Modelling and Developing Co-scheduling Strategies on Multicore Processors

Huanzhou Zhu[*], Ligang He[*,#], Bo Gao[*], Kenli Li[#], Jianhua Sun[#], Hao Chen[#], Keqin Li[#,$]
[*]. Department of Computer Science, University of Warwick, Coventry, CV4 7AL, United Kingdom
[#]. School of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, China
[$]. Department of Computer Science, State University of New York, New Paltz, New York 12561, USA
Email: {zhz44, liganghe, bogao}@dcs.warwick.ac.uk, lkl@hnu.edu.cn, jhsun, haochen@aimlab.org, lik@newpaltz.edu

***Abstract*—On-chip cache is often shared between processes that run concurrently on different cores of the same processor. Resource contention of this type causes performance degradation to the co-running processes. Contention-aware co-scheduling refers to the class of scheduling techniques to reduce the performance degradation. Most existing contention-aware co-schedulers only consider serial jobs. However, there often exist both parallel and serial jobs in computing systems. In this paper, the problem of co-scheduling a mix of serial and parallel jobs is modelled as an Integer Programming (IP) problem. Then the existing IP solver can be used to find the optimal co-scheduling solution that minimizes the performance degradation. However, we find that the IP-based method incurs high time overhead and can only be used to solve small-scale problems. Therefore, a graph-based method is also proposed in this paper to tackle this problem. We construct a co-scheduling graph to represent the co-scheduling problem and model the problem of finding the optimal co-scheduling solution as the problem of finding the shortest valid path in the co-scheduling graph. A heuristic A*-search algorithm (HA*) is then developed to find the near-optimal solutions efficiently. The extensive experiments have been conducted to verify the effectiveness and efficiency of the proposed methods. The experimental results show that compared with the IP-based method, HA* is able to find the near-optimal solutions with much less time.**

I. Introduction

Modern CPUs implement the multi-core architecture in order to increase their processing speed. Often, performance critical resources such as on-chip cache are not entirely dedicated to individual cores. This introduces resource contention between jobs running on different cores of a processor and consequently causes performance degradation (i.e., slows down the job execution) [21]. In order to reduce the impact of resource contention, many solutions have been proposed in recent years. In comparison to architecture-level [24], [28] and system-level solutions [23], [32], software-level solutions such as contention-aware co-schedulers [12], [16], [35] attract more researchers' attention because of its short development cycle. Results from these studies demonstrated that contention-aware co-schedulers can deliver better performance than conventional schedulers.

Existing studies of the co-scheduling problem can be classified into two categories. Researches in the first category aims at developing practical job scheduling systems that produce solutions on a best effort basis. Algorithms developed in this category are often heuristics-based in order to reduce computation cost. The work in the second category aims to develop the algorithms to either compute or approximate the optimal co-scheduling strategy (referred to as the *optimal co-scheduling problem* in the rest of this paper). Due to the NP-hard nature [19] of this class of problems, obtaining an optimal solution is often a computation-expensive process and is typically performed offline. Although an optimal solution is not suitable for direct uses in online job scheduling systems, its solution provides the engineer with an unique insight into how much performance can be extracted if the system were best tuned. Additionally, knowing the gap between current and optimal performance can help the scheduler designers to weight the trade-offs between efficiency and quality.

There are some research studies in contention-aware co-scheduling [13][17]. To the best of our knowledge, the existing methods that aim to find the optimal co-scheduling only consider the serial jobs [19]. However, there typically exist both serial and parallel jobs in the computing systems, such as Clusters and Clouds [31], [15], [27]. As shown in [19], when a set of serial jobs are scheduled to multi-core computers (with each job being scheduled to a core), the objective is to minimize the sum of the performance degradation of all serial jobs. However, this is not the case for parallel jobs.

We use Figure 1 to illustrate why different considerations

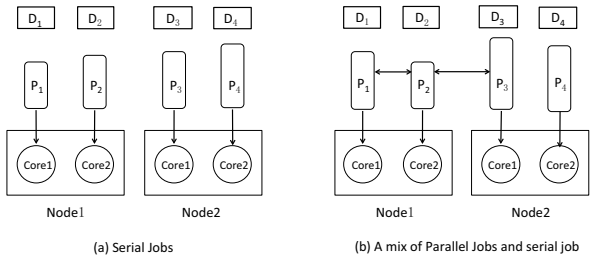


Fig. 1: An illustration example for the difference between serial and parallel jobs in calculating the performance degradation

∗ Dr. Ligang He is the correspondence author

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.31

should be taken when co-scheduling parallel jobs. The figure considers two co-scheduling scenarios. In Fig.1a, 4 serial jobs (i.e., 4 processes p_1, ..., p_4) are co-scheduled on two dual-core nodes, while in Fig.1b a serial job (p_4) and a parallel job with 3 processes (i.e., p_1, p_2 and p_3) are co-scheduled. D_i drawn above p_i is the degradation of p_i in the co-scheduling solution. The arrows between p_1 and p_2 as well as between p_2 and p_3 represent the interactions between the parallel processes. In Fig. 1a, the objective is to minimize the sum of the performance degradation suffered by each process (i.e.,$D_1 + D_2 + D_3 + D_4$). In Fig.1b, the performance degradation (i.e., increased execution time) of the parallel job is dependent on the processes that has been affected the most and therefore completing the execution last. Therefore, the performance degradation of the parallel job should be computed by $max(D_1, D_2, D_3)$. The objective in Fig.1b is to find the co-scheduling solution that minimizes $max(D_1, D_2, D_3) + D_4$. In this paper, we propose the contention-aware co-scheduling algorithms that recognize this distinction.

In this paper, we developed two methods, an Integer Programming-based (IP-based) and a graph-based method, to find the optimal co-scheduling solution for a mix of serial and parallel jobs. In the IP-based method, the co-scheduling problem is modelled as an IP problem and then existing IP solvers can be used to find the optimal solution. We find that the IP-based method incurs high time overhead. Therefore, a graph-based method is also developed in this paper. In the graph-based method, the co-scheduling problem is represented in a co-scheduling graph, and then a Heuristic A*-search (HA*) method is designed to find the near-optimal solutions much more efficiently. A communication-aware process condensation technique is proposed to further accelerate the HA* method.

