Performance Implications of Co-scheduling Modern Parallel Applications on NUMA Multi-core Systems

Cheol-Ho Hong and Chuck Yoo

Korea University

Abstract

The non-uniform memory access (NUMA) architecture has attracted attention as a state-of-the art multi-core solution for addressing the practical limitations of increasing the number of cores in symmetric multiprocessing systems. In order to utilize this architecture, the scheduling of parallel applications has become an important problem.

In this study, we investigate the performance impact of co-scheduling parallel threads in the recent NUMA platform. From our evaluation, we find that certain applications are found to show significant performance improvements when co-scheduled in the same memory domain. The reason for the improvement is examined, and the use of the memory usage patterns of threads in analyzing the co-scheduling impact is advocated on the basis of the examination result.

1 Introduction

The non-uniform memory access (NUMA) architecture has attracted attention as a scalable solution to the practical limitations of increasing the number of cores in symmetric multiprocessing systems. The NUMA architecture consists of several processors, each of which has its own local memory and memory controller. Any processor can access remote memories belonging to other processors through a cross-chip interconnect. Remote memory latency is quite high because of the long path, including the cross-chip interconnect and the remote memory bus. Recent AMD Opteron and Intel Nehalem processors are designed for the NUMA architecture; AMD's HyperTransport and Intel's QuickPath Interconnect are examples of cross-chip interconnect technologies.

A multi-core processor that consists of a NUMA platform shares some hardware resources, such as a last-level cache or memory controller, on the same processor chip. In terms of the shared last-level cache between cores, hardware designers intended parallel threads to have efficient inter-core communication. On the basis of this expectation, researchers have developed cache-sharing-aware schedulers at the operating system (OS) level[3][6][4]. The basic concept of these schedulers is that parallel applications can benefit from cooperative data accesses that induce cache hits between threads by co-scheduling threads within the same last-level cache. The proposed schedulers detect data sharing patterns between threads and cluster them onto cores that share the same cache domain.

Although the traditional perception that the performance of parallel applications is dependent on the place of threads, [8] has found that the significant performance improvement of PARSEC cannot be achieved by just placing threads in the same cache domain. PARSEC is a recently released benchmark program and "focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessor"[1]. In most cases, when threads are co-scheduled, the performance of the program deteriorates slightly because of the contention on the shared last-level cache and memory bus. [8] concluded that cache sharing has very limited influence on the performance of the PARSEC parallel benchmark suite.

However, significant *canneal* and *streamcluster* performance improvement are achieved in PARSEC when threads are co-scheduled using the *native* inputs in the recent NUMA platform. The co-scheduling running time was reduced by as much as 20% according to the number of threads, as compared to the default Linux scheduler. This result is different from the result of [8]. In [8]'s work, the authors report that the contention on a shared last-level cache mainly causes *canneal* and *streamcluster* performance degradation when the *native* inputs are used.

The different result is thought to partly originate from the capacity of the last-level cache. The processors used in our experiments are Intel Xeon E7540 processors, each of which has six cores and an 18 MB last-level cache, whereas the AMD Opteron processor used in [8] has four cores and a 2 MB last level-cache. The larger last-level cache capacity in our system can mitigate the contention on a shared cache. However, the mitigated contention is insufficient to address the performance improvement of the two applications.

In this study, we step back and reinvestigate the performance impact of co-scheduling parallel threads in the recent NUMA platform. From our evaluation, the performance of parallel threads was found to be deeply related to the last-level cache miss rate and the memory usage pattern of each thread. On the basis of this finding, a classification method that mainly analyzes the cache miss rate and the memory usage pattern is proposed in order to distinguish characteristics of each parallel thread. In addition, a simple scheduler that dynamically co-schedules parallel threads is provided using the former method.

The main contributions of this paper are as follows:

First, the memory usage pattern is found to be an important requirement for scheduling parallel threads in the recent NUMA system, and its use is advocated in analyzing co-scheduling impact. Recent research studies related to NUMA-aware scheduling focus on several factors, including the cache miss rate, but not the memory usage pattern[2][5]. Therefore, they tend to have limitations in analyzing the co-scheduling impact.

