Optimal Co-Scheduling to Minimize Makespan on Chip Multiprocessors

Kai Tian[†] Yunlian Jiang[⋆‡] Xipeng Shen[†] Weizhen Mao[†]

[†]Computer Science Department, The College of William and Mary
{ktian,xshen,wm}@cs.wm.edu

[‡]Google
yunlian@google.com

Abstract. On-chip resource sharing among sibling cores causes resource contention on Chip Multiprocessors (CMP), considerably degrading program performance and system fairness. Job co-scheduling attempts to alleviate the problem by assigning jobs to cores intelligently. Despite many heuristics-based empirical explorations, studies on *optimal* co-scheduling and its inherent complexity start only recently, and all have concentrated on the minimization of total performance degradations. There is another important criterion for scheduling, makespan, which determines the finish time of a job set. Its importance for job co-scheduling on CMP is increasingly recognized, especially with the rise of CMP-based compute cloud, data centers, and server farms. However, optimal co-scheduling for makespan minimization still remains unexplored.

This work compares makespan minimization problem with previously studied cost minimization (or degradation minimization) problem, revealing these connections as well as significant differences. It uncovers the computational complexity of the makespan minimization problem, and proposes a series of algorithms to either compute or approximate the optimal schedules. It proves that the problem is NP-complete in a general setting, but for a special case (dual-core without job migrations), the problem is solvable in $O(n^{2.5} \cdot \log n)$ time (n is the number of jobs). In addition, this paper presents a series of algorithms to compute or approximate the optimal schedules in the general setting. Experiments on both real and synthetic problems verify the optimality of the optimal co-scheduling algorithms, and demonstrate the reasonable accuracy and scalability of the approximation algorithms. The findings may advance the current understanding of optimal co-scheduling, facilitate the evaluation of real co-scheduling systems, and provide insights for the development of practical co-scheduling algorithms.

1 Introduction

In a Chip Multiprocessors (CMP) system, multicores typically share certain resource (e.g., last-level cache) on a chip. The sharing, although shortening the communication among cores, causes resource contention among co-running jobs. Many studies have

[⋆] This work was done when Yunlian Jiang was in The College of William and Mary.

reported considerable, and sometimes significant, effects of the contention on program performance and system fairness [4, 7, 9, 14, 22]. The urgency for alleviating the contention keeps growing as the processor-level parallelism increases continuously.

Recent years have seen many interests in using job co-scheduling to alleviate the contention [7, 10, 19]. The basic strategy of job co-scheduling is to assign jobs to cores in a way that the overall influence from resource contention is reduced.

The prior explorations fall into two categories. The first includes the research that aims at constructing practical on-line job scheduling systems. It concentrates on heuristics-based lightweight scheduling techniques. A typical example is the symbiotic co-scheduling by Snavely and Tullsen [19]. The co-scheduler in the system samples the performance of different schedules during runtime and selects a good one. Other examples include the fair-miss-rate-based co-scheduling by Fedorova et al. [7], thread clustering by Tam et al. [23], and so on.

The second category includes the studies on optimal co-scheduling. The goal is to uncover the complexity in finding optimal co-schedules and develop algorithms to either compute or approximate optimal co-schedules. Optimal co-scheduling typically happens offline, requiring considerable computation, certain knowledge obtained through profiling runs of jobs, and other conditions. It is not for direct uses in on-line job scheduling systems, but for exposing the limit to facilitate the evaluation of practical schedulers. Without knowing optimal schedules (or a reasonable approximation), it is hard to precisely determine how good a scheduling algorithm is—how far a solution given by the scheduling algorithm is from the optimal solution and whether further improvement will enhance performance significantly, both of which are important for the design and deployment of practical co-scheduling systems.

Research in optimal co-scheduling is still in a preliminary stage. Although some studies are relevant to optimal co-scheduling (e.g., co-run cache performance prediction [2, 4] may simplify the profiling requirement), direct attacks to the problem start only recently [10, 11, 24]. The objectives of the previous explorations are all on the minimization of co-run cost (i.e., the sum of each job's co-run performance degradation).

But besides cost, there is another important criterion in job scheduling, makespan. *Makespan* refers to the time between the start of a job set and the finishing of the last job in the set. Minimizing makespan is important in situations where a simultaneously received batch of jobs is required to be completed as soon as possible. For example, a multi-item order submitted by a single customer needs to be delivered in the minimal time. This kind of situation is especially common in server farms, data centers, and compute cloud (e.g., the Amazon Elastic Compute Cloud). With the rapid rise of these modern computing forms and their wide adoption of CMP, a good understanding to makespan minimization in multicore job co-scheduling becomes increasingly important. But to the best of our knowledge, this problem has remained unexplored.

Makespan minimization differs from cost minimization. The optimal schedules for the two criteria are typically different. In traditional job scheduling literature, the two criteria have led to drastically different algorithms and complexity analyses [12]. As to be shown in this paper, for multicore job co-scheduling, the implication of their difference is pronounced as well, ranging from complexity analysis to algorithm designs to the ultimate scheduling results (summarized in Section 7).

Motivated by the contrast of the increasing importance and the preliminary understanding of makespan minimization in multicore job co-scheduling, we initiate explorations on four aspects.

- First, we prove that makespan minimization in job co-scheduling is NP-complete on systems with more than 2 cores per chip. The proof is based on a reduction from the problem of Exact Cover by 3-Sets. We are not aware of any previous analysis of the computational complexity.
- Second, by offering an $O(n^{2.5} \cdot \log n)$ algorithm (n is the number of jobs), we prove that on dual-core systems with no job migrations, the problem is polynomial-time solvable. To the best of our knowledge, this algorithm is the first polynomial-time solution for this optimal co-scheduling problem.
- Third, we present a set of A*-search–based algorithms and a greedy algorithm to tackle optimal co-scheduling for makespan minimization in the general setting—with two or more cores per chip and with or without job migrations. A*-search has been applied for job co-scheduling [24], but not for makespan minimization. Our description focuses on the issues specific to makespan minimization, including the formulation of the search process, the design of the heuristic function, and the empirical exploration of the tradeoff between the scheduling overhead and quality.
- Finally, we evaluate the algorithms on both real and synthetic problems, verifying the optimality of the co-scheduling algorithms (under certain conditions), meanwhile showing that the algorithms may save orders of magnitude overhead over the brute-force search. Results of the approximation algorithms demonstrate their capability to achieve near optimal solutions with reasonable scalability.

