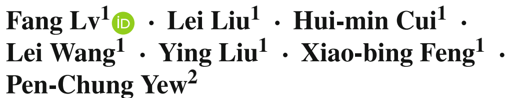
J Supercomput
DOI 10.1007/s11227-015-1427-7

WiseThrottling: a new asynchronous task scheduler for mitigating I/O bottleneck in large-scale datacenter servers


Fang Lv[1] · **Lei Liu**[1] · **Hui-min Cui**[1] · **Lei Wang**[1] · **Ying Liu**[1] · **Xiao-bing Feng**[1] · **Pen-Chung Yew**[2]

© Springer Science+Business Media New York 2015

Abstract Datacenter servers are stepping into an era marked by powerful multi-/many-core processors. Severe problems such as I/O contentions in those large-scale platforms pose an unprecedented challenge. Prior studies primarily considered I/O bandwidth as a major performance bottleneck. However, our work reveals that in many cases the fundamental cause of I/O contentions is the inefficiency of OS schedulers. Particularly, the modern system is not aware of this fact and thus suffers from poor I/O performance, especially for datacenter servers. Based on our findings, we propose a new software-based scheduling approach, *WiseThrottling*, to reduce I/O contention. WiseThrottling performs asynchronous and self-adjustment scheduling for concurrent tasks. We evaluate our approach across a wide range of C/OpenMP/MapReduce workloads on a 64-core server in Dawning Cluster datacenter. The experimental results exhibit that WiseThrottling is effective for reducing the I/O bottleneck and it can improve the overall system performance by up to 207 %.

Keywords Multi-/many-core server · I/O contention · Scheduling · Resource description

---

This research is supported by the National High Technology Research and Development Program of China under Grants No. 2012AA010902 and 2015AA011505; the NSFC under Grants No. 61202055, 61221062, 61303053, 61432016 and 61402445; and the National Basic Research Program of China under Grant No. 2011CB302504.

✉ Fang Lv
flv@ict.ac.cn

1 State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China

2 Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, MN, USA

Published online: 26 April 2015

1 Introduction

"I/O bottleneck" is known as one of the most serious problems that hurt the overall system performance in modern computing environments with multicore system such as datacenters and clusters [2,31] and many-core system such as GPUs [26]. However, in such multicore servers, two types of I/O conflicts, described as inter- and intra-task, lead to more severe I/O bottleneck. As shown in Fig. 1, inter-task I/O conflict typically comes from concurrent user tasks (e.g., batch-mode processing), while intra-task I/O conflict, such as the I/O contention between concurrent workers/reducers of a MapReduce application, often happens among the children of a single task. As people are increasingly dependent upon data intensive computing, I/O contention is becoming an increasingly significant issue [2,17,31].

Many studies aim to address the I/O problem. As hardware has once been considered the most critical factor that affects the I/O performance, lots of previous work primarily focused on optimizing the hard disk access behaviors to mitigate I/O latencies. For example, the efforts in [14,15] re-schedule I/O requests before they are sent to I/O devices, and [13] uses shared memory as disk caches, so that the expensive disk scanning latency can be dramatically reduced.

However, the root reason leading to serious I/O problem is not a hardware-only problem that can be solved by offering more I/O bandwidth and storage resources, but the inefficiency and ineffectiveness of I/O schedulers at the Operating System (OS) level in many common cases. The existing scheduler is ignorant of different sensitivities to I/O contentions from co-running jobs. To handle all I/O requests from concurrent tasks, it distributes the service timeslices to all co-runners "blindly". Under such scenario, the disk arm has to constantly move from one co-runner's requests to another. In general, doing so will destroy the original continuity of a task's file operation, and thus degrade the performance of the tasks. The problem becomes more severe in the large-scale servers equipped with a large number of CMPs, and this causes underutilization of existing I/O bandwidth resources. To solve this problem, some studies focus on optimizing the OS task scheduler or I/O scheduler. For instance, the effort in [22] for MapReduce optimization uses I/O throttling approach to mitigate the negative effects caused by streaming I/O contentions, which is an enforcement

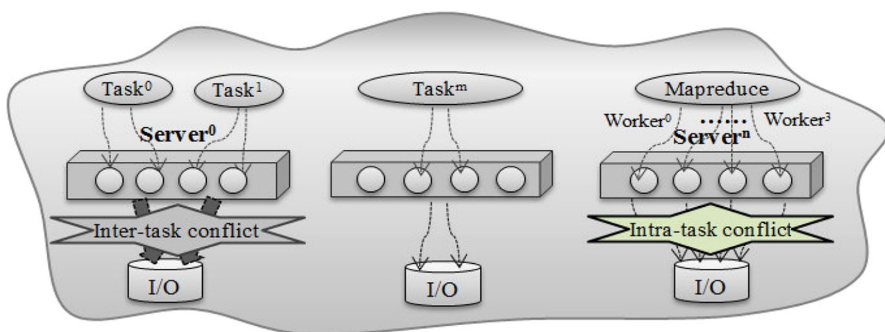


Fig. 1 Two types of I/O conflicts in a server node in datacenter

of the existing I/O scheduler. Another work [21] that proposes a simple dynamic scheduling method also shows that it is necessary, and often effective, to optimize the OS scheduler to benefit the overall system performance in some cases.

Optimizing the OS level scheduler is a cost-effective approach to improve the overall system performance because a software approach is easier to be used in practice, and the existing storage resource can be better utilized without any hardware modifications that may incur expensive overhead. Thus, many recent research efforts adopt a software-based approach. Nevertheless, the previous studies also have their shortcomings. First, they often use specific scheduling intervals and thus are not able to handle the irregular I/O contention behaviors efficiently. Second, they cannot detect the application's sensitivities to I/O contentions on-the-fly accurately in the cases where multi-threaded or multi-programmed tasks are running together. In summary, the existing I/O scheduler needs a "wise" scheduling approach, which is capable of handling diverse concurrent I/O demands and thus benefits the I/O bandwidth utilization. To the best of our knowledge, not only the fairness-oriented scheduler in Linux kernel, but also the mainstream I/O scheduling mechanism, faces the same problem of lacking a wise scheduling.

In this paper, we propose an asynchronous task scheduling mechanism, WiseThrottling, to meet the challenges in modern data intensive computing environment with heavy I/O contention. WiseThrottling works orthogonally with the default Linux I/O scheduler, and schedules I/O behaviors to avoid resource contention with reasonable heuristics. The proposed scheduling mechanism has the following features. First, it maintains an adaptive and non-uniform I/O resource description for each running task, and thus its scheduling is not "blind" but self-adjusted and scalable for all concurrent tasks. Second, WiseThrottling is sensitive to each task's I/O features within small "timeslices". Therefore, it is able to make proper scheduling decisions in real-time and dynamic way. Third, our mechanism supports both fine-grain and coarse-grain scheduling, and can mitigate both inter-task and intra-task I/O contention. Fourth, our approach can be easily deployed in modern CMP system without significant overhead, and the proposed scheduling methodology is scalable to large-scale CMP platforms.

