Journal of Theoretical and Applied Information Technology
15th February 2018. Vol.96. No 3
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

SCHEDULING ALGORITHMS FOR MULTICORE SYSTEMS BASED ON APPLICATION CHARACTERISTICS

[1]JUNG KYU PARK, [2*]JAEHO KIM, [3]HEUNG SEOK JEON
[1]Department of Digital Media Design and Applications, Seoul Women's University, Korea
[2]Department of Electrical and Computer Engineering Virginia Tech Blacksburg, USA
[3]Department of Computer Engineering, Konkuk University, Korea
E-mail: [1]smartjkpark@gmail.com, [2]kjhnet@gmail.com, [3]hsjeon@kku.ac.kr

ABSTRACT

In this paper, we research how an application effects on other applications when they are executed in the same processor. And we take advantage of PMU (Performance Monitoring Unit) to examine that shared resource has the strongest relation with the influence. Based on the analysis, we design a novel user-level scheduling scheme that monitors applications characteristics on-line utilizing PMU and allocates applications into cores so that it can reduce the contention of shared resources. The key idea of this scheme is separating high-influential applications into different processors.

Keywords: *CPU, LLC, NUMA, Performance Monitoring Unit*

1. INTRODUCTION

In order to improve the performance of the system, the number of cores of the processor is increasing in order to improve the parallelism of the system in addition to the processing speed of the CPU. By improving the parallelism, it is possible to perform different applications simultaneously on each core, or to allocate one application to each core, thereby improving the performance of the system [1].

For example, the Xeon E5-2697 processor and the AMD Opteron processor each have 14 cores and 16 cores, which can handle 14 and 16 applications simultaneously. The number of cores for each processor is no more than 20, but recent systems have more than two processors, the IBM x3850 system has 8 processors with 8 cores and the AMD Bulldozer has 4 cores with 16 cores Both systems have a total of 64 cores [2][3].

In such a Many Core environment, cores share resources of the system, causing competition for shared resources, and performance degradation occurs when cores perform applications due to competition. In order to improve the overall performance of the system, a task scheduling

Figure 4.2: *Behavior of various functions nearer to the proposed function*

method considering the contention of shared resources among cores has been studied [4][5][6].

For this work, we selected 11 workloads to determine their mutual influence. The workload has two patterns, pattern with frequently access data to memory, which results in high memory bandwidth and relatively regular execution patterns. Based on this result, we used the PMU (Performance Monitoring Unit) to analyze the causes of mutual influences [3]. As a result of analysis, PMU registers representing the characteristics and interactions of the core under application are selected and based on this, an optimal scheduling scheme for efficiently writing shared resources in the NUMA structure is proposed [7][8]. In order to verify the proposed scheme, we performed experiments comparing with the Linux 4.2 basic scheduling policy in two system environments.

The composition of the paper is as follows. In section 2, we review the NUMA environment and cache allocation works related this work. In Section 3, we show the motivation of this paper. In section 4, we discuss the application attribute-based scheduling algorithm proposed in this paper. In Section 5, we introduce the shared resource problem in inter-cores. In Section 6, we present evaluation results focusing on performance and describe the difference of prior work at section 7. Finally, Section 8 concludes the paper.

* Corresponding author

2. RELATED WORKS

Studies dealing with the competition of cores for limited shared resources have focused on the distribution of cache resources. In order to prevent the corruption of the cache resource, it is necessary to divide the available cache space according to the cache hit ratio or the operation characteristic of the core, or the cache hit ratio of the cores in the NUMA environment, have been proposed to balance the cache hit ratio between processors by competing shared resources in the same processor in the order of application having the highest priority [7][9]. There is also a technique that divides the entire memory space so that each cache space can only load data in a specific memory space, preventing a small number of cores from contaminating the cache. There are a lot of other researches, too. The reason why the study of cache resources is actively conducted is because competition for cache space is the most influential factor in application performance among shared resource competition.

Studies on cache resource allocation as well as studies on memory bandwidth allocation have been carried out in many researches related to virtualized environments with high memory bandwidth usage because they operate several virtual machines [10]. For example, in the NUMA environment, memory allocation and access have higher latency when accessing the remote memory, so that the local memory has a higher priority than the remote memory. However, when memory access is concentrated in local memory, and memory bandwidth usage exceeds available memory bandwidth, a bottleneck occurs.

