Journal of The Korea Society of Computer and Information
Vol. 22 No. 6, pp. 9-16, June 2017

www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.06.009

Application Characteristic-based Divided Scheduling for Multicore Systems

Jung Kyu Park*, Jaeho Kim**

Abstract

In this paper, we proposed a novel user-level scheduling scheme that monitors applications characteristics on-line using PMU and allocates applications into cpu cores. We utilize PMU (Performance Monitoring Unit) to analyze which shared resource has the strongest relation with the influence. Using the proposed scheduling method, it is possible to reduce the contention of shared resources. The key idea of this scheme is separating high-influential applications into different processors. The evaluation results have shown that the proposed scheduling scheme can enhance the performance up to 12% for a 8 core system and up to 25% for a 28 core system, respectively.

▸Keyword: NUMA, CPU, Performance Monitoring Unit, LLC

I. Introduction

최근 시스템의 성능향상을 위해서 CPU의 처리속도 향상 이외에 시스템의 병렬성을 향상시키는 방향으로 프로세서의 코어 개수를 증가시키는 방안이 대두되고 있다. 이렇게 병렬성을 향상시켜, 각각의 코어에 다른 응용들을 동시에 수행시키거나 한 가지 응용을 각각의 코어에 분담시켜 시스템의 성능향상을 기대할 수 있다 [1].

예를 들어 Xeon E5-2697 프로세서와 AMD Opteron 프로세서는 각각 14, 16개의 코어를 가지고 있고, 이는 14, 16개의 응용을 동시에 처리할 수 있다. 각각의 프로세서에 대한 코어 개수는 20개가 넘지 않지만 최근 시스템들은 두 개 이상의 프로세서로 이루어져 있는데, IBM x3850 시스템의 경우 8개의 코어를 가진 프로세서가 8개, AMD Bulldozer는 16개의 코어를 가진 프로세서가 4개로 구성되어 두 시스템 모두 총 64개의 코어를 가지고 있다 [2,3].

이러한 매니코어환경에서 코어들은 시스템의 자원을 공유하고 있어 공유자원에 대한 경쟁을 야기하고, 경쟁으로 인해 코어들이 응용을 수행하는데 성능하락이 일어난다. 코어 간 공유자원 경쟁을 고려하여 태스크를 스케줄링하는 방법으로 시스템의 전체 성능향상을 위한 연구를 수행하였다 [4-7].

이를 위해서 메모리로 데이터 접근을 빈번히 하여 메모리 대역폭이 높고, 워크로드 수행 패턴이 비교적 일정한 워크로드 11개를 선정하여 상호간의 영향력을 파악하였으며, 이 결과를 토대로 프로세서에서 제공하는 성능 모니터링을 위한 레지스터인 PMU(Performance Monitoring Unit)를 이용하여 상호영향력들에 대한 원인을 분석하였다 [3,8-11]. 분석결과로 응용을 수행중인 코어의 특징과 상호영향력을 대표하는 PMU 레지스터 들을 선정하였으며, 이를 기반으로 NUMA 구조에서 공유자원을 효율적으로 쓰기위한 최적의 스케줄링 기법을 제안하였다. 제안한 기법을 검증하기 위해 두 가지 시스템 환경에서 Linux 4.2 기본 스케줄링 정책과 비교하여 실험을 수행하였다.

논문의 구성은 다음과 같다. 2장에서는 본 논문의 주요 개념인 CPU 코어의 자원관리에 살펴본다. 3장에서는 본 논문에서 제안하는 어플리케이션 속성 기반 스케줄링 알고리즘에 대해서 살펴본다. 4장에서는 제안하는 알고리즘을 리눅스에 구현하고 테스트한 결과를 설명하고 마지막으로 5장에서 결론을 맺는다.

• First Author: Jung Kyu Park, Corresponding Author: Jaeho Kim
*Jung Kyu Park (smartjkpark@swu.ac.kr), Dept. of Digital Media Design and Applications, Seoul Women's University
**Jaeho Kim (jh.kim@unist.ac.kr), School of Electrical and Computer Engineering, UNIST
• Received: 2017. 05. 12, Revised: 2017. 06. 04, Accepted: 2017. 06. 19.

II. Related Works

1. Independent resources between cores

다른 코어들과는 별개로 각각에 코어가 독립적으로 활용할 수 있는 자원이다. Intel 프로세서는 L1, L2 캐시를 각각의 코어에 대해서 독립적으로 사용하도록 할당을 해주지만, AMD 프로세서에서는 L1 캐시까지 할당해주고 있다. 이와 같이 코어의 독립자원과 공유자원의 범위와 크기가 프로세서 마다 다르지만 어느 환경에서라도 공유자원이 존재한다면, 코어 간 공유하는 자원의 양의 따라서 그 정도는 다르겠지만 코어들의 자원경쟁은 불가피하다. 예를 들어 L1, L2 캐시와 같은 코어에 대한 독립자원이 많고 공유자원이 적을수록 공유자원 경쟁에 의한 영향을 적게 받지만, 대신 공유자원에 대한 경쟁이 크지 않을 때는 수행에 있어서 시스템 자원이용에 한계가 정해진다. 하지만 독립자원이 적고 공유자원이 커질수록 공유자원 경쟁 심화에 의해 코어들의 성능하락이 커지지만, 경쟁이 약해질수록 프로세서의 가용자원을 최대로 쓸 수 있다는 장점이 있다 [4-7].

Intel Xeon 프로세서의 경우 각각의 코어에 L1 캐시(32KB), L2 캐시(256KB)까지 독립적으로 할당되지만, AMD Opteron 프로세서의 경우 각각의 코어에 L1캐시(64KB)까지만 할당되고 두 개의 코어가 L2캐시(2MB)를 공유하고있다. AMD Opteron 코어들은 LLC 경쟁이 없을 때 독립자원 L1, L2 캐시를 포함해 LLC를 최대 16MB를 이용하지만, Intel Xeon 프로세서는 공유되는 LLC 크기가 35 MB이기 때문에 경쟁이 없을 때 LLC를 독립적으로 이용할 수 있다 [2,3].