We implement WiseThrottling on a 64-core server in Dawning cluster datacenter. To verify the effectiveness of our method, we compare it with the current Linux OS scheduler. The evaluations are conducted for more than 20 workloads that are composed of C/OpenMP/MapReduce applications. These workloads represent three typical service patterns, similar-pattern services, compounded-pattern services and batch-mode services in Dawning cluster datacenter. The experimental results show that WiseThrottling is effective to mitigate the negative performance pain from both inter- and intra-task I/O conflicts. The proposed mechanism can improve the overall system performance by up to 207.0 % for similar-pattern services, 11.2–28.7 % for compounded-pattern services, and 24.9–26.7 % for batch-mode services.

This paper makes the following contributions:

- We identify that, in large-scale CMP systems, I/O bandwidth is not always the determining factor for the system performance. The inefficiencies in mainstream schedulers contribute more to performance degradations than hardware factors, and are shown to be the major bottlenecks.

- We identify two key metrics, which are used to facilitate I/O optimization under the scenario of I/O competitions in modern large-scale CMP servers.
- We identify that different I/O interfaces can lead to different I/O competition behaviors. This finding motivates us to adopt a dynamic scheduling approach as the solution.
- We propose and implement a new asynchronous scheduling policy, WiseThrottling, for large-scale servers, which can mitigate both inter-task I/O conflicts and intra-task I/O conflicts, and thus improve the overall system performance.
- We extensively evaluate WiseThrottling on a 64-core server in Dawning datacenter using diverse applications including C/OpenMP/MapReduce applications. Experiments over more than 20 workloads show that WiseThrottling is very effective in mitigating I/O contentions (achieves up to 207 % performance improvement).

The rest of the paper is organized as follows: Sect. 2 discusses related work. Section 3 expatiates the two inefficiencies that the current schedulers encounter when dealing with I/O contentions. Section 4 describes the implementation of WiseThrottling. Section 5 presents the performance measurements and analyzes using the proposed method for a variety of tasks. Section 6 draws the conclusion and discusses the future work.

2 Related work

The I/O bottleneck, commonly considered as the mismatch between the file transaction power and the processor computing power or memory storage capacity, has been well studied. It has been tackled from a very broad scope, including optimizing from applications, operating system, or parallel task management, etc., within parallel computing environments.

2.1 Parallel I/O optimizations

Noncontiguous I/O requests are one of the key reasons that lead to poor I/O performance. There are many significant research works from last century, which have devoted to combining small and noncontiguous I/O operations.

Parallel I/O is an important research topic for high-performance computing. Two-phase I/O [6], data sieving and collective I/O [35], etc., are all efforts for enhancing parallel I/O. Static [35] or dynamic data sieving [20] is proven to be the most efficient approach which has been applied widely. Some other approaches are more practical and widely used through combination with parallel I/O API [35], e.g., the combination of data sieving with ROMIO [36] or the combination of list I/O with ROMIO [7]. Work in [27] also proposes a performance model to detect and combine parallel I/O by thread grouping for many-core system.

I/O operations scheduling is an effective approach for disk scanning reduction. The scheduling on I/O operations can benefit from high concurrency among I/O operations [14,15]. The work in [11] is an extension work based on previous works, which applies edge-coloring in scheduling of I/O operations for higher concurrency.

For industrial applications on service platforms, optimizations on the programming language or source applications used to be very effective approaches to enhance the I/O performance through higher parallelism [18].

Optimizations on I/O operations mentioned above concentrate more on optimizations on noncontiguous I/O operations rather than I/O conflicts among multiple tasks.

2.2 I/O scheduling on SMP/CMP systems

Software scheduling policies are always better choices for mitigating resource conflict, including I/O contention. Among all I/O schedulers, fairness-oriented I/O scheduler is the main type that has been thoroughly studied in the past. I/O scheduling policies, such as noop, deadline and cfq, are among the most commonly used polices in mainstream OS such as Linux [32]. The work in [32] gives a comparative study on all these policies. However, fairness-based schedulers often take little or no consideration in performance. Due to a lack of knowledge in the applications characteristics, shared resource competition is difficult to resolve with fairness-oriented policies.

The work in [21] is a preliminary work for I/O contentions on large-scale servers. It focuses on the serious influences from I/O contentions, and proposes a dynamic scheduling policy. This policy can benefit from dynamic scheduling which combines proper I/O behavior descriptions and sensitivity estimations. However, it can neither adapt to the dynamic I/O behavior of co-running applications, nor to the variation of sensitivities of applications to I/O contentions. In this paper, we further delve into the root reason for I/O bottlenecks. Based on deep insights into the inefficiencies of current fairness-based scheduler, WiseThrottling is built on adaptive heuristics. During the dynamic scheduling, I/O resource descriptions of each application keep on adjusting according to periodical I/O behaviors. The self-adjustment can make the scheduling more precise, and reduce unnecessary examinations, which makes it more beneficial.

Disk caching in memory is an effective technique to speed up I/O performance. The work in [13] demonstrated several I/O optimizations with shared memory for specific languages, e.g., MPI-IO applications. Since memory-associated I/O optimization will accelerate the memory consumption, a careful trade-off needs to be made. We have exposed the risks from this kind of optimizations in Sect. 5.

FIOS in [29] is a flash I/O scheduler that targets to solid-state drives (SSD) and takes both fairness and performance into consideration. The most important premise of FIOS is the discrepancy between read time and write time on SSD. Based on this asymmetry, the scheduler can serve for better performance with a preference to reads using timeslice-based scheduling. This could do well in some applications that used to stall by writes.

2.3 I/O scheduling in cluster or datacenter

For distributed environments such as cluster, workstations, etc., I/O bottlenecks are quite different to and more complicated than on a single server. In this kind of service environments, in addition to performance issue, resource utilizations and Quality of Service (QoS) are all key concerns for many scheduler or optimizations.

When a server owner could not fully exploit I/O resources, some guest jobs are allowed to make good use of idle cycles in some platforms. However, under such scenario, it is necessary to protect the performances of the machine owner. I/O throttling is such kind of optimization for higher resource utilizations, which is the most similar idea to our work from the perspective of coordinating I/O quantities [1,30]. The study in [22] is a very recent work which exploits I/O throttling in MapReduce. However, it would sacrifice low-QoS tasks to ensure the performance of high-QoS tasks.

Gang scheduler [25] is a popular timeslice job scheduler for environments of supercomputing center. However, it lacks of strategies for the unbalance between local and remote file location, which may lead to unsatisfactory performance. On this problem, IOGS is an enhanced job scheduler based on Gang Scheduler for supercomputing center [37]. According to the knowledge of file locations, IOGS can schedule the jobs to the nodes, which have less costs of file access, therefore, upgrade the performance of user applications.

I/O problems on virtual machines have also become a hot research topic. The work in [28] focuses on I/O contention among multiple guest domains. The work points out that the fairness in I/O resource allocation could lead to poor performance due to the differences in I/O requests. The work in [12] points out a key shortcoming in the scheduler of current virtual machine monitors (VMM) that may lead to poor performance because they are agnostic to the communication behavior of applications. Solutions include techniques such as booking pages for communication, anticipatory scheduling for sender, etc., which can make VMM more aware of the characteristics of applications. Flubber [19] is a more recent work for the utilization of I/O devices on VM environments. The two-level design makes it more efficient for both throughput and specified latency requirements.

