Improving co-running program's performance on CMP

Dawei Tian[1], Hongkui Li[2,a], Wenhui Niu[2], Ying Deng[2], Fujian Li[2]

[1]*State Grid Shandong Electric Power Company, 250001 Jinan, Shandong Province, China*
[2]*State Grid Shandong Heze Electric Power Company, 274000 Heze, Shandong Province, China*

Abstract. Chip multi-processor (CMP) has become the most common processor in the current cluster and desktop computer, and it is also the current development direction. On CMP, programs usually co-running with each other. However, programs commonly interfere with each other. Some time the interference takes big effect, which cause serious drop down of performance. In order to avoid the serious performance interference, programs should be scheduled reasonably to different socket to improve the program's performance and system's utilization. In this paper we propose a new scheduling method to realize a more reasonable scheduling. We do not only consider the LLC miss rate, but also consider the LLC reference of the programs. By the information of LLC reference and LLC miss rate, we schedule programs to different sockets, which realize a reasonable scheduling. The experiment result show that making use of the scheduling method proposed by our paper, program's performance can improve 4%, because the performance improve is realized by on-chip resource, which is a big contribution.

1 Introduction

Chip multi-processor (CMP) has become the most common processor in the current cluster and desktop computer, and it is also the current development direction [1, 2]. A great deal of work is directed at the scheduling of on-chip processors in an effort to improve program performance and system resource utilization.

On CMP, programs usually co-running with each other. However, programs commonly interfere with each other [3]. Some times the interference takes a little effect, which do little effect to program's performance. Some time the interference takes big effect, which cause serious drop down of performance [4]. Meanwhile, for programs co-running on CMP, serious performance dropdown is not expected.

In order to avoid the serious performance interference, programs should be scheduled reasonably to different socket to improve the program's performance and system's utilization [5]. How to schedule the programs is the problem should be deeply explored. The past work uses the last level cache (LLC), they schedule programs to different socket with evenly allocate the LLC miss rate [6], but they do not consider the LLC reference.

In this paper we propose a new scheduling method to realize a more reasonable scheduling. We do not only consider the LLC miss rate, but also consider the LLC reference of the programs. By the information of LLC reference and LLC miss rate, we schedule programs to different sockets, which realize a reasonable scheduling. The scheduling Algorithm consist of two components: the decision module and the scheduling module.

In the decision module, we use micro-architecture events LLC reference and LLC miss rate to do the decision. The value of LLC reference and LLC miss rate are composing to a complex value, and then a program queue with the complex value from small to big are obtained. Then the schedule decision can be make, resulting in the complex value of the two socket are basically same

In the scheduling module, we schedule the programs to the appropriate execution environment. The basic operation of *taskset* is used for our scheduling. we use *taskset* to bind programs to the object process core, then we can change OS scheduling to our schedule with in decision module.

The experiment result show that making use of the scheduling method proposed by our paper, compared with operating system scheduling, program's performance can be improved by 4%, because the performance improve is realized by on-chip resource, which is a big contribution.

2 Related work

There are many related work on CMP scheduling for program's performance and system's utilization.

Zhuravlev et al. [6] found that compared with the cache space contention, memory controller contention, memory bus contention and prefetching hardware contention contribute more to the performance of program. Then predict that it is necessary to minimize the total number of miss rate. Then they use LLC miss rate as the index, which is evenly distribute in each socket.

[a] Corresponding author: dkzxzdh@163.com

SDC [7] is first used for predict the cache contention between threads. It's basic idea is to model the program's LRU stack positions in the LLC and the extra miss produced by other program's contention.

Xie et al. [8] use the animalistic classification to classify program. They classify programs by the interference to each other when co-running on one socket. Then programs can be classified with: turtle (low LLC ref), sheep (low miss rate, insensitive to the number of cache ways), rabbit (low miss rate, sensitive to the number of cache ways) and devil (high miss rate, easy to the hurt co-running programs).

