Microarchitectural Resource Management Issues on Multicore NUMA Systems

Jongmoo Choi*
Dankook University, Gyeonggi-do, Republic of Korea
choijm@dankook.ac.kr

Abstract

Modern computer systems make use of multiple cores and NUMA (Non Uniform Memory Access) architecture. Since multiple cores share various microarchitectural resources such as LLC (Last Level Cache), memory interface and interconnect, contentions on these resources become a serious performance bottleneck. In this paper, we explore research issues how to manage these resources to mitigate such contentions. We first examine the internal structure of a multicore NUMA system and investigate interference among cores during memory accesses. Then, we discuss several existing microarchitectural resource management schemes such as cache partitioning, page placement, task scheduling and virtual machine migration. In addition, we inspect diverse application classification techniques including animal, color, and application slowdown model that analyze the interactions among applications on microarchitectural resources. Finally, we present our observations about how an application affects or being affected by other applications in terms of influentiality and sensitivity.

Keywords: Multicore, NUMA, Microarchitectural resource, Contention, Application Characteristics

1 Introduction

Recent computing paradigms such as bigdata analysis, deep learning and virtualization require high performance computing capability. To meet this requirement, multiple cores are employed prevalently, becoming the fundamental platform for modern computer systems. For instance, an AMD Bulldozer is equipped with four Opteron processors where each processor has 16 cores [15]. An IBM x3850 system has eight Intel Xeon processors, each with eight cores (with a total of 128 cores when we enable hyperthreading) [17]. A lot of smartphones utilize ARM's Big.Little architecture that combines Cortex-A15 cores (big) and Cortex-A7 cores (little) [3]. These trends make it feasible to configure a system with tens, hundreds, or even thousands of cores [18].

The increment of cores demands a scalable memory architecture. Hence, most computer systems adopt NUMA (Non Uniform Memory Access) architecture where each processor has its own memory connected directly through memory interface (e.g. IMC (Integrated Memory Controller)) while accessing other processor's memory through interconnect (e.g. Intel's QPI (Quick Path Interconnect) or AMD's HyperTransport) [2, 22, 14]. With the viewpoint of a processor, memory subsystem is organized hierarchically as local memory and remote memory. The access latency for local memory is faster than that of remote memory, this is why it is called as NUMA.

In a multicore NUMA system, when a core in a processor wants to access data in memory, it uses various shared resources such as LLC (Last Level Cache), memory interface and interconnect. We refer to these resources as microarchitectural resources [10]. Since they are shared by multiple cores, concurrent usage of these resources incurs contentions, leading to performance degradation. For instance,

IT CoNvergence PRActice (INPRA), volume: 4, number: 1 (March 2016), pp. 18-29
*Corresponding author: 152, Jukjeon-ro, Suji-gu, Yongin-si, Gyeonggi-do, 16890, Korea Tel: +82-1899-3700

doubling of cores in a processor results in the average LLC size available to a core being reduced to half. As an another example, requests from a processor can interfere with requests triggered by another processor on memory interface or interconnect. As the number of cores increases, the contentions on these shared microarchitectural resources also increase, becoming a critical point of performance degradation [7, 9, 17, 22, 25, 30].

In this paper, we discuss several existing schemes to mitigate the contentions of microarchitectural resources. The schemes can be grouped into three categories. The first one is partitioning LLC dynamically according to the core's access behavior [28, 29, 21, 11]. The second category is exploiting task scheduling and/or page placement not only to reduce the contentions but also to enhance memory parallelism [30, 7, 15, 14]. The third category is applying virtual machine migration to alleviate the interference on microarchitectural resources [17, 22, 25].

The effectiveness of the schemes heavily depends on the characteristics of applications. For instance, to avoid cache pollution in LLC, we need to identify applications that access LLC with low locality and isolate them from other applications. Also, it is preferable to schedule memory aggressive applications into different processors since scheduling them into a same processor may cause considerable memory interference. This paper examines several techniques, namely color, animal and application slowdown model that classify applications based on their characteristics [27, 16, 24].

