A Study on Optimally Co-scheduling Jobs of Different Lengths on Chip Multiprocessors*

Kai Tian
Computer Science
College of William and Mary
ktian@cs.wm.edu

Yunlian Jiang
Computer Science
College of William and Mary
jiang@cs.wm.edu

Xipeng Shen
Computer Science
College of William and Mary
xshen@cs.wm.edu

ABSTRACT

Cache sharing in Chip Multiprocessors brings cache contention among corunning processes, which often causes considerable degradation of program performance and system fairness. Recent studies have seen the effectiveness of job co-scheduling in alleviating the contention. But finding optimal schedules is challenging. Previous explorations tackle the problem under highly constrained settings. In this work, we show that relaxing those constraints, particularly the assumptions on job lengths and reschedulings, increases the complexity of the problem significantly. Subsequently, we propose a series of algorithms to compute or approximate the optimal schedules in the more general setting.

Specifically, we propose an A*-based approach to accelerating the search for optimal schedules by as much as several orders of magnitude. For large problems, we design and evaluate two approximation algorithms, A*-cluster and local-matching algorithms, to effectively approximate the optimal schedules with good accuracy and high scalability. This study contributes better understanding to the optimal co-scheduling problem, facilitates the evaluation of co-scheduling systems, and offers insights and techniques for the development of practical co-scheduling algorithms.

Categories and Subject Descriptors

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

General Terms

Algorithms,Performance,Experimentation

*This material is based upon work supported by the National Science Foundation under Grant No. CSR-0720499 and CCF-0811791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

CF'09, May 18–20, 2009, Ischia, Italy.

Keywords

co-scheduling, CMP scheduling, cache contention, A*-search, perfect matching

1. INTRODUCTION

With the popular adoption of Chip Multiprocessors (CMP) and Simultaneous Multithreading (SMT) in modern processors, it is typical for multiple computing units to share a single cache and other resources. The sharing, although helpful for the reduction of inter-thread latency, often causes resource contention among *co-runners* (referring to the processes or threads running together with some on-chip resources shared with one another), resulting in considerable degradation in program performance and system fairness [4, 8, 9, 11, 19, 26]. As processor-level parallelism grows continuously, the problem becomes increasingly important.

In recent studies, researchers have demonstrated the promise of job co-scheduling in alleviating the problem of contention. Job co-scheduling capitalizes on the differences among jobs. By cleverly assigning compatible jobs to cores (or hyperthreads), researchers have seen considerable improvement in computing efficiency [8, 23, 28], fairness [24], and performance isolation [9].

However, most previous studies have concentrated on the empirical aspect of the problem: How to conduct runtime explorations to heuristically infer the goodness of different co-run schedules and select a good one. Although such explorations are critical for producing practical co-scheduling systems, they are not enough for the assessment of the potential of co-scheduling—that is, the benefits an optimal schedule can bring.

Finding optimal schedules is important for two reasons. First, it enables a more thorough evaluation of co-scheduling systems than before. The baseline of most evaluations of current co-scheduling systems is just random schedulers. But in the design of a practical co-scheduling system, it is important to know the room left for improvement—that is, the distance from the optimal solution—to determine the efforts needed for further enhancement and the tradeoff between scheduling efficiency and quality. Without finding an optimal co-schedule, such an assessment is impossible to do. Second, although optimal co-scheduling algorithms may not be efficient enough for direct deployment in production uses, they can help understand the computational complexity of the co-scheduling problem, and offer insights to the enhancement of practical co-scheduling systems.

Unfortunately, the current exploration of optimal co-scheduling is still preliminary. A recent study [13] has

started to tackle this problem, but in a highly constrained setting, where all jobs are of the same length and no reschedulings are allowed. The two conditions make the problem tractable, but meanwhile, cause a significant departure from real scenarios.

The *goal* of this work is to uncover the complexity of, and develop scalable solutions to, the optimal co-scheduling problem in a more general setting, where jobs may have different lengths and can be rescheduled multiple times. The complexity of the problem increases significantly as the schedule space in this setting is exponentially larger than that in the special case. The schedules produced by the previously developed optimal co-scheduling algorithms [13] lose the optimality in this more general setting (showed in Section 5.1.)

This paper presents our two-fold attack to the optimal co-scheduling problem. First, we formulate optimal co-scheduling as a tree-search problem and develop an A*-search-based algorithm to determine optimal co-schedules for small problems, with orders of magnitude less overhead than a brute-force approach. A* search [20] is a technique stemming from artificial intelligence. Its appeals include the guarantee of the optimality of both the search result and the search efficiency. One of its key components is a cost estimation function, which determines the effectiveness of the algorithm in pruning the search space. In this work, we formulate the cost function as a linear programming problem, which helps the co-scheduling algorithm outperform a brute-force approach by 5 orders of magnitude in efficiency, with an exactly optimal solution produced. We are not aware of any previous A*-search-based algorithms for job co-scheduling.