Knauerhase et al. [9] propose contention-aware scheduling algorithm based on LLC miss rate. The hypothesis that the miss rate should explain contention contradicted the models based on stack- distance profiles

Qureshi et al. [10] use Utility Cache Partitioning to predict each program's number of hits and misses for all possible number of ways allocated to the program in the cache. Then thse cache is partitioned with the aim of minimizing the number of cache misses for the co-running programs.

2 Scheduling Algorithm

The scheduling algorithm consist of two components: the decision module and the scheduling module. In the decision module, we use micro-architecture events LLC reference and LLC miss rate to do the decision. In the scheduling module, we schedule the programs to the appropriate execution environment. The flow chart as in Figure 1.

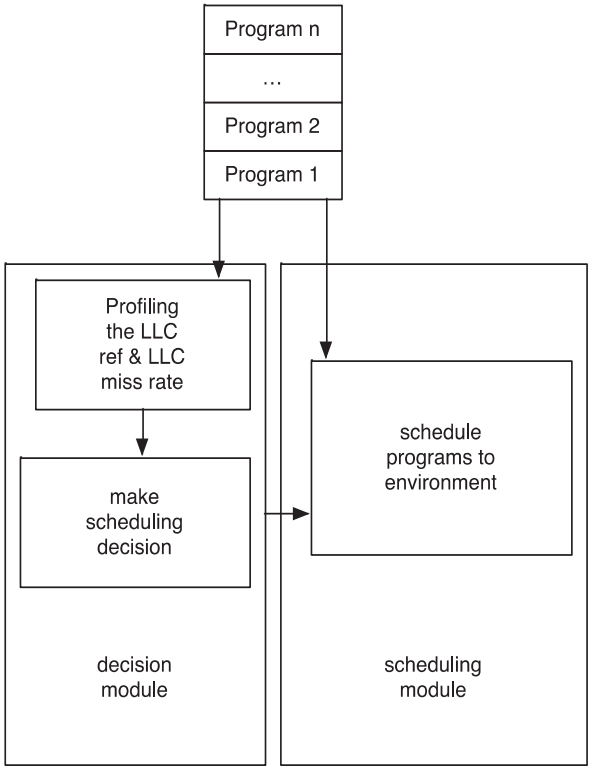

Figure 1. The flow chart of resource isolation method.

2.1 Decision module

In this module, related events of every programs should be profiled, and these events is the input of the decision module. The LLC ref and LLC miss rate can be obtained on-line via common profiling tools such as *perf* [11]. This procedure can be accomplished on-line or off-line.

The function of decision module is to decide which program should be scheduled to the separate socket. This part is the core content of our paper. Because the scheduling plan has directly effect to the program's performance.

If the scheduling plan is good, programs has litte contention for LLC and memory resource, then the program's performance will be good. If the scheduling plan is not good, programs has serious contention for LLC and memory resource, then the program's performance will be bad. The input of the decision module is the events of every programs, including LLC ref and LLC miss rate. The output of the decision module is the scheduling strategy of the programs. The decision module obeys the pseudo-code below.

Pseudo-code of the decision module

```
for i = 1 to n do
    collect the LLC ref and LLC miss rate information of the program
    normalize ref[i], miss[i]
end for
for i = 1 to n do
    complex[i]= ref[i]*0.2 + miss[i]*0.8
    reorder programs from small to big by complex
    schedule programs to different socket according to complex[i]
end for
```

Figure 2. Pseudo-code of the decision module

According this pseudo-code, first we profile program's LLC ref and LLC miss rate by performance profiling tool perf, we profile 108 instructions of every program. Then we normalize the LLC ref and LLC miss rate. The method is take the biggest value of LLC ref and LLC miss rate as 1 separately, and the other value of LLC ref and LLC miss rate is a value between [0,1]. Then calculate the complex value according the pseudo-code, and reorder the programs according the value of complex value. After that we can get a queue with the complex value from small to big. Then the schedule decision can be make. We put the smallest complex value and the biggest complex value into the first socket, then put the second smallest complex value and the second biggest complex value into the second socket, then put the third smallest complex value and the third biggest complex value into the first socket, and so on. At last, all the programs can be scheduled to the two sockets, and the complex value of the two socket are basically same. The above is our decision module.