2. Shared resources between cores

2.1 Shcard Cache

프로세서 내에서의 캐시는 크기가 다른 캐시가 Level 1, Level 2와 같이 구분되어 계층적 구조를 가진다. LLC(Last Level Cache)는 이름 그대로 마지막 Level 캐시로 코어들 사이에서 공유되는 캐시자원이다. 최근 시스템들은 Level 3 캐시가 대체적으로 Last Level이며 코어의 개수가 늘어날수록 LLC의 크기도 늘어난다. 프로세서에서 수행되는 코어들의 지역성이 적어 캐시적중률이 낮을수록 LLC 공간이 부족해 LLC 경쟁이 심화된다 [7,8].

예를 들어 수행중인 코어 A는 접근하는 데이터들의 지역성이 낮아 캐시 적중률이 낮다면, 재사용 되지 않는 코어 A의 데이터들은 LLC에서 공간을 차지하고 다른 코어의 적중될 데이터들은 LLC에 적재되지 못하게 된다. 또는 데이터 접근 패턴이 stream패턴과 같은 LLC를 오염시키는 패턴이거나, 데이터 접근이 빈번하여 다른 코어들보다 캐시공간을 많이 차지하는 코어일 경우 LLC 경쟁을 심화시키고 자신의 성능도 크게 하락하게 된다.

2.2 Memory bandwidth

시스템에서 활용 가능한 메모리 대역폭은 한정적이지만 코어들이 데이터에 접근하기 위한 메모리에 대한 접근이 빈번할 경우 병목현상이 일어난다. Linux 4.2 기본 NUMA 정책에서는 Local/Remote 메모리 접근 패널티 때문에 코어의 메모리 활용은 프로세서가 가진 Local Memory를 용량이 부족할 때까지 활용하고, Local Memory의 용량이 부족해 질 때 Remote Memory를 활용하게 되어있다 [10,11]. 그리고 페이지 단위로 주기적으로 메모리 마이그레이션을 수행하며, 페이지 데이터들은 자주 접근하는 코어가 있는 프로세서의 메모리로 마이그레이션 되도록 되어있다. 코어는 Remote 메모리로 접근빈도가 Local 메모리 접근빈도보다 많을 때 Remote 메모리를 가진 프로세서로 마이그레이션 되도록 되어있다.

최악의 경우는 프로세서 A의 코어들이 메모리 접근을 Local로 빈번히 하고 프로세서 B의 코어들이 프로세서 A의 메모리에 Remote로 빈번히 접근하여 프로세서 A의 가용 메모리 대역폭이 초과되는 것이다. 프로세서 B의 메모리 대역폭은 여유롭지만, 프로세서 A의 메모리 대역폭에 대해서는 병목현상이 일어나는 경우이다. 이러한 상황을 완화하기 위해서 NUMA 환경에서 프로세서들마다 코어들의 메모리 대역폭에 대한 수요를 밸런싱 하도록 많은 연구가 진행되었다.

2.3 Interconnection

NUMA 구조에서 하나의 프로세서와 다른 프로세서가 통신을 하기위한 대역폭이다. 주로 Remote 캐시라인에 대한 상태변경 요청이나, 데이터 요청, Remote Memory 데이터에 대한 요청이 있는데, Intel의 QPI(Quick Path Interconnect) 와 AMD's HyperTransport 기술이 이에 해당된다. 메모리 대역폭과 마찬가지로 코어들이 Remote 프로세서에 접근하는 빈도가 높을수록 Interconnection 대역폭에 대한 수요가 증가해 병목현상을 일으킬 수 있다[3,8-11]

3. Linux Scheduling Policy

Linux 4.2 버전의 기본 스케줄러의 코어 스케줄링은 LLC 밸런싱을 위해서 응용들의 데이터 접근 지역성을 참고하여 프로세서 간 응용들을 스케줄링 하거나, NUMA 구조의 Local/Remote 메모리 접근 페널티를 최소화하기 위해서 응용의 Local/Remote Memory 접근 빈도에 따라서 프로세서 간 응용 스케줄링을 수행한다. 이는 Remote Memory 접근 빈도가 Local Memory 접근 빈도보다 많으면, 접근하는 Remote Memory가 있는 프로세서의 코어로 응용을 스케줄링 해주는 방식이다 [12-14].

III. Application Characteristic-based Scheduling Algorithm

1. Influence between applications

그림 1은 리눅스 버전 4.2, 프로세서가 각각 4개의 코어를

가진 두 개의 Intel Xeon x3650 프로세서로 이루어진 시스템 이서 수행한 실험 결과이다. 프로세서당 메모리는 32GB의 크기를 가지고 있고, 8MB의 LLC, 32KB의 L1 Instruction/Data 캐시, 256KB의 L2캐시 환경이다. 워크로드들은 SPEC CPU 2006, PARSEC, NPB 워크로드 48개 중에서 선정을 하였다. 충분한 경쟁 상태를 구성하기 위해서, 데이터 접근을 빈번히 하여 메모리 대역폭이 높고 수행 패턴이 비교적 일정한 워크로드 11개를 선정하였다.