Second, we develop two approximation algorithms and examine their scalabilities and effectiveness in co-scheduling large problems. The A*-search-based algorithm mentioned in the previous paragraph is hard to be applied to large problems due to memory requirement. But with the insights shed from it, we design two approximation algorithms, namely A*-cluster and local-matching algorithms. The A*-cluster algorithm integrates online adaptive clustering into the A*-search-based approach to trade accuracy for scalability. The local-matching algorithm, driven by local optimum, applies graph theory at every scheduling time to find the schedule that best fits the remaining jobs without future rescheduling considered. Experiments on Intel quad-core and hyperthreading machines show that both algorithms produce near-optimal results, significantly outperforming random schedules. The local-matching algorithm, in particular, consistently outperforms the state-of-the-art co-scheduling algorithms [13], and meanwhile, shows excellent scalability.

The two-fold explorations in this work contribute better understandings to the optimal co-scheduling problem. The proposed A*-search-based algorithm and the local-matching algorithm offer useful tools for computing (or approximating) optimal co-schedules. They not only enable the assessment of the potential of practical co-scheduling algorithms, but also shed insights to the development of future practical co-schedulers on CMP and SMT systems.

In the rest of this paper, Section 2 defines the problem setting. Section 3 analyzes the complexity of the optimal co-scheduling problem and presents the A* search based approach. Section 4 presents two approximation algorithms for solving large problems. Section 5 reports the experimental results. Section 6 discusses the limitations of this work and future extensions. Section 7 reviews the related work, followed by a short summary.

2. PROBLEM SETTING AND NOTATIONS

2.1 Concept of Co-Run Degradation

On a CMP system, a process often runs slower when there are other corunning processes that compete with it for shared resources, such as on-chip cache or bus. This degradation is called *co-run degradation*, defined as the difference between its corun time and its single-run time, denoted as $(cT - sT)$ (c for corun, s for single-run.) The *co-run degradation rate* is defined as

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

where $cCPI_i$ and $sCPI_i$ are the numbers of cycles per instruction (CPI) when job i has or has no cache sharers (i.e., jobs co-running on the same cache) respectively; c stands for co-run and s stands for single-run.

In optimal co-scheduling, the degradation of every possible co-run and the length of the single run of each job are known beforehand. They may be obtained by offline profiling runs. Recall that the direct goal of optimal co-scheduling is to find the upper limit of co-scheduling for facilitating the evaluation of practical co-scheduling systems, rather than to produce a co-scheduler directly applicable in real runtime systems. Therefore, offline profiling—adopted in this work—is usually affordable. (But the trial of all possible co-schedules is typically infeasible because of the exponential increase of the co-scheduling space.)

It is worth noting that many recent studies have attempted to approximate single-run and co-run information through either runtime analysis [9, 14, 18, 23] or statistical models [4, 25]. Combined with those techniques, efficient optimal co-scheduling algorithms may show the promise of practical applicability (even though practical applicability is not the main goal of this research.) Detailed studies are out of the scope of this paper.

2.2 Optimal Co-Scheduling Problem Tackled in this Work

The problem this work attempts to solve is defined as follows. There are **N** processes to be assigned to M cores, and $N = M$[1], one process per core at each time. Every **K** (a factor of N) cores reside on a chip, sharing a cache. Let **I** be the number of chips, so $I = N/K$. The processes may finish at different times, therefore, reassignments may be necessary. There are various goal functions in job co-scheduling. In this work, the goal is to find a schedule that minimizes the total completion time of all jobs [2], as 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.

[1]The techniques in this paper are applicable to the cases when $N < M$ as well; one can simply consider that there are $M - N$ processes that consume no resource at all.

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

To avoid distractions from other complexities, we make the following assumptions. First, the performance of the processes on a chip is independent on how the other jobs are assigned on other chips. Second, all processes start at the same time running till their finish and there are no phase changes in a process. Third, as there are no phase changes, we assume that rescheduling occurs only when one process terminates, and process migration has negligible overhead. These assumptions keep the fundamental challenges of the general co-scheduling problem (Section 6 discusses them further.) The techniques developed under these assumptions lay the necessary foundations for the more general cases.

In the following description, we call each scheduling or rescheduling point as *a scheduling stage*. So, N jobs have at most N scheduling stages. If all degradations are non-negative (which is typical), the scheduling can stop at stage $N - I$ because the remaining I jobs can run alone with no degradations. We use *an assignment* to refer 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 each other. We use *a schedule* for the set of all sub-schedules that are used from the start to the end of the executions of all processes. A schedule is a solution to the co-scheduling problem.

3. A*-SEARCH FOR CO-SCHEDULING

In this section, we first examine the co-schedule space, revealing the challenges in finding optimal co-schedules. Then we present an A*-search-based approach to pruning the space and finding optimal co-schedules efficiently.

3.1 Co-Schedule Space

The optimal co-scheduling problem tackled in this work can be formulated as a tree-search problem as Figure 1 illustrates. For N jobs, there are at most N scheduling stages; each corresponds to a time point when one job finishes since the last stage. Each 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 between two stages by all jobs. Let n_2 represent a child of node n_1. Given the state at n_1, we assign the remaining jobs according to the sub-schedule contained in n_2; let t be the time required for the first remaining job to finish; the cost on the edge from n_1 to n_2 is $t * m$, where m is the number of jobs that are alive during that period of time.