3 Motivation: inefficiencies of the current schedulers in large-scale servers

I/O bottleneck in any large-scale server closely relates to the inefficiencies of current OS scheduling on dealing with concurrent I/O demands. These inefficiencies exist not only in the different fairness-oriented schedulers, but also in other schedulers. The root reason in these strategies is the ignorance of different sensitivities to I/O contentions from co-running jobs. To handle all I/O requests from concurrent tasks, the existing scheduler has to distribute the service timeslices to all co-runners "blindly". Under such scenario, the disk arm has to constantly move from one co-runner's requests to another. This strategy spends much more time on disk arm moving rather than effective data accesses, thus underutilizes the I/O bandwidth and degrades the performance of the tasks.

In this section, we will demonstrate the serious performance degradations from I/O contentions inside a large-scale server, and then further investigate the role that the current OS scheduler plays during I/O competitions. Moreover, we will introduce our quantitative method through which the potential performance influences caused by I/O contentions can be predicted and evaluated.

3.1 Platform

The platform employed in our work is a widely used Intel® Xeon® E7-8830 server in Dawning datacenter. The server has a total of 256 GB main memory.

In modern operating system, for an I/O request, OS scheduler handles it via a two-step scheduling process (task scheduler and I/O scheduler). Task scheduler directly affects the co-running tasks, and it also indirectly influences the I/O performance through interfering with the sequences of I/O requests. I/O scheduler then takes the charge of issuing I/O requests to I/O devices after task scheduler. Thus, these two scheduling policies have cooperative contributions to the entire I/O performance.

Many mainstream schedulers are based on fairness policies. For example, in Linux OS, the default task scheduler is Completely Fair Scheduler (cfs) and the default I/O scheduler is Completely Fair Queuing (cfq). In this section, we mainly study the collaborative effects of the default OS scheduler, which represents both cfs and cfq scheduling policies. The OS in our platform is Linux CentOS 6.3 with kernel 2.6.32.

3.2 Background: performance influences from I/O contentions

In this subsection, we will demonstrate the serious influences from I/O contentions through a series of experiments on our large-scale CMP server.

3.2.1 Experiments design for I/O contentions

We use a micro multi-threaded benchmark, IO_RANKING, to simulate a task with specific I/O bandwidth demands. As shown in Fig. 2, the kernel code in IO_RANKING reads continuous blocks (via ***fgets***) from an external file. To avoid the data caching effects, the process reads only once from external file. We only show the results with I/O read operations rather than write operations with ***fwrite*** and ***fflush***, since they exhibit the similar phenomena.

IO_RANKING has two key parameters: (1) Size of Block, representing the size of data that will be read, and (2) Interval, denoting the time interval between two successive block readings (via inserting nops). Through varying these two parameters, I/O operations with different granularities for an IO_RANKING task will be generated.

Fig. 2 Kernel of IO_RANKING

```
Procedure IO_RANKING
1:  #define block_size 32 //data size for one read
    #define nops 5000 /*intervals between two
2:                          *successive reads */
3:
4:  /*read file without repeatition in case
5:   *of page caching */
6:  while( !eof(file)) {
7:          fgets(file, block_size);
8:          //use nops to adjust the bandwidth density
9:          for (k=0; k<nops; k++) {
10:              asm ( "nop" );
11:         }
12: }
```

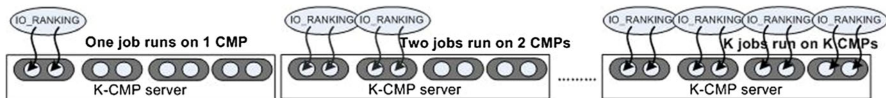


Fig. 3 The process of gradually increasing IO_RANKING tasks in "CMP Stacking" method [21]

However, the contentions and interferences among tasks occur not only on I/O level, but also in memory system (e.g., LLC, DRAM). Using I/O intensive task only is not sufficient to fully demonstrate the harm that I/O will bring. We need a method to clearly distinguish I/O contentions from contentions in other resources. In this paper, we use "CMP Stacking" [21] to guarantee that all performance influences in our experiments are mainly from the gradually increase in I/O contentions.

Figure 3 demonstrates the basic idea of "CMP Stacking" method. It uses CMP as the basic unit for resource allocation and task execution, i.e., adds a new IO_RANKING task on a new CMP each time during the experiments. Take the K-CMP server in the figure for example, following the rule of "CMP Stacking", one IO_RANKING task runs on one CMP, two tasks on two CMPs, and so on. This strategy includes two constraints, the ***confinement*** and the ***sustainability***, that can make sure that the performance impacts during co-running mainly come from the global I/O contentions, not no-chip-network resource contentions. The constraint of ***confinement*** satisfies any resource requirements of a task with the resources inside its own CMP. No cross-CMP resource requirements will be initiated, and no on-chip-network resources inside a CMP will be contended between any of the two co-runners. The constraint of ***sustainability*** can ensure that the confinement will last for the whole co-running process. With these two constraints, we can attribute the major performance impacts to I/O contentions during the process of "CMP Stacking". More details about this method can be found in [21].

Following the steps of "CMP Stacking" method [21], multiple IO_RANKING tasks are grouped together to simulate concurrent I/O contentions. By gradually increasing the number of concurrent IO_RANKING tasks, i.e., from 1 task to 8 tasks on a 8-CMP server, the performance impacts from global I/O contentions are able to be observed clearly through the variation of averaged task performance.

3.2.2 Impacts from I/O contentions

Figure 4 is from our previous work [21]. It illustrates the correlation between different I/O bandwidth demands and the corresponding performance degradations caused by I/O contentions. There are 12 groups of experiments represented by 12 curves. Each curve tracks the averaged performance degradation for a task during k-CMP co-running experiments (x-axis represents the number of k). Here, k-CMP is performed with k concurrent jobs ranging from 1 to 8 and each job is mapped to a separate CMP. 12 curves are experimented with 12 categories of tasks, which have different I/O bandwidth demands ranging from 1.5 MB/s per CMP to 26.4 MB/s per CMP. Data in each curve denote the degradation trends during the increasing number of concurrent I/O tasks, which is made up of totally duplicated tasks. Apparently, the performance degradation is becoming more and more obvious when more tasks take part in the co-

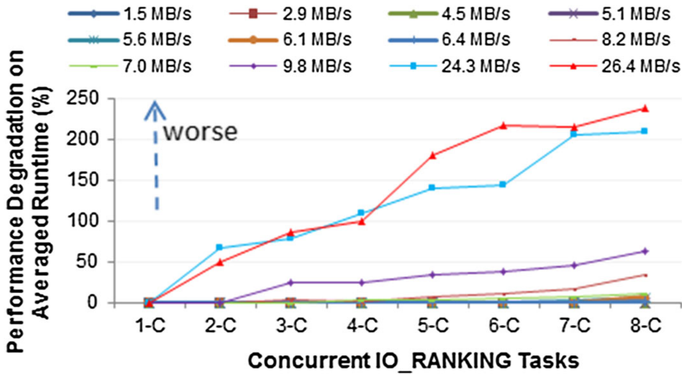


Fig. 4 Average performance degradation while increasing concurrent IO_RANKING tasks [21]

