A Graph based approach for Co-scheduling jobs on Multi-core computers

Huanzhou Zhu and Ligang He

University of Warwick
Coventry, UK
`{zhz44, liganghe}@dcs.warwick.ac.uk`

Abstract

In a multicore processor system, running multiple applications on different cores in the same chip could cause resource contention, which leads to performance degradation. Recent studies have shown that job co-scheduling can effectively reduce the contention. However, most existing co-schedulers do not aim to find the optimal co-scheduling solution. It is very useful to know the optimal co-scheduling performance so that the system and scheduler designers can know how much room there is for further performance improvement. Moreover, most co-schedulers only consider serial jobs, and do not take parallel jobs into account. This paper aims to tackle the above issues. In this paper, we first present a new approach to modelling the problem of co-scheduling both parallel and serial jobs. Further, a method is developed to find the optimal co-scheduling solutions. The simulation results show that compare to the method that only considers serial jobs, our developed method to co-schedule parallel jobs can improve the performance by 31% on average.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Co-scheduling algorithm, Multicore processor, Cache interference, Parallel Job

Digital Object Identifier 10.4230/OASIcs.ICCSW.2013.144

1 Introduction

Multicore processors have now become a mainstream product in CPU industry. In a multicore processor, multiple cores reside on the same chip and share the resources in the chip. However, running multiple applications on different cores in the same chip could cause resource contention, which leads to performance degradation. Many research studies have shown that it is possible to isolate some resources, such as disk bandwidth [15], network bandwidth [8] for the co-running jobs. However, it is very difficult to isolate the on-chip last level cache (LLC). This problem is known as the shared cache contention and has been studied in literature [9,11,18]. The existing approaches to addressing on-chip shared cache contention fall into the following three categories: 1) Architecture-level solutions that focus on improving the hardware to provide isolation among threads [13] [14], 2) System-level solutions that focus on partitioning the cache for each application [16] [12], and 3) Software-level solutions that tend to develop the contention-aware scheduler to reduce the contention [5] [7]. In the above three categories, the architecture-level solution is still under active development by the processor vendors. The cache partitioning solution requires many changes in the existing system-level software (such as operating system), and therefore incurs high implementation cost. The third approach, the contention-aware schedulers, is a fairly lightweight approach, and therefore attracts many researchers' attention, which is also the focus of this work.

© Huanzhou Zhu and Ligang He;
licensed under Creative Commons License CC-BY
2013 Imperial College Computing Student Workshop (ICCSW'13).
Editors: Andrew V. Jone, Nicholas Ng; pp. 144–151

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ICCSW

A number of contention-aware co-schedulers have been developed in the literature [1, 4, 7]. These studies demonstrated that contention-aware schedulers can deliver better performance than the conventional schedulers. However, they do not aim to find the optimal co-scheduling performance. It is very useful to know the optimal co-scheduling performance. With the optimal performance, the system and co-scheduler designers can know how much room there is for further performance improvement. In addition, knowing the distance between current performance and optimal performance can help the scheduler designers to make the tradeoff between scheduling efficiency and scheduling quality.

The co-schedulers discussed in literature only consider serial jobs (each of which runs on a single core), and do not take parallel jobs into account. However, both parallel jobs and serial jobs often exist in a multicore computer system. For example, both parallel and serial jobs are submitted to a cluster consisting of multi-core computers.

The work in [9] modelled the optimal co-scheduling problem for serial jobs as an integer programming problem. However, we will show in this paper (Section 2) that this modelling approach cannot be extended to parallel jobs. This motivates us to develop a new method that is flexible to model the problem of co-scheduling both serial and parallel jobs.

In this paper, we first present a new approach to modelling the problem of co-scheduling both parallel and serial jobs. Further, a method is developed to find the optimal co-scheduling solutions.

We have conducted the simulation experiments to evaluate the co-scheduling algorithms we developed. The results show that taking parallelism into account can significantly improve performance. More specifically, if the method developed for serial jobs is used to co-schedule a mix of parallel and serial jobs, the performance achieved by the new method that takes parallel jobs into account is 31% better on average than only considering serial jobs.