Finally, we present our observations conducted on a real multicore NUMA system. Our experimental system has two Intel Xeon x3650 processors and each processor has four cores, 32GB main memory and 8MB shared LLC. In this system, we execute applications selected from 3 benchmarks, SPEC CPU2006 [23], PARSEC [1] and NAS Parallel Benchmark [20]. Our experiments reveal that applications have quite different usage behaviors on microarchitectural resources in terms of influentiality and sensitivity. We also find out that, using PMU (Performance Monitoring Unit) supported most modern processors [6, 8], application characteristics can be monitored online without without a priori knowledge.

The remainder of this paper is organized as follows. In Section 2, we examine the microarchitectural resources in a multicore NUMA system and discuss how cores contend for the resources. The existing contention-aware resource management schemes and application classification techniques are explained in Section 3. Our observations about influentiality and sensitivity of applications are elaborated in Section 4. Finally, we describe the conclusion and directions for future work in Section 5.

2 Background

In this section, we first explain the internal structure of a multicore NUMA system, focusing on microarchitectural resources. Then, we discuss various types of contentions occurred in microarchitectural resources and how they differ according to the application characteristics.

2.1 Microarchitectural Resources

Figure 1 displays the internal structure of a multicore NUMA system. It consists of multiple processors (4 processors in this figure, labelled processor 0 to 3) which are connected through interconnect such as Intel's QPI (Quick Path Interconnect) and AMD's HyperTransport. Each processor (also known as socket and node) has multiple cores (4 cores per a processor in this figure, labelled C0 to C15) and microarchitectural resources such as LLC (Last Level Cache), IMC (Integrated Memory Controller), QPI and GQ(Global Queues). Microarchitectural resources are also called as uncore resources since they are out of a core and shared by cores in a processor [12, 5].

Each processor has its own memory connected directly through IMC. The directly connected memory is called as local memory. Also, each processor can access other processor's local memory through

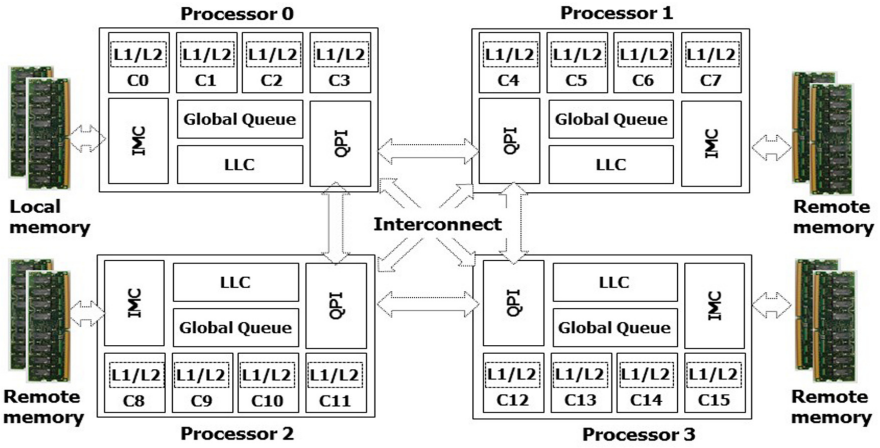

Figure 1: Internal Structure of a Multicore NUMA system

interconnect, which is called as remote memory. In the figure, with the viewpoint of the processor 0, the left upper memory modules become local memory, while other modules are remote memory. The access latency of local memory is faster than that of remote memory (e.g. 255 cycles for local memory vs. 273∼327 cycles for remote memory [4]). Furthermore, as the number of processors increases and interconnect becomes complex, remote memory is further divided into near and far remote memory [15].

When a core in a processor wants to access data in memory, it sends a request to GQ. Then, GQ first checks LLC. If the requested data is already in LLC (hit case), it is served here. Otherwise (miss case), the request is transferred to IMC or QPI based on the address mapping mechanism. Specifically, if a page related to the requested data is mapped into local memory, the request is transferred to IMC and served in local memory. Otherwise, the request is transferred to QPI, being delivered to a processor who has the mapped page through interconnect and eventually served in remote memory.

2.2 Contentions on Microarchitectural Resources

As shown in Figure 1, a core is not an independent unit but rather a part of a larger processor, sharing microarchitectural resources with other cores. Hence, when two or more cores access data in memory at the same time, contentions occur on the shared microarchitectural resources. There are three types of contentions, LLC, memory interface, and interconnect. LLC is contended by cores in a same processor while memory interface is contended by cores that have pages mapped into a same memory module. Finally, interconnect is contended by all cores who reference remote memory. Such contentions combine in complex ways, resulting in overall performance degradation.

