Information Sciences 526 (2020) 68–85

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Novel fairness-aware co-scheduling for shared cache contention game on chip multiprocessors[☆]

Zheng Xiao[a,*], Liwen Chen[a], Bangyong Wang[a], Jiayi Du[b,*], Keqin Li[a,c]

[a] Information Science and Engineering, Hunan University, and National Supercomputing Center in Changsha, Hunan 410082, China
[b] Computer and Information Engineering, Central South University of Forestry and Technology, Changsha, Hunan 410004,China
[c] Department of Computer Science, State University of New York,New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 6 November 2019
Revised 13 March 2020
Accepted 24 March 2020
Available online 2 April 2020

Keywords:
Chip multiprocessor
Co-scheduling
Cache contention
non-cooperative games
Nash equilibrium

ABSTRACT

Threads running on different cores of chip multiprocessors (CMP) can cause thread performance degradation due to contention for shared resources such as shared L2 cache. Some studies have shown that thread co-scheduling can effectively reduce contention for shared resources. However, in a multi-core system with shared caches, mutual interference between threads is unpredictable. As the number of cores increases, we are unlikely to exhaust all possible co-scheduling schemes. In this paper, a novel fairness-aware thread co-scheduling algorithm base on non-cooperative game is proposed to reduce L2 cache misses. We tried to improve the overall performance of the system by scheduling threads fairly. The originality of this work is to model thread scheduling using a non-cooperative game. The execution time of a thread varies depending on which threads are running on other cores of the same chip, because different thread combinations result in different levels of cache contention. Given the interdependence and competition between threads on the CMP architecture, non-cooperative game is used to solve the problem of thread co-scheduling where each thread is considered as a participant in the game. An iterative algorithm (***IA***) is proposed to solve the Nash equilibrium of the non-cooperative game in this paper. Subsequently, it is theoretically proved that ***IA*** has a potential game process and finally proves that ***IA*** can converge to Nash equilibrium in *N* iterations, where N is the number of threads. The co-scheduling scheme of all threads is obtained by solving the Nash equilibrium of the ***IA***. Finally, the convergence and effectiveness of ***IA*** proposed in this paper is verified by experiments. In addition, we use the cache partition to improve the performance of ***IA***. Experimental results show that the number of total cache misses of ***IA*** is less than that of the default scheduling algorithm, ***IA*** combined with cache partitioning can further reduce the total cache misses.

1. Introduction

Multiprocessor architectures provide a solution to achieve higher throughput and increased performance without breaking the physical limitations of Moore's Law, including chip multiprocessors that contain multiple cores on a single chip. The

[☆] This paper belongs to the special issue VSI:NPDTBA edited by Prof. W. Pedrycz.
[*] Corresponding authors.
E-mail addresses: zxiao@hnu.edu.cn (Z. Xiao), lwchen@hnu.edu.cn (L. Chen), bonnie_wang@hnu.edu.cn (B. Wang), dujiayi@csuft.edu.cn (J. Du), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.ins.2020.03.078
0020-0255/© 2020 Elsevier Inc. All rights reserved.

current CMP architectures primarily use shared LLC (last level cache) structures. Using a shared cache can reduce communication latency between multiple cores, but causes cache contention between threads running on different cores simultaneously, Additionally. Many studies have shown that the contention on shared LLC may be detrimental to co-scheduled threads [1]. It can cause problems such as thread starvation, cache pollution, and ultimately system performance degradation [2–4]. With the rapid increase in the number of processors in CMP, it has become increasingly important to ease shared cache contention between threads. Previous research has mainly solved the problem of cache contention through either thread scheduling or cache partitioning [5–9].

In this paper, we pay special attention to thread scheduling. Thread scheduling minimizes the conflicts and interference between threads by cleverly selecting the appropriate set of threads to run on the processor [4]. An effective thread scheduling scheme should minimize the contention for shared caches to maximize utilization and system performance [10]. To solve the thread scheduling scheme one has to determine which threads should be mapped to the same subset and which should be scheduled separately, thereby minimizing the impact of resource contention. However, in a multi-core system with shared caches, the execution time of a thread is affected by threads running on other cores of the same chip. Different threads combined together will compete for cache resources to varying degrees, thus resulting in different levels of performance degradation [11].

When multiple threads are executed at the same time, the interaction between threads cannot be measured [12], making the performance of CMPs unpredictable and not guaranteeing the Quality of Service (QoS) requirements of threads [13–15]. It is possible, but not feasible, to obtain an optimal thread co-scheduling scheme through brute force testing [16]. The brute force test is to obtain the actual co-scheduling degradation of the threads combination by executing all threads of all possible combinations. When the number of cores increases, the search space is too large to test all feasible combinations [11]. Therefore, it is necessary to reduce the search space for thread co-scheduling.

To the best of our knowledge, the problem of using game theory to solve thread co-scheduling has not been studied as such. Confronted with the interdependence and competition between threads, we use non-cooperative games to solve the problem of thread co-scheduling and reduce the search space for thread co-scheduling. On the one hand, threads have exclusive contention for shared resources, so there will be conflicts of interest between the co-scheduled threads, and the performance of each thread will be different under different co-scheduling schemes. On the other hand, for each thread, it is desirable to optimize its performance through co-scheduling, regardless of the performance of other threads, so each thread's decision is competitive and selfish. In this paper, we assume that the participants in the game are each thread, and define a policy set and utility function for each of them (see Section 3 for details). Each participant only considers its own utility in the game process, oblivious to other participants' utility. Since threads are interdependent and mutually influential, each thread's decision depends on the decisions that other threads have made. Nash equilibrium is achieved when each thread cannot obtain a better utility by changing its strategy individually. That is, when the strategy of any thread is the optimal response of other thread strategies, the obtained policy vector is the equalization strategy, that is, the thread coordinated scheduling scheme.

In this paper, we assume that L2 cache is the last-level cache and propose a fairness-aware thread co-scheduling model based on non-cooperative game to reduce the contention in threads. The threads in the system are divided into multiple scheduling batches, we try to achieve the system degradation balance of each scheduling batch as much as possible to achieve the purpose of fair scheduling threads. We first establish a linear relationship between the L2 cache miss rate of the threads and the total system performance degradation, which is used to predict the degradation of system performance when the threads are scheduled in the same batch. Then we consider the thread co-scheduling on the CMP architecture in accordance with game theory, where each thread is considered as a participant in the game. We propose an ***IA*** which is proved to be a potential game. Each iteration of the ***IA*** can further optimize the thread co-scheduling strategy and converge to a Nash equalization for a linear number of rounds. The time complexity of our analysed ***IA*** is $\Theta(N\log N)$, where N is the number of threads in the system. We obtain the co-scheduling scheme of all threads by solving the Nash equilibrium of ***IA***. Finally, we verify the convergence and effectiveness of ***IA***, and experimental results show that ***IA*** can reduce the total number of cache misses compared with the default scheduling algorithm. In addition, when multiple threads in the same batch share the cache space, there will be mutual interference between the threads and the performance of the threads will be reduced. Hassidi et al. [17] theoretically proved that the performance of the system can be improved by combining dynamic cache partitioning and dynamic job allocation. We use strict cache partitioning to further improve the performance of ***IA***. Experimental results show that the number of cache misses is further reduced by combining cache partition.

The main contribution of this paper can be summarized as follows:

- A novel fairness-aware thread co-scheduling model is proposed and the non-cooperative game is used to model thread co-scheduling.
- A thread co-scheduling algorithm ***IA*** is proposed to solve the Nash equilibrium of non-cooperative game model.
- It is theoretically proved that ***IA*** has a potential game process and finally proves that ***IA*** can converge to Nash equilibrium in N iterations, where N is the number of threads.
- The ***IA*** proposed in this paper effectively reduces the search space of the thread co-scheduling, and its effectiveness is verified by experiments. In addition, we use the cache partition to further improve the performance of ***IA***, it can further reduce the number of misses in the shared cache compared to the circumstance with only ***IA***.

The rest of the paper is organized as follows: Section 2 reviews related researches while Section 3 introduces the fairness-aware thread scheduling model on CMP architecture. In Section 4, a thread co-scheduling algorithm based on non-cooperative game is proposed, and the convergence of its Nash equilibrium is proved. Our experimental setup and experimental results are shown in Section 5. Finally, we summarize the work of this paper in Section 6.

2. Related work

In previous works, cache partitioning or thread scheduling were often used to reduce contention for shared caches. In this section, we review related works from aspects of cache partitioning and thread co-scheduling.

The cache partition divides independent cache space for each core, avoiding the extra cache miss caused by cache pollution, so that all thread requests can be responded. The easiest way is to distribute the shared cache space evenly to each core before running threads, that is, static cache partition. But when the size of the working set of the running threads is imbalanced, static partition may encounter a worst-case [18]. Therefore, the shared cache needs to be dynamically partitioned. In the previous work, there was a lot of work on the research of cache partitioning algorithm [6,19,20]. Since this paper particularly focuses on thread scheduling, the cache partition is not described in detail in this section.

Some recent studies have attempted to solve cache contention problems with clever co-scheduling. Platform throughput can be increased by cleverly executing multiple applications concurrently [21]. Shantharam et al. [22] studied co- scheduling based on speed-up profiles, which divides tasks into multiple packages to minimize the total execution time of all packages, but packs can have only one or two tasks and they reported the load is completed faster and the system power consumption is saved accordingly. Akturk et al. [10] introduced an adaptive cache-hierarchy-aware scheduler that attempts to schedule threads in a way that minimizes inter-thread contention, but this approach is concerned with thread L1 cache access features. Jiang et al. [8] ruled the selection of the thread set with the smallest conflict to the NP-hard problem, and the optimal set is solved by using greedy idea. The algorithm finds the optimal co-scheduling in the polynomial time, but the performance degradation of thread co-scheduling for each possible subset of threads is obtained by executing all possible subsets of threads. Aupy et al. [23] proposed an optimal processor assignment procedure when the tasks that form a pack are given, but this has a potentially exponential cost. Xie et al. [24] proposed a method of animal classification, which classifies lines according to the thread's memory access characteristics. However, the author did not give an accurate calculation method to classify threads. Hankendi and Coskun (2012) point out that the measurable benefits can be achieved by applying multi-level collaborative scheduling techniques at runtime (based on classifying tasks based on specific performance metrics) [25]. Hsu et al. [3] optimized system performance by balancing performance degradation across all threads. Zhuravlev et al. [9,26] proposed that the L2 cache miss rate of threads is the main factor leading to the performance degradation of the system, and separating threads with high miss rate can improve the overall performance of the system. However, the number of elements of each thread subset is equal, the miss rate of threads may not be evenly distributed to each subset. The idea of assigning a thread with a high miss rate in the cache is also proposed in [27], where the authors suggested reducing cache interference by spreading intensive threads and co-scheduling them with non-intensive threads. However, no studies have yet been conducted to quantify the relationship between miss rate and system degradation [11]. Merkel et al. [28] averted the competition for resources by jointly scheduling tasks with different characteristics, but they assumed that the task did little I/O and used computationally intensive tasks. Knauerhase et al. [29] introduced the OBS-X scheduling strategy, which observes the cache usage of each task and distributes cached-heavy threads throughout the system to help spread the cache load.