In addition, in the NUMA environment, a research has been conducted to allocate more cache resources considering the remote memory access penalty due to the difference of local / remote memory access latency in balancing the cache hit ratio between processors. Hardware Prefetch is mainly focused on improving the prediction accuracy of Prefetcher. For example, when the core detects data approaching a certain pattern, Hardware Prefetcher uses a filter or prefetch after checking previous logs. Decides whether to perform it. We have also studied the optimal use frequency of prefetch. The more aggressive use of

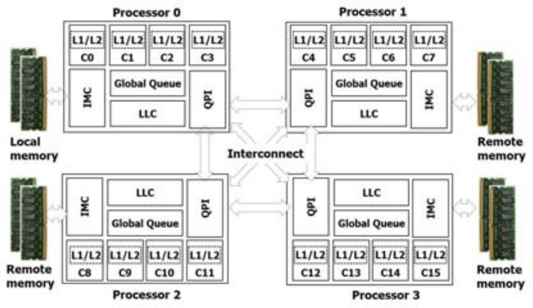


Figure 1: Architecture of NUMA(Non-Uniform-Memory-Access)

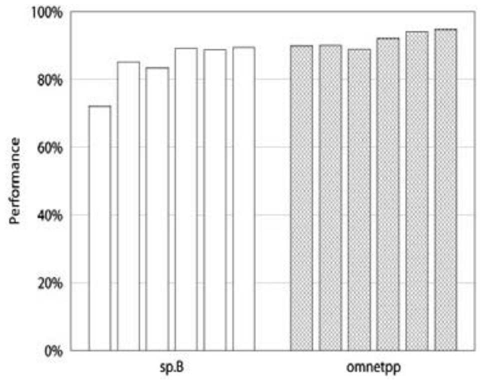
prefetch, the higher performance improvement can be expected when the prefetch prediction is successful. As a representative solution, research has been conducted to monitor the data access of the core and to switch the prefetch utilization frequency in real time.

Studies have also been conducted to better utilize PMU (Performance Monitoring Units) provided by chip manufacturers to provide system programmers with micro-architecture information about cores. There are studies to reduce the overhead to monitor the execution of cores in real time, and to improve the monitoring accuracy by changing the mechanism that kernel saves register information in Context Switch.

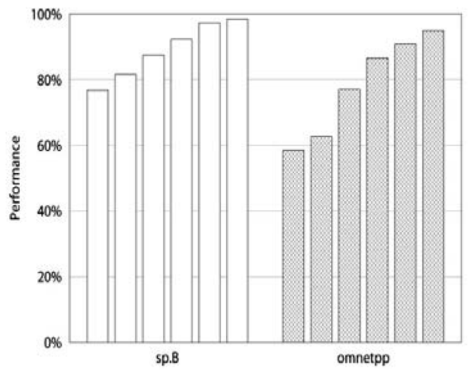
3. MOTIVATION

As the number of cores increases, recent systems are composed of NUMA (Non-Uniform Memory Access) in hardware. In order to solve the bottleneck of the memory bandwidth due to intensified competition for the memory bandwidth, which is one of the resources of the system sharing between the cores, the memory is physically accessed for each core of the processor as shown in Figure 1.

In this architecture, the cores still compete for resources such as LLC in the processor, memory bandwidth between processors, and interconnect bandwidth. In such a shared resource competitive environment, depending on the shared resource usage or the data access pattern of each core, the inter-core work is affected.



(a) Applications influencing other applications



(b) Applications being influenced by other applications

Figure 2 : Differences in the influence of sp.B and omnetpp

Figure 2 shows the impact of the sp.B and omnetpp workloads on the performance of different workloads and the impact of different workloads on different workloads and shared resource contention. As shown in the figure, the sp.B workload shows that the impact on other workloads is greater than the omnetpp workload, but stubborn against its performance degradation. In this way, the core tends to be different from the shared resource competition depending on the characteristics of the application being executed.

The applications that the core performs have different mutual influences. It is expected that the performance of the system can be improved if core applications are identified and classified according to their mutual influences, scheduling applications being executed by cores between different processors, and optimizing shared resource competition.

4. INTER-CORE SHARED RESOURCES

As the number of cores increases, recent systems are composed of NUMA (Non-Uniform Memory Access) in hardware. In order to solve the bottleneck of the memory bandwidth due to intensified competition for the memory bandwidth, which is one of the resources of the system sharing between the cores, the memory is physically accessed for each core of the processor as shown in Figure 1.

4.1 Shared Cache

The cache in the processor has a hierarchical structure in which caches having different sizes are classified as Level 1 and Level 2. The LLC (Last Level Cache) is a cache resource that is shared among cores as the name implies in the last Level cache. In recent systems, the Level 3 cache is mostly Last Level, and as the number of cores increases, the size of the LLC increases. As the cache hit ratio is low because of the low localization of the cores performed by the processor, the LLC space is insufficient and LLC competition is intensified.

For example, if core A is low in locality of accessing data and the cache hit ratio is low, data of Core A that is not reused will occupy space in LLC and data that will be hit by another core will not be loaded into LLC. Or, if the data access pattern is a pattern that contaminates the LLC like the stream pattern, or if the data access is frequent and occupies more cache space than the other cores, the LLC competition will be intensified and the performance of the LLC will drop significantly.

4.2 Memory Bandwidth

The memory bandwidth available in the system is limited, but bottlenecks occur when cores access memory frequently to access data. In Linux 4.2 basic NUMA policy, due to the local / remote memory access penalty, the memory utilization of the core utilizes the local memory of the processor until the capacity is short, and the remote memory is utilized when the capacity of the local memory becomes insufficient. The memory is migrated periodically on a page-by-page basis, and the page data is migrated to the memory of a processor with a frequently accessed core. The core is designed to migrate to a remote memory when the access frequency to the remote memory is greater than the local memory access frequency.

