2011

Analysis and Approximation of Optimal Co-Scheduling on CMP

Yunlian Jiang
College of William & Mary - Arts & Sciences

Analysis and Approximation of Optimal Co-Scheduling on CMP

Yunlian Jiang

Tongnan, Chongqing, China

Bachelor of Science, University of Science and Technology of China, 2003
Master of Engineering, University of Science and Technology of China, 2006

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

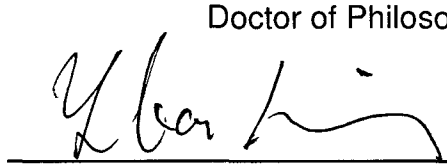
Department of Computer Science

The College of William and Mary
August 2011

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

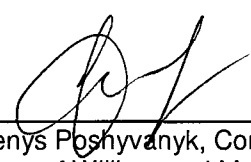


Yunlian Jiang

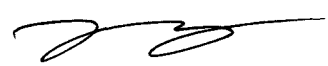
Approved by the Committee, July 2011

Committee Chair
Assistant Professor Xipeng Shen, Computer Science
The College of William and Mary

Associate Professor Weizhen Mao, Computer Science
The College of William and Mary



Assistant Professor Denys Poshyvanyk, Computer Science
The College of William and Mary



Associate Professor Haining Wang, Computer Science
The College of William and Mary

Senior Software Engineer Yaoqing Gao
IBM Canada Lab

ABSTRACT PAGE

In recent years, the increasing design complexity and the problems of power and heat dissipation have caused a shift in processor technology to favor Chip Multiprocessors In Chip Multiprocessors (CMP) architecture, it is common that multiple cores share some on-chip cache The sharing may cause cache thrashing and contention among co-running jobs Job co-scheduling is an approach to tackling the problem by assigning jobs to cores appropriately so that the contention and consequent performance degradations are minimized This dissertation aims to tackle two of the most prominent challengesin job co-scheduling

The first challenge is in the computational complexity for determining optimal job co-schedules This dissertation presents one of the first systematic analyses on the complexity of job co-scheduling Besides proving the NP completeness of job co-scheduling, it introduces a set of algorithms, based on graph theory and Integer/Linear Programming, for computing optimal co-schedules or their lower bounds in scenarios with or without job migrations For complex cases, it empirically demonstrates the feasibility for approximating the optimal schedules effectively by proposing several heuristics-based algorithms These discoveries facilitate the assessment of job co-schedulers by providing necessary baselines, and shed insights to the development of practical co-scheduling systems

The second challenge resides in the prediction of the performance of processes co-running on a shared cache This dissertation explores the influence on co-run performance *prediction imposed by co-runners, program inputs, and cache configurations Through a* sequence of formal analysis, we derive an analytical co-run locality model, uncovering the inherent statistical connections between the data references of programs single-runs and their co-run locality The model offers theoretical insights on co-run locality analysis and leads to a lightweight approach for fast prediction of shared cache performance We demonstrate the effectiveness of the model in enabling proactive job co-scheduling

Together, the two-dimensional findings open up many new opportunities for cache management on modern CMP by laying the foundation for job co-scheduling, and enhancing the understanding to data locality and cache sharing significantly

To my parents, my wife Zhen and my son Ruiyang.

Table of Contents

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support of many people. First and foremost I would like to thank my adviser Dr. Xipeng Shen. It has been my great honor and pleasure to be his Ph.D. student for the past five years. I sincerely thank him for the invaluable guidance, encouragement, and inspiration that he has given me over the course of my studies. I have always been encouraged by his passion and inspired by his keen wit.

Poshyvanyk, Denys

I would also like to thank my committee members, Dr. Yaoqing Gao, Dr. Wenzhen Mao, Dr. Denys Poshyvanyk and Dr. Haining Wang. I really appreciate their valuable time and efforts. They have given lots of feedback and suggestions that helped me improve the quality of this dissertation.

I am especially grateful to Dr. Yaoqing Gao, my mentor at IBM Toronto Lab. His patience as well as rich experience and knowledge have made my internships truly productive and memorable.

It is a pleasure to thank my colleagues and friends in William and Mary for their tremendous academic and personal support throughout past years. I enjoyed the pleasant research life with my collaborators, Eddy Zheng Zhang, Kai Tian, Feng Mao, Ziyu Guo. Thanks also go to Zhen Ren, Chuan Yue, Ningfang Mi, Bo Sheng, and many others, who have added tremendous enjoyment to my graduate life.

In addition, I am grateful to the staff in the Computer Science department for their assistance over the years. My Ph.D. journey would not have been this smooth without their help.

Finally, I would express my deepest gratitude to my family for their greatest love and support all these years. I am indebted to my parents for their love and belief. Above all, I would thank my wife Zhen for all the love and support through this journey.

List of Tables

List of Figures

Analysis and Approximation of Optimal Co-Scheduling on CMP

Chapter 1

Introduction

In a Chip Multiprocessors (CMP) system, multiple cores on a single chip typically share certain resource, including the last-level cache, off-chip pin bandwidth, and memory bus. The sharing, although shortening the communication among cores, causes resource contention among co-running jobs. Many studies have reported considerable, and sometimes significant, effects of the contention on program performance and system fairness [23, 50, 66, 10, 28, 78]. The urgency for alleviating the contention keeps growing as the processor-level parallelism increases continuously.

Recent years have seen many interests in using job co-scheduling to alleviate the contention [59, 23]. The basic strategy of job co-scheduling is to assign jobs to cores in a way that the overall influence from resource contention is reduced. Consider four jobs to run on a machine with two dual-core chips. As resource sharing between sibling cores is more intense than sharing across chips, different assignments of jobs to the four cores typically lead to different performance. Job co-scheduling helps find the appropriate job-core assignments to minimize the negative influence of resource contention.

Job co-scheduling consists of two tasks. The first is to estimate the influence of cache sharing on the performance of a job when it co-runs with other jobs. The second is to determine the suitable co-schedules based on the estimation.

For the first task, there are some studies that have tried to characterize the influence of cache sharing on program performance, most of them are either based on certain heuris-

tics (e.g., average access frequencies of cache sharers[1] [10]) or some hardware extensions (e.g., [49, 53]). Current treatments to cache sharing are primarily through runtime monitoring of low-level program behaviors (e.g., instructions per cycle (IPC), cache miss rates), no matter the goal is a better cache partition [63, 29, 28, 50] or an enhanced job scheduler [59, 60, 53, 23, 74, 20].

Unfortunately, the low-level behaviors, although easily obtainable, are often insufficient for those techniques to exert their full power. *First*, the observed behaviors are time-dependent and co-runner–dependent: They reflect the execution of the past, not the future, and they are for the co-run of a specific group of programs. As a result, one period of sampling is typically insufficient for predicting a program's cache requirement or co-run performance. Most existing cache management schemes address this issue by requiring many periodic samplings and reshufflings of co-run groups [23, 59]. This strategy leads to the *second* limitation: As a sampling phase only prepares for but does not really do cache-usage adjustment, the cache performance during those periods is often inferior, hence hurting the benefits from the optimizations considerably. *Finally and perhaps most importantly,* the reactive scheme is hard to scale up for many cases in shared-cache management, including job co-scheduling: The number of possible thread-to-core assignments increases exponentially in the problem size (e.g., 2 million different assignments for 16 jobs on 8 dual-cores), making it infeasible to determine the optimal schedule by sampling every possible co-run for a problem of reasonable size.

For the second task, most existing studies [10, 3, 59, 14, 23] rely on some simple techniques (for instance, trying a number of different co-schedules during runtime and choosing the best one.) The use of these techniques, although having shown interesting results, has left a comprehensive understanding of the determination of optimal co-schedules yet to achieve. This lack impairs the assessment of a co-scheduler—it is hard to tell how far the co-scheduling results depart from the optimum and whether further improvement would enhance the co-scheduler significantly—and hinders the development of job co-scheduling

[1] Cache sharers refer to the processes that run concurrently on a shared cache.

algorithms. Moreover, finding optimal co-schedules is critical for understanding how the various factors of CMP resource sharing affect program executions, as shown in a recent study [78].

The goal of this dissertation is to solve the two fundamental problems. First, we reveal the influence of cache sharing on program cache performance by uncovering the inherent connections between the locality of program single-runs (i.e. runs with no cache sharers) and that of their co-runs. Second, we explore the challenges on optimal co-scheduling of independent jobs (i.e., jobs with no data sharing among one another), we aim at answering three questions under different scenarios : 1) How difficult is it to find optimal co-schedules? 2) What algorithms can determine optimal co-schedules or reasonable lower bounds efficiently? 3)When the optimal are too hard to find, can heuristics-based algorithms approximate them effectively?

1.1 Definition of Problems

This section gives some basic definitions of the problem to be solved. First we describe the problem of the algorithm design and analysis for job co-scheduling. We then present the challenge on co-run performance prediction.

1.1.1 Job Co-Scheduling Algorithm Design

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

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

To make the problem tractable and meanwhile keep the analysis useful, we specify the following settings. Some of these settings may differ from certain practical scenarios.

However, as we will show (after presenting the settings) they do not prevent the use of the computed co-schedules from serving for its main goal facilitating the evaluation of practical co-schedulers

Machines The computing system assumed in this exploration contains m uniform chips, and each chip has u uniform cores[2] There is a certain amount of cache on each chip that is shared by the u cores on the chip Only one job can run on a core at each time point The execution speed of a job running on a chip depends on what jobs are placed on the same chip, but has negligible dependence on how the rest of the job set are placed on other chips The architecture is a generalized form of CMP architectures on the market, including the modern chips from Intel IBM, and so on

Jobs The number and starting time of jobs are set to be as follows The number of jobs (denoted as n) is equal to the number of cores $n = m * u$ This setting is to help focus on the placement of jobs on cores When $n < m * u$, the problem can be converted to the defined setting if we consider that there are $(m * u - n)$ extra dummy jobs that consume no resources If $n > m * u$, the problem is more complex, requiring the consideration of temporal complexity (e g context switch) besides the spatial placement of jobs The temporal complexity is out of the scope of this dissertation But we note that this work will be still useful for that setting, as spatial placement still exists as a sub problem in it All the jobs must start at the same time This is a typical assumption in traditional job scheduling [38]

Job Migrations A job can migrate from one core to another, but the migration only happens when any of the jobs terminates This setting comes from the following reason As well known, keeping a process on a processor is good for locality As a result, in practical systems like Linux, occurrences of job migrations are mostly triggered by load imbalance [2] In our setting, as the number of jobs equals the total number of cores, load changes only when some job finishes Therefore, allowing job migration only at those times does not cause large departure from real scenarios

[2]We use the term cores for simplicity of discussion The techniques can also be applied to thread scheduling in SMT systems

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

Criteria There are two criteria to evaluate the quality of a schedule, namely co-run cost and makespan.

- Co-run Cost Cost refers to the total running times of all the jobs to be scheduled. Minimizing the co-run cost means the computing efficiency of the CMP system is maximized. This criterion is critical when the throughput is the key requirement of a system.

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

Performance Data. The performance degradation of a job when it co-runs with other k ($0 < k < u$) jobs is a piece of critical information for job co-scheduling. We devote half of this dissertation (Chapter 3) to prediction of co-run degradations.

In our discussion of the complexity and algorithms for optimal co-scheduling (Chapter 2), however, we assume that the co-run degradation is known beforehand. This assumption helps us concentrate on the algorithm design.

Because a program execution may vary constantly, the performance degradation of a program in a co-run may vary across intervals. In our setting, we use the average degradation through the entire co-run. A future enhancement is to combine with program phase analysis [56, 57]. As previous studies do [59], we currently ignore phase changes to concentrate on co-scheduling itself.

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

1.1.2 Co-Run Performance Prediction

In a CMP system, the shared resource (e.g., last level cache, off-chip bandwidth) contention can often cause performance degradation. A program typically runs slower when it co-runs other programs in a CMP system than it runs alone. The co-run performance of a program is the running speed when it co-runs with some other programs in a given CMP system. In this work, we explore the techniques to predict the co-run performance before the programs actually run in the CMP system. The co-run performance prediction is fundamental to job co-scheduling because only if the scheduler obtains the co-run performance of all the programs can it make the appropriate schedule.

Co-Run performance prediction is a challenge because it involves many factors. Generally speaking, the performance of a program is decided by the following factors: program code, program inputs, system configurations, runtime environment, underlying architectures and so on. In our work, we assume that the underlying architecture is known, we only consider the influence of program inputs and its co-runners.

Currently a program's performance is heavily affected by its memory behaviors. In our work, we focus on the memory behaviors of a program in a CMP system such as last level cache miss rate and so on.

1.2 Contributions

The contributions of this dissertation are summarized as follows.

- *Algorithms.* We analyze the computational complexities of job co-scheduling, and create a set of co-scheduling algorithms, both optimal ones and heuristic ones.

- *Performance prediction.* By combining program behavior analysis with cache management, we propose a locality-based model for program co-run performance prediction on CMP.

- *Scheduler construction.* We design and evaluate a number of job co-schedulers for minimization of both co-run cost and makespan.

1.2.1 Algorithm Design

To get the optimal job co-scheduling is important for both theoretical analysis and practical co-scheduling policy. Getting the optimal schedule can help evaluate current schedulers. By comparing the performance different between optimal and current schedule, one can decides whether further improvement would enhance the co-scheduler significantly, and hinder the development of co-scheduling algorithms. Moreover, finding optimal co-schedules is critical for understanding how the various factors of CMP resource sharing affect program executions, as shown in a recent study [78]. In this dissertation, we propose algorithms to schedule jobs on CMP system under different scenarios.

- *Complexity Analysis.* We analyze the computational complexities of job co-scheduling under different scenarios (e.g. number of cores in one chip, different criteria, job migration allowed or not). We have proved that if job migration is not allowed and there are only two cores in one chip, the problem is polynomial solvable. If the number of cores in one chip is greater than 2, the problem is NP-Complete no matter job migration is allowed or not.

- *Algorithms for Dual-Core Systems* We have adopted the classic Blossom [19] algorithm on degradation graphs for finding optimal schedules for multi-socket dual-core sysstems for the minimization of both makespan and co-run cost We further reduce the complexity of makespan minimization problem from $O(n^4)$ to $O(n^{2.5} \log n)$

- *Approximation Algorithms* Because of the computational complexities, we have proposed heuristic algorithms for the cases where the optimal solution is hard to obtain The different optimization objectives stimulate different designs in the approximation algorithms for both makespan minimization and cost minimization Our designed algorithms include search based algorithms (e g , A*-cluster), graph based algorithms (e g , hierarchical perfect matching), and local-optimal algorithms (e g , greedy algorithm, local optimizations)

1.2.2 Performance Predictive Models

There have been some prior studies on predicting co-run performance of processes on CMP Some of them propose architecture extensions to facilitate runtime cache performance monitoring [63, 29, 28 50] Some rely on periodical process reshuffling by OS to sample performance of a process under various co-run settings [59, 60, 53, 23, 74, 20] These techniques are primarily based on low-level program behaviors (e g , instructions per cycle (IPC), cache miss rates) obtained through runtime monitoring Unfortunately, the low-level behaviors, although easily obtainable, are often insufficient for accurate, scalable, and large-scope performance prediction (elaborated in Section 3)

In this dissertation, we address co-run performance prediction by exploiting data locality analysis at the program level Our approach centers on the development of two locality models

- *Inclusive Reuse Signatures* We introduce the concept of *inclusive reuse signature*, which is a summary of LRU stack distances on a shared cache with all cache sharers' data references considered The model, through rigorous analysis, uncovers the

statistical connections between co-run locality and the locality of single runs, thus laying the foundation for approximating co-run cache miss rates from program locality analysis.

- *Sensitivity and Competitiveness* In light of inclusive reuse signatures, we develop a lightweight model for efficiently predicting co-run data locality (or cache usage) from the memory reference patterns of the programs' single runs. This model offers a simple, efficient way to characterize the statistical expectation of the influence that a process may impose on and receive from random co-runners. The high efficiency of the model is the key to practical uses in shared-cache management. It achieves the efficiency mainly by capitalizing the connection between time and locality.

1.2.3 Cache-Contention-Aware Proactive Scheduler

Designing an effective shared-cache-aware scheduler is challenging. It requires the ability to understand a program's demand and sensitivity to the shared cache. Moreover, the online system requires extremely low runtime overhead. In this dissertation, we propose a proactive online scheduler to schedule jobs onto CMP system online.

- *Offline Scheduler* We integrate the techniques proposed in Chapter 2 and Section 3 into a batch scheduler. Given a set of jobs, it first predicts the co-run performance degradations based on program behavior analysis and then uses algorithms proposed in Chapter 2 to find the best schedule.

- *Online Scheduler* To satisfy the time constraint of online scheduling, we, based on inclusive reuse distances, propose a model to predict a program's requirement to the shared cache. The model provides an efficient way to characterize the sensitivity of a process on cache contention and its potential influence on its co-runners. The predictive model is lightweight enough for online uses. Based on the sensitivity of programs, online schedulers partition programs into a sensitive group and an insensitive group.

It pairs sensitive jobs with insensitive jobs to improve the thoughput of the whole system.

1.3 Dissertation Organization

The dissertation is organized as follows. Chapter 2 describes our findings in the computational complexity and algorithms of optimal co-scheduling on CMP. Chapter 3 presents co-run performance predictive models. Chapter 4 concentrates on the construction of batch and online co-schedulers. Chapter 5 briefly summarizes some of our other efforts for enhancing the computational efficiency on CMP. And finally, Chapter 6 concludes this dissertation.

Chapter 2

Complexity Analysis and Algorithm Design

2.1 Introduction

In modern Chip Multiprocessors (CMP) architecture, it is common that multiple cores share certain levels of on-chip cache and off-chip bandwidth. As many studies have shown [23, 22, 50, 66, 10, 28], the sharing causes resource contention among co-running jobs, resulting in considerable and sometimes significant degradations to program performance and system fairness. Job co-scheduling is one of the approaches to addressing the contention problem. Its strategy is to assign jobs to computing units in a way that the overall influence from resource contention is minimized.

Unlike approaches proposed in architecture design (e.g., cache partition [49, 50, 28]), job co-scheduling can typically be implemented without hardware extensions. It has drawn many research interests in recent years, with a number of co-scheduling schemes developed [59, 23. 14]. Most of the techniques use reactive co-scheduling. The runtime system periodically changes the co-runners (i.e., the jobs sharing a cache) of a job to estimate its cache requirement (e.g., [23]) or co-run performance (e.g., [59]). The scheduler then changes the assignment of the jobs accordingly to group compatible jobs to the same chip

to reduce cache contention. Besides reactive scheduling, some research tries to predict co-run performance of programs (e.g., [37, 34]), which opens the opportunities for proactively co-scheduling jobs without the need for runtime trials.

Even though those schemes have shown effectiveness in alleviating co-run contention caused by cache sharing, efficiently finding optimal co-schedules or a good lower bound remains an open question. Answering this question is important in two aspects. First, it facilitates the evaluation of various co-scheduling systems. Without knowing optimal co-schedules, it is hard to precisely determine how good a co-scheduling system is—how far the co-scheduling results are from optimal co-schedules and whether further improvement would enhance system performance significantly. Second, optimal co-scheduling algorithms produced in answering that question can directly benefit the development of some co-scheduling schemes, especially proactive co-scheduling schemes [37, 30, 32]. These schemes co-schedule jobs based on the prediction of co-run performance, rather than through dynamically trying and measuring various co-schedules and picking the best as most other (reactive) schemes do. Therefore for proactive schemes, efficient optimal co-scheduling algorithms may simply serve as their co-scheduling algorithms, or as the base for the development of lightweight co-scheduling algorithms.

This chapter presents a systematic exploration towards optimal job co-scheduling on CMP. It aims at answering the following three questions:

- How difficult is it to find optimal co-schedules?
- What algorithms can determine optimal co-schedules or reasonable lower bounds efficiently?
- When the optimal are too hard to find, can heuristics-based algorithms approximate them effectively?

There are two different criteria in Job co-scheduling, namely co-run cost and makespan. The different optimization goals lead to different solutions. We first tackle the problem on cost minimization.

Our exploration on co-run cost minimization consists of two components. The first component is focused on the complexity of co-scheduling in a basic setting where no job length variance or job migrations are considered. The discoveries fall in four aspects. The first is a polynomial-time algorithm for finding optimal co-schedules on dual-core CMPs. The algorithm constructs a degradation graph, models the optimal scheduling problem as a *minimum-weight perfect matching problem*, and solves it using the Blossom algorithm [19]. The second is a proof that optimal co-scheduling on u-core processors is an NP-complete problem when u is greater than 2, with or without job migrations allowed.[1] The third is an Integer Programming (IP) formulation of the optimal co-scheduling problem for u-core systems ($u > 2$). The formulation offers a clean way for formal analysis; its Linear Programming (LP) form offers an efficient approach to computing lower bounds for job co-scheduling. The final is a series of heuristics-based algorithms for approximating the optimal schedules in u-core CMP systems ($u > 2$). The first algorithm, named the hierarchical matching algorithm, generalizes the dual-core algorithm to partition jobs in a hierarchical way. The second algorithm, named the greedy algorithm, schedules jobs in order of their sensitivities to cache contention. To further enhance the scheduling quality, we develop an efficient local optimization scheme that is applicable to the schedules produced by both algorithms.

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

Makespan minimization differs from cost minimization. The optimal schedules for the two criteria are typically different. In traditional job scheduling literature, the two criteria have led to drastically different algorithms and complexity analyses [38]. As to be shown

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

later, for multicore job co-scheduling, the implication of their differences is pronounced as well. The differences exist in every major aspect, from complexity analysis to algorithm design to the ultimate scheduling results.

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

- First, we prove that makespan minimization in job co-scheduling is NP-complete on systems with more than 2 cores per chip. The proof is based on a reduction from the problem of Exact Cover by 3-Sets. We are not aware of any previous analysis of the computational complexity.

- Second, by offering an $O(n^{2.5} \cdot \log n)$ algorithm (n is the number of jobs), we prove that on dual-core systems with no job migrations, the problem is polynomial-time solvable. To the best of our knowledge, this algorithm is the first polynomial-time solution for this optimal co-scheduling problem.

Finally, we evaluate the algorithms on both real and synthetic problems, verifying the optimality of the solutions produced by the co-scheduling algorithms (under certain conditions). Results of the heuristic algorithms demonstrate their capability to achieve near optimal solutions with reasonable scalability in different scenarios: Compared to sharing-oblivious scheduling, they reduce co-run degradation by 5–20% on average, 1.4% away from the optimum.

There has been a large body of research on optimal job scheduling. But to our surprise, despite an extensive survey [38, 12], we have found no previous work that tackles an optimal co-scheduling problem containing performance interplay among jobs as what the current co-scheduling problem involves. This work, although uncovering some interesting facts, is by no means to answer all questions on optimal job co-scheduling. Instead, it hopefully may serve as a trigger to stimulate further studies towards a comprehensive understanding to this intricate problem.

The rest of this chapter is organized as follows. Sections 2.2, Section 2.4 and Section 2.4 present the complexity analysis and algorithm design of the cost minimization problem without job migration. Section 2.5 describes our exploration on job co-scheduling with migrations. Section 2.6 reports our solutions for the makespan minimization problem. Section 2.7 evaluates our proposed algorithms on both real schedule problems and a set of synthetic problems. Section 2.9 reviews some related and Section 2.10 summarizes this chapter.

2.2 Definition of the Basic Min-Cost Co-Scheduling Problem

This section defines the basic min-cost co-scheduling problem, which concentrates on the primary challenges in assigning jobs to cores without the considerations of the complexities caused by job migrations. Section 2.5 will describe treatment of those complexities.

The problems discussed in this chapter concentrates on independent jobs—no jobs have data shared with each other. Cache contention and the associated contention on memory controllers and bus are the only effects of shared cache on the performance of co-running programs. Hence, co-running programs typically run slower than their single runs (i.e. the runs with no co-runners) due to resource contention. This kind of performance degradation is called co-run degradation. Formally, the *co-run degradation* of a job i when it co-runs with all the jobs in set S is defined as

$$d_{i,S} = \frac{cCPI_{i,S} - sCPI_i}{sCPI_i}$$

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

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

The basic optimal co-scheduling problem is as follows:

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

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

The problem of co-scheduling includes two parts. The first is to predict the degradation of every possible co-run. The second is to find the optimal schedules so that the total degradation is minimized given the predicted co-run degradations. Much research has explored the first part of the problem (e.g., [23, 37, 59]). This work specially focuses on the second part, in which, we assume that the degradations of all possible co-runs are known beforehand (although some algorithms to be presented do not require the full knowledge).