To the best of our knowledge, there is little research on using game theory to solve thread co-scheduling. In this paper, the non-cooperative game is used to model the thread scheduling, and the co-scheduling scheme of the thread is obtained by solving the Nash equilibrium.

3. Fairness-aware thread co-scheduling model on CMP architecture

In this paper, we attempt to reduce shared cache contention on CMP through fairness-aware thread co-scheduling. Before introducing the thread co-scheduling model, we first introduce the CMP architecture. As show in Fig. 1, we target multi-core chip with multi-layered memory structures. This architecture consists of multiple cognate cores, expressed as: $\boldsymbol{Core} = \{Core_1, Core_2, \ldots, Core_p\}$, where p represents the number of cores. Each core has a private L1 cache, and all cores share L2 Cache and main memory through a shared bus.

In order to obtain a thread fair co-scheduling schema, we need to obtain the performance degradation when co-scheduling threads, however, this is usually not easy to measure. The interaction between threads cannot be measured [12] when their are executed at the same time, making the performance of CMPs unpredictable and not guaranteeing the Quality of Service (QoS) requirements of threads [13–15]. A feasible prediction scheme is used to predict the L2 miss rate of the thread first, and then the linear regression model is used to convert the predicted L2 cache miss rate and currently measured L1 miss rate to the predicted CPI of the thread, but this method is rather complex [30]. Daci and Tartari [31] compared the L2 cache miss rate metric with other characterization metrics, and concluded that using the L2 cache miss rate to predict the performance degradation of thread co-scheduling is better than other metrics. However, they did not quantify the relationship between miss rate and performance degradation. Next, we will use the L2 cache miss rate of co-scheduling threads to predict the performance degradation of the system.

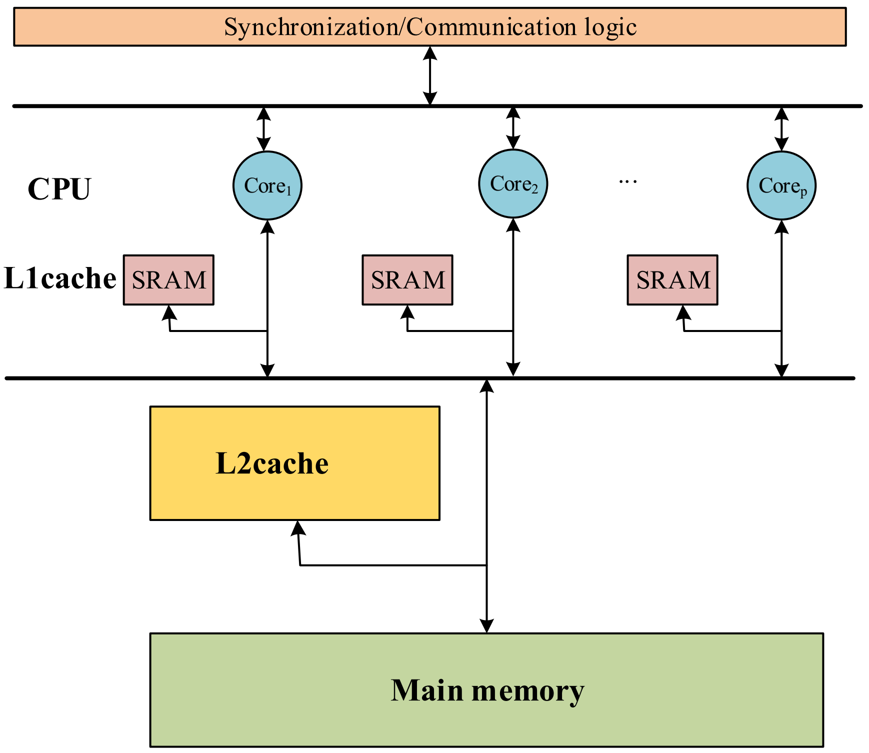


Fig. 1. System architecture.

3.1. Performance degradation prediction

In this section, we will predict the system performance degradation when threads are co-scheduled. We first give a quantitative formula for the performance degradation of the system. Thread co-scheduling on a CMP may be slower than running alone, and this degradation is called co-scheduling degradation, quantified in the following formula [8]:

$$D_i = \frac{cCPI_i - sCPI_i}{sCPI_i} \tag{1}$$

where D_i represents the degradation of thread i, $cCPI_i$ represents the number of cycles per instruction when thread i shares L2cache with other threads (i.e., threads co-running on the same L2 cache), and $sCPI_i$ represents the number of cycles per instruction when thread i is running alone. For threads of the same scheduling batch, the total performance degradation of the system is expressed as Eq. (2).

$$D = \sum_{i=1}^{n} D_i \tag{2}$$

Where D represents the total degradation of system, and n represents the number of threads that are co-scheduled.

In this paper, we try to illuminate the relationship between the L2 cache miss rate of threads and the performance degradation of the system. Our experimental data is derived from the experimental results of Chandra et al. [32]. There are 14 sets of experimental results. For each set of experimental results, we calculate the total performance degradation of each group threads according to Eq. (2), and the sum of their L2 Cache missing rates.

The calculated results are imported into MATLAB, and the experimental data is fitted and modeled using a curve fitting toolbox. A functional relationship was found that reflects the closeness between the total performance degradation of the system and the sum of thread miss rates. By comparing the performance index values of the fitted curves, the final functional relationship is as shown in Eq. (3):

$$\begin{aligned} &Pain = b \times \sum_{i=1}^{n} miss_i + c \\ &b = 0.3342 \quad \text{confidence interval :}(0.2847, 0.3834) \\ &c = 0.1296 \quad \text{confidence interval : } (0.06208, 0.187) \end{aligned} \tag{3}$$

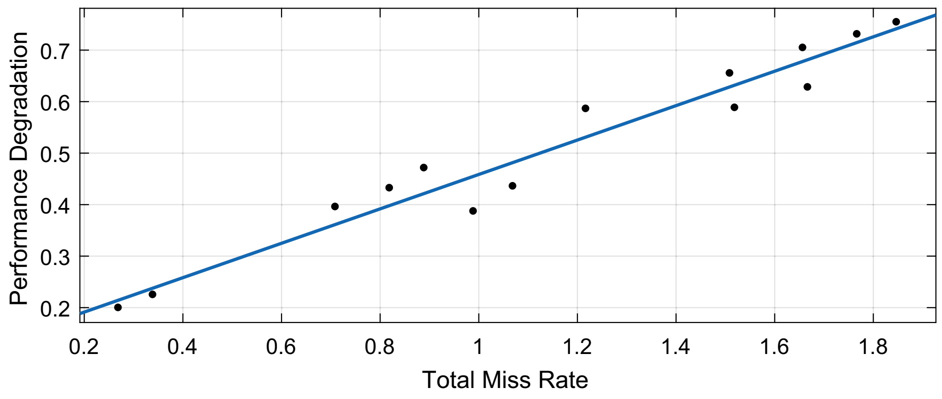


Fig. 2. Fit curve of performance degradation and the total miss rate.

	Thd_1	Thd_2	Thd_3	Thd_4	Thd_5	Thd_6
batch 1	0	0	0	1	1	0
batch 2	1	0	1	0	0	0
batch 3	0	1	0	0	0	1

Fig. 3. A co-scheduling scheme.

where *Pain* represents the total performance degradation of the system, n represents the number of co-scheduling threads and $miss_i$ represents the miss rate of thread i.

The confidence interval for each coefficient is 95%, R-square = 0.9478, and RMSE = 0.0424. Individual coefficient values can be adjusted according to the actual situation. The experimentally fitted curve is shown in Fig. 2.

3.2. Thread co-scheduling motivation

There are a number of N threads waiting to be scheduled in the ready queue. These N threads are represented as: $\boldsymbol{Thread} = \{Thd_1, Thd_2, \ldots, Thd_N\}$. In this paper, we use the CMP architecture, and all processor cores share the L2 cache. These N threads are divided $T(T > 0)$ scheduled batches, with the set of threads for the tth scheduled batch expressed as $\boldsymbol{batch_t}$. Each thread must select one of the T scheduled batches to meet: $\boldsymbol{batch_1} \cup \boldsymbol{batch_2} \cup \cdots \cup \boldsymbol{batch_T} = \boldsymbol{Thread}$ $(\boldsymbol{batch_i} \subseteq \boldsymbol{Thread}, 1 \le i \le T)$; $\boldsymbol{batch_t} \cap \boldsymbol{batch_{t'}} = \emptyset (1 \le t, t' \le T and\, t \ne t')$. Threads of the same scheduled batch share L2cache, and no resources are shared between threads of different scheduled batches. Once a thread is executed, it will not be interrupted by other threads until the thread finishes running. In other words, it is the turn of the next batch of threads to wait until all the threads in the previous batch have finished executing.

Assuming there are 6 threads in a ready queue: $\boldsymbol{Thread} = \{Thd_1, \ldots, Thd_6\}$ and the total scheduled batch is 3. The scheduled batch selected by each thread is represented as: (2,3,2,1, 1, 3), and we can get the co-scheduling scheme shown in Fig. 3. The scheduling scheme for all threads can be represented as a matrix of $T \times N$. Each row i represents a scheduled batch, and each column j represents a thread j. Each entry ij in the matrix indicates whether the thread j is scheduled in the batch i (the entry is equal to "1" if the thread j is scheduled in batch i). We can obtain: $\boldsymbol{batch_1} = \{Thd_4, Thd_5\}$; $\boldsymbol{batch_2} = \{Thd_1, Thd_3\}$; $\boldsymbol{batch_3} = \{Thd_2, Thd_6\}$.

In [33] we can know that for each thread Thd_i, its actual processing time will be affected by other threads that are co-scheduled. Placing threads with small competing relationships in the same scheduled batch can improve thread performance. In this paper, we try to reduce the competition between the co-scheduled threads by using appropriate thread scheduling scheme.

In this paper, the performance degradation of the jth scheduled batch of threads can be expressed as:

$$Pain_j = \begin{cases} b \times (\sum_{Thd_i \in \boldsymbol{batch_j}} R_i) + c & |\boldsymbol{batch_j}| > 1 \\ 0 & |\boldsymbol{batch_j}| \le 1 \end{cases} \tag{4}$$

where $Pain_j$ represents the total performance degradation of the jth scheduled batch, and $\boldsymbol{batch_j}$ represents the set of threads co-scheduled in the jth scheduling batch. R_i represents the L2 cache miss rate when the thread Thd_i is running alone,$|\boldsymbol{batch_j}|$represents the number of threads in $\boldsymbol{batch_j}$, b and c represents the relationship coefficient between the total miss rate and the degradation of system performance (Section 3.1). In this paper, the performance degradation of the thread is 0 when it is run alone.

As mentioned in Blagodurov et al. [9], the miss rate of threads is the main factor leading to the performance degradation of the system. Separating the threads with high miss rate into different caches can improve the overall performance of the system. In this paper, we also use this heuristic to evenly degrade system performance to each scheduled batch. Another benefit of doing so is that the threads are fairly scheduled and can respond to all thread scheduling requests. Because it does not cause large performance degradation of a batch, the response of some threads will not be delayed.