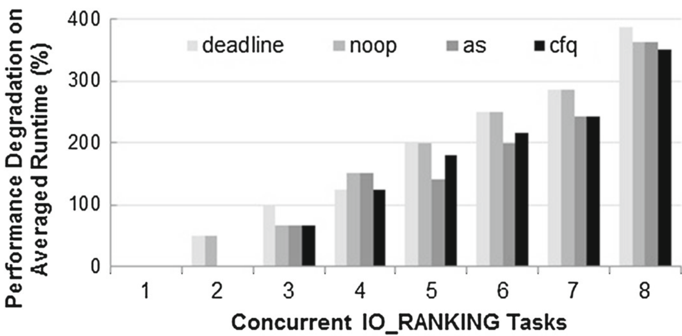


Fig. 5 Performance contrast between the four I/O schedulers in Linux system

running. Particularly, for the tasks with average I/O bandwidth of 24.3 and 26.4 MB/s, the degradations are more significant than those with lower bandwidth demands.

The experimental results exhibit that severe I/O contentions can damage the co-running performance eventually, and we come up with the following conclusions:

- I/O bandwidth demands: the performance in co-running cases correlates closely to the tasks' average I/O demands. The more the averaged I/O bandwidth demands are, the more serious the system's performance degradations will be.
- Number of concurrent tasks: the co-running performance correlates closely to the number of co-runner tasks. The more the co-runners are, the more severe the system's performance slowdowns will be.

3.2.3 Discussion for the difference between I/O schedulers

Not only the default scheduler, but also other schedulers may lead to performance degradation. In Fig. 5, we compare four I/O schedulers [noop, deadline, anticipatory and cfq (default)] [32] in current Linux OS. Using the method in [21], the experimental results show that with the increasing number of concurrent tasks, all the four schedulers exhibit the same performance degradation trend. cfq performs a little bet-

Table 1 Statistic entries with *iostat*

Entry	Description
rMB/s	The number of megabytes read from the device per second
wMB/s	The number of megabytes written to the device per second
await	The average time (in ms) for I/O requests issued to the device to be served
svctm	The average service time (in ms) for I/O requests that are issued to the device
%util	Percentage of CPU time during which I/O requests are issued to the device (bandwidth utilization for the device)

ter than others, in the cases where the concurrent IO_RANKING is less than 5. In our work, we use default I/O scheduler, cfq scheduler, in the paper, as it is typical and widely used in common productive environments. In later subsections, we mainly study the collaborative effects of the default OS scheduler, which represents both cfs task scheduler and the cfq I/O scheduler. We will further investigate the root reason why the default scheduler hurts the system performance.

3.3 Inefficiencies of the OS scheduler

The design of the existing OS scheduler cannot handle I/O contentions effectively, and thus hurts the overall system performance in many cases. This will lead to (1) low resource utilization and (2) high I/O latencies, and OS monitoring mechanism (e.g., *iostat*) can get the details of them. In Table 1, we list five entries adopted in our analyses. In our experiments, we sample these entries with *iostat* at a fixed time interval specified by the user (1 s at least).

3.3.1 Low resource utilization: low bandwidth utilization vs high device utilizations

The first inefficiency of the default OS scheduler is that it will lead to improper resource utilization when coping with multiple concurrent competitors. This is because, under the guidance of fairness-oriented policies, the current scheduler distributes the OS serving timeslices equally among tasks of the same priority, resulting in hopping of I/O devices, which not only destroys the original data continuity of file operations for a task, but also fast saturates I/O devices. We demonstrate this defect by a comparison between the real bandwidth utilization and the I/O device utilization. We use three entries, rMB/s, wMB/s and %util in Table 1, to generate the bandwidth utilization and the device utilization.

Figure 6 shows the experimental results in terms of IO_RANKING with 1.5, 5.1 and 26.4 MB/s per CMP, respectively. Data in the figure with each kind of I/O bandwidth demands are composed of two curves: the bandwidth utilization curve (the dotted curve denoted by bw) and the I/O device utilization curve (the solid curve denoted by util). Notably, no matter how many concurrent I/O tasks are running, the gap between the bandwidth utilization and the I/O device utilization always exists, indicating that bandwidth is not fully exploited and in “hungry” in common cases though the I/O

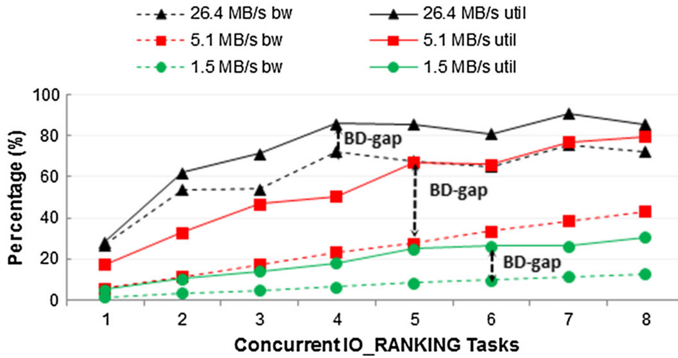


Fig. 6 Contrasts between the bandwidth utilization and the device utilization

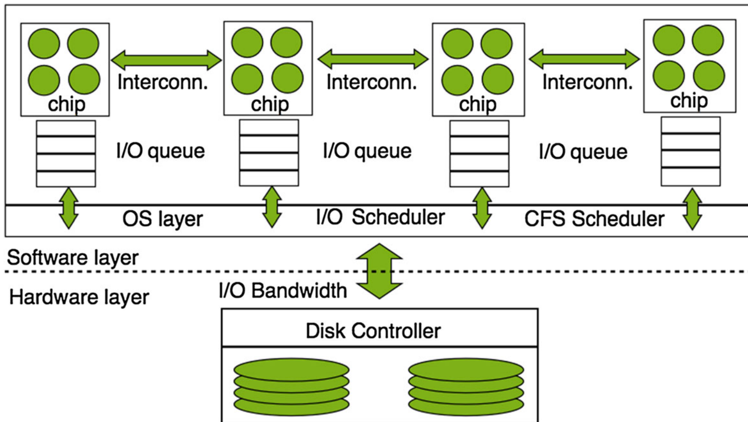


Fig. 7 Components of I/O costs [21]

devices are "busy". We denote the gap as ***bandwidth-device gap (BD-gap)***. BD-gap becomes more significant with the number of co-runner tasks increases. The reason is when more and more concurrent tasks begin to compete with each other, I/O device will approach to saturation rapidly, while only small part of the system bandwidth can be utilized efficiently during this process. Another key point, which should be noticed, is that with the default OS scheduling policy, the system's peak I/O bandwidth (about 90 MB/s) is very hard to be fully exploited under serious I/O contentions. To sum up, we come to the conclusion that improper I/O device service policy leads to low bandwidth utilization, and this is due to inefficiency of the default OS scheduler.