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

such assessment, it is usually acceptable to collect the co-run performance of some jobs offline even if that takes some amount of time.

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

In this basic co-scheduling problem, the co-schedule to be found is static, meaning that there are no job migrations during the execution of a job.

In the following description, we use *an assignment* to refer to a group of u jobs that are to run on the same chip. (The influence on the performance of a job imposed by the assignments of jobs on other chips is typically small and neglected in job co-scheduling.We use *a schedule* to refer to a set of assignments that cover all the jobs and have no overlap with each other—that is, a schedule is a solution to a co-scheduling problem.

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

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

We model optimal co-scheduling problems in this case as a graph problem. The graph is a fully connected graph, named *degradation graph*. As illustrated in Figure 2.1, every vertex in the graph represents a job, and the weight on each edge equals the sum of the degradations of the jobs represented by the two vertices when they run on the same chip.

With this modeling, the optimal co-scheduling problem becomes a *minimum-weight perfect matching* problem. A *perfect matching* in a graph is a subset of edges that cover all vertices, but no two edges share a common vertex. A minimum-weight perfect matching problem is to find a perfect matching that has the minimum sum of edge weights in a graph.

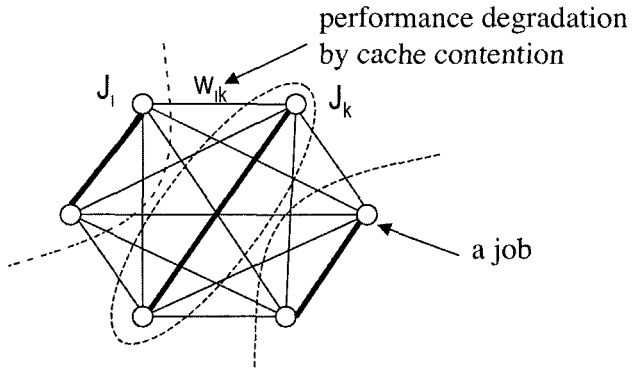


Figure 2.1: An example of a degradation graph for 6 jobs on 3 dual-cores. Each partition contains a job group sharing the same cache. Bold edges compose a perfect matching

It is easy to prove that a minimum-weight perfect matching in a degradation graph corresponds to an optimal co-schedule of the job set represented by the graph vertices. First, a valid job schedule must be a perfect matching in the graph. Each resulting job group corresponds to an edge in the graph, and the groups should cover all jobs and no two groups can share the same job, which exactly match the conditions of a perfect matching. On the other hand, a minimum-weight perfect matching minimizes the sum of edge weights, which is equivalent to minimizing the objective function of the co-schedule defined in Section 2.2.

One of the fundamental discoveries in combinational optimization is the polynomial-time *blossom* algorithm for finding minimum-weight perfect matchings proposed by Edmonds [19]. It offers the polynomial-time solution to optimal co-scheduling on dual-cores. The time complexity of the algorithm is $O(n^2m)$, where n and m are respectively the numbers of nodes and the number of edges in the graph. Later, Gabow and Tarjan develop an $O(nm + n^2 \log n)$ algorithm [24]. Cook and Rohe provide an efficient implementation of the blossom algorithm [11], which is used in this work.

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

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

2.4.1 Proof of the NP-Completeness

This section proves that when $u \geq 3$, optimal co-scheduling becomes NP-complete. The proof is via a reduction of *Multidimensional Assignment Problem* (MAP) [25], a known NP-complete problem, to the co-scheduling problem.

First, we formulate the co-scheduling problem as follows. There is a set S containing n elements. (Each element corresponds to a job in the co-scheduling problem.) Let S_u represent the set of all u-cardinality subsets of S. Each of those u-cardinality subsets, represented by G_i, has a weight w_i, where $i = 1, 2, \cdots, \binom{n}{u}$. ($G_i$ corresponds to a group of jobs scheduled to the same chip, and its weight corresponds to the sum of the degradation of all the jobs in the group.) The objective is to find n/u such subsets, $G_{p_1}, G_{p_2}, \cdots, G_{p_{n/u}}$ to form a partition of S that satisfies the following conditions:

- $\bigcup_{i=1}^{n/u} G_{p_i} = S$. (Every job belongs to a subset.)
- $\sum_{i=1}^{n/u} w_{p_i}$ is minimized. (Total weight is minimum.)

The first condition ensures that every job belongs to a single subset and no job can belong to two subsets (as all the subsets together contain only $(n/u) * u = n$ jobs). The second condition ensures that the total weight of the subsets is minimum.

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

- Input: u ($u \geq 3$) sets Q_1,Q_2,$\cdots$,Q_u, each containing m elements, a cost function C: $Q_1 \times Q_2 \times \cdots \times Q_u \rightarrow R$, and a given value O.

- Output: An assignment A that consists of m subsets, each of which contains exactly one element of every set Q_k, $1 \leq k \leq u$. Every member $\alpha_i = (a_{i1}, a_{i2}, \ldots, a_{ik})$ of A has a cost $c_i = C(\alpha_i)$, where $1 \leq i \leq m$ and a_{ik} is the element chosen from the set Q_k.

- Constraints: Every element of Q_k, $1 \leq k \leq u$, belongs to exactly one subset of assignment A and $\sum_{i=1}^{m} c_i$ is equal to the given value O.

MAP has been proven to be NP-complete by reduction from the three-dimensional matching problem [25], a well-known problem first shown to be NP-complete by R. Karp [36].

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

- Let $S = \bigcup_{k=1}^{u} Q_k$ and $n = m * u$.

- Build all the u-cardinality subsets of S, represented as G_i, $1 \leq i \leq \binom{n}{u}$. If a subset G_i contains exactly one element from every set Q_k, $1 \leq k \leq u$, its weight is set as $C(a_1, a_2, \cdots, a_u)$, where C is the cost function in the MAP instance, and a_k is an element chosen from Q_k. Otherwise the weight is set to positive infinity.

For a given value of u, the time complexity of the construction is $O(n^u)$. It is clear that a solution to this co-scheduling problem is also a solution to the MAP instance and vice versa. This proves that the co-scheduling problem is NP-hard. Obviously, the co-scheduling problem is an NP problem. Hence, the co-scheduling problem is an NP-complete problem when $u \geq 3$.

2.4.2 Integer/Linear Programming for Optimal Co-Scheduling

The NP-completeness suggests that it is difficult if not impossible to generalize the algorithm described in Section 2.3 into a polynomial-time algorithm for the cases when u is greater than two. This section shows that optimal co-scheduling can be formulated as an IP problem in general, and therefore many standard IP solvers may be used to compute the optimal

co-schedules directly. Furthermore, the LP relaxed form offers an efficient way to compute the lower bounds of the co-scheduling for an arbitrary u value.

2.4.2.1 Integer Programming Model

The IP formulation comes from the observation that optimal job co-scheduling defined in Section 2.2 is essentially a partition problem: To find a way to partition the n jobs into $m = \frac{n}{u}$ sets (corresponding to the m chips), with each job falling into one set and each set containing exactly u jobs, so that the total co-run degradation of all the jobs is minimized. We formulate it as the following IP problem.

The **variables** of the IP are:

x_{S_i}, where $1 \leq i \leq \binom{n}{u}$ and $S_i \subseteq \{1, 2, ..., n\}$ with $|S_i| = u$, and $S_i = S_j$ if and only if $i = j$ $(1 \leq i, j \leq \binom{n}{u})$.

Each x_{S_i} is a binary variable, indicating whether the job set S_i is one of the sets in the final partition result.

The **objective function** is:

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

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

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

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

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

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

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

2.4.2.2 Computing Lower Bounds in Polynomial Time

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

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

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

In our experiment, we employ the LP and IP solver in MATLAB to compute optimal co-schedules and the lower bounds. The LP solver, function *linprog*, is based on LIPSOL [75], which is a variant of Mehrotra's predictor-corrector algorithm [47], a primal-dual interior-point method. The IP solver, *bintprog*, uses a LP-based branch-and-bound algorithm to solve binary integer programming problems.

2.4.3 Heuristics-Based Approximation

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

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

2.4.3.1 Hierarchical Perfect Matching Algorithm

The hierarchical perfect matching algorithm is inspired by the solution on dual-core systems. For the purpose of clarity, we first describe the way this algorithm works on quad-core CMPs, and then present the general algorithm.

Finding the optimal co-schedule on quad-core CMPs is to partition the n jobs into $n/4$ 4-member groups. In this algorithm, we first treat a quad-core chip with a shared cache of size L as two virtual chips, with each containing a dual-core processor and a shared cache of size $L/2$. On the virtual dual-core system, we can apply the perfect matching algorithm to find the optimal schedule, in which, the job set is partitioned into $n/2$ pairs of jobs. Next, we create a new degradation graph, with each vertex representing one of the job pairs. After applying the minimum-weight perfect matching algorithm to the new graph, we will obtain $n/4$ pairs of job pairs, or in another word, $n/4$ 4-member job groups. These groups form an approximation to the optimal co-schedule on the quad-core system.

Using this hierarchical algorithm, we can approximate the optimal solution of u-core co-scheduling problem by applying the minimum perfect matching algorithm $\log(u)$ times, as shown in Figure 2.2. At each level, say level-k, the system is viewed as a composition of 2^k-core processors. At each step, the algorithm finds the optimal coupling of the job groups that are generated in the last step. Figure 2.3 shows the pseudo-code of this algorithm. Notice that, even though this hierarchical matching algorithm invokes the minimum-weight perfect matching algorithm $\log(u)$ times, its time complexity is the same as that of the basic minimum perfect matching algorithm, $O(n^4)$, because the number of vertices in the degradation graphs decreases exponentially.

2.4.3.2 Greedy Algorithm

The second heuristics-based algorithm is a greedy algorithm. Our initial design is as follows. We first sort all of the u-cardinality sets of jobs in the ascending order of the total degradation of the jobs in a set when they co-run together. Let S represent the final schedule, whose initial content is empty. We repeatedly pick the top set from the sorted order, none

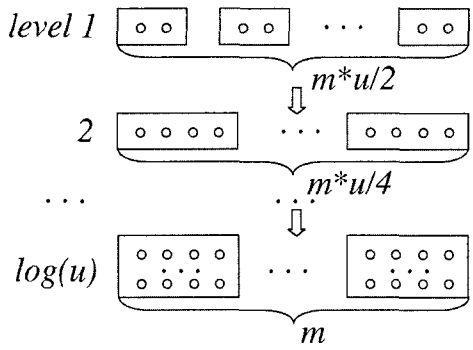


Figure 2.2: The hierarchical view of a CMP system used in the hierarchical perfect matching algorithm. Each box represents a virtual chip except the chips on the last level, which are real chips. Each circle represents a core. (u: the number of cores per real chip; m: the total number of real chips in the system.)

of whose members is covered by S yet, and put it into S until S covers all the jobs. This design is intuitive—every time, the co-run group with minimum degradation is selected. However, the result is surprisingly inferior—the produced schedules are among the worst possible schedules. We call this algorithm the naive greedy algorithm.

After revisiting the algorithm, we recognize the problem. Compared to other jobs, a job that uses little shared cache tends to be both "polite"—causing less degradation to its co-runners, and "robust"—suffering less from its co-runners. We call such a job a "friendly" job. Because of this property, the top sets in the sorted order are likely to contain only those "friendly" jobs. After picking the first several sets, the naive greedy algorithm runs out of friendly jobs, and has to pick those sets whose members are, unfortunately, all "unfriendly" jobs, causing the large degradation in the final schedule.

We observe that if we assign "unfriendly" jobs with "friendly" ones, the "friendly" jobs won't degrade much more than they do in the naive greedy schedule, whereas, the "unfriendly" programs will degrade much less.

This observation leads to the following improved algorithm. We first compute the *politeness* of a job, which is defined as the reciprocal of the sum of the degradations of all co-run groups that include that job. During the construction of the schedule S, each time,

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

Figure 2.3 Hierarchical minimum-weight perfect matching.

we add a co-run group that satisfies the following two conditions: 1) It contains the job whose *politeness* is the smallest in the unassigned job list; 2) its total degradation is minimum. Figure 2.4 shows the pseudo-code of this algorithm. This politeness-based greedy algorithm manages to assign "unfriendly" jobs with "friendly" ones and proves to be much better than the naive greedy algorithm.

The major overhead in this greedy algorithm includes the calculation of politeness and the construction of the final schedule. Both have $O(n\binom{n}{u})$ time complexity, so the greedy algorithm's time complexity is $O(n\binom{n}{u})$

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

Figure 2.4 Greedy Algorithm

2.4.3.3 Local Optimization

Local optimization is a post-processing step for refining the schedules generated by both heuristics-based algorithms For a given schedule, the algorithm optimizes each pair of assignments in the schedule For each pair, the algorithm enumerates all possible ways to evenly partition the jobs contained in them into two parts Each partition corresponds to one assignment for those jobs, and the one that minimizes the sum of co-run degradations of those jobs is taken as the final schedule for that pair Figure 2 5 shows the pseudo-code

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

2.5 Optimal Co-Scheduling with Migrations

With the understanding of the basic optimal co-scheduling problem, this section expands the scope of the problem to include job migrations into account In this case, jobs may finish

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

Figure 2.5: Local Optimization

at different times, and rescheduling of the unfinished jobs may be necessary when some job terminates and vacates a core. We call each scheduling or rescheduling *a scheduling stage*. This work concentrates on the settings where rescheduling happens only when a job finishes; there are at most n scheduling stages for n jobs.

Some terminology needs to be redefined in this setting. *An assignment* still refers to a group of K jobs that are to run on the same chip. We use *a sub-schedule* to refer to a set of assignments that cover all the unfinished jobs and have no overlap with one another. *A schedule* still refers to a solution to the co-scheduling problem. However, a schedule becomes a set of sub-schedules that have been used from the start of the jobs to the finish of the final job. Considering job length variances, we redefine the goal of the co-scheduling as to find a schedule that minimizes the total execution time of all jobs [3], expressed as

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

where, $cT_i^{(S)}$ is the time job i takes to finish in a co-schedule S. Other settings of the problem remain the same as those described in Section 2.2.

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

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

2.5.1 Co-Schedule Space

We model the optimal co-scheduling problem as a tree-search problem as shown in Figure 2.6. For n jobs, there are at most n scheduling stages; each corresponds to a time point when one job finishes since the previous stage. Every node in the tree, except the starting node, represents a sub-schedule of the remaining jobs. The nodes at a stage, say stage i, correspond to all possible sub-schedules for $n - i + 1$ remaining jobs. There is a cost associated with each edge, equal to the total execution time spent by all jobs between the two stages connected by the edge. Let $node_2$ represent a child of the node $node_1$. Given the state at $node_1$, we assign the unfinished jobs to cores according to the sub-schedule contained in $node_2$; let t be the time required for the first remaining job to finish; the cost on the edge from $node_1$ to $node_2$ is $t * m$, where m is the number of jobs that are alive during that period of time.

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

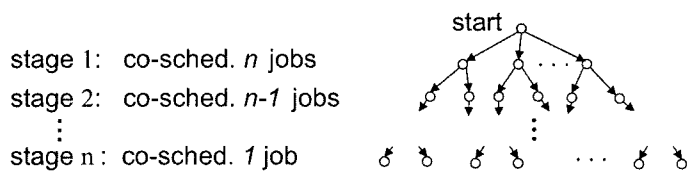


Figure 2.6: The search tree of optimal job co-scheduling with rescheduling allowed at the end of a job. Each node in the tree, except the starting node, represents a sub-schedule of the remaining jobs. Each edge represents one schedule of the unfinished jobs.

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

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

2.5.2.1 A*-Search Algorithm

A*-search is an algorithm stemming from artificial intelligence [51] for fast graph search. It has been used for many search problems, but not for job co-scheduling. This section presents the basic algorithm of A*-search, and the next section will describe the special challenges in applying A*-search to job co-scheduling.

A*-search is appealing in several aspects. It guarantees the optimality of its search results, and meanwhile, effectively avoids visiting certain portion of the search space that contain no optimal solutions. In fact, it has been proved that A*-search is optimally efficient for any given heuristic function. That is, for a given heuristic function, no other optimal algorithm is guaranteed to expand fewer nodes than A*-search [51]. Its completeness, optimality, and optimal efficiency trigger our exploration of using it for job co-scheduling.

We use Figure 2.6 to explain the basic algorithm of A*-search. In the graph, there is a cost associated with every edge. The objective is to find the cheapest route in terms of the total cost from the starting node to a goal node. In A*-search, each node, say node d, has two functions, denoted as $g(d)$ and $h(d)$. Function $g(d)$ is the cost to reach node d from the starting node. Function $h(d)$ is the estimated cost of the cheapest path from d to a goal node. So, the sum of $g(d)$ and $h(d)$, denoted as $f(d)$, is the estimated cost of the cheapest route that goes from the start to the goal and passes through node d.

Often, the graph to be searched through is conceptual and does not exist at the beginning of the search. During the search process, the A*-search algorithm incrementally creates the portion of the graph that possibly contains optimal paths. Specifically, A*-search uses a priority list to decide the next node to expand (i.e., to create its children nodes). Initially, the priority list contains only the starting node. Each time, A*-search removes the node with the highest priority from the top of the priority list and expands that node. After an

expansion, it computes the $f(d)$ values of all the newly generated nodes, and inserts them into the priority list according to their priority values, which are computed as $1/f(d)$. Such expansions continue until the top of the priority list is a goal node, a sign indicating that an optimal path has been found. The algorithm terminates. The use of the $f(d)$-based priority list is the key for A*-search to avoid unnecessary expansions without sacrificing the optimality of the search result.

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

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

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

2.5.2.2 A*-Search–Based Job Co-Scheduling

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

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

In this work, we resort to Linear Programming for defining $h(d)$. Suppose at node d there are U unfinished jobs. We define $h(d) = T_s + T_{deg}$, where T_s is the time the U jobs need

[4]We assume that the search is a tree search. There are some subtle complexities for other types of search [51].

to finish their remaining parts if they each run alone, and T_{deg} is the estimated minimum of the total degradation of the U jobs during their execution from the node d to any child of d.

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

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

The value of T_{deg} should be the minimum of d_j of all sub-schedules of the node d. To determine the sub-schedule that has the smallest d_j, we need to find the values of b_i so that $\sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(d)}$ is minimized under the constraint $\sum b_i = U$. This analysis leads to an Integer Linear Programming problem shown in Figure 2.7. By relaxing the constraint on x_i to $0 \leq x_i \leq 1$, the problem becomes a Linear Programming problem, which can be solved efficiently using existing tools [1].

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

$$T_{deg} = 2 * (U - I) * r_{2_{min}} * T_{min}(d). \tag{2.1}$$

Definitions:

U number of unfinished jobs,

I number of chips,

K cores per chip, $I < U \leq I * K$,

$r_{x_{m\ n}}$ minimum degradation rate,

$T_{min(d)}$ minimum single run time

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

The number of jobs on the c-th chip

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

Objective function:

$$min \sum_{c=1}^{I} m(c) * r_{m(c),\ i} \quad * T_{min(d)}$$

Linear constraint:

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

Figure 2.7 Integer Linear Programming for computing T_{deg} the lower bound of degradation

The intuition for the formula is that in any sub-schedule of this scenario there must be at least $(U - I)$ chips that have a pair of the unfinished jobs assigned Otherwise, some chips must have more than two jobs assigned, which is not allowed in the problem setting (Section 2 2) The application of the definition of T_{deg} to such a sub-schedule leads to Equation 2 1

2.5.3 Heuristics-Based Estimation

A limitation of A*-search based algorithms is its high requirement for memory space It keeps all open nodes in the priority list, while the number of open nodes grows in exponential to the problem size in job co-scheduling

In this section, we describe two heuristics-based algorithms for solving the optimal co-scheduling problem in a scalable manner One algorithm the A*-cluster algorithm, integrates clustering into the A*-search-based algorithm, the other algorithm, the local-matching algorithm, is a generalized version of the graph-matching-based co-scheduling algorithms mentioned in Section 2 4 3 1

2.5.3.1 A*-Cluster Algorithm

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

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

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

The integration of clustering into A*-search is implemented inside procedure *nextSubSchedule()* (invoked in the middle of procedure *Astar()*) as shown in Figure 2.8. 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 co-run 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.

In the algorithm, the starting node needs a special treatment: The procedure *nextSubSchedule()* skips the state update step as there are no sub-schedules yet.

The first strategy reduces the height of the search tree, while the second reduces the width. Together, they reduce the number of nodes at a stage significantly. 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 a given C and K C is the number of clusters).

Although there are many clustering methods (e.g., K-means, hierarchical clustering [26]), we use a simple distance-based clustering approach because the data to be clustered—the

```
/* 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 2.8 Integration of clustering into A* search algorithm for approximation of optimal co-scheduling

job lengths—are one-dimensional and the number of clusters is unknown beforehand Given a sequence of data, the distance-based clustering first sorts the data in ascending order It then computes the differences between every two adjacent data items in the sorted sequence Large differences indicate cluster boundaries A difference is considered large enough if its value is greater than $m + \delta$, where, m is the mean value of the differences in the sequence and δ is the standard deviation of the differences An example is as follows

times to finish	10	15	18		32	35		51	53	56
differences		5	3	14	3		16	2	3	
job clusters	x	$\bar{x}$	x		x	x		x	$\bar{x}$	x

mean difference = 6 5, std = 5 9

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

2.5.3.2 Local-Matching Algorithm

For even higher efficiency, we design a second approximation algorithm, which explores only one path from the root to the goal in Figure 2 6 At each scheduling point it selects the

schedule that minimizes the total running time of the remaining part of the unfinished jobs under the assumption that no reschedules would happen. The assumption leads to local optimum at each scheduling stage.

The key component of the algorithm is the procedure to compute the local optimum. This step is the same as the basic job co-scheduling problem discussed in Section 2.2, except that the number of jobs may be smaller than the number of cores as some jobs may have terminated. We take a simple strategy to handle this case: treating the jobs that have finished as pseudo-jobs, which exist but consume no computing resource. Therefore, if the co-runners of a job are all pseudo-jobs, that job has no performance degradation at all. As the pseudo-jobs have to be scheduled every time, this strategy introduces some redundant computation. However, it provides an easy way to generalize the perfect matching algorithm described in Section 2.3 and 2.4. Apparaently, the time complexity of the local-matching algorithm is $O(n^5)$: The co-scheduling algorithm on a stage has complexity of $O(n^4)$, and there are n stages.

2.6 Makespan Minimization

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

Makespan minimization differs from cost minimization. The optimal schedules for the two criteria are typically different. for multicore job co-scheduling, the implication of their

differences is pronounced as well. The differences exist in every major aspect, from complexity analysis to algorithm design to the ultimate scheduling results.

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

2.6.1 NP-Completeness ($u \geq 3$, With or Without Job Migration)

When more than two cores share a cache on a chip ($u \geq 3$), the makespan minimization is an NP-complete problem. We prove this result by reducing a known NP-complete problem, *Exact Cover by 3-Sets* (X3C) [27], to our problem.

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

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

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

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

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

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

In the proof, it is assumed that u, the number of cores on each chip, is an input parameter of the co-scheduling problem. When u is a treated as a constant, i.e., fixed once the hardware is chosen, an extra step is necessary to prove the NP-completeness when u is greater than 3. In that scenario, the known NP-complete problem, Exact Cover by k-Set, can be used in the reduction to the co-scheduling problem with any fixed u.

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

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

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

Before describing the algorithm, we introduce the concept of a perfect matching. A *perfect matching* in a graph is a subset of edges that cover all vertices of the graph, but no two edges share a common vertex. We define the *bound* of a perfect matching as the largest weight of all the edges it covers. It is easy to see that the perfect matching of a co-

run makespan graph with the minimum bound corresponds to a solution to the makespan minimization problem: Each edge corresponds to an assignment (i.e., co-run group) and the makespan equals to the bound of the perfect matching.

There are some algorithms for finding the minimum-weight perfect matching on a weighted graph [19, 27]. However, they cannot apply to our problem directly because their objective functions are typically the sum of edge weights, rather than the maximum of edge weights in our problem.

We develop an algorithm to determine a minimum-bound perfect matching as shown in Figure 2.9. We first construct a sorted list containing all the edges of a co-run makespan graph in an ascending order of their weights; the edge with the smallest weight resides on the top of the list. We then use a binary search to determine the smallest top portion of the sorted edge list that contains a perfect matching (regardless of weights) covering all vertices. The binary search starts with the top half of the edge list and checks whether a perfect matching can be found in those edges. A negative answer would suggest that more edges are needed, so the algorithm would try the top three quarters of the edge list. A positive answer would suggest that a smaller portion of the list may be enough to contain a perfect matching, so the algorithm would try the top quarter of the edge list. This binary search continues until it finds the smallest top portion of the edge list that contains a perfect matching. The perfect matching found by this algorithm indicates the best schedule of the jobs.

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

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

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

The time complexity of the perfect matching detection subroutine, *findPerfMatch(G)*, is $O(\sqrt{n} \cdot m)$ [27], where n and m are the numbers of vertices and edges in the graph. In the algorithm, the binary search process contains $O(\log n)$ invocations of perfect matching detection. The value of m can be no greater than n^2. The time complexity of the algorithm is $O(n^{2.5} \cdot \log n)$.

