The Complexity of Optimal Job Co-Scheduling on Chip Multiprocessors and Heuristics-Based Solutions

Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, Rahul Tripathi

Abstract—In Chip Multiprocessors (CMP) architecture, it is common that multiple cores share some on-chip cache. The sharing may cause cache thrashing and contention among co-running jobs. Job co-scheduling is an approach to tackling the problem by assigning jobs to cores appropriately so that the contention and consequent performance degradations are minimized. Job co-scheduling includes two tasks: the estimation of co-run performance, and the determination of suitable co-schedules. Most existing studies in job co-scheduling have concentrated on the first task but relies on simple techniques (e.g., trying different schedules) for the second.
This paper presents a systematic exploration to the second task. The paper uncovers the computational complexity of the determination of optimal job co-schedules, proving its NP-completeness. It introduces a set of algorithms, based on graph theory and Integer/Linear Programming, for computing optimal co-schedules or their lower bounds in scenarios with or without job migrations. For complex cases, it empirically demonstrates the feasibility for approximating the optimal effectively by proposing several heuristics-based algorithms. These discoveries may facilitate the assessment of job co-schedulers by providing necessary baselines, as well as shed insights to the development of co-scheduling algorithms in practical systems.

Index Terms—Co-scheduling, Shared cache, CMP scheduling, Cache contention, Perfect matching, Integer programming

1 Introduction

In modern Chip Multiprocessors (CMP) architecture, it is common that multiple cores share certain levels of on-chip cache and off-chip bandwidth. As many studies have shown [3], [9], [10], [15], [26], [30], the sharing causes resource contention among co-running jobs, resulting in considerable and sometimes significant degradations to program performance and system fairness. Job co-scheduling is one of the approaches to addressing the contention problem. Its strategy is to assign jobs to computing units in a way that the overall influence from resource contention is minimized.

Job co-scheduling consists of two tasks. The first is to estimate the influence of cache sharing on the performance of a job when it co-runs with other jobs. The second is to determine the suitable co-schedules based on the estimation. Most existing studies [2], [3], [6], [10], [29] in job co-scheduling have concentrated on the first task but relies on simple techniques—for instance, trying a number of different co-schedules during runtime and choosing the best one—for the second. The use of these techniques, although having shown interesting results, has left a comprehensive understanding of the determination of optimal co-schedules yet to achieve. This lack impairs the assessment of a co-scheduler—it is hard to tell how far the co-scheduling results depart from the optimum and whether further improvement would enhance the co-scheduler significantly—and hinders the development of co-scheduling algorithms. Moreover, finding optimal co-schedules is critical for understanding how the various factors of CMP resource sharing affect program executions, as shown in a recent study [35].

This paper presents a systematic exploration for the second task. With its focus on optimal co-scheduling of independent jobs (i.e., jobs with no data sharing among one another), this work aims at answering three questions:

- How difficult is it to find optimal co-schedules?
- What algorithms can determine optimal co-schedules or reasonable lower bounds efficiently?

- *The first three authors are with the Department of Computer Science, The College of William and Mary, VA, USA 23185. E-mail: {jiang,ktian,xshen}@cs.wm.edu.*
- *J. Zhang is with the Computer Science Department, University of North Carolina at Chapel Hill. E-mail: jing2009@cs.unc.edu.*
- *J. Chen is with the Scientific Computing Group of the Thomas Jefferson National Accelerator Facility, VA, USA 23606. E-mail: chen@jlab.org.*
- *R. Tripathi is with the Department of Computer Science and Engineering, University of South Florida, Tampa, Fl, USA 33620. E-mail: tripathi@cse.usf.edu.*
- *Xipeng Shen and Jinghe Zhang are IEEE members.*

- *This article extends our earlier publications in the 2008 PACT [17] and 2009 ACM Computing Frontiers [31] conferences with three improvements. First, it presents the challenges and solutions of optimal job co-scheduling in a systematic way, unifying the proofs and algorithms published in the two previous papers into a single theoretical framework. Second, it introduces an Integer Programming formulation of the optimal co-scheduling problem and the use of the Linear Programming relaxed form for efficiently computing the co-scheduling lower bounds (Section 4.2). Third, it adds a set of new experimental results, including the empirical confirmation of the optimality of the polynomial-time optimal co-scheduling algorithm (Section 6.2.1), the validation of the IP/LP models in determining optimal schedules or lower bounds (Section 6.2.2), and the results of the co-scheduling algorithms on a new set of real jobs when job migrations are allowed (Section 6.3). Finally, it reveals some insights for the development of practical co-scheduling systems by examining the results in a holistic manner (Section 7).*

- When the optimal are too hard to find, can heuristics-based algorithms approximate them effectively?

Our exploration consists of three components. The first component (Sections 3 and 4) is focused on the complexity of co-scheduling in a basic setting where no job length variance or job migrations are considered. The discoveries fall in four aspects. The first is a polynomial-time algorithm for finding optimal co-schedules on dual-core CMPs. The algorithm constructs a degradation graph, models the optimal scheduling problem as a minimum-weight perfect matching problem, and solves it using the Blossom algorithm [8]. The second is a proof that optimal co-scheduling on u-core processors is an NP-complete problem when u is greater than 2, with or without job migrations allowed.[1] The third is an Integer Programming (IP) formulation of the optimal co-scheduling problem for u-core systems ($u > 2$). The formulation offers a clean way for formal analysis; its Linear Programming (LP) form offers an efficient approach to computing lower bounds for job co-scheduling. The final is a series of heuristics-based algorithms for approximating the optimal schedules in u-core CMP systems ($u > 2$). The first algorithm, named the hierarchical matching algorithm, generalizes the dual-core algorithm to partition jobs in a hierarchical way. The second algorithm, named the greedy algorithm, schedules jobs in 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.

The second component of this research (Section 5) expands the scope of the study with explorations on the complexities brought by job migrations that are incurred by job length variance. It shows the exponential increase of the search space and investigates the use of A*-search for accelerating the search for optimal schedules. For large problems, it offers two approximation algorithms, A*-cluster and local-matching algorithms, to effectively approximate optimal schedules with good accuracy and scalability.

The third component (Section 6) is a series of evaluations on the proposed co-scheduling algorithms. The results on 16 programs running on two kinds of commodity CMP machines validate the optimality of the optimal co-scheduling algorithms, and empirically demonstrate that the heuristics-based algorithms may achieve good estimation of the optimal co-schedules: Compared to sharing-oblivious scheduling, they reduce co-run degradation by 5–20% on average, 1.4% away from the optimum.

There has been a large body of research on optimal job scheduling. But to our surprise, despite an extensive survey [5], [22], we have found no previous work that tackles an optimal co-scheduling problem containing performance interplay among jobs as what the current co-scheduling problem involves. This work, although uncovering some interesting facts, is by no means to answer all questions on optimal job co-scheduling. Instead, it hopefully may serve as a trigger to stimulate further studies towards a comprehensive understanding to this intricate problem.

2 Definition of the Basic Co-Scheduling Problem

This section defines the basic co-scheduling problem, which concentrates on the primary challenges in assigning jobs to cores without the considerations of the complexities caused by job migrations. Section 5 will describe the treatment of those complexities.

This work concentrates on independent jobs—no jobs have data shared with each other. Co-running programs hence typically run slower than their single runs (i.e. the runs with no co-runners) due to resource contention. This kind of performance degradation is called co-run degradation. Formally, the *co-run degradation* of a job i when it co-runs with all the jobs in set S is defined as

$$d_{i,S} = (cCPI_{i,S} - sCPI_i)/sCPI_i,$$

where $cCPI_{i,S}$ and $sCPI_i$ are the average numbers of cycles per instruction (CPI) respectively when the job i co-runs with the job set S or when it runs alone [2]. ("c" for "co-run"; "s" for "single run".) The definition uses CPI because it is a commonly used metric for computing efficiency [14]. Immediately following the definition, $d_{i,S}$ must be non-negative, and $d_{i,S'} \leq d_{i,S}$ if $S' \subseteq S$.

