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

D.4.1 [**Operating Systems**]: Process Management

General Terms

Algorithms,Performance,Experimentation

Keywords

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

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

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 corun 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:

$$corun\ \ degradation\ \ of\ \ job\, i = \frac{cCPI_i - sCPI_i}{sCPI_i},$$

where $cCPI_i$ and $sCPI_i$ 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:

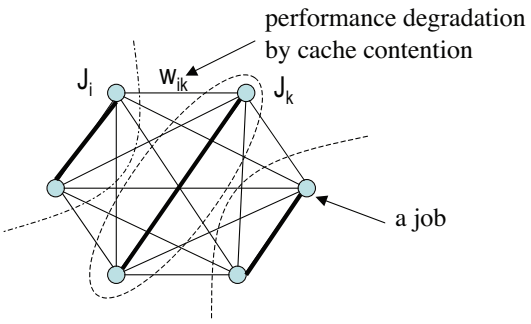


Figure 1: An example of a degradation graph for 6 jobs on 3 dual-cores. Each partition contains a job group sharing the same cache. Bold edges compose a perfect matching.

$$\min \sum_{i=1}^{N} \frac{cCPI_i - sCPI_i}{sCPI_i}. \quad (1)$$

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 combinational optimization is the polynomial-time *blossom* algorithm for finding minimum-weight perfect matchings proposed by Edmonds [6]. The time complexity of the algorithm is $O(n^2m)$, where n and m are respectively the numbers of nodes and edges in the graph. Later, Gabow and Tarjan develop an $O(nm+n^2logn)$ 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, *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 $i = 1, 2, \cdots, \binom{N}{K}$. ($G_i$ 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, $G_{p_1}, G_{p_2}, \cdots, G_{p_{\frac{N}{K}}}$ 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 $(N/K) * K = N$ 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,$\cdots$,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 $N = M * K$.
- 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 $C(a_1, a_2, \cdots, a_K)$, 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 enhance the scheduling 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 can 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 $logK$ 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 $logK$ times, its time complexity is the same as that of the basic minimum perfect matching algorithm, $O(N^4)$, 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

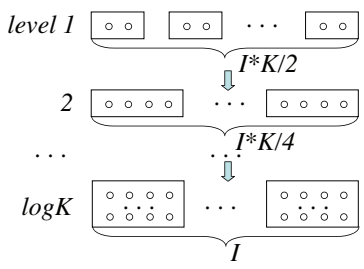


Figure 2: The hierarchical view of a CMP system used in the hierarchical perfect matching algorithm. Each box represents a virtual chip except the chips on the last level, which are real chips. Each circle represents a core; K is the number of cores per real chip.

```
/* N jobs; K cores per chip; L: cache size */
/* jobGroups contains the final schedule */
jobGroups ← {j1, j2, ··· , jN }
k ← 1
while (k < K) {
 cacheSize ← k * 2 * L/K;
 BuildGraph(jobGroups, cacheSize, V, E);
 /* conduct min-weight perfect matching and */
 /* store the matching pairs */
 R ← MinWeightPerfMatching(V, E);
 /* update jobGroups */
 reset jobGroups;
 i ← 0;
 for each node pair (vk, vl) in R {
  s ← vk.jobs ∪ vl.jobs;
  jobGroups[i++] ← s;
 }
 k ← k * 2;
}
/* Procedure to build a degradation graph */
BuildGraph(jobGroups, cacheSize, V, E) {
 reset V and E;
 for each element g in jobGroups; {
  node ← NewNode(g);
  V.insert(node);}
 for each pair of nodes (vi, vj) in V {
  s ← vi.jobs ∪ vj.jobs;
  w ← GetCorunDegradation(s, cacheSize);
  InsertEdgeWeight(E, vi, vj, w); }
}
```

Figure 3: Hierarchical minimum-weight perfect matching.

terms of the total degradation of the jobs in a set when they co-run together. Let S represent the final schedule, whose initial content is empty. We repeatedly pick the top set, none of whose members is covered by S yet, and put it into S until S covers all the jobs. This design is intuitive; every time, we greedily take the co-run group whose degradation is minimum. However, the result is surprisingly inferior—the produced schedules are among the worst possible schedules. We call this algorithm the naive greedy algorithm.