```
/* V : vertex set; E : edge set */
/* S : generated perfect matching */
L ← sortEdges(E);
lbound ← 1; ubound ← |L|;
G.vertices ← V; S ← ∅;
while (1) {
  curPos ← ⌊ (ubound+lbound)/2 ⌋;
  if (curPos == ubound) return S;
  G.edges ← L[1:curPos];
  S ← findPerfMatch(G);
  if (S≠ NULL)
   ubound ← curPos;
  else
   lbound ← curPos;}
```

Figure 2.9: Algorithm for minimum-bound perfect matching.

2.7 Evaluation

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

2.7.1 Methodology

The machines we use include both dual core and u-core ($u > 2$) systems For dual-core cases, we use a quad-core Dell PowerEdge 1850 server, which although named quad-core, includes two Intel Xeon 5150 2 66 GHz dual-core processors, each having a 4MB shared L2 cache Every core has a 32KB dedicated L1 data cache For the cases of $u \geq 3$, we use machines each equipped with two quad-core AMD Opteron processors running at 1 9 GHz Each core has 512KB dedicated L2 cache and shares a 2MB L3 cache with the other three cores

Table 2 1 lists the 16 programs used in the experiments, along with the ranges of their performance degradations when they co-run on the AMD machine The programs are chosen to cover both integer and floating-point benchmarks and span a wide range of the application areas Their executions exhibit various patterns in memory and cache accesses—from having few data reuses (e g , gzip) to having many (e g , swim) All programs come from SPEC CPU2000 except *stream* coming from a streaming benchmark [46][6] Most of them have no degradation in their best co-runs, whereas in the worst co-runs, all the programs show more than 50% slowdown The large degradation ranges suggest the potential for co-scheduling In addition, we employ some synthetic problems for large coverage and the test of extreme scenarios In those problems, the job lengths and co-run degradation rates are some random values

In the collection of co-run degradations, we follow Tuck and Tullsen's practice [71], wrapping each program to make it run 10 times consecutively and only collecting the behavior of co-runs, which are the runs overlapping with other programs The hierarchical perfect matching algorithm requires the co-run performance on smaller virtual chips In this experiment, we collect such information by running 2 instances of 2 programs (totally 4 jobs) on a quad-core processor The degradation is used as the estimation of that on a virtual dual-core chip for some algorithms applied to quad-core machines

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

Table 2.1: Performance degradation ranges on AMD Opteron without job migrations

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

2.7.2 Basic Optimal Co-Scheduling

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

2.7.2.1 Optimal Co-Scheduling by Perfect Matching

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

Figure 2.10 shows the comparison among 3 different scheduling results. We use *optimal* to represent the schedule found by the minimum-weight perfect matching algorithm. The *random* bars show the average scheduling results produced by 1000 random schedules, cor-

responding to most current CMP scheduling systems, which are oblivious to shared cache. The *worst* bars are the results from the worst among all schedules, demonstrating the possible consequence of careless scheduling. The co-run groups in the optimal co-schedule are {*ammp+parser, art+crafty, bzip+gap, equake+mesa, gzip+mcf, twolf+vpr*}.

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

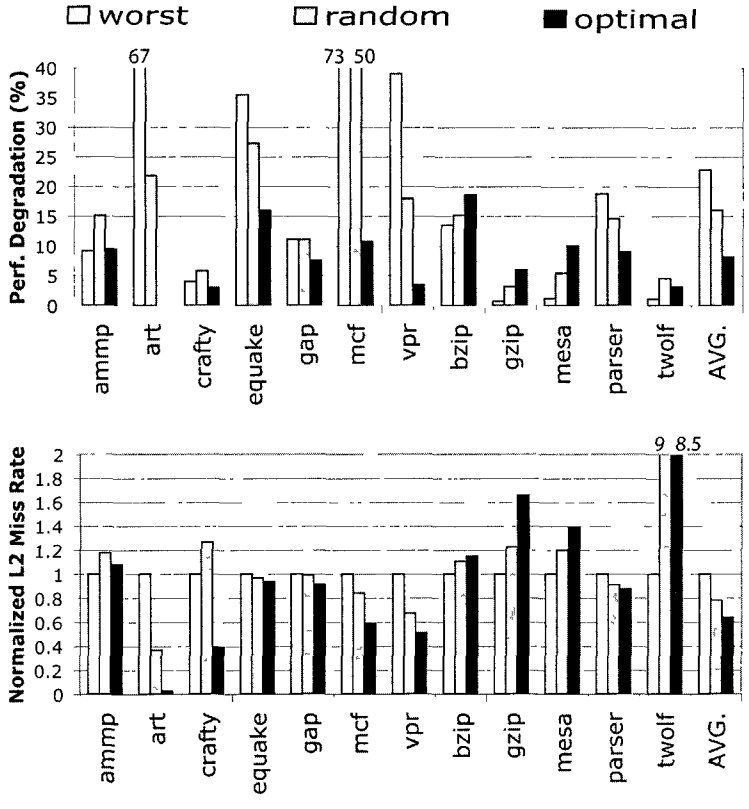


Figure 2.10: Performance degradations (top graph) and L2 cache miss rates (bottom graph) in different co-schedules in Intel Xeon 5150 (no migrations).

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

2.7.2.2 Lower Bounds by Linear Programming

Table 2.2: Co-Run degradations and scheduling times on synthetic problems, with three instances for each problem size (no migrations).

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

This section reports the results for validating the optimality of the solution produced by the IP model, and assessing the lower bounds by the LP relaxation. We use a sequence of synthetic problems (u is 4) to cover various cases. Table 2.2 reports the degradations of the resulting co-schedules, along with the time the scheduling algorithms take. The co-schedules produced by the IP algorithm always have the same degradations as the co-schedules found by the brute-force search. The IP algorithm takes much less times than the brute-force search does. The LP algorithm exhibits even better appeal: The degradations from it show minor difference (less than 10%) from the optimal, but can be obtained in less than 1% of the IP time for large problems.

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

2.7.2.3 Estimation by Heuristics-Based Algorithms

Using the collected degradations, we measure the effectiveness of the scheduling algorithms by a comparison of four types of schedules: the optimal, the random, the hierarchical perfect matching, and the greedy schedules, along with the enhanced version of the latter two when local optimization is applied. The metric we use is the average performance degradation of all programs.

We obtain the optimal schedule by solving the corresponding IP model; the result matches with the exhaustive search result. To schedule 16 jobs on four quad-core chips, the total number of possible schedules is $2,627,625$. The search time increases exponentially as the numbers of jobs and cores increase. We obtain the random scheduling result by applying 1000 random schedules to the jobs and getting the average performance. The random scheduling result corresponds to the performance of current CMP schedulers, which are oblivious to cache contention.

Table 2.3: Schedule results from different algorithms.

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

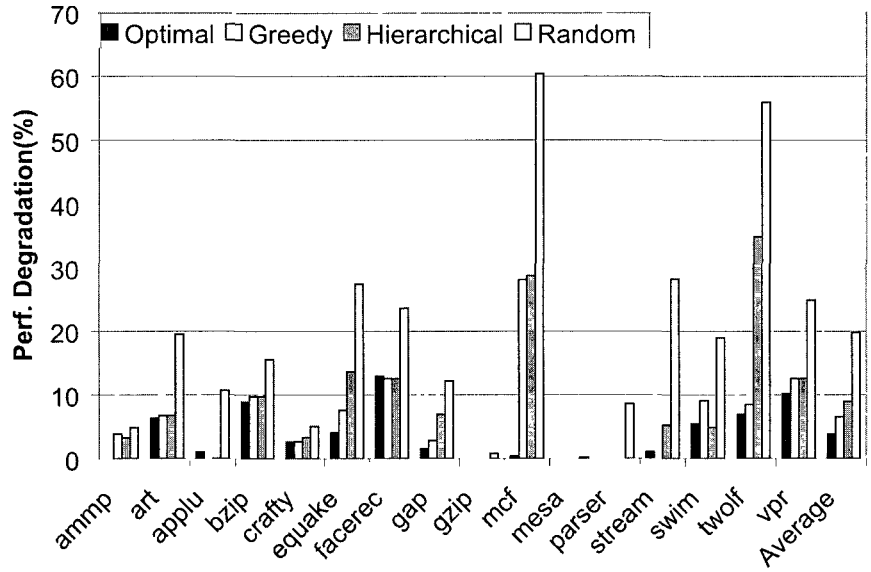


Figure 2.11: Performance degradation under different schedules.

To concentrate on the effectiveness of the two approximation algorithms, this section reports their results when local optimization is not applied. Table 2.3 presents an optimal schedule and the schedules generated by two approximation algorithms. (Random schedules are not listed since we used 1000 of them.) The 4 programs in each table cell compose a co-run group. Figure 2.11 shows the co-run degradation of each program in different schedules (some bars have 0 height and are thus invisible). The random schedules degrade the overall average performance by 19.81%. The hierarchical perfect matching algorithm reduces the degradation to 8.91%, whereas the greedy algorithm reduces it to 6.52%. The schedules produced by the two approximation algorithms have 5.08% and 2.40% more degradation than the optimal schedule.

The two approximation algorithms have similar effects on 5 programs, *art, bzip, facerec, parser*, and *vpr*. The greedy algorithm outperforms the hierarchical perfect matching algorithm on all the other programs except *ammp* and *swim*. On the program *stream*, the greedy algorithm outperforms the optimal schedule, which is not abnormal because

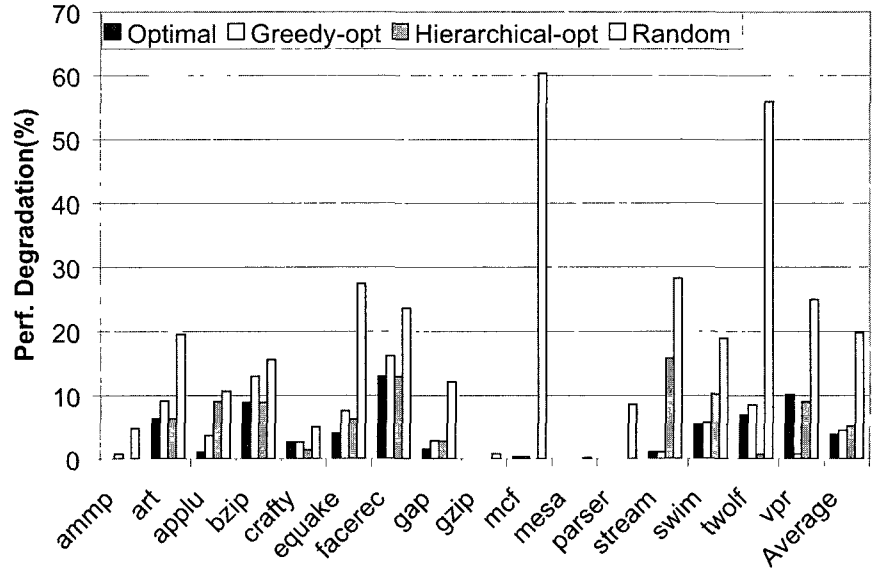


Figure 2.12: Performance degradation under different schedules.

our objective function is to minimize the overall performance. The better schedules assign jobs more balanced (as shown in Figure 2.13), which is the key to achieving better overall performance.

Although the two approximation algorithms cut performance degradation of the random schedules by 55.0% and 67.1% respectively, they still have considerable distances from the optimal schedule. The local optimization brings them closer to the optimal.

Figure 2.12 presents the performance of the schedules generated by the two approximation algorithms with local optimization. Local optimization boosts the performance in both schedules. For the hierarchical perfect matching algorithm, the average degradation is reduced by 41.2%, from 8.92% to 5.21%; for the greedy algorithm, the reduction is 30.7%, from 6.52% to 4.51%. Their average performance degradations become only 1.4% and 0.7% away from the optimal, respectively.

A detailed analysis shows that the local optimization improves the performance of 7 programs, including the drastic improvement on *equake, gap*, and *mcf*. For example, the

degradation of *mcf* is reduced from 29% to less than 0.38% when the local optimization is applied to the two approximation algorithms. Meanwhile, the local optimization slightly worsens the performance of *art, applu, bzip, facerec,* and *swim*, but the negative effects are remarkably smaller than the enhancements. This result again shows the importance of balance in co-scheduling.

Overall, the greedy algorithm slightly outperforms the hierarchical perfect matching algorithm in terms of the reduction of the average performance degradation. But with local optimization, both approximation algorithms produce close-to-optimal results, reducing average co-run degradation by over 74%.

Given that the local optimization enhances the approximation algorithms so much, we start to wonder whether local optimization alone is good enough for co-scheduling. To get the answer, we apply local optimization to 1000 random schedules. The results show that although sometimes the schedules are close to the optimal schedule, at many times, the produced schedules are much more inferior than the optimal schedules. The worst schedule result has up to 9.77% degradation. The average performance degradation is 6.27%, considerably larger than what we get from the greedy and hierarchical algorithms with local optimization.

We use a set of synthetic problems to evaluate the quality of the heuristics-based algorithms more comprehensively. Given that the greedy algorithm shows the better performance than other approximation algorithms, we concentrate on this algorithm for further evaluation. We use the LP model to compute the lower bounds. Table 2.4 shows the evaluation results. The co-schedules produced by the greedy algorithm exhibit less than 11% distance from the lower bound, indicating the high quality of the co-scheduler.

Co-Scheduling Fairnes

Fairness is another important factor in measuring the quality of scheduling. Following the previous work [74], we measure the fairness of a schedule by **unfairness factor**, defined as the coefficient of variation (standard deviation divided by the mean) of the normalized performance (IPC_{co}/IPC_{si}) of all jobs. A smaller unfairness factor means that the pro-

Table 2.4: Assessment of the greedy algorithm by comparing with the random scheduling results and the lower bound from the LP algorithm (no migrations).

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

grams are subject to more similar influence from cache sharing; thus, the system is more fair.

Figure 2.13 shows that the optimal schedule has the best fairness, the random schedule has the worst, and the local optimization improves fairness by about 30%. The consistency between unfairness factor and overall performance degradation confirms the intuition that in order to reduce the overall performance degradation, we need to balance the degradation among different programs.

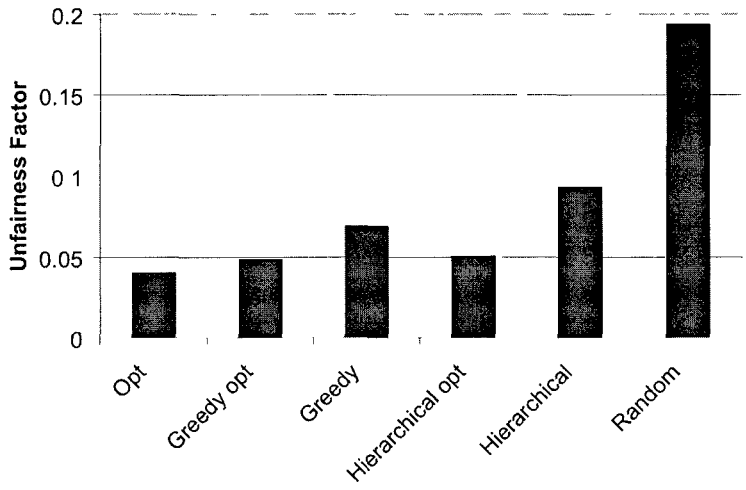


Figure 2.13: Unfairness factors of different schedules.

Co-Scheduling Scalability

As mentioned in previous sections, the time complexities of the heuristics-based algorithms are as follows: $O(nx\binom{n}{u}) + (\frac{n}{u})^2\binom{2u}{u})$ for the greedy algorithm, and $O(n^4) + (\frac{n}{u})^2\binom{2u}{u})$

for the hierarchical perfect matching (the $(\frac{n}{u})^2\binom{2u}{u}$) part is for the local optimization step), where, n for job numbers, u for the number of cores per chip, and $\frac{n}{u}$ for the number of chips.

The greedy algorithm has the same complexity as the hierarchical perfect matching algorithm when u is 4. However, as u increases, the overhead of the greedy algorithm increases much faster than the hierarchical method, which shows that the hierarchical method is more scalable. Given that n is typically much larger than u, the overhead of local optimization is often a small portion of the total time.

We use synthetic problems including 16 to 144 jobs to measure the running times of the two approximation algorithms with and without local optimization. Figure 2.14 depicts the running times of the four algorithms when u is 4. The greedy algorithms consume more time than hierarchical methods do. The result is consistent with the time complexity analysis presented earlier in this section.

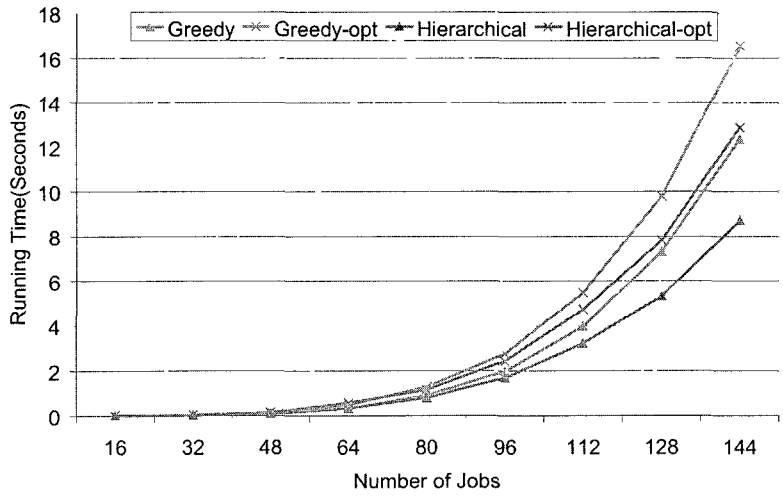


Figure 2.14: Scalability of different scheduling algorithms (no migrations).

2.7.3 Optimal Co-Scheduling with Migrations

This section evaluates the use of A*-search and the heuristics-based algorithms for co-scheduling jobs when job migrations are allowed. 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 [62] 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 2.5 lists the programs with their co-run degradation ranges on the Intel Xeon 5150 processors. The big ranges of degradations suggest the potential for co-scheduling.

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.

2.7.3.1 Optimal Co-Scheduling by A*-Search

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 2.5 to compare the performance of 6

Table 2.5: Benchmarks

Benchmark	single-run time (s)	co-run 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.

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 [31], 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.

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 2.15 shows the co-run 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

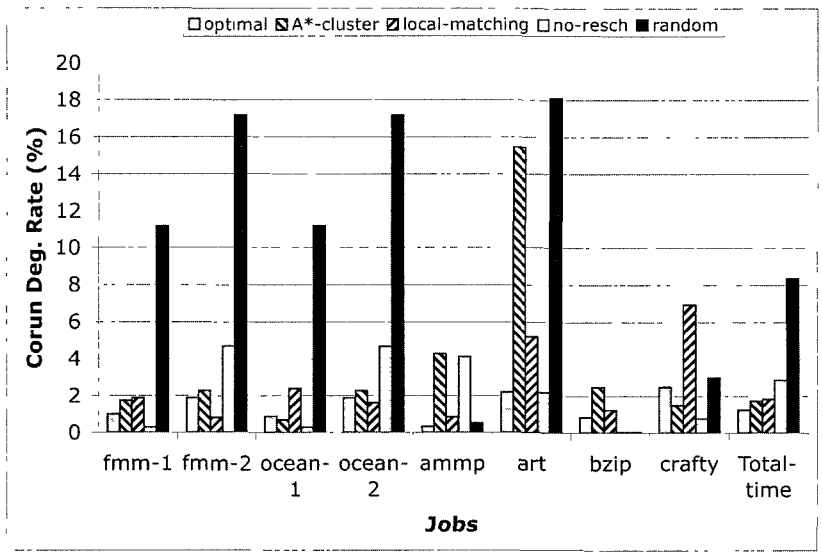


Figure 2.15: Performance degradation rates of 8 jobs co-running on quad-core Intel Xeon 5150 processors.

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 2.15. 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 achieves the overall optimum.

Table 2.6 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 co-run 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.

Table 2.6 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

2.7.3.2 Estimation by Heuristics-Based Algorithms

To get he optimal solution for 16 jobs, the brute-force algorithm would take years. Our implementation of the A*-search algorithm (in Java) is subject to memory shortage when scheduling more than 12 jobs. (A memory-bounded version [51] may help.) In this section, we concentrate on the evaluation of the two heuristics-based approximation algorithms on scheduling 16 jobs.

Co-Scheduling Performance on CMP

Figure 2.16 depicts the performance degradation rates on quad-core Intel Xeon 5150 processors, Table 2.7 reports the corresponding summary data. The random schedules cause 9.9% (up to 19.2%) degradation to the total running time. The non reschedule 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%, respectively. 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.

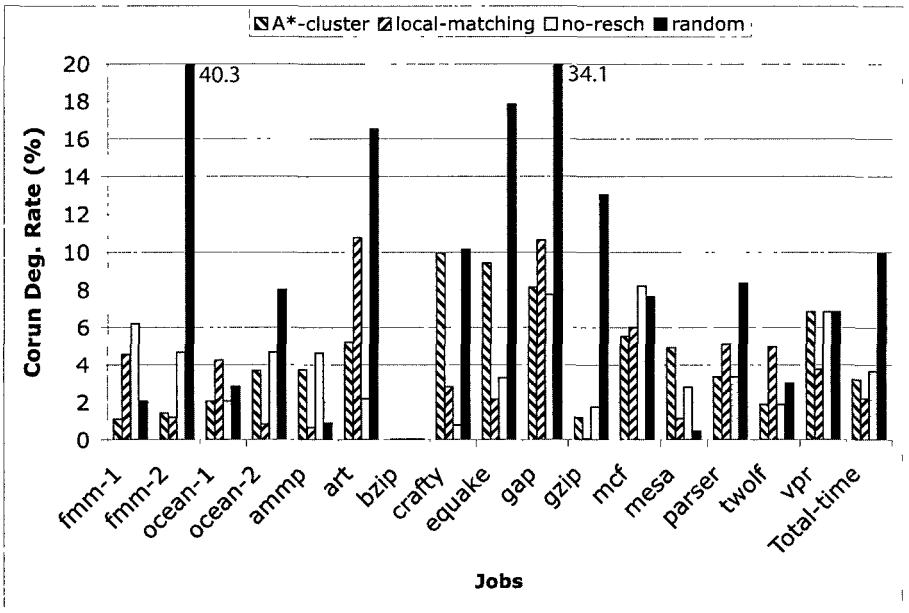


Figure 2.16: Performance degradation rates of 16 jobs co-running on quad-core Intel Xeon 5150 processors.

Table 2.7: 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

Co-Scheduling Performance on Hyper-Threads

Figure 2.17 and Table 2.8 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 2.7, 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

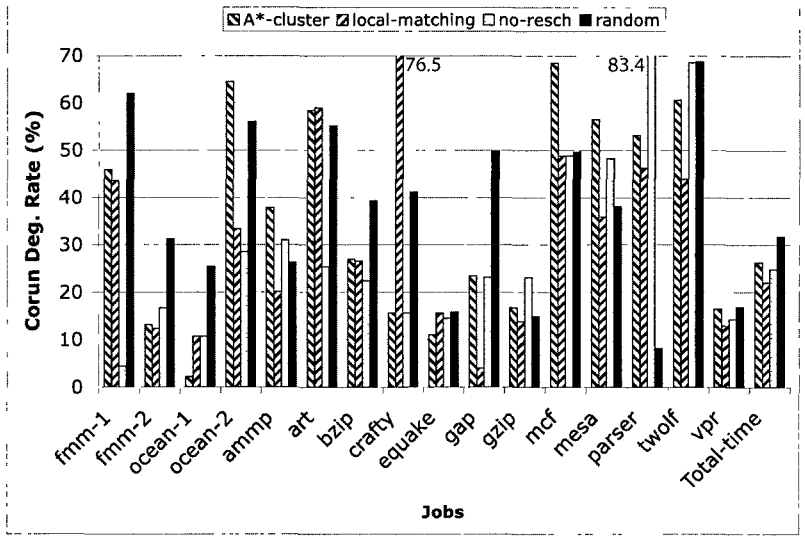


Figure 2.17: Performance degradation rates of 16 jobs co-running on the hyperthreads of Intel Xeon 5080 processors.

the multicore experiments, even though it visits fewer nodes; this is because of the difference in cluster sizes.

Table 2.8: 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

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 co-run degradations. Figure 2.18 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 sub-schedule at each scheduling stage.

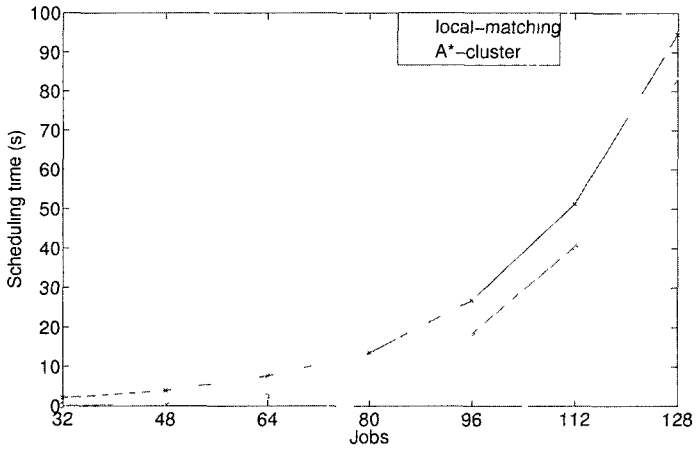


Figure 2.18: Scalability of the approximation algorithms

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

2.7.4 Makespan Results

This section presents the experimental results of our algorithms on job co-scheduling for minimizing the makespan.

We use two kinds of architecture for evaluating the co-scheduling algorithms. The CMP co-scheduling experiments are on machines equipped with quad-core Intel Xeon 5150 processors clocked at 2.66 GHz. Every chip has two 4MB L2 caches, each shared by two cores. Every core has a 32KB dedicated L1 data cache. For co-scheduling on Simultaneous Multithreading (SMT) machines, we use a system with Intel Xeon 5080 processors (two 2MB L2 cache per chip) running at 3.73 GHz. There are two hyper-threads on each core.

We use the same job suite listed in Table 2.5. In addition, we generate some sets of jobs whose single-run time and co-run degradations are set randomly. The use of these synthetic problems helps overcome the limitations imposed by the particular benchmark set.

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

Table 2.9: Co-schedule makespan on eight jobs without job migration. The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation

jobs	real		synthetic					
arch.	2-cmp	2-smt	2-core			4-core		
trial			1	2	3	1	2	3
brute-force	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65
A*	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65
matching	1.005	1.023	1.49	1.49	1.58	-	-	-
A*-cluster	1.005	1.167	1.55	1.75	1.58	2.38	2.3	1.65
greedy	1.005	1.17	1.49	1.9	1.8	2.77	2.34	1.85
rand-min	1.005	1.023	1.55	1.49	1.69	2.24	2.16	1.65
rand-med	1.016	1.255	1.81	2.7	2.22	2.55	2.34	1.88
rand-max	1.161	1.329	2.72	3.3	2.66	3.13	2.91	2.68

The data in Table 2.9 and Table 2.10 shows the schedule result for eight jobs on different architectures without and with job migration respectively. For the 8 real jobs on Xeon 5150 (2-cmp), for instance, the optimal schedule found by the algorithm A* and matching are

Table 2.10: Co-schedule makespan on eight jobs with job migration. The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation

jobs	real		synthetic					
arch.	2-cmp	2-smt	2-core			4-core		
trial			1	2	3	1	2	3
A*	1.002	1.013	1.33	1.21	1.19	1.99	1.93	1.56
matching	1.002	1.023	1.37	1.43	1.52	-	-	-
A*-cluster	1.012	1.023	1.55	1.48	1.29	2.19	2.12	1.63
greedy	1.005	1.17	1.43	1.9	1.8	2.32	2.08	1.87
rand-min	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65
rand-med	1.016	1.196	1.81	2.7	1.92	2.54	2.33	1.87
rand-max	1.161	1.329	2.72	3.3	2.66	3.13	2.91	2.68

all as follows: (fmm-1,ocean-1), (ammp,cafty), (art,bzip), (fmm-2,ocean-2), where fmm-n and ocean-n are their nth threads, and each pair of parentheses include a co-running group. The makespan is 0.5% larger than the makespan when the programs run in isolation.

The bottom 3 rows in the two tables reveal the minimum, median, and maximum of the makespans of 100 randomly generated schedules, corresponding to the scheduling in many existing systems, which work in a cache-sharing-oblivious manner. The minimum makespans are close to the optimal in the "no rescheduling" cases, but are mostly over 10% larger than the optimal in the "rescheduling" cases. The median and maximum are significantly larger than the optimal. For the eight real jobs, although random scheduling is likely to produce near optimal makespan in the Xeon 5150 system, it causes over 20% makespan increase on the SMT systems. These results indicate the risks of neglecting cache sharing in job scheduling.

Besides the optimal co-scheduling results, Table 2.9 and Table 2.10 also list the performance of the approximated schedules. On real jobs, the matching-based approximation produces near optimal results, the A*-cluster algorithm works similarly well except in the case of "no rescheduling" on "2-smt" architecture where the makespan is about 14% larger than the minimum. Because of the imprecision caused by clustering, both heuristic algorithms significantly outperform the greedy and random scheduling in most real and synthetic cases. On the other hand, their distances from the optimal reflect the room for improvement.

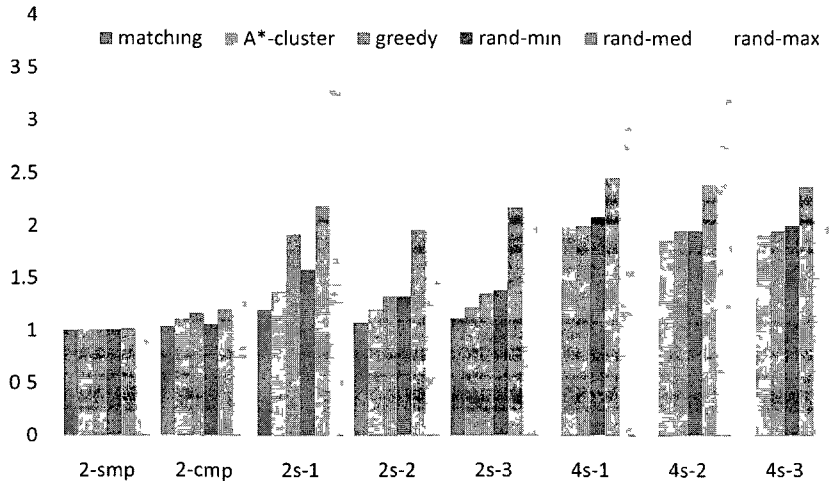


Figure 2.19· Co-schedule makespan on 16 jobs with job migration. The bars in the graph are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation. The first two groups are the results on real jos. The rest groups are the schedule results of synthetic jobs.

Figure 2.19 presents the results on 16 jobs when migration is allowed. It does not include the brute-force and A* results because the former takes too much time (up to years with job migrations) to finish and the latter requires too much memory to run. The results of the heuristic algorithms are consistent with the 8-job results. Although the minimum makespans from the random schedules occasionally get close to the results of the heuristic algorithms, most random scheduling results are significantly worse than the matching-based and A*-cluster-based approximations. The greedy algorithm, although performing not as well as the other two heuristic algorithms, outperforms the median results from random scheduling considerably.

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

Comparison with Cost Minimization

As mentioned earlier, the two scheduling criteria, makespan and total cost, typically lead to different results. It is confirmed by the experimental results. For example, Figure 2.20

shows the optimal schedules (without rescheduling) for both criteria on the Xeon 5080 (2-smt) machine The schedule with minimum total cost turns out to have 33% larger makespan than the schedule from the makespan minimization algorithms On the other hand, the schedule with minimum makespan causes extra cost as well This difference confirms the need for studies on each of the criteria and the application of the corresponding algorithms in different scenarios

cost minimization
schedule (fmm-1, crafty), (fmm-2, ocean-1), (occean-2, art), (ammp, bzip)
cost (ie , total degradation) 12 13
makespan 58 02 sec

makespan minimization
schedule (fmm-1, bzip), (fmm-2, art), (ocean-1, ammp), (ocean-2, crafty)
cost (ie , total degradation) 12 88
makespan 43 56 sec

Figure 2.20 Optimal schedules for cost minimization and makespan minimization on Xeon 5080 (2-smt) with no rescheduling

2.8 Insights for the Development of Practical Co-Scheduling Systems

The algorithms proposed in this work have two main uses The first is to help determine the potential for co-scheduling a set of jobs and to facilitate the assessment of practical co-scheduling systems, as exemplified by some recent work [78] The second is to inspire the development of co-scheduling mechanisms that are ready to be deployed in realistic settings This section presents some lessons and insights for the second use

Our first observation is that simple algorithms are capable of producing close-to-optimal results, as shown by the comparison between the simple greedy algorithm and the sophisticated hierarchical perfect matching algorithm (Section 2 7 2 3), and the comparison between the simple local matching algorithm and the A* algorithms (Section 2 7 3 2)

Second, in the design of greedy algorithms, it is important to distinguish "friendly" jobs from "unfriendly" ones, and couple them together (Section 2.4.3.2) in the produced schedule.

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

Finally, the local optimization is a cheap but effective way to refine co-scheduling results. The results in Section 2.7.2.3 are obtained after local optimizations cut degradations by 41.2% and 30.7% for the hierarchical perfect matching algorithm and the greedy algorithm respectively. Local optimizations may serve as a post-processing step for various co-scheduling algorithms.

2.9 Related Work

At the beginning of this project, we conduct an extensive survey, trying to find some existing explorations on similar problems in the large body of scheduling research. However, surprisingly, no previous work in traditional scheduling has been found tackling an optimal co-scheduling problem that contains performance interplay among jobs as what the current co-scheduling problem involves. As Leung summarizes in the *Handbook of Scheduling* [38], previous studies on optimal job scheduling have covered 4 types of machine environments: *dedicated*, *identical parallel*, *uniform parallel*, and *unrelated parallel* machines. On all of them, the running time of a job is fixed on a machine, independent of how other jobs are assigned, a clear contrast to the performance interplay in the co-scheduling problem tackled in this current work. Even though traditional Symmetric Multiprocessing (SMP) systems or NUMA platforms have certain off-chip resource sharing (e.g., on the main memory), the influence of the sharing on program performance has been inconsiderable for scheduling