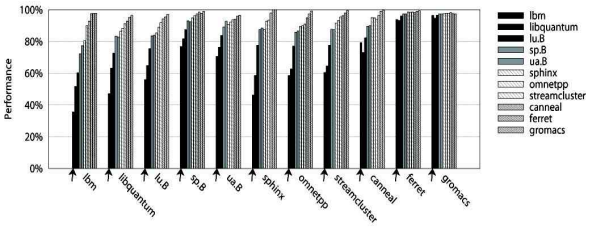

Fig. 1. Test Results of Tnvme

실험방법은 같은 프로세서에서 다른 코어에 두 개의 워크로드를 수행시켰으며 두 개의 워크로드가 모두 끝날 때까지 수행시켰다. 워크로드의 성능은 혼자서 수행될 때의 완료시간을 기준으로 하였으며 NUMA library를 통해서 리눅스의 NUMA Aware Load Balancer가 스케줄링하여 워크로드가 수행 되는 코어를 다른 코어로 매핑하지 못하도록 설정하였다.

x축 각각의 워크로드들은 다른 워크로드들에 의해 성능에 영향을 받는 정도를 볼 수 있다. x축 워크로드 하나에는 11개의 y축이 그려져 있는데, 이는 범례에 표시된 워크로드와 함께 수행된 결과를 뜻한다. y축의 순서와 범례에 표시된 워크로드의 순서는 같다. 다른 워크로드들의 성능에 영향을 끼치는 정도를 관찰하려면 각각의 x축 워크로드에서 범례를 참고하여 동일한 순번의 y축을 확인하면 된다. 예로 그림에 표시된 화살표가 있는데, 이는 lbm이 다른 워크로드들에게 얼마나 영향을 주는가를 보여주고 있다.

그림 1을 참고하면 상호간에 영향력에 따라 워크로드별로 그룹을 나눌 수 있다.

1) Group 1

Group 1에 해당하는 워크로드들은 lbm, libquantum, lu.B 이다. 이들은 공통적으로 다른 워크로드들의 성능에 큰 영향을 끼치고 있다. 이들은 다른 워크로드들에게 영향을 많이 주고 다른 워크로드들에 의해서 영향을 크게 받고 있다.

2) Group 2

Group 2에 해당하는 워크로드들은 sphinx, omnetpp, streamcluster 이다. 이들의 상호영향력 특징은 비교적 다른 워크로드들의 성능을 하락시키지 않지만, 자신의 성능은 Group 1들에 의해서 성능이 크게 하락한다.

3) Group 3

Group 3에 해당하는 워크로드들은 sp.B, ua.B, canneal, ferret, gromacs 이다. 이들은 어느 워크로드와 공유자원에 대해서 경쟁을 하여도 다른 워크로드들의 성능을 크게 하락시키지 않고, 자신의 성능도 거의 영향을 받지 않는다. sp.B와 ua.B는 Group 2 워크로드들보다 다른 워크로드의 성능을 하락시키는 특징을 가지고 있지만, Group 1나 Group 2 워크로드들에 비해서 자신의 성능에 영향을 별로 받지 않고 있다. 그래서 sp.B와 ua.B를 다른 그룹으로 따로 구분 지으려 했지만 워크로드들을 따로 구별하는 것의 중요성이 크지 않고, 응용그룹의 식별을 단순화하기 위해서 Group 3로 구분 하였다. 하지만 그 원인분석을 위한 연구를 수행하였다.

Table 1. Selected PMU events for application identification

Resource	Events
LLC	UNC_LLC_HITS.ANY number of LLC cache hits
	UNC_LLC_MISS.ANY number of LLC cache misses
Memory	UNC_IMC_NORMAL_READS.ANY number of read requests to IMC
	UNC_IMC_WRITE.FULL.ANY number of write requests to IMC
	OFFCORE_REQUEST_BUFFER_FULL number of requests bolocked due to buffer full

2. Analyze the properties of an application

2.1 PMU

코어들의 자원의 사용량 및 수행특징을 모니터링하기 위하여 하드웨어 수준의 성능 계수 PMU (Performance Counter Unit) 정보를 이용하였다. Intel과 AMD 사는 마이크로아키텍처 자원 사용량의 정보 제공을 위하여 특수 레지스터에 각 자원의 사용량을 카운팅 한다. 측정할 수 있는 자원 사용량은 대표적으로 클럭 사이클, 명령어 수행 횟수, 캐시 미스, 메모리 접근등과 같은 정보를 측정할 수 있다. 해당 레지스터에 대한 접근은 RDPMC (Read Performance Monitoring Counter) 명령어를 통해 측정할 수 있는데, RDPMC는 EAX, EDX, ECX 3개의 레지스터를 사용하여 자원 사용량을 측정한다.

실험에 사용된 Intel Xeon x5570 프로세서는 PerfEvtSelX MSR (Machine Specific Register)이 존재하며, EVTSEL의 8비트와 EVTMSK의 8비트를 설정하여 GQ 사용량, LLC 접근 횟수와 같은 특정 이벤트들을 지정할 수 있다. 특정 이벤트를 설정하기 위해 ECX 레지스터를 사용하고, 자원 사용량은 EAX와 EDX 2개의 레지스터에 저장된다. 자원 활용률을 측정하기 위한 비용은 하드웨어 레지스터를 읽는 수준으로, 큰 오버헤드가 없이 빠르게 모니터링을 수행할 수 있다. 측정결과 0.1%에 가까운 오버헤드를 보였고, 이를 활용하여 코어들의 수행 특징을 모니터링 하였다.

2.2 Realtime monitoring

PMU의 큰 특징 중 하나인 오버헤드가 거의 없다는 점을 활

용하여 본 논문에서 코어에 대한 모니터링과 식별을 응용들이 수행되고 있는 실시간으로 수행한다. 이로 인해 응용의 속성이 수행 중에 변화되어도 모니터링을 통해 다른 응용그룹으로 식별할 수 있다.

3. PMU events for group identification

표 1은 여러 가지 PMU 이벤트들 중에서 응용 프로그램들을 상호영향력 기반으로 식별하기 위한 이벤트들을 선정한 결과이다. 위 표에 표시되어있는 여러 개의 PMU 이벤트들을 모니터링하여 응용그룹들을 순차적으로 식별한다.

