Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors

Yunlian Jiang  
Computer Science Department  
College of William and Mary, VA, USA  
jiang@cs.wm.edu

Xipeng Shen  
Computer Science Department  
College of William and Mary, VA, USA  
xshen@cs.wm.edu

Chen Jie  
Scientific Computing Group  
Thomas Jefferson National Accelerator Facility, VA, USA  
chen@jlab.org

Rahul Tripathi  
Computer Science Department  
University of South Florida, FL, USA  
tripathi@cse.usf.edu

ABSTRACT
Cache sharing among processors is important for Chip Multiprocessors to reduce inter-thread latency, but also brings cache contention, degrading program performance considerably. Recent studies have shown that job co-scheduling can effectively alleviate the contention, but it remains an open question how to efficiently find optimal co-schedules. Solving the question is critical for determining the potential of a co-scheduling system. This paper presents a theoretical analysis of the complexity of co-scheduling, proving its NP-completeness. Furthermore, for a special case when there are two sharers per chip, we propose an algorithm that finds the optimal co-schedules in polynomial time. For more complex cases, we design and evaluate a sequence of approximation algorithms, among which, the hierarchical matching algorithm produces near-optimal schedules and shows good scalability. This study facilitates the evaluation of co-scheduling systems, as well as offers some techniques directly usable in proactive job co-scheduling.

Categories and Subject Descriptors

General Terms
Algorithms, Performance, Experimentation

Keywords
co-scheduling, CMP scheduling, cache contention, perfect matching

1. INTRODUCTION
In Chip Multiprocessors (CMP), it is common for several cores to share a cache. The sharing is important for reducing inter-thread latency, but also brings cache contention between co-running jobs. Many studies have shown considerable and sometimes significant effects of the contention on program performance and system fairness [2, 7–9, 12, 22, 31]. The urgency for alleviating the contention keeps growing as the processor-level parallelism rapidly increases.

Some of the recent research has attempted to address the cache contention by cleverly co-scheduling jobs. Most of the techniques use reactive co-scheduling. The runtime system periodically changes the co-runners (i.e., the jobs sharing a cache) of a job to estimate its cache requirement (e.g., [9]) and co-run performance (e.g., [28]). The scheduler then changes the assignment of the jobs accordingly to group compatible jobs to the same chip to reduce cache contention. Besides reactive scheduling, some research tries to predict co-run performance of programs (e.g., [14, 16]), which opens the opportunities for proactively co-scheduling jobs without the need for runtime trials.

A problem that faces both classes of co-scheduling but remains unanswered, is the computation of optimal co-schedules. Finding optimal co-schedules is important for two reasons. First, it facilitates the evaluation of various scheduling systems. Without knowing optimal schedules, it is hard to precisely determine how good a scheduling algorithm is—how far the scheduling algorithm is from the optimal ones and whether further improvement will enhance performance significantly. Second, efficient optimal co-scheduling algorithms can directly fit the need of proactive co-scheduling. After getting the prediction of co-run performance, the scheduler can use the algorithms to find the schedule that minimizes co-run degradation. For runtime reactive co-scheduling, those algorithms may serve as the base for the development of online lightweight co-scheduling systems.

In this work, we tackle the optimal co-scheduling problem from three aspects. First, we analyze the complexity of the problem and prove that it is NP-complete on k-core processors, when k is greater than $2^1$. Second, we present a polynomial-time algorithm for finding the optimal co-schedules on dual-core CMPs (job lengths and reschedul-

$^1$We assume all cores on a chip share a cache.
ing are ignored). The algorithm first constructs a degradation graph, and then treats the optimal scheduling problem as a minimum-weight perfect matching problem and solves it using blossom algorithm [6].

Finally, we develop a series of approximation algorithms for more complex CMP systems. The first algorithm, named hierarchical matching algorithm, generalizes the dual-core algorithm to partition jobs in a hierarchical way. The second algorithm, named greedy algorithm, schedules jobs in the order of their sensitivities to cache contention. To further enhance the scheduling quality, we develop an efficient local optimization scheme that is applicable to the schedules produced by both algorithms. We evaluate the various scheduling algorithms on 16 programs running on some recent quad-core machines. The results show that the hierarchical matching algorithm reduces the average co-run degradation from 19.81% to 5.21%, only 1.37% away from the optimal. Meanwhile, the near-optimal schedule improves scheduling fairness significantly.

