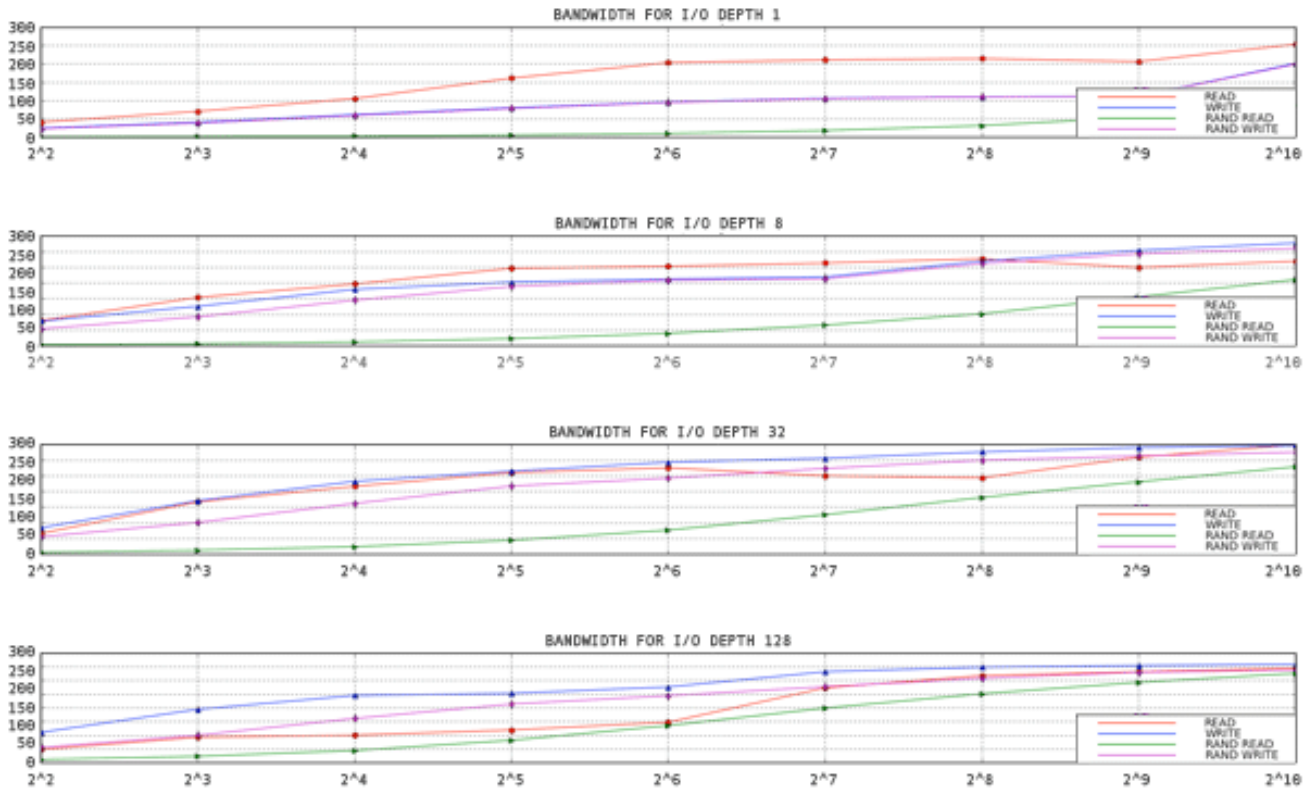
- SSD (Solid state disk)가 자동으로 인식되고 이러한 장치가 수행할 수 있는 초당 높은 I/O (IOPS)의 장점을 취하도록 I/O 스케줄러의 성능이 조정됩니다.
- 삭제 지원이 커널에 추가되어 기본 스토리지에 사용되지 않는 블록 범위가 보고됩니다. 이는 SSD의 웨어 레벨링 (wear-leveling) 알고리즘에 도움이 됩니다. 또한 사용 중인 실제 스토리지 용량에 가까운 탭을 유지하여 논리 블록 프로비저닝 (스토리지 용 가상 주소 공간의 일종)을 지원하는 스토리지에 도움이 됩니다.
- Red Hat Enterprise Linux 6.1에서 파일 시스템 장애 구현이 보다 철저히 정비되어 성능이 개선되었습니다.
- **pdflush**가 per-backing-device flusher 스레드로 대체되어 대량의 LUN 카운트 설정에서 시스템 확장성이 크게 개선되었습니다.

6.2. 분석

스토리지 스택 성능을 성공적으로 튜닝하려면 기본 적인 스토리지 관련 지식과 다양한 워크로드에 따른 실행 방법, 시스템을 통해 데이터가 이동하는 방법에 대해 이해하고 있어야 합니다. 또한 튜닝되는 실제 워크로드에 대한 이해도 필요합니다.

새로운 시스템을 도입할 때 마다 스토리지를 상향식으로 프로파일링하는 것이 좋습니다. 원 LUN 또는 디스크로 시작하여 성능을 직접 I/O (커널의 페이지 캐시를 우회하는 I/O)를 사용하여 평가합니다. 이는 실행할 수 있는 테스트 중 가장 기본적인 것으로 스택에서 I/O 성능을 측정할 때 기준이 되는 것입니다. 다양한 I/O 크기 및 큐 깊이에 걸쳐 연속 및 임의 읽기 및 쓰기를 수행하는 기본적인 워크로드 생성기 (예: **aio-stress**)로 시작합니다.

다음은 **aio-stress** 실행 그래프입니다. 각각 연속 쓰기, 연속 읽기, 임의 쓰기, 임의 읽기의 네 단계를 수행합니다. 이 예제에서 도구는 여러 레코드 크기 (x 축)와 큐 깊이 (그래프 당 하나)의 범위에 걸쳐 실행되도록 설정되어 있습니다. 큐 깊이는 주어진 시점에서 진행 중인 총 I/O 작업 수를 나타냅니다.



y 축은 초당 대역폭을 메가바이트 단위로 표시합니다. x 축은 I/O 크기를 킬로바이트 단위로 표시합니다.

그림 6.1. 1 스프레드, 1 파일 용 aio-stress 출력

왼쪽 하단 코너에서 오른쪽 상단 까지 처리량 선 기울기에 유의합니다. 주어진 레코드 크기의 경우 진행 중인 I/O 수를 늘려 스토리지에서 더 많은 처리량을 얻을 수 있음에 유의합니다.

이러한 스토리지에서 이러한 간단한 작업을 실행하여 부하에서 스토리지가 수행하는 방법을 이해할 수 있습니다. 이러한 테스트에서 생성된 데이터를 저장하여 보다 복잡한 워크로드를 분석할 때 비교합니다.

장치 매핑 또는 md를 사용하는 경우 해당 레이어를 추가하고 테스트를 반복합니다. 성능이 크게 저하될 경우 설명할 수 있는 것인지 또는 예상된 것인지를 확인합니다. 예를 들어 스택에 체크섬 RAID 계층이 추가된 경우 성능 저하가 예상됩니다. 의외의 성능 저하는 잘못 조정된 I/O 작업에 의해 발생할 수 있습니다. 기본적으로 Red Hat Enterprise Linux는 파티션 및 장치 매핑 메타 데이터를 최적으로 조정합니다. 그러나 모든 유형의 스토리지가 최적의 튜닝 상태를 보고하지는 않으므로 수동으로 튜닝해야 하는 경우도 있습니다.

장치 매핑 또는 md 레이어를 추가한 후 블록 장치에 파일 시스템을 추가하고 직접 I/O를 사용하여 이에 대한 테스트를 수행합니다. 테스트 결과를 테스트 이전의 것과 비교하여 일치하지 않는 부분을 이해하고 있는지 확인합니다. 직접 쓰기 I/O는 일반적으로 사전 할당 파일에서 성능이 더 나으므로 성능 테스트 전에 파일을 미리 할당하도록 합니다.

유용한 합성 워크로드 생성기는 다음과 같습니다:

- aio-stress
- iotop
- fio

6.3. 도구

I/O 서브시스템에서 성능 문제를 진단하기 위한 여러 유용한 도구가 있습니다. vmstat는 시스템 성능에

대한 개요를 제공합니다. 다음의 칼럼은 I/O에 가장 관련되어 있습니다: **si** (swap in), **so** (swap out), **bi** (block in), **bo** (block out), **wa** (I/O wait time). **si** 및 **so**는 스왑 공간이 데이터 파티션 및 전체 메모리 압력 표시자와 같은 장치에 있을 때 유용합니다. **si** 및 **bi**는 읽기 작업인 반면 **so** 및 **bo**는 쓰기 작업입니다. 각 카테고리는 킬로바이트 단위로 보고됩니다. **wa**는 대기 시간으로 실행 큐의 어느 부분이 I/O 완료를 기다리며 차단되어 있는지를 보여줍니다.

vmstat로 시스템을 분석하면 I/O 서브시스템이 성능 문제의 원인인지에 대한 여부를 알 수 있습니다. **free**, **buff**, **cache** 칼럼도 눈여겨볼 만 합니다. **cache** 값이 **bo** 값과 함께 증가하고 이후에 **cache**가 감소하고 **free**가 증가하는 경우 시스템은 페이지 캐시의 비활성화 및 다시 쓰기를 실행하고 있음을 나타냅니다.

vmstat에 의해 보고되는 I/O 수는 모든 장치의 모든 I/O 합계임에 유의합니다. I/O 서브시스템에 성능 차이가 있을 수 있다고 판단되면 장치에 따라 I/O 보고를 분류하는 **iostat**로 문제를 보다 자세히 검사할 수 있습니다. 평균 요청 크기, 초 당 읽기 및 쓰기 수, 진행 중인 I/O 병합 양과 같은 보다 자세한 정보를 검색할 수 있습니다.

평균 요청 크기와 평균 큐 크기 (**avgqu-sz**)를 사용하여 스토리지 성능을 특성화할 때 생성한 그래프를 사용하여 스토리지 성능을 예상할 수 있습니다. 여기서 일부 일반화가 적용됩니다. 예를 들어, 평균 요청 크기가 4KB이고 평균 큐 크기가 1일 경우 처리량이 매우 고성능일 가능성은 낮습니다.

성능 수가 예상 성능에 매핑되지 않는 경우, **blktrace**를 사용하여 보다 정교한 분석을 실행할 수 있습니다. **blktrace** 유틸리티는 I/O 서브시스템에 소요되는 시간에 대한 자세한 정보를 제공합니다. I/O에서의 출력은 바이너리 추적 파일 모음으로 이는 **blkparse**와 같은 다른 유틸리티를 사용하여 사후 처리할 수 있습니다.

blkparse는 **blktrace**에 동료 유틸리티입니다. 이는 추적에서 원 출력을 읽고 간략한 텍스트 버전을 만듭니다.

다음은 **blktrace** 출력의 예입니다:

```
8,64 3 1 0.0000000000 4162 Q RM 73992 + 8 [fs_mark]
8,64 3 0 0.000012707 0 m N cfq4162S / allocated
8,64 3 2 0.000013433 4162 G RM 73992 + 8 [fs_mark]
8,64 3 3 0.000015813 4162 P N [fs_mark]
8,64 3 4 0.000017347 4162 I R 73992 + 8 [fs_mark]
8,64 3 0 0.000018632 0 m N cfq4162S / insert_request
8,64 3 0 0.000019655 0 m N cfq4162S / add_to_rr
8,64 3 0 0.000021945 0 m N cfq4162S / idle=0
8,64 3 5 0.000023460 4162 U N [fs_mark] 1
8,64 3 0 0.000025761 0 m N cfq workload slice:300
8,64 3 0 0.000027137 0 m N cfq4162S / set_active
wl_prio:0 wl_type:2
8,64 3 0 0.000028588 0 m N cfq4162S / fifo=(null)
8,64 3 0 0.000029468 0 m N cfq4162S / dispatch_insert
8,64 3 0 0.000031359 0 m N cfq4162S / dispatched a
request
8,64 3 0 0.000032306 0 m N cfq4162S / activate rq,
drv=1
8,64 3 6 0.000032735 4162 D R 73992 + 8 [fs_mark]
8,64 1 1 0.004276637 0 C R 73992 + 8 [0]
```

출력 예에서 볼 수 있듯이 출력은 밀집되어 있어 읽기 어렵습니다. 어떤 프로세스가 장치에 I/O 출력을 실행하고 있는지를 아는데 유용하지만 **blkparse**를 사용하면 추가 정보 요약이 이해하기 쉬운 형식으로 제공됩니다. **blkparse** 요약 정보는 출력의 맨 마지막에 인쇄됩니다:

Total (sde):

Reads Queued:	19,	76KiB	Writes Queued:	142,183,
568,732KiB				
Read Dispatches:	19,	76KiB	Write Dispatches:	25,440,
568,732KiB				
Reads Requeued:	0		Writes Requeued:	125
Reads Completed:	19,	76KiB	Writes Completed:	25,315,
568,732KiB				
Read Merges:	0,	0KiB	Write Merges:	116,868,
467,472KiB				
IO unplugs:	20,087		Timer unplugs:	0

요약에서는 평균 I/O 비율, 병합 활동을 보여주며 읽기 워크로드와 쓰기 워크로드를 비교합니다. 하지만 대부분의 경우 **blkparse** 출력은 출력 자체를 사용하는데 너무 볼륨이 큼니다. 다행히 데이터를 시각화하는데 몇 가지 유용한 도구가 있습니다.

btt는 I/O 스택의 다른 영역에서 I/O가 소요한 시간에 대한 분석을 제공합니다. 이러한 영역은 다음과 같습니다:

- Q — 블록 I/O가 대기열에 있음
- G — 요청 취득

새로 대기열이된 블록 I/O는 기존 요청과 병합하기 위한 후보가 아니기 때문에 새로운 블록 레이아웃 요청이 할당됩니다.