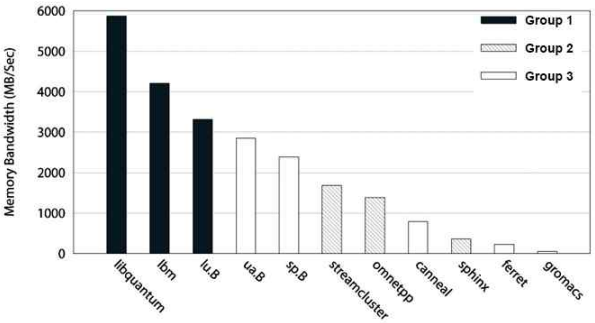

Fig. 2. Memory bandwidth usage

3.1 Memory bandwidth

코어는 수행 중에 데이터 접근이 빈번할수록 메모리에 더욱더 접근을 하게 되는데, 이는 메모리 대역폭 사용량이 클수록 공유자원 사용을 더 많이 한다고 볼 수 있다. 그림 2는 선정한 11개의 워크로드들의 메모리 대역폭 사용량을 나타내고 있다. 클럭이 2.93GHz인 프로세서에 워크로드를 홀로 수행시켜 64*(UNC_IMC_NORMAL_READS+ UNC_IMC_WRITES.FULL.ANY)*2.93*1000000000/ cpu_cycle/1000000 로 구하였다. 코어는 작업 수행 시 데이터 접근을 위해서 캐시를 접근한다. 만약 L1, L2 캐시를 접근해도 접근할 데이터가 적재되어있지 않았을 경우, 공유자원인 LLC를 접근한다. 만약 이때도 접근할 데이터가 적재되어 있지 않을 경우, 마지막으로 메모리의 해당 주소를 접근한다. 이는 접근할 데이터의 지역성이 낮아 공유자원인 LLC를 오염시키는 결과를 가져오고, 한정된 메모리 대역폭을 사용하게 된다. 또는 다른 워크로드들 보다 상대적으로 데이터 접근을 많이 할 경우도 있다. 그림 1과 그림 2를 참고하면 Group 1 워크로드들이 높은 메모리 대역폭 사용량을 가지고 있고, 다른 워크로드들의 성능에 피해를 많이 끼치고 있다.

Group 3 워크로드인 sp.B 와 ua.B 들은 Group 1 워크로드들에 이어 높은 메모리 대역폭 사용량을 보이고 있고, 그림 2를 참고하면 다른 워크로드의 성능에 끼치는 영향력도 Group 1 다음인 것을 확인할 수 있다. 그리고 2GB/Sec 메모리 대역폭 이하의 워크로드들은 Group 2 또는 나머지 Group 3들이며, 이들은 메모리 대역폭 사용량이 적고 그림1을 참고하면 다른 워크로드들에게 주는 영향력도 적은 것을 보인다.

3.2 Super Queue Full Count

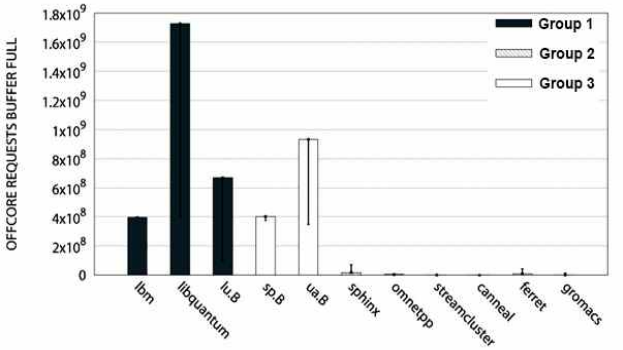

Fig. 3. Event of Offcore Requests Buffer Full Count

PMU를 이용해서 코어별로 메모리 대역폭 사용량을 모니터링 할 수 있으면 좋겠지만, 현재 PMU에서는 코어별로 메모리 대역폭 사용량을 모니터링 할 수 없다. 그래서 그 방안으로 PMU를 이용해 코어별로 Offcore Request Buffer Full 이벤트를 모니터링하기로 하였다. 그림 3을 참고하면, 코어별로 Super Queue라는 Queue가 존재한다. 이 Queue는 코어의 공유자원에 대한 Reuquest 들이 채워진다. 코어가 LLC에 대한 Cache Line 접근 요청을 하여 Reuqest를 보내면 코어에 해당하는 Super Queue에 해당 Request가 채워진다. Offcore Request Buffer Full 이벤트는 코어가 Request를 요청했지만 Super Queue가 Request들로 가득 채워져 있을 때 카운트가 올라가게 된다.

이 이벤트를 모니터링 한다면 코어마다 정확한 메모리 대역폭 사용량을 알 수 없지만, 대신 코어의 공유자원에대한 수행특징을 알 수 있다. Super Queue가 가득 차있는 상태에서 코어가 Cache Line Request를 지속적으로 요청했다는 것은 공유자원을 공격적으로 사용한다는 것을 뜻한다. 그림 2을 참고하면 Group 1 워크로드들을 구분할 수 있을 뿐만 아니라, 다른 워크로드들에게 영향을 주는 워크로드들과 그렇지 않은 워크로드간의 PMU 값 수치가 극단적으로 나누어진다. 그러므로 이 Offcore Request Buffer Full 이벤트를 다른 워크로드 들에게 높은 영향력을 가진 코어와 그렇지 않은 코어들 사이를 구분하기 위한 PMU 이벤트로 선정하였다.

3.3 Shared cache reference

LLC Reference 이벤트는 코어가 다른 워크로드에게 얼마나 영향을 받는지를 알 수 있다. LLC Reference 값은 표 2에 보이는 UNC_LLC_HITS.ANY(LLC 적중) 값과 UNC_LLC_MISS.ANY(LLC 미스)값을 더한 값과 같다. 이는 코어의 성능이 공유자원인 LLC에 얼마나 의존적인지 알 수 있는데, 그림 4를 보면 Group 2 워크로드들은 다른 워크로드들보다 훨씬 높은 LLC Reference 수치를 가지고 있는 것을 확인할 수 있다.