The rest of this paper is organized as follows: Section II shows how we model the problem of co-scheduling both serial and parallel jobs as an Integer Programming problem. Section III presents the graph-based method and Heuristic A*-search method to find the near-optimal co-scheduling solutions. In Section IV, we conduct the experiments to evaluate the effectiveness and efficiency of the co-scheduling methods proposed in this paper. Related work is discussed in Section V. Finally, section VI concludes this paper.

II. Modelling Co-Scheduling as Integer Programming Problems

In this section, Subsection II-A first briefly summarizes how co-scheduling serial jobs is modelled as an IP problem in [19]. Then Subsection II-B presents how we model the problem of co-scheduling a mix of serial and parallel jobs as an IP problem.

A. Co-Scheduling serial jobs

The work in [19] shows that due to resource contention, the co-running jobs generally run slower on a multi-core processor than they run alone. This performance degradation is called the co-run degradation. When a job i co-runs with the jobs in a job set S, the co-run degradation of job i can be formally defined as Eq. 1, where ct_i is the computation time when job i runs alone, S is a set of jobs and $ct_{i,S}$ is the computation time when job i co-runs with the set of jobs in S.

$$d_{i,S} = \frac{ct_{i,S} - ct_i}{ct_i} \tag{1}$$

In the co-scheduling problem considered in [19], a set P of n serial jobs are allocated to multiple identical u-core processors so that each core is allocated with one job. m denotes the number of u-core processors needed, which can be calculated as $\frac{n}{u}$ (if n cannot be divided by u, we can simply add ($u - n \bmod u$) imaginary jobs which have no performance degradation with any other jobs). The objective of the co-scheduling problem is to find the optimal way to partition n jobs into m u-cardinality sets, so that the sum of $d_{i,S}$ in Eq.1 over all n jobs is minimized. This objective can be formalized as the following IP problem shown in Eq. 2, where x_{i,S_i} is the decision variable of the IP and S_i is a job set that co-runs with job p_i. The decision constraints of the IP problem are shown in Eq.3 and Eq.4. Note that the number of all job sets that may co-run with job p_i (i.e., the number of all possible S_i) is $\binom{n-1}{u-1}$.

$$min \sum_{i=1}^{n} d_{i,S_i} x_{i,S_i} \tag{2}$$

$$x_{i,S_i} = \begin{cases} 1 & \text{if } p_i \text{ is co-scheduled with } S_i, \\ 0 & \text{otherwise.} \end{cases} \quad 1 \le i \le n \tag{3}$$

$$\sum_{all S_i} x_{i,S_i} = 1, \quad 1 \le i \le n \tag{4}$$

B. Co-scheduling a mix of serial and parallel jobs

In this section, Subsection II-B1 first constructs the IP model for the *Embarrassingly Parallel* (PE) jobs. In a PE job, there are no communications among its parallel processes. An example of a PE job is parallel Monte Carlo simulation [26]. In such jobs, multiple slave processes run simultaneously to perform the Monte Carlo simulations. Each slave process completes its part of the work without the need for communication. Once its computation finishes it sends the result back to the master process and halts. The master process then reduces the final result (i.e., calculating the average) from received data.

In Subsection II-B2, we extend the IP model to co-schedule the general parallel jobs that require inter-process communications during the job executions, which we call PC jobs (Parallel jobs with Communications). An example of a PC job is a MPI application for matrix multiplication.

In both types of parallel jobs, the finish time of a job is determined by their slowest process in the job.

1) IP model for PE jobs: Eq.2 cannot be used as the objective for finding the optimal co-scheduling of parallel jobs. This is because Eq.2 will sum up the degradation experienced by each process of a parallel job. However, as explained above, the finish time of a parallel job is determined by its slowest process. In the case of the PE jobs, a bigger degradation of a process indicates a longer execution time for that process. Therefore, no matter how small degradation other processes have, the execution flow in the parallel job has to wait until the process with the biggest degradation finishes. Thus, the finish time of a parallel job is determined by the biggest degradation experienced by all of its processes, which is denoted by Eq.5. Therefore, co-scheduling a mix of serial jobs and PE jobs can be modelled as the following IP problem. The total degradation should be calculated using Eq. 6, where n is the number of all processes (a serial job has one process and a PE has multiple processes), δ_j is a parallel job, P is the number of parallel jobs, S_i is the set of processes that may co-run with process p_i. The decision constraints of Eq.6 are the same as those for the IP modelling for serial jobs, i.e., Eq.3 and Eq.4.

$$d_{\delta_j} = \max_{p_i \in \delta_j} d_{i,S_i} \tag{5}$$

$$min(\sum_{j=1}^{P}(\max_{p_i \in \delta_j}(d_{i,S_i} \times x_{i,S_i})) + \sum_{i=1}^{n-P} d_{i,S_i} \times x_{i,S_i}) \tag{6}$$

The $\max$ operation in Eq 6 can be eliminated by introducing an auxiliary variable y_j for each parallel job δ_j. Each y_j has the following inequality relation with the original decision variables.

$$for\ all\ p_i \in \delta_j, d_{i,S_i} x_{i,S_i} \leq y_j \tag{7}$$

Therefore, the objective function in (6) is transformed to

$$min(\sum_{j=1}^{P} y_j + \sum_{i=1}^{n-P}(d_{i,S_i} \times x_{i,S_i})) \tag{8}$$

2) IP model for PC jobs: In the case of the PC jobs, the slowest process in a parallel job is determined by both performance degradation and communication time. Therefore, we define the *communication-combined degradation*, which is expressed using Eq. 9, where $c_{i,S}$ is the communication time taken by parallel process p_i when p_i co-runs with the processes in S_i. As with d_{i,S_i}, c_{i,S_i} also varies with the co-scheduling solutions. We can see from Eq. 9 that for all processes in a parallel job, the one with the biggest sum of performance degradation (in terms of the computation time) and the communication has the greatest value of d_{i,S_i}, since the computation time of all processes (i.e., ct_i) in a parallel job is the same when a parallel job is evenly balanced. Therefore, the greatest d_{i,S_i} of all processes in a parallel job should be used as the communication-combined degradation for that parallel job.