- M — 블록 I/O는 기존 요청과 병합됩니다.
- I — 요청이 장치 큐에 삽입됩니다.
- D — 요청이 장치에 발급됩니다.
- C — 요청은 드라이버에 의해 완료됩니다.
- P — 블록 장치 큐가 연결되어 요청 집계를 허용합니다.
- U — 장치 큐가 연결 해제되어 장치에 발급된 요청 집계를 허용합니다.

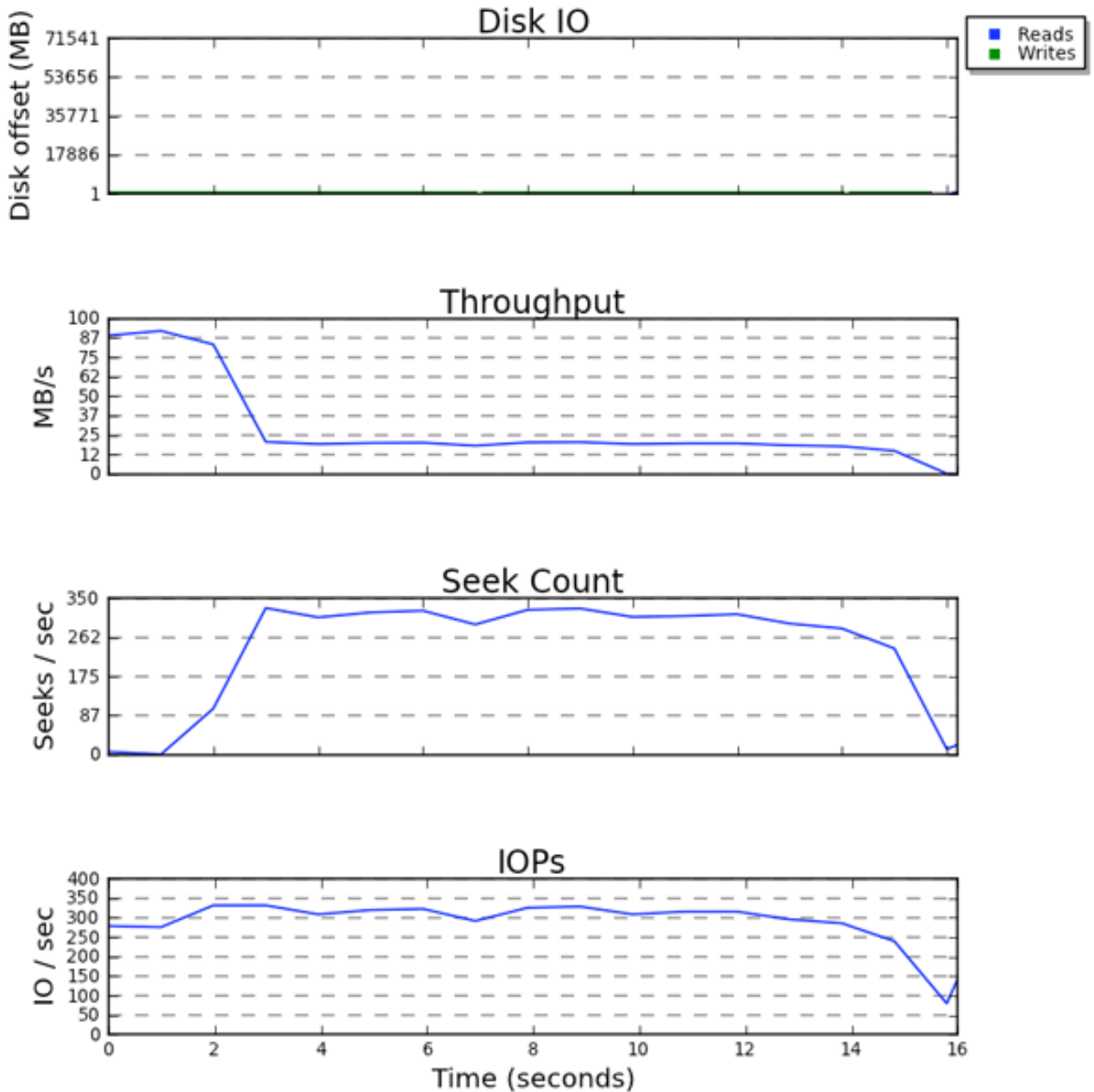
btt는 각 영역간 이동에 소요된 시간 뿐 만 아니라 다른 영역에서 소요된 시간도 분류합니다:

- Q2Q — 블록 층에 전송된 요청 사이의 시간
- Q2G — 블록 I/O가 대기 상태에서 요청이 할당될 때까지의 시간
- G2I — 요청이 할당되는 시간 부터 장치의 큐에 넣을 때까지의 시간
- Q2M — 블록 I/O가 대기열에 있는 시간 부터 기존 요청과 병합될 때까지의 시간
- I2D — 요청이 장치 큐에 삽입된 시간 부터 실제로 장치에 발급될 때까지의 시간
- M2D — 블록 I/O가 기존 요청과 병합된 시간 부터 요청이 장치 발급될 때까지의 시간
- D2C — 장치에서 요청된 서비스 시간
- Q2C — 요청에 대해 블록 층에서 총 소요된 시간

위의 표에서 워크로드에 대해 많은 것을 추론할 수 있습니다. 예를 들어, Q2Q가 Q2C 보다 훨씬 클 경우 이는 애플리케이션이 빠르게 연속으로 I/O를 실행하지 않는다는 것을 의미합니다. 즉, 성능 문제는 I/O 서

브시시스템과 전혀 관련되지 않을 수 있습니다. D2C가 매우 높은 경우 장치가 요청을 수행하는데 오랜 시간이 소요되는 것입니다. 이는 장치가 단순히 오버로드되고 있거나 (공유 리소스이기 때문일 수 있음) 또는 장치에 전송된 워크로드가 최적의 것이 아님을 의미하는 것일 수 있습니다. Q2G가 매우 높은 경우 동시에 많은 요청이 대기열에 있음을 의미합니다. 즉 스토리지가 I/O 로드를 처리할 수 없음을 의미합니다.

마지막으로 **seekwatcher**는 **blktrace** 바이너리 데이터를 소비하고 논리 블록 주소 (LBA), 처리 속도, 초당 탐색 횟수, 초당 I/O (IOPS)를 포함하는 플롯 세트를 생성합니다.



Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
252.57	31.72	305.99	16.21

그림 6.2. seekwatcher 출력의 예

모든 플롯은 X 축으로 시간을 사용합니다. LBA 플롯은 읽기 및 쓰기가 다른 색으로 표시되어 있습니다. 처리량과 초당 탐색 횟수 그래프의 관계를 주목해 볼 만 합니다. 탐색 구분 스토리지의 경우 이 두 플롯 사이에 반비례 관계가 존재합니다. IOPS 그래프는 장치에서 예상 처리량을 얻을 수 없지만 IOPS 제한에 도달한 경우에 유용합니다.

6.4. 설정

먼저 결정해야 할 사항중 하나는 어떤 I/O 스케줄러를 사용하는가 하는 것입니다. 다음 부분에서는 워크로드에 가장 적합한 스케줄러를 결정할 수 있도록 각각의 주요 스케줄러에 대한 개요를 제공합니다.

6.4.1. CFQ (Completely Fair Queuing)

CFQ는 I/O를 시작한 프로세스에 따라 I/O 스케줄링 결정에 공정성을 제공합니다. 실시간 (RT), 최선형 (BE), 유휴라는 세 가지 다른 스케줄링 클래스가 있습니다. 스케줄링 클래스는 **ionice** 명령을 사용하여 수동으로 할당하거나 **ioprio_set** 시스템 호출을 통해 프로그램에 할당할 수 있습니다. 기본값으로 프로세스는 최선형 스케줄링 클래스에 배치됩니다. 실시간 및 최선형 스케줄링 클래스는 각 클래스 내에서 8 개의 I/O 우선 순위로 나뉘며 우선 순위 0은 가장 높은것이고 7은 가장 낮은것이 됩니다. 실시간 스케줄링 클래스에 있는 프로세스는 최선형이나 유휴 상태에 있는 프로세스 보다 적극적으로 스케줄되기 때문에 스케줄된 실시간 I/O는 항상 최선형이나 유휴 I/O 보다 먼저 실행됩니다. 즉 실시간 우선 순위의 I/O는 최선형과 유휴 클래스 모두를 실행할 필요 없게 할 수 있습니다. 최선형 스케줄링은 기본값 스케줄링 클래스로 이 클래스에 있는 기본값 우선 순위는 4입니다. 유휴 스케줄링 클래스에 있는 프로세스는 시스템에 실행되지 않는 다른 I/O가 없는 경우에만 실행됩니다. 즉 프로세스에서 I/O가 프로세스 진행에 전혀 필요하지 않은 경우에만 프로세스의 I/O 스케줄링 클래스를 유휴로 설정하는 것은 매우 중요합니다.

CFQ는 I/O를 실행하고 있는 각 프로세스에 타임 슬라이스를 할당하여 공정성을 제공합니다. 타임 슬라이스 동안 프로세스는 (기본값으로) 공중에서 최대 8 개의 요청을 동시에 받을 수 있습니다. 스케줄러는 기록 데이터를 기반으로 애플리케이션이 조만간 더 많은 I/O를 실행할 지에 대한 여부를 예측 시도합니다. 프로세스가 더 많은 I/O를 실행할 것으로 예상되면 다른 프로세스에서 I/O가 실행을 기다리고 있는 상태에서도 CFQ는 유휴 상태가 되어 I/O를 기다립니다.

CFQ에 의해 수행되는 유휴 상태로 인해 빠른 외장 스토리지 배열이나 솔리드 스테이트 디스크 등과 같은 대형 탐색 패널티에서 손상을 받지 않는 하드웨어에 적합하지 않을 경우가 많습니다. 이러한 스토리지에서 CFQ의 사용이 필요한 경우 (예: **cgroup** 비례 가중 I/O 스케줄러도 사용하고자 하는 경우), CFQ 성능을 개선하기 위해 일부 설정을 튜닝해야 합니다. **/sys/block/device/queue/iosched/**에 있는 동일한 이름의 파일에서 다음과 같은 매개 변수를 설정합니다:

```
slice_idle = 0
quantum = 64
group_idle = 1
```

group_idle이 1로 설정된 경우에도 I/O 스톨 (백엔드 스토리지는 유휴 상태로 인해 사용 중이지 않음)의 가능성은 여전히 남아 있습니다. 하지만 이러한 스톨의 빈도는 시스템에 있는 각 큐의 유휴 상태보다 적습니다.

CFQ는 비작업 보존형 I/O 스케줄러로 이는 (위에서 설명한 대로) 보류 중인 요청이 있을 때에도 유휴 상태가 될 수 있다는 것을 의미합니다. 비작업 보존 스케줄러의 스택은 I/O 경로에 상당한 대기 시간을 도입할 수 있습니다. 이러한 스택의 예에는 호스트 기반 하드웨어 RAID 컨트롤러의 상단에 CFQ를 사용하는 것입니다. RAID 컨트롤러는 자신의 비작업 보존 스케줄러를 구현할 수 있습니다. 즉 이는 스택에서 두 가지 수준의 지연을 일으키는 원인이 될 수 있습니다. 비작업 보존 스케줄러는 결정의 기반이 되는 가능한 많은 데이터가 있을 때 가장 최선으로 작동합니다. 스케줄링 알고리즘과 같은 스택의 경우 가장 하단의 스케줄러는 상단의 스케줄러가 하단으로 보내는 것만 볼 수 있습니다. 따라서 하위 계층은 실제 워크로드와는 동떨어진 I/O 패턴을 보게 됩니다.