In the rest of this paper, Section 2 defines the problem setting. Section 3 proves the NP-completeness of optimal scheduling on general CMP systems, and presents the polynomial-time algorithm for dual-core systems. Section 4 presents a set of approximation algorithms for more complex systems. Section 5 reports the experimental results. Section 6 discusses the limitations of this work and future extensions. Section 7 reviews the related work, followed by a short summary.

2. PROBLEM SETTING AND NOTATIONS

Programs co-running on a CMP may run slower than their single runs. This degradation is called co-run degradation, quantified in the following formula:

\[
c_{\text{run}} \text{ degradation of job } i = \frac{c\text{CPI}_i - s\text{CPI}_i}{s\text{CPI}_i},
\]

where \( c\text{CPI}_i \) and \( s\text{CPI}_i \) are the numbers of cycles per instruction when job \( i \) has or has no cache sharers (i.e., jobs co-running on the same cache) respectively.

The problem of co-scheduling includes two parts. The first is to predict the degradation of every possible co-run. The second is to find the optimal schedule so that the total degradation is minimized given the predicted co-run degradations. Much research has explored the first part of the problem (e.g., [9, 16, 28]). This work specially focuses on the second part, in which, we assume that the degradations of all possible co-runs are known beforehand (although some algorithms to be presented do not require the full knowledge).

To avoid the distractions from other complexities, such as difference between program execution times and context switches in OS, in this work we consider the following scenario. There are \( N \) single-process jobs of the same length to be assigned to \( N \) cores, one job per core. The assignment is static, i.e., no reassignment occurs during the execution of the jobs. Every \( K \) (a factor of \( N \)) cores reside on a chip, sharing a cache. So there are \( I = N/K \) chips. The goal is to find the assignment that minimizes the total degradation of all jobs, expressed formally as follows:

\[
\min \sum_{i=1}^{N} \frac{c\text{CPI}_i - s\text{CPI}_i}{s\text{CPI}_i}.
\]

We stress that this simplified problem keeps the fundamental challenges of the general co-scheduling problem. The influence of the other complexities is discussed in Section 6.

In the following description, we use an assignment to refer to a group of \( K \) jobs that are to run on the same chip. We use schedule to refer to a set of assignments that cover all the jobs and have no overlap with each other. In another word, a schedule is a solution of co-scheduling.

3. OPTIMAL SOLUTIONS

This section analyzes the time complexity of optimal co-scheduling. It proves that the optimal solution can be found in polynomial time for dual-core chips (i.e., \( K = 2 \)); when \( K \geq 3 \), the problem becomes NP-complete.

3.1 Polynomial Solution \((K = 2)\)

On a system with dual-cores \((K = 2)\), we model optimal co-scheduling as a graph problem. The graph is a fully connected graph, named degradation graph. Every vertex represents a job. The weight on each edge equals to the sum of the degradations of the jobs represented by the two vertices when they run on the same chip. Figure 1 shows such a graph.

We model optimal co-scheduling as a minimum-weight perfect matching problem. A perfect matching in a graph is a subset of edges that cover all vertices, but no two edges share a common vertex. A minimum-weight perfect matching problem is to find a perfect matching that has the minimum sum of edge weights in a graph.

We argue that a minimum-weight perfect matching in a degradation graph corresponds to an optimal co-schedule of the job set represented by the graph vertices. First, a valid job schedule must be a perfect matching in the graph. Each resulting job group corresponds to an edge in the graph, and the groups should cover all jobs and no two groups can share the same job, which exactly match the conditions of a perfect matching. On the other hand, a minimum-weight perfect matching minimizes the sum of edge weights, which is equivalent to minimizing the objective function of co-schedule, expressed in Equation (1).
One of the fundamental discoveries in combinatorial optimization is the polynomial-time blossom algorithm for finding minimum-weight perfect matchings proposed by Edmonds [6]. The time complexity of the algorithm is \( O(n^2 m) \), where \( n \) and \( m \) are respectively the numbers of nodes and edges in the graph. Later, Gabow and Tarjan develop an \( O(nm + n^2 \log n) \) time algorithm [10]. Cook and Rohe provide an efficient implementation of the blossom algorithm [3].

3.2 Proof of NP-Completeness (\( K \geq 3 \))

When \( K \geq 3 \), the problem becomes an NP-complete problem. This section proves the NP-completeness via a reduction of a known NP-complete problem, \textit{Multidimensional Assignment Problem} (MAP) [11] to the co-scheduling problem.