The basic optimal co-scheduling problem is as follows:

> Given a set of n independent jobs, J_1, J_2, ..., J_n, and m identical chips with each equipped with u identical computing units that share certain on-chip resource uniformly, the goal is to find a schedule that maps each job to a computing unit so that the total co-run degradation, $\sum_{i=1}^{n} d_{i,S}$, is minimized, where, S is the set of jobs that are mapped to the chip that J_i is mapped to under the schedule.

We use the sum of co-run degradations as the goal function for the following reasons. A key object of co-scheduling is to maximize the computing efficiency of a CMP system, which suggests the use of the sum of CPIs of all jobs. However, the simple sum may cause an unfair schedule that favors high-CPI jobs to appear as effective. For instance, suppose two schedules for two jobs A and B produce (CPI_A=2, CPI_B=1) and (CPI_A'=1.4, CPI_B'=1.5) respectively. The second schedule appears to produce a smaller sum of CPIs than the first, but it degrades job B performance by 50% while improving job A performance by only 43%. Replacing the absolute CPI values with co-run degradations in the sum helps

1. For ease of explanation, the following description assumes a platform that contains multiple u-core single-threaded processors, with all cores on a chip sharing a cache.

2. Jobs are allowed to have different lengths. If a job finishes after its sharers do in a co-run, the $cCPI$ of the job is computed as the total cycles it takes to finish divided by its total number of instructions.

avoid the bias as degradation reflects the normalized computing efficiency.

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 schedules 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., [10], [21], [29]). 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).

This assumption does not prevent practical uses of the co-scheduling algorithms produced in this work. The first main use is to remove the obstacles for the evaluation of various co-scheduling algorithms. Most current evaluations of a co-scheduling system compares only to random schedulers. But in practical design of a co-scheduler, it is important to know the room for improvement—that is, the distance from the optimum—for determining the efforts needed for further enhancement and the tradeoff between scheduling efficiency and quality. That is exactly what the algorithms in this work provide or approximate. For such assessment, it is usually acceptable to collect the co-run performance of some jobs offline even if that takes some amount of time.

The second use of the algorithms is for proactive co-scheduling. Proactive co-scheduling decides the schedule of jobs before the jobs start running. They typically use predicted co-run performance of jobs [2], [3], [16]. The co-scheduling algorithms proposed in this work may help to determine the suitable schedules based on the predicted performance. We note that errors in performance prediction, although possibly hurting the quality of the resulting schedules, are tolerable to a certain degree in co-scheduling—even if the errors mislead a co-scheduling algorithm to consider an optimal schedule to be 10% (rather than 20% in truth) better (in terms of performance degradations) than other schedules, they do not prevent the algorithm from picking the optimal one.

In this basic co-scheduling problem, the co-schedule to be found is static, meaning that there are no job migrations during the execution of a job. (The influence on the performance of a job imposed by the assignments of jobs on other chips is typically small and neglected in job co-scheduling.)

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

3 Basic Optimal Co-Scheduling in Dual-Core Systems ($u = 2$)

In this section, we present an efficient algorithm for finding optimal schedules in a special case where the target systems have dual cores on each chip. It prepares for the explorations on more complex cases.

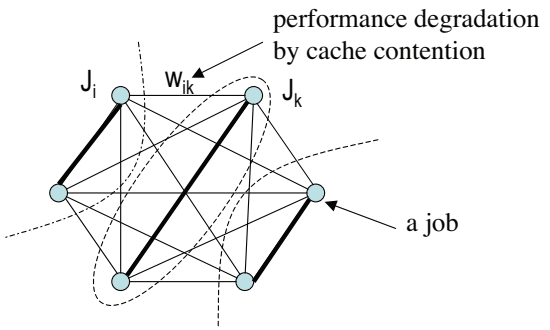


Fig. 1. An example of a degradation graph for 6 jobs on 3 dual-cores. Each partition contains a job group sharing the same cache. Bold edges compose a perfect matching.

We model optimal co-scheduling problems in this case as a graph problem. The graph is a fully connected graph, named *degradation graph*. As illustrated in Figure 1, every vertex in the graph represents a job, and the weight on each edge equals the sum of the degradations of the jobs represented by the two vertices when they run on the same chip. With this modeling, the optimal co-scheduling problem becomes 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.

It is easy to prove 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 the co-schedule defined in Section 2.

One of the fundamental discoveries in combinational optimization is the polynomial-time *blossom* algorithm for finding minimum-weight perfect matchings proposed by Edmonds [8]. It offers the polynomial-time solution to optimal co-scheduling on dual-cores. The time complexity of the algorithm is $O(n^2m)$, where n and m are respectively the numbers of nodes and the number of edges in the graph. Later, Gabow and Tarjan develop an $O(nm + n^2 \log n)$ algorithm [11]. Cook and Rohe provide an efficient implementation of the blossom algorithm [4], which is used in this work.

4 Basic Optimal Co-Scheduling in u-Core Systems ($u \geq 3$)

When $u \geq 3$, the optimal co-scheduling problem becomes substantially more complex than on dual-core systems. This section first analyzes the complexity of the problem, and then describes an IP/LP formulation of the problem for efficient lower-bound computation.

4.1 Proof of the NP-Completeness

This section proves that when $u \geq 3$, optimal co-scheduling becomes NP-complete. The proof is via a reduction of *Multidimensional Assignment Problem* (MAP) [12], a known NP-complete problem, 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_u represent the set of all u-cardinality subsets of S. Each of those u-cardinality subsets, represented by G_i, has a weight w_i, where $i = 1, 2, \cdots, \binom{n}{u}$. ($G_i$ 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/u such subsets, $G_{p_1}, G_{p_2}, \cdots, G_{p_{n/u}}$ to form a partition of S that satisfies the following conditions:

- $\bigcup_{i=1}^{n/u} G_{p_i} = S$. (Every job belongs to a subset.)
- $\sum_{i=1}^{n/u} w_{p_i}$ 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 $(n/u) * u = n$ jobs). The second condition ensures that the total weight of the subsets is minimum.

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

- Input: u ($u \geq 3$) sets Q_1,Q_2,$\cdots$,Q_u, each containing m elements, a cost function C: $Q_1 \times Q_2 \times \cdots \times Q_u \rightarrow R$, and a given value O.
- Output: An assignment A that consists of m subsets, each of which contains exactly one element of every set Q_k, $1 \leq k \leq u$. Every member $\alpha_i = (a_{i1}, a_{i2}, \ldots, a_{ik})$ of A has a cost $c_i = C(\alpha_i)$, where $1 \leq i \leq m$ and a_{ik} is the element chosen from the set Q_k.
- Constraints: Every element of Q_k, $1 \leq k \leq u$, belongs to exactly one subset of assignment A and $\sum_{i=1}^{m} c_i$ is equal to the given value O.

MAP has been proven to be NP-complete by reduction from the three-dimensional matching problem [12], a well-known problem first shown to be NP-complete by R. Karp [20].

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

- Let $S = \bigcup_{k=1}^{u} Q_k$ and $n = m * u$.
- Build all the u-cardinality subsets of S, represented as G_i, $1 \leq i \leq \binom{n}{u}$. If a subset G_i contains exactly one element from every set Q_k, $1 \leq k \leq u$, its weight is set as $C(a_1, a_2, \cdots, a_u)$, 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 u, the time complexity of the construction is $O(n^u)$. 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 $u \geq 3$.