조정 가능한 매개 변수

back_seek_max

뒤로 검색은 헤드의 위치 변경에 있어 앞으로 검색보다 상당한 지연을 발생시킬 수 있으므로 일반적으로 성능이 나빠집니다. 하지만 뒤로 검색 크기가 크지 않은 경우 CFQ가 여전히 이를 실행합니다. 이러한 조정 가능한 매개 변수는 I/O 스케줄러가 뒤로 검색을 허용하는 최대 거리를 KB 단위로 제어합니다. 기본값은 **16 KB**입니다.

back_seek_penalty

뒤로 검색의 비효율성으로 인해 각각에 패널티가 연결됩니다. 패널티는 승수로 예를 들어 1024KB의 디스크 헤드 위치를 생각해 봅시다. 큐에 두 개의 요청이 있다고 가정합니다. 하나는 1008KB이고 다른 하나는 1040KB입니다. 이러한 두 요청은 현재 헤드 위치에서 등거리에 있습니다. 하지만 뒤로 검색 패널티 (기본값: 2)를 적용하면 디스크에서 나중 위치에 있는 요청은 이전 요청의 위치보다 2배 가까운 거리에 있게 됩니다. 따라서 헤드가 앞으로 이동합니다.

fifo_expire_async

이러한 조정 가능한 매개 변수는 비동기 (버퍼링된 쓰기) 요청이 실행되지 않는 시간을 제어합니다. 만료 후 (밀리초 단위) 단일 비동기 요청은 디스패치 목록으로 이동합니다. 기본값은 **250** 밀리초입니다.

fifo_expire_sync

이는 동기화 (읽기 및 O_DIRECT 쓰기) 요청의 `fifo_expire_async` 조정 가능한 매개 변수와 같습니다. 기본값은 **125** 밀리초입니다.

group_idle

이것이 설정되면 CFQ는 cgroup에서 I/O를 실행하는 마지막 프로세스에서 유휴 상태가 됩니다. 비례 가중 I/O cgroup을 사용할 때 이를 **1**로 설정하고 `slice_idle`을 **0**으로 (일반적으로 고속 스토리지에서 실행) 설정해야 합니다.

group_isolation

그룹 분리(group isolation)가 활성화되어 있는 경우 (**1**로 설정) 이는 처리량을 대가로하여 그룹 간의 분리를 강화합니다. 일반적으로 그룹 분리가 비활성화되어 있는 경우 공정성은 연속 워크로드에만 제공 됩니다. 그룹 분리를 활성화하면 연속 및 임의의 워크로드 모두에 공정성을 제공합니다. 기본값은 **0** (비활성화)입니다. 보다 자세한 내용은 **Documentation/cgroups/blkio-controller.txt**에서 참조하십시오.

low_latency

낮은 대기 시간 (low latency)이 활성화되어 있는 경우 (**1**로 설정) CFQ는 장치에서 I/O를 실행하는 각 프로세스에 대해 최대 300 밀리초의 대기 시간을 제공하려 합니다. 이는 처리량 보다 공정성을 우선으로 합니다. 낮은 대기 시간을 비활성화하면 (**0**으로 설정) 대상 대기 시간이 무시되고 시스템에 있는 각 프로세스는 전체 타임 슬라이스를 가져옵니다. 낮은 대기 시간은 기본값으로 활성화되어 있습니다.

quantum

quantum은 CFQ가 한번에 스토리지에 전송하는 I/O 수를 제어하고 실질적으로 장치 큐 깊이를 제한합니다. 기본값으로 이는 **8**로 설정되어 있습니다. 스토리지는 더 깊은 큐 깊이를 지원할 수 있지만 **quantum**을 증가시키면 대기 시간에 부정적인 영향을 미치고 특히 대량의 연속 쓰기 워크로드가 있을 경우 더욱 그러합니다.

slice_async

이러한 조정 가능한 매개 변수는 비동기 (버퍼링된 쓰기) I/O를 실행하는 각 프로세스에 할당된 타임 슬라이스를 제어합니다. 기본값으로 이는 **40** 밀리초로 설정되어 있습니다.

slice_idle

이는 추가 요청을 기다리는 동안 CFQ가 유힬 상태가 될 시간을 지정합니다. Red Hat Enterprise Linux 6.1 및 이전 버전에서 기본값은 **8** 밀리초입니다. Red Hat Enterprise Linux 6.2 및 이후 버전에서 기본값은 **0**입니다. 값이 0일 경우 큐와 서비스 트리 레벨에 있는 모든 유힬 상태를 제거하므로 외부 RAID 스토리지 처리량이 향상됩니다. 하지만 이는 전체 검색 수를 증가시키기 때문에 내부의 비 RAID 스토리지 처리량을 저하시킬 수 있습니다. 비 RAID 스토리지의 경우 **slice_idle** 값을 0 보다 크게 할 것을 권장합니다.

slice_sync

이 매개 변수는 동기화 (읽기 또는 직접 쓰기) I/O를 실행하는 프로세스에 할당된 타임 슬라이스를 규정합니다. 기본값은 **100** 밀리초입니다.

6.4.2. 데드라인 I/O 스케줄러

데드라인 I/O 스케줄러는 요청에 대해 약속된 대기 시간을 제공하려 합니다. 요청이 I/O 스케줄러에 도달해야만 대기 시간 측정이 시작됨에 유의합니다 (애플리케이션이 요구 디스크립터가 해제될 때 까지 대기하며 수면 상태로 될 수 있으므로 이는 중요한 차이입니다). 기본값으로 쓰기 보다 읽기가 우선합니다. 이는 애플리케이션이 읽기 I/O를 차단할 가능성이 더 높기 때문입니다.

데드라인은 배치로 I/O를 보냅니다. 배치는 LBA 오름 차순으로 (편도 엘리베이터) 증가하는 읽기 또는 쓰기 I/O의 연속적 순서입니다. 각 배치를 처리한 후 I/O 스케줄러는 쓰기 요청이 장시간 사용되지 않았는지를 확인하고 읽기 또는 쓰기의 새로운 배치를 시작할 지에 대한 여부를 결정합니다. 요청 FIFO 목록은 각 배치를 시작할 때 만료된 요청이 있는지를 확인하고 그 후에 배치의 데이터 방향만을 확인합니다. 따라서 쓰기 배치가 선택되고 만료된 요청이 있을 경우 해당 읽기 요청은 쓰기 배치가 완료될 때 까지 실행되지 않습니다.

조정 가능한 매개 변수

fifo_batch

이는 단일 배치에서 실행할 읽기 또는 쓰기 수를 지정합니다. 기본값은 **16**입니다. 이를 높은 값으로 설정하면 처리량은 향상되지만 대기 시간은 길어질 수 있습니다.

front_merges

워크로드가 전면 병합을 생성하지 않는 경우 이 매개 변수를 **0**으로 설정할 수 있습니다. 이 체크의 오버헤드를 측정하지 않는 한 기본 설정 (**1**) 그대로 두는 것이 좋습니다.

read_expire

이 매개 변수로 읽기 요청이 실행되는 시간을 밀리초 단위로 설정할 수 있습니다. 기본값은 **500** 밀리초 (0.5초)로 설정되어 있습니다.

write_expire

이 매개 변수로 쓰기 요청이 실행되는 시간을 밀리초 단위로 설정할 수 있습니다. 기본값은 **5000** 밀리초 (5초)로 설정되어 있습니다.

writes_starved

이 매개 변수는 단일 쓰기 배치 처리 전 처리할 수 있는 읽기 배치 수를 제어합니다. 이 값을 높게 설정할 수록 읽기가 우선이 됩니다.

6.4.3. Noop

Noop I/O 스케줄러는 간단한 FIFO (first-in first-out) 스케줄링 알고리즘을 구현합니다. 요청 병합은 일반 블록 계층에서 발생하지만 이는 마지막 적중 항목 캐시입니다. 시스템이 CPU에 바인딩되어 있고 스토리지가 빠른 경우 이는 최상의 I/O 스케줄러가 됩니다.

블록 계층에서 사용할 수 있는 튜닝 가능한 매개 변수는 다음과 같습니다.

/sys/block/sdX/queue의 튜닝 가능한 매개 변수

add_random

일부 경우 **/dev/random**에 대한 엔트로피 풀에 기여하고 있는 I/O 이벤트의 오버헤드는 측정 가능합니다. 이러한 경우 이 값을 0로 설정하는 것이 좋습니다.

max_sectors_kb

기본값으로 디스크에 전송되는 최대 요청 크기는 **512 KB**입니다. 이 매개 변수는 값을 늘리거나 줄일 수 있습니다. 최소 값은 논리 블록 크기에 의해 제한되고 최대 값은 **max_hw_sectors_kb**에 의해 제한됩니다. I/O 크기가 내부 소거 블록 크기를 초과하면 성능이 악화되는 일부 SSD가 있습니다. 이러한 경우 **max_hw_sectors_kb**를 소거 블록 크기로 줄이는 것이 좋습니다. **iozone** 또는 **aio-stress**와 같은 I/O 생성기를 사용하여 레코드 크기를 **512** 바이트에서 **1 MB**로 변경하는 것과 같이 이를 테스트할 수 있습니다.

nomerges

이 매개 변수는 주로 디버깅에 도움이 됩니다. 대부분의 워크로드에는 요청 병합 (SSD와 같은 고속 스토리지에서도)에서 혜택을 받을 수 있습니다. 하지만 일부 경우 미리 읽기 또는 임의 I/O를 수행하지 않고 스토리지 백엔드에서 처리할 수 있는 IOPS 수를 알고자 하는 경우와 같이 병합을 비활성화하는 것이 좋은 경우도 있습니다.

nr_requests

각 요청 큐에는 읽기 및 쓰기 I/O 마다 할당할 수 있는 총 요청 디스크립터 수에 제한이 있습니다. 기본 값은 **128**이며 이는 프로세스를 수면 상태로 두기 전 한 번에 **128** 개의 읽기 및 **128** 개의 쓰기를 대기열에 둘 수 있음을 의미합니다. 수면 상태에 있는 프로세스는 요청 할당을 시도하는 다음 것으로 사용 가능한 모든 요청을 할당한 프로세스일 필요는 없습니다.

대기 시간에 민감한 애플리케이션이 있는 경우, 요청 큐에서 **nr_requests**의 값을 낮추고 스토리지의 명령큐 깊이를 낮은 (**1** 정도로 낮은) 수치로 제한하는 것이 좋습니다. 이렇게 하면 다시 쓰기 I/O는 사용 가능한 모든 요청 디스크립터를 할당할 수 없으며 쓰기 I/O로 장치 큐를 채울 수 없습니다.

nr_requests가 할당되면 I/O를 실행하는 다른 모든 프로세스가 요청을 사용할 수 있을 때 까지 대기하기 위해 수면 상태로 됩니다. 이로 인해 요청을 (하나의 프로세스가 빠르게 순차적으로 모두 소비하는 것이 아니라) 라운드 로빈 방식으로 분배하기 때문에 공정성이 유지됩니다. 이는 기본 **CFQ** 설정이 이러한 상황에 대해 보호하려는 것으로 데드라인 또는 **noop** 스케줄러 사용시에만 문제가 되는 점에 유의합니다.

optimal_io_size

상황에 따라 기초적인 스토리지가 최적의 I/O 크기를 보고합니다. 이는 최적의 I/O 크기가 스트라이프 크기로 되는 하드웨어 및 소프트웨어 **RAID**에서 가장 일반적인 것입니다. 값이 보고되면 애플리케이션은 가능한 최적의 I/O 크기로 조정되어 그 배수의 I/O를 실행합니다.

read_ahead_kb

운영 체제는 애플리케이션이 파일 또는 디스크에서 데이터를 순차적으로 읽을 때 이를 감지할 수 있습니다. 이러한 경우는 지능형 미리 읽기 알고리즘을 실행하여 사용자가 요청한 데이터보다 더 많은 데이터를 디스크에서 읽습니다. 즉 사용자가 다음 데이터 블록을 읽으려고 할 때 해당 데이터가 이미 운영

체제의 페이지 캐시에 있는 것입니다. 이것의 가능한 단점은 운영 체제가 디스크에서 필요 이상의 데이터를 읽을 수 있다는 것입니다. 이로 인해 메모리 압력이 높아져 데이터가 삭제될 때 까지 데이터는 페이지 캐시의 공간을 차지하게 됩니다. 이러한 상황에서 여러 프로세스가 미리 읽기를 잘못 실행할 경우 메모리 압력이 높아질 수 있습니다.

장치 매퍼 장치의 경우 **`read_ahead_kb`** 값을 **8192**와 같이 큰 값으로 하는 것이 좋습니다. 이는 장치 매퍼 장치가 여러 기초적인 장치로 구성되어 있기 때문입니다. 이 값을 매핑하려는 장치의 수를 곱한 기본값 (**128 KB**)으로 설정하면 튜닝을 위한 좋은 출발점이 됩니다.

rotational

기존 하드 디스크는 회전 (회전 원반으로 구성)했었지만 SSD는 다릅니다. 대부분의 SSD는 이를 제대로 알리지만 이러한 플래그를 정확하게 알리지 않는 장치의 경우 회전을 **0**으로 수동 설정해야 합니다. 회전이 비활성화된 경우 비회전 미디어에서 검색 작업에 약간의 패널티가 있기 때문에 I/O 엘리베이터는 검색을 줄이기 위해 논리를 사용하지 않습니다.

rq_affinity

I/O 완료는 I/O를 실행한 CPU와 다른 CPU에서 처리할 수 있습니다. **`rq_affinity`**를 **1**로 설정하면 커널은 I/O가 실행된 CPU에 완료를 전달합니다. 이는 CPU 데이터 캐싱 효과를 개선할 수 있습니다.

7장. 파일 시스템

다음 부분에서는 Red Hat Enterprise Linux에서 지원되는 파일 시스템 개요와 성능을 최적화하는 방법에 대해 설명합니다.