2.2 Scheduling module

The scheduling module do the implementation of scheduling programs to the object environment. For the operating system, programs are schedule to two sockets evenly. That is, the OS do not consider other thing such as interference, the OS consider only load balance []. For

example, there are four programs need schedule, the OS will schedule the first program to the first socket, then schedule the second program to the second socket, then schedule the third program to the first socket, then schedule the forth program to the second socket. However, the work in our paper can improve program's performance by considering interference. In decision module, all programs are planed to schedule to the two sockets, and the complex value of the two socket are basically same. In the scheduling module, the basic Linux command *taskset* is used to realize the schedule. According the strategy in decision module, we use *taskset* to bind programs to the object process core, then we can change OS scheduling to our schedule with in decision module.

3 Experiment

Our experiments are executed on a NUMA server, which with two socket and 4 core on each socket. The detail server information is in TABLE 1. We select programs from SPEC CPU2006. The SPEC CPU2006 benchmark is SPEC's industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. The programs as workloads are in TABLE 2. The LLC reference and LLC miss rate event collected by *perf* are listed in TABLE 3.

We compare our decision module with the OS scheduler to illustrate the advantage or our method. The OS scheduler scheduling programs according load balance. However, our method scheduling programs not only according load balance, but also according the interference of programs.

Table 1. Server configuration

CPU	Intel Xeon E5620
core	4 cores@2.13G
L1 caches	32K
L2 caches	256K
L3 caches	4M
Threads per core	1 thread
Sockets	2
Memory	8GB, DDR3

Table 2. Workloads

Workloads	Program
Workload 1	410, 416, 429, 444
Workload 2	445, 450, 453,482

Table 3. Collected events

Item	Events
LLC reference	last level cache reference
LLC miss rate	last level cache miss

We use two workloads for our experiment, Figure 3 shows the program's performance with different strategy, and the baseline is the program's performance when program running alone on a server. The programs in the first workload is 410, 416, 429, 444. It is shown by the OS scheduling strategy that the performance of the program is 95% of that when program running alone. But by our scheduling strategy, program's performance is 98% of the performance when program running alone. It is because compared with OS scheduling, our scheduling strategy can not only deal with the load balance, but also deal with the interference between programs running on one socket. Sadly, the OS scheduling can not do contribute on dealing with program's interference running on one socket. However, for this workload, both the two scheduling strategy do not result in serious performance drop down of programs. It is because of the characteristic of programs. These programs (expect 429) have relative little LLC ref and LLC miss rate value, meaning they have little contention on on-chip share resource. Which is means when these programs co-running on one socket, they have little interference and little performance download. But for workload 2, it is different.

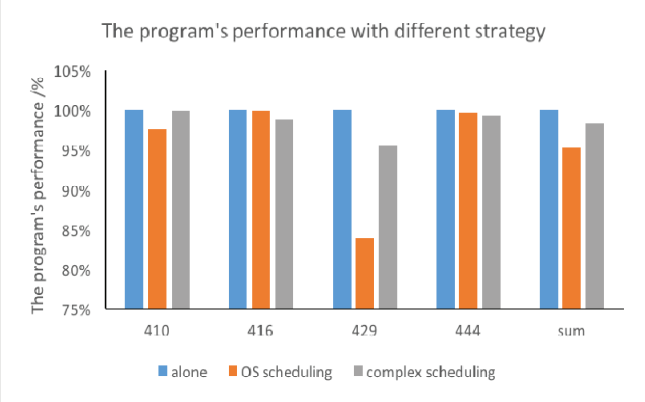

Figure 3. Program's performance with different strategy (workload 1)

Figure 4 shows the program's performance with different strategy of workload 2, and the baseline is the program's performance when program running alone on a server. The programs in the second workload is 445, 450, 453,482. It is shows by the OS scheduling strategy, program's performance is 94% of the performance when program running alone. But by our scheduling strategy, program's performance is 99% of the performance when program running alone. For workload 2, when using the OS scheduling strategy, the average program's performance drop down relative serious, it is because in workload 2, both 450 and 482 have relative large event on LLC ref and LLC miss rate value, which means they have sharp contention on on-chip share resource. when these programs co-running on one socket, they have serious interference and serious performance drop down. So for workload 2 when using the OS scheduling strategy, program 450 and 482 are co-running on one socket, which resulting in performance drop down.