and has not been the primary concern in previous scheduling studies. Some scheduling work [38] does have considered dependencies among jobs. But the dependencies differ from the performance interplay in co-scheduling in that the dependencies affect the order rather than performance of the execution of the jobs.

Recent studies on multi-core job co-scheduling fall into two categories. The first class of research aims at constructing practical on-line job scheduling systems. As the main effect of cache sharing is the contention among co-running jobs, many studies try to schedule jobs in a balanced way. They employ different program features, including estimated cache miss ratios, hardware performance counters, and so on [23, 59, 32]. All these studies aim at directly improving current runtime schedulers, rather than uncovering the complexity and solutions of optimal co-scheduling.

The second class of research is more relevant to optimal co-scheduling. A number of studies [10, 3] have proposed statistical models for the prediction of co-run performance. The models may ease the process for getting the data needed for optimal scheduling.

Beside co-scheduling, researchers have explored some other approaches to exploiting shared resource in multi-core architectures. In a recent study, Zhang and others [73] have found that the effects of thread co-scheduling become prominent for many multithreading applications only after some cache-sharing-aware transformations are applied. Several other studies [33, 52] have explored the effects of program-level transformations for enhancing the usage of shared cache. In addition, some other studies have tried to alleviate cache contention through cache partitioning [49, 17], cache quota management [50], and so forth.

2.10 Summary

This chapter describes a study on the analysis of the complexity and the design of efficient algorithms for determining the optimal co-schedules for jobs running on CMP. It presents a set of discoveries, including the polynomial-time optimal co-scheduling algorithm for dual-core systems, the proof of the NP-completeness of the co-scheduling problem for systems

with more than two cores per chip, the IP/LP formulation of the optimal co-scheduling problem, and a spectrum of heuristics-based algorithms for complex problems. Experiments on both real and synthetic problems validate the optimum of the results by the optimal co-scheduling algorithms, and demonstrate the effectiveness of the heuristics-based algorithms in producing near-optimal schedules with good efficiency and scalability.

Chapter 3

Co-Run Performance Prediction

3.1 Introduction

Starting with the adoption of Simultaneous Multithreading (SMT), cache sharing among computing units has become increasingly common, especially as processor designs enter the era of Chip Multiprocessors (CMP). The sharing is important for reducing inter-thread latency, but also brings cache contention between co-running processes. Many studies have shown considerable and sometimes significant effects of the contention on program performance and system fairness [23, 22, 50, 66, 20, 10, 28]. The urgency for alleviating the contention keeps growing as the processor-level parallelism rapidly increases.

Data locality (or data reuse) is a critical factor in both language design and implementation. Since 1960s, locality modeling—that is, analyses of data reuses patterns and the influence on cache or memory performance – has drawn decades of research interests, especially on the management of virtual memory and cache [13, 45]. The explorations have produced fundamental understanding to program locality and the behavior of dedicated cache. However, for shared-cache behavior, the current understanding remains preliminary.

The major change caused by cache sharing for locality analysis is on cache-level interactions among computing units. In dedicated cache systems, the interactions mainly occur at context switch time; while with shared cache, the interactions happen at almost every cache access. The significantly complicated interactions pose many new challenges to the locality

models that have been developed before—new explorations are necessary for an enhanced understanding of the implications of shared cache to program performance.

Even though some studies have tried to characterize the influence of cache sharing on program performance, most of them are either based on certain heuristics (e.g., average access frequencies of cache sharers[1] [10]) or some hardware extensions (e.g., [49, 53].) What is missing is a rigorous formulation of the interactions on shared cache and an in-depth understanding on how cache sharing influences program cache performance. As a result, current treatments to cache sharing are primarily through runtime monitoring of low-level program behaviors (e.g., instructions per cycle (IPC), cache miss rates), no matter the goal is a better cache partition [63, 29, 28, 50] or an enhanced job scheduler [59, 60, 53, 23, 74, 20].

In this chapter, we present some techniques to reveal the influence of cache sharing on program cache performance by uncovering the inherent connections between the locality of program single-runs (i.e. runs with no cache sharers) and that of their co-runs.

This work includes three components. First, we formulate the problem of predicting co-run cache contention as a problem of the prediction of program *inclusive reuse signatures*—which is a summary of LRU stack distances [45] on a shared cache with all cache sharers' data references considered—and conduct a theoretical analysis to expose the inherent statistical connections between single-run memory behavior and co-run inclusive reuse signatures. The theoretical analysis sheds insights on the prediction of co-run performance from single-run data locality. In light of that, we develop a lightweight model for efficiently predicting co-run data locality (or cache usage) from the memory reference patterns of the programs' single-runs. The high efficiency of the model is the key to its uses in shared-cache management. It achieves the efficiency mainly by capitalizing on the connection between time and locality. Finally, We analyze the influence of program inputs on the predictive models. Based on the analysis, we conduct an exploration in addressing the influence by constructing cross-input predictive models for some memory behaviors that are critical for the co-run performance prediction.

[1] Cache sharers refer to the processes that run concurrently on a shared cache.

On AMD Opteron quad-core machines, the scheduling derived from out model achieves close-to-optimal results, cutting cache-contention caused performance degradation by as much as 63% on average, improving program performance by 9% on average (up to 50% for individual applications.)

This work builds on decades of research in locality modeling. Enlightened by many seminal cache studies [13, 45, 58], it takes a statistical view at the relation between data references and cache behavior, and uncovers some inherent properties of co-run locality on shared cache systems. It resides in the area of program locality analysis but opens opportunities for proactive cache management on various levels of computing (e.g., scheduling on operating systems, cache partition in architecture design.)

In the rest of the chapter, Section 3.2 introduces the concept of co-run inclusive reuse signature and reveals its inherent connections with single-run locality through a statistical model. Section 3.3 describes a lightweight approach for co-run performance prediction. Section 3.4.1 presents our exploration on the impact from program inputs. Section 3.5 reports experimental results. Section 3.6 reviews the related work, followed by a short summary.

3.2 Inclusive Reuse Distance

This section first introduces the model of inclusive reuse signature and the relation with shared-cache performance. It then uncovers the statistical connection between inclusive reuse signatures and the single runs' memory behavior, laying the foundation for the lightweight inclusive locality model developed in the next section.

3.2.1 Inclusive Reuse Distance and Cache Sharing

On architectures without cache sharing, a widely used locality model is LRU stack distance, or *reuse distance*, which is defined as the number of distinct data elements accessed between the current and the previous reference to the same element [45]. Treating a cache block

as a data element leads to cache-block reuse distance. Researchers have used cache-block reuse distance histograms, also called *reuse signatures* [16], to predict the performance of a program when cache is not shared. Figure 3.1 illustrates the basic idea: Every memory reference to the right of the cache-size line is considered a cache miss because too many other data have been brought into cache since its previous reference. Although the prediction assumes fully-associative cache, experiments have shown high accuracy for set-associative cache as well [43, 76].

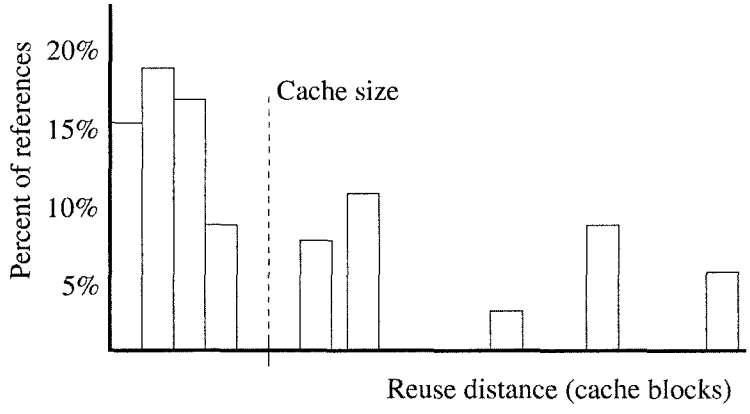


Figure 3.1: An example of cache-block reuse signature

Inclusive reuse distance is a straightforward extension of reuse distance for shared cache. It is defined as the number of distinct data elements of *all cache sharers* that are accessed between the current and the previous references to the same data element. Its histogram is called *inclusive reuse signature*. For clarity, we call traditional reuse distance *exclusive reuse distance*, and name the corresponding histogram as *exclusive reuse signature*.

Three features of inclusive reuse signature make it desirable for characterizing data locality on shared cache. First, it strongly correlates with cache performance. It can be used to predict shared-cache miss rates in the same way as illustrated in Figure 3.1. Second, it is independent to cache configurations; an inclusive reuse signature can be used to predict the miss rates of shared caches of different sizes. This feature is important for reconfigurable caches and cache partition. Finally, as we will show, inclusive reuse signatures can be derived

from single runs' memory behavior. This feature removes the need for direct collection of inclusive reuse signatures of (often a large number of) co-runs, a key to the lightweight model presented in the next section.

Like exclusive reuse signature, inclusive reuse signature cannot capture fine-grained cache conflicts. However, both the previous experiments on exclusive reuse signature [43, 21, 76], and the evaluation in Section 3.5 show that this limitation does not prevent effective uses of reuse signatures. Hardware extensions (e.g., [53]) allow the monitoring of fine-grained cache activities. Inclusive reuse signature on the other hand offers the overall cache requirement of a program without the need for hardware modifications. The two different techniques are complementary to each other.

3.2.2 Connections to Single Runs

This section presents the connection between single runs' memory behaviors and inclusive reuse signatures. This connection is critical for efficient attainment of inclusive reuse signatures. We capture the connection through a series of probability and mathematical inferences, expressed below. The intuition of the connection is that if we can compute the number of distinct data elements accessed by each cache sharer in an arbitrary time interval, we can easily derive the inclusive reuse signature. We prove that this number can be inferred from a special kind of reference histogram, namely *time distance histogram*, of the single-run of each process.

Time distance is defined as the number of memory references in a reuse interval[2]. In the reference sequence "a b b c a", the time distance of the final access is 4 (while the reuse distance is 2.) *Time distance histogram* is similar to reuse signature shown in Figure 3.1 except that the X-axis is replaced by time distance. Time distance histogram can be on different levels: An entire data trace may have one overall time distance histogram, while each data object in the trace may have its own time distance histogram with the time distances of only the references to that object contained.

[2]We use logical time—that is, the number of data references—for the length of an interval.

Let $M^{(j)}(\Delta)$ represent the statistical expectation of the number of distinct data accessed by process j in an arbitrary time interval that has length of Δ. There are three steps in computing $M^{(j)}(\Delta)$ from its time distance histogram. *Step 1:* From the time distance histogram of each data object, we calculate the probability for a data object, say (O_i), of process j to appear in the interval, denoted by $P_i(\Delta)$. *Step 2:* From $P_i(\Delta)$ $(i = 0, 1, \cdots, N$; N is the total number of distinct data objects accessed by process $j)$, we obtain the probability for that interval to contain k $(k = 0, 1, \cdots, N)$ distinct objects of process j, denoted by $P(k, \Delta)$. *Step 3:* From $P(k, \Delta)$, we compute the expected number of distinct objects that process j accesses in the interval, which is the value of $M^{(j)}(\Delta)$.

Compute $P_i(\Delta)$

For the object O_i to be accessed in a Δ-long interval, it can be either accessed in the first Δ-1 time points, or, not until the end of the interval. With $q_i(\Delta)$ representing the probability for the data to be not accessed until the end of the interval, $P_i(\Delta)$ can be expressed as

$$P_i(\Delta) = P_i(\Delta - 1) + q_i(\Delta).$$

Hence the following equations:

$$\begin{aligned} P_i(\Delta - 1) &= P_i(\Delta - 2) + q_i(\Delta - 1); \\ P_i(\Delta - 2) &= P_i(\Delta - 3) + q_i(\Delta - 2); \\ \cdots &\quad \cdots \\ P_i(1) &= P_i(0) + q_i(1). \end{aligned}$$

Apparently $P_i(0)$ is 0 (no objects can be accessed in a 0-long interval.) Deduction from these equations produces the following formula:

$$P_i(\Delta) = \sum_{\tau=1}^{\Delta} q_i(\tau). \tag{3.1}$$

Notice that $q_i(\tau)$ equals the probability for O_i to *1)* be the final data reference in an interval of length τ, and meanwhile, *2)* have a time distance larger than τ at that data reference (otherwise, it would be also accessed at other points in that interval.) With $p_i^{(1)}$ and $p_i^{(2)}$ respectively denoting the probabilities for the two conditions to hold, $q_i(\tau)$ can be computed as $q_i(\tau) = p_i^{(1)} p_i^{(2)}$.

The probability $p_i^{(2)}$ comes directly from the time distance histogram (denoted as H_i) of object O_i as $\sum_{\delta=\tau+1}^{T} H_i(\delta)$. With $p_i^{(1)} = n_i/T$ (n_i is the total references to O_i in all the T data references in the execution), $q_i(\tau)$ can be computed as

$$q_i(\tau) = \frac{n_i}{T} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{3.2}$$

Together, Equations 3.1 and 3.2 lead to the following computation of $P_i(\Delta)$ from the time distance histogram:

$$P_i(\Delta) = \frac{n_i}{T} \sum_{\tau=1}^{\Delta} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{3.3}$$

Compute $P(k, \Delta)$ and $M^{(J)}(\Delta)$

With $P_i(\Delta)$ ($i = 0, 1, \cdots, N$), we can compute the probability for an interval to contain k distinct data, denoted as $P(k, \Delta)$ as follows:

> $P(k, \Delta) = \sum_S$ (the probability for the interval to contain and only contain all the members of S).

where, S is a k-member subset of $A = \{O_1, O_2, \cdots, O_N\}$. Using $P_i(\Delta)$, $P(k, \Delta)$ can be computed as follows[3]:

[3]This computation, as most trace-based locality analyses (e g , [13, 58, 55]), assumes data distribute independently from one another Results of those previous studies have shown minor influence of the assumption on locality characterization when the program contains a large number of data

$$P(k,\Delta) = \sum_{S:|S|=k;S\subseteq A} \left(\left(\prod_{i\in S} P_i(\Delta)\right) \left(\prod_{j\in A-S} (1 - P_j(\Delta))\right)\right). \tag{3.4}$$

Recall that $M^{(j)}(\Delta)$ is the statistical expectation of the number of distinct data accessed by process j in an arbitrary time interval of length Δ. According to the definition of statistical expectation, we can compute $M^{(j)}(\Delta)$ from $P(k,\Delta)$ as follows:

$$M^{(j)}(\Delta) = \sum_{k=0}^{min(\Delta-1,N)} k \cdot P(k,\Delta) \tag{3.5}$$

Equations (3.3,3.4,3.5) together form a model for computing the co-run inclusive reuse signatures from the single-runs' time distance histograms and the numbers of data references to each data element.

This probabilistic model uncovers the connections between the locality of single-runs and co-runs. Although the high cost of the model prevents its direct uses (the time complexity is $O(N2^N)$), it lays the theoretical foundation for the prediction of co-run locality.

A Special Version on Cache-Line Level

Although the description of the model is on data object level, it applies to cache block level as well by regarding one cache block of data as a single object.

Moreover, under a common assumption on cache lines, the model can have a much simpler form. The assumption is that all cache lines are independent and identically distributed in the data reference trace. This is a typical assumption in previous cache behavior modeling, ranging from the early seminal work [13, 45, 58, 68] to recent explorations [10, 55].

Under this assumption, all data objects have the same $P(\Delta)$—that is, $P_i(\Delta) = P_j(\Delta)$ $(i, j = 1, 2, \cdots, N)$. Recall that $P(\Delta)$ is the probability for a given object to appear in an interval of length Δ. So, the assumption leads to that the probability for an interval to contain k distinct data—that is, $P(k,\Delta)$—obeys a binomial distribution. It's like having k heads in the toss of N coins, with $P(\Delta)$ probability of showing heads for a coin. According to binomial distribution, we have

$$P(k,\Delta) = \binom{N}{k} P(\Delta)^k (1 - P(\Delta))^{N-k}. \tag{3.6}$$

Furthermore, the assumption also significantly simplifies the computation of $P(\Delta)$. Consider Equation 3.3. Because of the assumption of the same distribution of all data objects, $n_i = T/N$, and the time distance histograms of all objects would be the same as the time distance histograms of the entire reference trace, denoted as $H_i(\delta) = H(\delta)$ $(i = 1, 2, \cdots, N.)$ Therefore, Equation 3.3 becomes

$$P(\Delta) = \frac{1}{N} \sum_{\tau=1}^{\Delta} \sum_{\delta=\tau+1}^{T} H(\delta). \tag{3.7}$$

Equations 3.7 and 3.6 are much simpler than their original version, Equations 3.3 and 3.4. Together with Equation 3.5, they compose a model for inclusive cache-line-level reuse signature prediction under the given assumption. (The results in Section 3.5.1 reflect the errors brought by the assumption.) The time complexity becomes $O(T^2 * S)$ (assuming $N < T$), where S is the number of sharers of a cache.

When there are no data sharing among cache sharers, a combination of their $M^{(j)}(\delta)$s $(j = 1, 2, \ldots, \#$ of sharers) is enough to approximate their concurrent reuse distance histograms. Let d be the time distance of a data reuse by process j. Suppose d_i is the number of memory references by one of its cache sharers, process i, during the same (physical) time period. The concurrent reuse distance of process j can be computed as $M^{(j)}(d) + \sum_{i \in j's\ co-runners} M^{(i)}(d_i)$. (Note, the values of d and d_is may be different, depending on the relative speeds of cache sharers.)

This combination, however, is not sufficient for co-running threads in multithreading applications because of the effects of inter-thread data sharing. Next, we will analyze the case when there is data sharing among jobs.

3.2.3 Data Sharing Case

In this section, we use the following example for explanation the cases where data sharing exists among jobs. There are two co-running threads T_1 and T_2. Suppose in a certain time period, the memory reference sequence is

a b X X b X c d X a

where, an X represents some reference conducted by T_2, and the other letters represent the references by T_1. Clearly, this time period corresponds to a reuse interval of reference to "a" in the standalone execution of T_1 with standalone reuse distance of 3 (for accesses to b, c, and d). We now examine its corresponding concurrent reuse distance for element "a" in three scenarios.

- Scenario 1: All Xs are something different from the data accessed by T_1. Let the four Xs be "p q p q". Apparently, the concurrent reuse distance of the reuse interval is just the sum of the numbers of distinct data in each of the two standalone reference sequences: $3 + 2 = 5$.

- Scenario 2: The four Xs are "p a p q". This scenario illustrates the first effect of data sharing. The reference to "a" breaks the reuse interval into two: "a b p a" and "a b p c d q a". The consequence is that the original reuse interval becomes meaningless. The approximation of the ultimate concurrent reuse distances of T_1 has to include a reuse distance of 2 (for "a b p a") and a reuse distance of 5 (for "a b p c d q a").

- Scenario 3: The four Xs are "p c p c". This scenario illustrates the second effect of data sharing. Because "c" is referenced by T_1 in that interval, the references to it by T_2 should not be counted in the concurrent reuse distance. So the resulting concurrent reuse distance is $3 + 1 = 4$ (rather than 5 as in Scenario 1).

The last two scenarios show the two effects of data sharing on concurrent reuse distance approximation.

To approximate the concurrent reuse distance of co-running threads, we first assume no data shared across the threads, and apply the model described in Part I to compute a concurrent reuse distance histogram, R' for each thread. We then revise R' by considering the two effects of data sharing. The revision tries to find the statistical expectation of the correct concurrent reuse distance for each reuse interval contained in R'.

To explain the revision step, we first introduce some notations. For simplicity, we assume there are only two co-running threads. Let N_1 and N_2 represent the total numbers of distinct data accessed by thread 1 and thread 2 (in their entire execution), S represent the set of data shared by the two threads. Suppose that there is a reuse interval V with ending elements as e accessed by thread 1 and its reuse distance in R' is d' (which needs to be revised in this revision process). Let n_1 and n_2 be the numbers of distinct data among the data accessed respectively by the two threads in V; both can be computed by Equation 3.5.

Treating the First Effect

The revision step first treats the interval-breaking effect that data sharing may impose to the concurrent reuse distance (the second effect is temporarily ignored). It computes the probability for the reuse interval V to be broken. That event happens only when the following two events both occur. The first is that e is a shared data element; clearly the probability is $|S|/N_1$. The second is that e ever appears in the references by thread 2 in the interval V; as any of the n_2 data elements could be e, the probability is n_2/N_2. So the probability for the reuse interval to be broken is $(|S|/N_1) * (n_2/N_2)$. Because e may appear anywhere in V, we assume the broken effect distributes to all sub-intervals of V uniformly. The probability for the resulting reuse intervals to have reuse distance of α ($\alpha = 0, 1, \ldots, d'$) is the same, which is $(|S|/N_1) * (n_2/N_2)/(d' + 1)$. Hence the number of reuse intervals of distance α in R' should increase by $(|S|/N_1) * (n_2/N_2)/(d' + 1)$. Meanwhile, because the original reuse interval is broken, the number of reuse intervals of distance d' in R' should decrease by $(|S|/N_1) * (n_2/N_2)$. We use R'' to denote the resulting histogram after this treatment.

Treating the Second Effect