7.1. 파일 시스템 튜닝 시 고려 사항

모든 파일 시스템에서 일반적으로 튜닝시 고려해야 할 사항이 몇 가지 있습니다. 시스템에서 선택한 포맷 및 마운트 옵션과 특정 시스템에서 성능을 향상시킬 수 있는 애플리케이션에서 사용 가능한 작업 등입니다.

7.1.1. 포맷 옵션

파일 시스템 블록 크기

블록 크기는 **mkfs** 실행 시 선택할 수 있습니다. 유효한 크기 범위는 시스템에 따라 다릅니다: 상한 범위는 호스트 시스템의 최대 페이지 크기에서 하한 범위는 사용되는 파일 시스템에 의존합니다. 기본값 블록 크기는 대부분의 사용 경우에 적합하도록 되어 있습니다.

기본 블록 크기보다 작은 파일을 여러개 생성하려는 경우 블록 크기를 작게 설정하여 디스크 공간 낭비를 최소화할 수 있습니다. 하지만 블록 크기를 작게 설정하면 파일 시스템의 최대 크기를 제한하게 되고 특히 선택된 블록 크기보다 파일이 큰 경우 추가적인 런타임 오버헤드를 일으킬 수 있습니다.

파일 시스템 지오메트리

시스템이 RAID5와 같은 스트라이프 스토리지를 사용하는 경우 **mkfs** 실행 시 기본적인 스토리지 지오메트리 사용으로 데이터 및 메타데이터를 정리하여 성능을 향상시킬 수 있습니다. 소프트웨어 RAID (LVM 또는 MD) 및 일부 엔터프라이즈 하드웨어 스토리지 일부의 경우 이 정보는 쿼리되어 자동으로 설정되지만 대부분의 경우 관리자가 명령행에서 **mkfs**를 사용하여 수동으로 지오메트리를 지정해야 합니다.

이러한 파일 시스템의 생성 및 유지 관리에 대한 보다 자세한 내용은 *스토리지 관리 가이드*에서 참조하십시오.

외부 저널

메타데이터 집약적 워크로드는 저널링 파일 시스템 (ext4 및 XFS 등)의 로그 섹션이 매우 자주 업데이트되는 것을 의미합니다. 파일 시스템에서 저널로의 탐색 시간을 최소화하려면 전용 스토리지에 저널을 배치합니다. 하지만 주요 파일 시스템보다 느리게 외부 스토리지에 저널을 배치하면 외부 스토리지 사용과 관련하여 얻을 수 있는 잠재적 이익을 무효화할 수 있다는 점에 유의합니다.



주의

외부 저널이 신뢰할 수 있는지 확인합니다. 외부 저널 장치가 손실되면 파일 시스템 손상의 원인이 됩니다.

외부 저널은 마운트 시에 지정된 저널 장치로 **mkfs** 실행 시 생성됩니다. 자세한 내용은 **mke2fs(8)**, **mkfs.xfs(8)**, 및 **mount(8) man** 페이지에서 참조하십시오.

7.1.2. 마운트 옵션

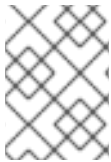
장애

쓰기 장애는 심하게 변동하는 쓰기 장애가 전원을 상실해도 해당 파일 시스템의 메타데이터가 올바르게 작성되어 영구적인 스토리지에 정렬되어 있는지 확인하는데 사용되는 커널 메커니즘입니다. 쓰기 장애를 갖는 파일 시스템은 **fsync()**를 통해 전송된 데이터가 정전 중에도 영구적으로 지속하게 합니다. Red Hat Enterprise Linux는 이를 지원하는 모든 하드웨어에 기본값으로 장애를 활성화합니다.

하지만 쓰기 장애를 사용하면 일부 애플리케이션 속도가 현저하게 느려집니다. 특히 **fsync()**를 많이 사용하는 애플리케이션이나 여러 작은 파일을 생성 및 삭제하는 애플리케이션의 경우 그러합니다. 심하게 변동하는 쓰기 캐시가 없는 스토리지가든 드물게는 정전 이후 파일 시스템 불이치 또는 데이터 손실이 허용되는 경우 **nobarrier** 마운트 옵션을 사용하여 장애를 비활성화할 수 있습니다. 보다 자세한 내용은 *스토리지 관리 가이드*에서 참조하십시오.

액세스 타임 (noatime)

기존에는 파일을 읽을 때 파일의 액세스 시간 (**atime**)은 추가 쓰기 I/O가 포함된 inode 메타데이터에 업데이트해야 했습니다. 정확한 **atime** 메타데이터가 필요하지 않은 경우 **noatime** 옵션으로 파일 시스템을 마운트하여 이러한 메타데이터 업데이트를 제거합니다. 하지만 대부분의 경우 Red Hat Enterprise Linux 6 커널에서 기본값 관련 **atime** (또는 **relatime**) 동작으로 인해 대량의 **atime** 오버헤드가 없습니다. **relatime** 동작은 이전 **atime**이 수정 시간 (**mtime**) 또는 상태 변경 시간 (**ctime**) 보다 오래된 경우에만 **atime**을 업데이트합니다.



참고

noatime 옵션을 활성화하면 **nodiratime** 동작도 활성화됩니다; **noatime**과 **nodiratime** 모두를 설정할 필요는 없습니다.

미리 읽기 지원 강화

미리 읽기는 데이터를 먼저 가져와 이를 페이지 캐시에 로딩하여 디스크에서가 아닌 메모리에서 먼저 사용가능하므로 파일 액세스 속도를 빠르게 합니다. 대량의 연속 I/O 스트리밍 작업과 같은 일부 워크로드는 미리 읽기 값이 높으면 효과를 얻습니다.

tuned 도구와 LVM 스트라이핑의 사용은 미리 읽기 값을 증가시키지만 일부 워크로드의 경우 항상 충분한 것은 아닙니다. 또한 Red Hat Enterprise Linux는 파일 시스템이 감지할 수 있는 내용에 따라 항상 적절한 미리 읽기 값을 설정할 수 없습니다. 예를 들어 강력한 스토리지 배열이 하나의 강력한 LUN으로 Red Hat Enterprise Linux로 제시되면 운영 체제는 이를 강력한 LUN 배열로 취급하지 않으며 따라서 기본값으로 스토리지에서 사용할 수 있는 잠재적 미리 읽기의 장점을 최대한 활용할 수 없는 것입니다.

blockdev 명령을 사용하여 미리 읽기 값을 표시하나 편집합니다. 특정 블록 장치에 대한 현재의 미리 읽기 값을 표시하려면 다음 명령을 실행합니다:

```
# blockdev -getra device
```

블록 장치의 미리 읽기 값을 편집하려면 다음과 같은 명령을 실행합니다. **N**은 512 바이트 섹터의 번호를 나타냅니다.

```
# blockdev -setra N device
```

blockdev 명령으로 선택된 값은 재부팅 후에도 유지되지 않는다는 점에 유의합니다. 부팅 시 이 값을 설정하려면 런레벨 **init.d** 스크립트를 작성하는 것이 좋습니다.

7.1.3. 파일 시스템 유지 보수

사용되지 않는 블록 삭제

배치 폐기와 온라인 폐기 작업은 파일 시스템에 의해 사용되지 않는 블록을 폐기하는 마운트된 파일 시스템의 기능입니다. 이러한 작업은 솔리드 스테이트 드라이브 및 썬 프로비저닝 스토리지 모두에 유용합니다.

배치 폐기 작업은 **fstrim** 명령으로 사용자에게 의해 명시적으로 실행됩니다. 이 명령은 사용자의 기준과 일치하는 파일 시스템에서 사용하지 않는 모든 블록을 폐기합니다. 두 작업 유형은 파일 시스템의 기반이 되는 블록 장치가 물리적 폐기 작업을 지원하는 한 Red Hat Enterprise Linux 6.2 및 이후 버전에서 XFS 및 ext4 파일 시스템과 사용 가능합니다. 물리적 폐기 작업은 `/sys/block/device/queue/discard_max_bytes` 값이 제로가 아닌 경우 지원됩니다.

온라인 폐기 작업은 마운트시 **-o discard** 옵션 (`/etc/fstab`에서 또는 **mount** 명령의 부분으로)으로 지정되어 사용자 개입없이 실시간으로 실행됩니다. 온라인 폐기 작업은 사용됨에서 사용 해제로 전환되는 블록만을 폐기합니다. 온라인 폐기 작업은 Red Hat Enterprise Linux 6.2 이상 버전에서는 ext4 파일 시스템에서 Red Hat Enterprise Linux 6.4 이상 버전에서는 XFS 파일 시스템에서 지원됩니다.

Red Hat은 시스템의 워크로드가 배치 폐기를 실행할 수 없는 경우 또는 성능 유지를 위해 온라인 폐기 작업이 필요한 경우를 제외하고 배치 폐기 운영을 권장하고 있습니다.

7.1.4. 애플리케이션 유의 사항

사전 할당

ext4, XFS, GFS2 파일 시스템은 **fallocate(2)** glibc 호출을 통해 효율적인 공간의 사전 할당을 지원합니다. 쓰기 패턴으로 인해 파일이 잘못 조각화되어 읽기 성능이 저하될 경우 공간의 사전 할당은 유용한 방법이 될 수 있습니다. 사전 할당은 데이터를 공간에 작성하지 않고 디스크 공간이 파일에 할당된 것처럼 표시합니다. 사전 할당 블록에 실제 데이터가 기록될 때 까지 읽기 작업은 제로를 반환합니다.

7.2. 파일 시스템 성능 프로파일

tuned-adm 도구를 사용하면 사용자는 특정 사용자 경우에 대해 성능 개선을 위해 고안된 여러 프로파일 간의 스위치를 용이하게 합니다. 스토리지 성능 개선에 특히 유용한 프로파일은 다음과 같습니다:

latency-performance

전형적인 지연 성능 튜닝 용 서버 프로파일입니다. 이는 **tuned** 및 **ktune** 절전 메커니즘을 비활성화합니다. **cpuspeed** 모드는 **performance**로 변경됩니다. 각 장치의 I/O 엘리베이터는 **deadline**으로 변경됩니다. **cpu_dma_latency** 매개 변수는 전원 관리 서비스에 대해 **0** 값 (최소 대기 시간)으로 등록되어 가능한 범위에서 전원 관리 대기 시간을 제한합니다.

throughput-performance

전형적인 처리량 성능 튜닝 용 서버 프로파일입니다. 시스템에 엔터프라이즈급 스토리지가 없는 경우가 프로파일이 권장됩니다. 이는 **latency-performance**와 동일하지만 다음과 같은 차이점이 있습니다:

- **kernel.sched_min_granularity_ns** (scheduler minimal preemption granularity)는 **10** 밀리초로 설정됩니다.
- **kernel.sched_wakeup_granularity_ns** (scheduler wake-up granularity)는 **15** 밀리초로 설정됩니다.
- **vm.dirty_ratio** (virtual machine dirty ratio)는 40%로 설정됩니다.
- transparent huge pages가 활성화됩니다.

enterprise-storage

이 프로파일은 배터리 백업 컨트롤러 캐시 보호 및 디스크 내장 캐시 관리 등 엔터프라이즈급 스토리지로 엔터프라이즈 크기 서버 설정에 권장됩니다. 이는 **throughput-performance** 프로파일과 동일하지만 다음과 같은 차이점이 있습니다:

- **readahead** 값은 **4x**로 설정됩니다.
- root/boot 파일 시스템 이외의 파일 시스템은 **barrier=0**으로 다시 마운트됩니다.

tuned-adm에 대한 보다 자세한 내용은 man 페이지 (**man tuned-adm**) 또는 *전원 관리 가이드* (http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/)에서 참조하십시오.

7.3. 파일 시스템

7.3.1. Ext4 파일 시스템

ext4 파일 시스템은 Red Hat Enterprise Linux 5에서 사용 가능한 기본 ext3 파일 시스템의 확장된 버전입니다. 현재 Ext4는 Red Hat Enterprise Linux 6의 기본 파일 시스템으로 단일 파일 및 파일 시스템 모두에서 최대 16 TB 크기 까지 지원합니다. 또한 ext3 파일 시스템에서 32000 까지의 서브 디렉토리 제한을 제거합니다.



참고

16TB 보다 큰 파일 시스템의 경우 XFS와 같은 확장 가능한 고가용성 파일 시스템을 사용할 것을 권장합니다. 보다 자세한 내용은 7.3.2절. “XFS 파일 시스템”에서 참조하십시오.

ext4 파일 시스템의 기본값은 대부분 워크로드에 대해 최적화되어 있지만 파일 시스템 동작이 성능에 영향을 미치고 있는 것으로 성능 분석에 나타날 경우 다음과 같은 튜닝 옵션을 사용할 수 있습니다:

Inode 테이블 초기화

대형 파일 시스템에서 **mkfs.ext4** 프로세스는 파일 시스템에 있는 모든 inode 테이블을 초기화하는데 시간이 오래 걸릴 수 있습니다. 이 프로세스는 **-E lazy_itable_init=1** 옵션으로 미룰 수 있습니다. 이 옵션을 사용하면 커널 프로세스는 파일 시스템을 마운트한 후 초기화를 계속 진행하게 됩니다. 이러한 초기화 발생 비율은 **mount** 명령의 **-o init_itable=n** 옵션으로 제어할 수 있습니다. 여기서 백그라운드 초기화 실행에 소요되는 시간은 약 1/n입니다. n의 기본값은 10입니다.

