Scalpel: High Performance Contention-Aware Task Co-Scheduling for Shared Cache Hierarchy

Song Liu, Jie Ma, Zengyuan Zhang, Xinhe Wan, Bo Zhao, and Weiguo Wu

***Abstract*—For scientific computing applications that consist of many loosely coupled tasks, efficient scheduling is critical to achieve high performance and good quality of service (QoS). One of the challenges for co-running tasks is the frequent contention for shared cache hierarchy of multi-core processors. Such contention significantly increases cache miss rate and therefore, results in performance deterioration for computational tasks. This paper presents Scalpel, a contention-aware task grouping and co-scheduling approach for efficient task scheduling on shared cache hierarchy. Scalpel utilizes the shared cache access features of tasks to group them in a heuristic way, which reduces the contention within groups by achieving equal shared cache locality, while maintaining load balancing between groups. Based thereon, it proposes a two-level scheduling strategy to schedule groups to processors and assign tasks to available cores in a timely manner, while considering the impact of task scheduling on shared cache locality to minimize task execution time. Experiments show that Scalpel reduces the shared cache miss rate by up to 2.14× and optimizes the execution time by up to 1.53× for scientific computing benchmarks, compared to several baseline approaches.**

***Index Terms*—Task co-scheduling, shared cache contention, high computing resource utilization, performance optimization.**

I. Introduction

THE Many-Task Computing (MTC) paradigm [1] is usually used in scientific computing to decouple a complex problem into many sub-problems for computation. In MTC, a large-scale loosely coupled application consists of many tasks, and it has achieved remarkable results in various scientific fields, such as quantum circuit [2], molecular dynamics [3], and nuclear physics simulation [4]. These tasks are generally compute-intensive and have different resource occupation within their execution time. An efficient task scheduling approach is therefore critical for the execution performance and quality of service (QoS) of scientific computing applications, including high processing throughput and low latency.

Existing scheduling approaches mainly concentrate on the scaling-out performance through distributing tasks to computing nodes for efficient execution, and maintain load-fairness [5], [6] and superior energy efficiency [7], [8]. However, as the multi-core processors are equipped with increasing number of cores, the contention for the shared resources by co-running tasks (or co-runners), especially the shared cache hierarchy, becomes the main challenge of task scheduling. Tasks' competition for the shared last level cache (LLC) raises the cache miss ratio and therefore, incurs expensive off-chip memory accesses, resulting in performance deterioration and energy consumption increase.

There are generally two kinds of approaches to mitigate shared cache contention, i.e., cache partitioning and cache-aware scheduling. The cache partitioning techniques [9], [10] isolate shared caches for exclusive use by tasks based on their caching requirements. But they are costly and require additional hardware components [11], or software control (e.g., page-coloring [12], partitioning-aware compiler [13]) to support implementation. On the other hand, the cache-aware scheduling approaches provide a more easily implemented and straightforward way. They improve the cache utilization through optimizations such as task resource consumption model [14], cache behavior estimation [15], and scheduling parameter analysis and tuning [16], thus alleviating cache contention to some extent. However, these approaches do not directly delve into and solve the shared cache contention problem, so there is still much room for competing tasks to improve the performance.

In this paper, we work on solving the problem of contention for shared cache hierarchy to improve the execution performance of many-task applications. Due to the variance in competition between tasks, **the contention for shared cache varies significantly with different co-runners, and thus leads to distinct execution performance**. Such contention affects both the individual and total execution time of tasks. Therefore, an effective task scheduling strategy allows to keep tasks shared-cache-friendly during their co-running and minimize the contention for a better execution performance. Intuitively, the brute-force algorithm may be a solution. But it is infeasible in practice due to the explosive searching overhead when the number of tasks is large.

This paper proposes *Scalpel*, a novel contention-aware task co-scheduling approach for shared cache hierarchy. It focuses on mitigating shared contention between co-running tasks and improving the execution performance. Scalpel achieves the goal by task grouping and task scheduling. The task grouping identifies

Received 15 February 2023; revised 21 October 2024; accepted 25 October 2024. Date of publication 18 November 2024; date of current version 20 January 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62002279, in part by the Natural Science Basic Research Program of Shaanxi under Grant 2020JQ-077, and in part by the Science Challenge Project under Grant TZZT2019-B2.2. Recommended for acceptance by P. Stenstrom. *(Corresponding author: Weiguo Wu.)*

Song Liu, Jie Ma, Zengyuan Zhang, Xinhe Wan, and Weiguo Wu are with the School of Computer Science and Technology, Xi'an Jiaotong University, Shaanxi 710049, China (e-mail: liusong@xjtu.edu.cn; mj_doris@stu.xjtu.edu.cn; zhangzengyuan@stu.xjtu.edu.cn; wanxinhe@stu.xjtu.edu.cn; wgwu@xjtu.edu.cn).

Bo Zhao is with the Department of Computer Science, Aalto University, 02100 Espoo, Finland (e-mail: bo.zhao@aalto.fi).

Digital Object Identifier 10.1109/TC.2024.3500381

appropriate groups of tasks to alleviate the contention between tasks within a group, and the task scheduling determines the schedules of task groups and tasks in each group to available processors and cores, maximizing computing resource utilization. It also keeps load balancing during tasks' execution. The main contributions of this work are as follows.

- We propose a shared cache-aware task grouping method to mitigate cache contention. It leverages the footprint to accurately capture the locality of co-running tasks for shared cache and groups tasks in a heuristic way to achieve equalized locality, while maintaining load balancing among task groups. This grouping method allows the co-running tasks in each group to mitigate shared cache contention and benefit from the locality of shared cache for execution.
- We propose a two-level task scheduling strategy to improve the execution performance of the grouped tasks. The first level of scheduling determines the execution order of groups on processors, and the second level assigns appropriate in-group tasks to available cores timely, thus avoiding idle computing cores and fully utilizing computing resources. It also considers the contention when tasks from different groups are reassociated during scheduling and reduces its impact on execution performance.
- We evaluate the proposed task co-scheduling approach over different types and numbers of scientific computing tasks on multiprocessors. The proposed approach has been verified to reduce the LLC miss rate by up to 2.14$\times$ and improve the execution performance by up to 1.53$\times$, compared to five alternative task scheduling approaches.

The rest of this paper is organized as follows. Section II discusses related work and investigates the problems that is not well considered in previous work. Section III gives an overview of our approach and the problem statements. Section IV describes the task grouping method. Section V presents the task scheduling strategy. Section VI shows the experiments and evaluation results. Section VII concludes this paper and points out the indications for future work.

II. Related Work

Resource contention on multi-core architectures has always been a concern, such as shared cache contention [18], [19], memory controller contention [20], data bus contention [20], and network contention [22]. The cores of multiprocessors on a chip share the LLC, and with that comes contention for the shared cache. The shared cache contention can lead to poor execution performance of applications. Existing approaches to tasks scheduling for large scale computing platforms mainly focus on improving the performance or the quality of service at the inter-node level [6], [8]. In this work, we aim at solving the performance degradation of parallel execution of tasks caused by shared cache contention on the intra-node multiprocessor to further improve the execution performance.

Cache partitioning is an effective solution to alleviate the shared cache contention by dividing shared cache into different partitions, so that parallel tasks can access different partitions exclusively and thus avoid inter-core interference. According to the partitioning unit, the partitioning can be divided into set-based partitioning [23], associativity-based partitioning [24], and block-based partitioning [25]. The implementations of cache partitioning generally need special hardware [26], [27], [28] or extra software support [12], [29]. The software-based cache partitioning (e.g., page coloring methods [29]) generally has poor scalability and high overhead. Besides, the implementations require modifications to operating systems. The hardware-supporting partition methods are more efficient. The cache allocation technology (CAT) [28] proposed by the Intel enables programmers to control shared cache through software means. CAT is employed in [27] to partition cache to avoid cache conflicts as much as possible. Selfa et al. [24], [30] allocated a cache partition for each application in the unit of cache way (a collection of cache lines), and dynamically adjusted the partition size at run time based on a hardware method. El-Sayed et al. [31] proposed KPart which uses hierarchical clustering to group applications into clusters according to cache miss curves. But KPart suffers from the overhead of detailed profiling. Pons et al. [32] proposed CPA to improve the turnaround time by adjusting the allocated cache ways, but it requires extensive experiments to obtain the thresholds for application classification. Saez et al. [33] proposed LFOC+ to classify applications into three classes according to different cache behaviors and assign with different cache ways. However, it focuses on the fairness of cache clustering policy. On the other hand, the scheduling of recurrent tasks in the real-time system [34], [35], [36], [37] also utilizes cache partitioning to address the problem of WCET prediction caused by the inter-core interference. However, this is different from the application scenario of this paper, i.e., to reduce the impact of shared cache contention on the makespan of co-running tasks without deadline requirements. These cache-aware partitioned scheduling methods are effective in avoiding shared cache contention by physically isolating tasks, but can lead to cache utilization problems.