Our goal is to make the performance degradation of each scheduled batch as equal as possible by solving a thread scheduling scheme, where the performance degradation can be calculated by Eq. (4). In this paper, we use $Pain_{\max}$ to represent the maximum value of performance degradation in these T batches. Our priority is to minimize $Pain_{\max}$.

In this paper, we have the following assumptions:

(1) Thread co-scheduler is space-shared machines. Space sharing refers to putting threads in the same scheduled batch execution so that they share the L2cache space. Since we are concerned with the overall system performance, the issue of which batch of threads is executed first has no substantial effect on the final result.

(2) In order for co-scheduling to be effective, the underlying CMP architecture must contain multiple cores.

(3) All L2 cache sharing is destructive interference. In other words, the performance of the two threads is not higher when the L2 cache is shared as compared to when it is not. The degree of destructive interference can vary greatly from negligible to significant, but it will never be negative ($Pain \geq 0$).

4. Non-cooperative game between threads

4.1. Game optimization goal

In this paper, we use the non-cooperative game to solve the problem of thread co-scheduling, and obtain the co-scheduling scheme by solving the Nash equilibrium. The game problem focuses on the game relationship between multiple participants, the purpose of which is to maximize the benefits of the participants themselves or to minimize negative influence on them. In this paper, we consider the problem of thread co-scheduling under the CMP architecture as a non-cooperative game between multiple participants. The non-cooperative game problem mainly includes participant collection, strategy collection and strategy of the participant selection. In this paper, the participant collection means that the N threads are in the ready state. The participants' strategy set is the T different scheduling batches. That is $\boldsymbol{Q_i} = \{1, 2, \ldots, T\}$, and the set of strategies for all participants is: $\boldsymbol{Q} = \boldsymbol{Q_1} \times \boldsymbol{Q_2} \times \cdots \times \boldsymbol{Q_N}$.

The strategy chosen by participant i is expressed as $q_i \in \boldsymbol{Q_i}$, and it denotes that the user i chooses to run on the q_ith scheduling batch. Therefore, for all threads waiting to be scheduled, we can get the scheduling strategy vector: $\boldsymbol{q} = \{q_1, \ldots, q_N\}$ $(\boldsymbol{q} \in \boldsymbol{Q})$, representing a policy set of N threads. We use $Pain_t(\boldsymbol{q})$ $(t \in \{1, \ldots, T\})$ to represent the degree of system performance degradation of the tth scheduling batch, when the strategy vector for all participants is $\boldsymbol{q}$. $Pain_{\max}(\boldsymbol{q})$ represents the biggest performance degradation of the system when the strategy vector for all participants is $\boldsymbol{q}$. $Pain_{\min}(\boldsymbol{q})$ represents the minimal performance degradation of the system when the strategy vector for all participants is $\boldsymbol{q}$. In this paper, we try to make the performance degradation of each scheduling batch as even as possible, that is, minimize the $Pain_{\max}(\boldsymbol{q})$. The optimization problem solved can be expressed as follows:

$$\text{minimize} \max_{t \in \{1, \ldots, T\}} Pain_t(\boldsymbol{q}), \boldsymbol{q} \in \boldsymbol{Q} \tag{5}$$

In the non-cooperative game problem, each participant i is selfish, and its tries to maximize its own benefit or minimize negative benefits, without considering the benefits of other participants. We define the utility function of participant $Thd_i\,(Thd_i \in \boldsymbol{Thread})$ as $u_i(q_i, \boldsymbol{q}_{-i}) = 1/(1 + Pain_{q_i})$ $(i \in \{1, \ldots, N\})$, and the utility is 1 when the thread is running alone. Each thread tries to jump to the scheduling batch t $(t \in \{1, \ldots, T\})$ with the least performance degradation, maximizing its own utility. Therefore, the optimization problem of Thd_i is equivalent to:

$$\text{maximize} \quad u_i(q_i, \boldsymbol{q}_{-i}) = 1/(1 + Pain_{q_i}) \tag{6}$$

where q_i is the strategy selected by Thd_i and $\boldsymbol{q}_{-i} = \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_N\}$ is a vector of $1 \times (N-1)$, representing the strategy of the remaining threads except Thd_i. If and only if $u_i({q_i}^*, \boldsymbol{q}_{-i}) > u_i({q_i}', \boldsymbol{q}_{-i})$, thread i tends to strategy ${q_i}^*$ more than strategy ${q_i}'$, where $({q_i}^*, \boldsymbol{q}_{-i})$ represents the scheduling strategy vector obtained after the Thd_i replaces the strategy ${q_i}'$ with ${q_i}^*$.

Definition 1 (Nash equilibrium)**.** For the non-cooperative game problem formed by thread co-scheduling, the Nash equilibrium solution is to find a scheduling strategy vector $\boldsymbol{q}^*$, so that it is satisfied for each participant Thd_i $(Thd_i \in \textbf{Thread})$:

$${q_i}^* \in \arg\max_{q_i \in \boldsymbol{Q_i}} u_i(q_i, \boldsymbol{q}^*), \boldsymbol{q}^* \in \boldsymbol{Q} \tag{7}$$

When the system reaches the Nash equilibrium, no one can make a higher utility by changing its own strategy, while the strategies of other participants are fixed.

An equilibrium scheduling strategy vector can be obtained when the strategy of any participant is the optimal response to the strategies of other participants. Given complete knowledge of the system, the strategy of participant Thd_i is a solution to the following optimization problem.

$$\text{minimize} \quad Pain_{q_i}(q_i, \boldsymbol{q}_{-i}), q_i \in \boldsymbol{Q_i} \tag{8}$$

In order to solve this optimization problem, the strategy of all other participants is fixed. Only the strategy q_i of the participant i is a variable, when it is the turn of the participant i to make a decision.

*4.2. Iterative Algorithms (**IA**)*

In this paper, we treat each thread as a participant in a non-cooperative game. In the process of the game, all participants make decisions in a certain order. In each round of decision making, only one participant will try to increase its own utility by changing its own strategy, while the strategies of other participants are fixed.

We assume that $\boldsymbol{q}^{k-1}$ represents the policy vector of all threads after the $(k-1)$th iteration, and it is the turn of participant i to make the decision at the kth iteration. Participant i will update its current strategy q_i to maximize its utility. In this paper, we propose an iterative algorithm based on greedy idea and it's specific implementation details of the ***IA*** are given in Algorithm 1. By solving a scheduling strategy vector of all threads, any participant cannot make its own utility greater

Algorithm 1 Iterative Algorithm (***IA***).

```
Input: thread collection $\boldsymbol{Thread} = \{Thd_1, \ldots, Thd_N\}$, L2cache miss rate set for N threads $\boldsymbol{MissRate} = \{R_1, \ldots, R_N\}$, the total
    scheduling batches $T$, the coefficients $b$ and $c$
Output: The scheduling policy vector $\boldsymbol{q}$ for all threads
 1: sort $\boldsymbol{MissRate}$ by non-ascending order, so that $R_1 \ge R_2 \ge \cdots \ge R_N$. Each thread $Thd_i$ is randomly assigned to a scheduling
    batch and set the initial iteration counter $k = 0$
 2: record the initial scheduling strategy $\boldsymbol{q}^0$ and calculate the performance degradation for all batches: $Pain_t(\boldsymbol{q}^0)(t \in$
    $\{1, \ldots, T\}) = b \times (\sum_{Thd_i \in \boldsymbol{batch_t}} R_i) + c \quad (|\boldsymbol{batch_t}| \ge 0)$.
 3: while $(\boldsymbol{q}^k \ne \boldsymbol{q}^{k-N}$ or $k \le N)$ do
 4:     set $k = k + 1$ and $i = k\%N$, the policy of thread $i$ is denoted as $q_i$
 5:     the strategy of thread $i$ after $k-1$ iterations is assigned to the variable $t$: $t = q_i$
 6:     record the batch with the smallest performance degradation after $k-1$ iterations as $t'$
 7:     if $Pain_t(\boldsymbol{q}^{k-1}) > Pain_{t'}(\boldsymbol{q}^{k-1}) + b \times R_i$ then
 8:         set $Pain_t(\boldsymbol{q}^k) = Pain_t(\boldsymbol{q}^{k-1}) - b \times R_i$
 9:         update the strategy $q_i$ of $Thd_i$, set $q_i = t'$
10:         set $Pain_{t'}(\boldsymbol{q}^k) = Pain_{t'}(\boldsymbol{q}^{k-1}) + b \times R_i$
11:         for $1 \le h \le T$ and $h \ne t, t'$ do
12:             set $Pain_h(\boldsymbol{q}^k) = Pain_h(\boldsymbol{q}^{k-1})$
13:         end for
14:     end if
15: end while
16: for $1 \le t \le T$ do
17:     if $|\boldsymbol{batch_t}| \le 1$ then
18:         set $Pain_t = 0$
19:     end if
20: end for
```

by changing its own strategy, when other participants' strategies are fixed. During the iterative process in the algorithm, We will ignore the number of threads in each scheduled batch during the algorithm iteration, and adjust the performance degradation of each batch and the benefits of each thread after the iteration. Before the algorithm iterates, we let the thread with the largest miss rate make the decision first.

In the Algorithm 1, we sort the miss rate of all threads in non-ascending order and randomly assign a scheduled batch to each thread (Line 1). At the kth iteration, it is the turn of Thd_i to make a decision. We need to record the strategy of thread i after the $k-1$th iteration and the scheduling batch with the least performance degradation(Line 5–6). If the condition $Pain_{q_i}(\boldsymbol{q}^{k-1}) > Pain_{\min}(\boldsymbol{q}^{k-1}) + b \times R_i$ is satisfied, the strategy of Thd_i will be updated. The scheduling policy vector $\boldsymbol{q}$ and the vector ***Pain*** of the performance degradation of all scheduled batches are updated (Line 7–14). Since we ignore the number of threads in each scheduling batch during the algorithm iteration process, after the end of the iteration process, we need to adjust the performance degradation of each scheduling batch according to Eq. (4) (Line 16–20).

4.3. Nash equilibrium analysis

This section will illustrate the convergence of Algorithm 1. Before that, we first introduce the concept of a state graph and a potential function. As described in Schulz et al. [34], a state graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a directed graph in which the node set $\boldsymbol{V} = \boldsymbol{Q}(\boldsymbol{Q} = \boldsymbol{Q_1} \times \boldsymbol{Q_2} \times \cdots \times \boldsymbol{Q_N})$. If the state $\boldsymbol{q}$ to the state $\boldsymbol{q}'$ has an edge, the difference between the state $\boldsymbol{q}$ and the state $\boldsymbol{q}'$ merely lies in that the strategy of one participant i is different, and the utility of the participant i in the state $\boldsymbol{q}$ is strictly less than $\boldsymbol{q}'$. A Nash equilibrium solution corresponds to a node with no outgoing edges. The existence of the potential function means that the state diagram is aperiodic, which determines the existence of the Nash equilibrium. A game with a potential function is called a potential game.