When the set of jobs to be co-scheduled includes both serial jobs and PC jobs, we use Eq.9 to calculate d_{i,S_i} for each parallel process p_i, and then we replace d_{i,S_i} in Eq.6 with that calculated by Eq. 9 to formulate the objective of co-scheduling a mix of serial and PC jobs.

$$d_{i,S_i} = \frac{ct_{i,S_i} - ct_i + c_{i,S_i}}{ct_i} \tag{9}$$

Whether Eq. 6 replaced with d_{i,S_i} calculated by Eq. 9 still makes an IP problem depends on the form of c_{i,S_i}. Next, we first present the modelling of c_{i,S_i}, and then use an example for illustration.

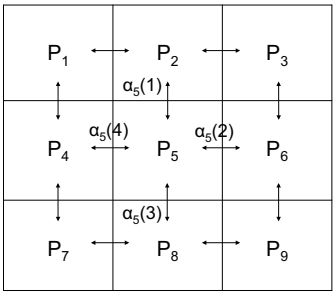
Assume that a parallel job has regular communication patterns among its processes. c_{i,S_i} can be modelled using Eq. 10 and 11, where γ_i is the number of the neighbouring processes that process p_i has corresponding to the decomposition performed on the data set to be calculated by the parallel job, $\alpha_i(k)$ is the amount of data that p_i needs to communicate with its k-th neighbouring process, B is the bandwidth for inter-processor communication (typically, the communication bandwidth between the machines in a cluster is same), $b_i(k)$ is p_i's k-th neighbouring process, and $\beta_i(k, S_i)$ is 0 or 1 as defined in Eq. 11. $\beta_i(k, S_i)$ is 0 if $b_i(k)$ is in the job set S_i co-running with p_i. Otherwise, $\beta_i(k, S_i)$ is 1.

Essentially, Eq. 10 calculates the total amount of data that p_i needs to communicate, which is then divided by the bandwidth B to obtain the communication time. $\beta_i(k, S_i)$ in Eq. 10 is further determined by Eq. 11. Note that p_i's communication time can be determined by only examining which neighbouring processes are not in the job set S_i co-running with p_i, no matter which machines that these neighbouring processes are scheduled to. Namely, c_{i,S_i} can be calculated by only knowing the information of the local machine where process p_i is located. Therefore, such a form of c_{i,S_i} makes Eq. 9 still be of an IP form that can solved by the existing IP solvers.

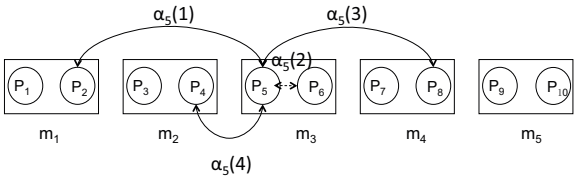
$$c_{i,S_i} = \frac{1}{B}\sum_{k=1}^{\gamma_i}(\alpha_i(k) \times \beta_i(k, S_i)) \tag{10}$$

$$\beta_i(k, S_i) = \begin{cases} 0 & \text{if } b_i(k) \in S_i \\ 1 & \text{if } b_i(k) \notin S_i \end{cases} \tag{11}$$

We use an example as shown in Fig. 2 to illustrate the calculation of c_{i,S_i}. Fig. 2a represents a data set to be calculated on. Assume that a typical 2-dimensional decomposition is performed on the data set, resulting in 9 subsets of data. These 9 subsets of data are calculated by 9 processes in parallel (e.g., using MPI). The arrows between the data subsets in Fig. 2a represent the communication pattern between the processes. Assume that the parallel job is labelled as δ_1 and the 9 processes in δ_1 are labelled as $p_1, ..., p_9$. Also assume that these 9 processes are scheduled on 2-core machines as shown in Fig. 2b. Now consider process p_5. It has intra-processor communication with p_6 and inter-processor communications with p_2, p_4 and p_8. Since the intra-processor communication can occur simultaneously with inter-processor communication and the intra-processor communication is always faster than the inter-processor communication, the communication



(a) An exemplar parallel job δ_1 and its inter-process communication pattern



(b) A schedule of the parallel job δ_1 and a serial job $p_1 0$ on 2-core machines

Fig. 2: An illustrative example for modelling the communication time

time taken by process p_5 in the schedule, i.e., $c_{5,\{p_6\}}$, is $\frac{1}{B}(\alpha_5(1)+\alpha_5(3)+\alpha_5(4))$. Note that in typical 1D, 2D or 3D decompositions, the data that a process has to communicate with the neighbouring processes in the same dimension are the same. In Fig. 2, for example, $\alpha_5(1) = \alpha_5(3)$ and $\alpha_5(2) = \alpha_5(4)$.

III. A GRAPH-BASED METHOD FOR FINDING OPTIMAL CO-SCHEDULING SOLUTIONS

This Section presents a graph-based method to find the co-scheduling solutions. Subsection III-A proposes a *co-scheduling graph* to model the co-scheduling problem. The problem of finding the optimal co-scheduling solution for both serial and parallel jobs can then be converted to the problem of finding the shortest VALID path in the constructed graph. In our previous work [18], we applied the A*-search algorithm to find the optimal co-scheduling solution. However, it takes the algorithm long time to find the optimal solution for the problems of relatively large scale. Therefore, in this paper, a heuristic A*-search algorithm is proposed (Subsection III-B) to find the near-optimal solution with much higher efficiency.

A. The graph model

As formalized in Section II, the objective of solving the co-scheduling problem for both serial and PC jobs is to find a way to partition n jobs into m u-cardinality sets, so that the total degradation of all jobs is minimized (note that degradation refers to performance degradation defined in Eq. 1 and communication-combined degradation defined in Eq. 9 for serial jobs and parallel jobs, respectively). The number of all possible u-cardinality sets is $\binom{n}{u}$. In the rest of this section, a process refers to a serial job or a parallel process unless it causes confusion.

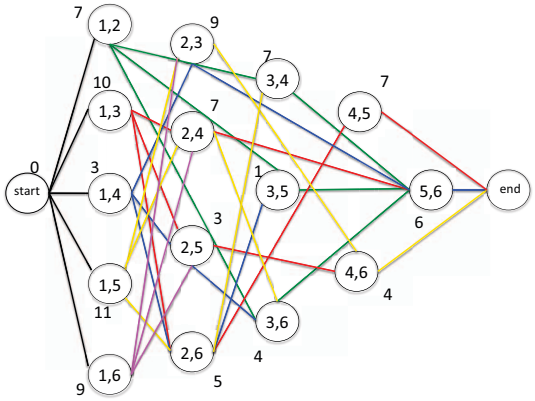


Fig. 3: The exemplar co-scheduling graph for co-scheduling 6 jobs on Dual-core machines; the list of numbers in each node is the node ID; A number in a node ID is a job ID; The edges of the same color form the possible co-scheduling solutions; The number next to the node is the node weight, i.e., total degradation of the jobs in the node.