The analysis and algorithms contributed in this paper help reveal (or approximate) the lower bound of makespan in multicore job co-scheduling, essential for the assessment of practical scheduling systems. The algorithms may shed insights to the development of effective lightweight co-scheduling systems as well.

We organize the rest of this paper as follows. Section 2 describes the problem setting and assumptions. Section 3 proves the NP-completeness of the optimal co-scheduling problem, and presents the polynomial-time algorithm and a set of A*-search algorithms as optimal solutions. Section 4 describes a set of approximation algorithms. Section 5 reports evaluation results. Section 6 discusses the limitations of this work and future extensions. After reviewing some related work in Section 7, we conclude the paper with a short summary in Section 8.

2 Problem Definition

Roughly speaking, the optimal job co-scheduling tackled in this work is to decide the placement of a set of jobs on a number of cores so that the makespan of the schedule is minimized.

Finding optimal co-schedules in a general setting is extremely difficult: A program's fine-grained behaviors may change constantly, a program may migrate to any cores, and programs may start, terminate, or go through context switch at any time. It is necessary to define the problem settings first.

2.1 Problem Settings

To make the problem tractable and meanwhile keep the analysis useful, we specify the following settings. Some of these settings may differ from certain practical scenarios. However, as we will show (after presenting the settings), they do not prevent the use of the computed co-schedules from serving for its main goal: facilitating the evaluation of practical co-schedulers.

Machines. The computing system assumed in this exploration contains m uniform chips, and each chip has u uniform cores[1]. There is a certain amount of cache on each chip that is shared by the u cores on the chip. Only one job can run on a core at each time point. The execution speed of a job running on a chip depends on what jobs are placed on the same chip, but has negligible dependence on how the rest of the job set are placed on other chips. The architecture is a generalized form of CMP architectures on the market, such as IBM Power5 and the Intel Core2 family.

Jobs. The number and starting time of jobs are set to be as follows. The number of jobs (denoted as n) is equal to the number of cores, $n = m * u$. This setting is to help focus on the placement of jobs on cores. When $n < m*u$, the problem can be converted to the defined setting if we consider that there are $(m*u-n)$ extra dummy jobs that consume no resources. If $n > m * u$, the problem is more complex, requiring the consideration of temporal complexity (e.g. context switch) besides the spatial placement of jobs. The temporal complexity is out of the scope of this paper. But we note that this work will be still useful for that setting, as spatial placement still exists as a sub-problem in it.

All the jobs must start at the same time. This is a typical assumption in both traditional job scheduling [12] and recent job co-scheduling [10]. This setting may differ from the scenarios in real-time scheduling. However, recall that the main targeted scenario of makespan minimization is batch job processing, in which, all jobs typically arrive at the system at the same time.

Job Migrations. A job can migrate from one core to another, but the migration only happens when any of the jobs terminates. This setting comes from the following reason. As well known, keeping a process on a processor is good for locality. As a result, in practical systems like Linux, occurrences of job migrations are mostly triggered by load imbalance [1]. In our setting, as the number of jobs equals the total number of cores, load changes only when some job finishes. Therefore, allowing job migration only at those times does not cause large departure from real scenarios.

This work focuses on job co-scheduling inside a multicore machine, which is the primary component of the scheduling in any large multicore-based systems. So it assumes that all processor chips are in the same machine and the migrations of a job among different chips have similar overhead. (With certain extensions, the developed algorithms may be applicable to clusters consisting of multiple nodes. The extensions

[1] We use the term “cores” for simplicity of discussion. As shown in Section 5, the techniques can also be applied to thread scheduling in SMT systems.

are mainly on the consideration of the different overhead of migration within and across cluster nodes.)

Performance Data. As assumed in previous work [10], the following performance information is given: the time for a job to finish if it runs alone (i.e., no other jobs running on the chip), and the performance degradation (defined as the rate between the co-run time and the single-run time of the job) of the job when it co-runs with k ($0 < k < u$) other jobs in the job set. These performance data can be obtained through offline profiling runs or predictive models [2, 4]. The overhead in gathering the data is not an issue for optimal co-scheduling: Finding optimal co-schedules is not for direct real-time scheduling, but for providing a reference for the evaluation of practical co-schedulers. For a given u, the overhead to gather the single run and co-run times is polynomial in the number of jobs. It is typically negligible compared to the overhead in brute-force search for optimal co-schedules, which is exponential in the number of jobs.

Because a program execution may vary constantly, the performance degradation of a program in a co-run may vary across intervals. In our setting, we use the average degradation through the entire co-run. A future enhancement is to combine with program phase analysis [17, 18]. As previous studies do [10, 19], we currently ignore phase changes to concentrate on co-scheduling itself.

In our setting, jobs may relate with one another, but all degradations are greater than 1. As co-runs are typically slower than single-runs because of cache and bus contention, this setting holds in most cases.

Short Discussion.

The settings described in this section do not prevent the use of the optimal co-scheduling for evaluating practical schedulers. For example, the evaluation of a scheduler S on a machine with m chips and u cores per chip can proceed as follows. The developers first find $m * u$ applications that are typical for the target system. They start the applications at the same time on the machine with the scheduler S running to get the makespan, T. They then run the applications a number of times to obtain the single-run times and co-run degradations of those applications. After that, all the information needed by the problem setting is ready. By applying the optimal co-scheduling algorithms (to be described), they will get the minimum makespan, $\hat{T}$. The comparison between T and $\hat{T}$ will indicate the room for improvement of the scheduler S.

2.2 Problem Definition and Terminology

With the problem settings defined, the definition of the optimal co-scheduling problem to be tackled in this work is straightforward. It is to find a schedule that maps each job to a core under the settings defined in the previous subsection, so that the makespan of the schedule is minimized.