4.2 Integer/Linear Programming for Optimal Co-Scheduling

The NP-completeness suggests that it is difficult if not impossible to generalize the algorithm described in Section 3 into a polynomial-time algorithm for the cases when u is greater than two. This section shows that optimal co-scheduling can be formulated as an IP problem in general, and therefore many standard IP solvers may be used to compute the optimal co-schedules directly. Furthermore, the LP relaxed form offers an efficient way to compute the lower bounds of the co-scheduling for an arbitrary u value.

4.2.1 *Integer Programming Model*

The IP formulation comes from the observation that optimal job co-scheduling defined in Section 2 is essentially a partition problem: To find a way to partition the n jobs into $m = \frac{n}{u}$ sets (corresponding to the m chips), with each job falling into one set and each set containing exactly u jobs, so that the total co-run degradation of all the jobs is minimized. We formulate it as the following IP problem.

The **variables** of the IP are:

x_{S_i}, where $1 \leq i \leq \binom{n}{u}$ and $S_i \subseteq \{1, 2, ..., n\}$ with $|S_i| = u$, and $S_i = S_j$ if and only if $i = j$ ($1 \leq i, j \leq \binom{n}{u}$.)
Each x_{S_i} is a binary variable, indicating whether the job set S_i is one of the sets in the final partition result.

The **objective function** is:

$$min \sum_{i=1}^{\binom{n}{u}} d(S_i) \cdot x_{S_i}$$

where, $d(S_i)$ is the sum of the co-run degradations of all the jobs contained in S_i when they co-run on a single chip, that is, $d(S_i) = \sum_{j \in S_i} d_{j, S_i - \{j\}}$.

The basic form of the **constraints** is:

$$x_{S_i} \in \{0,1\}, \quad 1 \le i \le \binom{n}{u};$$
$$\sum_{k:1\in S_k} x_{S_k} = 1; \sum_{k:2\in S_k} x_{S_k} = 1; \ldots; \sum_{k:n\in S_k} x_{S_k} = 1.$$

The first constraint says that x_{S_i} can only be either 0 or 1 (1 means that S_i is one of the sets in the final partition result; 0 means otherwise.) The first of the other n constraints means that there must be one and only one set in the final partition result that contains job J_1. The other constraints have the same meaning but on other jobs.

The basic form is intuitive but not amenable for efficient implementation. A refined form converts the last n constraints in the basic form into a matrix-vector multiplication form. In the form, A is an $n \times \binom{n}{u}$ matrix, with each element equaling either 0 or 1: The element of A at position (i, j) is 1 if and only if $i \in S_j$—that is, job J_i is in the job set denoted by S_j. Apparently, the matrix-vector equation is equivalent to the final n constraints in the basic IP form. We call the matrix A the *membership matrix* as it indicates which sets a job belongs to.

$$A \begin{pmatrix} x_{S_1} \\ x_{S_2} \\ \vdots \\ x_{S_{\binom{n}{u}}} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

4.2.2 Computing Lower Bounds in Polynomial Time

The IP problem is not polynomial-time solvable. But its lower bound can be efficiently computed through its LP form. The LP form is the same as the IP form except that the first constraint becomes

$$0 \le x_{S_i} \le 1, \qquad 1 \le i \le \binom{n}{u}.$$

It is easy to see that a feasible solution of the IP problem must be a feasible solution of the LP problem as well. The optimal value of the objective function in the LP, hence, must be no greater than the value in the IP. As LP problems can be solved efficiently, this relaxed form offers a fast way to compute lower bounds for optimal co-scheduling.

In our experiment, we employ the LP and IP solver in MATLAB to compute optimal co-schedules and the lower bounds. The LP solver, function *linprog*, is based on LIPSOL [34], which is a variant of Mehrotra's predictor-corrector algorithm [24], a primal-dual interior-point method. The IP solver, *bintprog*, uses a LP-based branch-and-bound algorithm to solve binary integer programming problems.

4.3 Heuristics-Based Approximation

Even though the IP model in the previous section formulates the optimal co-scheduling problem in a clean manner, solving the model may still be infeasible for a large problem given the NP-completeness of the job co-scheduling problem.

We design a set of heuristics-based algorithms to efficiently approximate the optimal schedules. The first algorithm is a hierarchical extension to the polynomial-time algorithm used when $u = 2$; the second is a greedy algorithm, which selects the local minimum in every step. In addition, we introduce a local optimization algorithm to enhance the scheduling results. We acknowledge that the theoretical accuracies of these algorithms are ideal to have, but yet to develop. Our discussion instead concentrates on the intuitions of their design and empirical evaluations.

4.3.1 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.

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 of size L as two virtual chips, with each containing a dual-core processor and a shared cache of size $L/2$. On the virtual dual-core system, we can apply the perfect matching algorithm to find the optimal schedule, in which, the job set is partitioned into $n/2$ pairs of jobs. Next, we create a new degradation graph, with each vertex representing one of the job 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 u-core co-scheduling problem by applying the minimum perfect matching algorithm $\log(u)$ 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 $\log(u)$ times, its time complexity is the same as that of the basic minimum perfect matching algorithm, $O(n^4)$, because the number of vertices in the degradation graphs decreases exponentially.

4.3.2 Greedy Algorithm

The second heuristics-based algorithm is a greedy algorithm. Our initial design is as follows. We first sort all of

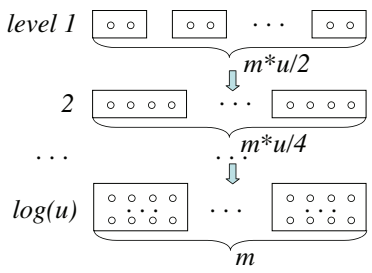


Fig. 2. The hierarchical view of a CMP system used in the hierarchical perfect matching algorithm. Each box represents a virtual chip except the chips on the last level, which are real chips. Each circle represents a core. (u: the number of cores per real chip; m: the total number of real chips in the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

```
/* n jobs; u cores per chip; L: cache size */
/* jobGroups contains the final schedule */
jobGroups ← {j1, j2, ··· , jn}
k ← 1
while (k < u) {
 cachePerVirtualChip ← k * 2 * L/u;
 BuildGraph(jobGroups, cachePerVirtualChip, V, E);
 /* compute 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, cachePerVirtualChip, 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, cachePerVirtualChip);
  InsertEdgeWeight(E, vi, vj, w); }}
```

Fig. 3. Hierarchical minimum-weight perfect matching.

the u-cardinality sets of jobs in the ascending order 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 from the sorted order, 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, the co-run group with minimum degradation is selected. 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.

```
/* 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 ← 1/w;}}
```

Fig. 4. Greedy Algorithm

Because of this property, the top sets in the sorted order are likely to contain only those "friendly" 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, causing 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.

This observation leads to the following improved algorithm. 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) It contains the job whose *politeness* is the smallest in the unassigned job list; 2) its total degradation is minimum. Figure 4 shows 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 $O(n\binom{n}{u})$ time complexity, so the greedy algorithm's time complexity is $O(n\binom{n}{u})$.

4.3.3 Local Optimization

Local optimization is a post-processing step for refining the schedules generated by both heuristics-based algorithms. For a given schedule, the algorithm optimizes each pair of assignments in the schedule. For each pair, the algorithm enumerates all possible ways to evenly partition the jobs contained in them into two parts. Each partition corresponds to one assignment for those jobs, and the one that minimizes the sum of co-run degradations of those jobs is taken as the final schedule for that pair. Figure 5 shows the pseudo-code.

```
/* 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];
    (a1', a2')← Opt2Assignments(a1, a2);
    a1 = a1';
    S[j] = a2';}
   S[i] = a1;}}
```

Fig. 5. Local Optimization

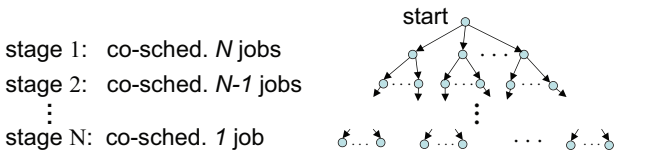


Fig. 6. The search tree of optimal job co-scheduling with rescheduling allowed at the end of a job. Each node in the tree, except the starting node, represents a sub-schedule of the remaining jobs. Each edge represents one schedule of the unfinished jobs.