Auto-fsync 동작

일부 애플리케이션은 기존 파일 이름을 바꾸거나 잘라내기 및 재작성 후 **fsync()**가 항상 제대로 작동한다고 할 수 없기 때문에 이름 바꾸기 (rename)를 통한 재배치 (replace) 작업 및 잘라내기 (truncate)를 통한 재배치 (replace) 작업 이후에 ext4는 파일의 자동 동기화를 기본값으로 설정합니다. 이 동작은 이전의 ext3 파일 시스템 동작과 일치합니다. 하지만 **fsync()** 작업은 시간이 걸리는 작업이므로 자동 동작이 필요없는 경우 **mount** 명령과 함께 **-o noauto_da_alloc** 옵션을 사용하여 이를 비활성화합니다. 즉 애플리케이션은 명시적으로 **fsync()**를 사용하여 데이터 일관성을 유지해야 합니다.

저널 I/O 우선 순위

기본값으로 저널 커밋 I/O는 일반적인 I/O 보다 약간 더 높은 우선 순위가 부여되어 있습니다. 이러한 우선 순위는 **mount** 명령의 **journal_ioprio=n** 옵션을 사용하여 제어할 수 있습니다. 기본값은 3입니다. 유효한 값의 범위는 0에서 7이며 0은 가장 높은 I/O 우선 순위입니다.

다른 **mkfs** 및 튜닝 옵션의 경우 **mkfs.ext4(8)** 및 **mount(8)** man 페이지 그리고 kernel-doc 패키지에 있는 **Documentation/filesystems/ext4.txt** 파일을 참조하십시오.

7.3.2. XFS 파일 시스템

XFS는 강력하고 확장성 높은 단일 호스트 64 비트 저널링 파일 시스템입니다. 이는 완전한 익스텐트 기반으로 매우 큰 파일 및 파일 시스템을 지원합니다. XFS 시스템이 저장할 수 있는 파일 수는 파일 시스템에서 사용 가능한 공간에 의해서만 제한됩니다.

XFS는 빠른 복구를 용이하게 하는 메타데이터 저널링을 지원합니다. XFS 파일 시스템은 마운트되어 활성화된 상태로 조각 모음 및 확장할 수 있습니다. 또한 Red Hat Enterprise Linux 6는 XFS에 특정 유틸리티의 백업 및 복원을 지원합니다.

XFS는 익스텐트 기반 할당을 사용하여 지연 할당 및 명시적인 사전 할당과 같은 여러 할당 체계를 가지고 있습니다. 익스텐트 기반 할당은 파일 시스템에서 사용된 공간을 추적하는 것보다 간결하고 효율적인 방법을 제공하여 메타데이터에 의해 소비되는 공간 및 조각화를 줄임으로써 대용량 파일의 성능을 향상시킵니다. 지연 할당은 파일이 연속적인 블록 그룹에 기록될 가능성을 높이는 것으로 단편화를 줄이고 성능을 향상시킵니다. 사전 할당은 애플리케이션이 사전에 기록해야 할 데이터 양을 알고 있는 경우 완전히 조각화하는 것을 방지하는데 사용될 수 있습니다.

XFS는 b-trees를 사용하여 우수한 I/O 확장성을 제공하고 모든 사용자 데이터 및 메타데이터를 인덱스합니다. 인덱스의 모든 작업이 기본 b-trees의 로그 확장성 특징을 상속하므로 개체 수는 증가합니다. mkfs 실행 시 XFS가 제공하는 튜닝 옵션 중 일부는 b-trees의 폭이 달라 이는 다른 서브 시스템의 확장성 특징을 변경합니다.

7.3.2.1. XFS의 기본적 튜닝

일반적으로 기본 XFS 형식과 마운트 옵션은 대부분의 워크로드에 최적화되어 있습니다. Red Hat은 특정 설정 변경이 파일 시스템의 워크로드에 혜택을 부여하지 않는 한 기본값 사용을 권장합니다. 소프트웨어 RAID를 사용하고 있는 경우 mkfs.xfs 명령은 올바른 스트라이프 장치와 하드웨어에 맞춰 폭 자체를 자동으로 구성합니다. 하드웨어 RAID를 사용하고 있는 경우 이를 수동으로 설정해야 할 수도 있습니다.

파일 시스템이 NFS를 통해 내보내기되고 레거시 32 비트 NFS 클라이언트가 파일 시스템으로의 액세스를 필요로 하는 경우를 제외하고 멀티 테라바이트 파일 시스템의 경우 inode64 마운트 옵션이 강력하게 권장됩니다.

자주 수정되거나 파열 상태의 파일 시스템의 경우 logbsize 마운트 옵션이 권장됩니다. 기본값은 MAX (32 KB, 로그 스트라이프 단위)이며 최대 크기는 256 KB입니다. 대폭적인 수정이 이루어지는 파일 시스템의 경우 256 KB 값이 권장됩니다.

7.3.2.2. XFS의 고급 튜닝

XFS 매개 변수를 변경하기 전 기본 XFS 매개 변수가 성능 문제를 일으키는 원인을 이해할 필요가 있습니다. 애플리케이션이 무엇을 실행하고 이에 대해 파일 시스템이 어떻게 반응하는지에 대해 이해하고 있어야 합니다.

일반적으로 튜닝으로 수정 또는 감소될 수 있는 눈에 띄는 성능 문제는 파일 시스템에 있는 파일 조각화 또는 리소스 경합에 의해 발생합니다. 이러한 문제를 해결하기 위해 여러 가지 방법이 있으며 일부 경우 파일 시스템 설정이 아닌 애플리케이션의 수정이 필요할 수도 있습니다.

이전에 이 과정을 실행해 본 적이 없는 경우 로컬 Red Hat 지원 엔지니어의 도움을 받는 것이 좋습니다.

다수 파일의 최적화

XFS는 파일 시스템에 저장할 수 있는 파일 수에 임의의 제한을 부과합니다. 일반적으로 이러한 제한은 초과할 수 없도록 높게 설정됩니다. 미리 기본 제한이 충분하지 않다고 판단될 경우 mkfs.xfs 명령을 사용하여 inodes에 허용되는 파일 시스템 공간의 비율을 늘릴 수 있습니다. 파일 시스템 생성 후 파일 제한이 발생한 경우 (파일 또는 디렉토리를 생성하려고 할 때 여유 공간이 충분해도 일반적으로 ENOSPC 오류가 표시됨) xfs_growfs 명령을 사용하여 제한을 조정할 수 있습니다.

단일 디렉토리에 다수의 파일을 최적화

디렉토리 블록 크기는 파일 시스템의 수명 동안 고정되어 있으며 **mkfs**로 초기화 포맷을 제외하고는 변경할 수 없습니다. 최소 디렉토리 블록은 파일 시스템 블록 크기로 기본값은 **MAX (4 KB, 파일 시스템 블록 크기)** 입니다. 일반적으로 디렉토리 블록 크기를 줄일 필요가 없습니다.

디렉토리 구조는 **b-tree** 기반이기 때문에 블록 크기를 변경하면 물리적 I/O 당 변경 또는 검색할 수 있는 디렉토리 정보의 양에 영향을 미칩니다. 디렉토리가 클수록 특정 블록 크기에 더 많은 I/O 작업을 필요로 합니다.

하지만 대형 디렉토리 블록 크기를 사용하고 있을 경우 소형의 디렉토리 블록 크기의 파일 시스템에서의 동일한 작업에 비해 각 수정 작업에 의해 더 많은 **CPU**가 소비됩니다. 즉 디렉토리 크기가 작은 경우 대형 디렉토리 블록 크기의 수정 성능이 더 낮아지는 것입니다. I/O가 성능 제한 요소가 되는 크기에 디렉토리가 도달하면 대형 블록 크기 디렉토리 성능이 향상됩니다.

항목 당 이름의 길이가 **20-40** 바이트이고, 항목 수가 최대 **1-2**백만인 디렉토리의 경우 **4 KB** 디렉토리 블록 크기와 **4 KB** 파일 시스템 블록 크기의 기본 설정이 가장 적합합니다. 파일 시스템에 더 많은 항목이 필요한 경우 대형 디렉토리 블록 크기는 성능이 향상하는 경향이 있습니다. **100-1000**만개 디렉토리 항목을 갖는 파일 시스템은 **16 KB** 블록 크기가 가장 적합하며 **1000**만개 이상의 디렉토리 항목을 갖는 파일 시스템은 **64 KB** 블록 크기가 가장 적합합니다.

워크로드가 수정보다 임의의 디렉토리 검색을 더 많이 사용하는 경우 (즉, 디렉토리 읽기가 디렉토리 쓰기보다 더 일반적이거나 중요한 경우) 블록 크기를 늘리면 위의 임계값은 약 **1** 자리 낮은 것이 됩니다.

동시성 최적화

다른 파일 시스템과 달리 작업이 비공유 객체에서 실행되고 있으면 **XFS**는 많은 유형의 할당 및 할당 해제 작업을 동시에 실행할 수 있습니다. 익스텐트 할당 또는 할당 해제는 이러한 작업이 다른 할당 그룹에서 실행되고 있는 경우 동시에 실행할 수 있습니다. 유사하게 **inode**의 할당 또는 할당 해제도 동시 실행 작업이 다른 할당 그룹에 영향을 미치고 있는 경우 동시에 실행할 수 있습니다.

동시에 작업을 수행하려고 시도하는 멀티 스레드 애플리케이션과 높은 **CPU** 수를 갖는 시스템을 사용할 때 할당 그룹 수는 중요합니다. **4** 개의 할당 그룹만이 존재하고 유지되는 경우 병렬 메타데이터 작업은 이러한 **4** 개의 **CPU** 까지만 확장됩니다 (시스템에 의해 제공되는 동시성 제한). 소형 파일 시스템의 경우 할당 그룹 수는 시스템이 제공하는 동시성에 의해 지원되도록 합니다. 대형 파일 시스템 (수십 테라 바이트 이상)의 경우 일반적으로 기본 포맷 옵션이 동시성을 제한하지 않도록 충분한 할당 그룹을 생성합니다.

XFS 파일 시스템의 구조에 고유의 병렬 처리를 사용하려면 애플리케이션은 단일 경합 지점을 알고 있어야 합니다. 동시에 디렉토리를 변경할 수 없기 때문에 다수의 파일을 생성 및 삭제하는 애플리케이션은 하나의 디렉토리에 모든 파일을 저장해서는 안됩니다. 생성된 각 디렉토리는 다른 할당 그룹에 배치되므로 여러 하위 디렉토리를 통한 파일 해시와 같은 기술은 단일 대형 디렉토리를 사용하는 것에 비해 더 확장 가능한 스토리지 패턴을 제공합니다.

확장된 속성을 사용하는 애플리케이션 최적화

inode에 사용 가능한 공간이 있을 경우 **XFS**는 직접 **inode**에 작은 속성을 저장할 수 있습니다. 속성이 **inode**에 잘 맞는 경우 다른 속성 블록을 검색하기 위한 별도의 I/O를 필요로 하지 않고 검색 및 수정할 수 있습니다. 인라인 (**inode**) 및 아웃 오브 라인 (**out-of-line**) 속성 간의 성능 차이는 아웃 오브 라인 속성이 한 자리 적을 정도로 다릅니다.

256 바이트의 기본값 **inode** 크기는 **inode**에 저장된 데이터 익스텐트 포인터 수에 따라 약 **100** 바이트의 속성 공간을 사용할 수 있습니다. 기본값 **inode** 크기는 적은 수의 작은 속성을 저장하는 경우에만 유용합니다.

mkfs 실행시에 **inode** 크기를 늘리면 속성을 인라인으로 저장하는데 사용 가능한 공간의 양을 늘릴 수 있습니다. **512** 바이트 **inode** 크기는 속성의 사용 가능한 공간을 약 **350** 바이트로 늘립니다. **2 KB inode**에는 약 **1900** 바이트의 사용 가능한 공간이 있습니다.

하지만 인라인으로 저장할 수 있는 개별 속성의 크기에는 제한이 있습니다. 속성 이름과 값 모두 최대 크기 제한은 254 바이트입니다 (즉, 254 바이트의 이름 길이와 254 바이트의 값 길이를 갖는 속성은 인라인으로 저장됩니다). 이러한 크기 제한을 초과하면 inode에 모든 속성을 저장할 충분한 공간이 있어도 속성은 아웃 오브 라인에 강제됩니다.

지속적인 메타데이터 수정을 위한 최적화

로그 크기는 지속적인 메타 데이터 수정의 달성 수준을 결정하는 주요 요인입니다. 로그 장치는 원형이기 때문에 후행 부분이 덮어쓰기되기 전 로그에 있는 모든 수정은 디스크의 실제 위치에 기록되어야 합니다. 이에 모든 더티 메타데이터를 다시 되돌려 작성하기 위한 상당한 노력이 포함될 수 있습니다. 기본값 설정은 전체 파일 시스템 크기와 관련하여 로그 크기를 확장하기 때문에 대부분의 경우 로그 크기를 튜닝할 필요가 없습니다.