For the sake of clarity, we define several terms. The allowance of job migration suggests the opportunities for rescheduling the remaining jobs when some job finishes. In the following description, we call each scheduling or rescheduling point as *a scheduling stage*. So, if no job migrations are allowed, there is only one scheduling stage; when migrations are permitted, there are up to n scheduling stages. We use *an assignment* to refer to a group of u jobs that are to run on the same chip. *A sub-schedule* is a set of assignments that cover all the unfinished jobs and do not overlap with one another. *A schedule* is a set of all sub-schedules that are used from the start to the end of the executions of all jobs.

3 Complexities and Solutions of Makespan Minimization

In this section, we analyze the inherent complexity of the makespan minimization in job co-scheduling. We classify the problem instances into four cases: $u \geq 3$ with or without job migration allowed, or $u = 2$ with or without job migration allowed. Here, u is the number of cores per chip. We prove that the first two cases are NP-complete problems, but the fourth is polynomial solvable by a perfect-matching-based algorithm. The complexity of the third case is to be studied in the future. In addition, we present heuristic algorithms for all the four cases.

3.1 Complexity Analysis ($u \geq 3$, With or Without Job Migration)

When more than two cores share a cache on a chip ($u \geq 3$), the makespan minimization is an NP-complete problem. We prove this result by reducing a known NP-complete problem, *Exact Cover by 3-Sets* (X3C) [8], to our problem.

First, we formulate our co-scheduling problem as a decision problem. Given a system with m chips, each with $u \geq 3$ cores, there is a set J containing $n = m \cdot u$ jobs, which are to be scheduled on the cores. Consider all possible subsets of J with cardinality u, denoted by $J_1, \cdots, J_{\binom{n}{u}}$. For each J_i, which represents a group of u jobs that may be co-scheduled on the same chip, let w_i be the maximum co-run time of all the u jobs in J_i. The question in the decision problem is whether there are m disjoint subsets $J_{p_1}, \cdots, J_{p_m}$, where $p_1, \cdots, p_m \in \{1, \cdots, \binom{n}{u}\}$, to form a partition of J such that $\max_{i=1}^{m}\{w_{p_i}\} \leq B$ for any given bound B.

Note that the partition of J into m subsets of cardinality u is actually the construction of a schedule of n jobs on $m \cdot u$ cores and that $\max_{i=1}^{m}\{w_{p_i}\}$ is in fact the makespan of the schedule.

The problem is clearly in NP. We prove that it is NP-complete via a reduction from X3C, in which given a set X with $|X| = 3m$ and a set $C = \{C_i | C_i \subseteq X \text{ and } |C_i| = 3\}$, the question to ask is whether C contains an exact cover for X, i.e., m disjoint members of C, say $C_{p_1}, \cdots, C_{p_m}$, that makes a partition of C.

The reduction from X3C to our co-scheduling problem is straightforward. Given any instance of X3C, namely X and C, we define an instance for co-scheduling, where (1) $J = X$ with $n = 3m$ and $u = 3$, (2) for any $J_i \subseteq J$ with $|J_i| = 3$, if $J_i \in C$ then let $w_i = 1$, and if $J_i \notin C$ then let $w_i = 2$, and (3) $B = 1$.

The construction of the instance for co-scheduling can be done in $O(n^3)$ time. Furthermore, it is easy to show that C contains an exact cover for X if and only if there is a schedule of jobs in J to the $3m$ cores with a makespan no more than 1. Therefore, the co-scheduling problem with $u = 3$ is NP-complete.

The above proof holds regardless of whether job migration is allowed or not. This is because in both settings, finding a schedule with makespan no more than 1 is equivalent to finding an exact cover.

3.2 Polynomial-Time Solution ($u = 2$, No Job Migration)

We prove that, when $u = 2$ and no job migrations are allowed, the optimal co-schedules can be found in polynomial time. We describe an $O(n^{2.5} \cdot \log n)$ algorithm as follows.

The algorithm uses a fully-connected graph, namely a *co-run makespan graph*, to model the optimal co-scheduling problem. Each vertex represents a job; the weight on an edge is the longer running time of the two jobs (represented by the two vertices connected by the edge) when they co-run together.

Before describing the algorithm, we introduce the concept of a perfect matching. A *perfect matching* in a graph is a subset of edges that cover all vertices of the graph, but no two edges share a common vertex. We define the *bound* of a perfect matching as the largest weight of all the edges it covers. It is easy to see that the perfect matching of a co-run makespan graph with the minimum bound corresponds to a solution to the makespan minimization problem: Each edge corresponds to an assignment (i.e., co-run group) and the makespan equals to the bound of the perfect matching.

There are some algorithms for finding the minimum-weight perfect matching on a weighted graph [6,8]. However, they cannot apply to our problem directly because their objective functions are typically the sum of edge weights, rather than the maximum of edge weights in our problem.

We develop an algorithm to determine a minimum-bound perfect matching as shown in Figure 1. We first construct a sorted list containing all the edges of a co-run makespan graph in an ascending order of their weights; the edge with the smallest weight resides on the top of the list. We then use a binary search to determine the smallest top portion of the sorted edge list that contains a perfect matching (regardless of weights) covering all vertices. The binary search starts with the top half of the edge list and checks whether a perfect matching can be found in those edges. A negative answer would suggest that more edges are needed, so the algorithm would try the top three quarters of the edge list. A positive answer would suggest that a smaller portion of the list may be enough to contain a perfect matching, so the algorithm would try the top quarter of the edge list. This binary search continues until it finds the smallest top portion of the edge list that contains a perfect matching.

We claim that the resulted perfect matching is an *optimal* perfect matching on the original co-run makespan graph—that is, no perfect matchings on the original co-run makespan graph have bounds smaller than the bound of the resulted perfect matching. The proof is as follows.

Let M be the perfect matching produced by the algorithm, T be the makespan of the corresponding schedule, and S be the smallest top portion of the edge list that contains M. According to the algorithm, S is the smallest among all top portions that contains a perfect matching.

Assume that there is a perfect matching M' whose makespan T' is smaller than T. Let E' be the set of edges included in M'. Let S' be a set containing all the edges in the sorted edge list from the top to the heaviest edge in E'. Because the edge list is sorted in the ascending order of edge weights, $E' \subseteq S'$. So, S' contains a perfect matching. Because $T' < T$, the weights of all the edges in E' and thus in S' must be smaller than T. While T is the weight of some edge in S, hence $S' \subset S$. This contradicts with the assumption that S is the smallest top portion of the edge list that contains a perfect matching, thus the proof completes.