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

5 Optimal Co-Scheduling with Migrations

With the understanding of the basic optimal co-scheduling problem, this section expands the scope of the problem to include job migrations into account. In this case, jobs may finish at different times, and rescheduling of the unfinished jobs may be necessary when some job terminates and vacates a core. We call each scheduling or rescheduling *a scheduling stage*. This work concentrates on the settings where rescheduling happens only when a job finishes; there are at most n scheduling stages for n jobs.

Some terminology needs to be redefined in this setting. *An assignment* still refers to a group of K jobs that are to run on the same chip. We use *a sub-schedule* to refer to a set of assignments that cover all the unfinished jobs and have no overlap with one another. *A schedule* still refers to a solution to the co-scheduling problem. However, in this new setting, a schedule becomes a set of sub-schedules that have been used from the start of the jobs to the finish of the final job.

Considering job length variances, we redefine the goal of the co-scheduling as to find a schedule that minimizes the total execution time of all jobs [3], expressed as

$$\arg\min_{S} \sum_{i=1}^{n} {cT_i}^{(S)},$$

where, ${cT_i}^{(S)}$ is the time job i takes to finish in a co-schedule S. Other settings of the problem remain the same as those described in Section 2.

Next, we first examine the increased co-schedule space of the extended problem, and then present the use of A*-search–based approaches for pruning the space to help find or estimate optimal co-schedules efficiently.

3. It is assumed that the clock starts at time 0 for all jobs no matter whether they are actually running.

5.1 Co-Schedule Space

We model the optimal co-scheduling problem as a tree-search problem as shown in Figure 6. For n jobs, there are at most n scheduling stages; each corresponds to a time point when one job finishes since the previous stage. Every node in the tree, except the starting node, represents a sub-schedule of the remaining jobs. The nodes at a stage, say stage i, correspond to all possible sub-schedules for $n-i+1$ remaining jobs. There is a cost associated with each edge, equal to the total execution time spent by all jobs between the two stages connected by the edge. Let $node_2$ represent a child of the node $node_1$. Given the state at $node_1$, we assign the unfinished jobs to cores according to the sub-schedule contained in $node_2$; let t be the time required for the first remaining job to finish; the cost on the edge from $node_1$ to $node_2$ is $t * m$, where m is the number of jobs that are alive during that period of time.

The goal of the optimal co-scheduling is to find a path from the starting node to any leaf node (called a goal node) so that the sum of the costs of all the edges on the path is minimum. The search space in this extended problem involves $O(n^n)$ nodes. In contrast, the scheduling space in the basic problem tackled in previous sections contains only the starting node and the first stage in the tree (without rescheduling); the total number of nodes is exponentially smaller than that in this extended problem.

5.2 Finding the Optimal through A*-Search and Linear Programming

To address the increased complexity, we investigate the use of A*-search, along with a linear programming model for cost estimation.

5.2.1 A*-Search Algorithm

A*-search is an algorithm stemming from artificial intelligence [27] for fast graph search. It has been used for many search problems, but not for job co-scheduling. This section presents the basic algorithm of A*-search, and the next section will describe the special challenges in applying A*-search to job co-scheduling.

A*-search is appealing in several aspects. It guarantees the optimality of its search results, and meanwhile,

effectively avoids visiting certain portion of the search space that contain no optimal solutions. In fact, it has been proved that A*-search is optimally efficient for any given heuristic function. That is, for a given heuristic function, no other optimal algorithm is guaranteed to expand fewer nodes than A*-search [27]. Its completeness, optimality, and optimal efficiency trigger our exploration of using it for job co-scheduling.

We use Figure 6 to explain the basic algorithm of A*-search. In the graph, there is a cost associated with every edge. The objective is to find the cheapest route in terms of the total cost from the starting node to a goal node. In A*-search, each node, say node d, has two functions, denoted as $g(d)$ and $h(d)$. Function $g(d)$ is the cost to reach node d from the starting node. Function $h(d)$ is the estimated cost of the cheapest path from d to a goal node. So, the sum of $g(d)$ and $h(d)$, denoted as $f(d)$, is the estimated cost of the cheapest route that goes from the start to the goal and passes through node d.

Often, the graph to be searched through is conceptual and does not exist at the beginning of the search. During the search process, the A*-search algorithm incrementally creates the portion of the graph that possibly contains optimal paths. Specifically, A*-search uses a priority list to decide the next node to expand (i.e., to create its children nodes). Initially, the priority list contains only the starting node. Each time, A*-search removes the node with the highest priority from the top of the priority list and expands that node. After an expansion, it computes the $f(d)$ values of all the newly generated nodes, and inserts them into the priority list according to their priority values, which are computed as $1/f(d)$. Such expansions continue until the top of the priority list is a goal node, a sign indicating that an optimal path has been found. The algorithm terminates. The use of the $f(d)$-based priority list is the key for A*-search to avoid unnecessary expansions without sacrificing the optimality of the search result.

Recall that $f(d)$ is the sum of $g(d)$ and $h(d)$. The function $g(d)$ is trivial to define—just the cost from the starting node to node d. The definition of $h(d)$ is problem-specific and critical. The following two properties of A*-search reflect the importance of $h(d)$:

- The result of A*-search is optimal if $h(d)$ is an admissible heuristic—that is, $h(d)$ must never overestimate the cost to reach the goal[4].
- The closer $h(d)$ is from the real lowest cost, the more effective A*-search is in pruning the search space.

Determining a good definition of $h(d)$ is the core of applying A*-search.

5.2.2 A*-Search–Based Job Co-Scheduling

Using A*-search for job co-scheduling is simply to apply the search algorithm in the co-scheduling space. The main complexity exists in the definition of the function $h(d)$.

Recall that $h(d)$ is the estimated cost of the cheapest path from the node d to a goal node. When all co-run degradations are non-negative, a simple definition is the sum of the single-run times of all the unfinished parts of the remaining jobs. This definition is legal— $h(d)$ does not exceed the actual costs—but may lead to large departure between the values of $h(d)$ and the actual costs because it does not consider co-run degradations.

In this work, we resort to Linear Programming for defining $h(d)$. Suppose at node d there are U unfinished jobs. We define $h(d) = T_s + T_{deg}$, where T_s is the time the U jobs need to finish their remaining parts if they each run alone, and T_{deg} is the estimated minimum of the total degradation of the U jobs during their execution from the node d to any child of d.

We concentrate on the common case when all degradation rates are non-negative. In this case, when U is not greater than the number of chips I, T_{deg} is clearly 0 as there is at most one job on each chip. Our following discussion is focused on the scenario where $U > I$.

Consider a sub-schedule represented by one of the children nodes of n. The total degradation of all U jobs in the sub-schedule equals the sum of the degradations on all chips. The minimum degradation on one chip with b jobs can be estimated as follows. Let $T_{min(d)}$ represent the minimum of the single-run times of the unfinished part of all the remaining U jobs. Notice that the time lasting from d to any of its children must be no less than $T_{min(d)}$ because of co-run degradations. Let $r_{b_{min}}$ be the minimum of the degradation rates of all jobs when a job co-runs with $b-1$ other jobs. It is clear that the degradation on the chip must be no less than $b * r_{b_{min}} * T_{min(d)}$, which is taken as the estimation of the minimum degradation of the chip. Therefore, the lower bound of the degradation of a sub-schedule j is $d_j = \sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(d)}$, where, b_i is the number of jobs assigned to chip i in the sub-schedule.