작은 로그 장치는 매우 자주 메타데이터 쓰기 저장 (writeback)을 발생시키게 됩니다 - 로그가 지속적으로 공간을 확보하기 위해 후행 부분에 푸시되므로 수정된 메타데이터는 자주 디스크에 기록되어 작업이 느리게 됩니다.

로그 크기를 늘리면 후행 부분 푸시 이벤트 간의 시간 간격이 증가합니다. 이는 더티 메타데이터를 보다 효율적으로 통합할 수 있게 하여 메타데이터 쓰기 저장 패턴을 개선하고 자주 수정되는 메타데이터의 쓰기 저장을 줄입니다. 대신 큰 로그는 메모리에 있는 모든 변경 사항을 추적하기 위해 더 많은 메모리가 필요하게 됩니다.

컴퓨터의 메모리가 한정되어 있는 경우 큰 로그는 유용하지 않습니다. 이는 큰 로그의 장점이 실현되기 전 메모리 제한으로 메타데이터 쓰기 저장이 발생하기 때문입니다. 이러한 경우 공간 부족이 발생하는 로그에서 메타데이터 쓰기 저장은 메모리 회복에 의한 쓰기 저장 보다 더 효율적이기 때문에 큰 로그 보다 작은 로그가 더 나은 성능을 제공합니다.

파일 시스템이 들어 있는 장치의 기본 스트라이프 단위에 항상 로그를 정렬해야 합니다. **mkfs**는 MD 및 DM 장치에 대해 기본값으로 이를 실행하지만 하드웨어 RAID의 경우 이를 지정해야 합니다. 이를 올바르게 설정하면 디스크에 수정 사항을 기록할 때 정렬되지 않은 I/O 및 후속하는 읽기-수정-쓰기 작업의 원인이 되는 로그 I/O의 모든 가능성을 해결합니다.

마운트 옵션을 편집하여 로그 작업을 더욱 개선할 수 있습니다. 메모리 내의 로그 버퍼의 크기 (**logbsize**)를 늘리면 로그에 변경 사항을 기록할 수 있는 속도가 증가합니다. 기본값 로그 버퍼 크기는 **MAX** (32 KB, 로그 스트라이프 단위)이며 최대 크기는 256 KB입니다. 일반적으로 값이 크면 성능 속도가 빨라집니다. 하지만 **fsync**가 많은 워크로드에서 작은 로그 버퍼는 대형의 스트라이프 단위 조정을 사용하면 큰 버퍼 보다 현저하게 빨라집니다.

delaylog 마운트 옵션은 로그에 변경 사항 수를 줄임으로써 지속적인 메타데이터 수정 성능을 향상시킵니다. 이는 로그에 변경 사항을 작성하기 전 메모리에 있는 개별적 변경 사항을 집계하여 달성합니다. 자주 수정되는 메타데이터는 수정할 때 마다가 아닌 정기적으로 로그에 기록됩니다. 이 옵션은 더티 메타데이터를 추적하는 메모리 사용량을 늘리고 충돌이 발생하면 작업 손실의 가능성이 증가하지만 메타데이터 수정 속도와 확장성을 한 자리수 이상 개선할 수 있습니다. 데이터 및 메타데이터가 디스크에 작성되는지 확인하기 위해 **fsync**, **fdatasync**, **sync**를 사용할 때 이 옵션을 사용해도 데이터나 메타데이터의 무결성이 감소되지 않습니다.

7.4. 클러스터링

클러스터 스토리지는 클러스터에 있는 모든 서버에 걸쳐 일관된 파일 시스템 이미지를 제공하여 서버가 단일 공유 파일 시스템에 읽기/쓰기를 가능하게 합니다. 이는 애플리케이션의 설치 및 패칭 같은 작업을 하나의 파일 시스템으로 제한함으로써 스토리지 관리를 단순화합니다. 전체 클러스터 파일 시스템은 애플리케이션 데이터의 중복 사본의 필요성을 제거하고 백업 및 장애 복구를 단순화합니다.

Red Hat의 고가용성 애드온은 Red Hat Global File System 2 (장애 복구형 스토리지 애드온의 일부)와 함께 클러스터 스토리지를 제공합니다.

7.4.1. GFS 2 (Global File System 2)

Global File System 2 (GFS2)는 기본 파일 시스템으로 Linux 커널 파일 시스템과 직접 상호 작용합니다. 이는 여러 컴퓨터 (노드)가 동시에 클러스터에서 동일한 스토리지 장치를 공유할 수 있게 합니다. GFS2 파일 시스템은 대부분 자체 튜닝을 하지만 수동 튜닝도 가능합니다. 다음 부분에서는 수동으로 성능을 튜닝을 할 때 성능 고려 사항에 대해 설명합니다.

Red Hat Enterprise Linux 6.4에서는 GFS2로 파일 조각화 관리를 개선할 수 있게 되었습니다. Red Hat Enterprise Linux 6.3 및 이전 버전에 의해 생성된 파일은 하나 이상의 프로세스에서 동시에 여러 파일이 작성된 경우, 파일의 조각화가 일어나는 경향이 있었습니다. 이러한 조각화로 인해 특히 큰 파일과 관련된 워크로드의 경우 속도가 느려졌습니다. Red Hat Enterprise Linux 6.4에서 동시 쓰기를 하면 파일 조각화가 줄어들기 때문에 이러한 워크로드의 성능이 향상됩니다.

Red Hat Enterprise Linux에서는 GFS2 용 조각 모음 도구가 없는 반면 **filefrag** 도구로 파일을 식별하여 이를 임시 파일에 복사하고 이 임시 파일의 이름을 변경한 후 원래 파일로 대체하여 개별 파일을 조각 모음할 수 있습니다. (이 절차는 쓰기가 지속적으로 이루어지고 있는 한 6.4 이전 버전에서도 실행할 수 있습니다.)

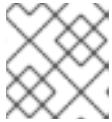
GFS2는 클러스터 노드 간의 통신을 필요로 하는 글로벌 잠금 메커니즘을 사용하기 때문에 시스템이 이러한 노드 간의 파일 및 디렉토리 경합을 방지하도록 설계되어 있는 경우 최상의 성능을 얻을 수 있습니다. 경합을 방지하는 방법 중 일부는 다음과 같습니다:

- 가능한 경우 **fallocate**로 파일 및 디렉토리를 사전할당하여 할당 프로세스를 최적화하고 소스 페이지를 잠금할 필요가 없도록 합니다.
- 여러 노드 간에 공유되는 파일 시스템의 영역을 최소화하고 노드간 캐시 무효화를 최소화하며 성능을 향상시킵니다. 예를 들어 여러 노드가 동일한 파일 시스템을 마운트해서 다른 서브 디렉토리에 액세스하면 서브 디렉토리를 다른 파일 시스템으로 이동하여 성능을 향상시킬 가능성이 높아집니다.
- 최적의 리소스 그룹 크기와 수를 선택합니다. 이는 전형적인 파일 크기와 시스템에서 사용 가능한 여유 공간에 따라 다르며 여러 노드가 리소스 그룹을 동시에 사용하려고 시도할 가능성에 영향을 미칩니다. 리소스 그룹이 너무 많으면 할당 공간이 배치되는 동안 블록 할당이 느려집니다. 반면 리소스 그룹이 너무 적으면 할당 해제 동안 잠금 경합을 일으킬 수 있습니다. 일반적으로 여러 설정을 테스트하고 워크로드에 가장 적합한 것을 지정하는 것이 가장 좋습니다.

하지만 경합은 GFS2 파일 시스템 성능에 영향을 미치는 유일한 문제가 아닙니다. 전반적인 성능을 개선하기 위한 기타 모범 사례는 다음과 같습니다:

- 클러스터 노드에서 예측되는 I/O 패턴 및 파일 시스템의 성능 요구 사항에 따라 스토리지 하드웨어를 선택합니다.
- 가능한 솔리드 스테이트 스토리지를 사용하여 검색 시간을 줄입니다.
- 워크로드에 적합한 크기의 파일 시스템을 생성하고 파일 시스템이 용량의 80%를 초과하지 않도록 합니다. 작은 파일 시스템은 크기에 비례하여 짧은 백업 시간을 가지며 파일 시스템 검사에 필요한 시간과 메모리가 적지만 워크로드에 대해 너무 작을 경우 높은 조각화가 진행될 수 있습니다.
- 메타데이터 집약적인 워크로드의 경우 또는 저널링 데이터가 사용되고 있는 경우 저널 크기를 크게 설정합니다. 이것이 더 많은 메모리를 사용하더라도 쓰기가 필요하기 전에 데이터를 저장하기 위해 더 많은 저널링 공간을 사용할 수 있으므로 성능이 향상됩니다.
- GFS2 노드에 있는 클럭이 네트워크 애플리케이션 문제를 방지하도록 동기화되어 있는지 확인합니다. NTP (Network Time Protocol) 사용을 권장합니다.

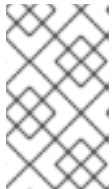
- 애플리케이션 작업에서 파일이나 디렉토리 액세스 시간이 중요하지 않은 경우 파일 시스템을 **noatime** 및 **nodiratime** 마운트 옵션으로 마운트합니다.



참고

Red Hat은 GFS2에서 **noatime** 옵션을 사용할 것을 권장합니다.

- 할당량을 사용할 필요가 있는 경우 쿼터 동기화 트랜잭션의 빈도를 낮추거나 **fuzzy** 쿼터 동기화를 사용하여 지속적인 쿼터 파일 업데이트에서 발생하는 성능 문제를 방지합니다.



참고

Fuzzy 쿼터 계산은 사용자 및 그룹이 쿼터 제한을 약간 초과하는 것을 허용합니다. 이 문제를 최소화하기 위해 사용자 또는 그룹이 쿼터 제한에 도달하면 GFS2는 동기화 주기를 동적으로 감소시킵니다.

GFS2 성능 튜닝의 각 측면에 대한 보다 자세한 내용은

http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에 있는 *Global File System 2* 가이드에서 참조하십시오.

8장. 네트워킹

Red Hat Enterprise Linux의 네트워크 스택은 장기간에 걸쳐 수많은 자동 최적화 기능으로 업그레이드되었습니다. 대부분의 워크로드의 경우 자동 구성된 네트워크 설정이 최적의 성능을 제공합니다.

대부분의 경우 네트워크 성능 문제는 실제로 하드웨어의 결함이나 인프라의 장애에 기인하고 있습니다. 이러한 원인은 이 문서의 범위에서 벗어나는 것입니다. 이 장에서 다루어지는 성능 문제 및 해결 방법은 완전하게 기능적인 시스템을 최적화하는데 유용합니다.

네트워크는 섬세한 서브시스템으로 다른 부분이 민감한 연결에 형성되어 있습니다. 오픈 소스 커뮤니티와 Red Hat이 네트워크 성능을 자동으로 최적화하기 위해 구현 방법에 많은 노력을 기울이고 있는 것은 이 때문입니다. 따라서 대부분의 워크로드에서 성능을 위해 네트워크를 재구성할 필요가 없을 수 있습니다.

8.1. 네트워크 성능 개선

Red Hat Enterprise Linux 6.1에는 다음과 같은 네트워크 성능 개선 사항이 제공됩니다:

8.1.1. RPS (Receive Packet Steering)

RPS는 단일 NIC **rx** 큐를 활성화하여 수신 **softirq** 워크로드를 여러 CPU에 분산하는 것을 허용합니다. 이는 네트워크 트래픽이 단일 NIC 하드웨어 큐에서의 병목 현상을 방지할 수 있습니다.

RPS를 활성화하려면 대상 CPU 이름을 **/sys/class/net/ethX/queues/rx-N/rps_cpus**에 지정합니다. 여기서 **ethX**는 NIC의 해당 장치 이름 (예: **eth1**, **eth2**)으로 **rx-N**은 지정된 NIC 수신 큐로 대체합니다. 이렇게 하면 파일에 지정된 CPU가 **ethX**에 있는 큐 **rx-N**에서 데이터를 처리할 수 있습니다. CPU를 지정할 때 큐의 캐시 친화도^[4]를 고려합니다.

8.1.2. RFS (Receive Flow Steering)

RFS는 RPS의 확장 버전으로 애플리케이션이 데이터를 수신하고 네트워크 스택이 질문하여 자동으로 채워지는 해시 테이블을 관리자가 설정할 수 있게 합니다. 이는 어떤 애플리케이션이 네트워크 데이터의 어느 부분을 수신할 지를 결정합니다 (source:destination 네트워크 정보에 따라).

이 정보를 사용하여 네트워크 스택은 각각의 패킷을 수신하기 위해 최적의 CPU를 스케줄링할 수 있습니다. RFS를 설정하려면 다음과 같은 매개 변수를 사용합니다:

/proc/sys/net/core/rps_sock_flow_entries

이는 커널이 지정된 CPU 방향으로 조정할 수 있는 소켓/흐름의 최대 수를 제어합니다. 이는 시스템 전체의 공유 제한입니다.

