# Cache Utilization-Aware Scheduling for Multicore Processors

Edward T.-H. Chu, Wen-wei Lu
{edwardchu, g9917722 }@yuntech.edu.tw
Department of Computer Science and Information Engineering,
National Yunlin University of Science & Technology, Taiwan

***Abstract*—A chip multiprocessor (CMP) consists of several cores which can execute tasks independently. Due to the budget and chip area limit, last level cache is usually shared among cores. If tasks running on different cores access the shared cache intensively and concurrently, it may lead to high cache miss rate and significant performance degradation. A commonly-used method is to co-schedule a task with good anti-interference ability and a task with poor anti-interference. However, if tasks have similar anti-interference abilities, it becomes difficult to generate a proper task assignment. In this paper, we identify two more indexes, intra-core cache contention and task interference ability, that primarily determine the utilization of shared cached. Based on the indexes, we develop a novel task scheduling, named cache utilization aware scheduling (CUAS), to reduce shared cache contention. CUAS first classifies tasks according to their anti-interference ability and interference ability. CUAS then distributes tasks to cores based on the effect of inter-core and intra-core cache contention. We conducted our experiments on an Intel Core2 Quad processor and adopted SPEC CPU2006 benchmark for evaluation. According to our experiment results, CUAS can significantly reduce shared cache contention and reduce total execution time at most 46% compared to existing methods.**

**Keywords: Multicore, Cache Contention, Task Scheduling, High Performance Computing**

## I. INTRODUCTION

Due to the limitation of semiconductor process, physical characteristic and thermal diffusion technology, processor speed is not expected to have a significant raise in the near future. In order to further improve the capability of processor, chip multiprocessor (CMP) has become widespread in today's computer systems ranging from smartphones, tablet PCs to high performance servers. A CMP consists of several cores and can concurrently execute multiple tasks. In most multicore processors, the last level cache (LLC) is shared among cores to reduce possible resource underutilization. However, when the tasks running on different cores access shared cache intensively, excessive cache miss may occur and result in performance degradation. Therefore, how to reduce the shared cache contention of multicore systems becomes an important design issue.

A commonly-used method to reduce shared cache contention is to schedule tasks based on their working set size (WSS) [12]. The total working set size of tasks that share a cache is smaller than the size of the shared cache. A task will idle until sufficient cache space has been freed. Similarly, Yang *et al*. [6] and J. Moses *et al*. [8] allocated tasks based on misses per instruction (MPI). However, neither working set size nor MPI can reflect how aggressively a task accesses cache. Two tasks with the same WSS or MPI could have entirely different cache sensitivity, which is defined as the performance degradation induced by insufficient cache space. If two non-cache-sensitive tasks are co-scheduled, the shared cache will be underutilized. As a result, using working set size or MPI to allocate tasks may lead to an improper assignment.

Many research efforts have been made to ease shared cache contention by considering inter-task interference [1, 2, 3]. J. Mars *et al.* [1] and Xie *et al.* [2] classified an application based on its abilities of anti-interference, which is defined as the performance lose when the task competes shared cache with other tasks. Intuitively, tasks lack anti-interference ability will have large performance degradation when sharing cache with other tasks. In order to reduce cache contention, they co-schedule a good anti-interference ability task with a little anti-interference ability task [1, 2]. However, if tasks have similar anti-interference abilities, it becomes difficult for the methods to generate a proper task assignment. In addition, how a task interferences co-scheduled task depends on how aggressively the task accesses cache. A task with little anti-interference ability may or may not seriously interferences co-scheduled applications. S. Zhuravlev *et al.* [3] combined interference and anti-interference ability when categorizing applications. But, their method relies on stack distance profiles, which need a special binary instrumentation tool support [4].

Recently, Sergey *et al.* [3, 5] co-scheduled a high miss rate application and a low miss rate application in the same shared cache in order to even out the miss rate across all the caches. By considering task priority, Shekofteh *et al*. [7] designed a risk function to approximate potential cache contention of a schedule and selected the least risk schedule. All these works assumed that the number of tasks is the same as the number of cores. Similarly, J. Mars *et al*. [1] developed a framework to measure interference sensitivity of applications and avoided

978-1-4577-1729-1/12/$26.00 ©2012 IEEE. 

co-scheduling two high sensitive applications in the same cache. They assumed that an application is composed of a single batch of jobs and are scheduled by FIFO algorithm. However, most existing operating systems adopt timing-sharing algorithms rather than FIFO to schedule tasks. As a result, there is a need to develop a new cache contention aware task scheduling