In the treatment to the second effect of data sharing on concurrent reuse distance, each interval is not breakable as the interval-breaking effect has already been considered. For a reuse interval V in R'', let S_1 denote the set of distinct data among all references conducted by thread 1 in that interval, and S_2 for thread 2. In R'', the reuse distance of that interval would be $n_1 + n_2$. In this step, we want to correct this distance value by considering that there may be some overlap between S_1 and S_2. Let C represent the overlap set. Apparently, $C \subseteq S$. The probability for $|C| = c$ is

$$\frac{1}{\binom{N_1}{n_1} * \binom{N_2}{n_2}} \sum_{d=c}^{|S|} \binom{|S|}{d} \binom{N_1 - |S|}{n_1 - d} \binom{d}{c} \binom{N_2 - d}{n_2 - c},$$

where, $\binom{N_1}{n_1} * \binom{N_2}{n_2}$ is the possible ways to have a reuse interval like V, $\binom{|S|}{d}\binom{N_1-|S|}{n_1-d}$ is the number of ways for d shared data to appear in S_1, and $\binom{d}{c}\binom{N_2-d}{n_2-c}$ is the number of ways for thread 2 to access c data in the d shared data accessed by thread 1.

Those probabilities are enough to compute the statistical expectation of the concurrent reuse distance for every reuse distance in R'. Although our explanation uses two threads as the example, the model supports an arbitrary number of co-running threads.

Recall that the time complexity to predict the co-run reuse distance is $O(T^2 * S)$ (assuming $N < T$), where S is the number of sharers of a cache. As the number of data references (T) is usually large, even with this simplified version, we need a still more lightweight model for making co-run locality prediction feasible for real applications.

3.3 Lightweight Model for Locality Prediction

Based on the connection uncovered in the previous section, we propose a lightweight model to predict inclusive reuse signatures efficiently enough for the uses in cache management.

3.3.1 Lightweight Model

The lightweight model is based on *distinct blocks per cycle (DPC)*, a concept offering lightweight connection between data reuses and time. Roughly speaking, *DPC* is the average footprint in a cycle. Formally, it is defined as the average number of distinct memory blocks that are accessed in a CPU cycle. For cache analysis, it is natural to use the width of a cache line as the size of a memory block. In that case, DPC equals the average frequency of new cache lines being accessed. As an example, suppose a program accesses the following memory blocks in 100 cycles: b1 b1 b3 b5 b3 b1 b4 b2. The corresponding DPC is 5/100 = 0.05 (footprint is 5). For cache sharers, their DPCs reflect their aggressiveness in competing for cache resources. The following theorem more precisely characterizes the connection between DPC and cache contention.[4]

Theorem 1 *Suppose, with a set of processes P, process p shares a fully-associative cache of size L but shares no data. For an access by p whose exclusive reuse distance is d ($d < L$), let σ and σ' be the average DPC of p and P in the reuse interval. Then, if and only if $\frac{d}{L-d} < \frac{\sigma}{\sigma'}$, the access remains a cache hit.*

The proof of the theorem is straightforward. According to the definition of DPC, the total number of distinct data elements accessed by processes P in the reuse interval is $(\sigma' d/\sigma)$. Therefore, the inclusive reuse distance of the current access by process p equals $(d + \sigma' d/\sigma)$. It is clear that the necessary and sufficient condition for the access to be a hit in the fully-associative cache is that its inclusive reuse distance is less than the cache size, that is, $d + \sigma' d/\sigma < L$, which leads to the conclusion in the theorem.

The theorem suggests that along with exclusive reuse signatures, knowing the DPC of every reuse interval is enough for computing the miss rates on a fully-associative shared cache. Our experiments (reported in Section 3.5) demonstrate that the results from the theorem can serve for estimation of the miss rates on set-associative cache as well, an observation consistent with prior studies on exclusive reuse signatures [43, 76].

[4]By default, the discussion in this chapter excludes the memory accesses that are hits in dedicated caches as they do not reference the shared cache

For the uses in cache management, it is necessary to use DPC at a larger granularity to trade accuracy for efficiency. Experiments on the tradeoff lead us to the use of σ (the average DPC of p in Theorem 1) at the granularity of reuse distance bars. At that granularity, each bar, say bar b, in the exclusive reuse signature of p has an average σ_b, equal to $\sum_{a \in b} \sigma_a / r$, where, σ_a is the DPC of the reuse interval of memory reference a, and r is the number of memory references covered by bar b. For σ' (the average DPC of P in Theorem 1,) we use an even larger granularity. For a process in set P, denoted as p'_i, we take the average DPC of all bars in the exclusive reuse signature of p'_i, expressed as

$$\sigma'_i = \sum_{b=1}^{B'_i} r'_{i_b} \sigma'_{i_b}, \tag{3.8}$$

where, B'_i is the number of bars in the exclusive reuse signature of p'_i, r'_{i_b} is the number of memory references covered in bar b, and σ'_{i_b} is the average DPC of the bar. The σ' of P is $\sum_{p'_i \in P} \sigma'_i$.

At this large granularity, according to Theorem 1, the data references in bar i in the exclusive reuse signature of p are cache misses if $d_i \sigma' / ((L - d_i)\sigma_i) \geq 1$, where, d_i is the average exclusive reuse distance of bar i in the signature of p. Therefore, the increase of cache miss rate of p caused by co-running with P can be estimated as follows:

$$\delta = \sum_{i \in A} r_i U\left(\frac{d_i \sigma'}{(L - d_i)\sigma_i}\right) \quad U(x) = \begin{cases} 1 & x \geq 1 \\ 0 & x < 1 \end{cases} \tag{3.9}$$

where, A is the set of bars in the exclusive reuse signature of process p that contain distances smaller than cache size L.

3.3.2 Analysis

The lightweight model essentially effects the following mapping:

$$(E_1, \sigma_1, E_2, \sigma_2, \cdots, E_K, \sigma_K) \longmapsto (m_1, m_2, \cdots, m_K)$$

where E_i and σ_i are the exclusive reuse signature and DPCs of process i, and m_i is the cache miss rate of the process when it co-runs with the other $K-1$ processes.

In contrast to the $O(T^2 * S)$ complexity of the inclusive reuse signature model, this lightweight model reduces the complexity to $O(B * S)$, where B is the number of bars in the longest exclusive reuse signature and S is the number of cache sharers. The value of B is limited by the data size and is usually small: It is less than 256 in our experiment, where each bar in reuse signatures is 1K-wide and a cache line is 64B wide.

The use of DPC is essential to the efficiency of the lightweight model, but as a tradeoff, it causes approximation inaccuracy. Besides the errors due to the large granularity, a second source of inaccuracy is that the DPCs we use are measured (or predicted) from the single-runs rather than co-runs of the processes. But it is important to notice that many schemes of cache management make *qualitative* rather than quantitative decisions, such as decisions on whether two programs should be scheduled together. This property grants cache management some degrees of tolerance to the performance prediction errors. Section 3.5 shows that with all the inaccuracy, the lightweight model still effectively keeps the relative difference among processes' co-run performance degradations, and thus strikes a reasonable tradeoff between accuracy and efficiency.

Efficient attainment of single-run locality information is a topic that has been studied intensively in previous research. Previous approaches include static analysis [9], offline profiling [45, 55], cross-run prediction [16] and so forth. In our experiments, we employ the simplest approach, using binary instrumentation to measure the exclusive reuse distance and DPC[5] at every memory reference in a profiling run of each program. The studies on this topic are orthogonal to this work: No matter how single run behaviors are obtained, this work shows that they can lead to the estimation of co-run performance through the proposed co-run locality models.

[5]Instrumentation affects running time So, for DPC, we measure the number of distinct blocks per instruction instead, and then multiply that number by the average IPC of the non-instrumented run of the program.

3.4 Handling Program Inputs for Co-Scheduling

On a given CMP architecture, cache contention depends on two factors: the programs that run together, and their inputs. The first factor has been the main focus of previous studies. This section concentrates on the second factor. The goal is to uncover the effects of program inputs on CMP co-scheduling and to explore the solutions.

3.4.1 Influence of Program Inputs on Co-Run Performance

To explore the influence of program inputs on co-scheduling, we measure the co-runs of a dozen SPEC CPU2000 programs on their *test*, *train*, and *ref* inputs. The machine we use is a Dell PowerEdge 1850 server with two Intel Xeon 5150 2.66 GHz dual-core processors, each equipped with a 4MB shared L2 cache. We measure the performance degradation of a program when it co-runs with the other programs with the same types of input.

The boxplots in Figure 3.2 show the results. The differences among the boxplots inside a group reveal the strong influence of program inputs on co-run performance. Among the 12 benchmarks, *twolf* and *vpr* are the two that have the largest performance variation across inputs. The *test* runs of both of them have no performance degradation, no matter which program is their co-runner. Whereas, their *train* runs show up to 15% and 36% degradations, and their *ref* runs show up to 76% and 64% degradations. For the other programs, the *train* and *ref* runs are 15% to 564% worse than those of their respective *test* runs (in terms of median values). The results demonstrate that program inputs affect co-run performance significantly.

The results also show a second phenomenon. Although the working sets of the programs usually increase as input size increases, the co-run performance degradation doesn't necessarily increase. For instance, the *ref* runs of *equake*, *mcf*, and *parser* clearly have less degradation than their *train* runs. This phenomenon shows that co-run degradation does not necessarily increase when the single-run cache miss rate increases. An extreme case may convey the intuition behind: A program whose single run has no cache hits clearly

won't have any more cache misses when it co-runs with other programs; hence, its co-run performance degradation must be negligible. This observation suggests that in the design of co-scheduler, cache miss rate may not provide the sufficient information.

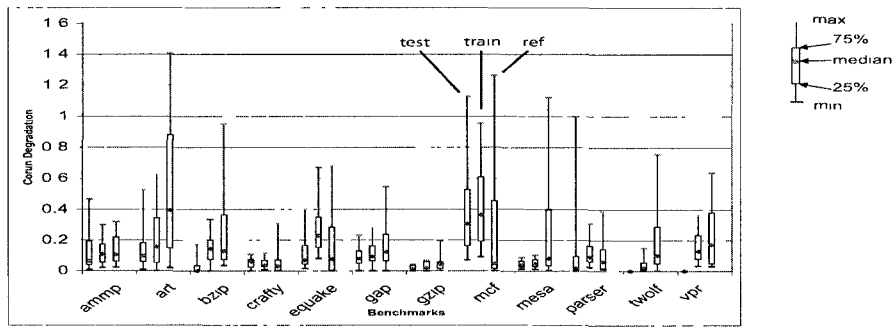


Figure 3.2: The boxplot showing the distribution of the performance degradation of each program when it co-runs with the other 11 programs. The three boxplots in a group respectively correspond to the executions on *test*, *train*, and *ref* inputs.

3.4.2 Predictive Input-Behavior Models

Our approach to addressing the influence of program inputs is to build predictive input-behavior models, which can predict program memory behavior from a given input. Because some co-schedulers can estimate co-run performance from single-run memory behavior and then derive the best schedules, we need only the mechanism to accurately predict the memory behavior of a program's single-runs (on arbitrary inputs). The performance predictive model is based on the following three memory behaviors: **Reuse Signature**, **Accesses per Instruction** and **Distinct Blocks per Cycle**. We focus on the construction of the predictive models for each of the three kinds of memory behaviors through statistical learning techniques.

The memory behavior of a single-run of a program, denoted by B, depends on the running environment E, the program code P, and the input I. In this work, E and P are given, and the goal is to find the function $f()$ mapping from I to B. With such a function, plugging any input into $f()$ will generate the predicted behavior of the program's corresponding single-run execution. We formalize the task as a statistical learning problem.

By feeding a program with different inputs, $I_1, I_2, \ldots, I_N$, we observe the corresponding behavior of the program's executions, represented by $B_1, B_2, \ldots, B_N$. The input-behavior pairs, $< I_i, B_i >$ $(i = 1, 2, \ldots, N)$, compose a training set, from which we use regression techniques to approximate function f.

Linear and Non-Linear Regression

Regression techniques are designed to discover the relation between a set of input attributes and a set of outputs. Linear regression assumes that the relation can be expressed by a linear function; non-linear regression permits more sophisticated functions.

Least Mean Squares (LMS) is a commonly used linear regression technique. Suppose f is a linear function mapping input $\overrightarrow{I}$ to a behavior B for a given program. Given training data set $< \overrightarrow{I_i}, B_i >$ (i=1,2,...,N), the goal of LMS is to find the approximation of function f, represented by $\hat{f}$, such that the mean error squares, $\frac{1}{N}\sum_{i=1}^{N}(B_i - \hat{f}(\overrightarrow{I_i}))^2$, is minimized.

LMS is simple and efficient, but applies to only linear functions. For non-linear regression, we choose the *k-Nearest-Neighbor* method. This method is an instance-based learning technique. For a new query instance, it retrieves a set of similar instances from memory and uses them to estimate the new output value. When $k = 1$, the method is named the Nearest-Neighbor method, or *NN* in short. The approximated function $\hat{f}()$ has an implicit and usually non-linear form [26]. The model building is simple, just recording the training instances into a data structure that can be efficiently searched. There are many other statistical learning techniques, such as Regression Trees and Support Vector Machines; they are more complex and costly. We restrain ourselves to a small number of training runs in order to limit the overhead of the offline profiling. Those more complex learning techniques often require a larger training data set.

Besides LMS and NN, we also use a hybrid method. For a given program, it chooses the better one between LMS and NN in terms of *training* errors. (The training error of a model is the prediction error of the model when being applied to the training data.) For each program showed in Figure 3.2, besides its *test*, *train*, and *ref* inputs included in the SPEC suite, we obtained another input from the collection of additional representative

inputs attained by Berube and Amaral [4]. For programs not included in the collection (*ammp*, *art*, *equake*, *mesa*, and *twolf*), we created an input by modifying the corresponding *ref* input. We use *train* inputs for model testing, and the others for training.

Next, we show the effectiveness of the three regression techniques on each of the three kinds of memory behavior that are used in the predictive model.

Prediction of Accesses per Instruction

The first question for building a model between program inputs and accesses per instruction is the representation of program inputs. Given the close relation between program data size (i.e., the number of distinct data items) and memory behavior, we adopt the approach proposed by Ding and Zhong, characterizing a program input by the estimated data size that can be obtained through distance-based sampling. Distance-based sampling observes data reuses at the beginning of an execution and estimates data size based on long reuse distances [16]. So, in this and the rest experiments, data size is the I_i in the input-behavior pair $< I_i, B_I >$, whereas the B_i is specific to each experiment; it is the accesses per instruction in this experiment.

The left half of Table 3.1 reports the accuracy in predicting accesses per instruction. The three methods produce similar accuracies: 86.43% by LMS, 88.27% by NN, and 88.69% by the hybrid method. Program *equake* shows the lowest accuracy (54.58%) mainly because of its more complex relations between inputs and accesses per instruction. More training inputs and more sophisticated models may be helpful.

Prediction of Distinct Blocks per Cycle

The statistic, distinct blocks per cycle, reflects the average cache requirement of a process. It can be regarded as a product of two factors:

$$DPC = DPI * IPC$$

where, DPI is the average number of distinct blocks accessed per instruction, and IPC is the instructions per cycle. DPI is an attribute solely determined by the program; whereas IPC

Table 3.1: Prediction accuracies of linear (LMS) and non-linear (NN and Hybrid) models.

Programs	Accesses per instruction			DPI		
	LMS	NN	Hybrid	LMS	NN	Hybrid
ammp	89.58	98.76	98.76	39.83	86.72	86.72
art	98.86	94.25	98.86	98.96	94.25	98.96
bzip	75.79	78.62	78.62	67.69	64.05	67.69
crafty	99.54	99.24	99.54	76.31	72.50	76.31
equake	54.58	54.42	54.58	82.27	82.13	82.27
gap	74.75	79.35	79.35	79.87	78.08	79.87
gzip	82.76	86.98	86.98	77.85	66.47	77.85
mcf	90.25	92.45	92.45	89.73	88.11	89.73
mesa	96.39	96.98	96.98	89.43	93.33	93.33
parser	96.02	98.61	98.61	89.49	70.42	89.49
twolf	97.11	98.10	98.10	52.12	86.75	86.75
vpr	81.50	81.50	81.50	96.30	95.28	96.30
Average	**86.43**	**88.27**	**88.69**	**78.32**	**81.51**	**85.44**

is a runtime behavior, attainable from hardware performance counters. The prediction of DPC therefore can be conducted in two steps. Given a new input, an offline-trained model predicts the DPI of the new execution. During the new execution, the DPC can be obtained by multiplying the predicted DPI with the runtime IPC. Therefore, building a predictive model for DPI is the key to the prediction of distinct blocks per cycle.

Because DPI is an average value for an interval, it is determined by the interval length. For an interval containing nothing except one memory access instruction, the DPI is 1, which is the upper bound of DPI under the assumption that one instruction may conduct at most one memory access. As the interval becomes larger, DPI changes non-monotonically, determined by the ratio of non-memory-access instructions and the frequency in which memory-access instructions access a new object. When the interval length becomes large enough to cover at least one access to all the blocks in the program, DPI decreases as the interval length increases.

The DPI used in CAPS is the average DPI of all the reuse intervals[6], computed in the following formula:

[6]The reuse interval of a data reuse is the interval between the previous and the current access to the same data item.

$$w = \frac{\sum_{i=1}^{B} r_i \bar{w}_i}{\sum_{i=1}^{B} r_i}$$

where, B is the number of bars in the reuse signature of the execution, r_i is the number of memory references in bar i, and $\bar{w}_i$ is the average of all the DPIs of the reuse intervals in bar i.

The right half of Table 3.1 shows that NN is slightly more accurate than LMS, 81.51% versus 78.3%. The hybrid model yields an accuracy of 85.4%.

Reuse Signatures Previous work has explored the cross-input predictability of reuse signatures. For example, Ding and Zhong have shown an accuracy of over 94% for the prediction of the reuse signatures of 15 complex programs [16]. Their technique is based on a desirable property of reuse signatures: No reuse distance of an execution can be larger than the data size of the execution. (This property comes from the definition of reuse distance.) They therefore test a set of sub-linear functions in training runs and choose the best one as the model for the prediction of reuse signatures. This work adopts their established technique.

3.5 Evaluation

In this chapter, we present an analytical model (including the variation on cache line level) and a lightweight model for co-run locality prediction. Although the analytical model offers insights on the underlying properties of the problem, the lightweight model is more suitable for real uses. In light of that, this section concentrates on the evaluation of the lightweight model. We first report the results when no data sharing is allowed. After that, we evaluate our model which consider the data sharing on both synthetic traces and traces from real programs.

In order to test the model on traces with various data reuse patterns, we develop a trace generator with the capability to produce data reference traces according to users' specifications, such as the number of distinct data, the frequency of memory accesses conducted by

each cache sharer and data reuse distributions. The parameters that control the generated trace include the following:

- n_1, n_2, ..., n_k: the number of unique data blocks (in the unit of cache lines) in the co-running programs.
- s: the data sharing rate. It is the total number of shared data blocks divided by n_1.
- *distribution*: the distribution of standalone reuse distances. We test the following typical distributions: the random, the exponential ($\lambda = -0.97$), the Normal ($mean = 100$, $std. = 33$). Choosing these distributions is because they have been widely used as the primitive distributions in statistical mixture models [26]; the reuse patterns in many real traces can be regarded as the combination of those distributions [55].

The underlying scheme of the trace generator is a stochastic process similar to the one used in standalone reuse distance studies [54].

3.5.1 Inclusive Reuse Signatures without Data Sharing

A direct evaluation of the basic analytical model is difficult because of the very high time complexity of the model (On the other hand, a short trace cannot capture the necessary statistical properties.) In this section, we instead evaluate the prediction accuracy produced by the simplified cache-line-level model (i.e. Equations 3.5, 3.6 and 3.7.) As it is a special case of the basic analytical model, its evaluation results show some indirect evidence to the validity of the basic model.

To completely expose prediction errors, we use the finest granularity: The width of each bar in the histograms is 1. Figure 3.3 shows the real and predicted inclusive reuse signatures for a trace containing 200000 references from two co-runners. Let n_1 and n_2 stand for the number of unique data blocks (in the unit of cache lines) in the two programs, r for the data references ratio—that is,the total number of references conducted by the second program divided by that of the first program. In the case shown in the graph,

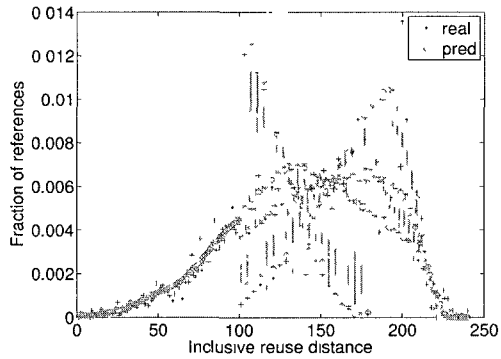


Figure 3.3: The real and predicted inclusive reuse signatures.

$n_1 = 200$, $n_2 = 100$, $r = 0.5$ and the data reuse distances in both programs obey a Normal distribution ($mean = 100$, $std. = 33$). The differences in the numbers of data and references in the two programs result in the drastic fluctuations in the middle part of the graph. Even with those fluctuations, the predicted signature matches the real one well. The accuracy is 95.5%. (Following previous work [16], we define accuracy as $(1 - E/2)$, where E is the sum of the absolute differences between the predicted and the real signatures at every reuse distance. Division by 2 is to normalize the accuracy to $[0, 1]$.)

Table 3.2 presents the accuracies on more traces whose reuse distances are of some typical distributions (the exponential distribution's exponent is $\lambda = -0.97$, the Normal distribution has $mean = 100$, $std. = 33$.) The reason for choosing these patterns is that they are some of the distributions that have been widely used as the primitive distributions in statistical mixture models [26]; the reuse patterns in many real traces can be regarded as the combination of those distributions [55]. The bottom 3 groups above the average row are the results when there are 4 co-runners, among which, the first pair both have $n1$ unique data items, and the second pair both have $n2$, and r is the reference ratio between the two pairs.

The overall average accuracy is 91.4%. For larger-grained histograms (e.g., 1K-wide bars in many real uses), the accuracy would be higher as errors inside a bar would be smoothed out. For instance, the average accuracy for the 4-co-runner cases in our experiment increases

Table 3.2: Accuracy of inclusive reuse signature prediction

distr.	r=0.5		r=1		avg.
	n_1=200 n_2=100	n_1=200 n_2=200	n_1=200 n_2=100	n_1=200 n_2=200	
random	93.8	93.3	94.9	93.3	93.8
expon.	89.4	90.7	93.2	92.3	91.4
normal	95.5	94.3	95.9	94.6	95.1