The value of T_{deg} should be the minimum of d_j of all sub-schedules of the node d. To determine the sub-schedule that has the smallest d_j, we need to find the values of b_i so that $\sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(d)}$ is minimized under the constraint $\sum b_i = U$. This analysis leads to an Integer Linear Programming problem shown in Figure 7. By relaxing the constraint on x_i to $0 \leq x_i \leq 1$, the problem becomes a Linear Programming problem, which can be solved efficiently using existing tools [1].

As a special case, when $K = 2$, the solution to the Integer Linear Programming is equivalent to the following simple formula:

$$T_{deg} = 2 * (U - I) * r_{2_{min}} * T_{min}(d). \quad (1)$$

The intuition for the formula is that in any sub-schedule of this scenario, there must be at least $(U - I)$ chips that have a pair of the unfinished jobs assigned. Otherwise, some chips must have more than two jobs

4. We assume that the search is a tree search. There are some subtle complexities for other types of search [27].

Definitions:
U: number of unfinished jobs,
I: number of chips,
K: cores per chip, $I < U \leq I * K$,
$r_{x_{min}}$: minimum degradation rate,
$T_{min(d)}$: minimum single run time.

$$x_i = \begin{cases} 1 & : \textit{the ith core has a job assigned} \\ 0 & : \textit{otherwise} \end{cases}$$

The number of jobs on the c-th chip:

$$m(c) = \sum_{i=(c-1)*K+1}^{c*K} x_i$$

Objective function:

$$min \sum_{c=1}^{I} m(c) * r_{m(c)_{min}} * T_{min(d)}.$$

Linear constraint:

$$\sum_{i=1}^{n} x_i = U;$$

Fig. 7. Integer Linear Programming for computing T_{deg}, the lower bound of degradation.

assigned, which is not allowed in the problem setting (Section 2). The application of the definition of T_{deg} to such a sub-schedule leads to Equation 1.

5.3 Heuristics-Based Estimation

A limitation of A*-search–based algorithms is its high requirement for memory space: It keeps all open nodes (i.e. the nodes that have unexpanded children nodes) in the priority list, while the number of open nodes grows in exponential to the problem size in job co-scheduling.

In this section, we describe two heuristics-based algorithms for solving the optimal co-scheduling problem in a scalable manner. One algorithm, the A*-cluster algorithm, integrates clustering into the A*-search-based algorithm; the other algorithm, the local-matching algorithm, is a generalized version of the graph-matching-based co-scheduling algorithms mentioned in Section 4.3.1.

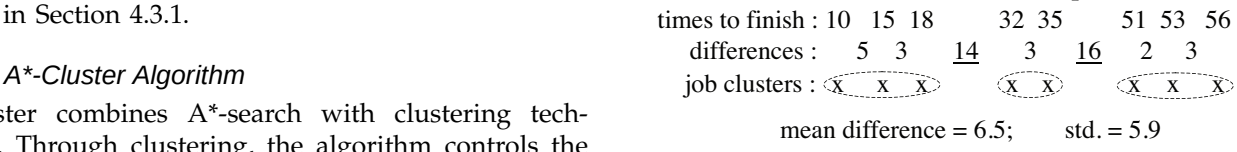
5.3.1 A-Cluster Algorithm*

A*-cluster combines A*-search with clustering techniques. Through clustering, the algorithm controls the number of rescheduling stages by rescheduling only when a cluster of jobs finish. Also through clustering, the algorithm avoids the generation of sub-schedules that are similar to one another. Together the two features reduce the time complexity of the problem significantly.

An option for job clustering is to group them based on their single-run times. However, jobs with similar single-run times may need very different times to finish in co-run scenarios. Our solution is an online adaptive strategy. At the beginning, jobs are clustered based on their single-run times. During the expansion of the search tree, at each node, the algorithm computes the state of the job set when the first *cluster* of the unfinished jobs complete under the current sub-schedule (**to reduce the number of scheduling stages**), and then regroups the other jobs into certain clusters based on the time needed for each of them to finish under the current sub-schedule. Based on the clustering results, during the generation of children nodes, the algorithm selects the sub-schedules that are substantially different from the already generated sub-schedules (**to reduce the number of nodes at a stage**). A sub-schedule is substantially different from another if they are not equivalent when we consider all jobs in a cluster equivalent. For example, four jobs fall into two clusters as {{1 2}, {3 4}}. The sub-schedule (1 3) (2 4) is considered equivalent to (1 4) (2 3), but different from (1 2) (3 4) (each parenthesis pair contain a co-run group). Finding those novel sub-schedules only needs to solve a first-order linear equation system, in which, each unknown is the number of the instances of a cluster mixture pattern[5] included in a sub-schedule. Each equation corresponds to one job cluster: On the left side is the sum of the number of the jobs falling into that cluster in a sub-schedule, and on the right side is the total number of jobs belonging to that cluster. Every solution to the equation system corresponds to a novel sub-schedule.

The first strategy reduces the height of the search tree, while the second reduces the width. Together, they reduce the number of nodes at a stage significantly.

Although there are many clustering methods (e.g., K-means, hierarchical clustering [13]), we use a simple distance-based clustering approach because the data to be clustered—the job lengths—are one-dimensional and the number of clusters is unknown beforehand. Given a sequence of data, the distance-based clustering first sorts the data in ascending order. It then computes the differences between every two adjacent data items in the sorted sequence. Large differences indicate cluster boundaries. A difference is considered large enough if its value is greater than $m + \delta$, where, m is the mean value of the differences in the sequence and δ is the standard deviation of the differences. An example is as follows.



The time complexity of the clustering algorithm is $O(J)$, where J is the number of remaining jobs.

5.3.2 Local-Matching Algorithm

For even higher efficiency, we design a second approximation algorithm, which explores only one path from the root to the goal in Figure 6. At each scheduling point, it selects the schedule that minimizes the total running time of the remaining part of the unfinished jobs under the assumption that no reschedules would happen. The

5. An example of cluster mixture patterns for quad-core chips is an assignment that contains one job from cluster 1, two jobs from cluster 2, and one job from cluster 3.

assumption leads to local optimum at each scheduling stage.

The key component of the algorithm is the procedure to compute the local optimum. This step is the same as the basic job co-scheduling problem discussed in Section 2, except that the number of jobs may be smaller than the number of cores as some jobs may have terminated. We take a simple strategy to handle this case: treating the jobs that have finished as pseudo-jobs, which exist but consume no computing resource. Therefore, if the co-runners of a job are all pseudo-jobs, that job has no performance degradation at all. As the pseudo-jobs have to be scheduled every time, this strategy introduces some redundant computation. However, it provides an easy way to generalize the perfect matching algorithm described in Section 3 and 4. Apparaently, the time complexity of the local-matching algorithm is $O(n^5)$: The co-scheduling algorithm on a stage has complexity of $O(n^4)$, and there are n stages.

6 Evaluation

In this section, we concentrate on the verification of the optimality of the results produced by the optimal co-scheduling algorithms, the departure of the results by the heuristics-based algorithms from the optimal, along with the efficiency and scalability of those algorithms.

6.1 Methodology

The machines we use include both dual-core and u-core ($u > 2$) systems. For dual-core cases, we use a quad-core Dell PowerEdge 1850 server, which although named quad-core, includes two Intel Xeon 5150 2.66 GHz dual-core processors, each having a 4MB shared L2 cache. Every core has a 32KB dedicated L1 data cache. For the cases of $u \geq 3$, we use machines each 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.

Table 1 lists the 16 programs used in the experiments, along with the ranges of their performance degradations when they co-run on the AMD machine. The programs are chosen to cover both integer and floating-point benchmarks and span a wide range of the application areas. Their executions exhibit various patterns in memory and cache accesses—from having few data reuses (e.g., gzip) to having many (e.g., swim). All programs come from SPEC CPU2000 except *stream* coming from a streaming benchmark [23][6]. 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 large degradation ranges suggest the potential for co-scheduling. In addition, we employ some synthetic problems for large coverage and the test of extreme scenarios. In those problems, the job lengths and co-run degradation rates are some random values.