Theorem 1. *There exists a potential game process in* ***IA***

Proof. For any state $\boldsymbol{q}$, we use $\boldsymbol{Pain(q)} = \{Pain_1(\boldsymbol{q}), \ldots, Pain_T(\boldsymbol{q})\}$ to represent the vector of the performance degradation of all scheduling batches at the state $\boldsymbol{q}$ and sort them in non-ascending order. We need to prove that at each iteration, when a thread increases its own utility by changing its own scheduling batch and moving from one batch to another, the vector $\boldsymbol{Pain}$ is logically reduced along with the thread changing its scheduling strategy. It is assumed that the scheduling strategy vector for all threads is $\boldsymbol{q}'$ after one thread changes its own strategy. Proof that: $Pain_{\max}\boldsymbol{(q')} \le Pain_{\max}\boldsymbol{(q)}$.

Assume that the system's scheduling strategy vector for all threads at the beginning of the kth iteration is $\boldsymbol{q} = (q_1, \ldots, q_i, \ldots, q_N)$, the system degradation vector for all scheduled batches is $\boldsymbol{Pain(q)} = (Pain_1(\boldsymbol{q}), \ldots, Pain_T(\boldsymbol{q}))$, and the decision made in the kth iteration is the Thd_i. Suppose the utility of Thd_i before the kth iteration is $u_i(q_i, \boldsymbol{q}_{-i}) = 1/(1 + Pain_t(q_i = t))$, and Thd_i jumps from the tth scheduling batch to the t'th scheduling batch under the idea of ***IA*** to increase its utility. The scheduling strategy vector of all threads after the kth iteration is $\boldsymbol{q}' = (q_1, \ldots, q_i', \ldots, q_N)(q_i' = t')$ and $\boldsymbol{Pain(q')} = (Pain_1(\boldsymbol{q}'), \ldots, Pain_T(\boldsymbol{q}'))$. The utility of Thd_i after the kth iteration is $u_i(q_i', \boldsymbol{q}_{-i}) = 1/(1 + Pain_{t'}(q_i' = t'))$. According to the Line 7 of ***IA***, we can know that $Pain_{t'}(\boldsymbol{q}') < Pain_t(\boldsymbol{q})$, although the strategy change of Thd_i caused an increase in the performance degradation of the t'th batch, the performance degradation of the t'th batch after the kth iteration was still lower than that of the tth batch before the kth iteration. And the change in the strategy of Thd_i does not change the performance degradation of the remaining batches except t and t', that is: $Pain_h(\boldsymbol{q}) = Pain_h(\boldsymbol{q}')\ \ (1 \le h \le T, h \ne t, t')$. Denote $\boldsymbol{Pain_h(q')} = \{Pain_h(\boldsymbol{q}')\ \ (1 \le h \le T, h \ne t, t')\}$, if $Pain_t(\boldsymbol{q}) = Pain_{\max}(\boldsymbol{q})$ is established, $Pain_{\max}(\boldsymbol{q}') = \max\{Pain_{t'}(\boldsymbol{q}'), Pain_t(\boldsymbol{q}'), \boldsymbol{Pain_h(q')}\}$ and $Pain_{t'}(\boldsymbol{q}') < Pain_t(\boldsymbol{q})$; $Pain_t(\boldsymbol{q}') < Pain_t(\boldsymbol{q})$ and $\max(\boldsymbol{Pain_h(q')}) \le Pain_t(\boldsymbol{q})$. Hence, $Pain_{\max}\boldsymbol{(q')} \le Pain_{\max}\boldsymbol{(q)}$. If the case $Pain_t(\boldsymbol{q}) \ne Pain_{\max}\boldsymbol{(q)}$ is established, we can deduce that: $\max(\boldsymbol{Pain_h(q')}) > Pain_t(\boldsymbol{q})$, $Pain_{\max}(\boldsymbol{q}') = \max(\boldsymbol{Pain_h(q)})$. Thus, $Pain_{\max}\boldsymbol{(q')} \le Pain_{\max}\boldsymbol{(q)}$ is established. □

Corollary 1. *The non-cooperative game problem of thread co-scheduling indicates that all participants' strategies will converge to a Nash equilibrium under the idea of* ***IA****.*

Theorem 1 proves that the participant's strategy converges to Nash equilibrium under the idea of ***IA*** does not necessarily mean that it can converge quickly to the Nash equilibrium. The convergence speed of the ***IA*** is also a key issue, and next we will prove the convergence of the Nash equilibrium of the ***IA***. Before that, we first prove the two important properties of Theorems 2 and 3.

Theorem 2. *Assume that the* ***IA*** *is in state* $\boldsymbol{q}$ *and the next state is* $\boldsymbol{q}'$*, then there is* $Pain_{\min}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}')$*.*

Proof. Assuming that the current state is $\boldsymbol{q}$, it is the turn of participation i to make a decision, and the current strategy of Thd_i is q_i. The participation i changes its own strategy to q_i' at state $\boldsymbol{q}'$, and we have $Pain_{q_i'}(\boldsymbol{q}) = Pain_{\min}(\boldsymbol{q})$. At state $\boldsymbol{q}'$, $Pain_{\min}(\boldsymbol{q}') = \min\{Pain_{q_i}(\boldsymbol{q}) - b * R_i, Pain_{q_i'}(\boldsymbol{q}) + b \times R_i, \boldsymbol{Pain_h(q)}\}(1 \le h \le T, h \ne q_i, q_i')$. We can get that $Pain_{q_i}(\boldsymbol{q}) > Pain_{\min}(\boldsymbol{q}) + b \times R_i$, according to the idea of the ***IA***. Clearly, $Pain_{\min}(\boldsymbol{q}) \le Pain_h(\boldsymbol{q})(1 \le h \le T,\ h \ne q_i,\ q_i')$, Thus, we can get that: $Pain_{\min}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}')$. □

Theorem 3. *Assume that participant i has updated its strategy, and the scheduling policy vector of all threads is* $\boldsymbol{q}$*. Participants who make decisions before participant i will not update their own strategies. i.e. for all participants* $Thd_j(1 \le j \le i)$*, we have* $q_j \in \underset{q_j' \in \boldsymbol{Q_j}}{\arg\max}\, u_j(q_j', \boldsymbol{q}_{-j})$.

Proof. To prove that $q_j \in \arg\max_{q_j' \in \boldsymbol{Q_j}} u_j(q_j', \boldsymbol{q}_{-j})$ is established, we need to proof that $q_j \in \arg\max_{q_j' \in \boldsymbol{Q_j}} Pain_{q_j'}(q_j', \boldsymbol{q}_{-j})$, i.e., $Pain_{q_j}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}) + b \times R_j$. In this paper, we prove the theorem by recursive methods. When $i = 1$,we need to prove that $Pain_{q_1}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}) + b \times R_1$. $\boldsymbol{q}$ is the strategy vector after the Thd_1 updates its strategy. Obviously, the current scheduling policy vector can maximize the utility of Thd_1 when other participants' strategies are fixed. Thus, $Pain_{q_1}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}) + b \times R_1$ is established.

Assuming that the theorem is true for the participant $i(1 \le i < N)$, that is, for all participants $j(1 \le j \le i)$, $Pain_{q_j}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}') + b \times R_j$ is true. Assuming that the scheduling strategy vector of all threads is $\boldsymbol{q}'$ after the participant $i+1$ updates his strategy. We need to prove that $Pain_{q_j}(\boldsymbol{q}') \le Pain_{\min}(\boldsymbol{q}') + b \times R_j\ \ (1 \le j \le i)$. The proof is as follows:

In the state $\boldsymbol{q}$, if the condition $Pain_{q_{i+1}} \le Pain_{\min}(\boldsymbol{q}) + b \times R_{i+1}$ is true, participant $i+1$ will not change its own strategy, that is $\boldsymbol{q}' = \boldsymbol{q}$. Under this condition we can get $Pain_{q_j}(\boldsymbol{q}') \le Pain_{\min}(\boldsymbol{q}) + b \times R_j$ for all participants $j(1 \le j \le i)$. Otherwise, $Pain_{q_{i+1}} > Pain_{\min}(\boldsymbol{q}) + b \times R_{i+1}$, and participant Thd_{i+1} jumps from the q_{i+1}th scheduling batch to the q_{i+1}'th scheduling batch. The change in the strategy of participant Thd_{i+1} will result in a change in the performance degradation of the two batches q_{i+1} and q_{i+1}', while the performance degradation of all others batches is unchanged. We have $Pain_{q_j}(\boldsymbol{q}') = Pain_{q_j}(\boldsymbol{q})(1 \le j \le i+1, q_j \ne q_{i+1}, q_{i+1}')$. By Theorem 2, we know that $Pain_{\min}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}')$. Therefore $Pain_{q_j}(\boldsymbol{q}') \le Pain_{\min}(\boldsymbol{q}') + b \times R_j(1 \le j \le i+1, q_j \ne q_{i+1}, q_{i+1}')$. Since the performance degradation of the q_{i+1}th batch is reduced, we can get $Pain_{q_j}(\boldsymbol{q}) \le Pain_{\min}(\boldsymbol{q}') + b \times R_j (1 \le j \le i+1, q_j = q_{i+1})$. According to the iterative algorithm, we have $R_j \le R_{i+1} (1 \le j \le i+1)$ and $Pain_{q_{i+1}'}(\boldsymbol{q}') \le Pain_{\min}(\boldsymbol{q}') + b \times R_{i+1}$. Therefore, $Pain_{q_j}(\boldsymbol{q}') \le Pain_{\min}(\boldsymbol{q}') + b \times R_j (1 \le j \le i+1, q_j = q_{i+1}')$.