Research shows that the degree of contention for shared caches mainly depends on the thread combination in collaborative scheduling [38]. Therefore, an effective scheduling strategy can alleviate the shared cache contention and improve the runtime performance of concurrent programs. Some studies try to use heuristic methods [39], [40] or graph modelling [41] to obtain task scheduling strategy. In [38], an improved particle swarm optimization algorithm was proposed for heterogeneous multiprocessors. Sheikh et al. [40] modeled the cache contention between tasks as a dependency graph for the minimum energy consumption scheduling algorithm. Besides, many studies designed scheduling strategies by analyzing different characteristics of programs, such as fair progress [42], cache conflicts [43], and data locality of tasks [44], [45]. It has been proven that cache contention is more likely to occur between warps with high data locality [44], and thus Li et al. [45] grouped the cooperative tasks with k-means clustering for scheduling. Zhang et al. [46] improved the utilization of shared cache through co-scheduling after transforming programs to be more compatible with multicore processor architecture. But it is only applied to the multithreaded programs and the situation will become very complicated when the number of cores is large. Such methods are applied to thread-level tasks with data sharing, which are usually spawned by the process of program or subroutine. For MTC applications, loosely-coupled tasks or subroutines are distributed to processors for execution as processes. Existing methods are not suitable for this scenario.

To the best of our knowledge, a few methods of process-level task scheduling are suitable for our target scenario, but they also have different problems. In the real-time systems, many studies [34], [35] consider the shared cache contention for process-level task scheduling. But they focus on the WCET prediction of tasks. For many computing scenarios [4], [47], there is a constant quest for the optimal execution performance of tasks. Zhuravlev et al. [48] proposed a coarse-grained task scheduling for independent process-level tasks. They evaluated the effects of various competing resources on the performance of shared cache, and used the cache miss rate of single task to form task groups for mitigating the resource contention. However, their approach assumes the same number of tasks and cores, and it does not provide in detail that how to apply the proposed scheduling to the case when the number of tasks does not match the number of cores. Xiao et al. [39] used non-cooperative game theory to construct a task scheduling model. It considered each independent task as a participant of the game, and obtained the cooperative scheduling strategy by solving the Nash equilibrium. This scheduling approach focuses only on minimal shared cache contention between task groups and does not consider computing core utilization during tasks' execution, thus leading to low resource utilization and poor execution performance.

Since the scheduling approach does not require additional hardware support with good practicality and scalability, this work tries to solve the shared cache contention problem by task scheduling. Our approach not only can effectively reduce cache contention for computational tasks, but also makes full use of available computing cores to improve the execution performance of co-running tasks.

III. Overview of the Proposed Approach

First, we present an overview of the proposed approach, as shown in Fig. 1. Scalpel can be divided into two phases, namely static procedure and dynamic procedure. Notice that the SMP architecture in Fig. 1 is just an example. There is no restriction on the number of processors and cores when applying Scalpel. Table I defines the notations commonly used in this paper.

Task grouping is first performed in a static procedure. The task analysis can obtain the information of each task, including the memory access trace and the execution time. Then, tasks are grouped according to shared cache locality quantification which is estimated using the obtained information of tasks as well as the hardware information, such as P, C, and $LLCSize$. The grouping method focuses on mitigating the shared cache contention within each group and keeping load balance of task groups. Based thereon, the efficient task grouping can be formulated as solving **Problem 1**.

Problem 1 (efficient task grouping). Given a task set of T tasks, $TaskSet = \{Task_1, Task_2, \ldots, Task_T\}$, to co-run on a SMP computing node of $<P, C>$ multi-core processors, the problem

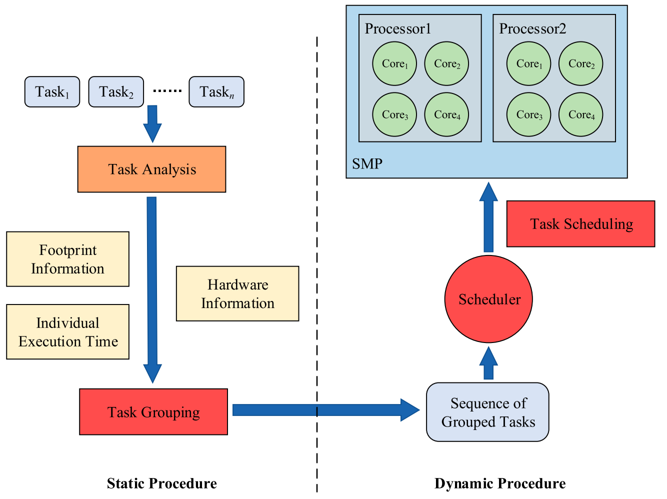


Fig. 1. Overview of Scalpel.

TABLE I
Notations

Notations	Explanation
T	# of tasks
P	# of processors in a computing node
C	# of cores on a processor, sharing an LLC
$LLCSize$	Capacity of LLC
G	Number of task groups
$TGroup[i]$	The i-th task group, $i = 1, 2, \ldots, G$
$LocTGroup_i$	Shared cache locality of $TGroup[i]$
$M_i = \|TGroup[i]\|$	Number of tasks in $TGroup[i]$, $M_i \in [1, C]$
$\boldsymbol{D}()$	Standard deviation function

of task grouping is to find the mapping f: $TaskSet \rightarrow TGroup[1{:}\,G]$ according to (1).

$$object: \min_f D(LocTGroup_1, LocTGroup_2, \cdots, LocTGroup_G)$$
$$subject\ \ to:$$
$$\begin{cases} T = \sum_{i=1}^{G} M_i \\ \min_f D(M_1, M_2, \cdots, M_G) \\ \max_f (M_i = C) \end{cases} \quad (1)$$

The goal is to minimize the variance between $LocTGroup_i$ under the constraints of load balance of groups and as many M_i as possible equal to C, making full use of computing cores for better execution time. After the static procedure, a sequence of sorted task groups is generated for the task scheduling. We propose a two-level task scheduling strategy that dynamically schedules groups to processors and timely distributes tasks to available cores. **The problem of the scheduling strategy is to provide an efficient mapping between groups and processors and prioritize task execution**, improving the utilization of computing cores and reducing the impact of inter-core interference between groups when scheduling tasks from different groups to run on the same processor.

IV. Shared Cache Contention-Aware Task Grouping

A. Profile of Shared Cache Locality

The contention between tasks for shared cache cannot be directly quantified. But according to the principle of locality, the cache locality of tasks can be used to reflect the contention for LLC. There are various locality metrics, such as miss ratio, inter-miss time, volume fill time, reuse distance, memory footprint, and etc. Each of these metrics is useful in different profiling of a program accesses to construct the cache miss rate curve [49]. Memory footprint describes the distinct data amount of a program accesses in a given length of window. It shows the memory occupation and is hardware independent. When a group of tasks are concurrently executing, the total accessed data amount is the sum of the memory footprint of each task running individually if there is no data sharing. For MTC applications, the parallel running tasks are generally independent. This feature of composability of memory footprint [50] can be used to profile shared cache locality. We estimate the shared cache locality of co-running tasks by using memory footprint. In fact, the mapping of tasks in LLC also has an impact on cache miss rate. But it needs detailed profiling of cache behavior to design sophisticated cache bank mapping policies [51], which usually requires extra hardware overhead and increases execution time overhead. In our method, we do not consider the impact of cache mapping for practicality.

According to the definition of memory footprint [50], its calculation is based on a time window. Given a task T_i with a length-n data access trace, its average footprint $fp(T_i)$ is the average footprint size in all windows of length l, given by Equation (2). The fp_w is the footprint of each window w, and $n - l + 1$ is the total number of length-l windows in the length-n trace. For example, there are 3 windows of length 2 for a task trace of "baab", i.e., "ba", "aa", and "ab". The footprint size of each window is 2, 1, and 2, respectively. And the average footprint of this trace is $fp(2) = (2 + 1 + 2)/3 = 5/3$.

$$fp(T_i) = fp(l) = \frac{\sum_{\text{all } w \text{ of length } l} fp_w}{n - l + 1} \tag{2}$$

Since the proposed task grouping method focuses on LLC, we use the equational cache space, i.e., the LLC capacity divided by C, as the window length l. The l is the window size in terms of data and is defined by Equation (3). The function *sizeof*(DataType) calculates the byte size of data. Combining Equation (2) and (3), we can calculate the cache locality of single task with its access trace.

$$l = \frac{LLCSize}{sizeof(\text{DataType}) \times C} \tag{3}$$

Due to the composability, the footprint of a group of tasks, i.e., the amount of data accessed by co-running tasks in LLC, is the sum of the footprint of each task. For a task group $TGroup[i]$, the combined footprint is defined by Equation (4), which is defined as $LocTGroup_i$.

$$LocTGroup_i = fp(TGroup[i]) = \sum_{m=1}^{Mi} fp(T_m) \tag{4}$$

In this work, the access trace of task is obtained by the pin tool [52], which involves a preliminary run of task. Since the computing kernel of a compute-intensive task is generally the nested-loop structure with regular access pattern, we sample a small portion of access trace over several continuous loop iterations to estimate the cache locality and the execution time of task by scaling up the sampling results. Although this sampling and estimation method only get the approximate footprint and execution time of tasks, these approximate values are sufficient for our task grouping and task scheduling methods. This is because they will be used to sort all tasks and guide our methods to group and schedule tasks according to the order (see following sections). As a result, it does not have extremely strict requirements for numerical accuracy. More importantly, it can greatly reduce the pre-run overhead of tasks. And the calculation of footprint based the sampled trace is done off-line, and thus it almost has no effect on the following task grouping.