The simplest way to mitigate these contentions is scheduling applications into different processors while allocating pages from local memory only. However, when the number of applications is larger than that of processors, it is unavoidable to schedule two or more applications into a same processor. Such a co-scheduling causes the LLC and memory interface contention. To reduce the LLC contention, we can co-schedule applications that do not use LLC aggressively. However, it may incur the underutilization of LLC or contention on other processor's LLC.

To reduce the memory interface contention, we can allocate some pages from local memory and others from remote memory. However, allocating remote memory incurs the interconnect contention, leading to a longer remote memory latency. In other words, there is a tradeoff between memory parallelism and latency. Task migration for the load balancing purpose also affects these contentions, issuing a tradeoff between core utilization and memory efficiency.

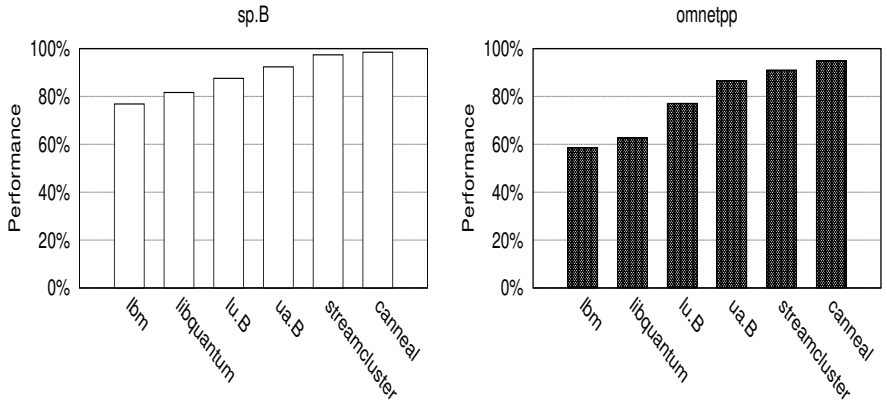

Figure 2: Performance degradation due to contention

Figure 2 presents one of our experimental results regarding the performance degradation due to the contentions. In this experiment, we execute two applications, namely *sp.B* and *omnetpp* chosen from the three benchmarks [23, 1, 20]. The left graph is the results of *sp.B* while the right one is the results of *omnetpp*. The bottom line of each graph shows six applications, namely *lbm*, *libquantum*, *lu.B*, *ua.B*, *streamcluster* and *canneal*, also chosen from the benchmarks.

When we execute *sp.B* (or *omnetpp*) in a core, we also execute one of the six applications in another core in a same processor. For instance, the first bar of the left graph is the result when we execute *sp.B* in C0 (core 0) and *lbm* in C1 (core 1) in Figure 1 . The y-axis is the performance degradation of *sp.B* (or *omnetpp*) due to the contention with the co-scheduled application.

From the results, we can make the following observations. First, performance is actually degraded due to the contentions. Degradation of *sp.B* is ranging from 23% to 2% while *omnetpp* suffers from 40%$\sim$5% degradation. It implies that interference at the shared microarchitectural resources such as LLC and memory slows down an application running on a different core. The second observation is that each application shows different degradation. In this experiment, *omnetpp* shows bigger performance degradation than *sp.B* for all cases. The third observation is that applications listed in the bottom line affects other applications differently. Specifically, *lbm* affects *sp.B* (or *omnetpp*) the most. On the contrary, *canneal* give the least degradation on the two applications. It reveals that contentions on microarchitectural resources have a strong relation with application characteristics.

3 Research Issues

This section presents various microarchitectural resource management schemes. Then, we examine application characteristics and techniques that classify applications based on their impact on microarchitectural resources.

3.1 Microarchitectural Resource Management Schemes

A variety of schemes have been proposed to reduce the contentions on microarchitectural resources. According to the mechanisms they exploit, we group these schemes into three categories: cache partitioning, task scheduling and/or page placement, and virtual machine migration, summarized in Table 1. We visit each category in sequence, exploring research issues and key ideas.