In this paper, a graph is constructed, called the co-scheduling graph, to model the co-scheduling problem. There are $\binom{n}{u}$ nodes in the graph and each node corresponds to a u-cardinality set. Each node represents a u-core processor with u processes assigned to it. The ID of a node consists of a list of the IDs of the processes in the node. In the list, process IDs are always placed in an ascending order. The weight of a node is defined as the total degradation of the u processes in the node. The nodes are organized into multiple levels in the graph. The i-th level contains all nodes in which the ID of the first process is i. Then the number of nodes in level i is $\binom{n-i}{u-1}$ since level i contains all combinations of $u-1$ jobs from $n-i$ jobs. In each level, the nodes are placed in ascending order of their ID's. A *start* node and an *end* node are added as the first (level 0) and the last level of the graph, respectively. The weights of the start and the end nodes are both 0. The edges between the nodes are dynamically established as the algorithm of finding the optimal solution progresses. Such organization of the graph nodes will be used to help optimize the co-scheduling algorithms proposed in this paper. Figure 3 illustrates the case where 6 processes are co-scheduled to 2-core processors. The figure also shows how to code the nodes in the graph and how to organize the nodes into different levels. Note that for the clarity we did not draw all edges.

In the constructed co-scheduling graph, a path from the start to the end node in the graph forms a co-scheduling solution if the path does not contain duplicated jobs, which is called a *valid path*. Finding the optimal co-scheduling solution is

equivalent to finding the shortest valid path from the start to the end node.

B. Heuristic A*-search Algorithm

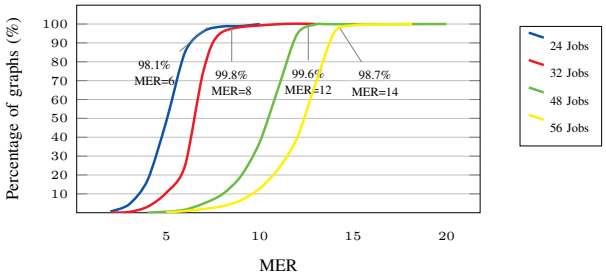
In this section, a Heuristic A*-search (HA*) method is proposed to trim the searching in the co-scheduling graph. The resulting search space is much smaller than the original one and therefore the co-scheduling solution, which is sub-optimal, can be computed more efficiently by order of magnitude than the A*-search algorithm in our previous work [18], which aims to find the optimal solution (we now call it OA*).

The principle of trimming the co-scheduling graph is based on the following insight. We re-arrange the nodes in each level of the co-scheduling graph in the ascending order of node weight. We then apply OA* to find the shortest path from the sorted graph. For each node on the shortest path, we record its rank in the graph level that the node is in (the *i*-th node in a level has the rank of *i*). Assume that a node on the computed shortest path has the rank of *i* in its level. We also record how many invalid nodes the algorithm has to skip from rank 1 to rank *i* in the level before locating this valid node of rank *i*. Assume the number of invalid nodes is *j*. Then (*i*-*j*) is the number of nodes that the algorithm has attempted in the level before reaching the node that is on the shortest path. We call this number, $i - j$, the *effective rank* of the node of rank *i* in the level. We calculate the effective rank for every node on the shortest path and obtain the maximum of them, which we denote by MER (Maximum Effective Rank of the shortest path). If we had known the value of MER, assuming it is *k*, before we apply the OA*-search algorithm, we can instruct the algorithm to only attempt the first *k* valid nodes in each level and the algorithm will still be able to find the shortest path of the algorithm.

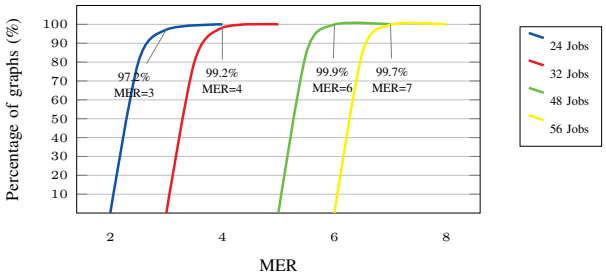
Given the above insight, we designed the following benchmarking experiment to conduct the statistical analysis for the value of MER. The numbers of jobs we used are 24, 32, 48 and 56 jobs. For each job batch and *u*-core machines, we randomly generated *K* different cache misses for a job (the cache miss rate of a job is randomly selected from the range of [15%, 75%]) and construct *K* different co-scheduling graphs. We then used OA* to find the shortest path of each graph and record the value of MER. Figure 4a and 4b depict the Cumulative Distribution Functions (CDF) of MER with 1000 graphs (i.e., *K*=1000) on Quad-core machines and 8-core machines, respectively.

As shown in Figure 4a, when the number of jobs is 24, the value of MER is no more than 6 for 98.1% of graphs. Similarly, when the numbers of jobs are 32, 48 and 56, the values of MER are no more than 8, 12 and 14 for 99.8%, 99.6% and 98.7% of graphs, respectively. The corresponding figures for the case where the jobs are co-scheduled on 8-core machines are shown in Figure 4b.

From these benchmarking results, we find that we can use the function $MER = \frac{n}{u}$, where n is the number of jobs and u is the number of cores in a machine, to predict the value of MER. With this MER function, the actual value of MER will



(a) On Quad-core machines



(b) On 8-core machines

Fig. 4: Cumulative Distribution Function (CDF) of MER

be no more than the predicted one in almost all cases. The reason why such a MER function generates very good results can be explained in principle as follows, although we found that it was difficult to give rigorous proof. We know that the nodes with too big weights has less chance to appear on the shortest path. Therefore, when OA* attempts the nodes in a level to expand the current path, if a node's effective rank is more than $\frac{n}{u}$, which is the number of machines that is needed to co-run this batch of jobs, the node will not be selected even if a poor greedy algorithm is used to map the node to one of the $\frac{n}{u}$ machines.