random+ expon.	95.0	93.3	94.0	93.3	93.9
random+ normal	94.2	93.0	93.9	93.5	93.7
expon.+ normal	94.9	93.2	93.6	94.2	94.0
2random+ expon.+ normal	89.4	86.4	88.2	88.5	88.1
random+ 2expon.+ normal	86.3	85.5	89.0	84.8	86.4
random+ expon.+ 2normal	82.9	90.1	85.0	85.9	86.0
avg.	91.3	91.1	92.0	91.2	91.4

r: the ratio of the number of references conducted by the two (pairs of) programs.
n_1, n_2: the number of distinct data of the two (pairs of) programs.

from 86.0% to 91.3% when a bar spans a distance range of 20. The results also show that the effectiveness of the prediction approach is insensitive to reuse patterns, indicated by the similar accuracy across distributions.

3.5.2 Inclusive Reuse Signatures with Data Sharing

3.5.2.1 Synthetic Traces

Table 3.3 presents the accuracies on a set of traces. The bottom three groups above the average row are the results when there are four co-runners, among which, the first pair both have n_1 unique data items, and the second pair both have n_2.

Following previous work [16], we define accuracy as $(1 - E/2)$, where E is the sum of the absolute differences between the predicted and the real reuse histograms at every

reuse distance. Division by 2 normalizes the accuracy to $[0, 100\%]$. To completely expose prediction errors, we use the finest granularity: The width of each bar in all the histograms used in this experiment is 1.

The overall average accuracy is 87.9%. For larger-grained histograms (e.g., 1K-wide bars in many real uses), the accuracy would be higher as errors inside a bar would be smoothed out. The results also show that the effectiveness of the prediction approach is not significantly sensitive to reuse patterns, indicated by the similar accuracy across distributions. The presence of data sharing reduces the prediction accuracy by 5–7%, reflecting the extra complications caused by the sharing to concurrent reuse distance approximation. For most cases, the prediction accuracy is above 80%, verifying the existence of the statistical connections between concurrent reuse distance and the memory behaviors of individual threads, and demonstrating the capability of the probabilistic model in capturing such connections.

Table 3.3: Accuracy of the Prediction of Concurrent Reuse Distance Histograms

distr.	s=0		s=10%		s=20%		average
	n_1=200	n_1=200	n_1=200	n_1=200	n_1=200	n_1=200	
	n_2=100	n_2=200	n_2=100	n_2=200	n_2=100	n_2=200	
random	94.9	93.3	91.3	90.0	89.7	79.8	89.8
expon.	93.2	92.3	91.1	92.2	93.4	90.1	92.1
normal	95.9	94.6	94.4	80.8	93.4	91.6	91.8
random+ expon.	94.0	93.3	88.5	87.2	84.0	79.0	87.7
random+ normal	93.9	93.5	87.4	90.9	91.6	89.1	91.1
expon.+ normal	93.6	94.2	92.5	79.9	92.2	89.9	90.4
2random+ expon.+ normal	88.2	88.5	83.3	82.0	82.5	81.6	84.4
random+ 2expon.+ normal	89.0	84.8	70.1	72.8	85.3	83.5	80.9
random+ expon.+ 2normal	85.0	85.9	84.1	80.0	81.2	81.2	82.9
average	92.0	91.2	87.0	84.0	88.1	85.1	87.9

s: the sharing ratio. n_1, n_2: the number of distinct data of the co-running programs.

3.5.2.2 Traces from Real Programs

Because instrumentation changes the relative speeds of cache sharers, the real memory traces of co-running threads are difficult to collect on real machines. For our evaluation purpose, we employ a simulator to record the traces. The simulator is constructed based on SIMICS [41] with GEMS [44], a cycle-accurate multiprocessor simulator. The simulated system is a dual-core UltraSPARC architecture with 1MB shared L2 cache.

We simulate three representative PARSEC programs [6]. For each program, we use the fast mode of the simulator to move into the region of interest (the labels to those regions come with the original benchmarks) and then collect memory references in one-million-cycle-long detailed simulation.

Program *swaptions* is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The program uses few (23) locks. There are 27% data that are shared between two threads in the collected memory reference trace. The prediction accuracy by the probabilistic model is 74%. The accuracy is relatively lower than those on synthetic traces. The reason is that this program accesses distinct data elements more frequently than the synthetic traces. The reuse distance tends to span a broader range.

Program *vips* is based on the VASARI Image Processing System (VIPS). It includes fundamental image operations such as an affine transformation and a convolution. The program uses locks intensively. There are totally over 33,000 locks. But there are negligible portion of data that are shared between threads. The probabilistic model is able to predict the concurrent reuse distance by 76% accuracy.

The last program is *streamcluster*. It is an RMS kernel developed by Princeton University that solves the online clustering problem. It is a data-level parallel program. This program uses modest number of locks, but many barriers (129,600). There are 3% data shared between two threads in the generated memory reference trace. The approximated concurrent reuse distance histogram has the highest error, 28%. It is mainly due to its irregular data references.

3.5.3 Predicting Co-Run Performance

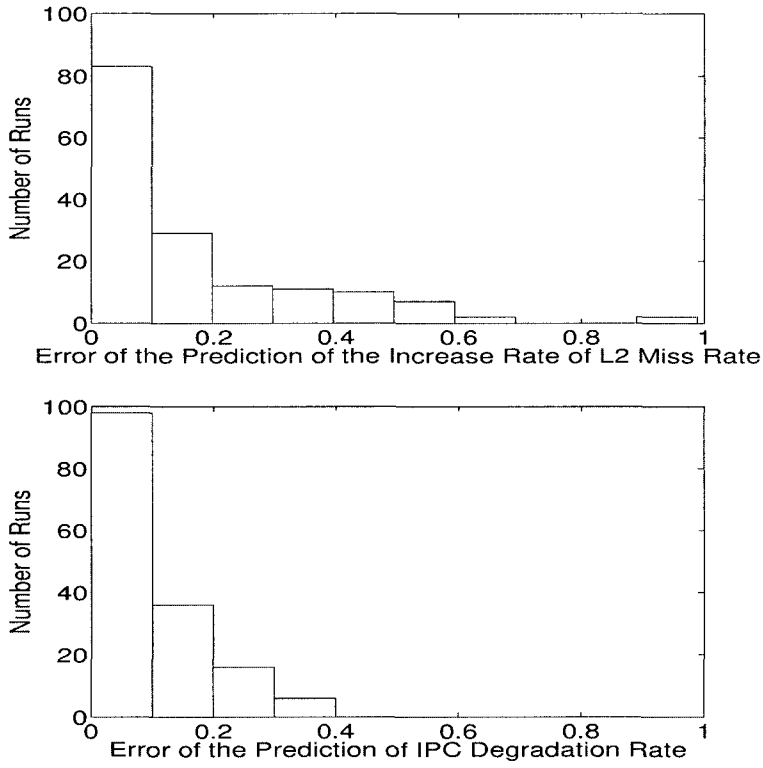


Figure 3.4: The distribution of the errors in the prediction of L2 cache misses and IPC.

We apply Equation 3.9 to all the 78 pair-wise co-runs of 12 programs (including the co-runs of two copies of the same program). Figure 3.4 shows the prediction error of the increase rate of L2 cache miss rate on the Intel Xeon machines, calculated as $|(\delta/r_s)-(\hat{\delta}/\hat{r_s})|$, where r_s (s for single-run) is the real L2 miss rate of the single-run of a program, δ is its increase because of co-run contention, $\hat{r_s}$ and $\hat{\delta}$ are their predicted values respectively from exclusive reuse signature and Equation 3.9. The average error is 16%. The runs having error larger than 20%, except for one program *mcf*, all have a very small ($< 0.6\%$) ratio between their L2 misses and total memory references. The small ratios make the relative errors look large, but the small absolute errors have only minor influence on the prediction of performance degradations, as shown in the IPC graph of Figure 3.4.

In the IPC graph, the prediction error of IPC increase rate is calculated as $|(IPC_c - IPC_s)/IPC_s - (\widehat{IPC_c} - \widehat{IPC_s})/\widehat{IPC_s}|$, where, IPC_s and IPC_c are the real IPCs of the single-run and co-run of a program, while $\widehat{IPC_s}$ and $\widehat{IPC_c}$ are their predicted values. The average error is 9%.

The IPC prediction is through a regression model obtained by offline training, in a way similar to previous explorations [23, 61]. We apply Ridge regression, a regression technique that tolerates feature correlations [26], to a training set consisting of the memory behaviors and IPCs of randomly picked 200 runs. The generated performance model is

$$IPC = 1.9894 - 1.7071h_1 - 4.7019h_2 - 8.8863h_3$$

where, h_1, h_2, and h_3 are the numbers of L1 cache hits, L2 cache hits, and other memory references, divided by the total number of instructions. They are derived from the measured single-run locality and the predicted miss rate increase.

3.6 Related Work

Since the early days in computing, the problem of how to model data reuse patterns and their connections to cache performance [13, 45] has continuously drawn great research interests. The decades of efforts have contributed a solid foundation for understanding the behavior of dedicated cache systems. The single-run reuse distance model used in this work is one example technique from prior dedicated cache research. It is initially proposed as LRU stack distance by Mattson et al. [45], and has later been widely used in locality analysis (e.g., some recent work on program optimizations [5, 77], cross-architecture program performance[43], and memory disambiguation[21].)

But the current understanding to shared cache behavior is much less mature. There has been some work on analyzing the interactions among different threads on the cache in a time-sharing environment. For example, in 1980s, Triebaut and Stone [68] develop a

footprint-based analytical model for cache-reload transients. More recently, Suh et al. [65] design an analytical cache model for estimating the effect of context switching. Their studies mainly focus on predicting the footprint size of a thread as the interactions on cache mainly occur at context switch time; while with shared cache, the interactions happen at almost every cache access. The prediction of footprint size becomes insufficient to address the significantly complicated interactions.

For shared cache on either SMT or CMP architectures, although some studies have tried to characterize the influence of cache sharing on program performance, most of them are either based on certain heuristics or some hardware extensions (e.g., [49, 53].) Chandra et al. [10] propose three models to predict shared cache performance from single-run cyclic stack distances. The models show good prediction accuracy on CMP simulators, but are based on coarse-grained heuristics and approximations and leave the inherent connections between co-run and single-run locality unexposed. It also remains unclear how the models would fit the requirements of proactive cache management in terms of accuracy and efficiency. Berg et al. use sampling techniques to estimate the behavior of CMP cache for a parallel application [3]. Many of recent studies on either SMT or CMP have tried to optimize shared cache performance through either hardware extensions[28, 50, 53], or operating system scheduling [23, 59]. They commonly use reactive schemes by relying on runtime profiling to estimate the co-run performance or cache requirement of programs.

This work is unique in two aspects. Through a rigorous model, it uncovers the underlying connections between single-run and co-run locality, and it initiates proactive management for shared cache through the support of a lightweight predictive model. It is complementary to many previous explorations. The prediction from the shared-cache locality models may provide guidance to many previous cache management techniques. On the other hand, the combination with sampling may further speedup single-run data collection, and certain runtime monitoring and a combination with reactive management may prevent poor predictions from causing inferior decisions.

3.7 Summary

This chapter has described an analytical model to uncover the inherent statistical connections between program single-run memory behaviors and co-run locality. The model offers theoretical insights on the prediction of shared cache performance, laying the foundation for proactive cache management. With those insights, we develop a lightweight model to enable the uses of co-run locality models in proactive cache management. Moreover, we explore the influence of program inputs on job co-scheduling. We also construct a set cross-input predictive models for a set of memory behaviors that are used in the performance predictive model. The results exhibit the potential of combining program behavior analysis by programming systems and global resource management by operating systems. The shared cache behavior analysis may open new opportunities for various execution layers in CMP systems to exploit the shared cache resource more effectively than before.

Chapter 4

Cache-Contention-Aware Proactive Scheduling

4.1 Introduction

The previous two chapters have described the prediction of co-run performance and the design of co-scheduling algorithms. This chapter concentrates on how to integrate these techniques into practical job co-schedulers for both batch and online job scheduling.

In operating systems (OS) research, the recent attempts in alleviating cache contention mainly focus on reactive process scheduling [59, 14, 48, 20, 22, 23, 8]. These techniques typically sample job executions periodically. During the sampling, they track hardware performance counters to estimate the cache requirement of each process and derive a better schedule. (For a system containing multiple CMP chips, a better schedule usually means a different assignment of jobs to processors or a different allocation of CPU timeslices to processes.)

Although these techniques work well under certain conditions, the strong reliance on runtime sampling imposes some limitations on their effectiveness and applicability. The main obstacle is that the sampled behavior only reflects the behavior of a process during a *certain time period* when it co-runs with a *certain subset* of processes. Whereas, good

scheduling needs to recognize the inherent cache requirement of a process and its influence on and from all possible co-runners.

As a result, most prior techniques require both periodic re-sampling and frequent reshuffles of processes among different co-run groups [23, 59].

These requirements not only cause more sampling overhead (cache performance is often inferior during sampling periods) but also limit the applicability of previous scheduling techniques. For instance, cache-fair scheduling needs the sampling of 10 different co-runs (i.e., runs with different co-runners) per process in every sampling phase, and requires the system to contain a mix of *cache-fair* and *best-effort* processes [23]; symbiotic scheduling [59, 14], which samples program performance under various schedules and estimates the best schedule, is difficult to be applied to large problems—the number of possible schedules increases exponentially with the numbers of jobs and processors (e.g., there are 2 million ways to co-schedule 16 jobs on 8 dual-cores).

This chapter attempts to free prior techniques from those constraints by integrating the knowledge of programming systems. Our exploration combines program behavior analysis with operating systems' control of underlying resources. It presents the design of cache-contention-aware proactive scheduling(CAPS). For batch processing, we adopt the performance prediction model in Chapter 3 to predict the co-run performance degradations and then use algorithms proposed in Chapter 2 to find the schedule.

Runtime scheduling has even higher requirement for efficiency. Our solution is cache-contention Competitiveness and Sensitivity models statistically derived to characterize the expected influence that a process may impose to and receive from other processes. We design a dual-queue scheduling system, which evenly separates processes into sensitive and insensitive queues and schedule them accordingly. The scheduling system adapts to the dynamic entries and exits of processes by periodically adjusting the threshold used for process separating.

4.2 CAPS for Batch Processing

This section describes the scheduler for a batch of jobs. In batch processing, the job set and the corresponding inputs are known beforehand. The particular problem to address for CMPs is how to partition the job set into co-schedule groups in order to achieve the best performance. CAPS consists of three components.

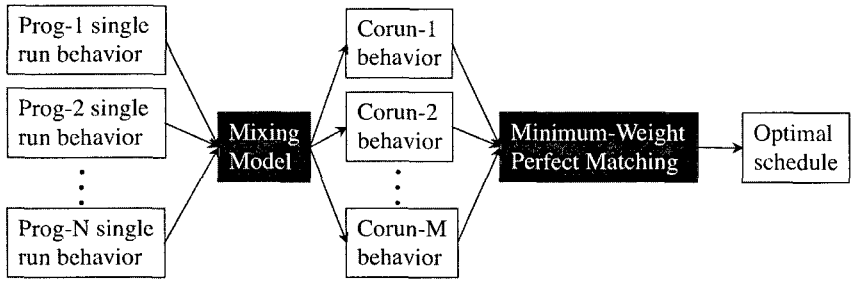


Figure 4.1: The key components of the cache-contention-aware proactive scheduler (CAPS).

As depicted in Figure 4.1, at the heart of CAPS are two components. The first component predicts the performance degradation of each possible co-run using the memory behavior of single-runs of each program. In this framework, we use the techniques proposed in Chapter 3 to predict the cross-input co-run performance. The second component maps the co-run performance to a fully connected graph, with each vertex representing a program, and each edge having a weight equal to the total performance degradation of the co-run of the two vertices. It then applies the minimum-weight perfect matching algorithm to efficiently determine the schedule that minimizes the total of the co-run degradation of all the programs. The detailed algorithm is presented in Chapter 2.

4.3 CAPS for Runtime Scheduling

Unlike in batch processing, the job set in a general computing system usually changes dynamically and thus requires continuous and frequent scheduling. More efficient and adaptive models are imperative. In this section, we propose a scheme for online job scheduling.

4.3.1 Cache-Contention Sensitivity and Competitiveness

To avoid dealing with every possible co-runs, we characterize the statistically expected influence that a process may impose to and receive from random processes through a competitiveness-sensitivity model. *Competitiveness* and *sensitivity* respectively characterize the statistical expectation of the influence that a process may impose on and receive from random co-runners. This model is important for making runtime proactive scheduling scalable. As we will see in Section 4.4, CAPS capitalizes on the model to make sensitive processes co-run with uncompetitive ones to achieve better performance.

4.3.1.1 Sensitivity

The definition of cache-contention sensitivity is as follows:

$$Sensitivity = \frac{\overline{CPI_{co}} - CPI_{si}}{CPI_{si}} \tag{4.1}$$

where, CPI_{si} is the cycles per instruction (CPI) of a process's single run, and $\overline{CPI_{co}}$ is the statistical expectation of the CPI of that process when it co-runs with random processes.

The estimation of CPI_{si} is straightforward: As explained in Chapter 3, we can predict the cache miss rate of a process's single run from its standalone reuse signatures; the corresponding CPI (given the cache miss rate) can be estimated using existing techniques (e.g. [61]).

To estimate $\overline{CPI_{co}}$ in the same way, we have to obtain the statistical expectation of the cache miss rates of the process's co-runs. The number of co-run misses equals the sum of single-run misses and the extra misses caused by co-run contention. Since single-run misses are obtainable as mentioned in the previous paragraph, the problem becomes the computation of the statistical expectation of the number of extra misses. The following corollary of Theorem 1 offers the solution.

Corollary 1 *Let $F()$ be the cumulative distribution function of the DPCs of all programs, and L be the shared cache size. Suppose a process p has H memory references whose*

standalone reuse distances, d_i, are smaller than L (i=1, 2, $\cdots$, H). Let σ_i represent the DPC of the corresponding reuse interval. When process p co-runs with some randomly-picked programs that share no data with p, the expectation of the cache miss rate of the H memory references is

$$\delta = 1 - \frac{1}{H}\sum_{i=1}^{H} F(\sigma_i(L - d_i)/d_i). \tag{4.2}$$

Proof: Let σ' represent the average DPC of the co-runners of p in the reuse interval corresponding to σ_i. Theorem 1 tells us that if and only if $\sigma' < \sigma_i(L - d_i)/d_i$, reference i remains a hit. Since the probability for that condition to happen is $F(\sigma_i(L - d_i)/d_i)$, the expectation of the number of cache hits among the H references is $\sum_{i=1}^{H} F(\sigma_i(L - d_i)/d_i)$. The conclusion follows. ■

With this corollary, we can compute the sensitivity of a process from its DPC and standalone reuse signature. Since references are grouped in bars in reuse signatures, the computation uses a bar as a unit; H thus equals the number of bars whose reuse distances are smaller than L. For computing the $F()$ items efficiently, we build a lookup table for F by using 3.9 billion data reuses from a dozen randomly chosen SPEC CPU2000 programs (included in Figure 4.2). The table contains 200 items corresponding to 200 evenly-spaced points between 0 and 0.237.

4.3.1.2 Competitiveness

We initially intended to use a process's average DPC as competitiveness. But our experiments reveal the strong correlation between the influence a process imposes on and receives from its co-runners. This observation leads to a unified competitiveness and sensitivity model.

Figure 4.2 plots the performance degradation of all the 66 pair-wise co-runs of a dozen SPEC CPU2000 programs (*train* runs) on an Intel Xeon 5150 processor (specified in Section 4.4). In the graph, points on solid curves show the program's own degradation and

points on broken curves show the degradation of its co-runner. For legibility, each program's data are sorted in ascending order of self degradation and then connected into curves. The two curves corresponding to every program show similar trends. The correlation coefficient between all the self and co-runner degradations is 0.75. (As an extra evidence, the coefficient is 0.73 for the 13 SPEC programs shown in Figure 4.5.)

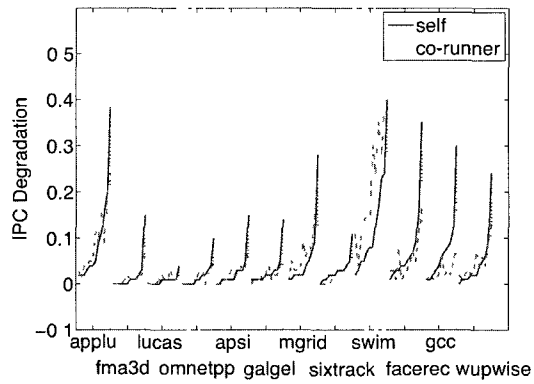


Figure 4.2: Each program has 11 pair-wise co-runs, respectively with each of the other 11 programs. The points on the solid curve show the degradations of this program in those co-runs; the points on the broken curve are of its co-runners. (The points are connected for legibility.) The similarity between the two kinds of curves shows the strong correlations between the degradations of a program and those of its co-runners.

The intuition behind the strong correlation is that, a program that is sensitive to cache contention tends to fetch data from a large portion of the shared cache frequently. Hence, it tends to impose strong influence on its co-runners, that is, it tends to be competitive. As an exception, stream programs are competitive but insensitive. Although they access cache intensively, those programs have few data reuses and thus rely on no cache for performance. Fortunately in offline training, it is easy to detect stream programs thanks to their distinctive data access patterns. The scheduling process, CAPS, treats those programs as competitive programs and pair them with other insensitive programs (detailed next). For other programs, CAPS simply uses sensitivity for competitiveness. This unified model simplifies the design of runtime scheduler.

4.3.2 Runtime Scheduling Policy

The principle of CAPS is to couple sensitive processes with insensitive (thus likely uncompetitive) processes. This section uses Linux as an example to explain how CAPS can be integrated in runtime schedulers.

In default Linux SMP scheduling (e.g., Linux 2.6.23), when a program is launched, one of the CPUs will receive that signal and assign the process to the best available CPU for execution. Each CPU has a scheduler managing the jobs assigned to it.

For CAPS, CPUs are classified evenly into two groups, G_s and G_i, dedicated to sensitive and insensitive processes respectively. For the CPUs sharing a cache, half of them belong to G_s and the others belong to G_i. The scheduler on each CPU maintains a sensitivity threshold h, which is equal to the decayed average of the sensitivities of all the processes that the scheduler has assigned (may or may not to this CPU). Formally, h is computed as follows when the scheduler assigns the nth process:

$$h_n = \alpha h_{n-1} + (1 - \alpha) S_n \tag{4.3}$$

where, α is a decay factor (0 to 1), and S_n is the sensitivity of the newly launched process. The use of the decay factor makes the scheduler adaptive to workload changes. Similar to other factors in OS, its appropriate value should be determined empirically.

When a program is launched, the CPU that receives the launching signal computes the sensitivity of the process, S_n. It then updates h using equation 4.3. If $S_n > h$, it schedules the process to a CPU in G_s, otherwise, to a CPU in G_i. The way to select a CPU inside a group is the same as in the default Linux scheme. (Stream programs are assigned to G_s directly.) For processes without locality models, the scheduler falls back to the Linux default scheduling.

Equation 4.3 attempts to obtain load balance by dynamically adjusting threshold h. If unbalance still occurs due to certain patterns in the sensitivities of subsequent jobs, the

existing load balancer in Linux, which is invoked periodically, can rebalance the workload automatically.

We note on two facts. First, the scheduler makes no change to the default management of run-queues and timeslice allocation in Linux. This is essential for maintaining the proper treatment to priorities. Second, although it is possible for different CPUs to get different h values, some degrees of difference is tolerable for CAPS. Furthermore, during rebalance, the rebalancer can obtain the average of all CPUs' h values and update the h values for every CPU accordingly.