First, we formulate the co-scheduling problem as follows. There is a set \( S \) containing \( N \) elements. (Each element corresponds to a job in the co-scheduling problem.) Let \( S_k \) represent the set of all \( K \)-cardinality subsets of \( S \). Each of those \( K \)-cardinality subsets, represented by \( G_i \), has a weight \( w_i \), where \( i = 1, 2, \ldots, \binom{N}{K} \). \( G_i \) corresponds to a group of jobs scheduled to the same chip, and its weight corresponds to the sum of the degradation of all the jobs in the group.

The objective is to find \( N/K \) such subsets, \( G_{p_1}, G_{p_2}, \ldots, G_{p_{N/K}} \) to form a partition of \( S \) to satisfy the following conditions:

- \( \bigcup_{i=1}^{N/K} G_{p_i} = S \). (Every job belongs to a subset.)
- \( \sum_{i=1}^{N/K} w_{p_i} \) is minimized. (Total weight is minimum.)

The first condition ensures that every job belongs to a single subset and no job can belong to two subsets (as all the subsets together contain only \( N/K \times K = N \) jobs). The second condition ensures that the total weight of the subsets is minimum.

We prove that this problem is NP-hard via the reduction from the MAP problem. The objective of the MAP is to match tuples of objects in more than 2 sets with minimum total cost. The formal definition of MAP is as follows:

- Input: \( K \) (\( K \geq 3 \)) sets \( Q_1, Q_2, \ldots, Q_K \), each containing \( M \) elements, and a cost function \( C: Q_1 \times Q_2 \times \cdots \times Q_K \rightarrow R \).
- Output: An assignment \( A \) that consists of \( M \) subsets, each of which contains exactly one element of every set \( Q_k \), \( 1 \leq k \leq K \). Every member \( \alpha_m = \{a_{m1}, a_{m2}, \ldots, a_{mk}\} \) of \( A \) has a cost \( c_m = C(\alpha_m) \), where \( 1 \leq m \leq M \) and \( a_{mk} \) is the element chosen from the set \( Q_k \).
- Constraints: Every element of \( Q_k \), \( 1 \leq k \leq K \), belongs to exactly one subset of assignment \( A \) and \( \sum_{m=1}^{M} c_m \) is equal to a given value \( O \).

MAP has been proven to be NP-complete by reduction from the three-dimensional matching problem [11], a well-known NP-complete problem first shown by R. Karp [15].

We now reduce MAP to the co-scheduling problem. Given an instance of MAP, we construct a co-scheduling problem as follows:

- Let \( S = \bigcup_{k=1}^{K} Q_k \) and \( N = M \times K \).
- Build all the \( K \)-cardinality subsets of \( S \), represented as \( G_i, 1 \leq i \leq \binom{N}{K} \). If a subset \( G_i \) contains exactly one element from every set \( Q_k, 1 \leq k \leq K \), its weight is set as \( C(a_1, a_2, \ldots, a_K) \), where \( C \) is the cost function in the MAP instance, and \( a_k \) is an element chosen from \( Q_k \). Otherwise the weight is set to positive infinity.

For a given value of \( K \), the time complexity of the construction is \( O(N^K) \). It is clear that a solution to this co-scheduling problem is also a solution to the MAP instance and vice versa. This proves that the co-scheduling problem is NP-hard. Obviously, the co-scheduling problem is an NP problem. Hence, the co-scheduling problem is an NP-complete problem when \( K \geq 3 \).

4. APPROXIMATION ALGORITHMS

Driven by the NP-completeness, we design a series of heuristic-based algorithms to efficiently approximate the optimal schedules. The first algorithm is a hierarchical extension to the polynomial-time algorithm used when \( K = 2 \); the second is a greedy algorithm, selecting the local minimum in every step. In addition, we introduce a local optimization algorithm to further improve the schedule results.

4.1 Hierarchical Perfect Matching Algorithm

The hierarchical perfect matching algorithm is inspired by the solution on dual-core systems. For the purpose of clarity, we first describe the way this algorithm works on quad-core CMPs, and then present the general algorithm in detail.