Therefore, $q_j \in \arg\max_{q_j' \in \boldsymbol{Q_j}} u_j(q_j', \boldsymbol{q}_{-j}) (1 \le j \le i, 1 \le i \le N)$ is proved. □

Theorem 4. *The **IA** can converge to the Nash equilibrium solution in N iterations, where N is the number of threads.*

Proof. We know that a thread with a higher miss rate starts the decision-making process earlier according to the ***IA*** proposed in this paper. In other words, the participant chooses the strategy to maximize its utility according to the sequence $R_1 \ge R_2 \ge \cdots \ge R_N$. By Theorem 3, we know that if the participant i has updated his strategy, the participant who updated the strategy before the participant i will keep its own strategy. Therefore, if participant N has updated its strategy, the strategies of all participants will not change, and each participant will update the strategy no more than once. Therefore, the ***IA*** for co-scheduling multi-threaded game problem can converge to the Nash equilibrium in the iteration of N. □

Theorem 5. *The Nash equilibrium that **IA** converges to is an optimal solution.*

Proof. Use $\boldsymbol{q}^*$ to represent an optimal scheduling strategy vector and assume $\boldsymbol{q}^0 = \boldsymbol{q}^*$. After ***IA*** is executed, we get a scheduling strategy vector $\boldsymbol{q}$ with: $Pain_{\max}(\boldsymbol{q}) \le Pain_{\max}(\boldsymbol{q}^*)$. Since $\boldsymbol{q}^*$ has been assumed to be an optimal scheduling strategy vector, $Pain_{\max}(\boldsymbol{q}^*)$ cannot be further reduced. It can be obtained that $Pain_{\max}(\boldsymbol{q}) = Pain_{\max}(\boldsymbol{q}^*)$, $\boldsymbol{q}$ is also an optimal scheduling strategy vector. Therefore, The Nash equilibrium obtained by ***IA*** convergence is an optimal solution. □

Theorem 6. *The time complexity of **IA** is $\Theta(N \times \log N)$.*

Proof. According to Line 1 of Algorithm 1, the time complexity for sorting the missing rates of all threads is $\Theta(N\log N)$, and then the time complexity of randomly assigning all threads to the dispatch batch is $\Theta(N)$. In Line 2, after calculating the performance degradation of all scheduling batches, the time complexity is $\Theta(T)$. As for Line 3 to Line 15, each iteration needs to select the minimum performance degradation, and the time complexity of selecting the minimum performance degradation is $\Theta(\log T)$. After the iteration is completed, we also need to adjust the performance degradation of each batch (Line 3 to Line 15 of Algorithm 1), and its time complexity is $\Theta(T)$. It takes at most N iterations (Theorem 4), so the time complexity of the ***IA*** is $\Theta(N \times \log N + N + T + N \log T + T) = \Theta(N \times \log N)$. □

5. Experiment and evaluation

In this section, we provide simulation experiments to verify the theoretical analysis in Section 4.3 and the performance of the ***IA*** proposed in this paper. The stack distance profiling is the theoretical basis of the experimental verification in this paper and we will introduce it.

5.1. Stack distance profiling

The stack distance profiling reflects the thread's reuse behavior on the cache space [35,36]. For a M-way group associative cache with least recently used (LRU) replacement algorithm, there are $M+1$ counters: $C_1, C_2, \ldots, C_M, C_{>M}$. On each access to the cache, the corresponding counter will be updated. If it is a cache access to a line in the ith position in the LRU stack of the set, C_i will be incremented. The first line in the stack is the most recently used row (MRU) of the set, and the last line in the stack is the least recently used (LRU) line. If it is a cache miss, the line will not be found in the LRU stack, causing $C_{>M}$ plus one. The information of the stack distance can easily be obtained by the compiler [35], either by simulation or by running the thread alone in the system without cache sharing [37].

We can easily calculate the number of cache misses and miss rate using the stack distance profile [5]. For example, the stack distance profile of the thread i running alone without cache sharing is as shown in Fig. 4, then the access frequency of thread i is expressed as: $Af_i = \sum_{j=1}^{M+1} \frac{C_j}{CPUcycle}$. If the cache space of the M' ways ($M' < M$) is allocated to the thread i, the number of misses of the thread i is as shown in $miss_i = C_{>M} + \sum_{j=M'+1}^{M} C_j$ and its miss rate is as shown in Eq. (9):

$$R_i = \frac{miss_i}{C_{>M} + \sum_{j=1}^{M} C_j} \tag{9}$$

5.2. Experiment setup

In the experimental environment setup, we consider a shared L2 Cache associated with an 8-way set (a total of 9 counters). The total number of threads is set as 15. We randomly generate stack distance profile for each thread, and the value of each counter C_j ($j \in \{1, \ldots, 9\}$) varies from 0 to 18 (unit: million). Without loss of generality, we set the *CPUcycle* of each thread to 1 million. We conducted the experiment with the total scheduling batches of 4, 6, and 8. We calculated the L2 cache miss rate for each thread running alone using Eq. (9), as shown in Table 1.

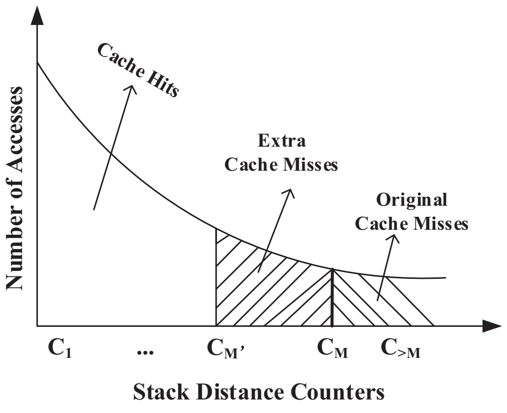


Fig. 4. Illustration of a stack distance profile[5].

Table 1
The miss rate of threads.

Thread	L2 Miss Rate	Thread	L2 Miss Rate
Thd_1	28%	Thd_9	20%
Thd_2	3%	Thd_{10}	18%
Thd_3	5.5%	Thd_{11}	4.9%
Thd_4	26%	Thd_{12}	8.2%
Thd_5	30%	Thd_{13}	19%
Thd_6	39%	Thd_{14}	6%
Thd_7	9%	Thd_{15}	25%
Thd_8	32%		

Table 2
Thread co-scheduling scheme based on ***IA***.

Scheduling batch	Threads
batch 1	Thd_8, Thd_{11}, Thd_{12}
batch 2	Thd_2, Thd_6
batch 3	Thd_5, Thd_{10}
batch 4	Thd_1, Thd_{13}
batch 5	Thd_4, Thd_9
batch 6	Thd_3, Thd_7, Thd_{14}, Thd_{15}

5.3. *The convergence of* **IA**

In our experiments, the initial scheduling strategy was randomly chosen from the set of strategies for all participants ***Q***,that is, each thread randomly selects a scheduling batch (line 1 of Algorithm 1).

Table 2 is a thread scheduling scheme obtained through ***IA***. According to Table 2, we can know that the sum of the miss rate of each scheduling batch is roughly equal(0.451 vs. 0.42 vs. 0.48 vs. 0.47 vs. 0.46 vs. 0.455, maximum difference of missing rate is only 0.06), with the factors that cause system performance degradation being even. This will improve the overall performance of the system, and this conclusion will be validated in experiments.

Fig. 5 shows that the value of the optimization problem (Eq. (5)) solved in this paper decreases as the number of iterations increases, eventually reaching a relatively stable state. This verifies the correctness of Theorem 1.

In Fig. 6, we compare the utility of all threads before and after ***IA***. The result of the Before Algorithm is the utility calculated from the initially randomly selected scheduling strategy (Algorithm 1 Line 1), and the result of the After Algorithm shows the utility of each thread after ***IA*** is executed. Obviously, the thread scheduling scheme based on the non-cooperative game can separate the threads with high miss rate into different scheduling batches, so that the utility of each thread are more balanced. It can be observed that after executing the ***IA***, the utility of each thread is almost the same, rendering the thread fair to be scheduled.

Fig. 7 shows the performance degradation of each scheduled batch as the number of iterations increases when the total scheduling batch is 6. We can see that the performance degradation of all scheduling batches will eventually reach a relatively stable state, that is, the Nash equilibrium. At this time, the performance degradation of all batches is roughly equal. Theorem 4 proves that the ***IA*** converges to Nash equilibrium at most in N iterations, where N is the number of threads.

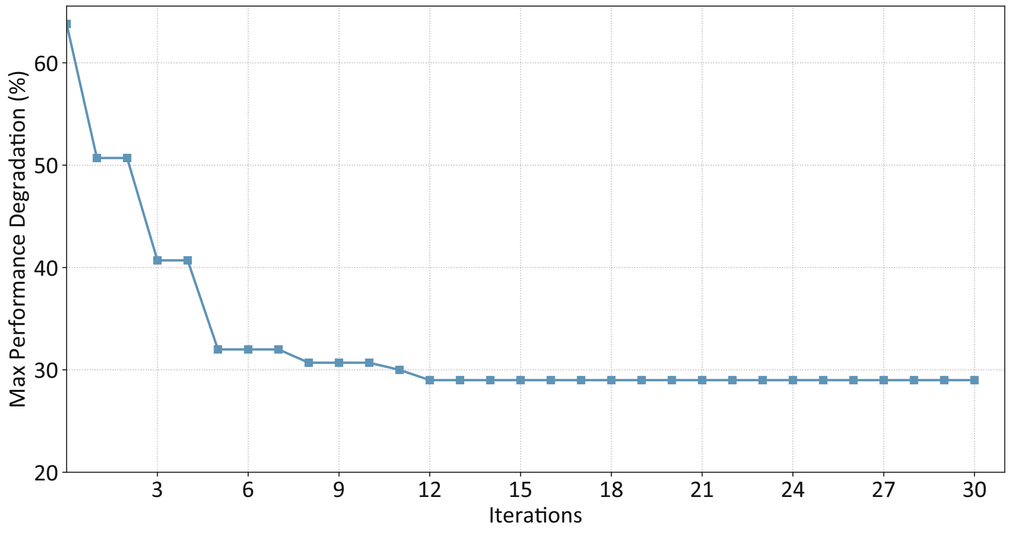


Fig. 5. Iterative process of optimization function.

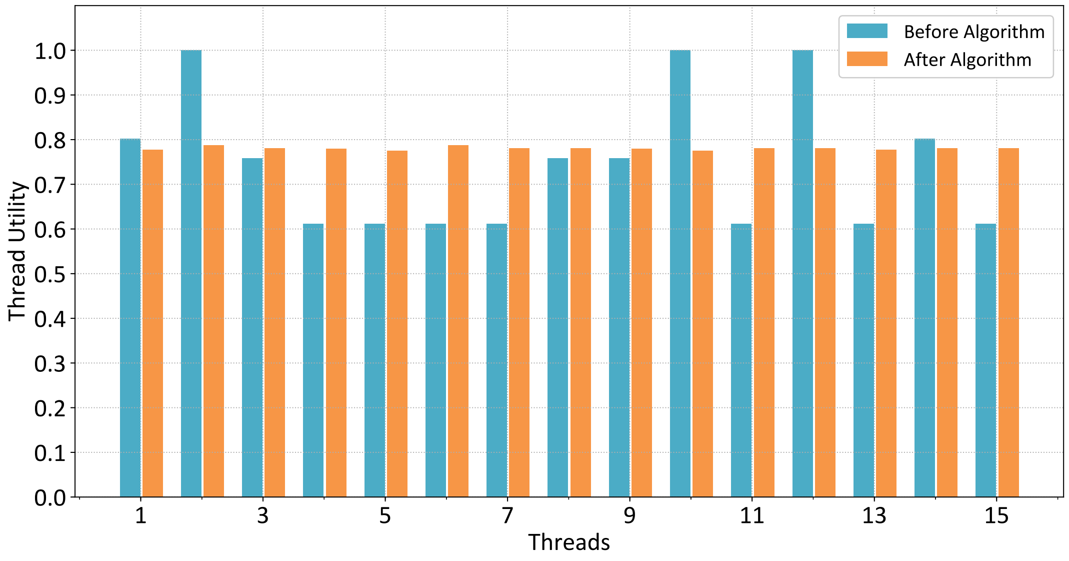


Fig. 6. Comparison of user utility before and after ***IA*** algorithm.

It can also be seen from Fig. 7 that the ***IA*** has reached a relatively stable state after 12 iterations, which confirms the correctness of Theorem 4 and also demonstrates the efficiency of the ***IA***.