B. Basic Grouping Process

We design a heuristic-based method to achieve task grouping. Its primary principle is to combine tasks with higher footprints and tasks with lower footprints into approximately equal groups to balance shared cache locality. We first substantialize the straightforward and effective method by illustrating an example of a basic case that the number of tasks equals the number of cores in a computing node.

Basic case. Given a set of T tasks $\{Task_1, Task_2, \ldots, Task_T\}$, and all tasks are sorted in descending order by footprint. Here $T = P \times C$ and $G = P$. We separately put the top G tasks of the task set into $TGroups[1{:}G]$ in the same order. Then, we separately put the last G tasks into task groups in an inverted order. We can get that $TGroups[1]=\{Task_1, Task_T\}$, $TGroups[2] = \{Task_2, Task_{T\text{-}1}\}$, $\ldots$, $TGroups[G]=\{Task_G, Task_{T\text{-}G\text{-}1}\}$. Next, we calculate the shared cache locality of each group according to Equation (4) and re-sort the task groups in descending order by $LocTGroup[i]$. In the same manner, we repeat the processes of re-grouping, shared cache locality calculation, and re-sorting groups until the task set is empty.

This basic grouping procedure can be described as a basic function *grouping_procedure*(), and it is used to implement the general task grouping algorithm.

C. Task Grouping Algorithm

Based on the grouping procedure, our method can be extended to a more general case that there is no limit to T.

General case. When $T < P \times C$, we bring in $(P \times C - T)$ empty tasks to perform the function *grouping_procedure*(), and the footprint of each empty task is set to 0. Once the task grouping is done, all empty tasks are removed to get the final task groups. If $T \leq P$, we just let each single task run on a processor without performing the grouping procedure.

When $T > P \times C$, the grouping procedure is somewhat different from that of the basic case. Here $G > P$ and $M_i \leq C$. If T is a multiple of $P \times C$, then $G = T/C$, and we can directly perform the *grouping_procedure*() function to obtain the task groups. However, this case is not common. We need to perform another

Algorithm 1 Task grouping

```
Input: $Task_1, Task_2, \ldots, Task_T$
  $fp_1, fp_2, \ldots, fp_T$ /*footprints of tasks*/
  $P$, $C$ /*# of processors and # of cores in a processor*/
Output: $TGroup[1:G]$ /*task groups*/
1: Initialization of $TGroup[1:G]$;
2: Sorting all tasks in descending order by the footprints to
   generate a task set: $Task_1, Task_2, \ldots, Task_T$;
3: if $T \le P$ then /*when # of tasks is less than or equals #
   of processors*/
4:   $G = T$;
5:   for $i = 1 : G$ do
6:     $TGroup[i] \leftarrow Task_i$;
7:   end for
8: end if
9: if $P < T \le P \times C$ then /*when # of tasks is less than or
   equals # of cores*/
10:  $G = P$;
11:  Adding $(P \times C - T)$ empty tasks at the end of the task set
     and set the footprint of each empty task to 0;
12:  grouping_procedure $(Task_1, Task_2, \ldots, Task_{P \times C})$;
13: end if
14: if $T > P \times C$ then /*when # of tasks is greater than # of
    cores*/
15:   $k = \text{floor}(T/(P \times C)) \times (P \times C)$;
16:   Dividing the task set into two parts:
      $P_1 = \{Task_1, \ldots, Task_{T-k}\}, P_2 = \{Task_{T-k+1}, \ldots, Task_T\}$;
      /*grouping tasks in $P_1$ */
17:   $G_1 = P$; /*# of task groups of $P_1$ */
18:   Adding $(P \times C - (T - k))$ empty tasks at the end of $P_1$ set
      and set the footprint of each empty task to 0;
19:   $TGroup_1[1 : G_1]$=grouping_procedure $(P_1)$;
      /*grouping tasks in $P_2$*/
20:   $G_2 = \text{floor}(T/(P \times C)) \times P$; /*# of task groups of $P_2$ */
21:   $TGroup_2[1:G_2]$=grouping_procedure $(P_2)$;
      //combining two parts of grouped tasks
22:   $G = G_1 + G_2$; /*# of all task groups*/
23:   $TGroup[1:G] \leftarrow TGroup_1 + TGroup_2$;
24: end if
25: Removing all empty tasks and getting the final $TGroup[1:G]$;
26: return $TGroup[1:G]$;
```

grouping procedure based on the basic one. First, we divide the initial task set into two parts, the last $k=\lfloor T/(P \times C)\rfloor \times (P \times C)$ tasks in the sorted task set form one part, while the remaining top $(T - k)$ tasks form the other part. For the larger part of k tasks, we use the *grouping_procedure*() function to group the tasks, here $G=\lfloor T/(P \times C)\rfloor \times P$. For the smaller part of tasks, the number of tasks is less than the number of cores, we add $(P \times C - (T - k))$ empty tasks to perform the *grouping_procedure*() function. At last, we combine the groups generated from the two parts together and remove all empty tasks to get final task groups.

The implementation of the task grouping method is described in Algorithm 1. Since the footprint of individual task is calculated beforehand, the cost of Equation (4) is trivial when calculating the *LocTGroup*[i] of re-combined groups. The computational complexity of the task grouping algorithm is O($n\log_2 n$), where n is the number of tasks.

To meet the constraints of minimizing $\boldsymbol{D}(M)$ and offering as many M_i as possible equal to C in **Problem 1**, the partition of task sequence at the "k point" makes a trade-off between load balance and cores' utilization. For example, on the SMP node in Fig. 1, 13 tasks can be divided into groups of {4 tasks}, {3 tasks}, {3 tasks}, and {3 tasks} for the best load balance, i.e., minimum $\boldsymbol{D}(M)$, while the groups of {4 tasks}, {4 tasks}, {4 tasks}, and {1 task} achieve the maximum $M_i = 4$. Our grouping method generates groups of {4 tasks}, {4 tasks}, {3 tasks}, and {2 tasks}, which best meet the requirements of **Problem 1**. Additionally, within each group, co-running tasks are relatively friendly for shared cache to alleviate the contention and therefore, are able to benefit from shared cache locality.

V. Two-Level Task Scheduling Strategy

A. First Level Scheduling of Groups

Task scheduling is the key to benefit from the grouping method and further improve the execution performance. Since tasks within a group cannot be completed simultaneously, waiting tasks need to be assigned to available cores as soon as possible to ensure high computing throughput that no resources are wasted. Thus, there will be a reassociation of tasks from two different groups during the scheduling process. It needs to carefully consider the effect of task reassociation on the shared cache locality. We propose a two-level task scheduling strategy based on the classic longest processing time (LPT) rule [53].

The first level scheduling strategy determines the execution order of groups to preserve shared cache locality and load balancing. A group which contains C tasks is considered a full group, otherwise a partial group. Full groups are scheduled with priority over partial groups, while the LPT groups have higher scheduling priority in each type of groups. To preserve the shared cache locality of task grouping, tasks of each group are mapped to the cores of same processor for execution.

There are two reasons supporting the scheduling strategy of groups. First, full groups first scheduled can preserve the shared cache locality of task groups. If a partial group is first assigned to a processor, some cores are idle. According to the principle of making full use of computing cores, the scheduling strategy will assign tasks to the same processor immediately. By this point, the shared cache locality of the partial group is broken and the efforts of task grouping fail. Although it is unavoidable to reassociate tasks in the subsequent scheduling, the impact can be as less as possible by our second level scheduling strategy (described in Section V-B).

Secondly, giving the LPT groups priority to schedule is in the favor of a shorter execution time. The LPT rule has been derived as an approximate optimal solution for task scheduling on multi-machine [53]. It can make the last task finish earlier. As shown in Fig. 2, four task groups are going to run on two identical processors. The blank strips represent task groups, and the length of the blank strip represents the execution time of the task group.

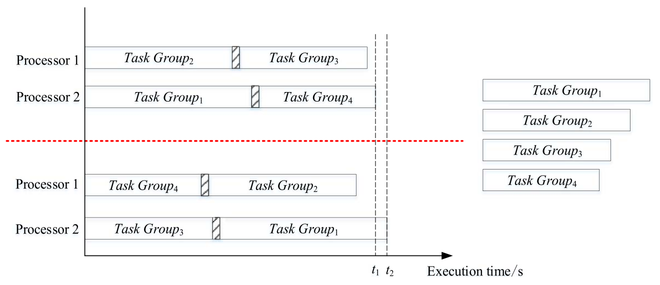


Fig. 2. Execution time of different scheduling strategies for groups.

The small shaded strips represent the process of scheduling. Assume that same task groups run for same time and the scheduling time is the same. The LPT-based scheduling, shown on the upside of the red dotted line in Fig. 2, completes the execution of all groups at t_1, while an opposite shortest processing time (SPT)-based scheduling strategy finishes the same task groups at t_2, shown on the downside of the red dotted line. When there are more groups, the difference between t_1 and t_2 is larger, and the advantages of our strategy are also more significant.