The goal of optimal co-scheduling is to find a path from the starting node to any leaf node so that the sum of the costs of all the edges on the path is minimum. As a comparison, the previously explored co-scheduling problem [13] contains only the starting node and the first stage in the tree (without rescheduling), the search space of which contains $n = \prod_{i=0}^{\frac{N}{K}-1} \binom{N-iK-1}{K-1}$ nodes. Whereas, the search space in the problem tackled in this work is exponentially larger with $O(n^N)$ nodes contained.

The significantly increased complexity makes previous approaches difficult to apply. As an example, Jiang et al. have used perfect matching on a fully connected graph to formulate the optimal co-scheduling [13]. That model, however, captures no influence from the difference among program lengths and rescheduling.

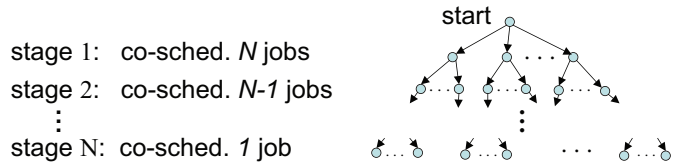


Figure 1: The search tree of optimal job co-scheduling with rescheduling allowed at the end of a job. Each node in the tree, except the starting node, represents a sub-schedule of the remaining jobs.

3.1.1 A* Search Algorithm

In this work, we treat the optimal co-scheduling problem as a tree-search problem, and apply A*-search algorithm to determine optimal co-schedules efficiently.

A* search is an appealing algorithm stemming from artificial intelligence [20]. It is designed for fast graph search. Under certain conditions, A* search guarantees the optimality of the solution, and meanwhile, effectively avoids the visit to some portion of the search space that are impossible to contain the optimal solutions. In fact, it has been proved that A* search is optimally efficient for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*, for a given heuristic function [20]. Its completeness, optimality, and optimal efficiency are the major reasons for us to choose A* search for the acceleration of the search for optimal schedules.

We use Figure 1 to explain the basic algorithm of A*. Each edge in the graph has a cost. The problem is to find the cheapest route in terms of the total cost from the starting node to the goal node. In A* search, each node, say node n, has two functions, denoted as $g(n)$ and $h(n)$. Function $g(n)$ is the cost to reach node n from the starting node. Function $h(n)$ is the estimated cost of the cheapest path from n to the goal. Therefore, the sum of $g(n)$ and $h(n)$, denoted as $f(n)$, is the estimated cost of the cheapest route that goes from the start to the goal and passes through n.

A* search has a priority list, which initially contains only the starting node. Each time, A* search removes the top—that is, the node with the highest priority—from the priority list, and expands that node. After an expansion, it computes the $f(n)$ values of all the newly generated nodes, and put them into the priority list. The priority is proportional to $1/f(n)$. The expansion continues until the top of the list is a goal node, which implies that its cost is no greater than the $f(n)$ of any other nodes in the list. The use of $f(n)$ is the key for A* search to prune the search space effectively. Figure 2 shows the algorithm of A* search.

A good definition of function $h(n)$ is critical for the solution's optimality and the algorithm's efficiency. There are two important properties of A*:

- A* is optimal if $h(n)$ is an admissible heuristic—that is, $h(n)$ must never overestimate the cost to reach the goal[3].
- The closer $h(n)$ is from the real lowest cost, the more effective A* search is in pruning the search space.

[3]We assume that the search scheme is a tree-search. There are some subtle complexities for other schemes of search [20].

```
/* start node contains all jobs to be scheduled */
Procedure Astar (){
  priorityList = { start };
  while priorityList.notEmpty() {
   x = priorityList.topItem();
   if (x.unfinishedJobs == 0)
    return x; // the solution route can be
              // traced from x
   priorityList.remove(x);
   while (y = x.nextSubSchedule())!= null {
    y.g_score = x.g_score + totalTime_between(x,y);
    y.h_score = estimatedTime_to_finishAll_from(y);
    y.f_score = y.g_score + y.h_score;
    priorityList.insert(y);}}
}
```

Figure 2: A* search for optimal schedules.

Therefore, the key in applying A*-search is to develop a good definition of the function $h(n)$. Next, we describe how linear programming can be used to form a suitable definition of the function, and how A*-search is used in job co-scheduling.

3.1.2 A*-Search-Based Job Co-Scheduling Algorithm

To use A* search in job co-scheduling, the key problem is to define the two functions, $g(n)$ and $h(n)$. The definition of function $g(n)$ is simple, just the total cost from the start to node n.

The definition of $h(n)$ is more interesting. When all corun 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 as $h(n)$ must be no more than the real cost. But for efficiency, we need more sophisticated definition to make $h(n)$ closer to the real lowest cost.

Definition of $h(n)$ through Linear Programming.