### A. Our Contributions

In this paper, we designed a novel cache-aware task scheduling, named cache utilization aware scheduling (CUAS), which includes two parts: application classifier and task scheduler. Our application classifier considers both anti-interference and interference ability of a task. We developed two micro-benchmarks: Attack (ATT) and Defend (DEF). The micro-benchmark ATT has strong interference ability and DEF has a strong anti-interference ability. In our implementation, ATT randomly and intensively pollutes all cache lines while DEF sequentially accesses each cache line. In order to evaluate the anti-interference ability of a task, we dispatch ATT and the task to different core but share the same cache. Similarly, we co-schedule the task with DEF in order to evaluate the interference ability of the task. Based on the results of co-scheduling with ATT and DEF, we grade each application's anti-interference and interference ability. For ease of presentation, we define unhealthy score of an application as the sum of its anti-interference and interference scores. The application with higher unhealthy scores has weaker anti-interference and stronger interference ability, and vice versa.

Based on our observation, seriously cache contention occurs if unhealthy tasks share the same cache. It is desirable to group unhealthy tasks on one core and healthy tasks on another core to reduce cache contention. However, workload may not be balanced among cores if the number of health and unhealthy tasks are not even. Therefore, our problem becomes grouping unhealthy tasks while balancing workload among cores. For the sake of simplicity, we define the unhealthy level of a core as the summation of unhealthy scores of tasks running on this core. Our goal is to maximize the difference of unhealthy level of each core that shares the same cache while balancing workload among cores. For this aim, we designed a novel task scheduling, which first calculates the workload of each core and then groups unhealthy tasks.

We conducted our experiments on an Intel Core2 Quad processor and adopted SPEC CPU2006 bench-mark for evaluation. We compared CUAS with the CiPE [1] and Linux default scheduling. According to our experiment results, CUAS can significantly reduce the total execution time in all configurations we tested. The reduction is as much as 46%.

## II. Related Work

### A. Task Classification

The main purpose of task classification is to estimate cache contention if tasks are scheduled to share the same cache. A proper task classification can help designers to schedule tasks efficiently and reduce shared cache contention. Xie *et al*. [2] classified tasks into animal personalities, such as turtles, sheep, rabbits and devils, based on some cache related metrics. Jia *et al*. [11] used L2 MRCs (Miss Rate Curve) to identify application cache behavior online. Both methods require special hardware, such as ATD and LRU counters [13]. Knauerhase *et al*. [10] obtained cache related information from hardware performance counters, and classified tasks into two types: cache-heavy and cache-light. Zhuravlev *et al*. [3] categorized tasks according to their solo miss rates. They avoid the situation where any cache has a much higher cumulative miss rate than any other cache. J. Mars *et al*. [1] assigned scores to a task based on the throughput of the task when it is scheduled with a specially designed benchmark. All these existing works did not consider the interference ability and anti-interference ability at the same time. As a result, their methods may lead to an improper classification and therefore a improper scheduling. Recently, Shekofteh *et al*. [7] designed a risk function to approximate potential cache contention of a schedule and selected the least risk schedule. But, they assumed that the number of tasks is the same as the number of cores. Our task clas-sification, on the contrary, considers both the interference ability and anti-interference ability. We do not limit the number of tasks.

### B. Shared Cache Aware Task Scheduling

Task scheduling is an effective technology to address the problem of cache contention. Calandrino *et al*. [12, 14] partitioned tasks into groups. For each group, the total working set size is less than the size of shared cache. However, working set size does not include the information of memory access patterns which are the key factor of shared cache contention. Two tasks with the same working set size may have very different cache behavior. As a result, the performance of their method may be highly variant from applications to applications. D. Eklov *et al*. [15] and R. P. Dicky *et al*. [16] developed models to predict cache contention of two co-scheduled applications. However, it is not clear how to apply the proposed models to predict cache contention when a group of tasks are scheduled in a shared cache. Zhuravlev *et al*. [3], Robert P. *et al*. [5] and X. Jia *et al*. [11] also developed methods to partition tasks. All these works assumed that the number of co-scheduled tasks is the same as the number of cores. Their methods perform well if tasks are scheduled by FIFO. However, most operating systems adopt time-sharing algorithms to schedule tasks rather than FIFO. Compared to existing works, we do not limit the number of tasks. Our scheduling considers both inter-core cache contention and intra-core cache contention in order to improve the utilization of shared cache. According to our experiment results, CUAS outperforms existing works.