Second, our research can contribute to NUMA-aware algorithms for contention management, such as DINO[2]. DINO "tries to co-schedule threads of the same application on the same memory domain, provided that this does not conflict with DINO's contention-aware assignment". From our finding, performance improvement is achieved when *streamcluster* threads that have high cache miss rates and that may conflict with DINO's policy are co-scheduled. Therefore, a new method for revising the algorithm in terms of parallel threads can be provided.

The remainder of this paper is structured as follows: parallel threads that could be deployed in the NUMA system are classified in Section 2. A scheduling method for parallel applications is illustrated in Section 3. Our evaluation results are presented in Section 4. Finally, the conclusions are presented in Section 5.

2 Parallel Threads in the NUMA System

Multi-core machine performance depends on the effective use of multiple threads within applications. A multithreaded application has many independent execution flows (threads) and shares the same memory address space. Therefore, threads in an application have identical memory views and share the same set of data structures, such as open files. Threads can be implemented by various thread libraries. Only conventional thread libraries such as NPTL (Native POSIX Thread Library) in Linux are covered in this paper, in order to limit the problem field.

Then, we illustrate the performance impact of co-scheduling parallel threads in terms of the last-level cache miss rate and the memory usage pattern. For this purpose, parallel threads are classified into two groups according to their cache miss rates: *devil* and *non-devil*. The devil terminology is borrowed from existing research[7] and is now common in application classification. Devils tend towards not reusing cached data and very frequently generate cache misses. In contrast, non-devils include turtles, sheep, and rabbits in [7] and exhibit low cache miss rates. A detailed explanation of non-devils is provided in subsection 2.2.

For simplicity, parallel threads in an application are assumed to exhibit uniform behaviors throughout their lifetimes. Therefore, they have the same classification. Few co-operative data accesses that induce cache hits between parallel threads are assumed in order to concentrate only on the memory usage pattern. Moreover, cache contention between threads is assumed to be mitigated.

2.1 Devils

Each parallel thread in an application independently allocates memory regions mainly from its local memory in a NUMA platform. Then,

1. Each thread may access mostly its own local memory, as shown in Figure 1.
2. Each thread may interact with other threads using others' allocated memory while accessing its own local memory, as shown in Figure 2.
3. A master thread primarily allocates some memory regions, and other client threads may interact with the master thread, as shown in Figure 3.

In the first case, devils' frequent last-level cache misses are satisfied from the local memory in both threads A and B. In this situation, suppose that thread B is migrated to the opposite node and two threads are co-scheduled. Then, thread B will suffer from performance degradation caused by remote memory accesses through the cross-chip interconnect, because its allocated memory remains in the original place. Therefore, clustered threads cannot take advantage of co-scheduling benefits.

In the second case, the two threads' last-level cache misses are satisfied from both the local and the remote memory. In this case, any thread with a relatively large number of remote memory accesses can be migrated to

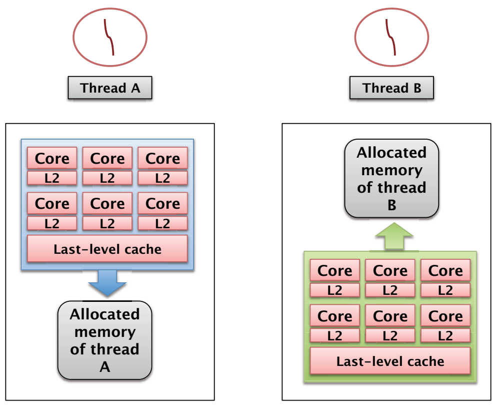
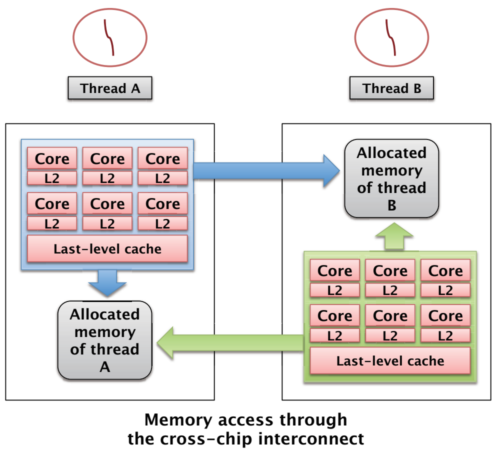


Figure 1: Each thread accesses mostly its own local memory.