6. 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 on AMD Opteron without job migrations

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

In the collection of co-run degradations, 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. 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 for some algorithms applied to quad-core machines.

6.2 Basic Optimal Co-Scheduling

In this section, we examine the capability of the perfect matching-based algorithm for finding optimal co-schedules in dual-core systems, the lower bounds computed by the Linear Programming model for u-core ($u > 2$) systems, and the quality of the co-schedules produced by the heuristics-based algorithms.

6.2.1 Optimal Co-Scheduling by Perfect Matching

On the Intel machine, we conduct an exhaustive search for the best schedule among all possible ones; the resulting schedule is the same as the schedule found by the minimum-weight perfect matching algorithm, confirming the optimality of the scheduling results. (Twelve of the 16 programs are used because of the high cost of the exhaustive search.)

Figure 8 shows the comparison among 3 different scheduling results. We use *optimal* to represent the schedule found by the minimum-weight perfect matching algorithm. The *random* bars show the average scheduling results produced by 1000 random schedules, corresponding to most current CMP scheduling systems, which are oblivious to shared cache. The *worst* bars are the results from the worst among all schedules, demonstrating the possible consequence of careless scheduling. The co-run

groups in the optimal co-schedule are {*ammp+parser, art+crafty, bzip+gap, equake+mesa, gzip+mcf, twolf+vpr*}.

The results show that the optimal schedule may reduce performance degradations significantly, from over 15% of random scheduling to 7% on average. For some programs, the cut is up to a factor of 5. The performance results match with the L2 miss rates shown in the bottom graph, although not proportionally due to the different sensitivity of the programs to L2 miss rates. On average, the optimal schedule reduces 20% L2 cache miss rates relative to the random schedule and 28% relative to the worst schedule.

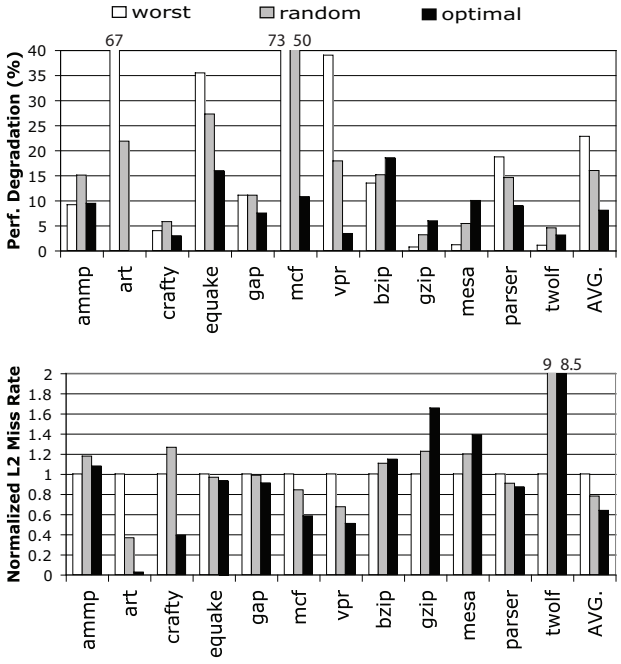


Fig. 8. Performance degradations (top graph) and L2 cache miss rates (bottom graph) in different co-schedules in Intel Xeon 5150 (no migrations).

It is worth noting that random scheduling may group some programs in the way the worst scheduling does; the consequence is severe: 67% degradation for *art*, 73% for *mcf*, and 22.8% on average. The optimal co-scheduling avoids those traps, making co-runs significantly faster than the worst schedule on average. (Note that our goal is to minimize the *overall* rather than each individual program's degradation. So, it is normal for certain programs to run worse in the optimal schedule than in other schedules.)

6.2.2 Lower Bounds by Linear Programming

This section reports the results for validating the optimality of the solution produced by the IP model, and assessing the lower bounds by the LP relaxation. We use a sequence of synthetic problems (u is 4) to cover various cases. Table 2 reports the degradations of the resulting co-schedules, along with the time the scheduling algorithms take. The co-schedules produced by the IP algorithm always have the same degradations as the co-schedules found by the brute-force search. The IP algorithm takes much less times than the brute-force search does. The LP algorithm exhibits even better appeal: The degradations from it show minor difference (less than 10%) from the optimal, but can be obtained in less than 1% of the IP time for large problems.

TABLE 2
Co-Run degradations and scheduling times on synthetic problems, with three instances for each problem size (no migrations).

Num of Jobs	average degradation		scheduling time (s)		
	brute-force/IP	LP	brute-force	IP	LP
8	0.35	0.32	0.01	0.09	0.03
8	0.29	0.29	0.01	0.04	0.05
8	0.26	0.26	0.01	0.04	0.03
12	0.28	0.27	0.31	2.07	0.05
12	0.28	0.27	0.84	1.28	0.06
12	0.27	0.26	0.56	2.06	0.05
16	0.26	0.26	14.07	12.11	0.16
16	0.26	0.26	11.77	8.25	0.15
16	0.26	0.25	11.72	16.48	0.12
20	0.26	0.25	13095	82.6	0.41
20	0.25	0.25	12728	48.82	0.4
20	0.25	0.25	12768	33.37	0.4

More experiments show that the LP model can be solved in less than 200 seconds for problems with less than 80 jobs, exhibiting good scalability. In the next section, the LP model shows the usefulness in the assessment of the quality of the scheduling results produced by heuristics-based algorithms.

6.2.3 Estimation by Heuristics-Based Algorithms

This section examines the quality of the co-schedules produced by the heuristics-based algorithms and their scalability. We compare four types of schedules: the optimal, the random, the hierarchical perfect matching, and the greedy schedules. The metric we use is the average performance degradation of all programs.

We obtain the optimal schedule by solving the corresponding IP model; the result matches with the exhaustive search result. The total number of possible schedules is $2,627,625$. We obtain the random scheduling result by applying 1000 random schedules to the jobs and getting the average performance.

Figure 9 shows the co-run degradations of each program in different schedules (some bars have 0 height and are thus invisible). The random schedules degrade the overall average performance by 19.81%. The hierarchical perfect matching algorithm reduces the degradation to 5.21%, whereas the greedy algorithm reduces it to 4.51%. The schedules produced by the two approximation algorithms have 1.4% and 0.7% more degradations than the optimal schedule on average.

We use a set of synthetic problems to evaluate the quality of the heuristics-based algorithms more comprehensively. Given that the greedy algorithm shows the better performance than other approximation algorithms, we concentrate on this algorithm. We use LP

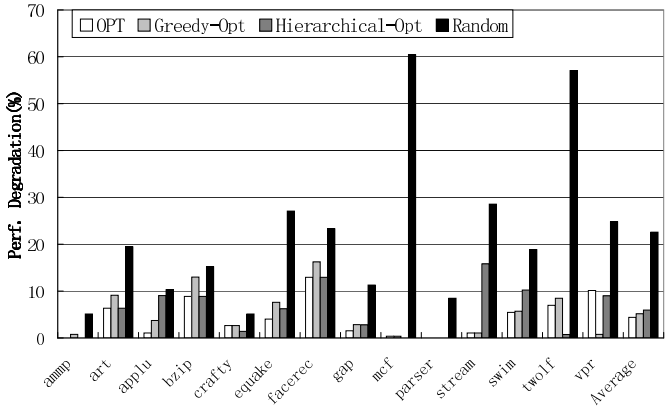


Fig. 9. Performance degradations in different co-schedules in AMD Opteron (no migrations).