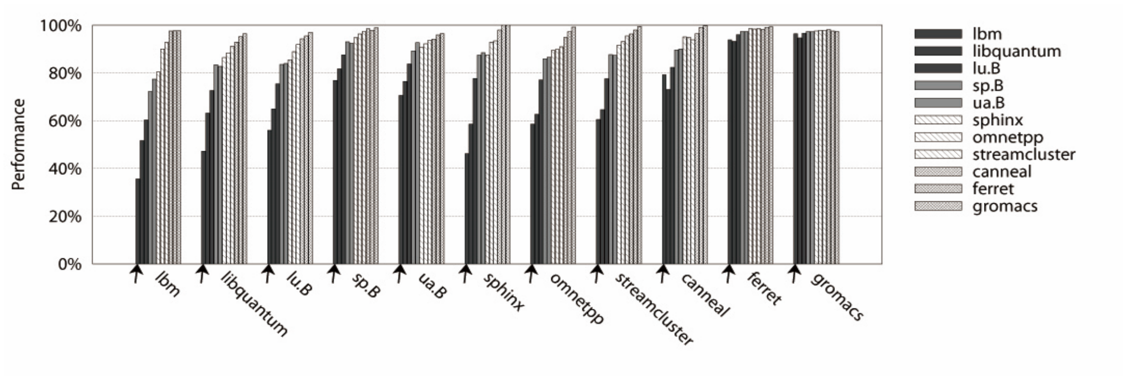


Figure 3 : Performance of workloads when competing for shared resources

In the worst case, the cores of processor A frequently access memory locally, and the cores of processor B frequently access the memory of processor A to the remote, thereby exceeding the available memory bandwidth of processor A. The memory bandwidth of processor B is sufficient, but the memory bandwidth of processor A is a bottleneck. To mitigate this situation, much research has been done to balance the demand for memory bandwidth of cores per processor in the NUMA environment.

4.3 Interconnection

In a NUMA architecture, the bandwidth is for one processor to communicate with the other. There are mainly requests for state change requests, data requests, and remote memory data for the remote cache line, including Intel's Quick Path Interconnect (QPI) and AMD's HyperTransport technology. As with memory bandwidth, the more frequently a core accesses a remote processor, the greater the demand for interconnection bandwidth and can become a bottleneck.

5. APPLICATION CHARACTERISTIC-BASED SCHEDULING ALGORITHM

5.1 Mutual influence between applications

Fig. 3 shows the results of an experiment conducted on a system consisting of two Intel Xeon x3650 processors with Linux version 4.2 and 4 cores each. Memory per processor is 32 GB in size, 8 MB of LLC, 32 KB of L1 Instruction/Data cache, and 256 KB of L2 cache. The workloads selected from 48 SPEC CPU 2006, PARSEC, and NPB workloads. In order to construct a sufficient competition state, we selected 11 workloads with frequent data access and high memory bandwidth and relatively consistent performance pattern.

The experiment performed two workloads on different cores in the same processor and performed until the end of both workloads. The performance of the workload is based on the completion time when it is performed by itself, and the NUMA Aware Load Balancer of the NUMA library is scheduled by the NUMA library to prevent the workload from being mapped to another core.

Table 1 : Performance of workloads in the presence of shared resources

		Influenced by others												
		lbm	libqu	lu.B	sp.B	ua.B	sphin	omnet	stream	cann	ferret	grom	Mean	STD
Influence to others	lbm	35.6	47.2	56.0	76.9	70.6	46.2	58.6	60.5	79.2	93.8	96.4	66.0	19.3
	libqu	51.7	63.2	65.0	81.7	76.5	58.6	62.6	64.6	73.0	93.7	94.7	71.6	13.8
	lu.B	60.3	72.7	75.4	87.6	83.7	77.6	77.1	77.6	82.3	96.0	96.7	81.0	10.3
	sp.B	72.2	83.4	83.6	93.0	89.2	87.4	85.9	87.7	89.6	97.4	97.4	88.2	7.0
	ua.B	77.4	82.7	84.0	92.4	92.7	88.5	86.6	87.3	90.0	97.4	97.2	89.0	5.9
	sphin	80.4	86.5	85.5	94.8	90.7	87.5	89.5	91.6	95.1	98.6	97.5	90.9	5.5
	omnet	90.1	88.4	88.9	96.3	92.1	93.0	89.8	93.1	94.9	98.4	97.9	93.3	3.4
	stream	92.8	91.2	91.9	97.4	93.6	94.5	91.0	95.5	93.8	98.6	97.7	94.4	2.5
	cann	97.5	92.9	94.2	98.5	94.2	98.0	94.9	96.4	97.5	98.3	98.2	96.6	1.8
	ferret	97.7	95.2	95.5	97.8	95.9	99.7	97.2	98.0	99.1	99.1	97.5	97.8	1.5
	grom	97.7	96.4	97.0	98.9	96.5	99.8	99.2	99.5	99.8	99.3	97.3	98.7	1.6
	Mean	77.6	83.4	83.4	92.3	88.8	85.2	84.8	87.5	90.3	97.5	97.1		
	STD	20.9	15.1	13.1	7.3	8.4	17.5	13.4	13.1	8.7	2.0	1.0		