The time complexity of the perfect matching detection subroutine, *findPerfMatch(G)*, is $O(\sqrt{n} \cdot m)$ [8], where n and m are the numbers of vertices and edges in the graph. In the algorithm, the binary search process contains $O(\log n)$ invocations of perfect matching detection. The value of m can be no greater than n^2. The time complexity of the algorithm is $O(n^{2.5} \cdot \log n)$.

```
/* $V$: vertex set; $E$: edge set */
/* $S$: generated perfect matching */
$L \leftarrow$ sortEdges($E$);
lbound $\leftarrow$ 1; ubound $\leftarrow$ |$L$|;
$G.vertices \leftarrow V$; $S \leftarrow \emptyset$;
while (1) {
  curPos $\leftarrow \lfloor$ (ubound+lbound)/2 $\rfloor$;
  if (curPos == ubound) return $S$;
  $G$.edges $\leftarrow$ $L$[1:curPos];
  $S \leftarrow$ findPerfMatch(G);
  if ($S \neq$ NULL)
   ubound $\leftarrow$ curPos;
  else
   lbound $\leftarrow$ curPos;}
```

Fig. 1. Algorithm for minimum-bound perfect matching.

3.3 Search-Based Optimal Co-Scheduling

The polynomial-time algorithm described in the previous section works only for dual-core systems without job migrations. This section presents a search-based approach, which is applicable to larger systems and supports job migrations.

Background on A*-Search A*-search, stemming from artificial intelligence, is designed for fast graph search. It is optimally efficient for any given heuristic function—that is, no other search-tree-based optimal algorithm is guaranteed to expand fewer nodes than A* search, for a given heuristic function [15]. Its completeness, optimality, and optimal efficiency lead to the adoption for the search of optimal schedules.

For a tree search, where the goal is to find a path from the root to an arbitrary leaf with the total cost minimized, A* search defines a function $f(v)$ to estimate the lowest cost of all the paths passing through the node v. A* search maintains a priority list, initially containing only the root node. At each step, A* search removes the top element—that is, the node with the highest priority—from the priority list, and expands that node. After the expansion, it computes the $f(v)$ values of all the newly generated nodes, and put them into the priority list. The priority is proportional to $1/f(v)$. This expansion process continues until the top of the list is a leaf node, indicating that no other nodes in the list need to be expanded any more as their lowest cost exceeds the cost of the path that is already discovered.

The definition of function $f(v)$ is the key for the solution's optimality and the algorithm's efficiency. There are two properties related to $f(v)$:

- A* search is optimal if $f(v)$ never overestimates the cost.
- The closer $f(v)$ is from the real lowest cost, the more effective A*-search is in pruning the search space.

Application to Minimize Makespan for Job Co-Scheduling Although A*-search has been used for cost minimization problems [24], some substantial changes are necessary for applying it to makespan minimization. Specifically, we need redefine the structure of the search tree and the cost estimation function $f(v)$. This section presents our respective definitions for the scenarios with and without job migrations.

No Job Migrations When no job migrations are allowed, the scheduling problem is essentially to partition jobs into a number of co-run groups. Figure 2 illustrates our definition of the search tree. Each non-root tree node (say v) corresponds to a set, $S(v)$, that contains u distinct jobs. The nodes in the tree are arranged as follows. Let $R(v)$ represent the set of jobs that have never been covered by any node on the path from root to v. Suppose w is a child node of v. All jobs in $S(w)$ must belong to $R(v)$ (i.e., $S(w) \subseteq R(v)$) and $S(w)$ must contain the job whose index[2] is the smallest in $R(v)$. With such an organization, each path from the root to a leaf offers a schedule. All the paths in the tree together constitute the schedule space.

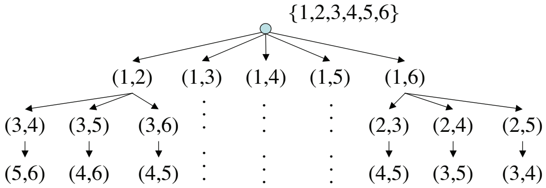

Fig. 2. An example of the search tree for the cases with no job migrations. There are 6 jobs to be scheduled to 3 dual-core chips. Each non-root tree node corresponds to a set of u (here $u = 2$) distinct jobs; the set must contain the job whose index is the smallest among all the jobs that are not covered from the root to this node. Each path from the root to a leaf therefore offers a schedule.

We define the cost estimation function $f(v)$ as follows. Let A represent the set of all n jobs, and P' be the path from the root to the node v. It is easy to see that the minimum

[2] We assume that each job has a unique index number.

makespan of any schedule (or path) passing node v must be either the makespan of the jobs $A - R(v)$ (i.e., the jobs covered by the path from the root to the node v) or the minimum makespan of the remaining jobs, $R(v)$. The former can be computed from the assignments represented by P'. The latter can be no smaller than the maximum of the minimal co-run times of the jobs in $R(v)$, which can be computed from the given co-run degradations. We then define $f(v)$ as the maximum of the two values.

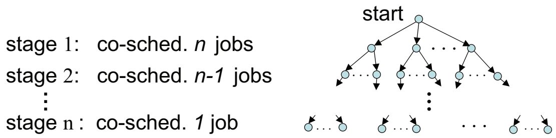
start
stage 1: co-sched. n jobs
stage 2: co-sched. n-1 jobs
stage n : co-sched. 1 job

Fig. 3. Search tree for the cases with rescheduling allowed at the end of a job. Each level corresponds to a scheduling stage. Each node, except the root, represents a sub-schedule of the jobs that have not finished yet.

With Job Migrations Similar to the previous work [11], we use a search tree to model the co-scheduling problem when job migrations are allowed, as illustrated by Figure 3. It differs from Figure 2 in that each non-root node represents a sub-schedule (i.e., a set of assignments that cover all the unfinished jobs and have no overlap with each other) rather than a group of co-running jobs.