그리고 Offcore Request Buffer Full PMU값이 낮고 영향을 받지 않는 Group 3 워크로드들인 canneal, ferret,

gromacs 워크로드들은 다른 워크로드들보다 낮은 LLC Reference 수치를 가진다. 먼저 Offcore Request Buffer Full 이벤트를 통해 다른 워크로드들에게 영향을 주는 Group 1 워크로드들을 식별한 후, 다른 워크로드들에게 영향을 받는 Group 2 워크로드들과 영향을 거의 받지않는 Group 3 워크로드들의 일부를 이 LLC Reference 이벤트를 통해서 식별할 수 있다. 기존연구들에서는 일반적으로 코어의 LLC Hit Ratio를 통해서 코어의 데이터 지역성을 확인하고 높은 LLC Hit Ratio를 가진 워크로드와 낮은 LLC Hit Ratio를 가진 워크로드를 같은 프로세서에 매핑함으로써 LLC 자원에 대한 밸런싱을 맞추지만, 본 논문에서는 워크로드의 LLC에 얼마나 의존적인가에 대한 동작특징을 모니터링하기 위해서 가변적인 LLC Hit Ratio를 보지않고 코어의 LLC Reference를 모니터링한다.

예를 들어 Group 1 워크로드의 경우 경쟁이 없는 상태에서는 70%의 높은 LLC Hit Ratio를 기록하지만, 1개의 다른 워크로드와 공유자원 경쟁을 한다면 20% 근처까지 떨어진다. 다른 워크로드도 마찬가지로 공유자원 경쟁을 하게 된다면 그림 3의 점 (0,0)으로 수렴하게 되어 구분을 할 수 없게 된다. 반면 LLC Reference의 경우 코어가 수행하는 워크로드의 고유속성이므로 경쟁이 심해져도 워크로드간의 차이는 유지된다.

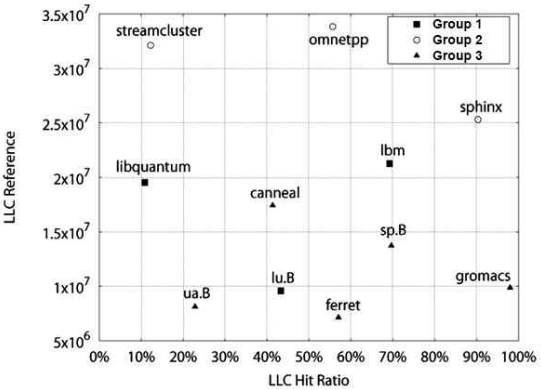

Fig. 4. Event count of LLC reference

3.4 Prefetch

캐시를 쓰는 이유는 코어가 데이터접근을위해 메모리에 접근하는 비용을 줄이기 위해서이다. 이러한 장점을 더욱더 활용하기 위해서 최근 프로세서들은 Hardware Prefetch를 지원한다. Hardware Prefetch는 코어가 연속적인 데이터 또는 규칙적인 데이터 접근을 하드웨어에서 인식하여 코어가 다음에 접근할 데이터들을 예측한다. 그리고 코어가 메모리의 데이터에 접근하기 전에 미리 메모리에서 캐시로 해당 데이터들을 적재한다.

예를 들어서 L2 Prefetch는 메모리로부터 L2 또는 LLC로 다음 접근이 예상되는 데이터들을 적재시키는 기술이다. 코어가 L1에서 캐시미스가 발생할 경우 L2의 캐시라인에서 데이터를 찾는다. 이때 L2캐시의 N번째 캐시라인에서 미스가 나고, 다음에도 코어가 L2 캐시의 N+1 번째 캐시라인에서 캐시미스가 발생하면 L2 Hardware Prefetcher는 해당 데이터의 다음 데이터들이 또 참조될 것으로 예측하고 메모리 주소를 참조하여 다음 데이터들을 L2 캐시로 미리 적재시킨다. 데이터들이 캐시에서 적중이 되면 코어가 메모리로 접근할 필요가 없어지고 빠른 작업수행을 할 수 있게 된다.

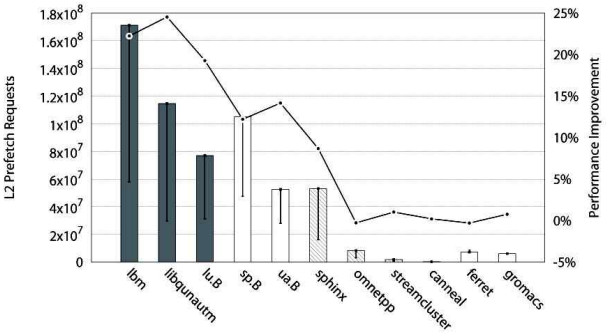

Fig. 5. Event count of L2 Prefetch Request

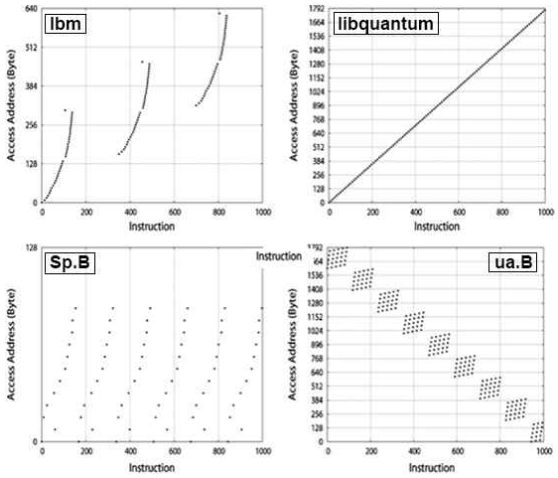

Fig. 6. Memory access addresses of Group 1 and Group 3

이 기술에 의해서 메모리에 접근이 빈번한 코어들은 큰 성능이득을 보지만, 응용들의 상호영향력의 원인은 코어가 데이터를 접근하면서 L2 Hardware Prefetch를 활용하는데 있다. 공유자원의 하나인 LLC를 오염시키는 요인은 코어가 연속적인 데이터를 접근하는 요인이 있다. 코어가 이러한 데이터를 접근하고 있다는 정보를 코어가 얼마나 L2 Hardware Prefetch를 활용하는지를 확인하면 얻을 수 있다.