3.3.2 High I/O costs: high software costs vs low hardware costs

Furthermore, the inefficiency of the default OS scheduler can be revealed through analyses upon the high I/O costs during I/O contentions. For each I/O request to local disk, the latency can be divided into two major portions (illuminated in Fig. 7 [21]): the software service time in OS layer, and the hard disk service time. Generally

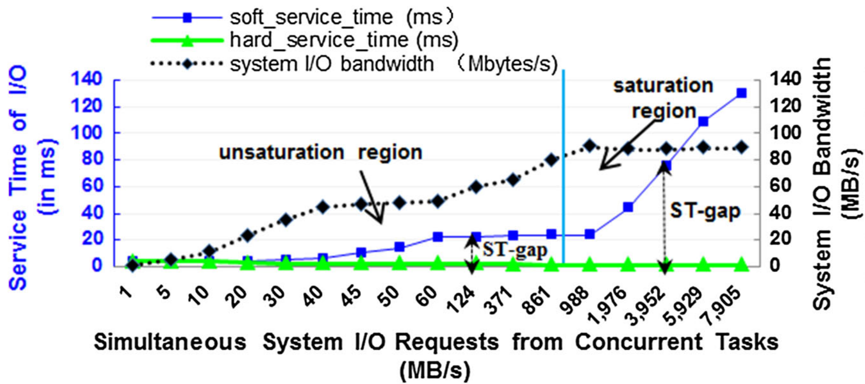


Fig. 8 Correlation between service time and simultaneous I/O quantities. Unsaturation region refers to the *left* of the *blue* "watershed" *line* where the system I/O bandwidth keeps increasing. Saturation region refers to the *right* of the "watershed" where the system I/O bandwidth reaches the peak and stops increasing (color figure online)

speaking, in our investigation, we find that the default OS scheduler will lead to very high software service time, which plays dominating role for the overall system performance. We will explain the inefficiency through contrasts between these two portions during co-running.

We use ***soft_service_time*** to stand for the software portion in I/O costs, which is mainly composed of the waiting time of an I/O request in I/O queues and the coordination time by the software scheduler.

We use ***hard_service_time*** to represent the hardware costs in I/O costs, which is the actual service time of an I/O request by the I/O device.

Therefore, we can calculate I/O latency with software service time and hard disk service time as in Eq. (1):

$$I/O__latency = soft__service_time + hard__service_time \quad (1)$$

We use two entries from *iostat* to analyze the variation of the two portions of I/O costs. The entry ***await*** in the report generated by iostat includes both the software service time and the hard disk service time. The entry ***svctm*** in the report shows the disk service time. Thus, the software service time can be deduced from these two values. The overhead that contributes to I/O impacts can be revealed via the contrasts between these two components of I/O latencies for each device operation.

The contrasts between the software portion and the hardware portion are performed through more experiments with IO_RANKING tasks, named ***expanded IO_RANKING*** experiments. Here, we generate more than 17 groups of simultaneous I/O requests through varying parameters in IO_RANKING to study the variation of different portions of the I/O costs. Average I/O demands of these tasks range from fine-grain I/O operations (about 1 MB/s) to coarse-grain I/O operations (8 GB/s).

In Fig. 8, our experimental results exhibit that the software service time (blue curve) always dominates I/O latency, and it keeps increasing with the growth of

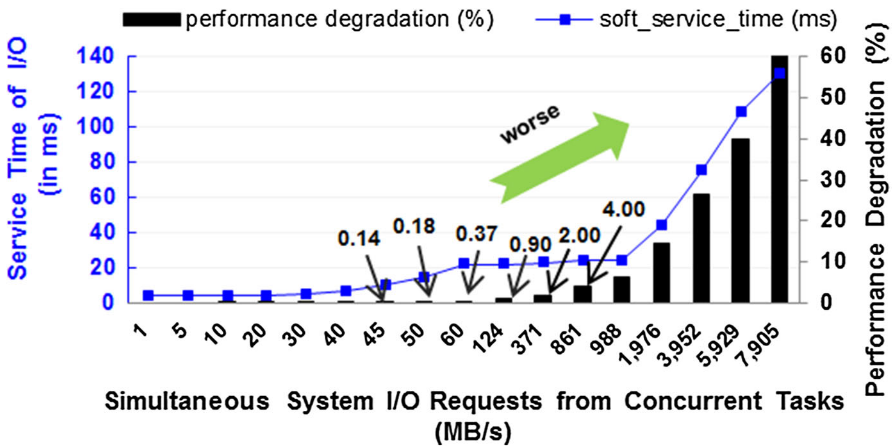


Fig. 9 Correlation between software_service_time and the performance degradation. They have similar trends. Higher software_service_time will lead to more severe performance degradation

simultaneous I/O requests (as the x-axis denotes). But, on the contrast, the hard disk service time (green curve) does not change obviously all the time. Thus, we conclude that I/O_latency approximates soft_service_time. As we analyzed in the previous section, the reason for the high software service time lies in that the current fairness-oriented scheduler is ignorant of different sensitivities to I/O contentions from co-running jobs. The existing scheduler distributes the service timeslices to all co-runners fairly and the disk arm has to constantly move from one co-runner's requests to another's. In general, this will destroy the original continuity of a task's file operation, lengthen the service time, and finally degrade the performance of the tasks severely.

Moreover, the gap between the software serve time curve and the hard disk service time curve becomes more and more significant and un-negligible with the users' I/O requests increase. We call the gap as ***service time gap (ST-gap)***. ST-gap begins even in the unsaturation region of the system's peak I/O bandwidth (the left of the blue "watershed" in the figure). Figure 8 shows when the overall I/O demands from concurrent tasks just reach 44 % of the system's peak bandwidth (40 MB/s out of 90 MB/s in the figure), ST-gap has already become obvious. It also illustrates that ST-gap is more obvious in I/O bandwidth saturation region than in unsaturation region, manifesting that the default OS scheduling policy does not work well under I/O intensive condition (even in the unsaturation region), and it becomes more severe when the system I/O bandwidth achieves its peak (saturation region). Apparently, the system I/O bandwidth could not be fully exploited via the default OS scheduler.

Additionally, our study uncovers a correlation between the software service time and the corresponding performance degradation caused by I/O contentions. As shown in Fig. 9, they appear the same trend, indicating that the higher software service time will incur more severe performance degradation. Therefore, these observations motivate us that when we use a "wise" scheduling policy that aims to reduce the software service time, we will improve the overall system performance.