For n jobs, there are at most n scheduling stages; each corresponds to a time point when one job finishes since the previous stage. The nodes at a stage, say stage i, correspond to all possible sub-schedules for the $n - i + 1$ remaining jobs. There is a cost associated with each edge. Consider an edge from node a to node b. The number of unfinished jobs in b is typically one less than in a. The weight on the edge is the time for that job to finish since the scheduling stage of a.

The makespan minimization becomes to find a path from the root to any leaf node so that the sum of the weights on the path is the minimum. We define $f(v)$ as the sum of two quantities. One is the total weights from the root to the node v, the other is the longest single-run time of the remaining jobs.

Given the NP-completeness of the problem, it is not surprising that A*-search is subject to scalability issues. Our explorations aim at revealing the extent of its scalability, and shedding insights for the design of approximation algorithms.

4 Approximation Algorithms

To achieve good scalability, we develop three approximation algorithms based on the enlightenment from the optimal co-scheduling algorithms presented in the previous section. The first two algorithms are applicable generally, while the third one applies only to dual-core cases.

4.1 Combination with Clustering

The combination of A*-search and clustering may provide further flexibility for striking a tradeoff between overhead and quality of co-scheduling. We call such combined algorithms “A*-cluster” algorithms. We first describe its application to the cases with job migrations, and then outline its uses when migrations are not allowed.

At each (re)scheduling time, the unfinished jobs are clustered based on the length of their remaining portions. The algorithm controls the number of rescheduling stages by rescheduling only when a cluster of jobs finish. It also avoids the generation of sub-schedules that are similar to one another.

We say that a sub-schedule is substantially different from another one if they are not equivalent when we regard all jobs in a cluster as the same. For example, four jobs fall into two clusters as $\{\{1\ 2\}, \{3\ 4\}\}$. We regard the sub-schedule (1 3) (2 4) equivalent to (1 4) (2 3), but different from (1 2) (3 4). (Each pair of parentheses contains a co-run group.)

Among various clustering techniques, we use a simple distance-based clustering approach as the data are one dimensional. Given a sequence of data to be clustered, we first sort them in an ascending order. Then, we compute the differences between every two adjacent data items in the sorted sequence. Large differences are considered as indication of cluster boundaries. A difference is regarded as large if its value is greater than $d + \delta$, where d is the mean value of the differences in the sequence and δ is the standard deviation of the differences.

The A*-cluster algorithm reduces both the height and the width of the search tree. The number of children of a node is reduced from factorial, $\prod_{i=0}^{\frac{r}{u}-1}\binom{r-i*u-1}{u-1}$, to polynomial, $O(r^{\gamma})$, where C is the number of clusters and r is the number of unfinished jobs, and $(\gamma = C + (C^u - C)/u!)$.

For the cases without job migrations, the jobs are clustered based on their single-run time. The algorithm prunes the width of the search tree by removing the nodes that are not significantly different from their siblings, similar to the pruning in the case with migrations. The tree height remains unchanged.

4.2 Greedy Algorithm

The greedy algorithm is simpler than the A*-cluster algorithm. At each (re)scheduling stage, the algorithm iteratively determines the assignments for the remaining jobs. In each iteration, it finds the best co-run group for the job whose remaining part has the longest single-run time among all the unfinished jobs. Its intuition is that the longest jobs typically determine the makespan of a schedule.

4.3 Local Perfect-Matching Algorithm

This approximation algorithm is a generalized version of the perfect-matching-based algorithm proposed in Section 3.2. The extension makes it applicable to dual-core cases with job migrations. At each (re)scheduling point, it applies the perfect-matching-based algorithm to the unfinished jobs to obtain a locally optimal sub-schedule. As the perfect-matching-based algorithm assumes that the number of remaining jobs equals the number of cores in the computing system, we treat the jobs that have finished as pseudo-jobs, which exist but consume no computing resource. This strategy may introduce certain amount of redundant work, but it offers an easy way to generalize the perfect-matching-based algorithm. The time complexity of this algorithm is $O(n^{3.5} \cdot \log n)$.

There is a side note on the scheduling algorithms that allow migrations. A migration of different programs may have different overhead. However, because in our setting, migrations happen only when some job finishes, the total number of job migrations is small (less than the number of jobs); the total overhead covers only a negligible portion of the makespan (confirmed in Section 6). In our experiments, we use the average overhead measured on the experimental platform as the overhead of a migration when comparing the makespan of different schedules.

Table 1. Benchmarks

Benchmark	single-run	co-run degrad rate		
	time (s)	min %	max %	mean %
fmm*	5.63	0.77	11.28	3.67
ocean*	13.52	2.13	58.81	19.73
ammp	21.10	1.66	30.24	12.62
art	2.22	2.31	75.42	27.78
bzip	10.90	0.00	38.95	3.31
crafty	6.75	0.07	12.33	4.95
equake	11.05	6.42	78.00	26.46
gap	2.90	2.09	34.34	11.02
gzip	14.10	0.00	13.06	2.19
mcf	7.86	8.23	125.36	42.37
mesa	15.33	0.65	15.15	5.18
parser	3.74	1.74	37.75	13.51
twolf	5.42	0.00	15.73	5.21
vpr	4.58	3.31	42.52	18.30

$*$: from SPLASH-2. Others from SPEC CPU2000.

5 Evaluation

We evaluate the co-scheduling algorithms on two kinds of architecture. For CMP co-scheduling, the machines are equipped with quad-core Intel Xeon 5150 processors running at 2.66 GHz. Each chip has two 4MB L2 cache, each shared by two cores. Every core has a 32KB dedicated L1 data cache. For Simultaneous Multithreading (SMT) co-scheduling, the machines contain Intel Xeon 5080 processors (two 2MB L2 cache per chip) clocked at 3.73 GHz with Hyper-Threading enabled (two hyperthreads per computing unit.)

The job suite includes 14 programs: 2 parallel programs from SPLASH-2 [21] and 12 sequential programs randomly selected from SPEC CPU2000. Each of the two parallel program has two threads; so, we have 16 jobs in total. We do not use the programs from the entire benchmark suites because the large problem size would make it infeasible to compare the scheduling algorithms, especially with the brute-force search algorithm. We use the two parallel programs to examine the applicability of the co-scheduling algorithms for parallel (in addition to sequential) applications. Table 1 lists the programs with the ranges of their co-run degradations on the Intel Xeon 5150 processors. The big ranges suggest the potential for co-scheduling.