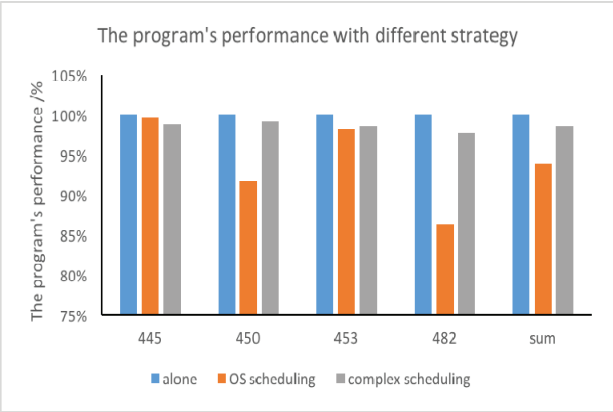

Figure 4. Program's performance with different strategy (workload 2)

4 Conclusion

In this paper we propose a new scheduling method to realize a more reasonable scheduling. By the information of LLC reference and LLC miss rate, we schedule programs to different sockets, which realize a reasonable scheduling. The scheduling Algorithm consist of two components: the decision module and the scheduling module.

In the decision module, we use micro-architecture events LLC reference and LLC miss rate to do the decision. The value of LLC reference and LLC miss rate are compose to a complex value, and then a program queue with the complex value from small to big are obtained. Then the schedule decision can be make, resulting in the complex value of the two socket are basically same

In the scheduling module, we schedule the programs to the appropriate execution environment. The basic operation of *taskset* is used for our scheduling. we use *taskset* to bind programs to the object process core, then we can change OS scheduling to our schedule with in decision module.

The experiment result show that making use of the scheduling method proposed by our paper, compared with operating system scheduling, program's performance can be improved by 4%, because the performance improve is realized by on-chip resource, which is a big contribution.

References

1. C. Dhruba, G. Fei, K. Seongbeom, et al. Predicting inter-thread cache contention on a chip multi-processor architecture[C]. 11th International Symposium on High-Performance Computer Architecture, 2005: 340-351.
2. K. Seongbeom, C. Dhruba, S. Yan. Fair cache sharing and partitioning in a chip multiprocessor architecture[C]. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, 2004: 111-122.
3. Y. Jiang, X. Shen, C. Jie, et al. Analysis and approximation of optimal co-scheduling on chip multiprocessors[C]. 2008 International Conference on Parallel Architectures and Compilation Techniques, 2008: 220-229.
4. K. Rob, B. Paul, H. Barbara, et al. Using OS observations to improve performance in multicore systems[J]. IEEE MICRO, 2008(28):0272-1732.
5. L. Wang, R. Wang, C. Fu, et. al. Interference-aware Program Scheduling for Multicore Processors[C]. ICA3PP 2013, 201312: 436-445.
6. S.Zhuravlev, S. Blagodurov, A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling[C]. Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10), New York, USA: ACM, 2010: 129-141
7. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter- Thread Cache Contention on a Chip Multi-Processor Architecture. In HPCA '05: Proceedings of the 11th International Symposium on High- Performance Computer Architecture, pages 340–351, 2005.
8. Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In Proc. of CMP-MSI, held in conjunction with ISCA-35, 2008.
9. R., P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. IEEE Micro, 28(3):54–66, 2008.
10. M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low- overhead, high-performance, runtime mechanism to partition shared caches. In MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 423–432, 2006.
11. https://perf.wiki.kernel.org/index.php/Main_Page