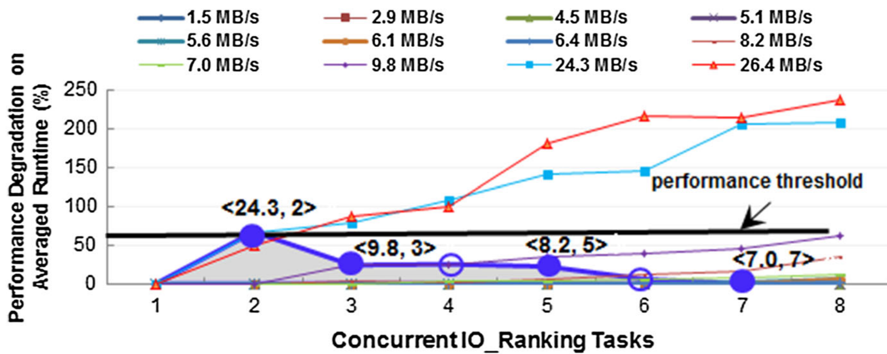


Fig. 10 Vector Points (color figure online)

3.4 Two metrics on potential performance impacts from I/O contentions

Above investigations show that the bandwidth is not the fundamental reason that leads to the degradation of system performance. So we need other metrics to illustrate the potential I/O contentions and the corresponding impacts.

3.4.1 Vector Points

The first metric is based on the two findings that are discussed in Sect. 3.2.2. They indicate that the system performance depends on two key features in co-running cases, not only the average I/O bandwidth requirements (abbr. A-BW), but also the number of co-runners (abbr. N-CO). We use a ***Vector Point*** ***<A-BW, N-CO>*** to describe these cooperative effects on the co-running performance. We can obtain Vector Points through experiments with IO_RANKING tasks. More details about the fine-grained experiments can be referred in Sect. 3.2.2.

Vector Points are empirical values which are picked out in two steps under a performance threshold. As the black horizontal line shows in Fig. 10, the threshold is the potential performance degradation which is allowed by users. Under this threshold line, the first step for Vector Points is to select a point (denoted with blue points) which has an average bandwidth demands as high as possible for each number on the x-axis. Those points, Vector Points, denote that for co-running tasks with average I/O demands of no more than A-BW, only when the number of co-runners keeps below the value of N-CO, the co-running execution can be guaranteed, and otherwise I/O contentions will hurt the co-running performance. For example, Vector Point <24.3, 2> denotes that an IO_RANKING task with averaged demands of no less than 24.3 MB/s may suffer degradation when only two such kind of co-runners are launched (in an 8-socket, 64-core server).

However, in the 12 groups of experiments, neighbor numbers on the x-axis always have very similar Vector Points. The second step is to coalesce those points into one. Since all these Vector Points are used to guide scheduling, coalescing is good for reducing the scheduling overheads. In the figure, the Vector Points on Number 4 and

Number 6 (denoted in blue rings) can be merged with Number 5 and Number 7 (blue points), respectively.

All Vector Points constitute a blue curve in Fig. 10. Separated by this curve, two different regions can be observed which are the grey region (below the curve) and the white region. In the grey region, all I/O demands from the co-runners can be satisfied within the user's performance demands. On the contrary, for co-running tasks with a certain I/O bandwidth requirements in the white region, once the co-runner exceeds a degree (e.g., 3 tasks for the task of 24.3 MB/s), obvious performance losses will occur. Therefore, these Vector Points can be used in the scheduler for judging potential performance losses. In our future work, we will develop automatic models for Vector Point selection so that they can be more precise.

3.4.2 BW_Ratio: ratio of the total I/O demands

Another metric used to detect potential performance losses is the ratio of the total I/O demands to the system's peak I/O bandwidth. Under the circumstance of I/O contentions, performance losses always happen when the total I/O demands accumulate to specific degree, e.g., 50 % of the system's peak I/O bandwidth (about 90 MB/s in our work). We call this ratio ***BW_Ratio***. Take the 26.4 MB/s task in Fig. 10 as example, 50 % performance degradation first appear when there are only 2 co-runners in the system. At that time, the total I/O demands are 52.8 MB/s, which is only about 58.7 % of the peak I/O bandwidth. Therefore, BW_Ratio is able to work as a metric for performance losses predication.

In summary, we are motivated to optimize I/O contentions from the following perspectives:

1. The current schedulers are the ultimate bottleneck for co-running performance degradations. They will result in two gaps, BD-gap and ST-gap, which stand for two inefficiencies, low bandwidth utilization and unreasonable high software service time.
2. The system I/O bandwidth is not the key factor for co-running performance. On the contrary, two metrics are effective to predict the potential performance losses under I/O competitions of concurrent execution. An efficient and effective "wise" task scheduling policy is necessary for large-scale servers, which regulates I/O contentions and thus improve the overall system performance.

4 WiseThrottling: an asynchronous scheduler for I/O contentions

WiseThrottling achieves wise scheduling through taking different I/O characteristics and sensitivities of co-runners into its consideration. To depict and mitigate I/O conflicts effectively, WiseThrottling is designed to provide: (1) an adaptive I/O resource description for depicting periodical I/O behaviors of cluster applications. (2) an asynchronous scheduling policy, which supervises I/O contentions according to the individual I/O description of each application. With these measures, WiseThrottling can detect I/O contentions. Moreover, it applies to both inter-task conflicts and intra-

task conflicts, which further improves the overall resources utilization through saving the whole serving time.

The following subsections show ideas in WiseThrottling mechanism. Section 4.1 introduces the framework of WiseThrottling. Section 4.2 discusses the core idea of WiseThrottling about how to be more sensitive to the individual I/O characteristics of each task. The detailed design for the individual I/O resource description will be stated in this subsection. Section 4.3 presents the step-by-step design of WiseThrottling. Section 4.4 gives an example for the process of WiseThrottling.

4.1 Overview of WiseThrottling's framework

Large-scale servers are quite different from small-scale ones in terms of the capability of supporting more co-runners that exhibit diverse I/O features. This leads to two issues: (1) How to depict I/O features for every task? (2) How to detect I/O conflicts accurately without incurring high overhead? Our WiseThrottling depends on the solution of them.

To fulfill the above requirements, we use a two-dimensional I/O description <***dtimeslice***, ***sensitivity***> in the implementation of WiseThrottling. In this description, ***dtimeslice*** is to describe the periodical I/O behaviors, while ***sensitivity*** is to reflect the potential performance vulnerability of a task in the cases of I/O contentions. Values of the description for each task are not immutable. On the contrary, to avoid unnecessary inspections and to reduce scheduling overhead, the properties are designed to be self-adaptive during the real-time scheduling process. Hence, WiseThrottling has two key modules (static and dynamic), which are responsible for the initial values and the adaptive values, respectively.