The sensitivity of a program is obtained from its predicted reuse signature and DPC, both of which have shown to be cross-input predictable [16, 30]. But predictive models have to be constructed for each program through an offline profiling and learning process. This step, although being automatic, may still seem to be a burden to scheduling. There are two ways to make it transparent to the users of CAPS. First, the learning step can occur during the typical performance tuning or correctness testing stage in the development of a software. The program developers only need to run the program on several of the inputs they have; whereas, the outcome is beneficial: Besides for scheduling, the predictive locality model can also benefit data reorganization [16], cache resizing [56], and cache partition [37]. In this case, the scheduler can use the model for free. The second solution is to make the learning occur implicitly in the real runs of an application through incremental learning techniques. Through multiple runs, online learner learns the relation between memory behavior and program inputs, and builds the predictive model for co-run locality prediction.

4.4 Evaluation

This section first presents the accuracy of the performance prediction model for shared cache CMPs. It then reports the effectiveness of CAPS for batch processing and runtime scheduling, with overhead analysis at the end.

4.4.1 Methodology

For evaluation, we employ 12 randomly chosen SPEC CPU2000 programs, as shown in Table 4.1, and a sequential stream program (derived from [46] with each data element covering one cache line) on a Dell PowerEdge 1850 server. The machine is equipped with Intel Xeon 5150 2.66 GHz quad-core processors; every two cores have a 4MB shared L2 cache (64B line, 4-way). Each core has a 32KB dedicated L1 data cache. The information shown in Table 4.1 are collected on the *ref* runs of the benchmarks on the Xeon machine. We use PIN as the instrumentation tool [40] for locality measurement, and use the PAPI [7] library for hardware performance monitoring. In the collection of co-run behavior, in order to avoid the distraction from program lengths, we follow Tuck and Tullsen's practice [71], wrapping each program to make it run 10 times consecutively, and only collecting the behavior of co-runs—that is, the runs overlapping with another program's run.

Table 4.1: Performance Ranges of Benchmarks on Intel Xeon 5150

Program	cycles per instruction			L2 misses per mem. acc.(%)		
	single-run	co-run-min	co-run-max	single-run	co-run-min	co-run-max
ammp	1.01	1.03	1.31	0.51	0.60	1.6
art	0.93	0.96	1.55	0.0028	0.095	3.8
bzip	0.49	0.49	0.66	0.11	0.18	0.76
crafty	0.72	0.73	0.80	0.00010	0.0028	0.21
equake	1.28	1.38	2.13	3.8	3.9	4.5
gap	0.91	0.91	1.16	1.3	1.5	1.6
gzip	0.72	0.72	0.77	0.078	0.079	0.14
mcf	2.47	2.70	4.84	4.4	5.0	8.6
mesa	0.51	0.52	0.56	0.23	0.26	0.38
parser	1.15	1.18	1.50	0.31	0.44	1.2
twolf	1.06	1.07	1.24	0.0014	0.0015	0.40
vpr	1.06	1.09	1.44	0.0053	0.0067	0.015

4.4.2 CAPS for Batch Processing

To evaluate the effectiveness , we measure 4 types of schedules: the optimal, worst, CAPS, and random schedules. We obtain the optimal and the worst schedule by a brute-force search among all possible schedules. We obtain the random schedule results by randomly

choosing 100 schedules and taking their average performance, which correspond to the default scheduling in the current CMP systems that are oblivious to cache contention.

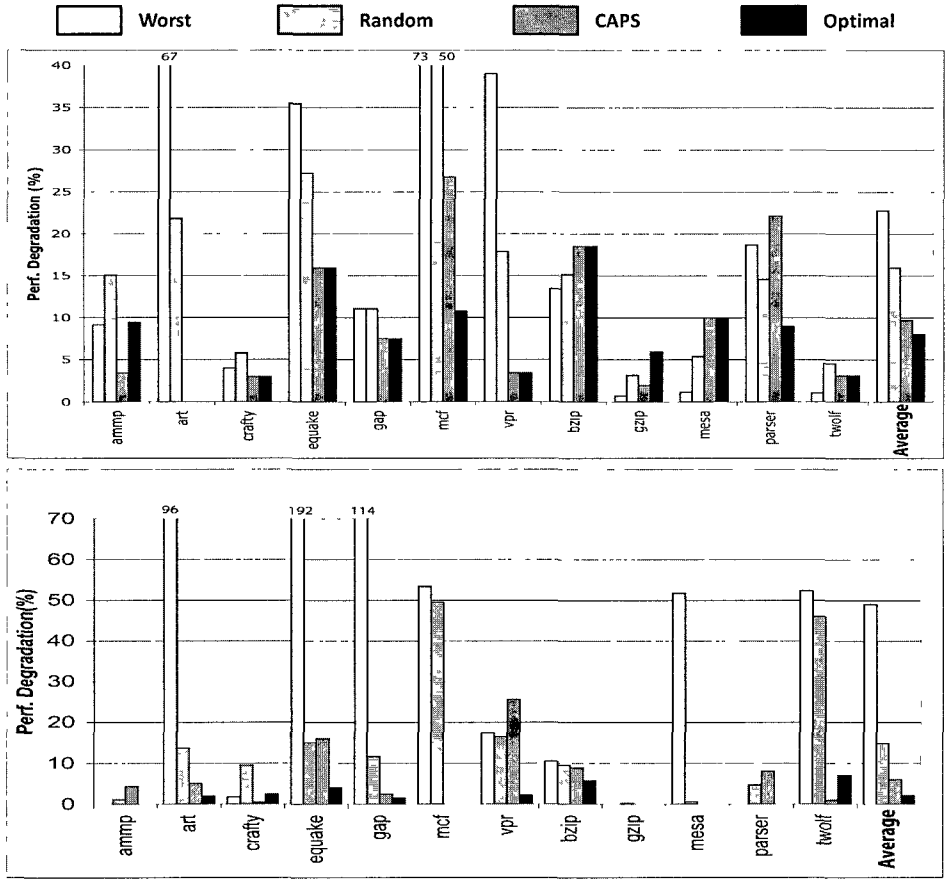


Figure 4.3: Performance degradation on dual-core (top graph) and quad-core (bottom graph) systems by different schedulings.

Table 4.2 shows the detailed coupling of the 12 programs in the worst, CAPS, and optimal schedules.

In the dual-core case (Intel Xeon 5150), except the two italic groups, the CAPS schedule matches the optimal schedule well. The two mismatches are mainly due to the performance prediction errors of some co-runs with the program *ammp*. The top graph in the figure contains the performance degradation of the co-runs in each schedule, measured by the CPI increase divided by the corresponding single-run's CPI.

Table 4.2: Detailed Coupling of Programs under Different Schedules

Optimal schedule		CAPS schedule		Worst schedule	
ammp+parser	art+crafty	*ammp+gzip*	art+crafty	ammp+bzip	art+mcf
bzip+gap	equake+mesa	bzip+gap	equake+mesa	crafty+mesa	equake+vpr
gzip+mcf	twolf+vpr	*mcf+parser*	twolf+vpr	gap+parser	twolf+gzip

On average, the mismatch causes the co-runs in CAPS schedule 1.6% more performance degradation than the optimal schedule. Compared to the random schedule result, CAPS schedule improves 9 programs' performance by 1.2% to 23.68%. As a tradeoff, it meanwhile worsens 3 programs' performance by 3.4%, 4.6%, and 7.6%. On average, the co-runs in CAPS schedule degrade performance by 9.7%, and outperform the random schedule result by 6.2%. It is worth to note that random scheduling may group some programs in the way the worst scheduling does; the consequence is severe: 67% degradation for *art*, 73% for *mcf*, and 22.8% on average. CAPS avoids those traps, making co-runs 13.1% faster than the worst schedule on average. (Note that our goal is to minimize the *overall* rather than each individual program's degradation. So, it is normal for certain programs to run better in the worst schedule than in other schedules.)

In the quad-core case (AMD Opteron), although there are more mismatches between the optimal schedule and the CAPS schedule, their average degradations are still similar. The CAPS schedule reduces the average performance degradation of the random schedule by 60%. Among the 12 programs, 7 of them have degradation reduction of more than 63%; 4 of them have slightly more degradations.

Comparison to Reactive Co-scheduling Reactive co-scheduling usually tries different schedules in a sampling phase and chooses the best one for the following execution phase. Because of the possibility of behavior changes after a sampling phase, reactive co-scheduling conducts resampling periodically. Previous reactive co-scheduling studies [59, 23] typically use 1 billion instructions as the length of a period, maintain a 1 to 10 ratio between the length of a profiling phase and the following execution phase in a period, and run each schedule for 10 million instructions in the sampling phase. In the dual-core case of the 12-program experiment, there are totally 20790 possible schedules; the probability

for a sample phase to cover the optimal schedule is 0.05%, and the probability is 4.7% for covering one of the top 100 schedules.

Unfortunately, it is difficult to conduct a direct comparison of the proactive co-scheduling performance with the performance obtained by previous reactive co-scheduling techniques, mainly because most of them have been implemented on hardware simulators with modifications to operating systems and hardware. To gain some insights on the comparison, we instead estimate the results of reactive co-scheduling in an ideal case, by assuming no behavior changes through the entire executions of the programs—that is, the schedule selected in the sampling phase would work the same in the following execution phase—and there is no process migration overhead. We can then compute the statistical expectation of the total degradation in the execution phase as $\sum_{i=1}^{K} p(s_i)d(s_i)$, where $p(s_i)$ is the probability for a schedule s_i to be chosen as the best schedule in a sampling phase, $d(s_i)$ is the total co-run degradation of all programs under schedule s_i, and K is the total number of possible schedules, equaling 20790 in our experimental setting. During the sampling phase, the statistical expectation of the degradation is just the statistical expectation of the degradations of all possible schedules, because the schedules to try in sampling phases are chosen randomly. On the data collected on the 12 programs used in the proactive co-scheduling experiment, the statistical expectation of the average degradation is 12.1%, 26% more than the proactive co-scheduling results. For a larger problem, the probability to cover top schedules would become even smaller.

We acknowledge that if the sampling ratio increases, the probability of choosing the optimal schedule would increase. But as the problem size increases, the probability increase would diminish quickly. Furthermore, the benefits from the chosen schedule would decrease as the execution phase becomes smaller.

Overhead of CAPS The overhead consists of two sections: the prediction of cache contention between every group of programs, and the computation of the scheduling algorithm. Let N represent the number of programs. The worst-case time complexity of the three segments are respectively $O(N^2)$, and $O(N^4)$ for a given shared-cache size. We mea-

sure the overhead of each part of CAPS on dual-cores with program numbers ranging from 4 to 1024. Figure 4.4 shows the data in the logarithmic scale. When the number of jobs is small, the scheduling time dominates the overhead. But as the number of jobs increases, the weight of co-run performance prediction increases and reaches 93% in the case of 1024 programs. The results suggest that even though the scheduling algorithm has higher worst-case time complexity, it weights less than the co-run prediction overhead for problems of reasonable size, thanks to the scalable implementation of the blossom algorithm [11]. The total overhead for 1024 programs is 0.21 seconds.

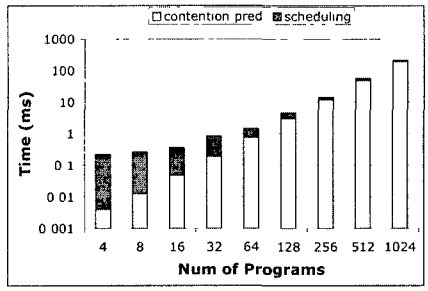


Figure 4.4: Overhead of job co-scheduling

4.4.3 CAPS for Runtime Scheduling

The focus of our evaluation is the examination of the effectiveness of the unified sensitivity model in serving as a locality model for shared-cache-aware scheduling. To avoid distractions from the many random factors (e.g., job arriving time, load balance) in online schedulers, we use offline measurement to uncover the full potential.

We compute the sensitivities of the programs from their reuse signatures and DPCs, based on which, we separate the 12 SPEC programs into two equal-size classes shown as the two sequences of *caps-pred* below. For comparison, we report the ideal separation as *caps-real*. We obtain them by first running all possible pairs of the 12 programs, and then taking the average co-run degradation of each program as its real sensitivity. In both separation results, we list the programs in descending order of sensitivity.

caps-pred:

Sensitive:	mcf	art	equake	vpr	parser	bzip
Insensitive:	twolf	ammp	crafty	gap	mesa	gzip

caps-real:

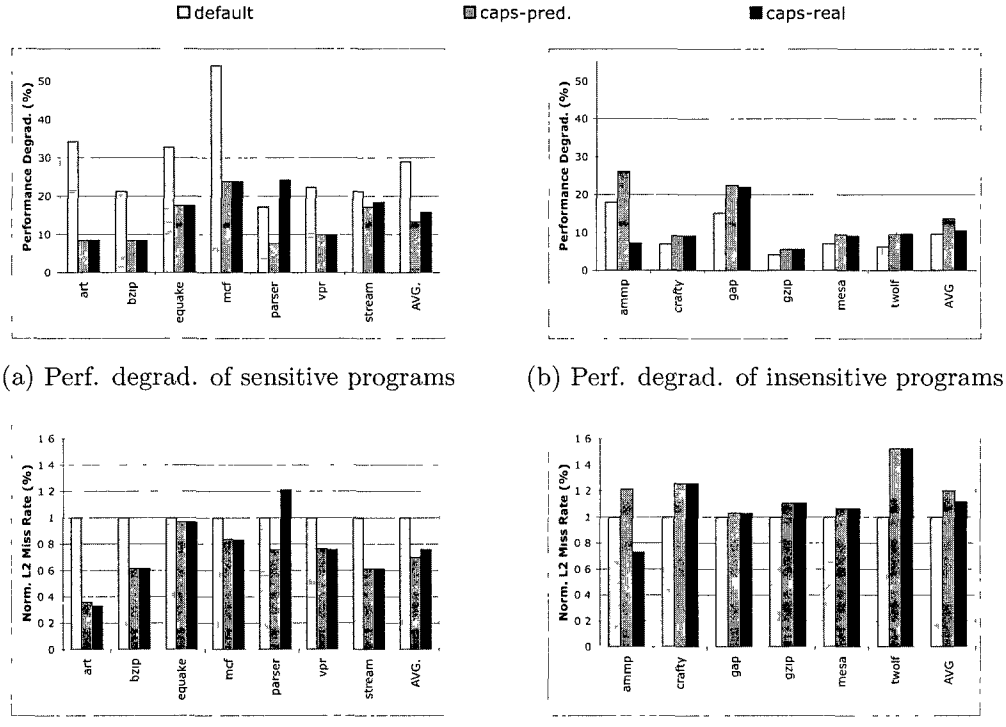
Sensitive:	mcf	equake	art	vpr	bzip	ammp
Insensitive:	parser	gap	crafty	mesa	twolf	gzip

The sequences, although differing in the relative positions of the benchmarks, only mismatch on two programs, *parser* and *ammp*. Two reasons cause the differences: locality prediction errors and the difference between statistical expectation and a particular problem instance. We note that CAPS has good tolerance to ordering difference: As long as programs are put into the right sequences, the order inside a sequence has no effects on CAPS. This property is essential for making the lightweight locality prediction applicable for CAPS.

We compare the performance result of CAPS on predicted sensitivities (denoted as *caps-pred*) with the results of the default Linux scheduler (*default*) and CAPS on real sensitivities (*caps-real*). We measure the performance of a program by **degradation factor**, defined as $(CPI_{co} - CPI_{si})/CPI_{si}$, where, CPI_{co} and CPI_{si} are the respective CPIs of the program's co-run and single run. Following prior work [74], we measure the fairness of a schedule by **unfairness factor**, defined as the coefficient of variation (standard deviation divided by the mean) of the normalized performance (IPC_{co}/IPC_{si}) of all applications.

To prevent randomness from obscuring the comparison, we obtain a program's performance in a schedule by averaging the performance of all the program's co-runs that are allowed by the schedule. The *default* scheduler, for example, allows all 12 possible co-runs per program, whereas *caps-pred* and *caps-real* allow a program to run with only the programs in a different class.

Figure 4.5 shows the performance of the three schedulers, with sensitive programs (judged by *caps-pred*) on the left and insensitive programs on the right. For sensitive programs, *caps-pred* reduces performance degradation by 4% to 30.2% (15.7% on average); as a tradeoff, insensitive programs have 1.4% to 8.1% more degradation (4.1% on aver-



(a) Perf. degrad. of sensitive programs

(b) Perf. degrad. of insensitive programs

(c) L2 miss rates of sensitive programs

(d) L2 miss rates of insensitive programs

Figure 4.5: Performance degradation and normalized L2 miss rates by different scheduling

age). In comparison, *caps-real* shows 2.5% less reduction for sensitive programs and 3.3% more for insensitive programs than *caps-pred*. It is important to note that the goal of job co-scheduling is to increase the overall computing efficiency of the system rather than maximize the performance of each individual program. So it is normal that some programs (e.g. *parser*) perform better in *caps-pred* than in *caps-real*.

Table 4.3 reports the performance, normalized to the default performance, of each program when they run in *caps-real* and *caps-pred*. The sensitive programs show 12% and 14% speedup on average. All of them have speedup over 11% except *parser* and *stream*. In *caps-real*, *parser* has 6% slowdown because it is classified as insensitive programs and co-runs with sensitive programs. The small speedup of *stream* is consistent with our intuition conveyed in Section 4.3.1.2—such programs are competitive but insensitive for their

special memory access patterns. It is remarkable that the significant speedup for sensitive programs comes with almost no slowdown of insensitive programs. The average slowdown is 1% in *caps-real* and 3% in *caps-pred*. The small slowdown is no surprise given that those program are insensitive to cache sharing. The program *ammp* shows 10% speedup in *caps-real* because the scheduler labels the program as a sensitive program and lets it co-run with insensitive programs.

The intuition behind the effectiveness of CAPS is that it successfully recognizes the programs to which cache contention matters significantly. By giving an favorable schedule to those programs, CAPS accelerates them without hurting the programs that are not sensitive to cache contention.

Table 4.3: Whole-Program Speedup Brought by CAPS

Sensitive Programs			Insensitive Programs		
Programs	*caps-real*	*caps-pred*	Programs	*caps-real*	*caps-pred*
art	1.24	1.24	ammp	1.10	0.94
bzip	1.12	1.12	crafty	0.98	0.98
equake	1.13	1.13	gap	0.94	0.94
mcf	1.24	1.24	gzip	0.99	0.99
parser	0.94	1.09	mesa	0.98	0.98
vpr	1.11	1.11	twolf	0.97	0.97
stream	1.03	1.02	–	–	–
Average	1.12	1.14	Average	0.99	0.97

Table 4.4 contains the overall performance degradation factors and unfairness factors of the schedules. The two *reduction* columns report the relative reduction ratios of *caps-pred* and *caps-real* compared to *default*. Schedule *caps-pred* reduces degradation factor by 32.6% and unfairness factor by 46.9%, respectively 1.3% and 2.4% less than *caps-real*.

Figure 4.5 (c) and (d) show the normalized L2 miss rates (L2 misses per memory reference) collected using PAPI library [7]. Although they roughly match the performance results, the L2 miss rates impose different influence on the programs. For example, the 52% more L2 miss rates of *twolf* only cause 3.2% performance difference, while 3.3% less miss rates of *equake* reduce 15% performance degradation. This difference is due to bus-contention differences and the different significance of L2 misses. The L2 miss rates of *twolf*

are hundreds of times smaller than those of *equake*. This agrees with the fact that both *caps-pred* and *caps-real* label *twolf* insensitive and *equake* sensitive.

Table 4.4: Overall Performance Degradation Factors and Unfairness Factors

	Performance Deg. (%)		Unfairness (%)	
	factor	reduction	factor	reduction
default	20.0	–	11.6	–
caps-pred	13.5	32.6	6.2	46.9
caps-real	13.4	33.3	6.0	48.5

These results demonstrate the potential of the locality model in supporting job co-scheduling. The performance of actual on-line schedulers depends on many other factors, such as the job arrival time and order, system load balance and its dynamic adjustment, job priorities, and so forth. c

Overhead of CAPS. The major runtime overhead of CAPS consists of the prediction of standalone reuse signatures and the computation of sensitivities, both determined by the granularity of standalone reuse signatures. Since reuse distances smaller than cache size are more critical for CAPS, reuse signatures organize them in linear scale (1K distance per bar), and use log scale for others. Because each bar in a signature corresponds to one linear function, there are $A + log(N/L)$ linear functions to solve in the reuse-signature prediction, where, A is the number of bars in the linear range, N is program data size (the upper bound of reuse distance), and L is cache line width. The computation of sensitivity relies on only reuse distances smaller than cache size, because only those references can be the victims of cache contention. Thus, the time complexity is $O(A)$.

In our experiments, $L = 64$, A is 64 and N is from 32,606 (*crafty*) to 4.1 million (*gap*) with average of 1.0 million. The numbers of linear functions range from 79 to 86 per program. The computation cost of CAPS is negligible.

4.4.4 Influence of Prediction Errors on Co-Scheduling

We feed CAPS the predicted memory behaviors to test the influence of the prediction errors on co-scheduling. Figure 4.6 shows the average performance degradation of the benchmarks included in Table 4.1. The baseline is an *a posteriori* schedule, which is the best over all possible schedules. We obtained it by applying the minimum-weight perfect matching to all real co-runs. (Recall that the algorithm minimizes the total degradation.) The *random* bar shows the average result of 100 random schedules. It reflects the performance of the default scheduler in the current CMP system.

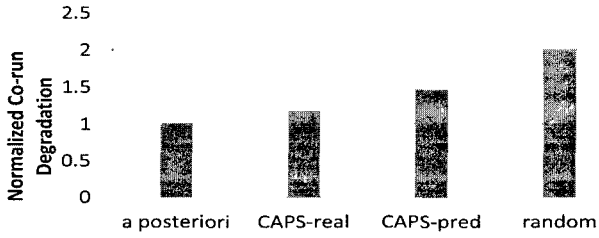


Figure 4.6: The average performance degradation under different schedules. The "a posteriori" schedule is the best schedule obtained on all co-run information; "CAPS-real" is the schedule by CAPS on real single-run behaviors; "CAPS-pred" is the schedule by CAPS on single-run behaviors predicted by the models described in Chapter 3; "random" reflects the default schedule in the CMP system.

4.5 Related Work

Recent years have seen a number of studies on scheduling in CMP. Some concentrate on scheduling threads in a single application. For example, thread clustering [67] tries to recognize patterns of data sharing using hardware performance counters and locates threads accordingly. The technique cannot apply to the problems discussed in this chapter as no data are shared among jobs. Some studies [39] tackle the scalability and fairness of scheduling on CMP, but without considering interferences on shared cache in the fairness criterion. Some studies [31, 70] conduct theoretical analysis to uncover the complexity of optimal co-scheduling on CMP. They are useful for offline analysis but not for runtime scheduling.

This section concentrates on the studies that schedule independent jobs to reduce the interferences on shared cache. Most of those studies have used simulators (e.g., [23, 20, 53, 59]), whereas, we use a real machine for all the experiments. Furthermore, CAPS has applicability different from previous techniques (elaborated next). We hence concentrate on qualitative comparisons.

First, the applicability of CAPS differs from prior techniques. Unlike techniques based on cache activity vectors or other hardware extensions (e.g., [20, 53, 64]), this work is a pure software solution applicable to existing systems. On the other hand, hardware extensions may reveal fine-grained cache conflicts, complementary to the coarse-grained locality information used in this work.

Previous explorations in scheduling for CMP or SMT rely on either hardware performance counters or offline memory profiling, showing different applicability from CAPS. The cache-fair scheduling [23] from Fedorova et al. is applicable when the processes have various cache-access patterns and have already been labeled either *cache-fair* or *best-effort*. Its main goal is performance isolation, accomplished by controlling CPU timeslice allocation instead of process assignment. Zhang et al. use hardware counters to guide scheduling on SMP machines without shared caches [74]. Snavely et al. have proposed symbiotic scheduling, which is based on sampling of various co-runs [59, 14], suiting the problems having a small number of jobs and processors. Some explorations use offline collected memory information to guide scheduling [20, 10]. They use the same program inputs for training and testing, not applicable to input-sensitive programs.

CAPS overcomes the above constraints, but requires each process of interest to be equipped with a cross-input predictive locality model (whose construction, fortunately, can be transparent to the users of CAPS as discussed in Section 4.3.2). The combination of CAPS with runtime sampling-based techniques may be beneficial: The former overcomes scalability issues, and the latter offers on-line adaptivity. In addition, the combination of CAPS with locality phases [56] may add adaptivity to phase shifts as well.

4.6 Summary

This chapter, based on the concept of concurrent reuse distance, develops the design of cache-contention aware proactive scheduling(CAPS). For batch processing, we adopt the performance prediction model in Chapter 3 to predict the co-run performance degradations and then use algorithms proposed in Chapter 2 to find the schedule. For online processing, it presents a lightweight locality model for shared-cache contention prediction. The model offers the basis for a runtime contention-aware proactive scheduling system. Experiments on a recent CMP machine demonstrate the effectiveness of the technique in alleviating cache contention, improving both system performance and fairness. On the high level, this work shows the potential of combining program behavior analysis by programming systems and global resource management by operating systems. Interactions between these two layers may also help other issues in computing systems.

Chapter 5

Other Work

This chapter briefly describes two techniques that relate with the maximization of the computing efficiency on CMP. The first one is correlation based proactive program behaviors. As mentioned at Chapter 3, our performance prediction model is input sensitive. To accurately predict the co-run performance degradation, we need characterize program input efficiently. The program behavior analysis offers a possible solution to tackle this problem. The second is adaptive software speculation. This technique can dynamically reduce the instances of useless speculative threads, hence increasing the system throughput and reducing energy consumption.

5.1 Correlation-Based Program Behavior Analysis

Accurate prediction of program behaviors is the basis of various program optimizations. Program behaviors refer to the operations of a program and the ensuing activities of the computing system, in relation to the input and running environment. Examples include memory references, data values, function calling frequencies, and so on. The prediction of program behaviors critically determines how optimizers transform a program and the resulting performance. As the complexity in modern hardware and software continuously grows, accurate behavior prediction becomes both more important and more challenging than before.

Besides accuracy, two other properties of behavior prediction are essential for optimizations: scope and timing. The scope of a prediction may be a small execution interval, a loop, a procedure, or the entire program. The larger the scope is, the more likely the optimizer is able to avoid local-optimum traps when making optimization decisions. The third property, the timing of prediction, refers to when a prediction can occur. The earlier the prediction occurs, the earlier an optimization can happen, and the larger the portion of the execution that may benefit from the resulting code. We also call the earliness the *proactivity* of a prediction.