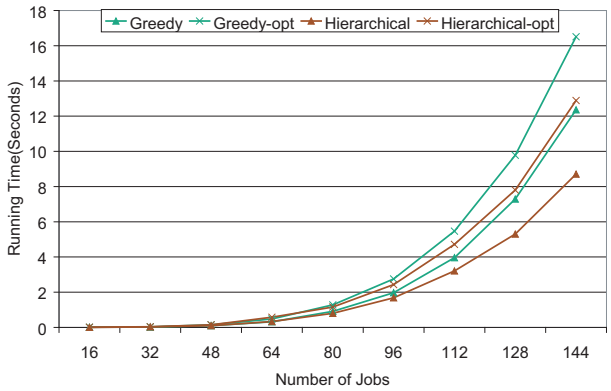


Fig. 10. Scalability of different scheduling algorithms (no migrations).

TABLE 3
Assessment of the greedy algorithm by comparing with the random scheduling results and the lower bound from the LP algorithm (no migrations).

Num of Jobs	16	32	48	64	80
Deg of random sch.	0.62	0.63	0.62	0.63	0.63
Deg of LP sch.	0.26	0.25	0.25	0.25	0.25
Deg of Greedy sch.	0.29	0.27	0.26	0.25	0.26
Reduct. over random (%)	117.5	128.2	139.9	145.9	142.3
Distance from LP (%)	10.9	8.5	3.9	1.6	3.1

model to compute the lower bounds. Table 3 shows the evaluation results. The co-schedules produced by the Greedy algorithm exhibit less than 11% distance from the lower bound, indicating the high quality of the co-scheduler.

Co-Scheduling Scalability

As mentioned in previous sections, the time complexities of the heuristics-based algorithms are as follows: $O(nx\binom{n}{u}) + (\frac{n}{u})^2\binom{2u}{u})$ for the greedy algorithm, and $O(n^4) + (\frac{n}{u})^2\binom{2u}{u})$ for the hierarchical perfect matching (the $(\frac{n}{u})^2\binom{2u}{u})$ part is for the local optimization step), where, n for job numbers, u for the number of cores per chip, and $\frac{n}{u}$ for the number of chips.

The greedy algorithm has the same complexity as the hierarchical perfect matching algorithm when u is 4. However, as u 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 u, the overhead of local optimization is often a small portion of the total time.

We use synthetic problems including 16 to 144 jobs to measure the running times of the two approximation algorithms with and without local optimization. Figure 10 depicts the running times of the four algorithms when u 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.

TABLE 4
Comparison of co-scheduling algorithms on 8 jobs when job migrations are considered

algorithm	visited nodes	scheduling time (s)	total exec time (s)	deg. rate (%)
brute-force	16 M	470	80.3	1.3
A*	7760	0.3	80.3	1.3
random	-	-	85.9–89.2	8.4–12.5

6.3 Optimal Co-Scheduling with Migrations

This section evaluates the use of A*-search and the heuristics-based algorithms for co-scheduling jobs when job migrations are allowed.

6.3.1 *Optimal Co-Scheduling by A*-Search*

We generate 100 co-scheduling problems, and apply brute-force search, A*-search, and random scheduling to them. Because of the rapidly increasing search time in the brute-force algorithm, we limit the number of jobs in each problem to eight. Table 4 reports the average result. The A*-search–based co-scheduler produces the same schedules as the brute-force search does, confirming the optimality of the A*-search results. On the other hand, the A*-search–based scheduler finds the optimal schedules by visiting only 0.05% of the nodes that brute-force search visits. It cuts the search time from 470 seconds to 0.3 seconds. The significant reduction demonstrates its effectiveness in space pruning. Compared to random schedules, the optimal schedules reduce performance degradation rates from up to 12.5% to 1.3%.

6.3.2 *Estimation by Heuristics-Based Algorithms*

For 16 jobs, the brute-force algorithm would take years. Our implementation of the A*-search algorithm (in Java) is subject to memory shortage when scheduling more than 12 jobs. (A memory-bounded version [27] may help.) In this section, we concentrate on the evaluation of the two heuristics-based approximation algorithms.

Figure 11 shows the degradation rates of 16 real jobs on Intel Xeon 5150 processors. The random schedules

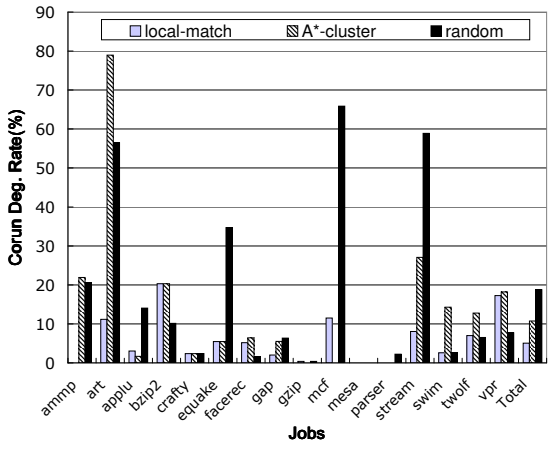


Fig. 11. Performance degradation rates of 16 jobs co-running on Intel Xeon 5150 processors.

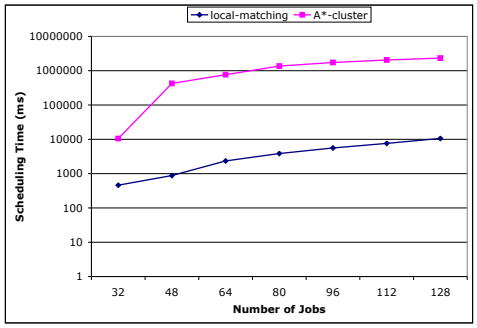


Fig. 12. Scalability of A*-cluster and local-matching algorithms (with job migrations).

cause 18% (up to 66%) degradation to the total running time. The A*-cluster algorithm reduces the degradation to 10.2% by visiting 721 nodes in the search tree, while the local-matching algorithm cuts the degradation further to 5% by visiting only 8 nodes in the search tree. The inaccuracy introduced by the clustering process in the A*-cluster algorithm appears to have considerable effects on the scheduling results. The times for the two algorithms to finish scheduling are 109 and 0.63 seconds respectively. This result indicates that despite the sophistication of the A*-cluster algorithm, it is not as attractive as the local-matching algorithm in terms of both efficiency and scheduling quality.

We use 32 to 128 synthetic jobs to further measure the running times of the two approximation algorithms ($K = 2$). As shown in Figure 12, the local-matching algorithm exhibits better scalability than the A*-cluster algorithm does: It takes only about 10 seconds to schedule 128 jobs, whereas, while the A*-cluster algorithm needs more than 2000 seconds.

7 Insights for the Development of Practical Co-Scheduling Systems

The algorithms proposed in this work have two main uses. The first is to help determine the potential for co-scheduling a set of jobs and to facilitate the assessment of practical co-scheduling systems, as exemplified by some recent work [35]. The second is to inspire the development of co-scheduling mechanisms that are ready to be deployed in realistic settings. This section presents some lessons and insights for the second use.

Our first observation is that simple algorithms are capable of producing close-to-optimal results, as shown by the comparison between the simple greedy algorithm and the sophisticated hierarchical perfect matching algorithm (Section 6.2.3), and the comparison between the simple local matching algorithm and the A* algorithms (Section 6.3.2).

Second, in the design of greedy algorithms, it is important to distinguish "friendly" jobs from "unfriendly" ones, and couple them together (Section 4.3.2) in the produced schedule.

Third, large potential (e.g., 73% for *mcf*) exists for using co-scheduling to improve the performance of some applications running on CMP systems. Co-scheduling for those applications is critical. On the other hand, some applications are less sensitive to co-scheduling than others. A mixture of them often means opportunities for effective co-scheduling results.

Finally, the local optimization is a cheap but effective way to refine co-scheduling results. The results in Section 6.2.3 are obtained after local optimizations cut degradations by 41.2% and 30.7% for the hierarchical perfect matching algorithm and the greedy algorithm respectively. Local optimizations may serve as a post-processing step for various co-scheduling algorithms.