Finding the optimal co-schedule on quad-core CMPs is to partition the \( N \) jobs into \( N/4 \) 4-member groups. In this algorithm, we first treat a quad-core chip with a shared cache \( L \) as 2 virtual chips, each of which contains a dual-core processor and an \( L/2 \) shared cache. On the virtual dual-core system, we apply the perfect matching algorithm to find the optimal schedule, in which, all jobs are partitioned into \( N/2 \) pairs. Next, we create a new degradation graph, with each vertex representing one of the \( N/2 \) pairs. After applying the minimum-weight perfect matching algorithm to the new graph, we will obtain \( N/4 \) pairs of job pairs, or in another word, \( N/4 \) 4-member job groups. These groups form an approximation to the optimal co-schedule on the quad-core system.

Using this hierarchical algorithm, we can approximate the optimal solution of \( K \)-core co-scheduling problem by applying the minimum perfect matching algorithm \( \log K \) times, as shown in Figure 2. At each level, say level-\( k \), the system is viewed as a composition of \( 2^k \)-core processors. At each step, the algorithm finds the optimal coupling of the job groups that are generated in the last step. Figure 3 shows the pseudo-code of this algorithm. Notice that, even though this hierarchical matching algorithm invokes the minimum-weight perfect matching algorithm \( \log K \) times, its time complexity is the same as that of the basic minimum perfect matching algorithm, \( O(N^2) \), thanks to that the number of vertices in the degradation graphs decreases exponentially.

4.2 Greedy Algorithm

The second approximation algorithm is a greedy algorithm. Our initial design is as follows. We first sort all of the \( K \)-cardinality sets of jobs in an ascending order in
S

erty, the top sets are likely to contain only those "friendly" co-runners, and "robust"—suffering less from its co-runners.

tends to be both "polite"—causing less degradation to its

Compared to other jobs, a job that uses little shared cache

produced schedules are among the worst possible schedules.

After revisiting the algorithm, we recognize the problem.

The major overhead in this greedy algorithm includes the
calculation of politeness and the construction of the final
schedule. This politeness-based greedy algorithm manages to assign "unfriendly" jobs with "friendly" ones, the "friendly" jobs won’t degrade much more than they do in the naive greedy schedule, whereas, the "unfriendly" programs will degrade much less.

Based on this rationale, we change the naive greedy algo-
rithm to the following. We first compute the politeness of a job, which is defined as the reciprocal of the sum of the degradations of all co-run groups that include that job. During the construction of the schedule $S$, each time we add a co-run group that satisfies the following two conditions: 1) One member of the group is the job whose politeness is the smallest in the unassigned job list; 2) the total degradation of the group is minimum. Figure 4 exhibits the pseudo-code of this algorithm. This politeness-based greedy algorithm manages to assign "unfriendly" jobs with "friendly" ones and proves to be much better than the naive greedy algorithm.

The major overhead in this greedy algorithm includes the calculation of politeness and the construction of the final schedule. Both have $O\left(\binom{N}{K}\right)$ time complexity, so the greedy algorithm’s time complexity is $O\left(\binom{N}{K}\right)$.

4.3 Local Optimization

To further enhance the schedules generated by both heuristic-based algorithms, we propose a local optimization algorithm. The algorithm adjusts a schedule by reassigning the jobs in every two assignments in the schedule.

The adjustment works in this way. Let $J$ represent the jobs included in two assignments. The algorithm checks all the possible ways to evenly partition the jobs into two groups and chooses the best one as the schedule for these jobs. The algorithm conducts the reassignment to every two assignments in a given schedule. Figure 5 shows the pseudo-code.

The optimization on two assignments needs to check $\binom{2K}{K}$/2 assignments. The algorithm in Figure 5 requires $\binom{N}{K}^2/2$ iterations. Therefore, the time complexity for this local optimization is $O\left(\binom{N}{K}^2\binom{2K}{K}\right)$.
5. Evaluation

In this section, we present the evaluation of the approximation algorithms for scheduling 16 programs running on a set of AMD Opteron quad-core machines. After reporting the performance and fairness of the schedules produced by those algorithms, we show the comparison of their time complexities, and present the scalability study.

Each evaluation machine is equipped with two quad-core AMD Opteron processors running at 1.9 GHz. Each core has 512KB dedicated L2 cache and shares a 2MB L3 cache with the other three cores. The 16 test programs consist of 15 randomly selected benchmarks from SPEC CPU2000 and a stream program derived from [18] (Table 1). Including a stream program is to cover the special behavior of such programs.