After revisiting the algorithm, we recognize the problem. Compared to other jobs, a job that uses little shared cache tends to be both "polite"—causing less degradation to its co-runners, and "robust"—suffering less from its co-runners. We call such a job a "friendly" job. Because of this property, the top sets are likely to contain only those "friendly"

```
/* J: job set; G: co-run groups */
/* S: schedule to compute */
CalPoliteness (J, G);
I ← politenessSort (J);
S ← ∅;
for i ← 1 to |J| {
 if job J[I[i]] not in S {
  s ← the group in G with the least degr. and
  containing J[I[i]] but not any jobs in S
  S ← S ∪ s;
 }
}
/* Procedure to compute politeness */
CalPoliteness (J, G){
 for i ← 1 to |J| {
  w ← 0;
  for each g in G that contains J[i] {
   w ← w + g.degradation;
  }
  J[i].politeness ← w;
 }
}
```

Figure 4: Greedy Algorithm

jobs. After picking the first several sets, the naive greedy algorithm runs out of friendly jobs, and has to pick those sets whose members are, unfortunately, all "unfriendly" jobs, which result in the large degradation in the final schedule.

We observe that if we 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 algorithm 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 $O(\binom{N}{K})$ time complexity, so the greedy algorithm's time complexity is $O(\binom{N}{K})$.

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 $(\frac{N}{K})^2/2$ iterations. Therefore, the time complexity for this local optimization is $O((\frac{N}{K})^2\binom{2K}{K})$.

```
/* S:  a given schedule */
LocalOpt (S) {
  M ← |S|;
  for i ← 1 to M − 1 {
    a1 = S[i];
    for j ← i + 1 to M {
      a2 ← S[j];
      (a'1, a'2)← Opt2Assignments(a1, a2);
      a1 = a'1;
      S[j] = a'2;
    }
    S[i] = a1;
  }
}
```

Figure 5: Local Optimization

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][2]. Including a stream program is to cover the special behavior of such programs. All the test programs are listed in Table 1.

In the collection of co-run performance, in order to avoid the distraction from the difference in program execution times, we follow Tuck and Tullsen's practice [32], wrapping each program to make it run 10 times consecutively, and only collecting the behavior of co-runs which are the runs overlapping with other programs.

There are totally $\binom{16}{4} = 1820$ co-runs. Table 1 shows the ranges of performance degradation of every program in its co-runs. Most of them have no degradation in their best co-runs, whereas, in the worst co-runs, all the programs show more than 50% slowdown. The mostly affected program is mcf, whose maximum slowdown is 191.49% with an average slowdown as much as 60%. The big ranges of degradations suggest the potential for co-scheduling.

The hierarchical perfect matching algorithm requires the co-run performance on smaller virtual chips. In this experiment, we collect such information by running 2 instances of 2 programs (totally 4 jobs) on a quad-core processor. The degradation is used as the estimation of that on a virtual dual-core chip. (Recall that each of the virtual chips has a half sized shared L3 cache.)

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.

[2]To focus on cache performance evaluation, we increased the size of a data element to the width of a cache line.

Table 1: Performance degradation ranges.

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

Table 2: Schedule results from different algorithms.

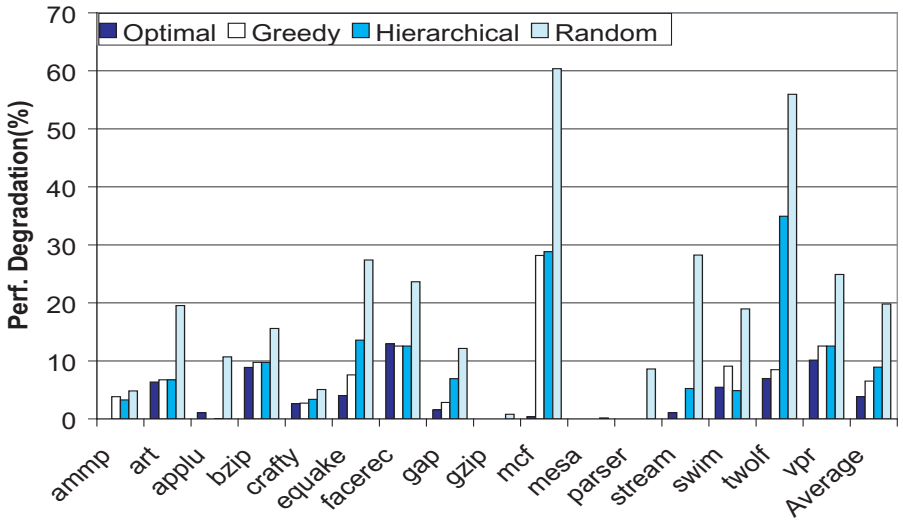
Algorithms	Programs on the same chip			
optimal	ammp	applu	crafty	equake
	art	parser	mcf	gap
	bzip	swim	mesa	gzip
	facerec	vpr	stream	twolf
hierarchical	ammp	art	applu	gzip
perfect	crafty	bzip	mesa	mcf
matching	equake	facerec	parser	stream
	gap	vpr	swim	twolf
greedy	ammp	art	applu	equake
	gzip	bzip	craft	gap
	mesa	facerec	mcf	parser
	stream	vpr	swim	twolf

We obtain the optimal schedule by conducting an exhaustive search[3]. The total number of possible schedules is $2,627,625$. 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.