/sys/class/net/ethX/queues/rx-N/rps_flow_cnt

이는 NIC (**ethX**)에서 커널이 특정 수신 큐 (**rx-N**)를 조정할 수 있는 소켓/흐름의 최대 수를 제어합니다. 모든 NIC에서 이러한 조정 가능한 매개 변수의 모든 큐마다 값의 합계는 같거나 **/proc/sys/net/core/rps_sock_flow_entries** 이하이어야 함에 유의합니다.

RPS와 달리 RFS는 패킷 흐름을 처리할 때 수신 큐와 애플리케이션 모두 동일한 CPU를 공유할 수 있게 합니다. 이는 경우에 따라 성능 개선으로 이어집니다. 하지만 이러한 개선은 캐시 계층, 애플리케이션 로드 등과 같은 요소에 따라 달라집니다.

8.1.3. TCP thin-stream의 getsockopt 지원

*Thin-stream*은 애플리케이션이 프로토콜의 재전송 메커니즘이 완전히 포화 상태가 되지 않도록 데이터를

매우 낮은 속도로 전송하는 것을 특징으로 전송 프로토콜에 사용되는 용어입니다. Thin-stream 프로토콜을 사용하는 애플리케이션은 일반적으로 TCP와 같은 신뢰성 프로토콜을 통해 전송합니다. 대부분의 경우, 이러한 애플리케이션은 매우 시간에 민감한 서비스를 제공합니다 (예: 주식 거래, 온라인 게임, 제어 시스템).

시간에 민감한 서비스의 경우 패킷 손실은 서비스의 품질에 막대한 손해를 가져올 수 있습니다. 이를 방지하기 위해 **getsockopt** 호출이 강화되어 다음과 같은 두 가지 옵션을 지원합니다:

TCP_THIN_DUPACK

이 부울 값은 하나의 dupACK 후 thin stream의 재전송 동적 트리거를 활성화합니다.

TCP_THIN_LINEAR_TIMEOUTS

이 부울 값은 thin stream의 선형 시간 초과 동적 트리거를 활성화합니다.

두 옵션 모두 애플리케이션에 의해 활성화됩니다. 이러한 옵션에 대한 자세한 내용은 **file:///usr/share/doc/kernel-doc-version/Documentation/networking/ip-sysctl.txt**에서 참조하십시오. thin-stream에 대한 보다 자세한 내용은 **file:///usr/share/doc/kernel-doc-version/Documentation/networking/tcp-thin.txt**에서 참조하십시오.

8.1.4. 투명 프록시 (TPProxy) 지원

커널은 투명 프록시를 지원하기 위해 비로컬 바인딩된 IPv4 TCP 및 UDP 소켓을 처리할 수 있습니다. 이를 활성화하려면 **iptables**를 적절히 구성해야 합니다. 또한 정책 라우팅을 적절하게 설정 및 활성화해야 합니다.

투명 프록시에 대한 보다 자세한 내용은 **file:///usr/share/doc/kernel-doc-version/Documentation/networking/tpoxy.txt**에서 참조하십시오.

8.2. 네트워크 설정 최적화

성능 튜닝은 일반적으로 선점 방식으로 이루어집니다. 종종 애플리케이션을 실행하거나 시스템을 배포하기 전 알려진 변수를 조정합니다. 조정이 효과적 없는 것으로 입증되면 다른 변수를 조정 시도합니다. 이러한 방법의 기초가 되는 논리는 기본적으로 시스템은 성능 최적의 수준에서 작동하지 않는다는 것입니다. 따라서 이에 따라 시스템을 적절히 조정해야 한다고 생각합니다. 일부 경우 계산된 추측을 통해 실행합니다.

이전에 설명했듯이 네트워크 스택은 대부분 자체 최적화됩니다. 또한 효과적으로 네트워크를 튜닝하는 데는 네트워크 스택이 작동하는 방법 뿐 만 아니라 특정 시스템의 네트워크 리소스 요구 사항도 철저히 이해하고 있어야 합니다. 잘못된 네트워크 성능 구성을 통해 실제로 성능이 저하될 수 있습니다.

예를 들어 **bufferfloat 문제**에 대해 생각해 봅시다. 버퍼 큐의 깊이를 증가시키면 링크가 허용하는 것보다 큰 혼잡 윈도우가 있는 TCP 연결을 초래합니다 (깊은 버퍼링으로 인해). 그러나 이러한 연결은 프레임이 상당한 시간을 큐에서 소모하기 때문에 거대한 RTT 값을 가지게 됩니다. 이는 결국 혼잡을 감지할 수 없게 되므로, 차선의 출력을 얻게 되는 결과를 초래합니다.

네트워크 성능의 경우 특정 성능 문제가 명백한 경우 이외는 기본값 설정을 유지하는 것이 좋습니다. 이러한 문제에는 프레임 손실 및 처리량 대폭 감소 등과 같은 문제가 포함됩니다. 이러한 문제도 단순히 튜닝 설정을 상향 조정하는 것 (버퍼/큐의 깊이를 늘리거나 인터럽트 대기 시간을 감소하는 등) 보다 문제를 세심하게 조사하는 것이 좋은 해결 방법이 되는 경우가 많습니다.

네트워크 성능 문제를 올바르게 진단하려면 다음과 같은 도구를 사용합니다:

netstat

네트워크 연결, 라우팅 테이블, 인터페이스 통계, 마스커레이드 연결, 멀티캐스트 구성원을 인쇄하는 명령행 유틸리티입니다. 이는 **/proc/net/** 파일 시스템에서 네트워킹 서브시스템에 대한 정보를 검색합니다. 이러한 파일은 다음과 같습니다:

- **/proc/net/dev** (장치 정보)
- **/proc/net/tcp** (TCP 소켓 정보)
- **/proc/net/unix** (Unix 도메인 소켓 정보)

netstat 및 **/proc/net/**에서의 관련 참조 파일에 대한 보다 자세한 내용은 **netstat man** 페이지인 **man netstat**에서 참조하십시오.

dropwatch

커널이 드롭한 패킷을 감시하는 모니터링 유틸리티입니다. 보다 자세한 내용은 **dropwatch man** 페이지인 **man dropwatch**에서 참조하십시오.

ip

라우트, 장치, 정책 라우팅, 터널을 관리 및 모니터링하기 위한 유틸리티입니다. 보다 자세한 내용은 **ip man** 페이지인 **man ip**에서 참조하십시오.

ethtool

NIC 설정을 표시하거나 변경하기 위한 유틸리티입니다. 보다 자세한 내용은 **ethtool man** 페이지인 **man ethtool**에서 참조하십시오.

/proc/net/snmp

snmp 에이전트의 IP, ICMP, TCP, UDP 관리 정보 베이스에 필요한 ASCII 데이터를 표시하는 파일입니다. 또한 이는 실시간 UDP-lite 통계도 표시합니다.

SystemTap 초보자 가이드에는 네트워크 성능을 프로파일링 및 모니터링에 사용할 수 있는 여러 스크립트의 예가 들어 있습니다. 이 가이드는 http://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/에서 확인하실 수 있습니다.

네트워크 성능 문제에 관련된 데이터를 수집한 후 이론 — 바라건데 해결책을 고안할 수 있어야 합니다. [5] 예를 들어 **/proc/net/snmp**에서 UDP 입력 오류가 증가하면 네트워크 스택은 새로운 프레임을 애플리케이션의 소켓으로 추가하려고 할 때 하나 이상의 소켓 수신 큐가 가득차다고 나타냅니다.

이는 패킷이 적어도 하나의 소켓 큐에서 병목 현상되는 것을 나타내고 있으며, 이는 소켓 큐가 너무 느리게 패킷을 배출하거나 소켓 큐에 대한 패킷 볼륨이 너무 큰 것을 의미합니다. 후자의 경우 손실된 데이터에 대한 네트워크 집약적 애플리케이션 로그를 확인합니다. 이 문제를 해결하려면 문제가 있는 애플리케이션을 최적화하거나 재구성해야 합니다.

8.2.1. 소켓 수신 버퍼 크기

소켓의 송수신 크기는 동적으로 조정되므로 수동으로 편집할 필요가 없습니다. **SystemTap** 네트워크 예 **sk_stream_wait_memory.stp**에 있는 분석과 같이 추가 분석할 경우 소켓 큐의 소진 비율이 너무 느리므로 애플리케이션의 소켓 큐의 정도를 증가시키는 것이 좋습니다. 이를 위해 다음 값 중 하나를 설정하여 소켓이 사용하는 수신 버퍼 크기를 늘립니다:

rmem_default

소켓이 사용하는 수신 버퍼의 기본값 크기를 제어하는 커널 매개 변수입니다. 이를 설정하려면 다음 명령을 실행합니다:

```
sysctl -w net.core.rmem_default=N
```

N을 원하는 버퍼 크기 (바이트 단위)로 대체합니다. 이 커널 매개 변수의 값을 지정하려면 `/proc/sys/net/core/rmem_default`에서 확인합니다. `rmem_default` 값은 `rmem_max` (`/proc/sys/net/core/rmem_max`)를 초과하지 않도록 주의합니다. 필요한 경우 `rmem_max` 값을 늘립니다.

SO_RCVBUF

소켓의 수신 버퍼의 최대 크기를 바이트 단위로 제어하는 소켓 옵션입니다. **SO_RCVBUF**에 대한 자세한 내용은 **man 7 socket**의 man 페이지에서 참조하십시오.

SO_RCVBUF를 설정하려면 **setsockopt** 유틸리티를 사용합니다. 현재 **SO_RCVBUF** 값은 **getsockopt**로 검색할 수 있습니다. 이러한 두 가지 유틸리티에 대한 보다 자세한 내용은 **setsockopt** man 페이지인 **man setsockopt**에서 참조하십시오.

8.3. 패킷 수신 개요

네트워크 병목 현상 및 성능 문제에 대한 분석을 위해 패킷 수신 작동 방법을 이해하고 있어야 합니다. 수신 경로에서 프레임이 자주 손실되기 때문에 패킷 수신은 네트워크 성능 튜닝에서 중요합니다. 수신 경로에서 손실된 프레임은 네트워크 성능을 크게 저하시킬 수 있습니다.

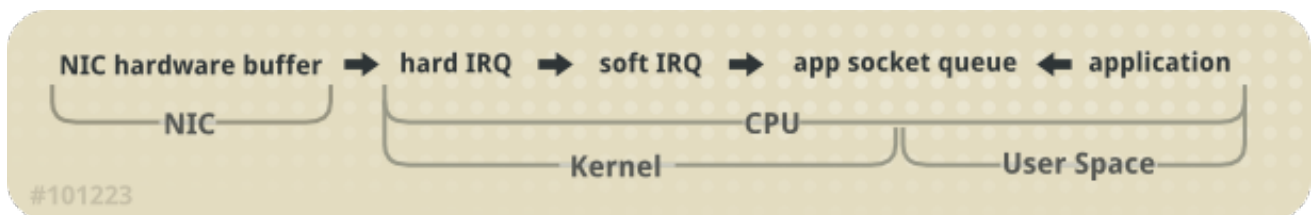


그림 8.1. 네트워크 수신 경로 다이어그램

Linux 커널은 각 프레임을 수신하고 다음과 같은 4 단계의 과정을 수행합니다:

1. **하드웨어 리셉션:** **NIC** (network interface card)는 와이어에서 프레임을 수신합니다. 해당 드라이버 설정에 따라 **NIC**는 프레임을 내부 하드웨어 버퍼 메모리 또는 지정된 링 버퍼로 전송합니다.
2. **하드 IRQ:** **NIC**는 **CPU**를 인터럽트하여 넷 프레임의 존재를 주장합니다. 이는 **NIC** 드라이버가 인터럽트를 승인하고 **소프트 IRQ** 작업을 스케줄링합니다.
3. **소프트 IRQ:** 이 단계에서는 실제 프레임 수신 프로세스를 구현하고 **softirq** 문맥에서 실행합니다. 즉 이 단계는 지정된 **CPU**에서 실행되고 있는 모든 애플리케이션을 먼저 실행하지만 하드 **IRQ**의 주장을 허용합니다.

이 문맥에서 (하드 **IRQ**로 동일한 **CPU**에서 실행하여 잠금 오버헤드를 최소화하고 있는) 커널은 실제로 **NIC** 하드웨어 버퍼에서 프레임을 제거하여 네트워크 스택을 통해 처리합니다. 여기서 프레임은 대상 청취 소켓에 전송, 삭제 또는 전달됩니다.

소켓에 전달되면 프레임은 소켓을 소유하는 애플리케이션에 추가됩니다. 이러한 프로세스는 **NIC** 하드웨어 버퍼가 프레임을 소진할 때 까지 또는 **장치 무게 (device weight) (dev_weight)** 까지 반복됩니다. 장치 무게에 대한 보다 자세한 내용은 **8.4.1절. "NIC 하드웨어 버퍼"**에서 참조하십시오.

4. **애플리케이션 수신**: 애플리케이션은 프레임을 수신하고 표준 POSIX 호출 (**read**, **recv**, **recvfrom**)을 통해 소유한 소켓에서 큐를 분리합니다. 이 때에 네트워크를 통해 수신된 데이터는 네트워크 스택에 더 이상 존재하지 않습니다.