The rest of the paper is organized as follows. Section 2 formalizes the problem of co-scheduling a mix of parallel and serial jobs. Section 3 presents a method to find the optimal co-scheduling solutions. The experimental results are presented in Section 4. Finally, the paper is concluded in Section 6.

2 Formalizing the problem of co-scheduling parallel jobs

The work in [9] i) formalized the problem of co-scheduling serial jobs, and ii) proposed an approach to modelling and finding the optimal co-scheduling solution. In this section, we first briefly summarize their formalization method in Subsection 2.1, then in Subsection 2.2 extend the method to incorporate parallel jobs, and present our own approach to modelling the problem of co-scheduling a mix of parallel and serial jobs, and developing the methods to solve the model for optimal co-scheduling solutions.

2.1 Formalizing the problem of co-scheduling serial jobs [9]

The work in [9] shows that on a multicore processor, the co-running jobs are generally slower than when they run alone due to resource contention. This performance degradation is called co-run degradation. The co-run degradation of a job is defined as the difference between the execution time of the job when it co-runs with a set of other jobs and its execution time when it runs alone. Formally, the performance degradation of a job i is expressed in Eq. 1, where ft_i is the execution time when job i runs alone, S is a set of jobs and $ft_{i,S}$ is the

execution time when job i co-runs with the set of jobs in S.

$$D_i = \frac{ft_{i,S} - ft_i}{ft_i} \quad (1)$$

In the co-scheduling problem, n jobs need to be allocated to a cluster of u-core multiprocessors so that each core is allocated with one job. m denotes the number of u-core multiprocessors needed, which can be calculated as $\lceil \frac{n}{u} \rceil$. The objective of the co-scheduling problem is to find the optimal way to partition n jobs into m u-cardinality sets, so that the sum of D_i in Eq. 1 over all n jobs is minimized, which is shown in Eq. 2. Note that if n is not the multiple of u, i.e., $n\%u \neq 0$, we can simply generate $u - n\%u$ imaginary jobs whose performance degradation with any job is 0.

$$min \sum_{i=1}^{|n|} D_i \quad (2)$$

2.2 Formalizing the problem of co-scheduling parallel jobs

As defined in Eq. 2, in order to find the optimal co-scheduling solution, the objective is to minimize the sum of the performance degradation experienced by each job. This objective function is designed for co-scheduling serial jobs. A parallel job consists of multiple processes (or threads). Applying Eq.2 directly to the case involving parallel jobs, the total degradation of a mix of parallel and serial jobs is the sum of the degradation experienced by each process in all parallel jobs plus the sum of the degradation by each serial job. However, the finishing time of a parallel job is determined by its slowest process. A larger degradation for a process indicates a longer execution time for that process. Therefore, no matter how small the degradation is for other processes, they have to wait until the process with the largest degradation finishes, which essentially means that all processes suffer the same degradation as the largest degradation. Thus, the total degradation for a parallel job is the largest degradation among all degradations experienced by its processes multiplied by the number of the processes in the parallel job, which can be formally defined as $M_i \times max(D_{ij})$, where M_i is the number of processes in parallel job i and D_{ij} is the degradation of process j in parallel job i. Based on this analysis, we re-formulate the objective function of finding the optimal co-scheduling solution for a set of jobs containing parallel jobs. The objective function is expressed in Eq. 3, where J is a set of all jobs, PJ is the set of parallel jobs in J, $|J|$ and $|PJ|$ represent the number of the jobs in the set J and PJ, respectively.

$$min(\sum_{i=1}^{|PJ|}(M_i \times max\{D_{ij}\}) + \sum_{i=1}^{|J|-|PJ|} D_i) \quad (3)$$

In order to solve this problem, we propose a new method to model the problem of co-scheduling serial jobs. The new modelling approach is flexible and can be extended to incorporate parallel jobs.