A more sophisticated definition of $h(n)$ is through the form of linear programming. Suppose at node n there remain U unfinished jobs. We define $h(n) = T_s + T_d$. T_s is the time the U jobs need to finish their remaining parts if they each run alone. T_d is the estimated minimum of the total degradation that the U jobs have during their execution from node n to any child of n.

In this section, 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_d is clearly 0 (no degradations as each chip has only one or no jobs.) 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 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 $T_{min(n)}$ represent the minimum of the single-run times of the unfinished part of all the remaining U jobs. Notice that the time lasting from n to any of its children must be no less than $T_{min(n)}$ because of corun degradations. Let $r_{b_{min}}$ be the minimum of the degradation rates of all jobs when a job coruns with $b-1$ other jobs. It is clear that the degradation on the chip must be no less than $b * r_{b_{min}} * T_{min(n)}$, 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

Definitions: (details at Page 3 bottom section)
U: number of unfinished jobs,
I: number of chips,
K: cores per chip, $I < U \leq I * K$,
$r_{x_{min}}$: minimum degradation rate,
$T_{min(n)}$: 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:

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

Linear constraint:

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

Figure 3: Integer linear programming for computing T_d, the lower bound of degradation.

$d_j = \sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(n)}$; b_i refers to the number of jobs assigned to chip i in the sub-schedule.

The value of T_d should be the minimum of d_j of all sub-schedules of node n. To determine the sub-schedule that has the smallest d_j, we need to find the values of b_i so that $\sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(n)}$ is minimized under the constraint that $\sum b_i = U$. This analysis leads to the integer linear programming shown in Figure 3. 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:

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

The rationale behind 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 2 jobs assigned, which is not allowed in the problem setting (Section 2.) The application of the definition of T_d to such a sub-schedule leads to Equation 1.

With the definitions of the two functions, $g(n)$ and $h(n)$, we implement the A*-search-based optimal co-scheduling algorithm. Experiments, reported in Section 5, verify the optimality of the schedules produced by the algorithm.

4. APPROXIMATION OF OPTIMAL SCHEDULES FOR LARGE PROBLEMS

One typical drawback of A*-search-based algorithms is the high requirement of memory space. The reason is that the algorithm requires to keep all open nodes (i.e. the nodes that still have unexpanded children nodes) into the priority list. As the problem size increases, the number of open nodes may become too large to fit into the memory.

In this section, we describe two approximation algorithms for solving the optimal co-scheduling problem in a scalable manner. One algorithm, the A*-cluster algorithm, comes

from the insights shed by the A*-search-based algorithm; the other algorithm, the local-matching algorithm, is a generalized version of graph-matching-based co-scheduling algorithms.

4.1 A*-Cluster Algorithm

The first algorithm, A*-cluster, combines A* algorithm 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.

One option to cluster jobs 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.

The integration of clustering into A* search is implemented inside procedure *nextSubSchedule()* (invoked in the middle of procedure *Astar()*) as shown in Figure 4. The A* algorithm uses this procedure to generate a child of the current node in the search tree. Suppose the current node is not the starting node. At the first invocation of procedure *nextSubSchedule()* by this node, the procedure 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 clustering results, during the generation of children nodes, each time the procedure *nextVeryNewSubSchedule* returns a sub-schedule that is 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 one if they are not equivalent when we regard all jobs in a cluster as the same. As an example, suppose 4 jobs fall into 2 clusters as {{1 2}, {3 4}}. The sub-schedule (1 3) (2 4) is regarded as equivalent to (1 4) (2 3), but different from (1 2) (3 4) (each pair of the parentheses contains a corun group.) Finding those novel sub-schedules only needs to solve a first-order linear equation system. The unknowns are the numbers of instances of different mixing patterns of clusters; they must be non-negative. 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 each mixing pattern, on the right side is the total number of jobs belonging to that cluster. Each solution of the equation system corresponds to one novel sub-schedule. Details are skipped for lack of space.

The two reduction strategies decrease the search space significantly. The first strategy reduces the height of the search tree, while the second strategy reduces the width. They reduce the number of nodes at a stage from factorial, $\prod_{i=0}^{\frac{N}{K}-1} \binom{N-iK-1}{K-1}$, to polynomial, $O(n^\gamma)$ ($\gamma = C + (C^K - C)/K!$), for given C and K values (C is the number of clusters.)

The clustering in our implementation is based on the time needed for a job to finish under the current sub-schedule. (For the starting node, we use single-run times.) The times are estimated by the single run time and corun degradation rates of each job. Machine learning researchers have proposed many clustering techniques, such as K-means, hierarchical clustering [10]. In our problem, the data are one dimensional and we do not know the number of clusters beforehand. So, we use a simple distance-based clustering approach. Given a sequence of data, we first sort the data in an ascending order. Then, we compute the differences between every two adjacent data items in the sorted sequence. Large differences are considered as indiction to cluster boundaries. A difference is regarded as 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.

```
/* Jobs contains unfinished jobs; */
/* isFirstInvoke is 1 initially.  */
Procedure nextSubSchedule() {
  if (isFirstInvoke) {
    foreach job in jobs
      estimate_timeToFinish(job);
    if (this! = start) {
      C1 = getEarliestCluster(Jobs);

      /* update to the state when C1 finishes */
      Jobs = Jobs - C1;
      update_timeToFinish(Jobs);
    }
    Cs = ReCluster(Jobs);
    isFirstInvoke=0;
  }
  /* get a substantially new sub-schedule */
  nextVeryNewSubSchedule(Jobs, Cs);
}
```

Figure 4: Integration of clustering into A* search algorithm for approximation of optimal co-scheduling.