We obtain the optimal schedule by conducting an exhaustive search. The total number of possible schedules is 2,627,652. The search time increases exponentially as the numbers of jobs and cores increase. We obtain the random scheduling result by applying 1000 random schedules to the jobs and getting the average performance. The random scheduling result corresponds to the performance of current CMP schedulers, which are oblivious to cache contention.

5.1 Coscheduling Performance

Using the collected degradations, we measure the effectiveness of the scheduling algorithms by a comparison of four types of schedules: the optimal, the random, the hierarchical perfect matching, and the greedy schedules, along with the enhanced version of the latter two when local optimization is applied. The metric we use is the average performance degradation of all programs.

We obtain the optimal schedule by conducting an exhaustive search. The total number of possible schedules is 2,627,652. The search time increases exponentially as the numbers of jobs and cores increase. We obtain the random scheduling result by applying 1000 random schedules to the jobs and getting the average performance. The random scheduling result corresponds to the performance of current CMP schedulers, which are oblivious to cache contention.

5.1.1 Approximation without Local Optimization

To concentrate on the effectiveness of the two approximation algorithms, this section reports their results when local optimization is not applied. Table 2 presents an optimal scheduling result by applying 1000 random schedules to the jobs and getting the average performance. The random scheduling result corresponds to the performance of current CMP schedulers, which are oblivious to cache contention.

We obtain the optimal schedule by conducting an exhaustive search. The total number of possible schedules is 2,627,652. The search time increases exponentially as the numbers of jobs and cores increase. We obtain the random scheduling result by applying 1000 random schedules to the jobs and getting the average performance. The random scheduling result corresponds to the performance of current CMP schedulers, which are oblivious to cache contention.

5.1 Coscheduling Performance

Using the collected degradations, we measure the effectiveness of the scheduling algorithms by a comparison of four types of schedules: the optimal, the random, the hierarchical perfect matching, and the greedy schedules, along with the enhanced version of the latter two when local optimization is applied. The metric we use is the average performance degradation of all programs.

\[
\text{LocalOpt (S) \{ for } i \leftarrow 1 \text{ to } M - 1 \text{ \{ for } j \leftarrow i + 1 \text{ to } M \text{ \{ a }_1 \leftarrow S[j]; (a'_1, a'_2) \leftarrow \text{Opt2Assignments}(a_1, a_2); a'_1 = a'_2; S[j] = a'_2; \} \} S[i] = a_1; \}
\]

Figure 5: Local Optimization

<table>
<thead>
<tr>
<th>Programs</th>
<th>min %</th>
<th>max %</th>
<th>mean %</th>
<th>median %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>0</td>
<td>79.97</td>
<td>5.12</td>
<td>2.93</td>
</tr>
<tr>
<td>applu</td>
<td>0</td>
<td>165.76</td>
<td>10.30</td>
<td>7.07</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>174.65</td>
<td>19.44</td>
<td>15.09</td>
</tr>
<tr>
<td>bzip</td>
<td>0</td>
<td>55.90</td>
<td>15.17</td>
<td>13.35</td>
</tr>
<tr>
<td>crafty</td>
<td>0</td>
<td>149.90</td>
<td>5.11</td>
<td>3.18</td>
</tr>
<tr>
<td>equake</td>
<td>0.32</td>
<td>191.77</td>
<td>27.08</td>
<td>18.35</td>
</tr>
<tr>
<td>facerec</td>
<td>0</td>
<td>192.20</td>
<td>23.30</td>
<td>17.98</td>
</tr>
<tr>
<td>gap</td>
<td>0</td>
<td>198.41</td>
<td>11.31</td>
<td>7.40</td>
</tr>
<tr>
<td>gzip</td>
<td>0</td>
<td>57.76</td>
<td>0.79</td>
<td>0.00</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>191.49</td>
<td>60.41</td>
<td>56.83</td>
</tr>
<tr>
<td>mesa</td>
<td>0</td>
<td>51.77</td>
<td>0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>parser</td>
<td>0</td>
<td>87.14</td>
<td>8.46</td>
<td>5.88</td>
</tr>
<tr>
<td>stream</td>
<td>0</td>
<td>93.23</td>
<td>28.55</td>
<td>24.43</td>
</tr>
<tr>
<td>swim</td>
<td>0.84</td>
<td>176.32</td>
<td>18.85</td>
<td>15.23</td>
</tr>
<tr>
<td>twolf</td>
<td>0</td>
<td>182.89</td>
<td>57.05</td>
<td>54.44</td>
</tr>
<tr>
<td>vpr</td>
<td>0</td>
<td>83.42</td>
<td>24.78</td>
<td>21.66</td>
</tr>
<tr>
<td>average</td>
<td>0.07</td>
<td>133.29</td>
<td>19.75</td>
<td>16.49</td>
</tr>
</tbody>
</table>