3 Modelling the co-scheduling problem

3.1 Graph Model of the problem

As formalized in Section 2, the objective of solving the co-scheduling problem is to find a way to partition n jobs, j_1, j_2, ..., j_n, into m u-cardinality sets, so that the total performance

degradation of all jobs is minimized. The number of all possible u-cardinality sets is $\binom{n}{u}$. In this paper, a graph called the co-scheduling graph is constructed, to model the co-scheduling problem. There are $\binom{n}{u}$ nodes in the graph and a node corresponds to a u-cardinality set. The ID of a node is coded a u-digit number, using the IDs of the jobs in the corresponding u-cardinality set. In the encoding, the job IDs are always placed in an ascending order from the most to the least significant digit. The weight of a node is defined as the total performance degradation of the u jobs in the node. The nodes are organized into different levels in the graph. The i-th level contains all nodes whose first digit of the ID is i. In each level, the nodes are placed in the ascending order of their ID's. A *start* node and an *end* node are added as the first level (level 0) and the last level of the graph, respectively. The weights of the start and the end nodes are both 0. Figure 1 illustrates when co-scheduling 6 jobs to 2-core processors, how to code the nodes in the graph, and how to organize the nodes into different levels. Note that for the clarity of the figure we did not draw all edges.

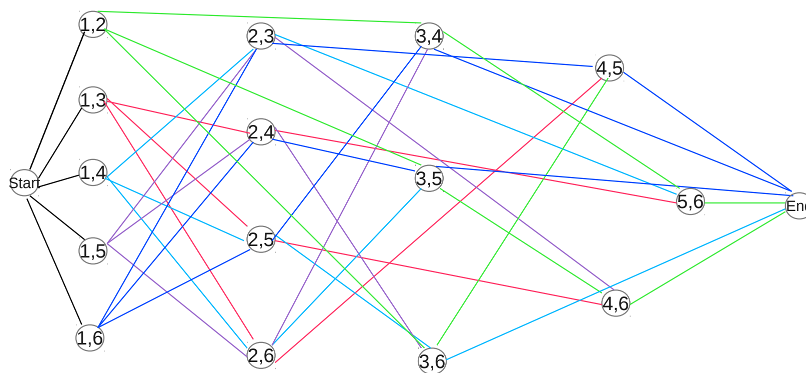

Figure 1 Degradation graph for 6 jobs on dual core system, the number in each node represents a Job ID, and edges with same color forms a group of possible schedule.

3.2 Optimal Parallel aware Shortest Path algorithm

A path from the start to the end node in the graph forms a co-scheduling solution if the path does not contain duplicated jobs, which is called a valid path. The distance of a path is defined as the sum of the weights of all nodes on the path. Finding the optimal co-scheduling solution is equivalent to finding the shortest valid path from the start node to the end node. In this paper, a algorithm, called OP-SCG (Optimal Parallel aware Shortest path algorithm for the Co-scheduling Graph) is developed to find the shortest valid path in the constructed graph. OP-SCG is adapted from Dijkstra's shortest path algorithm [3]. The main differences between OP-SCG and Dijkstra's algorithm lie in three aspects: 1) there are no edges between nodes in the graph in OP-SCG, and the edges are established as the algorithm progresses, 2) the invalid paths, which contain the duplicated jobs, have to be disregarded, and 3) the ability to compute degradation of parallel jobs.

In this algorithm, every node v of the graph contains some attributes, in which the *v.distance* attribute records the length of the shortest path from the start node to node v, the *v.path* attribute is a list and records the sequence of nodes in the shortest path up to v. The purpose of this design is to avoid spending time checking whether adding a new node will invalidate the resulting path.