The execution time of task groups is not known before real execution. But the execution time of tasks is estimated in advance according to the pre-run. In fact, due to the contention, the execution time of a task becomes longer when it co-runs with other tasks compared to the execution time of its single run. However, it is not important to get the accurate execution time, what matters for our scheduling strategy is the order of the length of the task execution time. To avoid extra overhead for another run of task groups, we made a trade-off that uses the longest execution time of single task in a group as the group execution time and uses the single run time of the task as its co-running time.

B. Second Level Scheduling of Task

The second level scheduling strategy identifies which task in a waiting group should be first assigned to available cores during the scheduling process. This strategy aims to make better use of computing cores, while minimizing the impact of task reassociation.

This scheduling strategy gives priority to the LPT task in a group. It can manage computing cores more reasonable, thus shortening the execution time. The reason is similar to the second argument supported for the first level scheduling strategy. For example, two same groups of 8 tasks are separately running on two identical quad-core processors, as shown in Fig. 3. Assume that same task runs for same time. As a result, our scheduling strategy completes all tasks at t_2 on Processor 1, while the opposite SPT-based scheduling strategy finishes same tasks at t_3 on Processor 2.

The second level scheduling strategy also makes a trade-off between computing throughput and performance fluctuation caused by task reassociation. As shown in Fig. 3, during the time slot from t_1 to t_1', $Task_8$, $Task_7$ and $Task_6$ will gradually regroup with $Task_2$, $Task_3$ and $Task_4$. To preserve the shared cache locality and reduce the impact of reassociating new tasks, the time slot of task reassociation should be as short as possible,

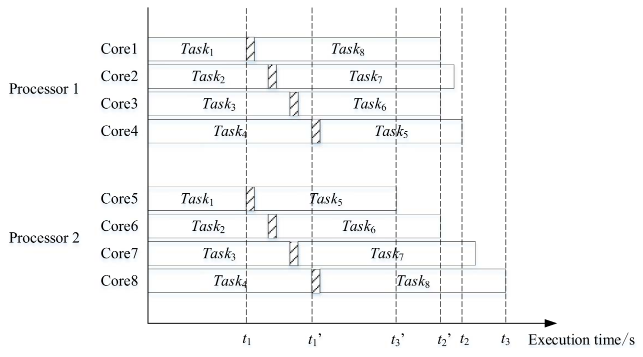


Fig. 3. Execution time of different scheduling strategies for tasks.

while the time slot of concurrent task execution in the original group should be as long as possible. The time slot from t_1' to t_2', i.e., concurrent task execution in original group with LPT-based strategy, is longer than the time slot from t_1' to t_3' with SPT-based strategy. Meanwhile, the time slot from t_2' to t_2, i.e., subsequent task reassociation with LPT-based strategy, is shorter than the time slot from t_3' to t_3 with SPT-based strategy.

Besides, the time slot of task reassociation is relatively short compared with that of concurrent task execution in original group. In fact, through extensive experimental observation and verification, we found that making full use of computing cores with task reassociation yields greater performance gains than only maintaining shared cache locality of task groups without task reassociation.

C. Implementation of Task Scheduling Strategy

Assume that all P processors are available at the beginning and task groups $TGroup[1{:}G]$ are ready. We need to set P empty buffers, $TBuffer[1{:}P]$, to store the mapping tasks for each processor. First, the top P task groups, $TGroup[1{:}P]$, are separately assigned to each processor at random. Tasks of each group are bound to cores of a processor to simultaneously run. Once a computing core has finished its current task, the buffer of related processor will be checked. If the buffer is empty, the first group in the waiting group set will be stored in the buffer and the first task of this group will be assigned to the core for execution. Otherwise, the first task in the buffer is assigned to the core. The process is repeated until there are no tasks in the group set or buffers. The detailed implementation of the task scheduling strategy is described in Algorithm 2. The computational complexity of the task scheduling algorithm is the maximum of $O(nm\log_2 m)$ and $O(n\log_2 n)$, where n is the number of groups and m is the number of tasks in a group.

D. Discussion

Scalpel targets compute-intensive tasks of MTC applications. Theoretically, it can also be applied to other loosely coupled applications composed of many tasks.

The proposed approach uses off-line analysis to calculate the locality and the execution time by a pre-run of tasks. The used sampling and estimation method is effective for compute-intensive tasks with regular access pattern, specifically for the

Algorithm 2 Two-level task scheduling

```
Input: $TGroup[1:G]$ /*task groups*/
       $P$ /*# of processors*/
Output: task scheduling scheme
1: if $G \le P$ then /*special case that # of groups is not greater than # of processors*/
2:    Assigning $TGroup[1:G]$ to processors separately for execution; /*no scheduling*/
3: else
4:    for each task group $TGroup[1:G]$ do
5:       Sorting tasks in descending order by the execution time;
6:       Setting the execution time of longest running task as the execution time of the group;
7:    end for
8:    Sorting task groups in descending order by the execution time of groups while following the full groups priority to get the ordered task group set $TGroup'[1:G]$;
9:    Initialization of $TBuffer[1:P]$; /*setting $P$ empty task buffers for processors*/
10:   Assigning $TGroup'[1:P]$ to $P$ processors separately for execution;
11:   while($TGroup'[1:G]$ or $TBuffer[1:P]$ is not empty) do
12:      if $Core_i$ is available then
13:         Checking that $Core_i$ is located in $Processor_j$;
14:         if $TBuffer[j]$ is empty then
15:            Assigning the first task in the first group of $TGroup'[1:G]$ to $Core_i$ for execution;
16:            Storing the rest tasks in the same group into $TBuffer[j]$ in the same order;
17:         end if
18:         else /*there are tasks in the buffer*/
19:            Assigning the first task in $TBuffer[j]$ to $Core_i$ for execution;
20:      end if
21:   end while
22: end if
```

computing kernel of loop structures. For some scientific computing applications and specific computing systems [54], the types of subroutines and tasks are relatively fixed, and the pre-run of tasks can be done once and for all. For some scenarios, tasks are not compute-intensive and have irregular access pattern. The used off-line analysis method may be no more suitable. To apply Scalpel for such scenarios and make it more automated, the pre-run analysis method can be substituted by on-line analysis and random sampling and prediction methods.

To clarify the core idea of Scalpel, we assume that tasks have same execution priority in this paper. In practice, tasks of different applications, and even tasks of the same application, may have different priorities due to dependency or completion time constraints, such as for real-time systems. For such scenarios, this approach can be applied separately to tasks with the same priority or similar completion time constraints, and prioritizes higher priority tasks or completion time-critical tasks for scheduling. By drawing on the essence of our approach to reduce shared cache contention, we believe that such applications can further improve the performance on computing nodes and ensure the real-time requirements for tasks.

Besides, to verify the effectiveness of Scalpel, we only target the scenario of static tasks in this paper. However, this approach can also be applied to the scenario that new tasks are constantly arriving at computing nodes. The grouping method and further scheduling strategy for the new arriving tasks are basically the same. But under this circumstance, we should lay emphasis on when to group new tasks, whether the waiting tasks should be re-grouped with new tasks, or which waiting tasks should be re-grouped.

All these concerns can help Scalpel expanding application scenarios and enhancing the practicability, but they go beyond the scope of this work and will be investigated in our future work.

VI. Experimental Results

In this section, we design various sets of experiments to evaluate Scalpel. Since our work lays emphasis on fine-grained task scheduling on the computing nodes, we assume a good inter-node level scheduling approach has distributed the tasks of applications to the computing nodes. We focus on evaluating the scheduling of tasks which are mapped to a node for concurrent execution.

A. Experiment Settings

All experiments were performed on an Intel Xeon server. The server has two Intel Xeon E5620 processors at 2.40 GHz. Each processor integrates 4 cores and has a three-level cache hierarchy that each core has a 64KB private L1 cache and a 256KB private L2 data cache, and four cores share a 12MB L3 data cache. The operating system is CentOS 7.6.1810.

We selected 23 benchmarks from Polybench [17]. The benchmarks are kernels extracted from computational operations in various domains, such as linear algebraic calculation, image processing, physical simulation, dynamic programming, and statistical information. These kernels, as compute-intensive tasks, are widely used in scientific computing applications. We chose different problem sizes for the benchmarks to make them run under 100 seconds. Table II shows the benchmarks and their individual execution time and calculated footprints.

To evaluate Scalpel, we use the FIFO (First-In, First-Out) and RR (Round Robin) scheduling policies [55] which are provided by Linux scheduler, two state-of-the-art cache-aware task scheduling algorithms called DI (Distributed Intensity) [48] and IA (Iteration Algorithm) [39], and a cache partitioned scheduling algorithm SF3-4K [30] to carry out comparison experiments.