Table 1: Performance degradation ranges.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Programs on the same chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimal</td>
<td>ammp art applu crafty mcf equake gap gzip</td>
</tr>
<tr>
<td>hierarchical</td>
<td>ammp art applu mcf gzip</td>
</tr>
<tr>
<td>perfect matching</td>
<td>equake gzip stream twolf</td>
</tr>
<tr>
<td>greedy</td>
<td>ammp art applu mcf gzip</td>
</tr>
</tbody>
</table>

Table 2: Schedule results from different algorithms.
grade the overall average performance by 19.81%. The hierarchical perfect matching algorithm reduces the degradation to 8.91%, whereas the greedy algorithm reduces it to 6.52%. The schedules produced by the two approximation algorithms have 5.08% and 2.40% more degradation than the optimal schedule.

The two approximation algorithms have similar effects on 5 programs, *art, bzip, facerec, parser*, and *vpr*. The greedy algorithm outperforms the hierarchical perfect matching algorithm on all the other programs except *ammp* and *swim*. On the program *stream*, the greedy algorithm outperforms the optimal schedule, which is not abnormal because our objective function is to minimize the overall performance. The better schedules assign jobs more balanced (as shown in Figure 7, discussed in Section 5.2), which is the key to achieving better overall performance.

Although the two approximation algorithms cut performance degradation of the random schedules by 55.0% and 67.1% respectively, they still have considerable distances from the optimal schedule. The local optimization brings them closer to the optimal.

5.1.2 Approximation with Local Optimization

Figure 6(b) presents the performance of the schedules generated by the two approximation algorithms with local optimization. Local optimization boosts the performance in both schedules. For the hierarchical perfect matching algorithm, the average degradation is reduced by 41.2%, from 8.92% to 5.21%; for the greedy algorithm, the reduction is 30.7%, from 6.52% to 4.51%. Their average performance degradations become only 1.4% and 0.7% away from the optimal, respectively.

A detailed analysis shows that the local optimization improves the performance of 7 programs, including the drastic
improvement on equake, gap, and mcf. For example, the degradation of mcf is reduced from 29% to less than 0.38% when the local optimization is applied to the two approximation algorithms. Meanwhile, the local optimization slightly worsens the performance of art, applu, bzip, facerc, and swim, but the negative effects are remarkably smaller than the enhancements. This result again shows the importance of balance in co-scheduling.

Overall, the greedy algorithm slightly outperforms the hierarchical perfect matching algorithm in terms of the reduction of the average performance degradation. But with local optimization, both approximation algorithms produce close-to-optimal results, reducing average corun degradation by over 74%.

5.1.3 Local Optimization Alone

Given that the local optimization enhances the approximation algorithms so much, we started wondering whether local optimization alone is enough for co-scheduling. So, we apply local optimization to 1000 random schedules. The results show that although sometimes the schedules are close to the optimal schedule, at many times, the produced schedules are much more inferior than the optimal schedules. The worst schedule result has up to 9.77% degradation. The average performance degradation is 6.27%, considerably larger than what we get from the greedy and hierarchical algorithms with local optimization.

5.2 Co-scheduling Fairness

Fairness is another important factor in measuring the quality of scheduling. Following the previous work [33], we measure the fairness of a schedule by an unfairness factor, defined as the coefficient of variation (standard deviation divided by the mean) of the normalized performance (\(\text{IPC}_{\text{co}}/\text{IPC}_{\text{su}}\)) of all jobs. A smaller unfairness factor means that the programs are subject to more similar influence from cache sharing; thus, the system is more fair.

Figure 7 shows that the optimal schedule has the best fairness, the random schedule has the worst, and the local optimization improves fairness by about 30%. The consistency between unfairness factor and overall performance degradation confirms the intuition that in order to reduce the overall performance degradation, we need to balance the degradation among different programs.

5.3 Co-scheduling Scalability

This section concentrates on the scalability of different approximation algorithms. We first summarize the time complexity of those algorithms, and then report the running times of those algorithms on scheduling problems of different sizes.