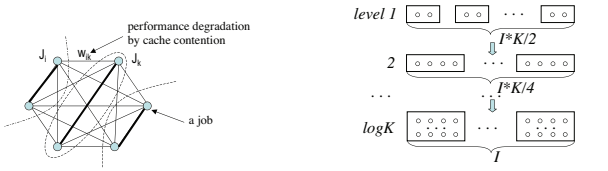
The time complexity of the clustering algorithm is $O(J)$, where J is the number of remaining jobs.

4.2 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 1. 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 assumption leads to local optimum at each scheduling stage.

The key component of the algorithm is the procedure to compute the local optimum. The problem is NP-complete when a chip has more than two cores. Previous work has demonstrated the effectiveness of hierarchical perfect matching in approximating the optimal schedule when the number of jobs equals the number of cores and all jobs are of the same length [13]. In this current work, we relax those two conditions to make the algorithm more generally applicable.

Our explanation of the algorithm starts with a simple case, to co-schedule N jobs on a system with I dual-core chips and $N = I * 2$ (the number of jobs equals the number of cores.) On such a system, the local optimal co-scheduling problem can be modeled in a degradation graph, illustrated in Figure 5 (a). Every vertex represents a job. Unlike the previous exploration [13], in this work, the weight on each edge equals the sum of the times required for the remaining parts of the two jobs represented by the two vertices to



(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.

(b) Hierarchical perfect matching algorithm. Each box represents a virtual chip except the chips at the bottom, which are real chips. Each circle represents a core; K is the number of cores per real chip.

Figure 5: Illustration of perfect-matching-based co-scheduling algorithms.

finish if they corun on a chip. The optimal co-scheduling thus 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. Clearly, a valid job schedule must be a perfect matching in the graph, and a minimum-weight perfect matching must minimize the total time for remaining jobs to finish. Hence a minimum-weight perfect matching must correspond to an optimal schedule of the remaining jobs when there is no rescheduling on those jobs.

The minimum-weight perfect matching problem can be solved by the polynomial-time *blossom* algorithm proposed by Edmonds [7]. In job co-scheduling context, the time complexity of the algorithm is $O(N^4)$ (N is the number of jobs.)

On systems with more than two cores per chip, the optimal co-scheduling becomes NP-complete [13]. Hierarchical perfect matching approximates the optimal schedule by applying minimum-weight perfect matching iteratively. Figure 5 (b) illustrates the basic idea. Given I K-core chips (still assuming $N = I * K$), the algorithm first treats each chip as $K/2$ dual-core virtual chips, and the on-chip cache is evenly partitioned into $K/2$ portions with every dual-core sharing one portion. By applying the minimum-weight perfect matching algorithm, we can find the assignment of jobs to those virtual chips, which partitions the job set into $N/2$ pairs. Next, by treating one pair as one scheduling unit, in the similar manner, we couple the pairs into $N/4$ pairs of pairs, or in another word, $N/4$ 4-member job groups. This process continues until the size of job group becomes K, when the schedule finishes. Each $K-$member group is one assignment in a final sub-schedule.

The hierarchical matching algorithm, although invoking the minimum-weight perfect matching algorithm $logK$ times, has the same time complexity as $O(N^4)$, thanks to that the number of vertices in the degradation graphs decreases exponentially.

In the above description, we assume that the number of remaining jobs equals the number of cores in the computing system. This assumption is needed for perfect matching algorithm to work. However, it does not hold in this work when some jobs complete early. We take a simple strategy to handle this case: treat 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 by keeping the number of jobs always equal to the number of cores.

It is easy to see that the time complexity of the local-matching algorithm, in both dual-core and larger systems, is $O(N^5)$: The co-scheduling algorithm on a stage has complexity of $O(N^4)$, and there are N stages.

Application in SMT Co-Scheduling.

Although the description of the three co-scheduling algorithms assumes that the co-scheduling is on different cores in a CMP system, all those algorithms are applicable to SMT co-scheduling as well. The only change is that the co-run degradations are of jobs that run together on multiple hyperthreads that reside on the same processing unit, rather than on multiple cores on the same chip. The next section reports the experimental results on both CMP and SMT co-scheduling.

5. EVALUATION

In this section, we first present the evaluation of the scheduling algorithms on a mix of 14 parallel and sequential programs, and then show a study of their scalability. We use 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 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 14 test programs consist of 2 parallel programs from SPLASH-2 [2] and 12 programs randomly selected from SPEC CPU2000. As we use two threads for each of the two parallel programs, we have 16 jobs in total. We did 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 (two threads per program) to examine the applicability of the co-scheduling algorithms for parallel (in addition to sequential) applications. Table 1 lists the programs with their corun degradation ranges on the Intel Xeon 5150 processors. The big ranges of degradations suggest the potential for co-scheduling.