Based on the above statistical analysis, we adjust the co-scheduling graph and trim the searching for the shortest path in the following way. In each level of the graph, the nodes are arranged in the ascending order of node weight. When OA* searches for the shortest path, it only attempts $\frac{n}{u}$ valid nodes in each level to expand the current path, if $\frac{n}{u}$ is less than the number of valid nodes in the level. This way, the graph scale and consequently the number of operations performed by the algorithm are reduced by order of magnitude. We call the A*-search algorithm operating in this fashion the *Heuristic A*-search* (HA*) algorithm.

It is difficult to analyze the time complexity of the A*-search algorithm since it depends on the design of the $h(v)$ function. However, the time complexity of our A*-search algorithm mainly depends on how many valid nodes the algorithm have to attempt when extending a current path to a new graph level, which can be used to analyze the complexity difference between OA* and HA*. Assume that OA* is searching for co-scheduling solutions of *n* jobs on

u-core machines and that the current path includes *k* nodes. When OA* extends the current path to a new level, the number of valid nodes that OA* may have to attempt in the new level can be calculated by $\binom{(n-1)-k\cdot u}{u-1}$, since all nodes that contain the jobs that appear in the current path are not valid nodes. Under the same settings, the number of valid nodes that HA* needs to attempt is only $\frac{n}{u}$. The following example is given to show the complexity difference between these two methods. When *n* is 100, *u* is 4 and *k* is 2, $\binom{(n-1)-k\cdot u}{u-1}$ is 121485 while $\frac{n}{u}$ is 25. The value of $\binom{(n-1)-k\cdot u}{u-1}$ becomes less than 25, only when *k* is bigger than 23 (The value of $\binom{(n-1)-k\cdot u}{u-1}$ is 35 when *k* is 23). But the biggest value of *k* is 24 since there are total 25 nodes in a complete path when co-scheduling 100 jobs on Quad-core machines. This means that in almost all cases (except the last graph level) $\binom{(n-1)-k\cdot u}{u-1}$ is bigger than $\frac{n}{u}$ by orders of magnitude. This is the reason why HA* is much more efficient than OA*, which is also supported by the experimental results presented in Section IV.

IV. Evaluation

This section evaluates the effectiveness and the efficiency of three co-scheduling methods proposed in this work: the IP-based method (IP), the OA*-search algorithm (OA*) and the heuristic A*-search algorithm (HA*).

We conducted the experiments with data collected from real jobs. The serial jobs are taken from the NASA benchmark suit NPB3.3-SER [9] and SPEC CPU 2000 [10]. NPB3.3-SER has 10 serial programs and each program has 5 different problem sizes. The problem size used in the experiments is size *C*. The PC jobs are selected from these ten MPI applications in the NPB3.3-MPI benchmark suite. As for PE jobs, 5 embarrassingly parallel programs are used: PI [4], Mandelbrot Set(MMS) [2], RandomAccess(RA) from HPCC benchmark [1], EP from NPB-MPI [9] and Markov Chain Monte Carlo for Bayesian inference (MCM) [22]. In all these 5 embarrassingly parallel programs, multiple slave processes are used to perform calculations in parallel and a master process reduces the final result after it gathers the partial results from all slaves. These set of parallel programs are selected because they contain both computation-intensive programs (e.g, MMS and PI) and memory-intensive programs (e.g, RA).

Three types of machines, Dual-core, Quad-core and 8-core machines, are used to run the benchmarking programs. A dual-core machine has an Intel Core 2 Dual processor and each core has a dedicated 32KB L1 data cache and a 4MB 16-way L2 cache shared by the two cores. A Quad-core machine has an Intel Core i7 2600 processor and each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache. A further 8MB 16-way L3 cache is shared by the four cores. The processor in the 8-core machine is Intel Xeon E5-2450L. Each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache, and a 16-way 20MB L3 cache is shared by 8 cores. The interconnect network is the 10 Gigabit Ethernet.

The method presented in [25] is used to estimate the co-run execution times of the programs. CPU_Time denotes the computation time of a job. According to [25], CPU_Time is calculated using Eq. 12.

$$CPU\_Time = (CPU\_Clock\_Cycle + Memory\_Stall\_Cycle) \times Clock\_Cycle\_Time \quad (12)$$

$Memory\_Stall\_Cycle$ in Eq. 12 is computed by Eq. 13, where $Number\_of\_Misses$ is the number of cache misses.

$$Memory\_Stall\_Cycle = Number\_of\_Misses \times Miss\_Penalty \quad (13)$$

The values of CPU_Clock_Cycle and $Number\_of\_Misses$ for a single-run program can be obtained using *perf* [5] in Linux. Then the value of $Memory\_Stall\_Cycle$ for a single-run program can be obtained by Eq. 13. CPU_Time for a single-run program can also be obtained by *perf*.

The value of CPU_Time for a co-run program can be estimated in the following way. We use the *gcc-slo* compiler suite [11] to generate the SDP (Stack Distance Profile) for each benchmarking program offline, and then apply the SDC (Stack Distance Competition) prediction model in [14] to predicate $Number\_of\_Misses$ for the co-run programs. Then Eq. 13 and Eq. 12 are used to estimate $Memory\_Stall\_Cycle$ and CPU_Time for the co-run programs.

With the single-run and co-run values of CPU_Time, Eq. 1 is used to compute the performance degradation.

In order to obtain the communication time of a parallel process when it is scheduled to co-run with a set of processes, i.e., c_{i,S_i} in Eq. 10, we examined the source codes of the benchmarking MPI programs used for the experiments and obtained the amount of data that the process needs to communicate with each of its neighbouring processes (i.e., $\alpha_i(k)$ in Eq. 10). And then Eq. 10 and Eq. 11 are used to calculate c_{i,S_i}.