## III. CUAS

CUAS is a user level middleware which includes two parts: application classifier and task scheduler. The application classifier first estimates the cache contention and interference among the applications. The task scheduler next takes the results of the application classifier as input and properly dispatches tasks to cores to improve system performance at run-time.

### A. Application Classifier

Our application classifier uses DEF and ATT micro-benchmarks to categorize tasks. Let $I_i$ denote the interference ability of task $\tau_i$; that is, how aggressively $\tau_i$ interferes with the co-running task. We co-schedule $\tau_i$ and DEF and calculate $I_i$, by

$$I_i = T_{i,d} - A_d , \qquad (1)$$

in which $T_{i,d}$ is the execution time of DEF when it is co-scheduled with $\tau_i$ . The $A_d$ is DEF's execution time when it is executed solely. In our implementation, DEF has a strong anti-interference ability because it sequentially accesses each cache line and has good locality attribute. The larger $I_i$ is, the more aggressive $\tau_i$ interferes with co-scheduled task. Let $AI_i$ denote the anti-interference ability of $\tau_i$; that is, how defensive $\tau_i$ is due to the cache pollution. We co-schedule $\tau_i$ and ATT in order to evaluate $AI_i$, which is obtained by

$$AI_i = T_{i,a} - A_d , \qquad (2)$$

in which $T_{i,a}$ is the execution time of ATT when it is co-scheduled with $\tau_i$. Also, $A_i$ is $\tau_i$'s execution time when it executes solely. In our implementation, ATT has strong interference ability because it randomly and intensively pollutes all cache lines. The larger $AI_i$ is, the weaker the anti-interference ability of $\tau_i$ is. Finally, we define the unhealthy score of a task $H_i$ as the sum of $I_i$ and $AI_i$. We obtain $H_i$ by

$$H_i = I_i - AI_i , \qquad (3)$$

A task with higher unhealthy scores will have more negative impact on system performance.

### B. Task Scheduler

We have $m$ tasks and a multicore processor with $n$ L2 caches. Every two cores share a L2 cache and every task is associated with an unhealthy score that presents its interference level. The higher the unhealthy score is, the more execution time it has when the task is co-scheduled with other tasks. In order to reduce the total number of cache misses, our task scheduler considers both inter-core cache contention and intra-core cache contention. The inter-core cache contention is the competition among tasks that are executed on different cores but share the same cache. The intra-core cache contention is the competition among tasks that are executed on the same core. To reduce inter cache contention, we average unhealthy scores of tasks run on the shared caches. In order to reduce intra-core cache contention, the task scheduler groups the tasks with higher unhealthy scores on one core and the tasks with lower unhealthy scores on another core.

As Algorithm 1 shows, tasks are sorted by their unhealthy scores in descending order. Our task scheduler first determines $p$ which is the maximum number of tasks inside a core (line 1). Let $S_i$ demote the $i$-th shared cache. Based on $p$, our task scheduler determines the task executed on the first core ($C_0$) of each shared cache $S_i$ (line 3 to 9). We than determine the task executed on the second core ($C_1$) of each shared cache $S_i$ (line 10 to 17) by consider inter-core and intra-core cache contention.

**Algorithm 1** Task Scheduler Algorithm

```
1: p = ceiling of m/(2*n) ;
2: for i = 1 to n do
3:   if (m>p)
4:     assign the first p unhealthy tasks to C0 of Si ;
5:     m = m –p ; // remove dispatched tasks
6:   else
7:     assign tasks to C0 of Si and exit;
8:   endif
9: end for
10: for i = n to 1 do
11:  if (m>p)
12:    assign the first p unhealthy tasks to C1 of Si ;
13:    m = m- p ; //remove dispatched tasks
14:  else
15:    assign tasks to C1 of Si and exit ;
16:  endif
17: end for
```

## IV. EXPERIMENT

### 1) Experiment Setup

We adopted Intel Core2Quad Q8400 CPU for our experiment. The CPU has four cores running at 2.66GHz. As Table 1 shows, the four cores are arranged into two groups of two cores and each group shares a 2MB L2cache whose cache line size is 64 byes [9]. We implemented CUAS on Ubuntu with Linux kernel 2.6.32. The SPEC CPU2006 is selected for performance evaluation.