Table 1: Microarchitectural Resource Management Schemes

Category	Scheme	Description
Cache partitioning	UCP [21]	Utility (benefit) based dynamic LLC partitioning
	PIPP [28]	Control insertion and promotion position adaptively
	Page coloring [29]	Allocating different colored pages into applications
	NightWatch [11]	Restrictive mapping for the weak locality data to limit cache pollution
Task scheduling and/or Page placement	DI and DINO [30, 2]	Distribute applications across caches to balance miss rates and eliminate superfluous migrations
	Carrefour [7]	Apply page co-location, interleaving, replication and thread clustering adaptively to manage memory traffic
	AsymSched [15]	A thread and memory placement with the consideration of the bandwidth asymmetry of interconnect
	Shoal [13]	A runtime library with an NUMA-aware array abstraction
Virtual machine migration	A-DRM [25]	A cost-benefit model to assess the interference among VMs
	BRM [22]	Migrate a VM based on the uncore memory access penalty
	VM memory mgmt. [17]	Leverage NUMA overhead awareness in hypervisor's memory management

As discussed in Section 2, LLC is shared by cores in a processor, being contended when cores access data in memory. To mitigate this contention, Qureshi and Patt design a scheme, called UCP (Utility-based Cache Partitioning) that partitions LLC among cores depending on the reduction in cache misses [21]. They define utility as benefit that can be achieved by allocating more cache space into an application. To monitor utility efficiently, UCP introduces a hardware circuit called UMON. In addition, UCP devises a lookahead algorithm to evaluate partitioning decisions when there are a large number of applications sharing cache.

Y. Xie and G. Loh suggest a novel shared cache management policy, called PIPP(Promotion/Insertion Pseudo-Partitioning). When a newly referenced data is inserted in cache (miss case), the traditional LRU policy places it on the MRU (Most Recently Used) position. On the contrary, PIPP places it on the dynamically selected insertion position. Also, when a data is re-referenced in cache (hit case), PIPP promotes it not the MRU position but the promotion position (e.g. promoted by one). By adaptively configuring the insertion and promotion position according to application's cache access behavior, PIPP gives an implicit pseudo-partitioning effect.

To control cache partitioning, an operating system based approach, called page coloring, is proposed by X. Zhang et al. [29]. It assigns the same color to memory pages that map to the same cache blocks. Then, by allocating different colored pages into applications, the scheme can govern the portion of cache used by each application. In addition, to reduce the coloring overhead, the authors devise a hot-page

coloring mechanism that enforces coloring only a small set of frequently accessed pages. Hot-pages are identified by scanning page table entries on-the-fly while leveraging the spatial locality.

R. Guo et al. design NightWatch, a cache management scheme that provides general, transparent and low overhead pollution control [11]. By applying dual-mapping, restrictive mapping for the weak locality data and open mapping for the strong locality data, it can limit cache polluters to evict valuable data from cache. Besides, it makes use of the Intra-chunk locality similarity and Inter-chunk locality similarity to infer the weak or strong locality with low overhead.

Now let us discuss schemes that make use of task scheduling and/or page placement to alleviate the contentions of microarchitectural resources. S. Zhuravlev et al. give an outstanding paper that addresses shared resource contentions in a multicore system via scheduling [30]. They examine various factors causing contentions including LLC, memory controller, memory bus, and prefetching hardware and analyze how much each factor contributes on the total degradation. Then, they propose a novel scheduling algorithm, called DI (Distributed Intensity), that distribute applications across caches such that the miss rates are distributed as evenly as possible. They extend DI for NUMA, called DINO (Distributed Intensity NUMA Online), that considers not only cache miss rates but also remote memory access overhead [2]. The key idea of DINO is eliminating superfluous application migrations across processors and migrating pages selectively based on a metric, called SRA (Saved Remote Accesses).

Dashti et al. propose Carrefour, an memory management scheme for NUMA [7]. It makes use of various mechanisms including page co-location, page interleaving, page replication and thread clustering for reducing memory traffic. It shows that each mechanism affects differently on applications, some obtain performance gains from co-location while others from page-interleaving. It applies each mechanism adaptively using performance monitor hardware such as MC-IMB (Memory controller imbalance) and MAPTU(Memory accesses per time unit). The authors also propose AsymSched, a thread and memory placement algorithm that takes into account the bandwidth asymmetry of NUMA systems [15].