Figure 11 displays the general scheme of WiseThrottling. As mentioned above, it relies on a two-dimensional I/O description <***dtimeslice***, ***sensitivity***> for both I/O contention detection and conflicts regulation. WiseThrottling is composed of two

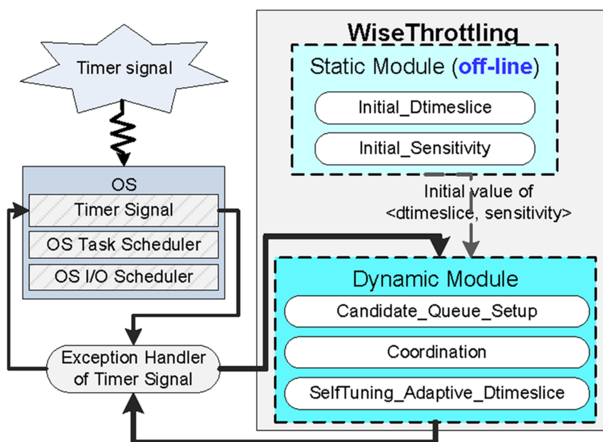


Fig. 11 Framework of WiseThrottling

```
Procedure Dynamic_Module
   {
      /* 1: update candidate queue with  tasks
      * that are busy at the current interval. This is made
      * according to the dtimeslice property.    */
1:    Candidate_Queue_Setup();
2:
3:   /* 2: the kernel part of the dynamic module, which
4:    * inspects I/O conflicts and implements scheduling */
5:
6:    Coordination();
7:
8:   /* 3: adaptive tuning for dtimeslice */
9:
10:   SelfTuning_Adaptive_Dtimeslice();
11:
12:  /* exit the current interval */
     exit ;
   }
```

Fig. 12 The dynamic module is registered as an exception handler for the timer interrupt

modules, i.e., the static module and the dynamic module. Static module creates a two-dimensional I/O vector description for each task off-line. The dynamic module plays a more important role to deal with real-time I/O conflicts on-the-fly. It uses a real-time asynchronous scheduling, i.e., to detect and mitigate I/O conflicts under the guidance of the each task's individual I/O description during co-running. The dynamic module is designed as a user-level scheduler triggered by the timer signal at a specified time interval.

Static module: Because WiseThrottling is based on I/O descriptions, the major function of the static module is to generate initial I/O descriptions for every task. The initialization includes two processes: ***Initial_Dtimeslice*** and ***Initial_Sensitivity*** in Fig. 11. More details about the algorithms will be introduced in Sect. 4.2. These initial values will be used as the input for the dynamic module. In WiseThrottling, resource descriptions for all co-runners can be either the same or different. To reduce overheads, the static module works in off-line way.

Dynamic module: Besides the adaptive two-dimensional I/O descriptions, another feature of WiseThrottling is that it uses an asynchronous scheduling policy to perform its scheduling, i.e., to schedule co-running tasks according to their individual I/O properties. The dynamic module in Fig. 11 is the core part to implement the asynchronous scheduling policy. It is responsible for creating the candidate queue, coordinating I/O contentions, and adjusting I/O descriptions adaptively according to the real-time I/O behaviors for co-runners. These steps are described with algorithms of ***Candidate_Queue_Setup***, ***Coordination***, ***SelfTuning_Adaptive_Dtimeslice***, respectively, as the pseudo codes shown in Fig. 12. More details about the dynamic module will be introduced in Sect. 4.3.2.

The dynamic module is registered as an exception handler for the timer interrupt, and is triggered periodically by the timer interrupt. The pseudo-code of ***Exception_Handler_of_Timer_Signal*** in Fig. 13 demonstrates the trigger process. Each time when OS receives a timer signal at a specified interval, named *CHECK_INTERVAL*

```
Procedure Exception_Handler_of_Timer_Signal
   {
1: /* The timer interrupt uses Dynamic_Module
2:  * as the exception handler. The interval for
3:  * the timer interrupt is set with CHECK_INTERVAL
4:  */
5:   struct itimerval IO_Interval;
6:   IO_Interval.it_interval.tv_sec = CHECK_INTERVAL;
7:   Signal(SIGALRM, Dynamic_Module);
8:   int res = setitimer(ITIMER_REAL, & IO_Interval, NULL);
9: }
```

Fig. 13 The algorithm of the dynamic module

(at least 1 s in our work), the exception handler will transfer the task management control to the dynamic module, named ***Dynamic_Module***. It should be noticed that *CHECK_INTERVAL* only specifies fixed interval to start scheduling routines without imposing any unnecessary scheduling. After the routine of the dynamic module is initiated, it will only examine the co-runners which are I/O busy at the moment. Using the information of tasks' I/O descriptions, the dynamic module starts to coordinate I/O contentions, and adjusts I/O descriptions according to the current I/O behaviors for co-runners under examination via I/O descriptions at the moment. The descriptions are useful in depiction for I/O behaviors and regulation for I/O conflicts during co-running process. After WiseThrottling finishes its dynamic scheduling, the OS task scheduler and the OS default I/O scheduler will take back the charge once again.

As a standalone scheduling policy, WiseThrottling can be turned on or off according to users' demands. Moreover, these modules are portable for other OS after a few platform-dependent modifications.

4.2 Adaptive two-dimensional I/O resource description

4.2.1 Core idea about I/O conflict depiction

Applications often exhibit different, diverse and periodical resource behaviors. Many researches have devoted to this topic and proposed many efficient methods for periodical behaviors descriptions [9,10,16]. Timeslice [32] and Resource description [24] are common measures for periodical behavior depictions that scheduling policies usually use. However, constant timeslice [21,32] is less effective for large-scale servers with many co-runners that exhibit complicated periodical behaviors, and further leads to un-negligible scheduling overhead.

Under the scenario of co-running, the depictions of periodical behaviors should cover at least two important aspects to be more accurately and efficiently. First, the description should reflect the periodical behaviors for resource demands. The second is that the description should reflect the periodical performance effects during resource contentions.

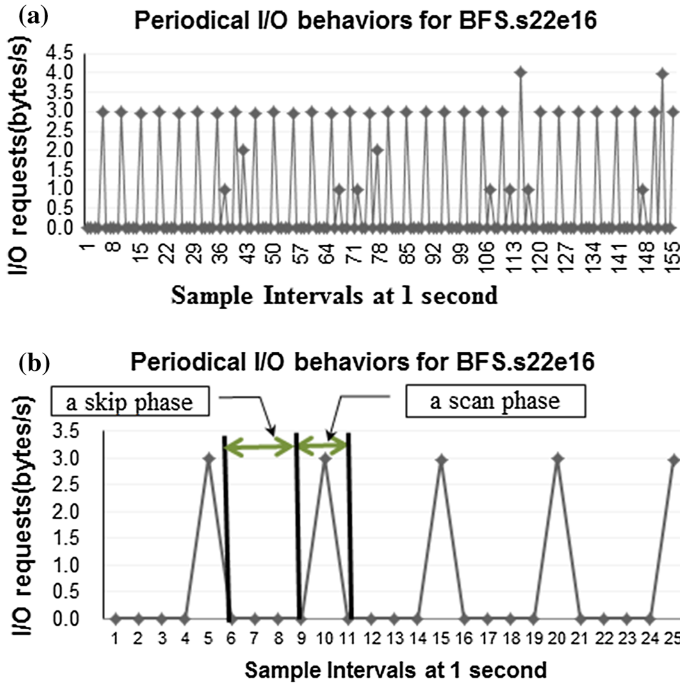


Fig. 14 A sample of periodical I/O behaviors. **a** A sample of periodical I/O behaviors. **b** Amplified fragment of the samples

4.2.1.1 Periodical behaviors for resource demands (dtimeslice) Figure 14a demonstrates the periodical behaviors for BFS.s22e16, which are sampled with *iostat* at interval of 1 s. More details about BFS.s22e16 are introduced in Sect. 5.1.2. The fragment of the samples in Fig. 14b shows that the task's execution process can be described with two phases: skip phase and scan phase.