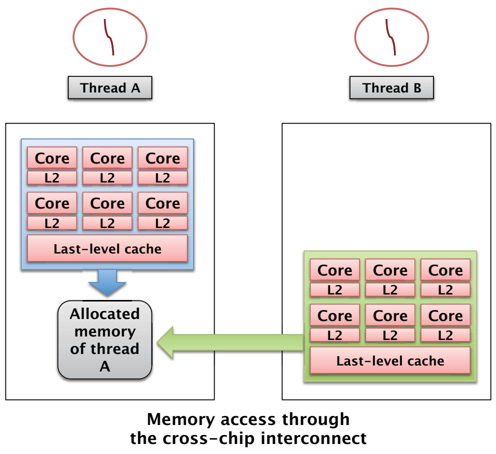


Thread A
Thread B
Core
L2
Last-level cache
Allocated memory of thread A
Allocated memory of thread B
Memory access through the cross-chip interconnect

Figure 2: Each thread interacts with other threads using others' allocated memory while accessing its own local memory.

the opposite node. However, performance improvement is expected to be small because the original local accesses are converted to remote accesses after thread migration. Therefore, in this case, memory migration is required before threads are co-scheduled in the same memory domain.

In the last case, the two threads' frequent last-level cache misses are satisfied from thread A's allocated memory. Then, the performance of thread B cannot help but decrease because of excessive accesses to the remote memory. The best solution for this case is to migrate client threads to the node where the master thread exists and to co-schedule them. However, the main problem of this strategy is that a node cannot accommodate threads that are greater than its size. The OS cannot help this situation; therefore, application re-design is recommended [2].



Thread A
Thread B
Core
L2
Last-level cache
Allocated memory of thread A
Memory access through the cross-chip interconnect

Figure 3: A master thread primarily allocates some memory regions, and other client threads interact with the master thread.

2.2 Non-devils

Turtles in non-devils do not make much use of the shared last-level cache because the application has very few memory instructions or has a very small working set[7]. Sheep exhibit a high last-level cache access rate and are satisfied with the small number of cache ways allocated to them. Finally, rabbits also access the last-level cache very frequently; however, they are sensitive to the number of cache ways allocated to them.

In any case, non-devils do not exhibit high cache miss rates. Then, non-devils do not put pressure on the memory controller and the cross-chip interconnect, regardless of the extent of each thread's allocated memory scattering. Therefore, they are unconstrained by the execution location and co-scheduling them in the same memory domain has no performance impact.

In summary, co-scheduling parallel applications is most effective when threads in an application exhibit high last-level cache misses and interact with each other using a single thread's allocated memory. In addition, memory migration is required before co-scheduling threads in the same memory domain when the allocated memory is scattered across NUMA nodes.

In all cases in devils, the memory usage pattern is found to be an important requirement for co-scheduling or memory migration decisions. Therefore, its use is advocated in analyzing the co-scheduling impact.

3 Scheduling Parallel Threads

As illustrated in the previous section, the performance impact of co-scheduling parallel threads is deeply related to the last-level cache miss rate and the memory usage pattern of each thread. The last-level cache miss rate can be easily obtained via general performance counters. However, it is not simple to obtain the memory usage pattern of each thread via the general performance counters. In [4]'s research, the authors predict the co-scheduling benefit using performance counters related to the coherency protocol events in the Intel Nehalem system. However, such performance counters cannot be used here because these counters only accumulate the number of events and do not indicate where cache misses are satisfied among several NUMA nodes.