The FIFO scheduling policy assigns a core to the task at the top of run queue list for execution if no other higher-priority task is runnable, even if other same-priority tasks are runnable. The RR scheduling policy assigns a core to the task for execution until running out of time slices, then the scheduler will reallocate time slices and put it at the end of execution queue list. The RR policy ensures a fair assignment of CPU time to all processes that have same priority. The DI algorithm targets process-level tasks that have same priority and no shared data. It tries to reduce

TABLE II
BENCHMARKS AND INDIVIDUAL RUNNING PARAMETERS

Benchmarks	Execution Time (s)	Footprints
adi	41.47	423390.47
apop	39.34	2757.30
corcol	37.05	298517.59
covcol	62.61	299600.31
doitgen	43.53	253464.25
dsyr2k	35.65	418249.56
dsyrk	26.07	512879.16
fdtd-1d	58.09	41532.93
fdtd-2d	47.64	524701.13
trisolv	92.66	228854.97
game-of-life	27.75	187982.25
heat-1d	22.06	14075.23
heat-2d	42.06	114258.60
matmul	26.26	350203.13
tmm	66.44	524286.97
strsm	31.24	159474.95
ssymm	55.59	521052.78
floyd- warshall	62.90	317657.13
jacobi-2d	48.51	530276.44
nussinov	22.10	317490.28
seidel-2d	35.99	530390.13
2mm	46.02	315584.22
3mm	30.36	315604.09

shared cache miss rate by forming tasks with high cache miss rate and tasks with low cache miss rate into task groups to ensure cache miss is evenly distributed among groups. The IA uses the non-cooperative game theory to present a fairness-aware task co-scheduling algorithm for reducing shared cache contention. It obtains the best scheduling scheme by solving the Nash equilibrium. While the SF3-4K employs the cache partitioning technology to address the shared cache contention. It first groups tasks with similar amount of core stalls due to LLC accesses into clusters. And then, each cluster will be allocated a number of ways using a linear mathematic model. These algorithms have been proved to be very effective in solving the problem of performance degradation caused by shared cache contention.

In the experiments, to exclude the effect of intra-core interference, we set CPU affinity to ensure that each task monopolizes a computing core during its execution.

B. Overall Performance

First, we evaluate the overall performance of Scalpel. Different types and numbers of tasks are selected from Table II to verify the performance. Tasks of different sets were randomly picked from the benchmark suit for fairness. We use the execution time and L3 cache miss rate to evaluate the performance of co-running tasks. Execution time is the duration between the start of execution and the completion of the last task in a set of co-running tasks. It includes the overhead of scheduling. The L3 cache miss rate is obtained by the Linux performance profiling tool *perf* to reflect the intensity of shared cache contention between co-running tasks. We also provide the L3 cache MPKI rate to evaluate the shared cache contention. The cost of scheduling is trivial compared to the total execution time.

Fig. 4 shows the performance results delivered by FIFO, RR, and Scalpel on different sets of co-running tasks. The captions

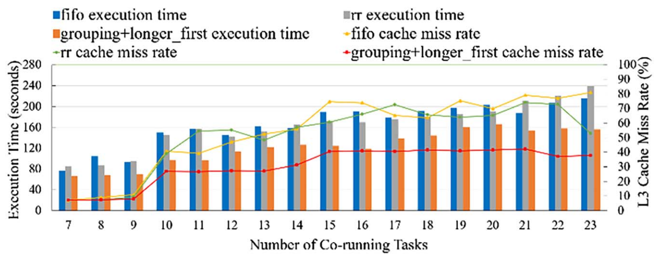


Fig. 4. Performance Results of Scalpel Compared with FIFO and RR.

with "fifo", "rr", and "grouping+longer_first" refer to FIFO, RR, and Scalpel, respectively. When the number of tasks is greater than 9, the competition among tasks is increasingly fierce and the L3 cache miss rates under all scheduling approaches are steeply raising. In these cases, Scalpel still offers a significant performance improvement. For example, the execution time is 156.31 seconds when applying Scalpel, while FIFO and RR take about 1.38× (215.30 seconds) and 1.53× (239.01 seconds) longer, respectively, for the set of 23 tasks. And Scalpel reaches a L3 cache miss rate at 37.80% for the same set, which is better than FIFO and RR by about 2.14× (80.95%) and 1.40× (52.88%), respectively. Scalpel improves the execution time of co-running tasks respectively by 25.86% and 24.97% on average over FIFO and RR scheduling methods for all test sets. And the average L3 cache miss rate has fallen by over 20%, e.g., from 54.26% with FIFO, and 51.32% with RR, to 30.87% with Scalpel. The performance improvement of Scalpel is attributed to the decline in shared cache miss rate and the full use of computing cores. We will specifically verify the contributions of grouping method and scheduling strategy on performance improvement in Section VI-C and VI-D.

Since it is infeasible to carry out all possible experiments on a given number of tasks, we used different tasks to conduct several experiments. Although the experimental results are little different, the overall performance has shown the same trend as in Fig. 4 that Scalpel offers distinct advantages on both execution time and L3 cache miss rate. Due to the length limitation, we do not present the specific results here.

We compared Scalpel with other two state-of-the-art cache-aware task scheduling algorithms DI [48] and IA [39]. For the 8-core experiment node, we respectively selected 8, 11, 15 and 16 tasks from Table II to evaluate the performance of these methods. For the given number of tasks, we randomly selected different sets of tasks. Table III shows the comparison results. Since the DI is limited to that the number of tasks equals the number of cores, we only carried out the evaluation of DI with 8 tasks.

For the sets of 8 tasks, three methods show very close results. The DI reaches the highest L3 cache miss rate and a slightly worse execution time than Scalpel in the experiments. The idea behind DI is similar to ours that divides tasks into equalized groups in terms of shared cache locality. However, Scalpel uses the footprint of each task rather than the cache miss rate (employed by the DI) to evaluate the shared cache locality. Footprint has been proven to accurately quantify the locality of concurrent tasks, while shared cache miss rate cannot be directly combined by the individuals as the competition between tasks may impact cache behavior. In addition, DI does not provide a

TABLE III
PERFORMANCE COMPARED WITH TWO SHARED CACHE-AWARE TASK SCHEDULING METHODS

# of tasks	Methods	Grouping Schemes	Execution Time (s)	L3 Cache miss Rate (%)
8	DI [48]	{dsyrk, covcol, corcol, fdtd-1d}, {dsyr2k, adi, apop, doitgen}	65.49	8.20
	IA [39]	{dsyr2k, adi, doitgen, fdtd-1d}, {dsyrk, covcol, apop, corcol}	65.13	7.95
	Scalpel	{covcol, doitgen, apop, dsyrk}, {fdtd-1d, adi, corcol, dsyr2k}	64.72	8.03
8	DI [48]	{dsyr2k, adi, heat-2d, game-of-life}, {trisolv, dsyrk, covcol, heat-1d}	93.99	54.65
	IA [39]	{dsyr2k, adi, trisolv, covcol, heat-1d}, {dsyrk, heat-2d, game-of-life}	95.89	51.09
	Scalpel	{dsyr2k, trisolv, heat-2d, heat-1d}, {adi, dsyrk, game-of-life, covcol}	93.62	51.52
11	DI [48]	—	—	—
	IA [39]	{fdtd-2d, adi, covcol, corcol, apop}, {dsyr2k, trisolv, heat-1d, doitgen}, {game-of-life, fdtd-1d}	151.89	26.93
	Scalpel	{trisolv, covcol, apop, game-of-life}, {fdtd-1d, doitgen, corcol, heat-1d}, {adi, dsyr2k}, {fdtd-2d}	97.52	28.49
11	DI [48]	—	—	—
	IA [39]	{tmm, fdtd-2d, heat-2d, covcol, apop}, {strsm, dsyr2k, trisolv, dsyrk}, {adi, corcol}	140.31	49.47
	Scalpel	{trisolv, covcol, adi, apop}, {heat-2d, corcol, dsyr2k, dsyrk} {fdtd-2d, strsm}, {matmul}	112.43	50.76
11	DI [48]	—	—	—
	IA [39]	{matmul, dsyrk, heat-2d}, {tmm, strsm, dsyr2k, covcol}, {fdtd-2d, adi, trisolv, game-of-life}	143.45	63.51
	Scalpel	{covcol, fdtd-2d, adi, dsyrk}, {trisolv, heat-2d, dsyr2k, game-of-life}, {tmm, strsm}, {matmul}	101.28	66.01
11	DI [48]	—	—	—
	IA [39]	{strsm, dsyr2k, dsyrk, covcol, heat-1d}, {fdtd-2d, adi, heat-2d, game-of-life}, {trisolv, apop}	143.70	48.87
	Scalpel	{trisolv, covcol, dsyrk, heat-1d}, {heat-2d, adi, apop, game-of-life}, {fdtd-2d, dsyr2k}, {strsm}	100.24	51.39
11	DI [48]	—	—	—
	IA [39]	{matmul, fdtd-2d, game-of-life, heat-1d, apop}, {tmm, dsyr2k, dsyrk, heat-2d}, {trisolv, corcol}	166.39	56.80
	Scalpel	{trisolv, apop, dsyrk, heat-1d}, {heat-2d, corcol, dsyr2k, game-of-life}, {tmm, fdtd-2d}, {matmul}	100.86	60.05
15	DI [48]	—	—	—
	IA [39]	{matmul, fdtd-2d, heat-2d}, {adi, jacobi-2d, seidel-2d}, {tmm, dsyrk, heat-1d, 3mm}, {dsyr2k, trisolv, apop, corcol, doitgen}	145.18	44.07
	Scalpel	{jacobi-2d, heat-2d, corcol, 3mm}, {doitgen, apop, seidel-2d, heat-1d}, {trisolv, tmm, dsyr2k, dsyrk}, {fdtd-2d, adi, matmul}	122.85	45.40
15	DI [48]	—	—	—
	IA [39]	{matmul, strsm, dsyrk}, {dsyr2k, heat-2d, jacobi-2d}, {2mm, trisolv, seidel-2d, covcol}, {fdtd-2d, adi, game-of-life, heat-1d, apop}	147.19	67.20
	Scalpel	{covcol, heat-2d, seidel-2d, heat-1d}, {jacobi-2d, apop, game-of-life, dsyrk}, {trisolv, tmm, fdtd-2d, adi}, {dsyr2k, strsm, matmul}	125.52	69.20
16	DI [48]	—	—	—
	IA [39]	{matmul, strsm, dsyrk}, {dsyr2k, heat-2d, game-of-life}, {tmm, trisolv, covcol, apop, fdtd-1d}, {fdtd-2d, adi, heat-1d, corcol, doitgen}	146.33	41.13
	Scalpel	{tmm, covcol, doitgen, dsyr2k}, {trisolv, heat-2d, corcol, strsm}, {fdtd-1d, adi, matmul, heat-1d}, {fdtd-2d, apop, game-of-life, dsyr2k}	103.38	42.07
16	DI [48]	—	—	—
	IA [39]	{matmul, strsm, dsyrk}, {dsyr2k, heat-2d, jacobi-2d}, {tmm, trisolv, seidel-2d, covcol, 3mm}, {fdtd-2d, adi, game-of-life, heat-1d, apop}	160.36	66.73
	Scalpel	{trisolv, tmm, jacobi-2d, apop}, {covcol, fdtd-2d, seidel-2d, dsyr2k}, {heat-2d, adi, strsm, heat-1d}, {3mm, game-of-life, matmul, dsyrk}	109.30	68.00