In Algorithm 1, object Q is a list that holds jobs in ascending order of all paths that have been visited by the algorithm. For example, if the algorithm visited the paths [(1,3),(2,4)] and

Algorithm 1 The OP-SCG Algorithm

```
1  for every node v in the graph do
2      v.distance = 0;
3      v.previous = NULL;
4      Q = start.ID;
5      v = start;
6      while v != end
7          for every level l from v.level + 1 to end.level do
8              if l is not a digit of the ID of the nodes in v.path
9                  valid_l = l;
10         for every node k in the valid_l level do
11             if the jobs in node k are not in v.path:
12                 if k contains parallel job:
13                     label = compute_label(k, v + v.previous);
14                 else:
15                     label = v.distance + k.weight;
16             if k + v.path is not in Q or label < Q[k + v.path]:
17                 Q[k + v.path] = label;
18                 k.path = v.path + k;
19     remove node v from Q;
20     obtain such a node v from Q that has the smallest v.distance;
```

[(1,3),(2,5)], the data stored in Q is [(1234),(1235)]. For every node it visits, the algorithm searches for a valid level (Line 7-9), which is a level that contains at least one node that can form a valid path with the nodes in the current partial path. After a valid level is found, Algorithm 1 continues to search this level for the valid nodes that can form a valid path (Line 10-11). If a valid node, k, is found, the algorithm further checks if there are any parallel jobs in this node, and calculates the distance with function *compute_label* if the node k contains parallel jobs. Otherwise, the distance is computed by adding the previous distance with weight of k (Line 12). If node k has not been visited yet or the new path will form a shorter path (Line 16), the algorithm either adds node k to Q or updates the distance stored in $Q[k]$ (Line 16-18).

Also, by removing Line 12-14 from Algorithm 1, we obtain another algorithm, called B-SCG (Basic Shortest path algorithm for the Co-scheduling Graph) which is used to find the shortest path in the graph when there are only serial jobs.

4 Evaluation

In this section, we present the scheduling results produced by our optimal algorithm through a set of simulation tests. The configuration of simulation environment is based on Intel Core 2 series processor with 2 cores, 4 cores, 6 cores respectively, all running 2.66GHz, and having one shared 6MB, 4096 sets*24 way set associative last level cache.

4.1 Co-scheduling Result

In order to examine the capability of OP-SCG algorithm, we compare the results produced by B-SCG algorithm with it. The simulation is conducted by scheduling a set of artificially generated jobs. The degradation value of every node in the co-scheduling graph is computed by prediction model developed by Dhruba et.al [2]. As required by Dhruba's method, a stack distance profile with randomly generated cache hits and misses is assigned to each job, since we are interested in difference between results produced by two algorithms, the randomness has negligible influence on this experiment since both algorithms will operate on same job set.

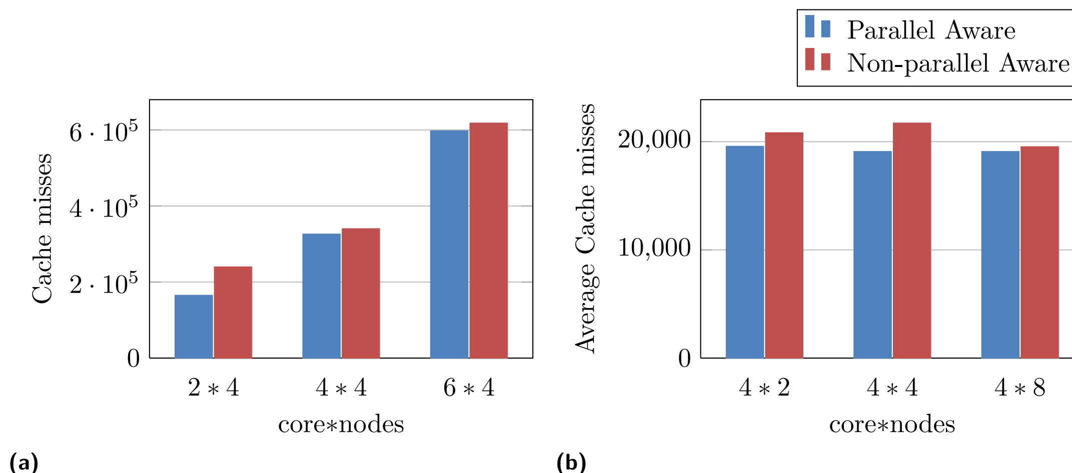

Figure 2 Changing core number and node number.