Figure 3 present how each workload on the x-axis is affected by the performance of different workloads. One x-axis workload depicts eleven y-axes, which is the result of the work performed with the workloads shown in the legend. The order of the y-axis is the same as the order of the workloads shown in the legend. To see how it affects the performance of different workloads, you can refer to the legend on each x-axis workload and identify the y-axis of the same sequence number. For example, there is an arrow in the Figure 3 that shows how the '*lbm*' affects other workloads. The table 1 shows the values of the above Figure 3.

5.2 Grouping applications by mutual influence

As you can see in Figure 3 above, you can group workloads according to their influence. Groups used simple numbers to compare the mutual influence of workloads.

(1) Group 1

The workloads for Group 1 are lbm, libquantum, and lu.B. They have a big impact on the performance of other workloads in common. They have an impact on other workloads and are heavily impacted by other workloads.

(2) Group 2

Group 2 workloads are sphinx, omnetpp, and streamcluster. Their mutual influence feature does not degrade the performance of relatively different workloads, but their performance drops significantly by Group 1.

(3) Group 3

Group 3 workloads are sp.B, ua.B, canneal, ferret, gromacs. They compete for workloads and shared resources but do not significantly degrade the performance of other workloads, and their performance is hardly affected. Sp.B and ua.B have features that degrade the performance of other workloads than Group 2 workloads, but they do not affect their performance as much as Group 1 or Group 2 workloads. Therefore, we tried to separate sp.B and ua.B into different groups, but it was not so important to distinguish between workloads and Group 3 to simplify the identification of application groups. However, a study was conducted to analyze the causes.

Table 2 : 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	IMC_NORMAL_READS.ANY *number of read requests to IMC*
	UNC_IMC_WRITE.FULL.ANY *number of write requests to IMC*
	REQUEST_BUFFER_FULL *number of requests blocked due to buffer full*

5.3 Identify the properties of an application group

5.3.1 PMU

Performance level counter (PMU) information was used to monitor resource usage and performance characteristics of cores. Intel and AMD count the usage of each resource in a special register to provide information on microarchitecture resource usage. Measurable resource usage can typically measure information such as clock cycles,

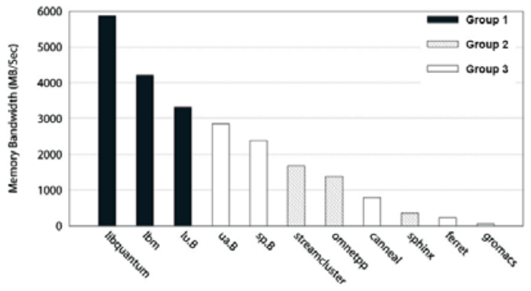


Figure 4 : Memory bandwidth usage

command execution times, cache misses, and memory accesses. Access to these registers can be measured by the RDPMC (Read Performance Monitoring Counter) command, which uses three registers EAX, EDX, and ECX to measure resource usage.

The Intel Xeon x5570 processor used in the experiment has a PerfEvtSelX Machine Specific Register (MSR), and can set specific events such as GQ usage, LLC access count by setting 8 bits of EVTSEL and 8 bits of EVTMSK. The ECX register is used to set a specific event, and the resource usage is stored in two registers, EAX and EDX. The cost of measuring the resource utilization rate is the level of reading hardware registers, and can be monitored quickly without large overhead. As a result of the measurement, the overhead was close to 0.1%.

5.4 PMU events for group identification

Table 2 shows the results of selecting events for identifying application programs based on mutual influence among various PMU events. Monitor multiple PMU events listed in the table to identify application groups sequentially.

5.4.1 Memory bandwidth

The more frequently data accesses the core during execution, the more memory is accessed. It can be seen that the larger the memory bandwidth usage, the more the use of shared resources. Fig. 4 shows the memory bandwidth usage of the 11 selected workloads. The workload of the processor with the clock of 2.93GHz alone was (64 * (UNC_IMC_NORMAL_READS+UNC_IMC_WRITES.FULL.ANY)*2.93*1000000000/cpu_cycle/1000000). The core accesses the cache for data access when the task is performed. If L1 and L2 cache are accessed but data to be accessed is not loaded, access to the shared resource, LLC. If the data to be accessed is not loaded at this time, the address of the memory is finally accessed. This results in

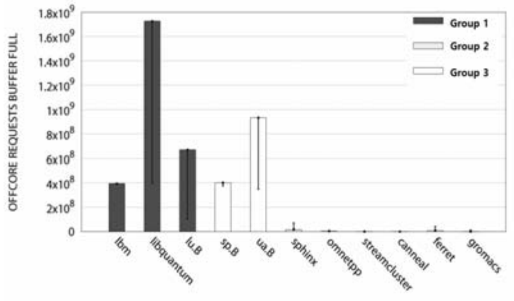


Figure 5 : Event Count of Offcore Requests Buffer Full Count

contamination of the LLC, a shared resource, due to the low locality of the data to be accessed, and uses limited memory bandwidth. On the other hand, they may have more data access than other workloads. Referring to Figure 3 and Figure 4, Group 1 workloads have high memory bandwidth usage and are detrimental to the performance of other workloads.