In the collection of co-run degradation rates, we follow Tuck and Tullsen's practice [27], 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 exponentially growing co-scheduling space makes it infeasible to determine the optimal schedule for even 16 jobs through exhaustive search. So, we first use 8 jobs to reveal the detailed comparisons among the co-scheduling algorithms, verifying the optimality of the solution provided by the A*-search-based algorithm. We then use all the 16 jobs to examine the performance and scalability of the two approximation algorithms.

Table 1: Benchmarks

Benchmark	single-run time (s)	corun degrad rate 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.

5.1 Comparison to the Optimal

This experiment runs on Intel Xeon 5150 processors. We use the top 6 programs (8 jobs as *fmm* and *ocean* have two threads each) in Table 1 to compare the performance of 6 different scheduling algorithms: brute-force, A*, A*-cluster, local-matching, no-resch, and random schedulers. The *brute-force scheduler* conducts an exhaustive search of the entire schedule space to find the best schedules. The *no-resch scheduler* implements the optimal co-scheduler proposed in the previous work [13], which considers no job-length differences or possibilities of job rescheduling. The *random scheduler* schedules jobs in a random manner, corresponding to the default schedulers in most existing systems, which are oblivious to on-chip resource sharing. We obtain the random scheduling results by conducting random scheduling for 100 times and picking the one with median performance.

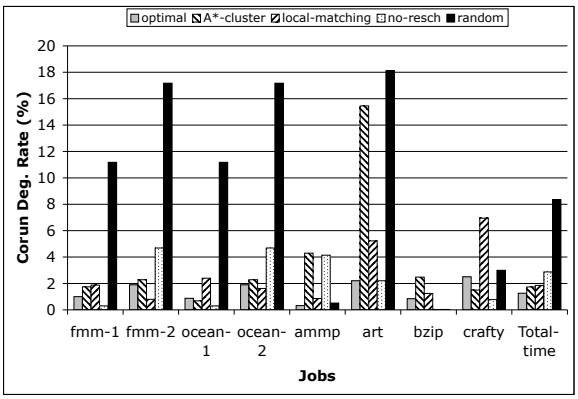


Figure 6: Performance degradation rates of 8 jobs co-running on quad-core Intel Xeon 5150 processors.

Table 2: Comparison of co-scheduling algorithms on 8 jobs on quad-core Intel Xeon 5150 processors

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
A*-cluster	11	0.008	80.6	1.7
local-matching	4	0.06	80.7	1.8
no-resch	1	0.02	81.5	2.9
random	-	-	85.9–89.2	8.4–12.5

The results verify the optimality of the scheduling results from the A* scheduler. It produces the same schedule as the brute-force search scheduler does. Figure 6 shows the corun degradation rates of the 8 jobs in different schedules. The "optimal" bars represent the results of the brute-force search and the A* scheduler. The random scheduler causes 8.4% degradation to the total running time. The schedule by the no-resch scheduler is 2.9% worse than the optimal, confirming that the scheduling algorithm, although able to produce optimal schedules for the previously explored special setting, cannot guarantee the optimality in this more general scenario. The two approximation algorithms, A*-cluster and local-matching algorithms, both achieve close-to-optimal results, only 0.4–0.5% away from the optimal performance.

It is important to notice that the optimal schedule is a schedule that minimizes the total running time, but not the running time of each individual program. Therefore, it is normal to see that the optimal schedule causes larger degradation to some programs (e.g., *crafty*) than other schedulers do in Figure 6. By degrading the performance of some programs a little more, the optimal scheduler succeeds in decreasing the degradations of other more significant programs, and hence achieve the overall optimum.

Table 2 compares the schedulers in other aspects. The A* scheduler finds the optimal schedule 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. The two approximation algorithms use even less time for scheduling. The right-most two columns report the total running times and corun degradation rates of the 8 jobs under those schedules. The random scheduling results include both the median and the worst performance of 100 random schedules to show the potential risks of current sharing-oblivious scheduling.

5.2 Results on 16 Jobs

For 16 jobs, the brute-force algorithm would take years. Our implementation of A* algorithm (in Java) is subject to memory shortage when scheduling more than 12 jobs. (A memory-bounded version [20] of the algorithm may help.) In this section, we concentrate on the evaluation of the two approximation algorithms on both multi-cores and hyper-threads.