Next, we will show the results of changes in utility of different threads calculated by ***IA*** as the number of iterations increases. We randomly select 6 threads (thread 3, thread 4, thread 6, thread 10, thread 13 and thread 15) from 15 threads. When the total number of scheduled batches is 6 ($T = 6$), the utility of these 6 threads changes as shown in Fig. 8. We can see that the utility of all threads eventually reaches a relatively stable state with the increase in the number of iterations. Nonetheless, in the process of iteration, the utility of the thread fluctuates up and down. This is because during the iteration process, when a thread's strategy changes, it affects the utility of other threads. For example, the strategy of thread i changes from t to t', which increases the utility of threads with policy t, while the utility of threads with strategy t' decreases. When the utility of all threads reaches a relatively stable state, then the scheduling strategy of all threads will not change at this time. That is, after a certain number of iterations, the strategy of N threads reaches the Nash equilibrium, which reflects the convergence of the ***IA***. Fig. 8 also shows that when the Nash equilibrium is reached, the utility value of threads is slightly different, which also achieves the purpose of fairly scheduling threads.

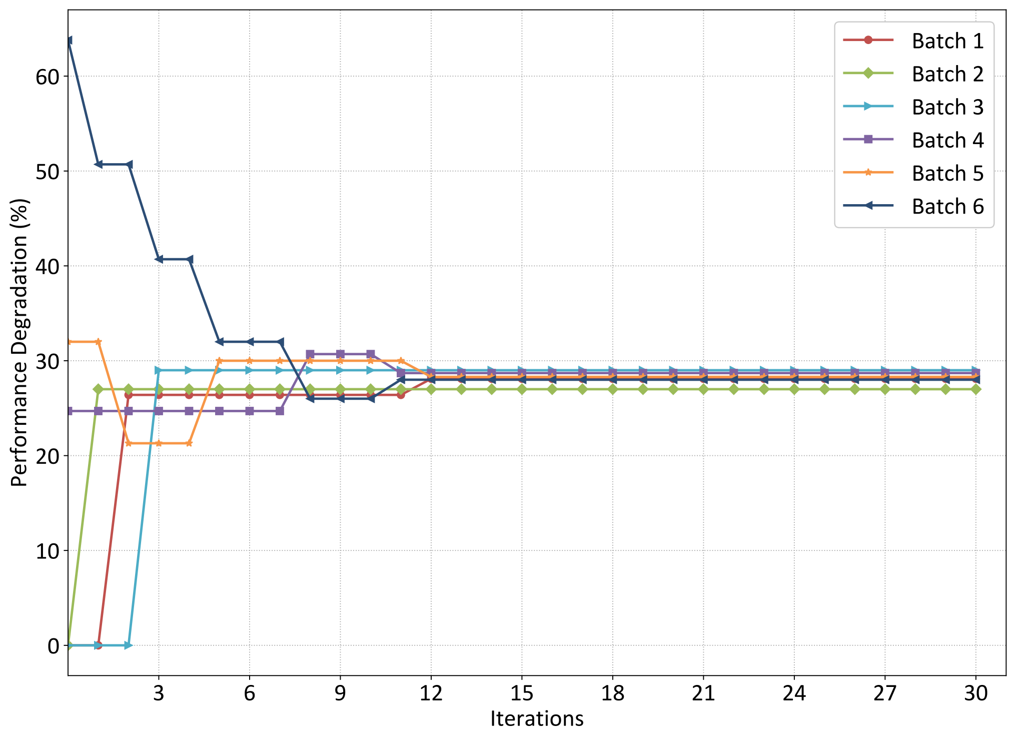


Fig. 7. Iterative process of performance degradation.

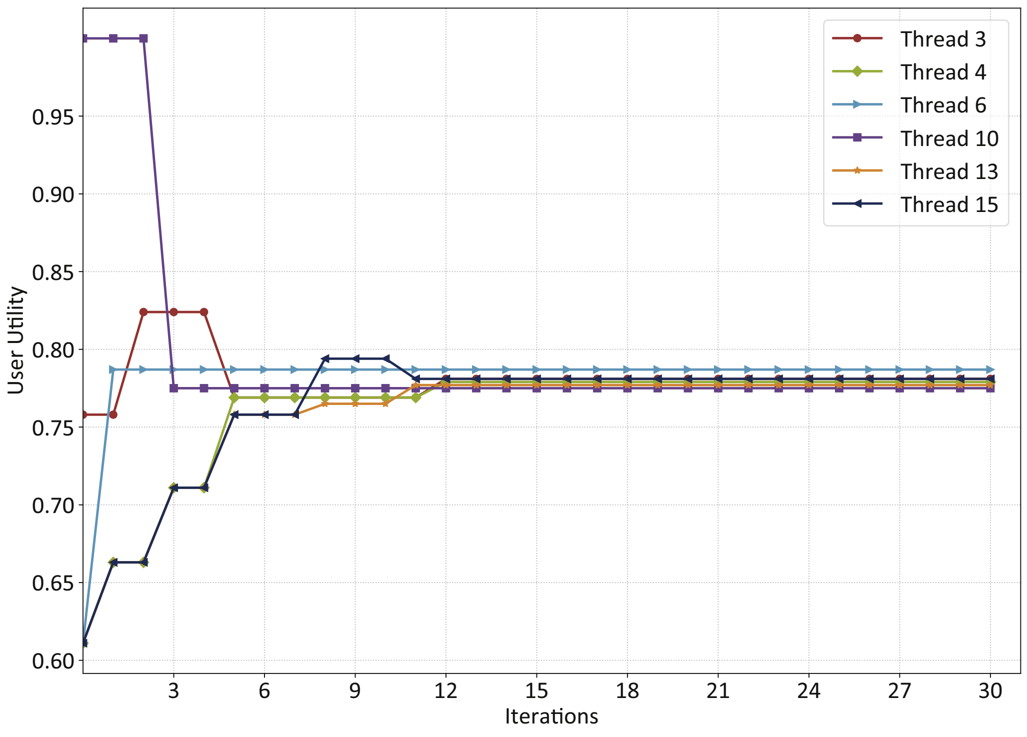


Fig. 8. Iterative process of user utility.

Table 3
Thread scheduling scheme base on default algorithm.

Scheduling batch	Threads
batch 1	Thd_1, Thd_7, Thd_{13}
batch 2	Thd_2, Thd_8, Thd_{14}
batch 3	Thd_3, Thd_9, Thd_{15}
batch 4	Thd_4, Thd_{10}
batch 5	Thd_5, Thd_{11}
batch 6	Thd_6, Thd_{12}

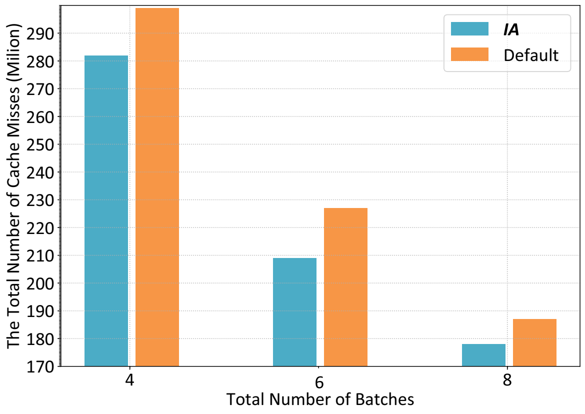


Fig. 9. Comparison of the total number of cache misses between ***IA*** and default algorithm.

5.4. Performance simulation

In this paper, we use the Frequency of Access (FOA) model proposed in [32] to predict cache contention when threads shared L2 cache. When n threads are co-scheduled and share the L2 cache, the effective size of the cache obtained by thread i can be calculated from:

$$effCacheSize_i = \frac{Af_i}{\sum_{j=1}^{n} Af_j} \times TotalCacheSize \tag{10}$$

where $TotalCacheSize$ represents the total L2 cache size, and Af_i represents the access frequency of thread i, and then the effective cache space of thread i can be calculated from:

$$M'_i = \frac{effCacheSize_i}{numCacheSet \times LineSize} \tag{11}$$

where $numCacheSet = \frac{TotalCacheSize}{M \times LineSize}$, $LineSize$ represents the cache size of each line. Then, by plugging M' into Eq. (9), the cache miss rate of threads i under cache sharing can be calculated.

We have conducted a total of 2 sets of comparative experiments, showing the number of L2 cache misses when the ***IA*** and the default scheduling algorithm with and without the cache partition. In this paper, the default scheduling algorithm assigns each thread to the scheduling batch in turn according to the order in which the threads arrive (the subscript of the thread). The scheduling scheme of the default scheduling algorithm is shown in Table 3, maximum difference of missing rate is 0.211.

Fig. 9 shows the total number of L2 cache misses for 15 threads running under ***IA*** and the default scheduling algorithm when the total scheduling batch is 4, 6, and 8. It can be seen that the total L2 cache misses under the ***IA*** is significantly less than the number of misses under the default algorithm. In this paper, the ***IA*** is used to solve the thread co-scheduling scheme, which minimizes the total misses of L2 cache.

Fig. 10 shows the number of L2 cache misses for each thread under ***IA*** and the default scheduling algorithm. We can see that the ***IA*** algorithm is not strictly superior to the default scheduling algorithm for the number of L2 misses per thread. For some threads (such as Thd_{12}, Thd_{15}), the default scheduling algorithm is better than the ***IA***, because when using different thread scheduling schemes, the impact of each thread suffering from co-scheduled threads is not the same. In this paper, we mainly consider the overall performance, and Fig. 9 shows that the total number of misses based on the ***IA*** is lower than the total misses of the default scheduling algorithm.