8.3.1. CPU/캐시 친화도

수신 경로에서 높은 처리량을 유지하려면 L2 캐시를 *hot*으로 유지하는 것이 좋습니다. 이미 설명했듯이 네트워크 버퍼는 존재를 알리는 IRQ와 동일한 CPU에서 수신됩니다. 즉 버퍼 데이터는 수신 CPU의 L2 캐시에 있는 것입니다.

이 기능을 사용하려면 L2 캐시와 동일한 코어를 공유하는 NIC에서 가장 많이 데이터를 수신할 것으로 예상되는 애플리케이션에 프로세스 친화도를 배치합니다. 이는 캐시 적중 확률을 극대화하여 성능을 향상시킵니다.

8.4. 일반적인 큐/프레임 손실 문제 해결

지금까지 프레임 손실의 가장 일반적인 이유는 *큐 오버런 (queue overrun)*입니다. 커널은 큐의 길이에 제한을 설정하고 있기 때문에 경우에 따라서는 큐의 배출보다 더 빨리 채워집니다. 이 현상이 오래 지속되면 프레임 드롭이 시작됩니다.

그림 8.1. “네트워크 수신 경로 다이어그램”에서 보여주듯이, 수신 경로에는 NIC 하드웨어 버퍼와 소켓 큐라는 두 가지 주요 큐가 있습니다. 두 큐 모두는 큐 오버런을 방지하기 위해 적절하게 설정해야 합니다.

8.4.1. NIC 하드웨어 버퍼

NIC는 프레임으로 하드웨어 버퍼를 채웁니다. 다음으로 버퍼는 인터럽트를 통해 NIC가 주장하는 **softirq**에 의해 배출됩니다. 큐의 상태를 확인하려면 다음과 같은 명령을 사용합니다:

```
ethtool -S ethX
```

ethX를 NIC의 해당 장치 이름으로 대체합니다. 이는 **ethX** 내에서 얼마나 많은 프레임이 드롭되었는지를 표시합니다. 드롭이 발생하는 대부분의 이유는 큐가 프레임을 저장할 버퍼 공간이 부족하기 때문입니다.

이러한 문제를 해결하려면 다음과 같은 여러가지 방법이 있습니다:

입력 트래픽

입력 트래픽을 느리게 하여 큐 오버런을 방지할 수 있습니다. 이는 통합된 멀티캐스트 그룹의 수를 필터링하여 줄이고 브로드 캐스트 트래픽을 감소시켜 달성할 수 있습니다.

큐 길이

다른 방법으로 큐의 길이를 늘릴 수 있습니다. 이는 드라이버가 허용하는 최대 값까지 지정된 큐에 있는 버퍼 수를 늘리는 것입니다. 이를 위해 다음과 같이 **ethX**의 **rx/tx** 링 매개변수를 편집합니다:

```
ethtool --set-ring ethX
```

앞서 언급한 명령에 해당 **rx** 또는 **tx** 값을 추가합니다. 보다 자세한 내용은 **man ethtool**에서 참조하십시오.

장치 무게

큐가 배출되는 속도를 높일 수 있습니다. 이를 위해 NIC의 *장치 무게 (device weight)*를 적절하게 조정합니다. 이 속성은 **softirq** 컨텍스트가 CPU를 포기하고 자체 일정을 변경하기 전 NIC가 받을 수 있는 최대 프레임 수를 말합니다. 이는 **/proc/sys/net/core/dev_weight** 변수에 의해 제어됩니다.

대부분의 관리자는 세 번째 옵션을 선택하는 경향이 있습니다. 하지만 이 옵션을 실행하면 그에 따른 결과가 수반된다는 점에 유의하십시오. 한 번의 반복에서 NIC에서 수신할 수 있는 프레임 수를 늘리면 CPU 사이클이 증가하게 되어, 해당 CPU에서 애플리케이션을 스케줄할 수 없습니다.

8.4.2. 소켓 큐

NIC 하드웨어 큐처럼 소켓 큐는 **softirq** 컨텍스트에서 네트워크 스택으로 채워집니다. 그러면 애플리케이션은 **read**, **recvfrom** 등의 호출을 통해 해당 소켓의 큐를 배출합니다.

이 큐의 상태를 모니터링하려면 **netstat** 유틸리티를 사용합니다. **Recv-Q** 칼럼은 큐의 크기를 표시합니다. 일반적으로 소켓 큐에서 오버런은 NIC 하드웨어 버퍼 오버런과 같은 방법으로 관리됩니다 (예: [8.4.1 절. “NIC 하드웨어 버퍼”](#)).

입력 트래픽

첫 번째 옵션은 큐가 채워지는 비율을 설정하여 입력 트래픽을 느려지게 합니다. 이렇게 하려면 프레임을 필터링하거나 미리 드롭합니다. NIC의 장치 무게를 낮추어 입력 트래픽을 느려지게 할 수 있습니다 [6].

큐 깊이

큐 깊이를 증가시켜 소켓 큐 오버런을 방지할 수 있습니다. 이렇게 하려면 **rmem_default** 커널 매개변수 또는 **SO_RCVBUF** 소켓 옵션의 값 중 하나를 늘립니다. 이에 대한 자세한 내용은 [8.2절. “네트워크 설정 최적화”](#)에서 참조하십시오.

애플리케이션 호출 빈도

가능하면 자주 호출을 수행하는 애플리케이션을 최적화합니다. 이에에는 보다 자주 POSIX 호출 (**recv**, **read**)을 수행하기 위해 네트워크 애플리케이션을 수정 또는 재설정하는 것이 포함됩니다. 이렇게 하면 애플리케이션이 더 빠르게 큐를 배출할 수 있습니다.

대부분의 관리자에게는 큐의 깊이를 증가하는 것이 바람직한 해결 방법입니다. 이는 가장 간단한 해결 방법이지만 항상 장기적으로 작동하지 않을 수 있습니다. 네트워킹 기술의 속도가 높아지면서 소켓 큐는 더 빠르게 채워질 것입니다. 이는 시간이 지남에 따라 해당 큐의 깊이를 다시 조정하는 것을 의미합니다.

가장 좋은 해결 방법은 애플리케이션 공간에 데이터를 추가하더라도 커널에서 데이터를 더 빠르게 배출하도록 애플리케이션을 강화하거나 설정하는 것입니다. 이렇게 하면 데이터를 필요에 따라 스왑하거나 다시 페이지징할 수 있게 되므로 데이터를 보다 유연하게 저장할 수 있습니다.

8.5. 멀티캐스트에 있어서 고려할 사항

여러 애플리케이션이 멀티 캐스트 그룹을 수신할 때 멀티캐스트 프레임을 처리하는 커널 코드에는 서로 다른 개별적 소켓의 네트워크 데이터를 복제하기 위한 디자인이 필요합니다. 이러한 복제는 시간이 오래 걸리고 **softirq** 컨텍스트에서 발생합니다.

하나의 멀티 캐스트 그룹에 여러 개의 리스너를 추가하면 **softirq** 컨텍스트의 실행 시간에 직접적인 영향을 미칩니다. 멀티 캐스트 그룹에 리스너를 추가하면 커널이 해당 그룹에 대해 수신된 각 프레임에 대해 추가 복사본을 작성해야 함을 의미합니다.

낮은 트래픽 볼륨과 리스너 수가 적은 경우 이 효과는 최소화됩니다. 하지만 여러 소켓이 트래픽이 높은 멀티 캐스트 그룹을 청취하면 **softirq** 컨텍스트의 실행 시간이 증가하여 네트워크 카드 및 소켓 큐 모두에서 프레임 드롭이 발생합니다. **softirq** 런타임이 증가하면 애플리케이션은 높은 부하 시스템에서 실행하는 기회가 감소되도록 변환되므로 고용량의 멀티 캐스트 그룹을 청취하는 애플리케이션 수의 증가로 멀티 캐스트 프레임 손실 비율이 증가합니다.

8.4.2절. “소켓 큐 ” 또는 8.4.1절. “NIC 하드웨어 버퍼 ”에서 설명하고 있듯이 소켓 큐와 NIC 하드웨어 버퍼를 최적화하여 이러한 프레임 손실을 해결할 수 있습니다. 다른 방법으로는 애플리케이션의 소켓 사용을 최적화할 수 있습니다. 이를 위해 애플리케이션이 단일 소켓을 제어하도록 설정하고 수신된 네트워크 데이터를 신속하게 다른 사용자 공간 프로세스에 배포합니다.

[4] CPU와 NIC 간의 캐시 친화도를 보장하는 것은 동일한 L2 캐시를 공유하도록 설정하는 것을 의미합니다. 보다 자세한 내용은 8.3절. “패킷 수신 개요 ”에서 참조하십시오.

[5] 8.3절. “패킷 수신 개요 ”에는 네트워크 스택에서 병목 현상이 발생하기 쉬운 영역을 표시하고 매핑하는데 도움이 되는 패킷 전송에 대한 개요가 들어 있습니다.

[6] 장치 무게는 `/proc/sys/net/core/dev_weight`를 통해 제어됩니다. 장치 무게와 조정의 의미에 대한 자세한 내용은 8.4.1절. “NIC 하드웨어 버퍼 ”에서 참조하십시오.

부록 A. 고친 과정

고침 4.0-22.2.400 Rebuild with publican 4.0.0	2013-10-31	Rüdiger Landmann
고침 4.0-22.2 한국어 번역 완료	Thu Jun 27 2013	Eun-Ju Kim
고침 4.0-22.1 XML 소스 4.0-22 버전과 번역 파일을 동기화	Thu Apr 18 2013	Chester Cheng
고침 4.0-22 Red Hat Enterprise Linux 6.4 공개	Fri Feb 15 2013	Laura Bailey
고침 4.0-19 일관성과 관련하여 약간의 수정 (BZ#868404).	Wed Jan 16 2013	Laura Bailey
고침 4.0-18 Red Hat Enterprise Linux 6.4 베타 버전 공개	Tue Nov 27 2012	Laura Bailey
고침 4.0-17 re. numad 섹션에 SME 피드백 추가 (BZ#868404).	Mon Nov 19 2012	Laura Bailey
고침 4.0-16 numad에 초안을 추가 (BZ#868404).	Thu Nov 08 2012	Laura Bailey
고침 4.0-15 블록 삭제 설명에 SME 피드백을 추가하여 이 섹션을 마운트 옵션으로 이동 (BZ#852990). 성능 프로파일 설명을 업데이트 (BZ#858220).	Wed Oct 17 2012	Laura Bailey
고침 4.0-13 성능 프로파일 설명을 업데이트 (BZ#858220).	Wed Oct 17 2012	Laura Bailey
고침 4.0-12 문서 검색을 개선 (BZ#854082). file-max 의 정의를 수정 (BZ#854094). threads-max 의 정의를 수정 (BZ#856861).	Tue Oct 16 2012	Laura Bailey
고침 4.0-9 파일 시스템 부분에 FSTRIM 권장 사항을 추가 (BZ#852990). 고객의 의견에 따라 threads-max 매개 변수의 설명을 업데이트 (BZ#856861). GFS2 조각 모음 관리 개선에 대한 유의 사항을 업데이트 (BZ#857782).	Tue Oct 9 2012	Laura Bailey
고침 4.0-6 numastat 유틸리티에 대한 새로운 섹션을 추가 (BZ#853274).	Thu Oct 4 2012	Laura Bailey
고침 4.0-3 새로운 성능에 대한 주의 사항을 추가 (BZ#854082). file-max 매개 변수에 대한 설명을 수정 (BZ#854094).	Tue Sep 18 2012	Laura Bailey
고침 4.0-2 BTRFS 섹션과 파일 시스템의 기본적인 소개를 추가 (BZ#852978). GDB와의 Valgrind 통합을 기록 (BZ#853279).	Mon Sep 10 2012	Laura Bailey
고침 3.0-15 tuned-adm 프로파일의 업데이트된 설명을 추가 (BZ#803552).	Thursday March 22 2012	Laura Bailey

고침 3.0-10

Friday March 02 2012

Laura Bailey

threads-max 및 file-max 매개 변수 설명을 업데이트 ([BZ#752825](#)).
slice_idle 매개 변수 기본값을 업데이트 ([BZ#785054](#)).

고침 3.0-8

Thursday February 02 2012

Laura Bailey

[4.1.2절](#). “CPU 성능 튜닝 ” 에 numactl로 메모리 할당, CPU 바인딩, 작업 세트에 대한 상세 설명을 추가 및 재구성 ([BZ#639784](#)).
내부 링크를 수정 ([BZ#786099](#)).

고침 3.0-5

Tuesday January 17 2012

Laura Bailey

[5.3절](#). “프로파일 메모리 사용에 Valgrind 사용 ” 를 약간 수정 ([BZ#639793](#)).

고침 3.0-3

Wednesday January 11 2012

Laura Bailey

내부 및 외부 하이퍼링크간의 일관성을 확인 ([BZ#752796](#)).
[5.3절](#). “프로파일 메모리 사용에 Valgrind 사용 ” 추가 ([BZ#639793](#)).
[4.1.2절](#). “CPU 성능 튜닝 ” 추가 및 [4장](#). CPU 재구성 ([BZ#639784](#)).

고침 1.0-0

Friday December 02 2011

Laura Bailey

Red Hat Enterprise Linux 6.2 GA 릴리즈