A. Comparing the effectiveness of IP and OA*

This section reports the results for validating the optimality of IP proposed in this paper. It has been shown that the OA* constructed in [18] produces the optimal solution. Therefore, we first compare IP with the OA* algorithm in [18] for co-scheduling serial jobs on Dual-core and Quad-core machines. In our experiments, we employ the IP solver, CPLEX [7], to compute the optimal co-schedules. The experiments use all 10 serial benchmark programs from the NPB-SER suite and 6 serial programs (applu, art, ammp, equake, galgel and vpr) are selected from SPEC CPU 2000. The experimental results are presented in Table I. We also compare OA* and the IP model constructed in this paper for co-scheduling a mix of serial and parallel programs. The results are listed in Table II. In these experiments, two MPI applications (i.e., MG-Par and LU-Par) are selected from the NPB3.3-MPI and combined with serial programs chosen from NPE-SER and SPEC CPU 2000. The processes of each parallel application varies from 2 to 4. The

detailed combinations of serial and parallel programs are listed below:

- In the case of 8 processes, MG-Par and LU-Par are combined with applu, art, equake and vpr.
- In the case of 12 processes, MG-Par and LU-Par are combined with applu, art, ammp, equake, galgel and vpr.
- In the case of 16 processes, MG-Par and LU-Par are combined with BT, IS, applu, art, ammp, equake, galgel and vpr.

TABLE I: Comparison between OA* and IP for serial jobs

Number of Jobs	Average Degradation			
	Dual Core		Quad Core	
	IP	OA*	IP	OA*
8	0.12	0.12	0.34	0.34
12	0.22	0.22	0.36	0.36
16	0.13	0.13	0.27	0.27

TABLE II: Comparison of IP and OA* for serial and parallel jobs

Number of Jobs	Average Degradation			
	Dual Core		Quad Core	
	IP	OA*	IP	OA*
8	0.07	0.07	0.098	0.098
12	0.05	0.05	0.074	0.74
16	0.12	0.12	0.15	0.15

As can be seen from Table I and II, OA* achieves the same performance degradation as that by the IP model. These results verify the optimality of OA*.

*B. Efficiency of IP and OA**

This subsection investigates the efficiency of IP and OA*, i.e., the time spent by the methods in finding the optimal co-scheduling solutions. We used various IP solvers, CPLEX [7], CBC [6], SCIP [3] and GLPK [8], to solve the same IP model. The results are shown in Table III. As can be observed from Table III, CPLEX is the fastest IP solver. Note that when the number of processes are 16, the solving times by SCIP are all around 1000 seconds. This is only because the SCIP solver gave up the searching after 1000 seconds, deeming that it cannot find the final solution. It can be seen that the IP solvers are not efficient in solving the optimal co-scheduling problem. In fact, our records show that none of these IP solvers can manage to solve the IP model for more than 24 processes. Our previous work [34] proposes the O-SVP algorithm to compute the co-scheduling solutions for parallel and serial jobs. We also used O-SVP to find the co-scheduling solutions in the experiments and present the solving times in Table III.

It can be seen that OA* finds the optimal solution much more efficiently than O-SVP and that the time gap becomes increasingly bigger as the number of jobs increases.

C. Heuristic A-search algorithm*

The experiments presented in this subsection aim to verify the effectiveness of HA*. We conducted the experiments to compare the solutions obtained by HA* and those by OA*. We also compared HA* with the heuristic algorithm (denoted by PG) developed in [19] for finding co-scheduling solutions.

TABLE III: Efficiency of different methods on Quad-core machines

Number of Jobs	Solving time (seconds)				
	CPLEX	CBC	SCIP	GLPK	OA*
8(se)	0.086	0.19	0.28	0.049	0.004
8(pe)	0.33	0.26	0.21	0.041	0.005
8(pc)	0.48	0.45	0.24	0.038	0.006
12(se)	3.44	72.74	51.09	51.58	0.15
12(pe)	0.998	13.56	30.32	15.97	0.24
12(pc)	2.23	21.09	29.82	16.42	0.2
16(se)	33.4	704	1000	33042	0.63
16(pe)	32.52	303	1001	1231	1.52
16(pc)	11.76	313	1001	1170	1.63

PG first calculates the politeness of each job based on the degradation that the job causes when it co-runs with other jobs, and then applies the greedy algorithm to co-schedule "polite" jobs with "impolite" jobs.

In the experiments, we choose 12 applications from NPB3.3-SER and SPEC CPU 2000 (BT, CG, EP, FT, IS, LU, MG, SP, UA, DC, art and ammp) and co-schedule them on Quad-core machines using OA*, HA* and PG. We also conducted the similar experiments on 8-core machines, in which 16 applications were used from NPB3.3-SER and SPEC CPU 2000 (BT, CG, EP, FT, IS, LU, MG, SP, UA, DC, art, ammp, applu, equake, galgel and vpr). The experimental results for Quad-core and 8-core machines are presented in Figure 5 and Figure 6, respectively. Note that the algorithms aim to optimize the average performance degradation of the batch of jobs, which is labelled by "AVG", not to optimize performance degradation of each individual job.

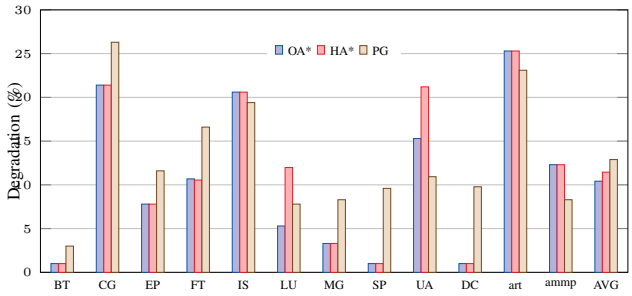


Fig. 5: Comparing performance degradations of benchmarking applications on Quad-core machines under OA*, HA* and PG

In Figure 5 and 6, the average performance degradation obtained by HA* is worse than OA* only by 9.8% and 4.6% on Quad-core and 8-core machines, respectively, while HA* outperforms PG by 12.6% and 14.6% on Quad-core and 8-core machines, respectively. These results show the effectiveness of the heuristic approach, i.e., using the MER function, in HA* and that the heuristic method can deliver the near-optimal performance.