Shoal is a runtime library for parallel programs on NUMA machines, designed by Kaestle et al. [13]. It provides a new memory abstraction using arrays that can be automatically replicated, distributed, or partitioned across memory modules based on annotating memory allocation statements or compiler's hints on access patterns. It also supports rich memory allocation interfaces and utilizes additional hardware such as programmable DMA copy engines to further improve parallel program performance.

Kim et al. analyze the tradeoff among LLC, memory parallelism and remote memory access overhead in a multicore NUMA system [14]. Using PMU (Performance Monitor Unit) [26], they monitor various events including LLC misses, number of local and remote memory accesses, and observe that microarchitectural resources affect differently on applications. Some applications are affected greatly by the remote memory latency while others by the memory parallelism. Z. Majo and T. Gross report similar observations that it may be more advantageous to allocate data on remote memory than to store data of all threads in local memory due to the overload of the on-chip memory controller [19].

Now let us turn our attention to the virtual technology. Hypervisor (also known as Virtual machine monitor) is another good layer to manage microarchitectural resources. H. Wang et al. propose A-DRM (Architectural-aware Distributed Resource Management) for a virtual cluster where multiple VMs(Virtual Machines) run concurrently on multicore NUMA systems [25]. The conventional DRM scheme makes use of CPU utilization and memory capacity demand for VM allocation or migration decision. However, the authors find out that many applications exhibit different memory bandwidth and/or LLC usage behavior even though they have similar CPU utilization and memory capacity demand. A-DRM devise a cost-benefit model using LLC misses and memory bandwidth to assess the interference among VMs and gives better VM allocation decisions.

M. Liu and T. Li investigate four different sources of microarchitectural resource contentions on server virtualization and workload consolidation environments [17]. The four sources are LLC contention, remote memory access, memory controller conflict and Interconnect congestion. They design

three optimization techniques, namely estimation memory access overhead, a NUMA-aware buddy allocator and a P2M swap FIFO, and incorporate them into the hypervisor's virtual machine memory allocation and page fault handling routines.

BRM (Bias Random vCPU Migration) is suggested by J. Rao et al. [22]. It is a novel virtual machine scheduling algorithm that dynamically migrates vCPUs to minimize the system-wide resources contentions. The authors observe that the penalty of referencing the uncore memory subsystem is a good runtime metric of application performance. Hence, BRM selects a biased processor who has the minimum penalty as the migration destination. In addition, it employs randomness to produce more predictable system performance for dynamic workloads and to avoid expensive synchronization on multiple independent cores.

3.2 Analysis of Application Characteristics

In a multicore NUMA system, applications co-scheduled at a same processor affect each other due to the sharing of microarchitectural resources. Some applications suffer from performance degradation when co-scheduled applications contend for shared resources in a destructive manner. On the contrary, other applications do not experience such degradation when co-scheduled applications scarcely utilize shared resources.

Hence, application characteristics such as working set size, cache utilization and memory access rate give a significant impact on shared resources usage, eventually leading to different performance degradation. Several techniques have been proposed to classify applications for determining how they affect each other when competing for shared resources. The classification can be used effectively for identifying which applications should and should not be scheduled together.

Xie and Loh propose an interesting classification technique for identifying the personalities of applications with respect to their cache sharing behaviors [27]. Their technique is referred to as an animal classification since they classify applications into four animals, *turtle, sheep, rabbit* and *devils*. In the technique, when an application does not access LLC a lot (specifically, less than 1000 during one million cycles), it is classified as turtle. Among other applications, if its miss rate is high ($>$ 10%) or the total number of misses is high ($>$ 4,000), it is classified as devil. Next, an application that requires at least half of LLC size to achieve more than 95% hit rate is classified as rabbit. The remaining applications become sheep.