In addition, we generate some sets of jobs whose single-run time and co-run degradations are set randomly. The use of these synthetic problems helps overcome the limitations imposed by the particular benchmark set.

For each set of jobs, we test the scheduling in cases both with and without job migrations (denoted as "no rescheduling" and "rescheduling" respectively.) The difference reflects the benefits of rescheduling.

5.1 Comparison to the Optimal

This section concentrates on the verification of the optimality of the A*-search and perfect-matching-based algorithms. We stress that the optimality is under the settings defined in Section 2. We compare the results of those scheduling algorithms with the best schedules found through brute-force search.

Because of the scalability issue of the brute-force search, we use 8 jobs for the comparison. We use the top 6 programs (8 jobs considering the parallel threads) in Table 1, along with a number of synthetic job sets. Table 2 reports the results. For each synthetic setting, we generate 3 problems in that setting, referred to as the 3 trials in the table.

The data surrounded by boxes are from the optimal co-scheduling algorithms (the rest are from the approximation algorithms.) The two halves of the "matching" row correspond to the precise algorithm in Section 3.2 and the approximation algorithm in Section 4.3, respectively. Both algorithms are applicable only to 2-core cases.

Optimality The data in Table 2 show that the optimal algorithms generate schedules with the same makespans as the schedules found by the brute-force search. For the 8 real jobs on Xeon 5150 (2-cmp), for instance, the optimal schedule found by the 3 algorithms are all as follows: (fmm-1,ocean-1), (ammp,cafty), (art,bzip), (fmm-2,ocean-2), where fmm-n and ocean-n are their nth threads, and each pair of parentheses include a co-running group. The makespan is 0.5% larger than the makespan when the programs run in isolation.

The bottom 3 rows in Table 2 reveal the minimum, median, and maximum of the makespans of 100 randomly generated schedules, corresponding to the scheduling in many existing systems, which work in a cache-sharing-oblivious manner. The minimum makespans are close to the optimal in the "no rescheduling" cases, but are mostly over 10% larger than the optimal in the "rescheduling" cases. The median and maximum are significantly larger than the optimal. For the 8 real jobs, although random scheduling is

Table 2. Co-schedule makespan on 8 real jobs and a series of synthetic scheduling problems (each has 8 jobs). The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation. The real jobs run on two architectures: Intel Xeon 5150 (2-cmp) and Intel Xeon 5080 (2-smt). The synthetic scheduling problems use both dual-core (2-core) and quad-core (4-core) systems.

	no rescheduling								rescheduling							
jobs	real		synthetic						real		synthetic					
arch.	2-cmp	2-smt	2-core			4-core			2-cmp	2-smt	2-core			4-core		
trial			1	2	3	1	2	3			1	2	3	1	2	3
brute-fc	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65	1.002	1.013	1.33	1.21	1.19	1.99	1.93	1.56
A*	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65	1.002	1.013	1.33	1.21	1.19	1.99	1.93	1.56
matching	1.005	1.023	1.49	1.49	1.58	-	-	-	1.002	1.023	1.37	1.43	1.52	-	-	-
A*-clstr	1.005	1.167	1.55	1.75	1.58	2.38	2.30	1.65	1.012	1.023	1.55	1.48	1.29	2.19	2.12	1.63
greedy	1.005	1.170	1.49	1.90	1.80	2.77	2.34	1.85	1.005	1.170	1.43	1.90	1.80	2.32	2.08	1.87
rand-min	1.005	1.023	1.55	1.49	1.69	2.24	2.16	1.65	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65
rand-med	1.016	1.255	1.81	2.70	2.22	2.55	2.34	1.88	1.016	1.196	1.81	2.70	1.92	2.54	2.33	1.87
rand-max	1.161	1.329	2.72	3.30	2.66	3.13	2.91	2.68	1.161	1.329	2.72	3.30	2.66	3.13	2.91	2.68

likely to produce near optimal makespan in the Xeon 5150 system, it causes over 20% makespan increase on the SMT systems. These results indicate the risks of neglecting cache sharing in job scheduling.

The comparison between the "no rescheduling" and "rescheduling" results shows that when the "no rescheduling" algorithms cause non-negligible makespan increase, rescheduling is usually able to reduce the makespan considerably.

Table 3. The numbers of nodes visited and the time spent by different co-scheduling algorithms on 8 jobs on Intel Xeon 5080

	no resch.		resch.	
	nodes	time (ms)	nodes	time (ms)
brute-fc	210	41	16643446	419332
matching	1	47	4	179
A*	37	23	4405	718
A*-clstr	8	5	32	35
greedy	1	2	4	6
random	-	1	-	1

Overhead Table 3 reports the numbers of search-tree nodes visited and the milliseconds spent by different co-scheduling algorithms on 8 jobs on Intel Xeon 5080. In the "rescheduling" case, relative to the brute-force search, both of the two optimal co-scheduling algorithms save the search time by several orders of magnitude.

Comparison with Cost Minimization As mentioned earlier in this paper, the two scheduling criteria, makespan and total cost, typically lead to different results. It is confirmed by the experimental results. For example, Figure 4 shows the optimal schedules (without rescheduling) for both criteria on the Xeon 5080 (2-smt) machine. The schedule with minimum total cost turns out to have 33% larger makespan than the schedule from the makespan minimization algorithms. On the other hand, the schedule with minimum makespan causes extra cost as well. This difference confirms the need for studies on each of the criteria and the application of the corresponding algorithms in different scenarios.

cost minimization:
schedule: (fmm-1, crafty), (fmm-2, ocean-1), (occean-2, art), (ammp, bzip)
cost (ie., total degradation): 12.13
makespan: 58.02 sec

makespan minimization:
schedule: (fmm-1, bzip), (fmm-2, art), (ocean-1, ammp), (ocean-2, crafty)
cost (ie., total degradation): 12.88
makespan: 43.56 sec

Fig. 4. Optimal schedules for cost minimization and makespan minimization on Xeon 5080 (2-smt) with no rescheduling.