grouping strategy for any number of tasks, nor does it provide a scheduling strategy of co-running tasks.

For all sets of tasks, the IA delivers the best L3 cache miss rate. Because it uses the non-cooperative game to model the competition relationship between tasks, and each task decides to update its current strategy for a higher utility in an iterative process. The well-designed algorithm is proved to reach the Nash equilibrium after a certain number of iterations and achieves the goal of fairly scheduling threads and minimizing the total misses of shared cache. Whereas, Scalpel is much more straightforward, and it divides tasks into groups in a heuristic way according to an approximate estimation of shared cache locality. But the L3 cache miss rated achieved by Scalpel is very close to that of IA. More importantly, Scalpel shows an apparent advantage in the execution

TABLE IV
PERFORMANCE COMPARISON ON L3 CACHE MPKI RATE

# of tasks	Methods	MPKI Rate
15	FIFO	0.356
	RR	0.324
	IA	0.315
	Scapel	0.274
16	FIFO	0.289
	RR	0.313
	IA	0.281
	Scapel	0.252

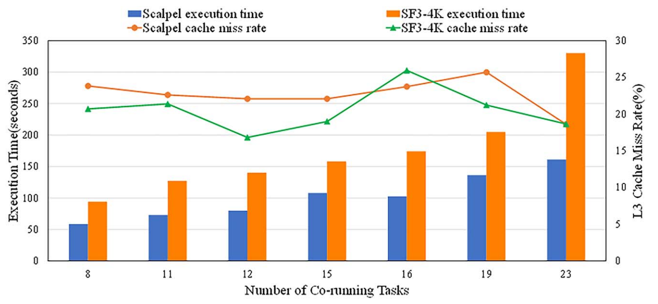


Fig. 5. Performance results of Scalpel compared with SF3-4F.

time, particularly, for the sets of 11, 15, and 16 tasks. To run these sets of tasks, Scalpel is faster than the IA by over 27% on average. This is because the IA only concerns how to select the most balanced groups of tasks, while Scalpel also considers the load balancing and computing cores' utilization. Moreover, the IA does not allow tasks in different groups to run on the same processor simultaneously to broke the Nash equilibrium of the scheduled tasks with respect to the shared cache contention. And thus, it will lead to a huge waste of computing cores that tasks must wait for the current executing group to complete, and many available cores have to be idle.

When we broke the rule of IA algorithm that allows tasks from different IA groups to run on the same processor simultaneously, it leads to more conflicts on shared cache. For example, we used a set of 15 tasks in Table III to conduct experiment, IA increases the L3 cache miss rate by 7.55%, but shortens the execution time that it is close to ours. On the contrary, Scalpel distributes appropriate tasks on cores timely, while reducing the interference between tasks from different groups. Therefore, Scalpel achieves a better performance on execution time. This result also confirms that the impact of task reassociation on shared cache locality can be covered by the performance improvement brought by full utilization of computing cores.

To further substantiate the effect of Scalpel on mitigating shared cache contention, we evaluated the L3 cache MPKI rates of Scalpel, IA, FIFO, and RR using the sets of 15 and 16 tasks from Table III. Table IV shows the comparison results. Scalpel achieves the lowest MPKI rate compared to other methods. We can find that IA yields the second best MPKI rate. Although IA obtains the best L3 cache miss rate in Table III, the MPKI performance demonstrates that Scalpel can best minimize the L3 cache contention between co-running tasks.

We also carried out comparison with a cache partitioned scheduling algorithm SF3-4F [30]. Fig. 5 shows the comparison results of

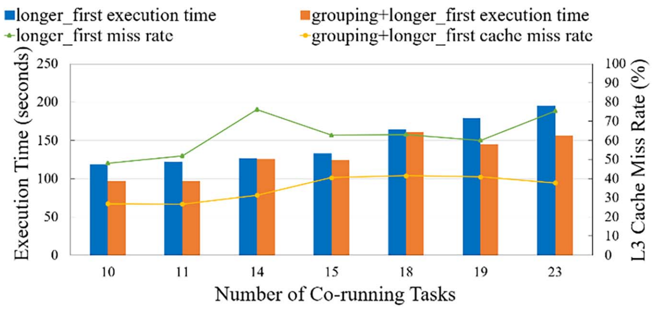


Fig. 6. Performance results when task grouping is disabled.

Scalpel and SF3-4F. In terms of L3 cache miss rate, two algorithms deliver comparative results, i.e., SF3-4F shows an average L3 cache miss rate of 20.53% across all the experiment sets, and Scalpel achieves 22.66%. According to [30], SF3-4F can achieve state-of-the-art system fairness of cache contention without harming performance, the comparison results demonstrate the effectiveness of Scalpel on minimizing shared cache contention between co-running tasks. Moreover, Scalpel outperforms SF3-4F across the board in terms of task execution time. The average execution time of all task sets achieved by Scalpel is 102.83 seconds, compared to 175.49 seconds for SF3-4F, showing an improvement of 41.40%.

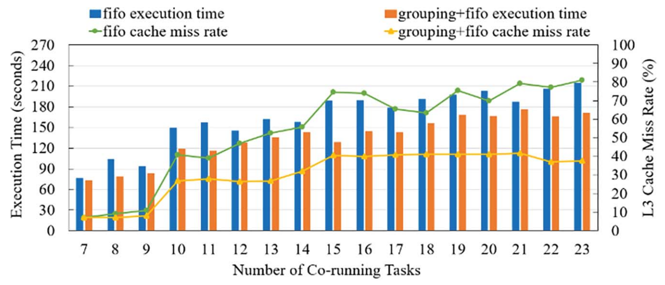
Scalpel has also been evaluated on other benchmark suits, including SPEC2017 [56], NAS Parallel Benchmark [57], and SPLASH2 [58], which includes compute-intensive kernels, memory-intensive kernels, and mixed kernels. Since memory-intensive kernels primarily compete for I/O bandwidth, Scapel's performance on them is not as significant as on compute-intensive kernels. However, our method consistently exhibits better performance than the compared methods across these benchmark suites.

C. Evaluation of Task Grouping

Scalpel consists of two important components, i.e., the shared cache contention-aware task grouping and the LPT-based two-level task scheduling. We separately evaluated the contribution of each component to the performance gains. The importance of the grouping method and its contribution to performance are first verified.

We designed three sets of experiments to show the effect of task grouping. In the first set of experiments, the task grouping is disabled and only the scheduling strategy is retained that the LPT tasks are given priority to scheduling on available cores. And this redesigned method, denoted by "longer_first", was evaluated over different numbers of tasks, comparing with Scalpel, i.e., "grouping+longer_first". Fig. 6 shows the evaluation results. The L3 cache miss rate is obviously raised when the task grouping is turned off and the execution time of tasks increased for all tested cases. This is because the pure LPT-based scheduling method is with no regard for the shared cache contention between co-running tasks, which has an important impact on the performance. When disabling the task grouping, there is an average performance loss of 14.79% on execution time, and the L3 cache miss rate has been averagely raised by 78.05%.