We also used the synthetic jobs to conduct larger-scale experiments and compare HA* and PG. The synthetic jobs are generated in the same way as in Figure 4a and 4b. The results on Quad-core and 8-core machines are presented in

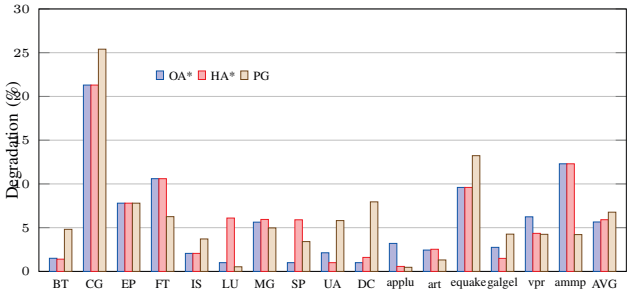
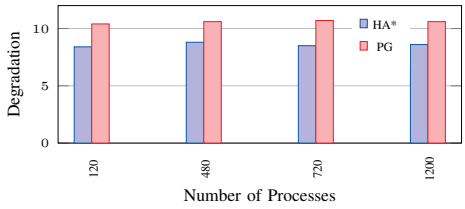
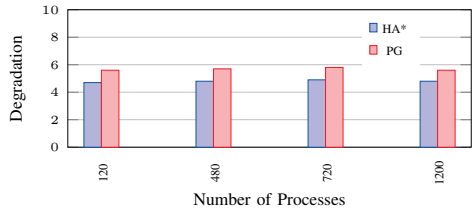


Fig. 6: Comparing performance degradations of benchmarking applications on 8-core machines under OA*, HA* and PG

Figure 7a and 7b, respectively. It can be seen from the tables that HA* outperforms PG in all cases, by 20%-25% on Quad-core machines and 16%-18% on 8-core machines).



(a) 4 Core



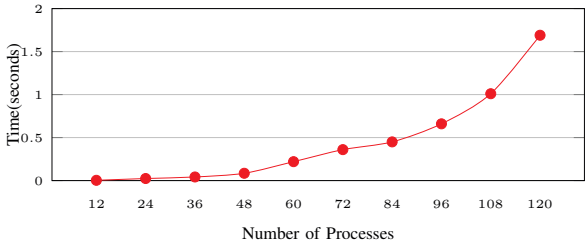
(b) 8 Core

Fig. 7: Comparing the degradation under HA* and PG algorithms

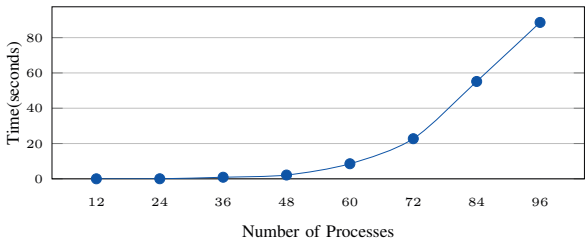
We further investigated the scalability of HA* in terms of the time spent in finding the co-scheduling solutions. Figure 9 shows the scalability for co-scheduling synthetic jobs on Quad-core and 8-core machines. The synthetic jobs in this figure are generated in the same way as in Figure 4a and 4b.

By comparing the scalability curve for Quad-core machines in Figure 9 with that in Figure 8b, it can be seen that HA* performs much more efficiently than OA*. Another interesting observation from Figure 9 is that HA* spends much less time to find solutions on 8-core machines than on Quad-core machines. This is because we use the MER function, $\frac{n}{u}$, to trim the searching in HA*. When there are more cores in a machine (consequently, less machines are needed to run the same batch of jobs), less number of valid nodes will be examined in each level of the co-scheduling graph and therefore less time is taken by HA*. The scalability trend of OA* is different as shown in Figure 8a and 8b. In OA*, when the jobs are scheduled on machines with more cores, the co-scheduling graph becomes bigger with more graph nodes. OA* will examine all nodes in the graph in any case. Therefore, the solving time of OA* increases as the number of cores in a machine increases.

Figure 8a and 8b show the scalability of OA* on Dual-core and Quad-core machines, respectively, as the number of serial processes increases. The scalability trend of OA* is different as shown in Figure 8a and 8b. In OA*, when the jobs are scheduled on machines with more cores, the co-scheduling graph becomes bigger with more graph nodes. OA* will examine all nodes in the graph in any case. Therefore, the solving time of OA* increases as the number of cores in a machine increases.



(a) On dual-core machines



(b) On Quad-core machines

Fig. 8: Scalability of OA*

V. Related Work

This section first discusses the co-scheduling strategies proposed in the literature. Similar to the work in [19], our method needs to know the performance degradation of the jobs when they co-run on a multi-core machine. Therefore,

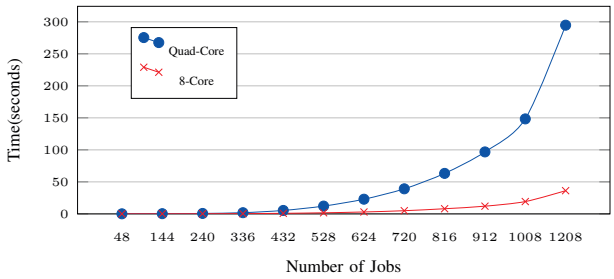


Fig. 9: Scalability of HA* on Quad-core and 8-core machines

this section also presents the methods that can acquire the information of performance degradation.

A. Co-scheduling strategies

Many co-scheduling schemes have been proposed to reduce the shared cache contention in a multi-core processor. Different metrics can be used to indicate the resource contention, such as Cache Miss Rate (CMR), overuse of memory bandwidth, and performance degradation of co-running jobs. These schemes fall into the following two classes.

The first class of co-scheduling schemes aims at improving the runtime schedulers and providing online co-scheduling solutions. The work in [13] developed the co-schedulers that reduce the cache miss rate of co-running jobs. The fundamental idea of these co-schedulers is to uniformly distribute the jobs with high cache requirements across the processors. Wang et al. [30] demonstrated that the cache contention can be reduced by rearranging the scheduling order of the tasks. The center idea of their algorithm is to distinguish the jobs with different cache requirements, and change the scheduling order so as to schedule the jobs with high cache requirements on the same chip as the jobs with low cache requirements. In doing so, the contention for the shared cache can be reduced. Recent studies on resource contention in multi-core processors have pointed out that the contention may appear along the entire memory hierarchy. To address this problem, Feliu et al. [17] proposed an algorithm to tackle the problem of overusing memory bandwidth. The work measures the available bandwidth at each level of memory hierarchy and balances the bandwidth utilization along the job execution times of the jobs.

The work in [19] defined the performance degradation of the co-running jobs as the metric to measure the level of resource contention. A greedy algorithm was developed in [19] to find the co-scheduling solution efficiently. In the greedy algorithm, the polite and impolite jobs were defined. The strategy was then to assign friendly jobs with unfriendly jobs. However, the work only considers serial jobs.