In the above experiment, we only reduced the contention of the shared cache from the level of thread co-scheduling. Recently showed experimentally that important gains could be reached by co-scheduling applications with strict cache par-

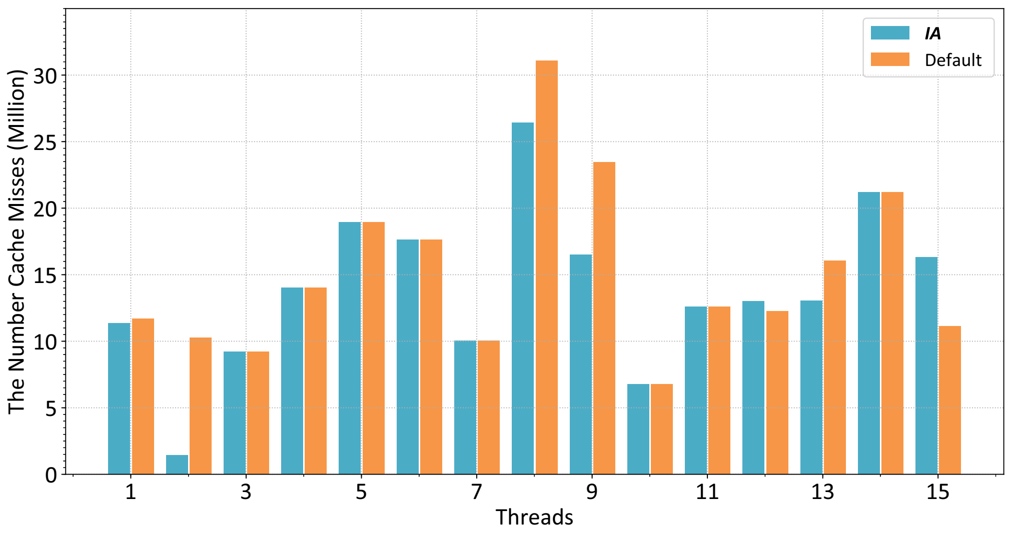
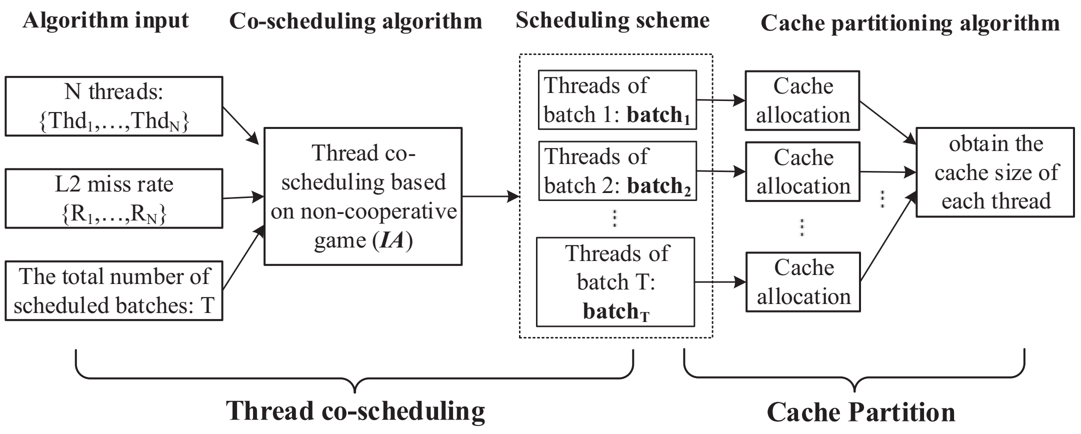


Fig. 10. Comparison of the number of thread misses between ***IA*** and default algorithm.



Algorithm input
Co-scheduling algorithm
Scheduling scheme
Cache partitioning algorithm
N threads: {Thd1,…,ThdN}
L2 miss rate {R1,…,RN}
The total number of scheduled batches: T
Thread co-scheduling based on non-cooperative game (IA)
Threads of batch 1: batch1
Threads of batch 2: batch2
Threads of batch T: batchT
Cache allocation
Cache allocation
Cache allocation
obtain the cache size of each thread
Thread co-scheduling
Cache Partition

Fig. 11. Combine thread co-scheduling with cache partitioning.

titioning enabled [38]. Next, we will use cache partitioning to further improve ***IA*** performance, the number of shared cache misses can be reduced from the two aspects of thread co-scheduling and cache partitioning.

5.5. Performance improvement

As shown in Fig. 11, to further improve IA performance, we combine thread co-scheduling with cache partitioning. the overall optimization scheme consists of two parts: thread co-scheduling and cache partitioning. In this paper, we optimize the overall system performance by adjusting the thread's execution order and selecting threads with small conflicts and small disturbances to be scheduled in the same batch (see Sections 3 and 4 for details). After obtaining the total co-scheduling scheme, we solve the result of a cache partition for each scheduled batch, with an appropriate separate cache space allocated to each thread (see Section 5.5.1 for details).

5.5.1. A cache partitioning algorithm

In this section, we will introduce the cache partitioning algorithm, which aims to maximize throughput [39]. Suppose that n threads are running in the same scheduled batch, the shared L2cache is divided into S groups, each group containing M ways. To find the optimal cache partitioning, the total number of cache misses of n threads shall be the smallest, and then the minimum value of $\sum_{i=1}^{n} miss_i$ is obtained. This paper uses a greedy allocation strategy to partition the cache, and allocates a cache to one of the threads in each iteration until the cache resources are divided. The steps of the algorithm are as follows:

Step 1. Define the gain function $g(x)$ for each thread, where $g_i(x)$ represents the number of cache hits added to thread i when thread i obtains the cache from x ways to $x+1$ ways. In this paper $g_i(x) = C_i(x)$ $(x \in \{1, \ldots, M\})$.

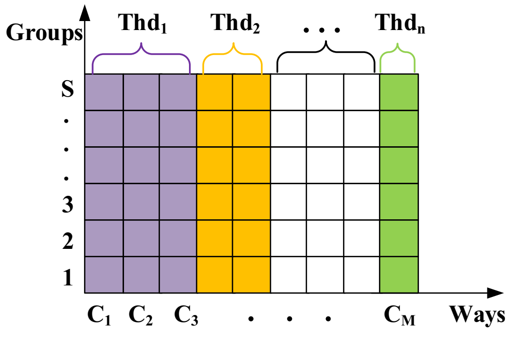


Fig. 12. Illustration of cache partitioning.

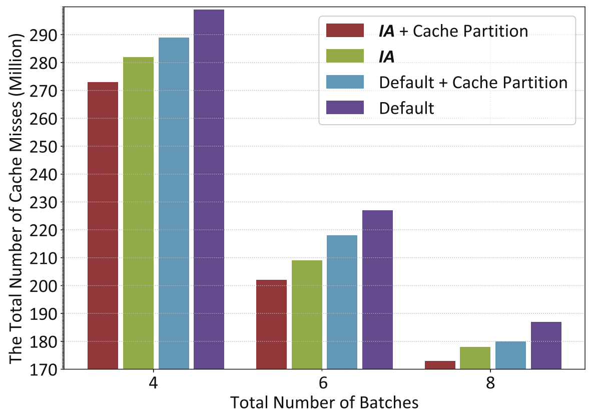


Fig. 13. Comparison of the total number of cache misses.

Table 4
Comparison of total number of misses.

Total number of batches	Decrease (%)		
	IA+Cache Partition vs. *IA*	*IA* vs. Default	*IA*+Cache Partition vs. Default
$T = 4$	3.19	5.69	8.7
$T = 6$	3.35	7.93	11.01
$T = 8$	2.81	4.81	7.49

Step 2. Initialize $\boldsymbol{S} = (S_1, S_2, \cdots, S_n) = (1, 1, \cdots, 1)$, where S_i represents the number of cache lines allocated to thread *i*. At the beginning of the algorithm, all threads are guaranteed to obtain the cache resources.

Step 3. Add 1 line cache to the thread with the largest gain in the current partition.

Step 4. Repeat step 3 until the cache resources are divided.

Step 5. Returns to the cache allocation scheme ***S*** for each thread.

After executing the cache partitioning algorithm, the cache space obtained by each thread is roughly as shown in Fig. 12. The sum of the cache spaces obtained by the same batch of scheduled threads is the total capacity of the shared cache. Because different threads have different sensitivity to the obtained cache space, the cache space obtained by different threads may be different.

5.5.2. Experimental results

Fig. 13 shows the total number of L2 cache misses for 15 threads running in 4 cases (***IA, IA***+Cache Partition, Default Algorithm, Default Algorithm+Cache Partition) when the total scheduling batch is 4, 6, and 8. It can be seen that the total number of L2 cache misses after using the cache partitioning algorithm is significantly less than that of misses without the cache partitioning algorithm(***IA*** + Cache Partition vs. ***IA***, Default + Cache Partition vs. Default). In this paper, the ***IA*** is used to solve the thread scheduling scheme, and then the cache partitioning algorithm proposed in Section 5.5.1 is used to allocate L2 cache space for each thread, which minimizes the total misses of L2 cache.

As can be seen from Table 4, when the total scheduled batch is 6, the total number of misses under the ***IA*** is reduced by 7.93% in comparison with the default algorithms. When using the ***IA*** and the cache partitioning algorithm (***IA***+Cache Partition), the total number of misses is reduced by 11.01% compared with the default scheduling algorithm. When changing

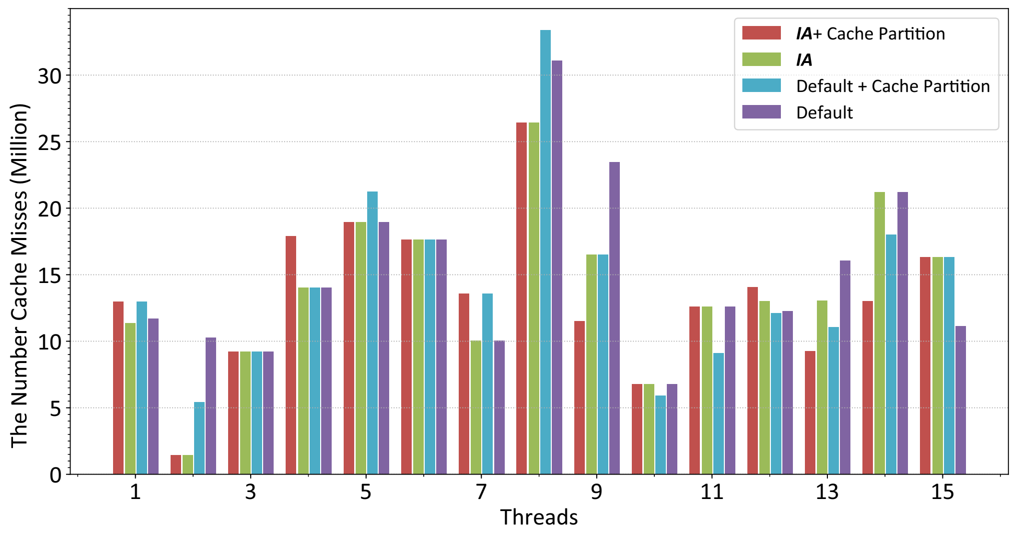


Fig. 14. Comparison of the number of thread misses.

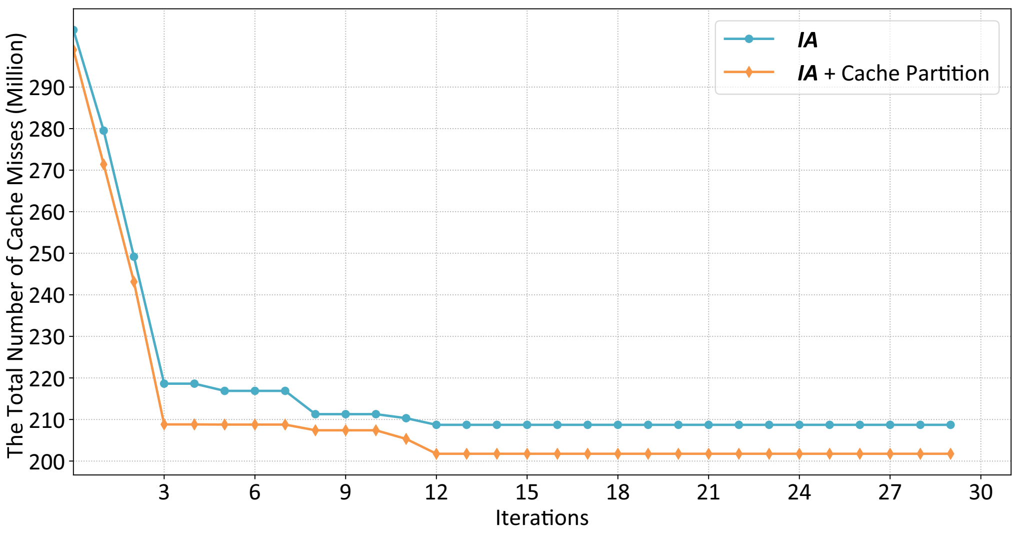


Fig. 15. Iterative process of the total number of cache misses.

the number of total scheduled batches, using the combinatorial optimization scheme can still achieve better system performance than optimizing system performance from a single perspective. The algorithm in this paper does not guarantee the optimal performance of a single thread, but rather improves the overall performance of the system by reducing cache contention.