In the other two sets of experiments, we evaluated the effectiveness of the task grouping in a more direct way. We applied


(a) Comparison with FIFO

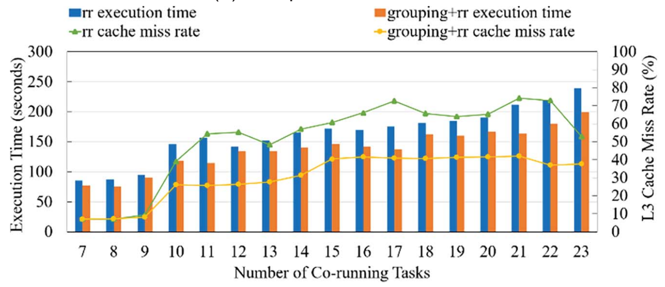
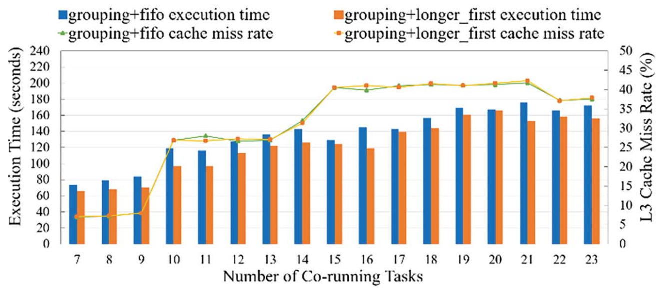

(b) Comparison with RR

Fig. 7. Performance results when applying task grouping to FIFO and RR.


(a) Comparison with “grouping + fifo”

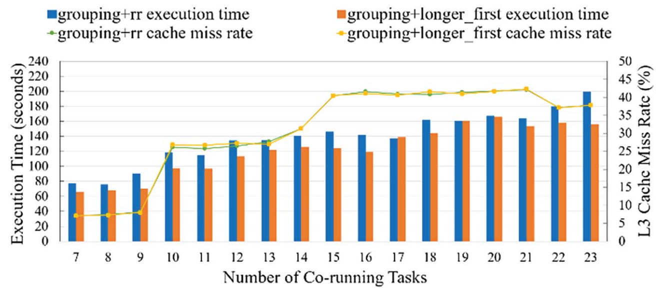

(b) Comparison with “grouping + rr”

Fig. 8. Performance comparison results for evaluating the task scheduling strategy.

the grouping method to the FIFO and RR scheduling that the task grouping is first performed and then corresponding scheduling strategies are used on the grouped tasks. The reimplemented FIFO and RR scheduling methods, respectively denoted by “grouping+fifo” and “grouping+rr”, were compared with the original FIFO and RR scheduling. The test results are shown in Fig. 7. The execution performance of both FIFO and RR scheduling is significantly improved. When the number of co-running tasks is greater than the number of cores, the contention for L3 cache resources is becoming more intense among competing tasks. And the positive effects of the task grouping are more outstanding. When applying the task grouping to FIFO, the average performance improvements of execution time and L3 cache miss rate are 21.93% and 76.05%, respectively. And for the RR, corresponding performance improvements achieve 18.27% and 66.57% on average. The experiment results have effectively substantiated the contributions of the task grouping method and its necessity to achieve a better performance of co-running tasks.

Furthermore, to verify how far is the task grouping from the optimum grouping of tasks, we also carried out another set of experiments. The brute-force enumeration approach was used to obtain the optimum combination. Due to costly time overhead, we only picked 8 tasks to execute totally 70 different combinations of tasks on the experiment node. The best combination of tasks ran 68.38 seconds with a L3 cache miss rate of 91.36%. And our grouping method reached a total execution time of 69.63 seconds and a L3 cache miss rate of 91.77%, which runs the tenth position in the best execution time of all combinations. However, it is very close to the performance of the optimal one. Besides, the time cost of our method is negligible compared to the time-consuming brute-force enumeration. Our method is more practical while the brute-force approach is infeasible when the number of tasks is large.

D. Evaluation of Two-Level Task Scheduling

We evaluated the contributions of the LPT-based two-level task scheduling strategy to performance improvement. The procedure of task grouping is retained and the scheduling strategy is altered to carry out the evaluation experiments. We chose the FIFO and RR scheduling as alternatives, denoted by “grouping+fifo/rr”, and we compared these two methods with Scalpel.

Fig. 8 shows the comparison results. Since all methods employed task grouping to mitigate shared cache contention, experiment results show almost the same L3 cache miss rate that the line charts of three methods are basically overlapped. But there are still differences in the execution time. Compared with FIFO, our LPT-based scheduling shortens the execution time by about 1 second to 23 seconds for all test cases, achieving an almost 10% performance improvement on average. Compared with RR, the execution time is averagely shortened by over 11%. The results have demonstrated that the proposed scheduling strategy can make full use of computing cores and further increase execution performance.

VII. CONCLUSION

In this work, we propose Scalpel, a high-performance shared cache contention-aware task co-scheduling approach, to reduce the inter-core interference and improve the execution performance of co-running tasks on computing nodes for scientific computing applications. We use the footprint of individual tasks to quantify the shared cache locality of co-running tasks. And based on this, tasks are divided into groups with equalized shared cache locality in a heuristic way to mitigate the contention for shared cache between competing tasks and ensure load balancing. Then, we propose a two-level scheduling strategy based on the LPT rule to schedule groups and tasks on processors and cores, respectively, so as to further improve the execution performance by making full use of computing cores. Experimental results demonstrate that Scalpel achieves a shorter execution time and a lower cache miss rate for different types and numbers of tasks, compared with five alternatives on multiprocessor node.

In the future work, we intend to employ online sampling-based locality analysis to enhance the automated capability of Scalpel, which also facilitates incorporating it into the operating system scheduler. Using the powerful artificial intelligence technique for task scheduling is another potential direction for future research.

References

[1] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proc. 1st Workshop Many-Task Comput.*, Grids Supercomputers, 2008, pp. 1–11.

[2] Y. Liu et al., "Closing the 'Quantum Supremacy' Gap: Achieving real-time simulation of a random quantum circuit using a new Sunway supercomputer," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Analysis (SC)*, 2021, no. 3, pp. 1–12.

[3] W. Jia et al., "Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)* no. 1, 2020, pp. 1–14.

[4] H. Y. Ryu, A. I. Titov, A. Hosaka, and H. C. Kim, "Φ photoproduction with coupled-channel effects," *Prog. Theor. Exp. Phys.*, vol. 2014, no. 2, 2014, Art. no. 023D03.

[5] A. Alhubaishy and A. Aljuhani, "A load-fairness prioritization-based matching technique for cloud task scheduling and resource allocation," *Comput. Syst. Sci. Eng.*, vol. 45, no. 3, pp. 2461–2481, 2023.

[6] E. Hwang, J. Kim, and Y. Choi, "Achieving fairness-aware two-level scheduling for heterogeneous distributed systems," *IEEE Trans. Serv. Comput.*, vol. 14, no. 3, pp. 639–653, May/Jun. 2021.

[7] A. Marahatta, S. Pirbhulal, F. Zhang, R. M. Parizi, K. R. Choo, and Z. Liu, "Classification-based and energy-efficient dynamic task scheduling scheme for virtualized cloud data center," *IEEE Trans. Cloud Comput.*, vol. 9, no. 4, pp. 1376–1390, Oct./Dec. 2021.

[8] R. Ghafari, F. H. Kabutarkhani, and N. Mansouri, "Task scheduling algorithms for energy optimization in cloud environment: A comprehensive review," *Clust. Comput.*, vol. 25, no. 2, pp. 1035–1093, 2022.

[9] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Evaluation of cache partitioning for hard real-time systems," in *Proc. 26th Euromicro Conf. Real-Time Syst.*," 2014, pp. 15–26.

[10] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 27–39, 2017.

[11] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely, Jr., and J. Emer, "The gradient-based cache partitioning algorithm," *ACM Trans. Archit. Code Optim.*, vol 8, no. 4, pp. 1–21, 2012.

[12] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, vol. 14, 2014, pp. 381–392.

[13] F. Mueller, "Compiler support for software-based cache partitioning," in *Proc. ACM SIGPLAN Workshop Lang., Compilers Tools Real-Time Syst.*, 1995, pp. 125–133.

[14] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell, "Resource-aware task scheduling," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 1–25, 2015.

[15] S. A. Rashid, M. A. Awan, P. F. Souto, K. Bletsas, and E. Tovar, "Cache-aware schedulability analysis of PREM compliant tasks," in *Proc. Design, Automation & Test Europe Conf. & Exhib. (DATE)*, 2022, pp. 1269–1274.

[16] H. N. Tran, F. Singhoff, S. Rubini, and J. Boukhobza, "Cache-aware real-time scheduling simulator: Implementation and return of experience," *ACM SIGBED Rev.*, vol. 13, no. 1, pp. 22–28, 2016.