The second class of co-scheduling schemes focuses on providing the basis for conducting performance analysis. It mainly aims to find the optimal co-scheduling performance offline, in order to providing a performance target for other co-scheduling systems. The extensive research is conducted in [19] to find the co-scheduling solutions. The work models the co-scheduling problem for serial jobs as an Integer Programming (IP) problem, and then uses the existing IP solver to find the optimal co-scheduling solution (it also proposes a set of heuristics-based algorithms to find the near optimal co-scheduling). Although the work in [19] can obtain the optimal co-scheduling solution, their approach is only for serial jobs.

In this paper, two new methods, an IP-based method and a graph-based method, are developed to find the optimal co-scheduling solution offline for both serial and parallel jobs. Furthermore, based on the co-scheduling graph constructed for representing the co-scheduling problem, a heuristic method is also proposed in this paper to find the near-optimal solutions with much less time.

Our previous work [34] [18] also studied the co-scheduling problem. This work differs from our previous work in the following aspects. First, this paper models the co-scheduling problem as an Integer Program (IP). This way, the existing IP solvers can be used to find the co-scheduling solutions. Second, the work in [34] proposed an O-SVP algorithm, while the work in [18] designed the OA* algorithm. Both algorithms aim to find the optimal co-scheduling solutions. Due to the nature of the problem, the algorithm can only be used to find the solution for the problems with relatively small scale. When the number of jobs becomes big, the solving time will become very long. In this work, a Heuristic A*-search (HA*) algorithm is proposed. The algorithm is able to find the co-scheduling solution much more efficiently. In terms of solution quality, HA* is only worse than OA* by a very small margin (less than 10% shown in our experiments) and is better than the existing heuristic method presented for the co-scheduling problem [19].

B. Acquiring the information of performance degradation

When a job co-runs with a set of other jobs, its performance degradation can be obtained through prediction [14] or offline profiling [29]. Predicting performance degradation has been well studied in the literature [20], [33]. One of the best-known methods is Stack Distance Competition (SDC) [14]. This method uses the Stack Distance Profile (SDP) to record the hits and misses of each cache line when each process is running alone. The SDC model tries to construct a new SDP that merges the separate SDPs of individual processes that are to be co-run together. This model relies on the intuition that a process that reuses its cache lines more frequently will occupy more cache space than other processes. Based on this, the SDC model examines the cache hit count of each process's stack distance position. For each position, the process with the highest cache hit count is selected and copied into the merged profile. After the last position, the effective cache space for each process is computed based on the number of stack distance counters in the merged profile.

The offline profiling can obtain more accurate degradation information, although it is more time consuming. Since the goal of this paper is to find the optimal co-scheduling solutions offline, this method is also applicable in our work.

VI. Conclusion and future work

This paper explores the problem of finding the co-scheduling solutions for a mix of serial and parallel jobs on multi-core processors. The co-scheduling problem is first modelled as an IP problem and then the existing IP solvers can be used to find the optimal co-scheduling solutions. In this paper, the co-scheduling problem is also modelled as a co-scheduling graph and the problem of finding the optimal co-scheduling solution can then converted into the problem of finding the shortest valid path in the co-scheduling graph. A heuristic A*-search algorithm is then proposed to find the near-optimal solutions efficiently. Future work has been planned

in the following two folds. 1) It is possible to parallelize the proposed co-scheduling methods to further speedup the solving process. We plan to investigate the parallel paradigm suitable for this problem and design the suitable parallelization strategies. 2) We plan to extend our co-scheduling methods to solve the optimal mapping of virtual machines (VM) on physical machines. The main extension is to allow the VM migrations between physical machines.

VII. Acknowledgement

This research is partly supported by the Key Program of National Natural Science Foundation of China (Grant Numbers: 61133005 and 61432005), and the National Natural Science Foundation of China (Grant Numbers: 61272190 and 61173166).

References

[1] http://icl.cs.utk.edu/hpcc/.
[2] http://people.ds.cam.ac.uk/nmm1/mpi/programs/mandelbrot.c.
[3] http://scip.zib.de.
[4] https://computing.llnl.gov/tutorials/mpi/samples/c/mpi_pi_reduce.c.
[5] https://perf.wiki.kernel.org/index.php/main_page.
[6] https://projects.coin-or.org/cbc.
[7] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.
[8] http://www.gnu.org/software/glpk/.
[9] http://www.nas.nasa.gov/publications/npb.html.
[10] http://www.spec.org.
[11] K. Beyls and E.H. D'Hollander. Refactoring for data locality. *Computer*, 2009.
[12] M Bhadauria and S.A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, 2010.
[13] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 2010.
[14] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.
[15] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.
[16] A. Fedorova, M. Seltzer, and M.D Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.
[17] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-hierarchy contention aware scheduling in cmps. *Parallel and Distributed Systems, IEEE Transactions on*, 2013.
[18] Ligang He, Huanzhou Zhu, and S.A Jarvis. Developing graph-based co-scheduling algorithms on multicore computers. *Parallel and Distributed Systems, IEEE Transactions on*, 2015.
[19] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *Parallel and Distributed Systems, IEEE Transactions on*, 2011.
[20] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.
[21] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z Wen, and C Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007.*, 2007.
[22] E.J Kontoghiorghes. *Handbook of Parallel Computing and Statistics*. Chapman & Hall/CRC, 2005.
[23] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 451–462. ACM, 2012.
[24] K.J Nesbit, J. Laudon, and J.E Smith. Virtual private caches. In *ACM SIGARCH Computer Architecture News*, 2007.
[25] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.
[26] J.S Rosenthal. Parallel computing and monte carlo algorithms. *Far east journal of theoretical statistics*, 2000.
[27] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, 2013.
[28] S. Srikantaiah, M. Kandemir, and M.J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ACM Sigplan Notices*, 2008.
[29] N. Tuck and D.M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, 2003.
[30] Y. Wang, Y. Cui, P. Tao, H. Fan, Y. Chen, and Y. Shi. Reducing shared cache contention by scheduling order adjustment on commodity multi-cores. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing*, 2011.
[31] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.
[32] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009.
[33] Q. Zhao, D. Koh, S. Raza, D. Bruening, W. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. *ACM SIGPLAN Notices*, 2011.
[34] Huanzhou Zhu, Ligang He, and S.A. Jarvis. Optimizing job scheduling on multicore computers. In *Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 61–70, 2014.
[35] S. Zhuravlev, S. Saez, J. Cand Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 2012.