코어가 접근하는 데이터를 예측하고 메모리에서 캐시로 적재시키기 위해서 Hardware Prefetcher가 L2 Request를 발생시킨다. 그림 5를 보면 워크로드들이 L2 Prefetch Request를 얼마나 요청했는가에 대한 PMU 이벤트를 볼 수 있다. 그림에서 볼 수 있듯이 Group 1 워크로드들은 높은 L2 Prefetch Request를 발생시켰으며 다른 워크로드들에게 영향을 끼치는 Group 3 워크로드인 sp.B, ua.B도 높은 메모리 대역폭 사용량과 함께 높은 L2 Prefetch Request를 발생시키고 있다. Group 2 워크로드 중 하나인 sphinx 워크로드는 높은 메모리 대역폭을 사용하고 있지는 않고 Offcore Request Buffer Full 이벤트 수치가 높지 않지만, L2 Prefetch를

다른 Group 2와 비교해서 많이 활용하고 있어 다른 Group 2들보다 다른 워크로드들에게 영향을 조금 더 주고 있다.

다음으로 메모리 대역폭 사용량이 많고 L2 Prefetch Request 발생도 많은 Group 1 워크로드들과 Group 3 워크로드의 차이를 알기위해 offline으로 Group 1워크로드인 lbm, liquantum과 Group 3 워크로드인 Sp.B, ua.B 의 워킹셋을 확인하였다. 워킹셋을 확인하기 위해서 PIN Tool을 이용하여 각 워크로드가 수행 중 접근하는 메모리 주소를 추출하였다. 캐시라인 크기가 보통 64 Byte 인 것을 고려해서 그림 6을 보면 Group 1 워크로드인 lbm은 한 개의 워킹셋이 최소 5개의 캐시라인인 것에 반해 Group 3 워크로드인 Sp.B는 2개의 캐시라인으로 1개의 워킹셋을 끝낼 수 있다. 또한 libquantum은 캐시라인을 재사용하지 않는 일련의 stream 데이터 패턴이지만, ua.B의 경우는 stream 패턴이 캐시라인을 재사용하고 있다. 이와 같이 워킹셋이 큰 접근 데이터 패턴과 워킹셋이 작은 접근 데이터 패턴이 경쟁상태에서 방해가 되었을 때 성능이 하락하는 정도는 다르게 된다.

예를 들어 하나의 작업을 처리하는데 10의 데이터가 필요하다면 5의 단위로 데이터를 prefetch할 경우에서 수행에 방해받는 것보다 2의 단위로 데이터를 prefetch할 경우 수행에 방해받을 경우가 피해가 더 적고 캐시를 오염시키는 정도도 더 적다.

3.5 Classification and placement of tasks based on characteristics

위 실험 및 분석을 토대로 PMU를 이용하여 실시간으로 응용들을 Group 1, Group 2, Group 3로 나누어 최적화된 배치를 한다. 먼저 각각의 코어에서 수행하고 있는 응용을 식별을 하기 위해서 Offcore Reqeust Buffer Full 이벤트 수치를 활용하여 Group 1 응용그룹으로 추정되는 응용들을 구분하고, 나머지 워크로드들에 대해서 LLC Reference 이벤트 수치를 활용하여 Group 2 응용그룹들을 구분한다. 그 외의 응용들은 Group 3 응용그룹으로 구분한다.

식별 시 PMU 이벤트 수치에 대해서는 Threshold Value를 활용하며 이 값은 위에서 분석한 PMU 이벤트 수치들을 기반으로 정하였다. 이외에 다른 방법으로는 응용식별에 Threshold Value를 적용시키지 않고 수행되고 있는 응용들에 대해서 상대적으로 Group 1, Group 2, Group 3 그룹으로 구분하는 것이 있다.

예를 들어 9개의 응용이 수행 중이라면, Offcore Request Buffer Full 이벤트 수치가 높은 응용 3개를 Group 1 응용그룹, 나머지에서 LLC Reference 이벤트 수치가 높은 3개의 Group 2 응용그룹, 나머지 3개의 응용을 Group 3 응용그룹으로 구분하는 것이다.

식별한 응용들을 코어에 배치시키는 매커니즘은 간단한데, 이는 Group 1 응용들을 Group 2 응용들과 격리시켜 배치하는 것이다. 이는 Group 1 워크로드들이 각기 다른 프로세서에 흩어져 곳곳에서 다른 워크로드들의 성능을 하락시키는 것보다 Group 1 워크로드들을 모으고, 성능에 영향을 크게 받는 Group 2 워크로드들을 보호하는 방식이다. 이와 반대인 최악의 경우는 Group 1들과 Group 2들이 골고루 퍼져 모든 응용들이 경쟁에 의해 성능 하락이 일어날 때이다. 이러한 매커니즘으로 Group 1 워크로드들이 Default 보다 훨씬 느려질 것이라 예상하였지만, Group 2들이 공유자원에 대한 경쟁으로부터 자유로워져 빠르게 끝난 후 Group 2들이 배치되었던 비어있는 코어에 Group 1들이 나누어져 경쟁상태가 완화되어 Default와 비슷한 성능, 혹은 더 빠른 성능을 보였다.