Table 4. Co-schedule makespan on 16 real jobs and a series of synthetic scheduling problems (each has 16 jobs). The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation. The real jobs run on two architectures: Intel Xeon 5150 (2-cmp) and Intel Xeon 5080 (2-smt). The synthetic scheduling problems use both dual-core (2-core) and quad-core (4-core) systems.

	no rescheduling								rescheduling							
jobs	real		synthetic						real		synthetic					
arch.	2-cmp	2-smt	2-core			4-core			2-cmp	2-smt	2-core			4-core		
trial			1	2	3	1	2	3			1	2	3	1	2	3
matching	1.005	1.033	1.26	1.14	1.2	-	-	-	1.002	1.033	1.2	1.07	1.11	-	-	-
A*-clstr	1.005	1.059	1.42	1.22	1.21	1.97	1.87	1.93	1.005	1.107	1.37	1.20	1.22	1.99	1.86	1.91
greedy	1.005	1.158	1.92	1.32	1.37	2.35	2.97	2.42	1.005	1.158	1.92	1.32	1.35	2.00	1.95	1.95
rand-min	1.005	1.062	1.70	1.48	1.38	2.11	1.92	2.05	1.005	1.056	1.58	1.32	1.38	2.08	1.95	2.00
rand-med	1.029	1.197	2.19	2.03	2.17	2.46	2.49	2.42	1.016	1.197	2.19	1.96	2.17	2.45	2.39	2.37
rand-max	1.161	1.468	3.48	2.92	2.75	3.10	3.23	2.99	1.161	1.468	3.48	2.92	2.75	3.03	3.46	2.86

5.2 Approximation Algorithms

Besides reporting the optimal co-scheduling results, Table 2 also lists the performance of the approximated schedules (outside the boxes.) On real jobs, the matching-based approximation produces near optimal results, the A*-cluster algorithm works similarly well except in the case of "no rescheduling" on "2-smt" architecture where the makespan is about 14% larger than the minimum. Because of the imprecision caused by clustering, both approximation algorithms significantly outperform the greedy and random scheduling in most real and synthetic cases. On the other hand, their distances from the optimal reflect the room for improvement.

Table 4 presents the results on 16 jobs. It does not include the brute-force and A* results because the former takes too much time (up to years with job migrations) to finish and the latter requires too much memory to run. The results of the approximation algorithms are consistent with the 8-job results. Although the minimum makespans from the random schedules occasionally get close to the results of the approximation algorithms, most random scheduling results are significantly worse than the matching-based and A*-cluster-based approximations. The greedy algorithm, although performing not as well as the other two approximation algorithms, outperforms the median results from random scheduling considerably.

Tables 3 and 5 report that the approximation algorithms take less than one second, in contrast to the up to years of time the brute-force search needs.

Table 5. The numbers of nodes visited and the time spent by different co-scheduling algorithms on 16 jobs on Intel Xeon 5080

	no resch.		resch.	
	nodes	time (ms)	nodes	time (ms)
matching	1	86	8	889
A*-clstr	76	39	122	94
greedy	1	3	8	9
random	-	1	-	2

Scalability: Figure 5 shows the scheduling overhead of the three approximation algorithms on a spectrum of problem sizes (with migrations and $u = 2$.) The greedy algorithm shows the best efficiency for its simplicity; the local matching-based algorithm shows less scheduling time than the A*-cluster algorithm.

Short Summary

We summarize the experimental results as follows:

1. The experiments empirically verify the optimality of the scheduling results of the perfect matching and the A*-search algorithm. The two algorithms are orders of magnitude more efficient than the brute-force search. They are applicable when the size of the problem is not large.
2. The local matching-based approximation algorithm is preferable when $u = 2$ (with or without job migrations) for its high scheduling quality and little scheduling time.

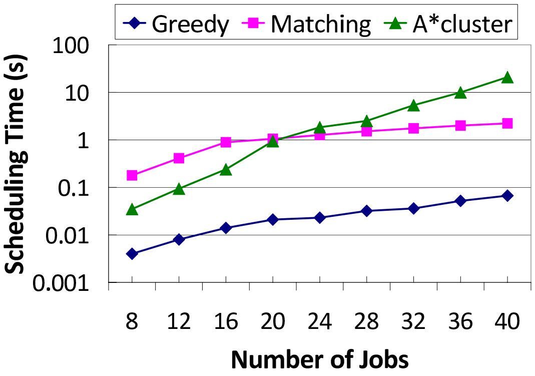

Fig. 5. Scalability of approximation algorithms

3. When $u > 2$, the A*-cluster algorithm offers a reasonable solution that produces good schedules in a moderate amount of time.
4. When scheduling overhead is the main concern, the greedy algorithm may be favorable, which uses slightly more time than random scheduling but offers consistently better results.

6 Discussion

This section discusses some limitations of this work and the influence on practical uses.

The requirement of all co-run degradations may seem to be an obstacle preventing the direct uses of the proposed algorithms in practical co-scheduling systems. However, that requirement does not impair the main goals of this work.

This work is a limit study. The primary goal is to offer feasible ways to uncover the optimal solutions in job co-scheduling, rather than to develop another heuristics-based runtime co-scheduler. Besides offering theoretical insights into co-scheduling, this work enables better evaluation of co-scheduling systems than before, offering the facility for efficiently revealing the potential of a practical co-scheduler in a general setting, which has been infeasible in the past for even small problems.

Furthermore, the algorithms proposed in this work may provide the insights for the development of more effective online scheduling algorithms both in operating systems and during the runtime of parallel applications. There has been some advancement in predicting co-run performance from program single runs (e.g., [2, 4]). The research in locality analysis has continuously enhanced the efficiency and accuracy in locality characterization [3, 16]. These studies make it possible to obtain co-run performance through lightweight prediction, hence offering the opportunity for the integration of the proposed scheduling algorithms in runtime scheduling systems.

In our experiments, we conduct a measurement of the migration overhead in the experimental machines by leveraging the system call, “sched_setaffinity”. The system call binds processes to cores. By invoking the call at some random locations in an application with different parameters, we can migrate the process among chips in a machine. (Migrations among machines are typically too expensive to support.) The results show that a migration may cause 0.1–1.1% changes to the execution times of the benchmarks used in our experiments, depending on the length of the original execution. Recall that in our experiments, we use the average value of migration overhead as the cost of a migration. The introduced inaccuracy is hence less than 1.1% of the computed minimum makespan.