5.4.2 Super Queue Full Count

It would be nice to be able to monitor the memory bandwidth usage per core using the PMU, but currently PMU can not monitor the memory bandwidth usage per core. Therefore, we decided to use the PMU to monitor Offcore Request Buffer Full events for each core. There is a queue called Super Queue for each core. This Queue is filled with Reuquest for the shared resources of the core. When the core sends a reach request to the LLC for a cache line access request, the corresponding request is filled in the super queue corresponding to the core. The Offcore Request Buffer Full event is incremented when the core requests a request but the Super Queue is full of requests.

If you monitor this event, you will not know the exact memory bandwidth usage per core, but instead you will see performance characteristics for the core shared resources. When the Super Queue is full and the core has continually requested a Cache Line Request, it means that it is using the shared resources aggressively. In Figure 5, not only can you distinguish Wolf workloads, but also the number of PMU values between workloads that have different workloads and those that do not.

Therefore, we have chosen this Offcore Request Buffer Full event as a PMU event to distinguish between cores with high impact and cores that are not affected by other workloads.

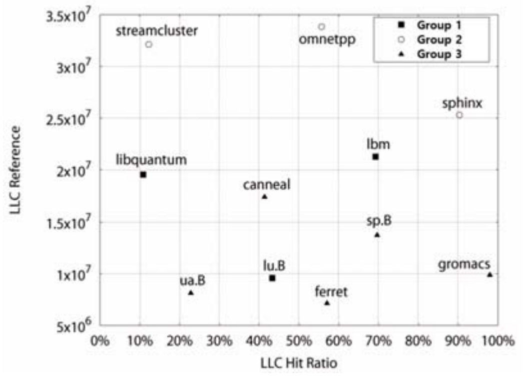


Figure 6 : Event Count of LLC Reference

5.4.3 Shared Cache Reference

The LLC Reference event can tell how much the core is affected by other workloads. The LLC reference value is equal to the UNC_LLC_HITS.ANY (LLC hit) value shown in

Table 2 plus the UNC_LLC_MISS.ANY (LLC miss) value. This shows how the performance of the core is dependent on the LLC, which is a shared resource. Figure 7 shows that Sheep workloads have a much higher LLC reference value than other workloads. And the Offcore Request Buffer Full PMU values and the unaffected Hippo workloads canneal, ferret, and gromacs workloads have lower LLC reference numbers than other workloads.

First, the Offcore Request Buffer Full event identifies Wolf workloads that affect other workloads, and then part of the affected Group 2 workloads and Group 1 workloads that are hardly affected by this workload it can be identified through the Reference event. In previous studies, it was common to see the data locality of the core through the core's LLC Hit Ratio and to balance workloads with high LLC hit ratio and low LLC hit ratio to the same processor, thereby balancing LLC resources. In this paper, we monitor the LLC reference of the core without monitoring the variable LLC Hit Ratio in order to monitor the operating characteristics of how much workload is dependent on the LLC.

For example, Group 1 workloads have a 70% higher LLC Hit Ratio in the absence of competition, but drop to around 20% if they compete with one other workload for shared resources. If other workloads compete for shared resources as well, they will converge to the point

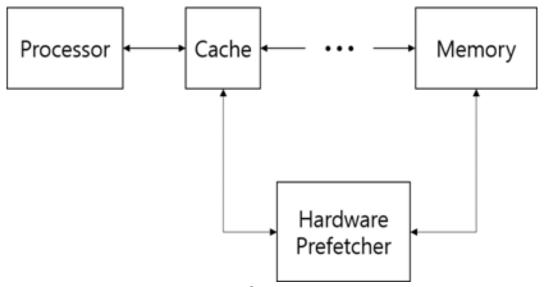


Figure 7: Hardware Prefetch

(0,0) in Figure 6. On the other hand, in the case of the LLC Reference, the difference between the workloads is maintained even if the competition is intensive because it is a unique property of the workload that the core performs.

5.4.4 Prefetch

The reason for writing the cache is to reduce the cost of core access to memory for data access. To take advantage of these advantages, recent processors support Hardware Prefetch. Hardware Prefetch predicts data that the core will access next by recognizing continuous data or regular data access in hardware. The core loads the data from memory into cache before it accesses the data in memory.