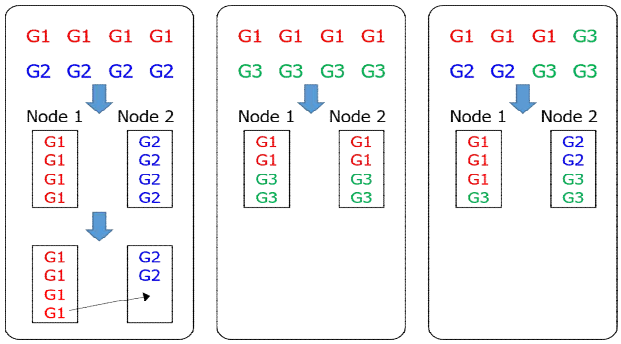

Fig. 7. Memory access addresses of Group 1 and Group 3

하지만 이 Group 2와 Group 1 워크로드들을 격리시키는 방법의 성능향상 전제조건은 공유자원에대해서 충분히 경쟁이 일어날 때이다. 예를 들어서 Group 1 워크로드의 숫자가 적어서 퍼져있을 때 공유자원에 대한 경쟁이 거의 일어나지 않았을 때 Group 1 워크로드들을 모아서 Group 1 워크로드가 모인 프로세서에 괜한 공유자원 경쟁을 일으키는 경우가 있다.

그림 7로 알고리즘의 의도를 간단하게 나타낼 수 있는데, 가장 기본적으로 첫 번째 박스에 있는 경우와 같을 경우 Group 1 응용들과 Group 2 응용들을 나누어 각각의 프로세서에 배치한다. 두 번째 박스와 같이 Group 1 워크로드들만 있을 경우 Group 1 워크로드들을 나누어 공유자원에 대한 경쟁을 최소화하고, Group 3 워크로드들을 빈 코어에 나누어 배치한다. 세 번째 박스의 경우에는 Group 2 워크로드들을 Group 3 워크로드들과 함께 배치하여 보호하고 Group 1 워크로드들을 Group 2로부터 독립시켜 배치하는 경우이다.

IV. Evaluation

제안하는 알고리즘을 구현한 스케줄러를 유저레벨에서 구현하였다. 스케줄러는 주기적으로 PMU 이벤트에 대해서 10초를 주기로 모니터링을 하며, 원하는 코어에서 응용들을 수행시키기 위해 시스템 콜을 이용하였다. 그리고 쓰레드를 이용하는 응용들을 위해서 시스템 콜을 추가해 적용하였다. NUMA에 대한 설정이나 활용이 필요하여 NUMA 라이브러리를 이용하였으며, NUMA 환경에서 리눅스의 기본 메모리 할당정책인 Local Memory 우선 할당정책을 그대로 적용하였다. 응용들에 대한 응용그룹 식별을 위해서 PMU 이벤트에 대한 Threshold Value를 적용한 버전과 Threshold Value를 적용하지 않고 PMU 이벤트 값에 대한 내림차순을 이용해 상대적으로 응용그룹을 식별하는 버전을 구현하였다.

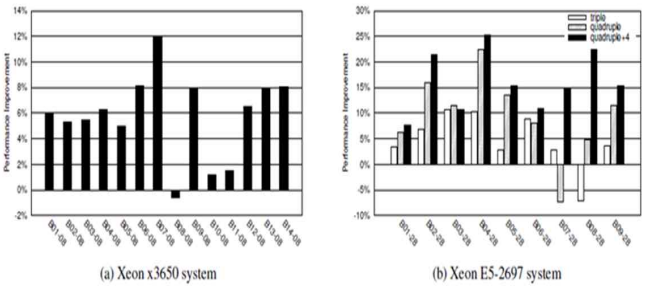

Fig. 8. Performance improvement using threshold value

Table 2. Experimental environment

System 1	- Intex Xeon 3650 (4 cores) × 2 CPU 32GB per CPU, 8MB LLC, 32KB L1 cache, 256KB L2 cache - Linux Kernel 4.2
System 2	- Intex Xeon E5-2697 (14 cores) × 2 CPU 32GB per CPU, 35MB LLC, 32KB L1 cache, 256KB L2 cache - Linux Kernel 4.2

구현한 유저 스케줄러와 리눅스 4.2 버전의 기본 스케줄러를 대조하기 위해서 워크로드 조합들을 이용하였다. Basic 워크로드 조합들은 위 분석들에 이용되어 이미 어느 응용그룹에 속하는지 알고 있는 워크로드들로 구성된 조합들이다. Extended 워크로드는 반면에 SPEC CPU2006, Parsec, NPB 워크로드 중 분석에 이용되지 않은 워크로드들을 포함시킨 워크로드 조합이다. Extended 워크로드 조합을 이루고 있는 워크로드들 또한 Basic 워크로드들을 이루고 있는 워크로드들과 같이 수행패턴이 비교적 일정한 워크로드들이다.

Basic 1~9, Extended 1~3 까지의 조합은 6개의 워크로드, Basic 10~14, Extended 4 까지는 8개의 워크로드로 이루어져 있다. 또한 실험에 사용한 장비는 표 2와 같다.

4.1 Application group identification using threshold value

그림 8은 Threshold Value를 활용한 유저 스케줄러 버전을 두 가지 환경에서 수행하여 리눅스 4.2 버전의 스케줄러에 비해 향상하는 성능을 나타낸다. x축의 의미는 B는 Basic 워크로드 조합을 의미하고 있고, 다음에 오는 01~14는 워크로드 조합이 몇 번째인지 의미한다. -08, -28은 수행된 환경을 의미하고 있다. 코어를 8개 가진 프로세서가 두 개인 환경 (a) 그림에서는 평균 5~6%의 성능향상과 함께 Basic 워크로드 7번이 12%의 최대 성능향상을 보이고 있다. 첫 번째 환경의 Basic의 8번째 워크로드는 2개의 Group 1와 6개의 Group 3로 충분한 공유자원 경쟁이 일어나지 않아 큰 성능향상을 보이지 않거나, 오히려 성능의 하락을 보이고 있다. 코어를 14개 가진 프로세서가 두 개인 환경 (b) 그림에서 triple은 6개 워크로드의 3배로 18개의 워크로드, quadruple은 6개 워크로드의 4배로 24개의 워크로드, quadruple+4는 워크로드의 4배에 임의의 4개 워크로드를 수행시킨 결과이다. (a) 환경보다 코어가 많아진 (b) 환경에서 최대 25%의 성능향상을 보이고 있으며, triple에서 quadruple+4로 수행되는 코어의 수가 많아질수록 성능이 향상하는 추세를 확인할 수 있다.