[17] L. N. Pouchet, "Polybench/C: The polyhedral benchmark suite," 2012. Accessed: Jun. 15, 2022. [Online]. Available: https://web.cs.ucla.edu/~pouchet/software/polybench/

[18] Y. Wang, Y. Cui, P. Tao, H. Fan, Y. Chen, and Y. Shi, "Reducing shared cache contention by scheduling order adjustment on commodity multi-cores," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Ph.D. Forum* 2011, pp. 984–992.

[19] D. Hoornaert, S. Roozkhosh, R. Mancuso, and M. Caccamo, "Work in progress: Identifying unexpected inter-core interference induced by shared cache," in *Proc. IEEE 27th Real-Time Embed. Technol. Appl. Symp. (RTAS)*, 2021, pp. 517–520.

[20] S. Rodrigo, F. O. Sem-Jacobsen, H. Tatenguem, T. Skeie, and D. Bertozzi, "Cost-effective contention avoidance in a CMP with shared memory controllers," in *Proc. 18th Int. Conf. Parallel Process. (ICPP)*, 2012, pp. 741–752.

[21] S. A. Rashid, G. Nelissen, and E. Tovar, "Cache persistence-aware memory bus contention analysis for multicore systems," in *Proc. Design, Automation & Test Europe Conf. & Exhib. (DATE)*, 2020, pp. 442–447.

[22] T. Maqsood, K. Bilal, and S. A. Madani, "Congestion-aware core mapping for network-on-chip based systems using betweenness centrality," *Future Gener. Comput. Syst.*, vol. 82, pp. 459–471, 2018.

[23] Z. Cui, L. Chen, Y. Bao, and M. Chen, "A swap-based cache set index scheme to leverage both superpage and page coloring optimizations," in *Proc. 51st ACM/EDAC/IEEE Des. Automat. Conf. (DAC)*, 2014, pp. 1–6.

[24] V. Selfa, J. Sahuquillo, S. Petit, and M. E. Gómez, "A hardware approach to fairly balance the inter-thread interference in shared caches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3021–3032, Nov. 2017.

[25] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, vol. 14, 2014, pp. 356–367.

[26] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "Swap: Effective fine-grain management of shared last-level caches with minimum hardware support," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 121–132.

[27] G. Aupy, A. Benoit, B. Goglin, L. Pottier, and Y. Robert, "Co-scheduling HPC workloads on cache-partitioned CMP platforms," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2018, pp. 348–358.

[28] K. T. Nguyen, "Introduction to cache allocation technology in the intel Xeon processor E5 v4 family," [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/software-enabling-for-cache-allocation-technology.html.

[29] J. Park, H. Yeom, and Y. Son, "Page reusability-based cache partitioning for multi-core systems," *IEEE Trans. Comput.*, vol. 69, no. 6, pp. 812–818, Jun. 2020.

[30] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with Intel's cache allocation technology," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Tech.*, vol. PACT'17, 2017, pp. 194–205.

[31] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, vol. 18, 2018, pp. 104–117.

[32] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, "Phase-aware cache partitioning to target both turnaround time and system performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2556–2568, Nov. 2020.

[33] J. C. Saez, F. Castro, G. Fanizzi, and M. Prieto-Matias, "LFOC+: A fair OS-level cache-clustering policy for commodity multicore systems," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1952–1967, Aug. 2022.

[34] Y. Shen, J. Xiao, and A. D. Pimentel, "TCPS: A task and cache-aware partitioned scheduler for hard real-time multi-core systems," in *Proc. ACM SIGPLAN/SIGBED Int. Conf. Lang., Compilers, Tools Embedded Syst. (LCTES)*, 2022, pp. 37–49.

[35] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-conscious offline real-time task scheduling for multi-core processors," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, 2017, pp. 14:1–14:22.

[36] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2009, pp. 245–254.

[37] Z. Guo, K. Yang, F. Yao, and A. Awad, "Inter-task cache interference aware partitioned real-time scheduling," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2020, pp. 218–226.

[38] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2005, pp. 340–351.

[39] Z. Xiao, L. Chen, B. Wang, J. Du, and K. Li, "Novel fairness-aware co-scheduling for shared cache contention game on chip multiprocessors," *Inf. Sci.*, vol. 526, pp. 68–85, 2020.

[40] S. Z. Sheikh and M. A. Pasha, "Energy-efficient real-time scheduling on multicores: A novel approach to model cache contention," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 4, pp. 1–25, 2020.

[41] Z. Fu, Z. Tang, L. Yang, and C. Liu, "An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2406–2420, Oct. 2020.

[42] T. Marinakis, A.-H. Haritatos, K. Nikas, G. Goumas, and I. Anagnostopoulos, "An efficient and fair scheduling policy for multiprocessor platforms," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2017, pp. 1–4.

[43] C. Tessler, V. P. Modekurthy, N. Fisher, and A. Saifullah, "Bringing inter-thread cache benefits to federated scheduling," in *Proc. IEEE Real-Time Embed. Technol. Appl. Symp. (RTAS)*, 2020, pp. 281–295.

[44] J. Zhang, S. Gao, N. S. Kim, and M. Jung, "CIAO: Cache interference-aware throughput-oriented architecture and scheduling for GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2018, pp. 149–159.

[45] J. Li, L. Liu, Y. Wu, X. Feng, and C. Wu, "Two-level task scheduling for irregular applications on GPU platform," *Int. J. Parallel Prog.*, vol. 45, no. 1, pp. 79–93, 2017.

[46] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?," in *Proc. 15th ACM SIGPLAN Symp. Prin. Practice Parallel Prog (PPoPP)*, 2010, pp. 203–212.

[47] "Montage." 2018. Accessed: May 08, 2024. [Online]. Available: http://montage.ipac.caltech.edu/

[48] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. 15th Int. Conf. Archit. Support Prog. Lang. Oper. Syst. (ASPLOS)*, 2010, pp. 129–141.

[49] J. Fu, D. Arteaga, and M. Zhao, "Locality-driven MRC construction and cache allocation," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, 2018, pp. 19–20.

[50] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," in *Proc. Int. Conf. Archit. Support Prog. Lang. Oper. Syst. (ASPLOS)*, 2013, pp. 343–356.

[51] F. Hameed, L. Bauer, and J. Henkel, "Reducing inter-core cache contention with an adaptive bank mapping policy in DRAM Cache," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis (CODES+ISSS)*, 2013, pp. 1–8.

[52] C. K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[53] F. D. Croce and R. Scatamacchia, "The longest processing time rule for identical parallel machines revisited," *J. Scheduling*, vol. 23, pp. 163–176, 2020.

[54] Z. Mo et al., "JASMIN: A parallel software infrastructure for scientific computing," *Frontiers Comput. Sci.*, vol. 4, no. 4, pp. 480–488, 2010.

[55] W. F. Ma and L. H. Wang, "Analysis of the Linux 2.6 Kernel Scheduler," in *Proc. Int. Conf. Comput. Des. Appl. (ICCDA)*, 2010, pp. 71–74.

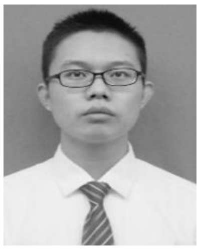
[56] "SPEC CPU 2017 Benchmark." Standard Performance Evaluation Corporation (SPEC). Accessed: Aug. 17, 2024. [Online]. Available: https://www.spec.org/cpu2017/

[57] "NAS Parallel Benchmarks (NPB)," NASA. Accessed: Aug. 17, 2024. [Online]. Available: https://www.nas.nasa.gov/software/npb.html

[58] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

Song Liu received the B.S. degree in computer science and technology from the Northwestern Polytechnical University, China, in 2009, and the Ph.D. degree in computer science and technology from Xi'an Jiaotong University, China, in 2018. He is currently with the School of Computer Science and Technology, Xi'an Jiaotong University as an Associate Professor. His research interests include parallel computing and code optimization.

Jie Ma received the B.S. degree in network engineering from Nanchang University, China, in 2019. She is currently working toward the M.S. degree in computer technology with Xi'an Jiaotong University. Her research interests include parallel computing and task scheduling.



Zengyuan Zhang received the B.S. degree in computer science and technology from Xi'an Jiaotong University, China, in 2020. He is currently working toward the M.S. degree in computer technology with Xi'an Jiaotong University. His research interests include compiler optimization and parallel computing.

Xinhe Wan received the B.S. degree from Xi'an Jiaotong University, China, in 2021. He is currently working toward the M.S. degree in computer technology with Xi'an Jiaotong University. His research interests include parallel computing and compiler optimization.

Bo Zhao received the Ph.D. degree from Humboldt Universität zu Berlin, Berlin, Germany, in 2022. He is currently an Assistant Professor with the Department of Computer Science, Aalto University, Finland. His research focuses on data-intensive systems, including scalable machine learning systems, distributed data management systems, and code optimization techniques.

Weiguo Wu received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, China, in 1986, 1993, and 2006, respectively. He is currently with the School of Computer Science and Technology, Xi'an Jiaotong University as a Professor. He is a senior member of the CCF. His research interests include high performance computer architecture, storage system, cloud computing, and embedded system.