The rationale behind this classification is as follows. An application classified as turtle usually has a small working set that completely fits within the L1/L2 cache of a core and therefore rarely accesses LLC. On the contrary, an application classified as devil accesses LLC very frequently, but still have very high miss rate. The authors mention that devil applications do not play well with other applications. Both sheep and rabbit applications access LLC frequently and have high hit rate. But, rabbit requires large LLC size for high hit rate while sheep does not sensitive to the size, achieving a high hit rate with a small LLC size. The authors verify the effectiveness of their classification using dynamic cache partitioning (e.g., cache partitioning by catching devils) and scheduling (e.g., co-schedule rabbit-sheep and turtle-devil pairing instead of rabbit-devil and turtle-sheep pairing)

Lin et al. design a classification technique that classifies applications into 4 different colors, namely *red, yellow, green* and *black* [16]. They compare application's performance under the two cases; running with 1MB and running with 4MB LLC. Any program with greater than 20% slowdown was classified as red while 5%$\sim$20% slowdown as yellow. Out of the remaining applications (less than 5%), an application whose cache misses per thousand cycles is larger than 14 is classified as green, otherwise as black. It implies that an application classified as red suffers from the cache reduction the most. A yellow application suffers less while a green or black application the least. A black application not only shows small performance degradation but also hardly uses LLC. This classification suggests that we can reap

the most benefits when a red application is co-scheduled with a green or black one.

The pain classification is proposed by Zhuravlev et al. [30]. The technique is based on two new concepts: cache sensitivity and intensity. The sensitivity is a measure of how much an application will suffer when cache space is taken away from it due to contention, while the intensity is a measure of how much an application will hurt others by taking away their space in a shared cache. Then, the pain of A due to B is calculated as the intensity of B multiplied by the sensitivity of A. Finally, the performance degradation of co-scheduling A and B together is the sum of the Pain of A due to B and the Pain of B due to A.

Subramanian et al. devise an interesting model, called application slowdown model that accurately estimates application slowdowns due to interference at both the shared cache and main memory [24]. The slowdown is assessed by comparing the co-scheduled performance with the alone performance. The former is measured on-the-fly while the latter is measured periodically in an aggregate manner by giving the application's requests the highest priority at the memory controller for minimizing memory interference to the measured application. The authors also suggest three use cases of their model: slowdown-aware cache partitioning, slowdown-aware memory bandwidth partitioning and soft slowdown guarantees.

4 Observation

As discussed in Section 3, applications affect microarchitectural resources in a different way which eventually leads to distinctive performance degradation. To analyze these effects quantitatively, we choose 11 applications from SPEC CPU2006 [23], PARSEC [1] and NAS Parallel Benchmark [20] and execute them on our experimental system that have two processors, four cores in each processor, 32GB main memory and 8MB shared LLC. Each core has 32KB Instruction/Data L1 Cache and 256KB L2 Cache.

In this experiment, we compare two scenarios; one is the single execution scenario and the other is the concurrent execution one. In the single execution scenario, we run an application alone in a processor and measure its execution time. In contrast, in the concurrent execution scenario, we run two applications simultaneously within a same processor. Note that, in the singe scenario, all microarchitectural resources are utilized by the application. On the contrary, in the concurrent scenario, the utilization of the microarchitectural resources is interfered by the co-scheduled application. Hence, we think that the performance differences between the single and concurrent scenario are mainly due to the contention of the microarchitectural resources.

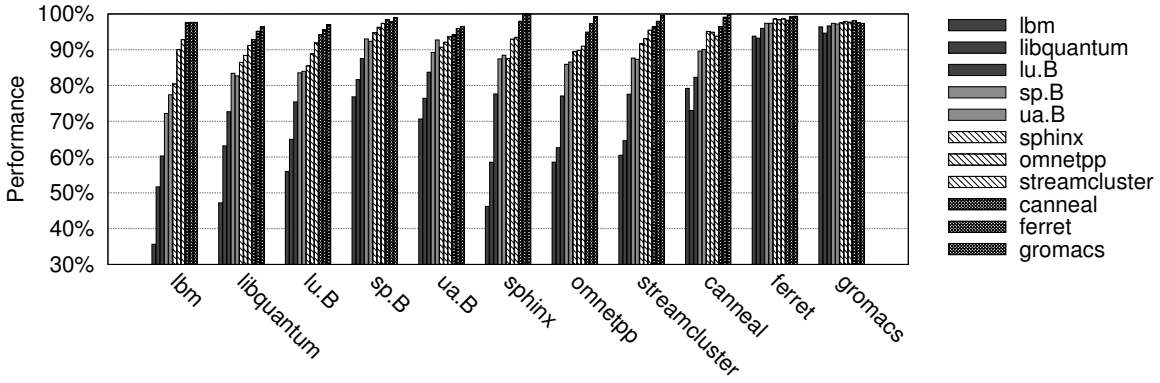

Figure 3: Impact of an application on others

Figure 3 presents our experimental results. In the figure, the x-axis represents the applications where