5.2.1 Co-Scheduling on Multi-Cores

Figure 7 show the degradation rates on quad-core Intel Xeon 5150 processors; Table 3 reports the corresponding summary data. The random schedules cause 9.9% (up to 19.2%) degradation to the total running time. The no-resch

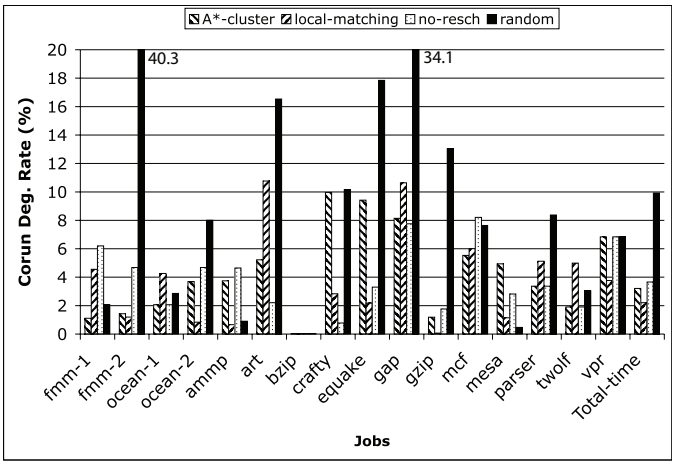


Figure 7: Performance degradation rates of 16 jobs co-running on quad-core Intel Xeon 5150 processors.

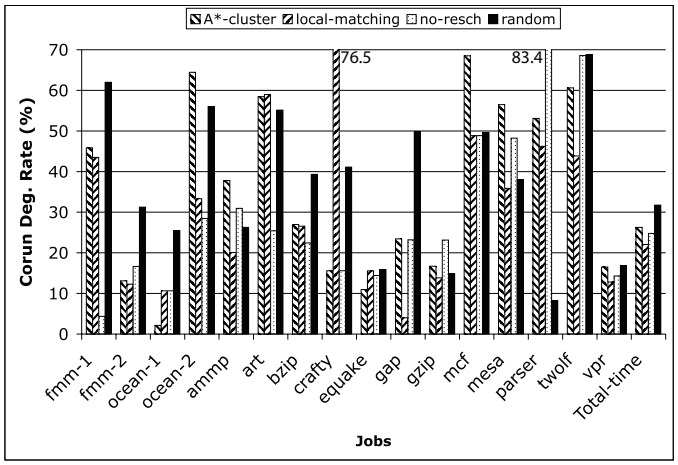


Figure 8: Performance degradation rates of 16 jobs co-running on the hyperthreads of Intel Xeon 5080 processors.

Table 3: Co-scheduling 16 jobs on quad-core Intel Xeon 5150 processors

algorithm	visited nodes	sched. time (s)	total exec time (s)	deg. rate (%)
A*-cluster	721	109	149	3.2
local-matching	8	0.63	147	2.2
no-resch	1	0.03	150	3.7
random	-	-	159–172	9.9–19.2

Table 4: Co-scheduling 16 jobs on hyperthreads of Intel Xeon 5080 processors

algorithm	visited nodes	sched. time (s)	total exec time (s)	deg. rate (%)
A*-cluster	315	198	325	26
local-matching	8	0.24	315	22
no-resch	1	0.03	322	25
random	-	-	340–382	32–48

algorithm reduces the degradation to 3.7%, while the A*-cluster and the local-matching algorithms further reduce the degradation to 3.2% and 2.2%. It is remarkable that the local-matching algorithm achieves the better result by taking less than 0.6% time of what the A*-cluster algorithm takes. This result indicates that even though the A*-cluster algorithm visits more nodes in the schedule space, the inaccuracy due to the clustering has caused considerable errors to the scheduling results.

5.2.2 Co-Scheduling Performance on Hyper-Threads

Figure 8 and Table 4 shows the experimental results when the 16 jobs run on the Intel Xeon 5080 processors with hyperthreads enabled. The schedule from A*-cluster reduces the median degradation rates of random schedules from 31.7% to 25.9%. The local-matching algorithm reduces the degradations to 22%, outperforming the no-resch algorithm by 2.8%.

Compared to the results in the multi-core experiments in Table 3, the degradation rates are clearly higher in this hyperthreading experiments because of the more extensive sharing of on-chip resource among jobs. The A*-cluster algorithm takes more time than in the multicore experiments, even though it visits fewer nodes; this is because of the difference in cluster sizes.

5.2.3 Co-Scheduling Scalability

We use 32 to 128 jobs to measure the running times of the two approximation algorithms ($K = 2$). The jobs are artificial jobs with random values as their single-run times and corun degradations. Figure 9 depicts the running times of the algorithms on the Intel Xeon 5150 processors. The local-matching algorithm shows much better scalability than the A*-cluster algorithm does: It takes only about 10 seconds to schedule 128 jobs, whereas, the A*-cluster algorithm needs more than 2000 seconds. The reason for the difference is that the number of paths A*-cluster needs to explore in the schedule tree increases as the number of jobs increases, while the local-matching algorithm always explore a single path. The time increase of local-matching algorithm is merely due to the increased computation for obtaining the best subschedule at each scheduling stage.

Short Summary.

We draw the following conclusions from all the experimental results:

- The A*-search-based algorithm effectively prunes search space. When the problem size is small, it can produce optimal schedules efficiently.
- The local-matching algorithm show consistently better results than other approximation algorithms. Together with its good scalability, this algorithm is a desirable choice for large co-scheduling problems.
- The previously proposed optimal co-scheduling algorithm loses the guarantee of the optimality of its scheduling results when job lengths are different and rescheduling is allowed. Even though it still produces good results, it is consistently outperformed by the local-matching algorithm.