5.3.1 Time Complexity

We assume that there are \(N\) jobs, each chip has \(K\) cores, and there are \(\frac{N}{K}\) chips. The complexities of the approximation algorithms are as follows:

- Greedy Algorithm: The calculation of politeness and the construction of the final schedule both have time complexity of \(O\left(\binom{N}{K}\right)\). The total time complexity is \(O\left(\binom{N}{K}\right)\).
- Hierarchical Perfect Matching: The time complexity of perfect matching algorithms on a degradation graph with \(V\) vertices is \(O(V^2)\). The hierarchical algorithm conducts perfect matching \(\log K\) times. However, because the numbers of vertices in the degradation graphs decrease exponentially as the algorithm proceeds, the total time complexity is still \(O(N^4)\).
- Local Optimization: To get the optimal schedule of 2K jobs, we need to enumerate \(\binom{\frac{N}{K}}{2K}\) possible schedules and pick the best one. In local optimization, there are \(\binom{\frac{N}{K}}{2K}\) local reschedulings. The time complexity is \(O\left(\binom{\frac{N}{K}}{2K}\right)\).

The greedy algorithm has the same complexity as the hierarchical perfect matching algorithm when \(K\) is 4. However, as \(K\) increases, the overhead of the greedy algorithm increases much faster than the hierarchical method, which shows that the hierarchical method is more scalable. Given that \(N\) is typically much larger than \(K\), the overhead of local optimization is smaller than the approximation algorithms.

5.3.2 Running Times

We use 16 to 144 jobs to measure the running times of the two approximation algorithms with and without local optimization. The jobs are artificial jobs with random values as their corun degradations (the randomness has negligible influence on the running time measurement).

Figure 8 depicts the running times of the four algorithms when \(K\) is 4. The greedy algorithms consume more time than hierarchical methods do. The result is consistent with the time complexity analysis presented earlier in this section.

6. DISCUSSIONS

Besides providing theoretical insights into co-scheduling problems, this work can benefit the practice of co-scheduling in two ways. First, it removes the obstacles that prevent the evaluation of various co-scheduling systems. Most current evaluation of a co-scheduling system compares the system only to random schedulers. But in the design of a practical co-scheduling system, it is important to know the room left for improvement—that is, the distance from the optimal solution—to determine the efforts needed for further enhancement and the tradeoff between scheduling efficiency and quality. Through the techniques presented in this work, the optimal schedules can be either attained precisely or approximated accurately.
Architecture designs for alleviating cache contention have balanced thread scheduling for energy and performance [5]. DeVuyst et al. exploit un-estimated cache miss ratios [7, 33]. Kim et al. study cache partitioning for fairness in CMPs [16]. Recent studies employ the similar idea for CMP cache usage but use different program features, including estimated cache miss ratios [8, 9], and hardware performance counters [7, 33]. Kim et al. study cache partitioning for fairness in CMPs [16]. DeVuyst et al. exploit unbalanced thread scheduling for energy and performance [5]. Architecture designs for alleviating cache contention have focused on cache partitioning [13, 30], cache quota management [22], cache policies [12], and heterogeneous design [17].

8. CONCLUSIONS

This paper explores the problem of optimal job co-scheduling on CMPs. It proves that when more than 2 cores share a cache, the problem is NP-complete. The paper describes an efficient co-scheduling algorithm for dual-core systems by formulating the problem as a minimum weight perfect matching problem. Based on the algorithm, the paper furthermore proposes a polynomial-time hierarchical perfect matching algorithm that efficiently approximates the optimal schedules for general CMP systems. The algorithm outperforms a greedy algorithm in both efficiency and scheduling quality. With local optimizations, the hierarchical perfect matching algorithm produces near-optimal schedules for a set of 16 benchmarks on some quad-core AMD machines.

The theoretical results and approximation algorithms produced in this work offer the insights and practical support for the evaluation of co-scheduling systems. The algorithms can be directly used in proactive co-scheduling when co-run performance is predictable, and may serve as the base for enhancing reactive co-scheduling as well.

9. ACKNOWLEDGMENTS

We thank Weizhen Mao from William and Mary for her constructive comments to the draft of this paper. We thank Cliff Stein from Columbia University and William Cook from Georgia Tech for their helpful comments on perfect matching algorithms. We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IBM. Research of Rahul Tripathi was supported by the New Researcher Grant of the University of South Florida.
10. REFERENCES