each application has 10 bars (each application has 10 cases for the concurrent scenario since there are 11 applications). The y-axis is the performance of the concurrent scenario relative to the single scenario. From the results, we can make the following three observations. First, each application gives different impact on other applications. Some influence strongly while others are not. For instance, libquantum drops the performance of lbm to 35% (the first bar of the lbm graph) while gromacs drops lbm's performance to 97% (the last bar of the lbm graph).

The second observation is that some applications are more sensitive than others. Specifically, libquantum's performance degradation is quite sensitive to the co-scheduled application, ranging from 51% (the first bar of the libquantum graph where it is co-scheduled with lbm) to 96% (the last bar of libquantum when libquantum is co-scheduled with gromacs). On the contrary, gromacs is less sensitive to the co-scheduled application whose degradation ranges from 94% to 97%,

The final observation is that each application shows some consistent influence patterns on other applications. For instance, lbm and libquantum affect other applications strongly for all applications. In the animal classification taxonomy, they can be classified as devil [27]. On the contrary, ferret and gromacs influence the least, like turtle in the animal classification. In the legend of Figure 3, we intentionally place applications according to their influentiality. Then, all bars in each graph have a tendency to the rising pattern.

Our observations provide useful hints for application allocation decisions. For instance, allocating lbm and libquantum into a same processor incurs considerable performance degradation for both applications. Hence, we need to schedule these applications into different processors. On the contrary, ferret and gromacs are good candidates for co-scheduling with lbm or libquantum since they are insensitive to co-scheduled applications. In some cases, especially when there is a serious contention on microarchitectural resources, an admission control that delayes an application dispatching is a better choice even though there is an available core.

5 Conclusion

In this paper, we explain various types of contentions on microarchitectural resources. Then, we explore the existing schemes such as LLC partitioning, task scheduling, page placement and virtual machine migration that are proposed to mitigate the contentions. In addition, we examine the application analysis techniques to provide useful heuristics for the schemes and discuss several tradeoffs between memory parallelism and remote memory latency and between core-level load balancing and microarchitectural interference. Finally, we describes our observations about the influentiality and sensitivity among applications in a multicore NUMA system.

We will extend our study into two directions. The first one is designing a novel scheduling scheme that exploits hints discussed in Figure 3. It considers both application characteristics and the microarchitectural resource utilizations for application allocation/migration decision. The second direction is estimating application characteristics on-line without a priori knowledge. We are currently analyzing several PMU (Performance Monitoring Unit) events such as LLC misses, memory requests per time unit, and local/remote memory access to correlate these events with application characteristics.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2015R1A2A2A01007764) and also supported by the research fund of Dankook University (BK21 Plus) in 2014.

References

[1] Princeton Application Repository for Shared-Memory Computers. PARSEC. http://parsec.cs.princeton.edu/.

[2] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX'11), Portland, Oregon, USA*, pages 1–1. USENIX, June 2011.

[3] Shekhar Bojkar. Composite cores: Pushing heterogeneity into a core. In *Proc. of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12), Vancouver, BC, Canada*, pages 317–328. ACM Press, December 2012.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proc. of the 8th USENIX conference on Operating systems design and implementation (OSDI'08), San Diego, California, USA*, pages 43–57. USENIX, December 2008.

[5] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. Core vs. uncore: the heart of darkness. In *Proc. of the 52th ACM/EDAC/IEEE Design Automation Conference (DAC'15), San Francisco, California, USA*, pages 1–6. ACM Press, June 2015.

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

[7] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 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 (ASPLOS'13), Houston, Texas, USA*, pages 381–394. ACM Press, March 2013.