### 2) Effects on Task Scheduling:

In order to investigate the performance of our task scheduling, we increase the task number from four to eight. For example, WL1 includes two lbm, two libquntum, two bzip2 and two milc. We compare our method with CiPE [1] and Linux default scheduling. For the CiPE, tasks are first sorted by their CIS scores in descending order. The first task is co-scheduled with the last task. The second task is co-scheduled the second-last task and so on. As shown in Figure 1, for WL1 and WL2, there is a significant performance degradation if tasks are scheduled by CiPE because of serious inter-core cache contention between lbm and libquantum. On the contrary, we group lbm and libquantum so as to reduce cache contention. In WL2, our scheduling reduces total execution time by 46% compared to CiPE and by 43% compared to Linux default scheduling.

Table 1: The experiment workloads

| | **Combination of Benchmarks** | | | |
|---|---|---|---|---|
| WL1 | **1 cache sensitive, 3 cache insensitive** | | | |
| | lbm | libquantum | bzip2 | milc |
| WL2 | **2 cache sensitive, 2 cache insensitive** | | | |
| | lbm | libquantum | soplex | mcf |
| WL3 | **3 cache sensitive, 1 cache insensitive** | | | |
| | lbm | omnetpp | soplex | mcf |
| WL4 | **4 cache sensitive, 0 cache insensitive** | | | |
| | omnetp | bzip2 | astar | gcc |
| WL5 | **0 cache sensitive, 4 cache insensitive** | | | |
| | lbm | libquantum | milc | namd |

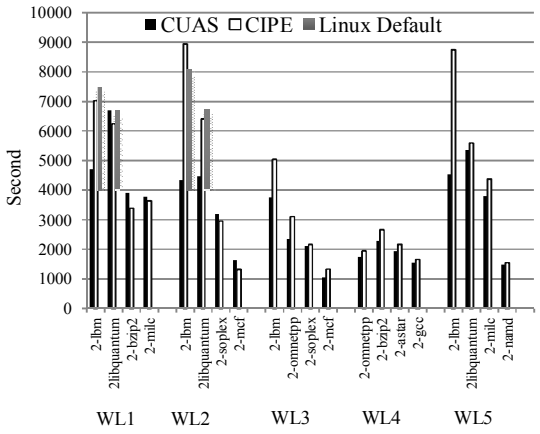


Figure 1 performance of different scheduling

## V. CONCLUSION

We develop a novel task scheduling CUAS to reduce shared cache contention based on two indexes, intra-core cache contention and task interference ability, that primarily determine the utilization of shared cached. CUAS first classifies tasks according to their anti-interference ability and interference ability. CUAS then distributes tasks to cores based on the effect of inter-core and intra-core cache contention. Our experiment results show that CUAS can significantly reduce shared cache contention and reduce total execution time at most 46% compared to existing methods.

## ACKNOWLEDGMENT

This work was supported in part by National Science Council of Taiwan under Grants NSC 100-2218-E-224-001-MY2 and 100-2219-E-224-001.

## REFERENCES

[1] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011, pages 167–176.

[2] Y. Xie and G. H. Loh. Dynamic classification of program memory behaviors in cmps. In *Proceedings of CMP-MSI*, 2008.

[3] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2010**,** pages 129–142.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Akula: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pages 249–260.

[6] T.-F. Yang, C.-H. Lin, and C.-L. Yang. Cache-aware task scheduling on multi-core architecture. *International Symposium on VLSI Design Automation and Test*, 2010, pages 139 – 143.

[7] S. Shekofteh, H. Deldari, and M. B. Khalkhali. Reducing cache contention in a multi-core processor via a scheduler. *International Conference on Advanced Computer Theory and Engineering*, 2010.

[8]J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, and S. Makineni. Cmpsched$im: Evaluating OS/CMP interaction on shared cache management. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[9] X. Jiang, A. Mishra, L. Zhao, R. Iyer, Z. Fang, S. Srinivasan, S. Makineni, P. Brett, and C. Das. Access: Smart scheduling for asymmetric cache cmps. *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (HPCA).

[10] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 2008, pages 54–66.

[11] X. Jia, J. Jiang, T. Zhao, S. Qi, and M. Zhang. Towards online application cache behaviors identification in cmps. *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, 2010, pages 1 – 8,.

[12] J. Calandrino and J. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems, pages 194 – 204*, 2009.

[13] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, *pages 423 – 432*.

[14] J. M. Calandrino and J. H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. *Euromicro Conference on Real-Time Systems*, 2008, pages 299 – 308.

[15] D. B.-S. David Eklov and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011, pages 147–157.

[16] R. P. Dicky, C. Xu, X. Chen, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems Software*, 2010, pages 76 – 86.