In existing program optimizers, behavior predictions are based on either training runs (in profiling-based optimizers) or runtime sampling (in runtime optimizers). Their strategies are essentially the same: using the behaviors of a program component (e.g., a procedure or loop) observed previously (in either a training run or the earlier part of the current execution) to predict the future behaviors of the *same* component. This strategy, although effective for many programs, can lead to a *proactivity-adaptivity dilemma*: Predictions based on training runs have good proactivity, but cannot adapt to input changes, whereas, predictions based on runtime sampling have good adaptivity but limited scope and proactivity.

Recent studies show that prediction based on program inputs may gain the strengths of both approaches, improving optimizations significantly. For instance, improvements of 7%–21% have been observed on a variety of Java programs [42]. However, that approach relies on programmers' manual specifications on program inputs. An automatic solution to the proactivity-adaptivity dilemma remains an open question.

We attack the problem by exploiting the correlations among the behaviors of program components. The intuition is simple. Consider the trip-counts (number of iterations) of two loops, L1 and L2. Suppose that they strongly correlate with each other (e.g., the trip-counts of L1 are always about double those of L2). Then, as soon as the trip-counts of one of them become known in an execution, the trip-count of the other will be easily predicted.

In this work, we first find that strong statistical correlations exist not only among the behaviors of different program components commonly, but also among different types of

program-level behaviors (e.g., loop trip-counts versus data values). Even though conditional branches in a program sometimes weaken the correlations between loops and basic block execution frequencies, overall, strong statistical correlations exist between loop trip-counts, and from loop trip-counts to other types of behaviors. It suggests the possibility of using the correlations for runtime behavior prediction. When the values of certain types of behaviors of some program components (e.g., a set of loop trip-counts) are exposed in an execution, we may use them as the predictors of the behaviors of other (to-be-executed) components in the program. This kind of prediction is both proactive, occurring before the execution of the other program components, and adaptive, being specific to the current input data set.

We then introduce a technique to exploit the correlations for program behavior prediction and optimizations. The technique centers on a new concept, *seminal behaviors*, which refers to a small set of behaviors that strongly correlate with most other behaviors in the program, and meanwhile, expose their values early in typical executions.

We select two types of behaviors as the candidates for seminal behaviors. The first is program *interface behaviors*, which mainly include the values directly obtained from program inputs. Specifically, this type of behaviors include the values obtained directly from command lines and file operations. The second type of behaviors we include are the trip-counts of all the loops in the program. This inclusion is due to the importance of loops and the correlations between loop trip-counts and other program behaviors.

From the definition of seminal behaviors, we know that they must be able to lead to accurate prediction of other behaviors. For a given set of behaviors B, we define *predictive capability* of a set S as the number of behaviors in the set $B - S$ that can be predicted from S with an accuracy above a predefined threshold (80% in this study).

For the reduction of complexity, we take a simplification as follows. We limit B to loop trip-counts during the examination of the predictive capability of different candidate behavior sets. The intuition is that because there are strong correlations between loops and other types of behaviors, the sets selected in this way are likely to show good predictive capability on other types of behaviors as well.

The computation of predictive capabilities in our experiments is based on the standard 10-fold cross-validation [26]. It works iteratively. Suppose we did N profiling runs of a program, and obtained N instances of S and B. In each iteration, 9/10 of the N instances are used to construct predictive models from S to B, and the other 1/10 are used to test the model for prediction accuracy.

We take an incremental approach, which gradually builds a number of affinity lists. *An affinity list* is a list consisting of two sets of behaviors, a header set and a body set, such that the values of the behaviors in the header can lead to accurate prediction of the values of those behaviors in the body.

The union of the headers of the affinity lists forms a possible seminal behavior set as all other candidate behaviors are predictable from it. These header sets may be ranked in a descending order of the sizes of their bodies. The exclusion of the low-rank header sets may have little influence on the prediction of most behaviors.

We employ two standard regression techniques, namely LMS linear regression and Regression Trees [26]. The former handles linear relations among behaviors, the latter for non-linear relations. The construction process applies Regression Trees only if the linear regression results are not good enough (automatically assessed through cross-validation). During the construction of the first affinity list, the standard forward stepwise feature selection [26] is used so that only important interface behaviors are stored in the header.

Both LMS and Regression Trees models are efficient to build and use. The resulting models are represented by only a small number of coefficients (for linear models) and questions (for Regression Trees). (We limit the tree size to be no greater than 10.)

In our experiment, most of performance related program behaviors can be predicted from the seminal behaviors with over 90% accuracy. The high accuracy indicates that this technique can be used efficiently for cross-input adaptation.

5.2 Adaptive Speculation

Recent years have seen a rapid shift of processor technology to favor chip multiprocessors. Many existing programs, however, cannot fully utilize all CPUs in a system yet, even though dynamic high-level parallelism exists in those programs. Examples include a compression tool processing data buffer by buffer, an English parser parsing sentence by sentence, and an interpreter interpreting expression by expression, and so on. These programs are complex and may make extensive use of bit-level operations, unrestricted pointers, exception handling, custom memory management, and third-party libraries. The unknown data access and control flow make such applications difficult if not impossible to parallelize in a fully automatic manner. On the other hand, manual parallelization is a daunting task for complex programs, especially for those pre-existing ones. Moreover, the complexity and the uncertain performance gain due to input-dependence make it difficult to justify the investment of time and the risk of errors of the manual efforts.

Software speculation has recently shown promising results in parallelizing such programs [15, 69]. The basic idea is to dynamically create multiple speculative processes (or threads), which each skips part of the program and speculatively executes the next part. As those processes run simultaneously with the main process, their successes shorten the execution time.

But speculative executions may fail because of dependence violations or being too slow to be profitable. In systems with no need for rollback upon speculation failures—such as the *behavior-oriented parallelization* (*BOP*) system [15], failed speculations result in the waste of computing resources (e.g., CPU and memory) and hence inferior computing efficiency. The waste is a serious concern especially for multi-programming or power-constrained environments (e.g., laptops, embedded systems.) For systems where rollback is necessary, an additional consequence is the degradation of program performance.

Therefore, the avoidance of speculation failures is important for the cost efficiency of modern machines. Previous studies—mostly in thread-level speculation—have tried to

tackle this problem through profiling-based techniques (e.g., [18, 72, 35].) The main idea is to determine the regions in a program that are most beneficial for speculation by profiling some training runs.

The strategy, however, is often insufficient for coarse-grained software speculation, because of the *input-sensitive* and *dynamic* properties of the parallelism. In a typical application handled by software speculation, the profitability (i.e., likelihood to succeed) of a speculative region often differs among executions on different program inputs, or even among different phases of a single execution. The profiling-based region selection can help, but unfortunately, is not enough for software speculation to adapt to the changes in program inputs and phases.

This work proposes adaptive speculation. The goal is to make *BOP* avoid unprofitable speculations but meanwhile keep profitable speculations unaffected, hence improving the cost-efficiency without sacrificing the parallelized program performance. As a side benefit, adaptive speculation can also make *BOP* easier to use by allowing users to label *PPRs* more flexibly: The unprofitable *PPRs* will be turned off automatically.

It is however difficult to predict speculation profitability through program code analysis, because the profitability depends on program inputs and runtime behavior. By treating the problem as a statistical learning task, we develop two adaptive algorithms that are able to learn the profitability patterns of a *PPR* during runtime. A complexity in the learning is that the profitability of the earlier instances is not always unveiled: If a *PPR* instance is not executed speculatively, *BOP* cannot determine its profitability. The two algorithms manage to learn from the partial information and adapt themselves to the dynamic changes in profitability patterns.

The first algorithm is an extension to last-value predictors and uses a dynamically adjustable threshold for adaptation. The second algorithm exploits long-term history and offers more flexibility in control by separating different factors apart. Both algorithms are reconfigurable, providing some "knobs" for users to adjust the tradeoff between parallelism exploitation and cost savings.

We implement both techniques in *BOP* [15], a recent software speculation system. Evaluations on a chip multiprocessor machine demonstrate that the proposed techniques are effective in preventing unprofitable speculations without sacrificing profitable ones. The techniques help *BOP* save a significant amount of cost, and meanwhile, cause little decrease but often increase to the program performance. The cost efficiency is enhanced significantly.

5.3 Summary

This section briefly discusses two techniques related with the enhancement of computing efficiency on CMP. The first is correlation-based program-level behavior analysis. By employing a set of statistical learning techniques, we can use the values of a small set of seminal behaviors to predict other kinds of program behaviors. This technique can facilitate the cross-input adaptation in job co-scheduling. The second one is adaptive software speculation. It can dynamically reduce the number of useless speculative threads and hence improve the overall schedule quality on the system.

Chapter 6

Conclusion

On-chip resource sharing among sibling cores causes resource contention on CMP, considerably degrading program performance and system fairness. Job co-scheduling attempts to alleviate the problem by assigning jobs to cores appropriately. There are two challenges for obtaining a good schedule. First, how to find the best schedules if we have the information that how the jobs interact with other jobs on the same CMP. Second, how to predict the interaction among jobs. This dissertation proposes several techniques for answering these two questions.

This dissertation first concentrates on the analysis and design of algorithms based on the assumption that we know the performance degradations of all the possible co-run cases. We investigate the scenarios with two different goals: minimizing total cost and minimizing the makespan.

We prove that the job co-scheduling is NP-Complete on systems with more than two cores per chip for both cases. For dual core system without job migrations, we propose optimal algorithms for both goals. If the goal is to minimize the total cost, the problem can be solved by adopting a classic graph algorithm, minimal weighted perfect matching algorithm. The optimal solution can also be obtained for minimizing the makespan by using graph perfect matching algorithms.

If the number of cores is greater than two, the optimal solution cannot be acquired in polynomial time unless P=NP. In this case, we present a set of heuristics to approximate

the optimal schedule. When the job migration is not allowed, we proposed a hierarchical algorithm and a greedy algorithm. If the job migration is allowed, we design a A-star and cluster based algorithm and local matching algorithms. Experiments on both real and synthetic problems validate the optimum of the results by the optimal co-scheduling algorithms, and demonstrate the effectiveness of the heuristics-based algorithms in producing near-optimal schedules with good efficiency and scalability.

The second part of this dissertation aims to understanding the interaction among programs running on the same CMP. We present some techniques to reveal the influence of cache sharing on program cache performance by uncovering the inherent connections between the locality of program single-runs and that of their co-runs.

We formulate the problem of predicting co-run cache contention as a problem of the prediction of program *inclusive reuse signatures*—-which is a summary of LRU stack distances on a shared cache with all cache sharers' data references considered—-and conduct a theoretical analysis to expose the inherent statistical connections between single-run memory behavior and co-run inclusive reuse signatures. The theoretical analysis sheds insights on the prediction of co-run performance from single-run data locality. In light of that, we develop a lightweight model for efficiently predicting co-run data locality (or cache usage) from the memory reference patterns of the programs' single-runs. The high efficiency of the model is the key to its uses in shared-cache management. It achieves the efficiency mainly by capitalizing on the connection between time and locality. Finally, we implement a proactive job co-scheduling system to demonstrate the potential benefits of the co-run locality model. The scheduling achieves close-to-optimal results, cutting cache-contention caused performance degradation by as much as 63% on average, improving program performance by 9% on average (up to 50% for individual applications.)

To make the predictive model lightweight enough for online scheduling, we further reduce the overhead of the model by introducing a competitiveness and sensitivity model. Competitiveness and sensitivity respectively characterize the statistical expectation of the influence. This model can compute the sensitivity of a program online and then schedule

a sensitive program with an insensitive program onto the same dual-core system. Experimental results show that this balanced job co-schedule scheme can improve the overall performance by 7%.

Modern computing has exhibited the trends towards highly parallel, heterogeneous processors and increasingly complicated software running on a multi-layered execution stack. Along with the trends, effective co-run performance prediction and resource management become more critical than ever for the maximization of computing efficiency. This dissertation has described our multi-dimensional efforts to tackle the challenges on multi-socket, multi-core systems. It lays the foundation for locality analysis on systems with non-uniform relations among cores, and offers a set of algorithms and techniques for analyzing and predicting the interactions among co-running threads or processes, hence preparing for an array of resource management in current and future computing systems.

Bibliography

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

[2] The linux kernel archives. `http://www.kernel.org`.

[3] E. Berg, Hakan Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–99, 2006.

[4] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

[5] K. Beyls and E.H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, August 2001.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[7] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

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

[9] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-time based performance prediction. In *Proc. of the 12th Intl. Workshop on Languages and Compilers for Parallel Computing*, 1999.

[10] 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)*, pages 340-351, 2005

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

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

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

[14] 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.

[15] C. DING, X. SHEN, K. KELSEY, C. TICE, R. HUANG, AND C. ZHANG. Software behavior-oriented parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, San Diego, USA, 2007.

[16] C. DING AND Y. ZHONG. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.

[17] X. DING, J. LIN, Q. LU, P. SADAYAPPAN, AND Z. ZHANG. Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 367–378, 2008.

[18] Z. DU, C. LIM, X. LI, C. YANG, Q. ZHAO, AND T. NGAI. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2004.

[19] 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.

[20] 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 the International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[21] C. FANG, S. CARR, S. ONDER, AND Z. WANG. Feedback-directed memory disambiguation through store distance analysis. In *Proceedings of the 20th ACM International Conference on Supercomputing*, 2006.

[22] A. FEDOROVA, M. SELTZER, C. SMALL, AND D. NUSSBAUM. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[23] 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*, pages 25–38, 2007.

[24] H.N. GABOW AND R. E. TARJAN. Faster scaling algorithms for general graph-matching problems. *Journal of ACM*, 38:815–853, 1991.

[25] M.R. GAREY AND D.S. JOHNSON. *Computers and Intractability.* Feeman, San Francisco, CA, 1979.

[26] T. HASTIE, R. TIBSHIRANI, AND J. FRIEDMAN. *The elements of statistical learning* Springer, 2001.

[27] D. S. HOCHBAUM. *Approximation Algorithms for NP-Hard Problems* PWS Publishing Company, 1995.

[28] 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*, pages 13–22, 2006.

[29] 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.

[30] Y. JIANG AND X. SHEN. Exploration of the influence of program inputs on cmp co-scheduling. In *European Conference on Parallel Computing (Euro-Par)*, August 2008.

[31] 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)*, pages 220–229, October 2008.

[32] Y. JIANG, K. TIAN, AND X. SHEN. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of The International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*. pages 201–215, 2010.

[33] Y JIANG, E. ZHANG, K. TIAN, AND X. SHEN. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of the International Conference on Compiler Construction*, 2010.

[34] YUNLIAN JIANG AND XIPENG SHEN. Exploration of the influence of program inputs on cmp co-scheduling. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 263–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] T. A JONHSON, R EIGENMANN, AND T. N VIJAYKUMAR. Speculative thread decomposition through empirical optimization. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, March 2007.

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

[37] 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.

[38] JOSEPH Y-T. LEUNG. *Handbook of Scheduling.* Chapman & HallCRC, 2004.

[39] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 65–74, 2009.

[40] C-K Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, Illinois, June 2005.

[41] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.

[42] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machine. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 92–101, 2009.

[43] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.

[44] M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, September 2005.

[45] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78-117, 1970.

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

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

[48] 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.

[49] 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*, pages 423–432, 2006.

[50] 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*, pages 2–12, 2006.

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

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

[53] 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.

[54] X. SHEN AND J. SHAW. Scalable implementation of efficient locality approximation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[55] 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)*, pages 55–62, 2007.

[56] X. SHEN, Y. ZHONG, AND C. DING. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.

[57] T. SHERWOOD, E. PERELMAN, G. HAMERLY, AND B. CALDER. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[58] A. J. SMITH. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 286–292, 1976.

[59] A. SNAVELY AND D.M. TULLSEN. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 66–76. 2000.

[60] 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*, pages 66–76, 2002.

[61] Y. SOLIHIN, V. LAM, AND J. TORRELLAS. Scal-tool: Pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing*, 1999.

[62] SPLASH. Stanford parallel applications for shared memory (SPLASH) benchmark. http://www-flash.stanford.edu/SPLASH/.

[63] H.S. STONE, J. TUREK, AND J.L. WOLF. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), September 1992.

[64] G. SUH, L. RUDOLPH, AND S. DEVADAS. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28:7–26, 2004.

[65] 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.

[66] 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*, pages 117–128, 2002.

[67] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.

[68] D. Thiebaut and H.S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), 1987.

[69] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

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

[71] 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*, pages 26–35, 2003.

[72] T.N. Vijaykumar and G.S Sohi. Task selection for a multiscalar processor. In *Proceedings of the International Symposium on Microarchitecture*, December 1998.

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

[74] 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.

[75] Y. Zhang. Solving large-scale linear programs by interior-point methods under the matlab environment. Technical Report 96-01, University of Maryland, July 1995.

[76] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3):328–343, March 2007.

[77] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266, June 2004.

[78] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, pages 129–142, 2010.

VITA

Yunlian Jiang

Yunlian Jiang received his Bachelor of Engineering and Master of Engineering degrees, both in Computer Science, from the University of Science and Technology of China in 2003 and 2006 respectively. He has been a PhD student in the Department of Computer Science at the College of William and Mary since 2006. He has become a PhD candidate since 2008. His research interests lie in compiler technology, program language analysis, shared cache and memory management, program locality analysis, input-centric computing and dynamic program optimization.