A decision was taken to adopt hardware support for data address sampling in order to address this problem. This feature is present in recent multi-core architectures, including the Intel Nehalem, Intel Itanium, IBM POWER5, Sun UltraSparc, and AMD Family 10h processors. The data linear address register is captured using the load latency facility of the Intel Nehalem processors when last-level cache misses occur. The captured address is the linear address of the target of the load instruction. The NUMA node in which each cache miss is satisfied can be discovered after converting the linear address to the physical address using the page tables of each thread. The sampling module is periodically enabled for low overhead, and the module is disabled for the remaining duration. In our research, this low sampling accuracy does not hamper the process of obtaining the memory usage pattern of each thread.

Parallel thread scheduling is now explained using a simple example.

First, parallel threads are classified according to the cache miss rate of each thread. Devils are defined as threads that generate more than one last-level cache miss per 1000 instructions. The threshold value is determined through the examination of the evaluation result.

Second, the linear address of the target of the load instruction for devil threads only is periodically captured when last-level cache misses occur. The captured address is changed to the physical address in order to determine the node number. The memory usage pattern of each thread can then be obtained by using counters for accumulating the number of referenced nodes.

	Node no.	MPKI	Ratio of references to node 0	Ratio of references to node 1
Thread 1	0	15	95%	5%
Thread 2	1	16	90%	10%
Thread 3	0	14	93%	7%
Thread 4	1	15	85%	15%

Third, suppose that there are four threads of a parallel application scattered in two nodes, such as in the table above. In the table, MPKI means misses per kilo instruction.

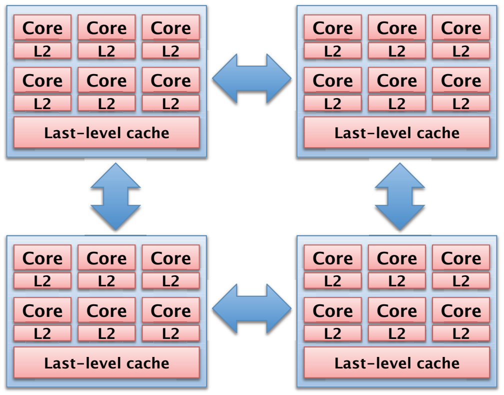


Figure 4: A schematic view of our quad-socket system with four Intel Nehalem processors.

Finally, it can be inferred from this information that the master thread has allocated some memory regions in node 0 and that four threads intensively access the same memory regions. Therefore, they are co-scheduled in node 0. The threshold value for migration is determined to be 80% after examination of the sample evaluation.

4 Evaluation

A state-of-the art quad-socket system with four Intel Nehalem EX processors was used for evaluation. Each processor is an Intel Xeon E7540 processor with six cores and an 18 MB last-level cache. Our system has a 4 GB main memory per node. The CPU layout of this system is illustrated in Figure 4. The system is hosted by Linux-2.6.39 that is modified in order to utilize the load latency facility of the Intel Nehalem processor.

PARSEC is chosen as a benchmark program because it reflects the current CMP's characteristics and has been adopted by many system groups in both research and industry. The *native* PARSEC inputs are used in all experiments.

Programs are run in the benchmark with four threads using the co-scheduling method and the default Linux scheduler, respectively. Figure 5 shows the performance comparison between the co-scheduling method and the default method. A bar represents the running time for co-scheduling normalized to the default scheduling time, and a bar lower than 1 implies better performance.

This result is analyzed as illustrated in Section 2. Each application is classified into devils or non-devils according to the average MPKI of all threads.

As shown in the below table, devils consist of *canneal*, *facesim*, *fluidanimate*, *streamcluster*, and *x264*; non-devils consist of *blackscholes*, *bodytrack*, and *swaptions*. In Figure 5, each bar of non-devils that consist of *blackscholes*, *bodytrack*, and *swaptions* approaches

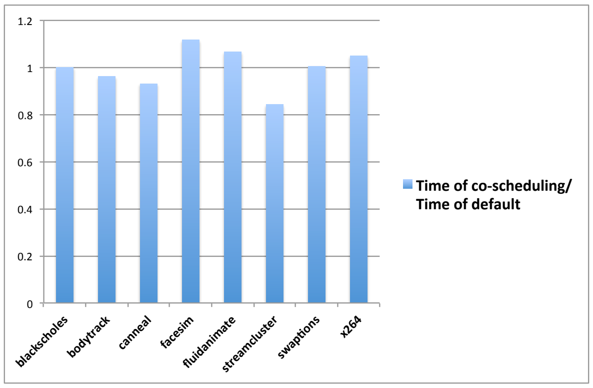


Figure 5: The running time of the case of co-scheduling normalized to the time of the case of default scheduling.