The first simulation tested the correctness of mathematical model and OP-SCG algorithm. This simulation was conducted by scheduling 8, 16, 24 jobs to 4 nodes cluster with 2, 4, and 6 cores respectively, half of which were parallel processes. The result is shown in Figure 2a. The metric used in this experiment is the total cache misses of each job. It is worthwhile to note that lower cache misses suggests lower performance degradation.

As shown in the Figure 2a, the parallel aware optimal scheduler produces lower cache misses in all cases due to consideration of parallelism. The average distance between the two algorithms is 31%. This result not only demonstrates the correctness of the algorithm presented in this paper but also proves that considering parallelism can significantly improve performance. In addition, the result also shows that the cache misses increase as the core number increases. Because the cache size does not increase as core number increases, the degradation increases since there are more jobs competing for the limited number of cache lines.

Since the basic principle of co-scheduling algorithm is to balance the on-chip shared resource usage, the second experiment was conducted to examine this ability of algorithm presented in this paper. In this experiment, 8, 16, 32 jobs with 50% parallel processes were scheduled to clusters with 2, 4, and 8 nodes respectively, each node has a quad-core processor. The results are shown in Figure 2b.

The metric used in this experiment is average cache misses of all jobs, the parallel aware optimal scheduler produced the lowest average cache misses among three cases again. The average cache misses between three cases are very similar, the difference between the highest and lowest result being 3%. This result suggests that the algorithm balanced the resource usage among every node within the cluster well, which also means that the fairness of this algorithm is good.

5 Discussion

The primary goal of this paper is to provide the theoretical insight for finding optimal co-scheduling with parallel job considered. Apart from that, practical co-scheduler designs can benefit from this work in two ways: Firstly, the optimal model presented in this paper provided a sufficient way of evaluating the co-scheduling results when parallel jobs are

considered. Knowing the optimal solution is important for practical co-schedule system design, because the tradeoff between efficiency and quality can be made based on knowledge about the distance between current results and the optimal solution. Secondly, the algorithm proposed in this work can be directly used in a proactive co-scheduling system. Predicting the co-run performance has been widely studied by many researchers (e.g., [2] [6] [10] [17]). Those studies make it possible to predict co-run performance accurate and efficiently. With accurate prediction, the proactive schedulers may use the algorithm proposed in this work to determine the optimal or near-optimal schedules.

In this work, we assumed that each core will execute only single job, however, the effectiveness of our approach is not strictly limited by this assumption. If there are multiple jobs running on same core, they are more likely to be executed in a time-sharing basis, therefore, our algorithm will still be an essential component for finding the optimal schedule in each time slice. The scheduler can re-schedule jobs at time slice boundaries.

6 Conclusion

This paper explored the problem of parallel aware optimal job co-scheduling on multiprocessor system. The paper built a mathematical model that can be used to find the optimal scheduling with consideration of parallel jobs. Based on this model the paper described an optimal parallel aware co-scheduling algorithm (OP-SCG) for multicore processor systems by formulating the problem as a shortest path problem. The experiment results show that by considering parallelism, the parallel aware optimal algorithm decreases the average performance degradation by 31%.

The mathematical model and algorithm in this paper offer the theoretical and practical support for the evaluation of co-scheduling systems.

References

1 Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4):8, 2010.

2 Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

3 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001.

4 Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57, 2010.

5 Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

6 Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE Computer Society, 2007.

7 J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-hierarchy contention aware scheduling in cmps. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2013.

8 Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*, pages 342–362. Springer, 2006.

9 Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, and Rahul Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1192–1205, 2011.

10 Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

11 Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 200–209. IEEE, 2007.

12 Min Lee and Karsten Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 451–462. ACM, 2012.

13 Kyle J Nesbit, James Laudon, and James E Smith. Virtual private caches. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 57–68. ACM, 2007.

14 Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ACM Sigplan Notices*, 43(3):135–144, 2008.

15 Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage (TOS)*, 2(3):283–308, 2006.

16 Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

17 Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. *ACM SIGPLAN Notices*, 46(7):27–38, 2011.

18 Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 129–142. ACM, 2010.