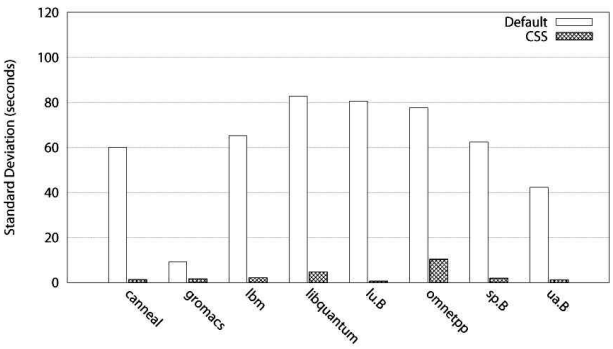

Fig. 9. Performance standard deviation for each B14-08 workload

4.2 Standard deviation of application performance

또한 구현한 유저 스케줄러는 각 워크로드에게 일정한 성능을 보장한다. 그림 9는 Basic 워크로드 조합 14을 각 프로세서마다 8개의 코어가 있는 두 개의 프로세서 환경에서 수행한 결과이다. 각각 워크로드 성능의 표준편차를 5회의 워크로드 조합 수행결과를 토대로 계산하였다. 기본 리눅스 스케줄러는 수행할 때마다 워크로드의 완료시간이 최소 10초에서 최대 81초 사이로 변동이 있는 반면, 구현한 유저 스케줄러는 워크로드 완료시간이 최소 2초에서 최대 11초의 변동이 있어 비교적 일정하여 예측 가능한 성능을 보여주고 있다.

V. Conclusions

본 논문에서는 공유자원 경쟁을 일으키는 대표적인 응용들에 대한 실험을 통해서 응용들의 수행특징을 분석하였다. 그리고 수행특징을 나타내는 PMU 이벤트인 Offcore Request Buffer Full, LLC Reference를 모니터링하여 응용의 상호영향력에 따라 Group 1, Group 2, Group 3로 응용들을 그룹화하여 식별하였다. Group 1와 Group 2를 격리시키는 유저 스케줄러를 통해 여러 가지 워크로드 조합에 대해 최대 25%의 성능향상을 보였고 각각의 워크로드들의 성능이 일정한 효과를 얻었다.

REFERENCES

[1] J. Rao, K. Wang, X. Zhou, and C. Xu, “Optimizing virtual machine scheduling in NUMA multicore systems,” in Proc. of the 19th International Symposium on High Performance Computer Architecture, Feb. 2013.

[2] D. Levinthal, “Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processor,” https://software.intel.com/, 2009.

[3] INTEL., “Intel 64 and IA-32 Architectures Optimization Reference Manual,” https://www.intel.com.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, “A case for NUMA-aware contention management on multicore systems,” in Proc. of the USENIX Annual Technical Conference, June, 2011.

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorava, “Addressing shared resource contention in multicore processors via scheduling,” in Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, Mar. 2010.

[6] R. Lachaize, B. Lepers, and V. Quema, “MemProf: a memory profiler for NUMA multicore systems,” in Proc. of the USENIX Annual Technical Conference, June, 2012.

[7] M. Dasthi, A. Fedorova, J. Funston, F. Gaud, R. Lachize, B. Lepers, V. Quema, and M. Roth, M, “Traffic management: A holistic approach to memory placement on NUMA systems,” in Proc. of the 18th international Conference on Architectural Support for Programming Languages and Operating Systems, Mar. 2013.

[8] B. Lepers, V. Quema, and A. Fedorova, “Thread and memory placement on NUMA systems: Asymmetry matters,” in Porc. of the USENIX Annual Technical Conference, June. 2015.

[9] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Rroscoe, A. Schpbach, and A. Singhania, “The multikernel: A new OS architecture for scalable multicore systems,” in Proc. of the 22th ACM Symposium on Operating Systems Principles, Oct. 2009.

[10] M. Liu, and T. Li, “Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads,” in Proc. of the 41th International Symposium on Computer Architecture, June. 2014.

[11] Z. Majo, and T. R. Gross, “Memory system performance in a NUMA multicore multiprocessor,” in Proc. of the 4th Annual International Systems and Storage Conference, May, 2011.

[12] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesperrev, F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of Linux scalability to many cores,” in Proc. of the 9th Symposium on Operating Systems Design and Implementation Oct, 2010.

[13] M. V. Weaver, “Linux perf event features and overhead,“ in the 2nd International Workshop on Performance Analysis of Workload Optimized Systems, Apr. 2013.

[14] D. Eklov, N. Nikoleris, D. Black-Schaffer, D, and Hagersten, “Bandwidth bandit: Quantitative characterization of memory contention,” in Proc. of the 11th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Feb. 2013.

[15] G. W. Lee, “Energy-Efficient Fault-Tolerant Scheduling based on Duplicated Executions for Real-Time Tasks on Multicore Processors ,” KSCI, vol. 19, no 5, May, 2014.

Authors

Jung Kyu Park received the M.S. and Ph.D. degrees in computer engineering from Hongik University in 2002 and 2013, respectively. He has been a research professor at the Dankook University since 2014. In 2017, he joined the assitant professor of Department of Digital Media Design and Applications, Seoul Women’s University. His research interests include operating system, new memory, embedded system and robotics theory and its application.

Jaeho Kim received the BS degree in information and communications engineering from Inje University, Gimhae, Korea, in 2004, and the MS and PhD degrees in computer science from the University of Seoul, Seoul, Korea, in 2009 and 2015, respectively. He is currently a postdoctoral researcher in the School of Electrical and Computer Engineering at UNIST (Ulsan National Institute of Science and Technology), Ulsan, Korea. His research interests include storage systems, operating systems, and computer architecture.