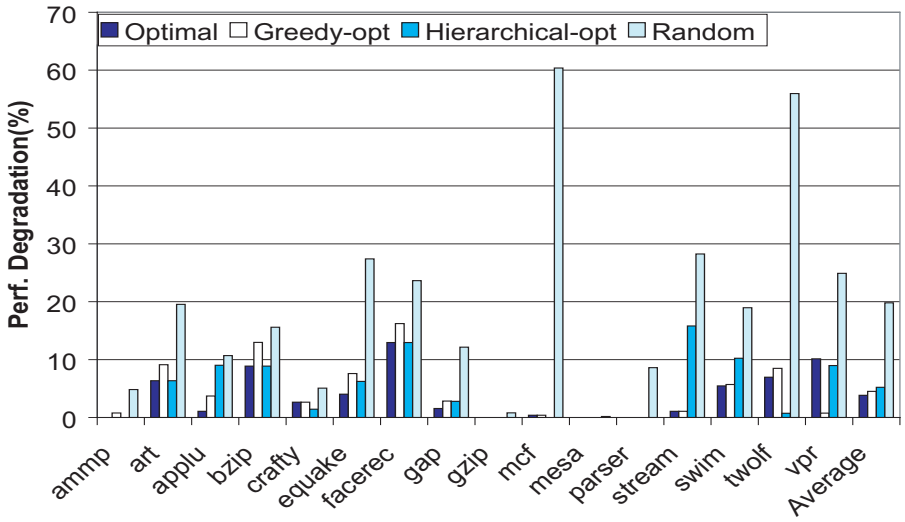
5.1.1 Approximation without Local Optimization

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 schedule and the schedules generated by two approximation algorithms. (Random schedules are not listed since we used 1000 of them.) The 4 programs in each table cell compose a corun group. Figure 6 (a) shows the corun degradation of each program in different schedules (some bars have 0 height and are thus invisible). The random schedules de-

[3]The optimal schedule is subject to the assumption that the performance of one assignment is independent on the schedule of other assignments. The assumption holds when cross-chip resource contention is negligible to program performance.



(a) Without local optimization.



(b) With local optimization.

Figure 6: Performance degradation under different schedules.

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

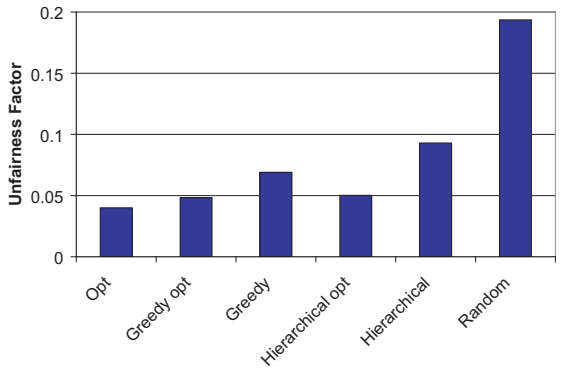


Figure 7: Unfairness factors of different schedules.

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, facerec,* 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 Coscheduling 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 **unfairness factor**, defined as the coefficient of variation (standard deviation divided by the mean) of the normalized performance (IPC_{co}/IPC_{si}) 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 Coscheduling Scalability

This section concentrates on the scalability of different approximation algorithms. We first summarize the time complexity of those algorithms, and then reports 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 $O(\binom{N}{K})$. The total time complexity is $O(\binom{N}{K})$.
- Hierarchical Perfect Matching: The time complexity of perfect matching algorithm on a degradation graph with V vertices is $O(V^4)$. The hierarchical algorithm conducts perfect matching $logK$ 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 $\binom{2K}{K}/2$ possible schedules and pick the best one. In local optimization, there are $(\frac{N}{K})^2/2$ local reschedulings. The time complexity is $O((\frac{N}{K})^2\binom{2K}{K})$.

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.