For example, L2 Prefetch is a technique for loading data that is expected to be next accessed from memory to L2 or LLC. When the core misses a cache miss in L1, it looks for data in the L2 cache line. At this time, if the cache miss occurs in the Nth cache line of the L2 cache and the core next misses a cache miss in the (N + 1)th cache line of the L2 cache, the L2 Hardware Prefetcher predicts that the next data of the corresponding data will be referred to again, And then loads the next data into the L2 cache in advance.

When the data is hit in the cache, the core does not have to access the memory and can perform the task quickly. While cores with frequent accesses to memory by this technique have great performance gains, the main reason for the mutual influence of applications is to utilize the L2 Hardware Prefetch as the core accesses the data. One of the shared resources, the LLC polluting factor, is that the core accesses continuous data. We can get information about how the core is using L2 Hardware Prefetch to access this data.

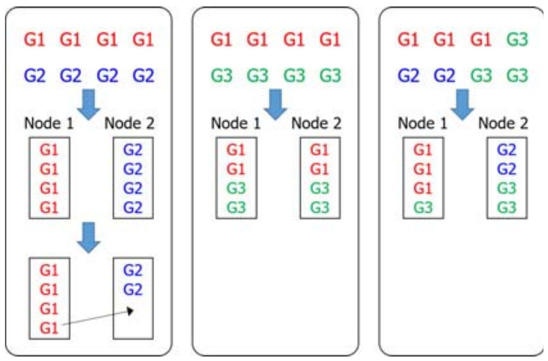


Figure 8: Arrangement group Group 1 and Group 3

5.5 Classification and placement of tasks according to their characteristics

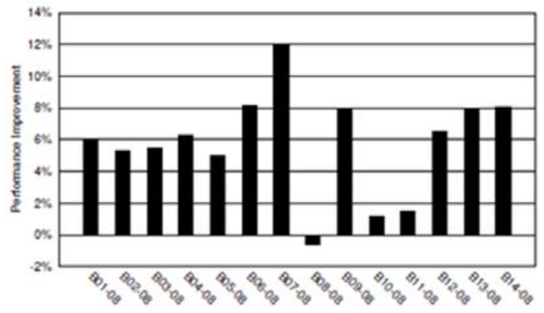
Based on the above experiments and analysis, PMU is used to group applications 1, 2, and 3 in real time. First, to identify applications that are being executed on each core, we use Offcore Reqeust Buffer Full event numbers to identify applications that are estimated to be Group 1, and Group 2 to the remaining workloads do. Other groups are group 3. The threshold value is used for the PMU event value at the time of identification, and this value is determined based on the PMU event values analyzed above.

In other methods, applications that are being performed without applying threshold value to application identification can be classified into groups relatively. For example, if nine applications are running, three applications with high Offcore Request Buffer Full event counts are classified as

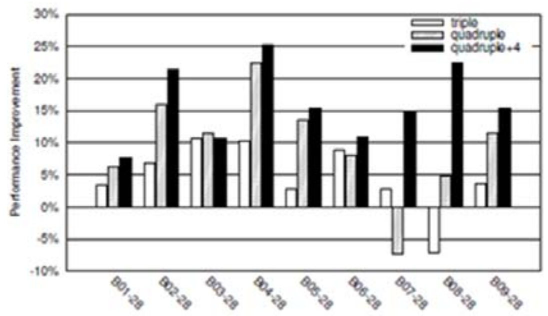
Group 1, three as Gruop 2 with the highest LLC Reference event value, and the remaining three applications as Group 3.

The mechanism for placing the identified applications on the core is simple, placing Group 1 applications in isolation from Group 2 applications. This is how Group 1 workloads are scattered across different processors, collecting Group 1 workloads and protecting Group 2 workloads that are heavily impacted by performance rather than degrading the performance of other workloads.

On the contrary, the worst case is when Gruop 1 and Group 2 spread evenly and all applications are competing to degrade performance. With this mechanism, we expected that Group workloads would be much slower than the default. However, Group 2 was free from competition for shared resources and ended quickly. For this reason, Group 1 was placed in the empty core where Group 2 was deployed, and the competition condition was relaxed, showing performance similar to or faster than default.



(a) *Xeon x3650 system*



(b) *Xeon E5-2697 system*
Figure 9 : Improved performance using threshold value

However, the performance improvement prerequisites for how to isolate this Group 2 and Group 1 workload are when there is enough competition for shared resources. For example, when the number of Group 1 workloads was small and spread, there was little competition for shared resources. At this time, Group 1 workloads may be collected to cause the Group 1 workload to compete for shared resources on the processor.

Figure 8 provides a brief description of the intent of the algorithm. If it is the same as in the first box, group 1 applications and group 2 applications are allocated to each processor. If there are only Group 1 workloads, as in the second box, divide Group 1 workloads to minimize contention for shared resources and place Group 3 workloads on empty cores. In the case of the third box, Group 2 workloads are deployed with Group 3 workloads to protect and Group 1 workloads are deployed independently from Group 2.