8 Related Work

At the beginning of this project, we conduct an extensive survey, trying to find some existing explorations on similar problems in the large body of scheduling research. However, surprisingly, no previous work in traditional scheduling has been found tackling an optimal co-scheduling problem that contains performance interplay among jobs as what the current co-scheduling problem involves. As Leung summarizes in the *Handbook of Scheduling* [22], previous studies on optimal job scheduling have covered 4 types of machine environments: *dedicated*, *identical parallel*, *uniform parallel*, and *unrelated parallel* machines. On all of them, the running time of a job is fixed on a machine, independent of how other jobs are assigned, a clear contrast to the performance interplay in the co-scheduling problem tackled in this current work. Even though traditional Symmetric Multiprocessing (SMP) systems or NUMA platforms have certain off-chip resource sharing (e.g., on the main memory), the influence of the sharing on program performance has been inconsiderable for

scheduling and has not been the primary concern in previous scheduling studies. Some scheduling work [22] does have considered dependencies among jobs. But the dependencies differ from the performance interplay in co-scheduling in that the dependencies affect the order rather than performance of the execution of the jobs.

Recent studies on multi-core job co-scheduling fall into two categories. The first class of research aims at constructing practical on-line job scheduling systems. As the main effect of cache sharing is the contention among co-running jobs, many studies try to schedule jobs in a balanced way. They employ different program features, including estimated cache miss ratios, hardware performance counters, and so on [10], [18], [29]. All these studies aim at directly improving current runtime schedulers, rather than uncovering the complexity and solutions of optimal co-scheduling.

The second class of research is more relevant to optimal co-scheduling. A number of studies [2], [3] have proposed statistical models for the prediction of co-run performance. The models may ease the process for getting the data needed for optimal scheduling.

Beside co-scheduling, researchers have explored some other approaches to exploiting shared resource in multi-core architectures. In a recent study, Zhang and others [33] have found that the effects of thread co-scheduling become prominent for many multithreading applications only after some cache-sharing–aware transformations are applied. Several other studies [19], [28] have explored the effects of program-level transformations for enhancing the usage of shared cache. In addition, some other studies have tried to alleviate cache contention through cache partitioning [7], [25], cache quota management [26], and so forth.

9 Conclusions

This paper describes a study on the analysis of the complexity and the design of efficient algorithms for determining the optimal co-schedules for jobs running on CMP. It presents a set of discoveries, including the polynomial-time optimal co-scheduling algorithm for dual-core systems, the proof of the NP-completeness of the co-scheduling problem for systems with more than two cores per chip, the IP/LP formulation of the optimal co-scheduling problem, and a spectrum of heuristics-based algorithms for complex problems. Experiments on both real and synthetic problems validate the optimum of the results by the optimal co-scheduling algorithms, and demonstrate the effectiveness of the heuristics-based algorithms in producing near-optimal schedules with good efficiency and scalability.

Acknowledgments

We thank Cliff Stein from Columbia University and William Cook from Georgia Tech for their helpful comments on perfect matching algorithms. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and 0954015 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors 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.

References

[1] "Gnu linear programming kit," available at http://www.gnu.org/software/glpk/glpk.html.

[2] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 89–99.

[3] 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, pp. 340–351.

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

[5] S. Dandamudi, *Hierarchical Scheduling in Parallel and Cluster Systems*. Kluwer, 2003.

[6] 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, pp. 117–126.

[7] X. Ding, J. Lin, Q. Lu, P. Sadayappan, and Z. Zhang, "Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008, pp. 367–378.

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

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

[10] 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, pp. 25–38.

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

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

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning*. Springer, 2001.

[14] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantiative approach*. Morgan Kaufmann, 2007.

[15] 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, pp. 13–22.

[16] 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, pp. 263–273.

[17] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, October 2008, pp. 220–229.

[18] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *Proceedings of The International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, 2010, pp. 201–215.

[19] Y. Jiang, E. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Proceedings of the International Conference on Compiler Construction*, 2010.

[20] R. Karp, "Reducibility among combinatiorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[21] 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, pp. 111–122.

[22] J. Y.-T. Leung, *Handbook of Scheduling*. Chapman & HallCRC, 2004.

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

[24] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, pp. 575–601, 1992.

[25] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the International Symposium on Microarchitecture*, 2006, pp. 423–432.

[26] 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, pp. 2–12.

[27] S. Russell and P. Norvig, *Artificial Intelligence*. Prentice Hall, 2002.

[28] S. Sarkar and D. Tullsen, "Compiler techniques for reducing data cache miss rate on a multithreaded architecture," in *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, 2008, pp. 353–368.

[29] A. Snavely and D. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 66–76.

[30] G. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002, pp. 117–128.

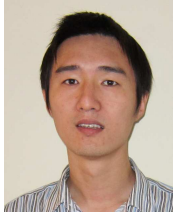
[31] K. Tian, Y. Jiang, and X. Shen, "A study on optimally co-scheduling jobs of different lengths on chip multiprocessors," in *Proceedings of ACM Computing Frontiers*, 2009, pp. 41–50.

[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*, 2003, pp. 26–35.

[33] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs?" in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 203–212.

[34] Y. Zhang, "Solving large-scale linear programs by interior-point methods under the matlab environment," University of Maryland, Tech. Rep. 96-01, July 1995.

[35] S. Zhuravlev, S. Blagodurov and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 129–142.



Yunlian Jiang is a PhD student in computer science department at the College of William and Mary, USA. He received his Master and Bachelor degrees from the Univeristy of Science and Technology of China in 2006 and 2003, both in computer science. His research interests focus on cache-contention-aware job co-scheduling, program behavior profiling and analysis for performance improvement.

Kai Tian is a PhD student in computer science department at the College of William and Mary, USA. He received his Bachelor degree in Computer Science (esp. Software Engineering) from Shanghai Jiao Tong University, China in 2006. His research area focuses on program dynamic optimization, online input-centric optimization for Java programs, program online profiling and analysis for performance improvement, parallel computing and its applications.

Xipeng Shen is an Assistant Professor of Computer Science at the College of William and Mary, USA. He received his Ph.D. and Master degrees in Computer Science from University of Rochester. He is an IBM CAS Faculty Fellow, a recipient of the National Science Foundation CAREER Award. He received the best paper award from ACM PPoPP 2010. Xipeng Shen's research in Compiler Technology and Programming Systems aims at helping programmers achieve high performance as well as good programming productivity on both uniprocessor and multiprocessor architectures. He is particularly interested in the effective usage of memory hierarchies, the exploitation of program inputs in program behavior analysis, and the employment of statistical learning in runtime systems and dynamic program optimizations.

Jinghe Zhang is a graduate student in computer science at the University of North Carolina (UNC) at Chapel Hill. Her current research interests include optimal sensor placement for the power grid, and large scale estimation in general. Zhang has a B.S. from the University of Science and Technology of China (2006), and an M.S. from the University of Idaho (2008), both in mathematics. She is an IEEE member.

Jie Chen is a Senior Computer Scientist at Thomas Jefferson National Accelerator Facility (Jefferson Lab). He obtained his Ph.D degree in physics from Rutgers University in 1993. His research interests are in the areas of parallel message passing, cache coherence and heterogeneous computing.

Rahul Tripathi received his Ph.D. in Computer Science from the University of Rochester, Rochester, NY in May 2005. Rahul holds a B.Tech. in Computer Science and Engineering from the Indian Institute of Technology, Kanpur in 2000 and an M.S. in Computer Science from the University of Rochester in 2003. He is currently an assistant professor in the Department of Computer Science and Engineering at the University of South Florida, Tampa. His research interests are in the areas of algorithms and computational complexity theory.