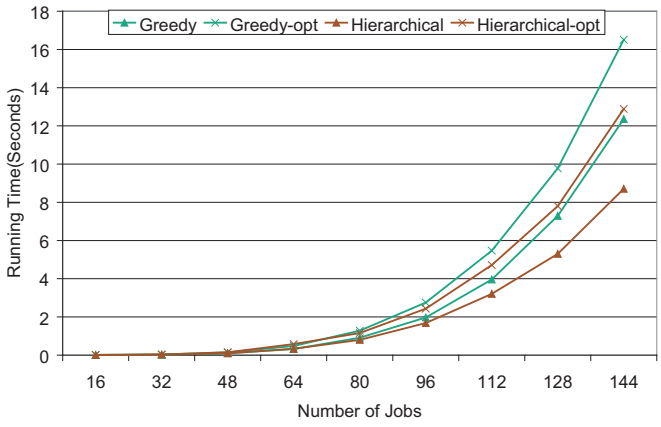


Figure 8: Scalability of different scheduling algorithms.

Second, proactive co-scheduling may directly benefit from the algorithms proposed in this work. There has been some work on predicting co-run performance from program single runs (e.g., [14, 16]), though the prediction techniques are not yet mature for practical uses. Some recently proposed techniques in locality analysis has advanced the efficiency and accuracy in locality characterization [24, 25, 34]. With accurate predictions, the proactive schedulers may benefit from this work by applying the co-scheduling algorithms to determine the optimal or near-optimal schedules.

In this paper, we assume that all co-run performance is given. This condition can be met in both uses of this work: Evaluations can typically afford the time for collecting all co-run performance, and proactive co-schedulers can efficiently predict co-run performance. We admit that obtaining co-run performance takes time, but we emphasize that the time for brute-forth search for optimal schedules will dominate for large-size problems, because the search time grows exponentially whereas the number of co-runs grows polynomially as the number of jobs increases.

Previous work has shown that program executions usually contain multiple phases of different performance [19, 26, 27]. In our discussion, we ignored phase changes and the difference between program execution times. If the purpose is to test a scheduler in job assignment, the user can simply use one phase of the programs and consider the co-run parts of their executions only. If the purpose is to test dynamic reassignment, the algorithms in this work can still serve as the component for finding the best schedule in each phase. The scheduler can re-schedule jobs at phase boundaries.

7. RELATED WORK

In 1968, Denning proposed balance-set scheduling to improve virtual memory usage by grouping programs based on their working set size [4]. 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].

Cache sharing also exists in Simultaneous Multithreading (SMT) processors. Parekh et al. introduce thread-sensitive scheduling. They demonstrate that by greedily selecting jobs with the highest IPC can improve system throughput significantly [21]. Snavely et al. propose symbiotic scheduling [28, 29]. It uses sample-optimization-symbios (SOS) scheme to try different combinations of jobs and pick the best one as the optimal schedule [28]. Their later work also considers process priorities in the symbiotic scheduling [29]. Settle et al. use an L2 cache activity vector to enhance the scheduler [23]. Some other work changes process affinity according to hardware performance counters [1, 20]. The algorithms presented in this paper may benefit SMT co-scheduling as well, as long as co-run performance is attainable.

To the best of our knowledge, this work is the first systematic study devoted to finding the optimal schedules for CMP systems. We are not aware any prior work on proving the NP-completeness of the problem and revealing the optimal algorithm for dual-cores. The approximation algorithms for more complex systems can benefit prior scheduling systems when co-run performance is predictable.

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

[1] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *2005 USENIX Annual Technical Conference*, pages 103–106, 2005.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[3] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[4] P. Denning. Thrashing: Its causes and prevention. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, volume 33, pages 915–922, 1968.

[5] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[6] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[7] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[8] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conference*, 2005.

[9] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[10] H. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph-matching problems. *Journal of ACM*, 38:815–853, 1991.

[11] M. Garey and D. Johnson. *Computers and Intractability*. Feeman, San Francisco, CA, 1979.

[12] L. R. Hsu, S. K. Reinhardt, R. Lyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[13] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of International Conference on Supercomputing*, pages 31–40, 2005.

[14] Y. Jiang and X. Shen. Exploration of the influence of program inputs on cmp co-scheduling. In *European Conference on Parallel Computing (Euro-Par)*, August 2008.

[15] R. Karp. Reducibility among combinatiorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[16] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2004.

[17] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[18] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 1995. http://www.cs.virginia.edu/stream.

[19] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2006.

[20] Nakijima and Pallipadi. Enhancements for hyperthreading technology in the operating system — seeking the optimal scheduling. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[21] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for smt processors. Technical Report 2000-04-02, University of Washington, June 2000.

[22] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[23] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced smt job scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 63–73, 2004.

[24] X. Shen and J. Shaw. Scalable implementation of efficient locality approximation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[25] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, 2007.

[26] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[28] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of ASPLOS*, 2000.

[29] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2002.

[30] H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), 1992.

[31] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2002.

[32] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.

[33] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[34] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*, 2008.