7 Related Work

To the best of our knowledge, this work is the first systematic study on finding the *optimal* schedules that minimize the *makespan* of jobs running on CMP systems.

7.1 Comparison with Cost Minimization

As mentioned earlier, there have been some studies in optimal co-scheduling for minimum cost [10,11,24]. We have mentioned the connections and differences between that co-scheduling problem and our makespan minimization problem throughout this paper. We here give a summary.

Connections The two problems do have some connections, mainly in two aspects. First, they have similar dimensions to explore: dual-core or more than two cores per chip, migration allowed or not. Second, they both model the dual-core problems with a fully connected graph, and model the general cases as a search problem and derive solutions based on A*-search.

Differences However, these two problems are fundamentally different. Their different co-scheduling goals determine that important differences exist in almost every aspect of the explorations to the two problems.

- *Complexity Analysis.* Despite that the two problems are both proved to be NP-complete in a general setting, their proofs are substantially different. The previous work [10, 11] analyzes the computation complexity of cost minimization by formulating the problem as an Multidimensional Assignment problem (MAP). But for makespan minimization, the MAP formulation cannot be applied because of the mismatch of the objectives. We have to analyze and formulate the problem in a different way, proving the NP-Completeness through the reduction from the problem of Exact Cover by 3-Sets.
- *Algorithms on Dual-Core Systems without Migrations.* For algorithm design, the classic Blossom [6] algorithm can be directly used for finding the optimal schedule for dual-core cases for cost minimization [10, 11]. But it cannot be applied to makespan minimization because the algorithm aims to minimizing the total weights of a perfect matching on a graph, rather than the largest weight as what makespan minimization requires. The solution introduced in this work (Section 3.2) turns out to be even more efficient than the Blossom algorithm, with complexity of $O(n^{2.5} \cdot \log n)$ versus $O(n^4)$.

- *Algorithms in Other Settings.* A* is a classical search algorithm widely used in many areas. We do not claim the use of it for job co-scheduling in the general settings as a contribution of this work. In fact, previous work [24] has used it for approximating optimal schedules for cost minimization. However, the key in applying A* is in the formulation of the search problem and the design of the approximation functions $f(v)$ used in every search-tree node. Both are specific to the problem to be addressed. They are where the extensions are made by the approximation algorithms in this work. In addition, the empirical exploration of the tradeoff between the approximation efficiency and the quality of resulting schedules is also specific to the makespan minimization problem.
- *Scheduling Results.* As Section 5.1 shows, the optimal schedules for the two problems typically differ from each other, confirming the need for studies on each of them.

Overall, the previous work on cost minimization [10, 11, 24] has given some insights to this work. But because of the different goals of the two problems, this systematic exploration is imperative for achieving a good understanding of the makespan minimization problem.

7.2 Comparisons with Other Scheduling Work

Scheduling is a topic with a large body of relevant work. As summarized in the *Handbook of Scheduling* [12], 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 on how other jobs are assigned, a clear contrast to the performance interplay in the co-scheduling problem tackled in this current work.

Traditional Symmetric Multiprocessing (SMP) systems or NUMA platforms have certain off-chip resource sharing (e.g., on the main memory), but 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 [12] has considered dependences among jobs. But the dependences differ from the performance interplay in co-scheduling in that the dependences constrains the order rather than performance of the execution of the jobs.

The hierarchical scheduling algorithm [5] in traditional job scheduling also uses a tree-like hierarchy for job scheduling. However, it is about how to move tasks among queues along a path to feed an idle processor, considering no performance influence caused by co-running jobs.

7.3 Other Work on Shared Cache Management

Due to the importance of shared cache, recent years have seen a large number of relevant studies. Some of them try to construct practical on-line job scheduling systems. They employ different program features, including estimated cache miss ratios, hardware performance counters, and so on [7, 19, 20]. Architecture designs for alleviating cache contention have focused on cache partitioning [13], cache quota management [14], and so forth. But none of them has focused on the optimal co-scheduling to minimize makespan. In addition, some studies [2, 4] 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.

8 Conclusion

As the processor-level parallelism increases, the urgency for alleviating the resource contention among co-running jobs is continuously growing. This work concentrates on the theoretical analysis and the design of optimal co-scheduling algorithms for minimizing the makespan of co-running jobs. It proves the computational complexity of the problem, proposes an $O(n^{2.5} \cdot \log n)$ algorithm and A*-search-based algorithms to solve the makespan minimization problem, and empirically verifies the optimality of the algorithms and examines the effectiveness and scalability of several approximation algorithms. The analysis and algorithms contributed in this paper may complement previous explorations by both revealing the lower bound for evaluation, and offering insights in the development of lightweight co-scheduling systems.

Acknowledgment

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 Grant (0811791), NSF CAREER and DOE Early Career Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

1. The linux kernel archives. http://www.kernel.org.
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.
3. G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
4. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, pages 340–351, 2005.
5. S. Dandamudi. *Hierarchical Scheduling in Parallel and Cluster Systems*. Kluwer, 2003.
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. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, pages 25–38, 2007.
8. D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.
9. 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 *PACT*, 2006.
10. Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, pages 220–229, October 2008.

11. Y. Jiang, X. Shen, J. Chen, and R. Tripathi. The complexity and approximation of optimal job co-scheduling on chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems.*, 22(7), 2011. (DOI: 10.1109/TPDS.2010.193).
12. J. Y.-T. Leung. *Handbook of Scheduling*. Chapman & Hall / CRC, 2004.
13. M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Micro*, pages 423–432, 2006.
14. N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, pages 2–12, 2006.
15. S. Russell and P. Norvig. *Artificial Intelligence*. Prentice Hall, 2002.
16. X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, 2007.
17. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, pages 165–176, 2004.
18. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, 2002.
19. A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, pages 66–76, 2000.
20. A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, pages 66–76, 2002.
21. SPLASH. Stanford parallel applications for shared memory (SPLASH) benchmark. Available at http://www-flash.stanford.edu/SPLASH/.
22. G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, pages 117–128, 2002.
23. D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.
24. 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*, pages 41–50, 2009.