Fig. 14 shows the number of L2 cache misses for each thread under 4 scheduling schemes. When comparing the total number of L2 cache misses for each thread with and without the cache partitioning algorithm, we can also find that the difference between the two cases is not obvious, because the purpose of the cache partitioning algorithm is to minimize the total number of cache misses. Table 2 demonstrates that thread 1 and thread 13 are in the same scheduled batch based on ***IA***. As seen from Fig. 14, when the cache partitioning algorithm is executed, the number of cache misses of thread 1 is increased (red bar vs. green bar), but the number of cache misses of thread 13 experiences a sharper decrease (red bar vs. green bar). The number of cache misses of batch 4 is reduced compared with no cache partition. There is no guarantee that the number of misses of each thread will be reduced.

Fig. 15 shows that, when the total number of scheduled batches is 6, the total number of L2 cache misses decreases as the number of iterations increases. The green line in the figure indicates the change in the number of misses in the case where only the ***IA*** is used, and the orange line indicates the change in the number of misses in the case of using the ***IA***+cache partition algorithm. As seen from the figure, the number of total miss using the ***IA*** decreases (green line) as the number of iterations increases and eventually reaches a stable state, this proves the effectiveness of the ***IA***. We can also see that partitioning the cache after obtaining the ***IA*** based scheduling scheme can further reduce the total cache miss.

6. Conclusion

In this paper, thread co-scheduling is used to reduce the contention of shared caches on the CMP architecture, and the non-cooperative game is used to model multi-thread co-scheduling. Firstly, a linear relationship between the miss rate of thread co-scheduling and the performance degradation is established, which is used to calculate the performance degradation during the thread co-scheduling. Then, the competition relationship between threads on shared resources is modeled as a non-cooperative game and treat each thread as a participant in the game. We propose an ***IA*** for solving the Nash equilibrium of the game and its convergence is theoretically demonstrated. The time complexity of our analysed ***IA*** is $\Theta(N\log N)$ and it can converge to Nash equilibrium in N iterations, where N is the number of threads in the system. After obtaining the co-scheduling scheme by solving the Nash equilibrium of ***IA***, we further optimized the space for sharing the L2 cache. Each thread is allocated a separate L2 cache space based on the greedy idea of minimizing the total number of cache misses. Finally, we verified the convergence and validity of ***IA*** through experiments, and the effectiveness of thread co-scheduling combined with cache partitioning scheme is also verified. Experiments have proved that the total L2 cache misses of ***IA*** is less than that of the default scheduling algorithm. The combination of the cache partitioning and thread co-scheduling can further reduce the number of misses in the shared cache compared to the circumstance with only thread co-scheduling.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Zheng Xiao: Conceptualization, Methodology, Validation, Funding acquisition, Resources. **Liwen Chen:** Investigation, Formal analysis, Writing - original draft. **Bangyong Wang:** Data curation, Visualization, Software. **Jiayi Du:** Visualization, Software. **Keqin Li:** Writing - review & editing.

Acknowledgments

This work is partially supported by Natural Science Foundation of China (No. 61872129 and No. 61802444), Hunan Provincial Natural Science Foundation of China (No. 2019JJ40024) and Doctoral Scientific Research Foundation of Central South University of Forestry and Technology (No. 2016YJ047).

References

[1] J. Leverich, C. Kozyrakis, Reconciling high server utilization and sub-millisecond quality-of-service, in: Proceedings of the European Conference on Computer Systems, 2014.
[2] M.K. Qureshi, Y.N. Patt, Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches, in: Proceedings of the IEEE/ACM International Symposium on Microarchitecture, 2006.
[3] L.R. Hsu, S.K. Reinhardt, R.R. Iyer, S. Makineni, Abstract communist, utilitarian, and capitalist cache policies on CMPS: caches as a shared resource, in: Proceedings of the International Conference on Parallel Architectures & Compilation Techniques, 2006.
[4] A. El-Moursy, R. Garg, D.H. Albonesi, S. Dwarkadas, Compatible phase co-scheduling on a CMP of multi-threaded processors, in: Proceedings of the International Conference on Parallel & Distributed Processing, 2006.
[5] S. Kim, D. Chandra, S. Yan, Fair cache sharing and partitioning in a chip multiprocessor architecture, 2004.
[6] D. Sanchez, C. Kozyrakis, Vantage: scalable and efficient fine-grain cache partitioning, ACM SIGARCH Comput. Arch. News 39 (3) (2011) 57–68.
[7] J.L. Kihm, A. Settle, A. Janiszewski, D.A. Connors, Understanding the impact of inter-thread cache interference on ILP in modern SMT processors., J. Instruct.-level Parall. 7 (2005).
[8] Y. Jiang, X. Shen, C. Jie, R. Tripathi, Analysis and approximation of optimal co-scheduling on chip multiprocessors, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2008.
[9] S. Blagodurov, S. Zhuravlev, A. Fedorova, Contention-aware scheduling on multicore systems, ACM Trans. Comput. Syst. 28 (4) (2010) 1–45.
[10] I. Akturk, O. Ozturk, Adaptive thread scheduling in chip multiprocessors, Int. J. Parallel Program (1) (2019) 1–31.
[11] S. Zhuravlev, J.C. Saez, S. Blagodurov, A. Fedorova, M. Prieto, Survey of scheduling techniques for addressing shared resources in multicore processors, ACM Comput. Surv. 45 (1) (2012) 1–28.
[12] G. Nan, M. Stigge, Y. Wang, Y. Ge, Cache-aware scheduling and analysis for multicores, in: Proceedings of the ACM & IEEE International Conference on Embedded Software, 2009.
[13] M. Moreto, F.J. Cazorla, A. Ramirez, R. Sakellariou, M. Valero, Flexdcp: a QOS framework for CMP architectures, ACM SIGOPS Oper. Syst. Rev. 43 (2) (2009) 86–96.
[14] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, S. Yan, C. Kozyrakis, From chaos to QOS: case studies in CMP resource management, ACM SIGARCH Comput. Arch. News 35 (1) (2007) 21–30.
[15] K.J. Nesbit, J. Laudon, J.E. Smith, Virtual private caches, ACM SIGARCH Comput. Arch. News 35 (2) (2007) 57–68.
[16] A.D. Blanche, T. Lundqvist, Addressing characterization methods for memory contention aware co-scheduling, J. Supercomput. 71 (4) (2015) 1451–1483.
[17] A. Hassidim, H. Kaplan, O. Tuval, Joint cache partition and job assignment on multi-core processors, in: Proceedings of the International Conference on Algorithms and Data Structures, 2013.
[18] C. Gang, H. Kai, H. Jia, A. Knoll, Cache partitioning and scheduling for energy optimization of real-time MPSOCS, in: Proceedings of the IEEE International Conference on Application-specific Systems, 2013.

[19] M. Jahre, L. Natvig, A light-weight fairness mechanism for chip multiprocessor memory systems, in: Proceedings of the ACM Conference on Computing Frontiers, 2009.
[20] A. Hassidim, Cache replacement policies for multicore processors, in: Proceedings of the Symposium on Innovations in Computer Science, 2009.
[21] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, H.J. Siegel, A methodology for co-location aware application performance modeling in multicore computing, in: Proceedings of the Parallel & Distributed Processing Symposium Workshop, 2015.
[22] M. Shantharam, Y. Youn, P. Raghavan, Speedup-aware co-schedules for efficient workload management, Parallel Process. Lett. 23 (02) (2013).
[23] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, P. Raghavan, Co-scheduling algorithms for high-throughput workload execution, J. Schedul. 19 (6) (2016) 1–14.
[24] Y. Xie, G. Loh, Dynamic classification of program memory behaviors in CMPS, in: Proceedings of the Second Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2008.
[25] C. Hankendi, A.K. Coskun, Reducing the energy cost of computing through efficient co-scheduling of parallel workloads, in: Proceedings of the Conference on Design, 2012.
[26] S. Zhuravlev, S. Blagodurov, A. Fedorova, Addressing shared resource contention in multicore processors via scheduling, in: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages & Operating Systems, 2010.
[27] G. Dhiman, G. Marchetti, T. Rosing, vgreen: a system for energy efficient computing in virtualized environments, in: Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, 2009.
[28] A. Merkel, J. Stoess, F. Bellosa, Resource-conscious scheduling for energy efficiency on multicore processors, in: Proceedings of the European Conference on European Conference on Computer Systems, 2010.
[29] R. Knauerhase, P. Brett, B. Hohlt, T. Li, S. Hahn, Using os observations to improve performance in multicore systems, IEEE Micro 28 (3) (2008) 54–66.
[30] M. Banikazemi, E.P. Dan, B. Abali, Pam: A novel performance/power aware meta-scheduler for multi-core systems, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2008.
[31] G. Daci, M. Tartari, A comparative review of contention-aware scheduling algorithms to avoid contention in multicore systems, 2013.
[32] D. Chandra, F. Guo, S. Kim, S. Yan, Predicting inter-thread cache contention on a chip multi-processor architecture, in: Proceedings of the International Symposium on High-performance Computer Architecture, 2005.
[33] R. Friese, T. Brinks, C. Oliver, H.J. Siegel, A.A. Maciejewski, Analyzing the trade-offs between minimizing makespan and minimizing energy consumption in a heterogeneous resource allocation problem, in: Proceedings of the INFOCOMP, Second International Conference on Advanced Communications and Computation, 2012, pp. 81–89.
[34] A.S. Schulz, N. Immorlica, L. Li, V.S. Mirrokni, Coordination mechanisms for selfish scheduling., Theor. Comput. Sci. 410 (17) (2008) 1589–1598.
[35] C. Cascaval, D.A. Padua, Estimating cache misses and locality using stack distances, in: Proceedings of the International Conference on Supercomputing, 2003.
[36] C. Cascaval, L. Derose, D.A. Padua, D.A. Reed, Compile-Time based performance prediction, 1999.
[37] G.E. Suh, S. Devadas, L. Rudolph, A new memory monitoring scheme for memory-aware scheduling and partitioning, in: Proceedings of the International Symposium on High-performance Computer Architecture, 2002.
[38] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Improving resource efficiency at scale with heracles, ACM Trans. Comput. Syst. 34 (2) (2016) 1–33.
[39] G.E. Suh, L. Rudolph, S. Devadas, Dynamic partitioning of shared cache memory, J. Supercomput. 28 (1) (2004) 7–26.