1 because they have no co-scheduling performance impact. In the case of devils, only *canneal* and *streamcluster* have a significant performance improvement when threads are co-scheduled.

Applications	MPKI	Classification
blackscholes	0.1545	Non-devil
bodytrack	0.2237	Non-devil
canneal	22.6062	Devil
facesim	1.1320	Devil
fluidanimate	0.9352	Near-devil
streamcluster	16.2745	Devil
swaptions	0.0032	Non-devil
x264	1.7703	Devil

To address the performance improvement and decline of threads in devils, we compare the memory usage pattern of each thread of both *streamcluster* and *fluidanimate*.

In the case of *streamcluster*, when threads are scheduled by the Linux scheduler, the scheduler scatters threads across nodes in order to achieve load balance. *Streamcluster* threads exhibit the following pattern. In the table below, RRN denotes the ratio of references to nodes n.

	RRN 0	RRN 1	RRN 2	RRN 3
Thread 1	0.1%	99.2%	0.7%	0%
Thread 2	0%	99.3%	0.6%	0.1%
Thread 3	0.5%	97.8%	1.5%	0.2%
Thread 4	0.3%	98.6%	1.1%	0%

From this table, we can infer that *streamcluster* has a master thread that primarily allocates some memory regions, and other client threads interact with the master thread. Therefore, the co-scheduling method is effective.

In the case of *fluidanimate*, when threads are scheduled by the Linux scheduler, they exhibit the following pattern.

	RRN 0	RRN 1	RRN 2	RRN 3
Thread 1	2.1%	92.2%	1.2%	4.5%
Thread 2	0%	53.8%	1.4%	44.8%
Thread 3	7.1%	47.2%	45.5%	0.2%
Thread 4	46.7%	47.2%	5.9%	0.2%

From this table, we can infer that each thread in *fluidanimate* interacts with other threads using the other threads' allocated memory while accessing its own local memory. In this case, memory migration is required before co-scheduling threads in the same memory domain.

By using the simple scheduler described in Section 3, maximum performance improvement of *canneal* and *streamcluster* can be achieved when six threads are co-scheduled using the *native* inputs. The running time of the co-scheduling case is reduced by up to 20%, as compared to the default Linux scheduler.

5 Conclusion

In this paper, the performance of parallel threads was found to be deeply related to the last-level cache miss rate and the memory usage pattern of each thread. Therefore, the memory usage pattern is advocated as an important requirement for scheduling parallel threads. Further, we hope that our research will contribute to NUMA-aware contention management algorithms.

In the future, we plan to develop a more sophisticated scheduling algorithm on the NUMA platform, integrating our memory usage pattern research into existing cache contention and sharing research.

6 Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0029848).

References

[1] Bienia, C., Kumar, S., Singh, J., and Li, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), ACM, pp. 72–81.

[2] Blagodurov, S., Zhuravlev, S., Fedorova, A., and Kamali, A. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 557–558.

[3] Jiang, Y., Shen, X., Chen, J., and Tripathi, R. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel*

architectures and compilation techniques (2008), ACM, pp. 220–229.

[4] Kamali, A. Sharing aware scheduling on multicore systems. *Master's thesis, Simon Fraser University* (2010).

[5] Majo, Z., and Gross, T. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management* (2011), ACM, pp. 11–20.

[6] Tam, D., Azimi, R., and Stumm, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *ACM SIGOPS Operating Systems Review 41*, 3 (2007), 47–58.

[7] Xie, Y., and Loh, G. Dynamic classification of program memory behaviors in cmps. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects* (2008), Citeseer.

[8] Zhang, E., Jiang, Y., and Shen, X. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *ACM SIGPLAN Notices* (2010), vol. 45, ACM, pp. 203–212.