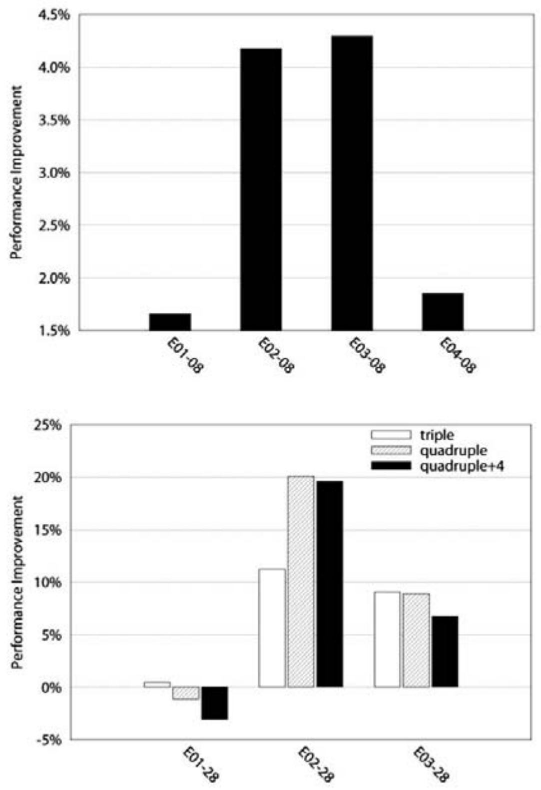


Figure 10 : Improved performance using threshold value with extended workload combination

6. RESULTS

The scheduler implementing the above algorithm is implemented at the user level. The scheduler periodically monitors the PMU events every 10 seconds and uses the system call to perform applications on the desired core. And we apply the system calls for applications that use threads. The NUMA library was used for setting and utilizing NUMA. In the NUMA environment, the local memory priority allocation policy, which is the basic memory allocation policy of Linux, was applied as it is. In order to identify application groups for applications, we implemented a version that applies a threshold value for PMU events and a version that identifies application groups relatively in descending order of PMU event values without applying a threshold value.

We use the workload combinations in Table 3 to compare the implemented user scheduler with the default scheduler in Linux 4.2. Basic workload combinations are combinations of workloads that are used in the above analyzes and already know which application group they belong to. Extended workload is a workload combination

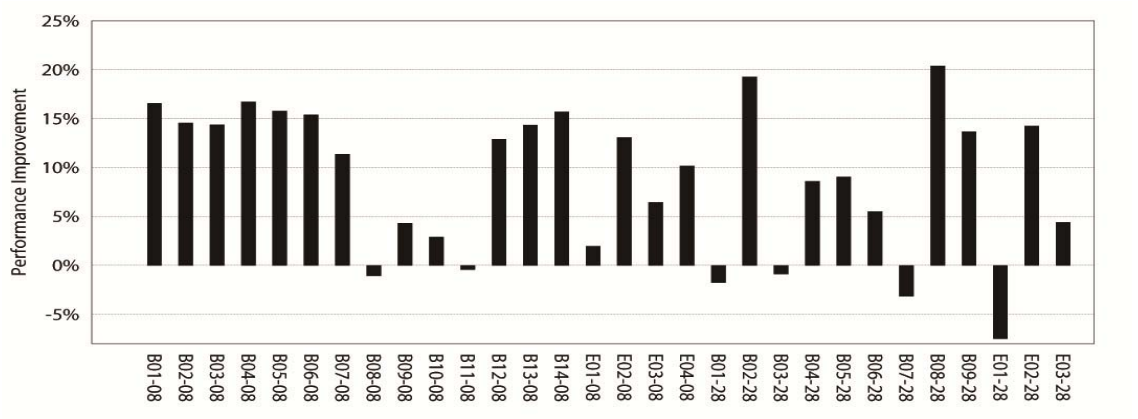


Figure 11 : Improved performance using relativity for application group identification

that includes workloads not used for analysis of SPEC CPU2006, Parsec, and NPB workloads. Workloads that make up an extended workload combination are also workloads that have relatively consistent execution patterns, such as workloads that make up basic workloads. The combination of Basic 1 to 9 and Extended 1 to 3 consists of 6 workloads, Basic 10 to 14, and 8 workloads to Extended 4.

6.1 Identify application group using threshold value

Figure 9 shows the performance improvement of the user scheduler version using the threshold value compared with the scheduler of the Linux version 4.2 by executing the user scheduler version in two environments. The meaning of the x-axis is that B represents the basic workload combination, and the following 01 through 14 represent the number of workload combinations. -08, -28 means the performed environment. In an environment with two processors with eight cores, the average performance of 9(a) shows an average improvement of 5% to 6% and the basic workload 7 shows a 12% maximum performance improvement.