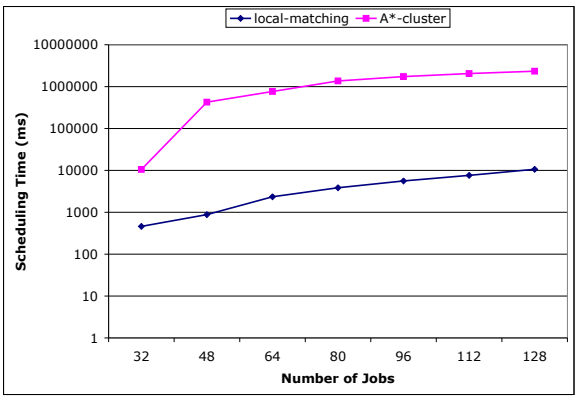


Figure 9: Scalability of the approximation algorithms.

- The combination of clustering with A*-search shows good scheduling results, but is not as scalable as the local-matching algorithm.

6. DISCUSSIONS

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, proactive co-scheduling (i.e., scheduling before executions) may directly benefit from the algorithms proposed in this work. There has been some work on predicting co-run performance from program single runs (e.g., [14]). Some recently proposed techniques in locality analysis has advanced the efficiency and accuracy in locality characterization [22,25]. Those studies make it possible to predict corun performance efficiently, and thus pave the way for proactive scheduling. The algorithms proposed in this work may serve as the base for proactive scheduling algorithms. They may also provide the insights for the development of more effective online (reactive) scheduling algorithms in both operating systems and the runtime of parallel applications.

Some assumptions made in this work are not difficult to resolve. For instance, if jobs start at different times or contain phase shifts, it may suffice to add rescheduling at those changing points in the scheduling algorithms. But, some other complexities, such as process migration overhead and rescheduling at any time, may need further explorations to resolve.

7. RELATED WORK

The closest work to this research is the analysis and approximation of optimal job co-scheduling from Jiang et al. [13]. The analysis and algorithms in that work is based on the assumptions of same job lengths and no rescheduling allowed. This research eliminates those assumptions and offers algorithms applicable to more general co-scheduling problems.

As Early as in 1960s, scheduling has been used for improving the usage of memory system. Denning proposed balance-set scheduling to improve virtual memory usage by grouping programs based on their working set size [5]. Recent studies employ the similar idea for CMP cache usage but use different program features, including estimated cache miss ratios [9], and hardware performance counters [8,28]. Kim et al. study cache partitioning for fairness in CMPs [14]. DeVuyst et al. exploit unbalanced thread scheduling for energy and performance [6]. Architecture designs for alleviating cache contention have focused on cache partitioning [12,18], cache quota management [19], cache policies [11], and heterogeneous design [15].

Cache sharing also exists in Simultaneous Multithreading (SMT) processors. Parekh et al. introduce thread-sensitive scheduling. They demonstrate that by greedily selecting jobs with the highest IPC can improve system throughput significantly [17]. Snavely et al. propose symbiotic scheduling [23,24]. It uses sample-optimization-symbios (SOS) scheme to try different combinations of jobs and pick the best one as the optimal schedule [23]. Their later work also considers process priorities in the symbiotic scheduling [24]. Settle et al. use an L2 cache activity vector to enhance the scheduler [21]. Some other work changes process affinity according to hardware performance counters [3,16]. The algorithms presented in this paper may benefit SMT co-scheduling as well if co-run performance is attainable.

To the best of our knowledge, this work is the first systematic study devoted to finding the *optimal* schedules for CMP systems *with rescheduling allowed*. We are not aware of any prior work that uses A* search to reduce shared-cache contention. The combination of A* and clustering and the local-matching algorithms for co-scheduling are also novel to the best of our knowledge.

8. CONCLUSION

This work explores the problem of optimally co-scheduling jobs of different lengths. We propose an A*-based approach to accelerating the search for optimal schedules by as much as orders of magnitude. For large problems, we design and evaluate two approximation algorithms, A*-cluster and local-matching algorithms, to effectively approximate the optimal schedules with good accuracy and scalability. Experiments on both multicores and hypterthreads demonstrate that when the number of jobs is small, the A*-search-based algorithm can find the optimal co-schedule efficiently. For large problems, the local-matching algorithm is a desirable choice. It achieves better co-scheduling results than state-of-the-art algorithms, meanwhile, showing remarkable scalability.

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

line co-scheduling and the runtime of parallel applications as well.

9. REFERENCES

[1] Gnu linear programming kit. Available at `http://www.gnu.org/software/glpk/glpk.html`.

[2] Stanford parallel applications for shared memory (splash) benchmark. Available at `http://www-flash.stanford.edu/SPLASH/`.

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

[4] 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.

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

[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.

[7] 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.

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

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

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

[11] 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.

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

[13] 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)*, 2008.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

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

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

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

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

[25] G.E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, 2001.

[26] G.E. 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.

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

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