[8] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proc. of the 38th International Symposium on Computer Architecture (ISCA'11), San Jose, California, USA*, pages 353–364. ACM Press, June 2011.

[9] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Proc. of the 11th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13), Shenzhen, China*, pages 1–10. IEEE, February 2013.

[10] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan and Claypool, 2010.

[11] Rentong Guo, Xiaofei Liao, Hai Jin, Jianhui Yue, and Guang Tan. Nightwatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX'15), Santa Clara, California, USA*, pages 307–318. USENIX, July 2015.

[12] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *Proc. of the USENIX Annual Technical Conference (USENIX'12), Boston, MA, USA*, pages 1–12. USENIX, June 2012.

[13] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: smart allocation and replication of memory for parallel programs. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX'15), Santa Clara, California, USA*, pages 263–276. USENIX, July 2015.

[14] Haecheon Kim, Seungmin Lim, Junkee Yoon, Seungjae Baek, Jongmoo Choi, and Cho Seong-je. Analysis of micro-architecture resources interference on multicore numa systems. In *Proc. of the 31st Annual ACM Symposium on Applied Computing (SAC'16), Pisa, Italy*, pages 1871–1876. ACM Press, April 2016.

[15] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX'15), Santa Clara, California, USA*, pages 277–289. USENIX, July 2015.

[16] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of the 14th International Symposium on High Performance Computer Architecture (HPCA'08), Salt Lake city, Utah, USA*, pages 367–378. IEEE, February 2008.

[17] Ming Liu and Tao Li. Optimizing virtual machine consolidation performance on numa server architecture for

cloud workloads. In *Proc. of the 41st annual international symposium on Computer architecuture (ISCA'14), Minneapolis, Minnesota, USA*, pages 325–336. IEEE, June 2014.

[18] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Thousand core chips: A technology perspective. In *Proc. of the 44th ACM/EDAC/IEEE Design Automation Conference (DAC'07), San Diego, California, USA*, pages 746–749. ACM Press, June 2007.

[19] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proc. of the 4th Annual International Systems and Storage Conference (SYSTOR'11), Haifa, Israel*, pages 1–10. ACM Press, May-June 2011.

[20] NAS ADVANCED SUPERCOMPUTING DIVISION. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[21] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), Orlando, Florida, USA*, pages 423–432. IEEE, December 2006.

[22] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proc. of the 19th International Symposium on High Performance Computer Architecture (HPCA'13), Shenzhen, China*, pages 1–12. IEEE, February 2013.

[23] Standard Performance Evaluation Corporation. SPEC CPU2006. https://www.spec.org/cpu2006/.

[24] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proc. of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15), waikiki, Hawaii, USA*, pages 62–75. ACM Press, December 2015.

[25] Hui Wang, Canturk Isci, , Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu. A-drm: Architecture-aware distributed resource management of virtualized clusters. In *Proc. of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15), Istanbul, Turkey*, pages 1–14. ACM Press, March 2015.

[26] Vincent M. Weaver. Linux perf event features and overhead. In *Proc. of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems (FASTPATH'13), Austin, Texas, USA*, pages 1–7, April 2013.

[27] Yuejian Xie and Gabriel H. Loh. Dynamic classification of program memory behaviors in cmps. In *Proc. of the 1 In the 2nd Workshop on CMP Memory Systems and Interconnects (CMP-MSI'08), Phoenix, Arizona, USA*, pages 1–9, June 2008.

[28] Yuejian Xie and Gabriel H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. of the 36th annual international symposium on Computer architecture (ISCA'09), Austin, Texas, USA*, pages 174–183. ACM Press, June 2009.

[29] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multi-core cache management. In *Proc. of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EUROSYS'09), Nuremberg, Germany*, pages 89–102. ACM Press, April 2009.

[30] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 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 (ASPLOS'10), Pittsburgh, PA, USA*, pages 129–142. ACM Press, March 2010.

Author Biography

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993, and the MS and PhD degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. He is a professor in the Department of Software, Dankook University, Korea. Previously, he was a senior engineer at Ubiquix Company, Korea. He held a visiting faculty position at the University of California, Santa Cruz, from 2005 to 2006, and at the Carnegie Melon University, Pittsburgh, from 2014 to 2015. His research interests include system software, file systems, mobile storage, and virtualization.