The eighth workload of Basic in the first environment consists of two Group 1 and six Group 2 do not show sufficient performance enhancement due to the lack of enough shared resource competition. In the Figure 9(b), the triple shows 18 workloads, three times six workloads, 18 workloads, quadruple four times six workloads, 24 workloads, quadruple + 4 workload and 4 random workloads. Figure 9(a) shows a performance improvement of up to 25% in a Figure 9(b) environment with more cores than in the environment, and the performance is improved as the number of cores in a triple to quadruple + 4 increases

6.2 Extended workload combinations

Figure 10 show that the user scheduler version using the threshold value for the extended workload combination was performed in two environments. Extended workload combinations do not provide enough shared resource contention as well as previous performance degradation, resulting in only 4.3% improvement in performance.

In the case of E01-28, there is a tendency that the performance decreases as the number of cores to be executed increases. This means that the user scheduler, which is a Group 1 soplex workload with insufficient mutual influence when it is spread, Rather than the other.

In E02-28 and E03-28, there is a Group 1 workload with interworking effects such as lbm and libquantum in the workload combination, enough workloads are degraded due to competition in the shared resources, and performance is improved by the user scheduler.

6.3 Identify application groups using relativity

Figure 11 shows the performance enhancement of the user scheduler, which is implemented by dividing the applications to be executed into application groups. In 28 core environments, workload combination quadruple + 4 was used. In both environments, the performance improvement of up to 20% is shown, but the Extened 1 workload combination also shows that the effect of the implemented scheduler is degraded by 5% because there is not enough shared resource competition. However, the above results show that the performance improvement is achieved even when the threshold value is not used and that the application is relatively divided into Group1, Group 2 and Group 3, and that the performance improvement is enough to overcome the performance degradation of the workload combination E01 or B07 in various workload combinations It can be seen that it happens.

7. DIFFERENCE OF PRIOR WORK

As the number of cores increases in multicore systems, there have been many studies to efficiently or evenly distribute shared resources among the cores. Typically, there are technologies such as Cache Partitioning and Page Coloring that allocate the Last Level Cache separately for each core. Or NUMA environments, research has been conducted on coordinating core and task mapping to solve resource bottlenecks such as memory bandwidth.

In this paper, we have directly tested the mutual influence on performance when typical applications compete for other applications and shared resources in order to efficiently distribute shared resources in NUMA environment. Among the 48 SPEC CPU 2006, PARSEC, and NPB workloads, 11 workloads with high memory bandwidth usage and consistent application performance pattern were selected to generate sufficient shared resource competition. To analyze the causes of mutual influence between applications, PMU supported by processor was used.

We have implemented a user scheduler that monitors performance characteristics of each core in real time and identifies the mutual influence of cores to schedule tasks. The user scheduler avoids mutual impacts by placing applications that have a large impact on the performance of other applications and applications that are greatly damaged in performance by different applications on different processors.

8. CONCLUSION

In this paper, we analyzed the performance characteristics of applications through experiments on representative applications that cause shared resource competition. Moreover, we monitored the PMU event, Offcore Request Buffer Full, LLC Reference, which shows the performance characteristics, and grouped the applications into Group 1, Group 2, and Group 3 according to the mutual influence of the application. The user scheduler that isolates Group 1 and Group 2 provides up to a 25% performance improvement over various workload combinations and the performance of each workload has a certain effect.

REFRENCES:

[1] J. Rao, K. Wang, X. Zhou, and C. Xu, “Optimizing virtual machine scheduling in NUMA multicore systems,” Proc. of the the 19th International Symposium on High Performance Computer Architecture (HPCA), IEEE (China), February 23-27, 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,” Proc. of the USENIX Annual Technical Conference (ATC), USENIX(USA), June 14-17, 2011, pp. 1-15.

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorava, “Addressing shared resource contention in multicore processors via scheduling,” Proc. of the 15th International Conference on Architectural support for programming languages and operating systems (ASPLOS), ACM(USA), March 13-17, 2010, pp. 1291-142.

[6] R. Lachaize, B. Lepers, and V. Quema, “MemProf: a memory profiler for NUMA multicore systems,” Proc. of the USENIX Annual Technical Conference (ATC), USENIX(USA), June 26-28, 2012.

[7] R. Ge, P. Zou and X. Feng, “Application-Aware Power Coordination on Power Bounded NUMA Multicore Systems,” Proc. of 46th International Conference on Parallel Processing (ICPP), Aug. 14-17, 2017, pp. 591-600.

[8] N. Saranya and R. C. Hansdah, “An implementation of partitioned scheduling scheme for hard real-time tasks in multicore Linux with fair share for Linux tasks,” 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE (China), August 20-22, 2014, pp. 1-9.

[9] B. Lepers, V. Quema, And A. Fedorova, “Thread and memory placement on NUMA systems: Asymmetry matters,” In Proceeding of the USENIX Annual Technical Conference (ATC), USENIX(USA), June 22-24, 2015, pp. 277-289.

[10] Y. Cheng, W. Chen, Z. Wang, X. Yu, “Performance-Monitoring-Based Traffic-Aware Virtual Machine Deployment on NUMA Systems,” IEEE Systems Journal, Vol. 11, No. 2, 2017, pp. 973-982.