A skip phase is composed of some continuous intervals in which the task keeps to be I/O free. Conversely, a scan phase is composed of some continuous intervals in which the task keeps on I/O busy.

However, it is not a desirable method to record all these phases to guide the dynamic scheduling. Especially, descriptions for independent execution and co-running execution are not the same at all. Therefore, WiseThrottling designs an adaptive property described with *dtimeslice* for the depiction of a task's periodical behaviors. It has an initial value and an adaptive value. The initial value of *dtimeslice* can be obtained through samples and analyses off-line, such as sampling with *iostat* (or periodical reading the system I/O files under /proc/pid/io). Then, this value keeps on self-tuning according to the real-time I/O behaviors during the dynamic execution.

4.2.1.2 Periodical performance during I/O contentions (sensitivity) Intuitively, different task exhibits different I/O behaviors and different sensitivity to I/O competitions. Our investigations in Sect. 3.2.2 have shown this phenomenon. An important conclu-

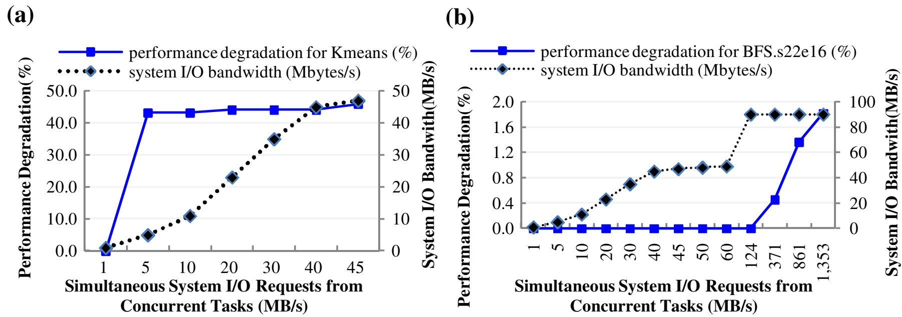


Fig. 15 Relation between I/O bandwidth demands and the sensitivity to I/O contentions for a task. **a** Sensitivity of Kmeans. **b** Sensitivity of BFS.s22e16 (color figure online)

sion from Sect. 3.2.2 is that tasks with higher I/O demands will suffer more significant performance degradation from I/O contentions. The following experiments are used to further disclose the relation between the average bandwidth demands and the sensitivity to I/O contentions in co-running cases.

Some studies adopt micro-benchmark to classify the priorities of applications [5]. In our work, we use expanded IO_RANKING experiments (as in Sect. 3.3.2) as detectors for the combined effects of a task's whole execution. In the experiments, we vary the number of nops and the number of CMPs to generate more IO_RANKING (as in Fig. 2) tasks. These IO_RANKING tasks in the expanded IO_RANKING experiments have different I/O bandwidth demands ranging from 1 to 1.3 GB/s. Through concurrent executions of a user task and other tasks that exhibit diverse I/O behaviors, we can observe the performance changes resulted from I/O competitions.

In Fig. 15, each figure demonstrates the co-running performance degradation of a user application in co-running with IO_RANKING tasks. The two figures in Fig. 15 also show a sharp contrast between the two cases. In Fig. 15a, the co-running performance degradation curve (blue) indicates that Kmeans is much easier to be interfered in I/O competition conditions, and thus suffers a 40 % performance lost in most cases. On the contrary, in Fig. 15b, as a result of memory optimization, I/O operations of BFS are greatly reduced. Therefore, the performance degradation curve of BFS (blue) shows no significant performance loss during the co-running execution. The different performances relate closely to the diversified I/O demands of these two applications. Kmeans has relatively higher I/O demands (about 37 MB/s) and BFS has low I/O demands (about 6 MB/s). The sort of tasks that have relatively higher I/O demands, e.g., Kmeans, is more sensitive to I/O contentions than that with lower I/O demands, and we conclude that the more the I/O demands are, the more performance lost the application has to suffer in co-running cases.

The above observations and the observations in Sect. 3.2.2 inspire us that, to mitigate I/O contentions in large-scale platforms, the sensitivity to I/O contention is a key factor which should not be neglected. Therefore, WiseThrottling uses an adaptive property described with *sensitivity* for the depiction of a task's sensitivity to I/O contentions. More details of the design about *dtimeslice* and *sensitivity* will be discussed in Sect. 4.2.2.

4.2.2 Two-dimensional I/O depiction

4.2.2.1 dtimeslice We design *dtimeslice* to indicate the periodical I/O behaviors for a task. WiseThrottling uses *dtimeslice* in the dynamic module to guide the inspection intervals for a task and to regulate I/O conflicts. As mentioned before, *dtimeslice* has an initial value and an adaptive value. The initial value is to outline the general I/O behavior statically, while the adaptive value is to describe periodical I/O behaviors more appropriately. Therefore, the major deference between these two values is that the initial value is calculated with the static information while the adaptive value is adjusted according to more real-time information within a short period. These two values cooperate closely to realize a wise scheduling. Using *dtimeslice*, the scheduling can be more efficient in dealing with contention cases and skip these intervals without I/O requests (the skip phase in Fig. 14b).

Initial value of dtimeslice

As the investigation in Sect. 4.2.1, considering the scan phases (Fig. 14b) that are busy with I/O, we set the initial value of dtimeslice as in Eq. (2).

$$dtimeslice_{initial} = \left(\frac{\sum_{i=0}^{SCAN_IDX} scan^i}{SCAN_IDX} \right) / IO_BUSY_THRESHOLD \quad (2)$$

In this equation, for a task, the $scan^i$ stands for the average I/O demands in the ith scan phase. *SCAN_IDX* is used to represent the number of the scan phase. Since I/O operations are much longer than memory operations, the interval for scheduling should not be so fine-grained in case that I/O operations are always interrupted before they have enough time to complete. Thus, *dtimeslice* for the task is set with a ratio of the average I/O bandwidth in scan phases, and *IO_BUSY_THRESHOLE* (It is an empirical value set according to specific hardware I/O characteristics). The higher the average I/O bandwidth, the longer is the inspection interval.

Figure 16a shows the detailed algorithm upon the initialization process of *dtimeslice* through ***Initial_Dtimeslice*** in the static module. For any task (denoted as *task[i]*), the module collects the number of the scan phases in I/O log file (generated by *iostat*). The entire process is off-line, and it includes five steps. (1) Deciding the current I/O status according to the number of I/O requests. We use a tunable constant IO*_BUSY_THRESHOLD* to identify busy status. In steps (2) to (4), our mechanism put the current state into appropriate records. In (2), if the current interval starts a new scan phase, it will be counted in a new scan phase record. In (3), if a scan phase is completed, the average I/O quantities for the phase will be calculated. In (4), we just increase the number of corresponding record if the current status is a continuation of the previous scan interval. In (5), finally, we calculate the *dtimeslice* with all the collected information.

Adaptive value of dtimeslice

Applications often exhibit irregular I/O behaviors, e.g., intensive I/O reading at the initialization and sparse I/O operations at the middle. To adapt to the variation of I/O behaviors, *dtimeslice* will increase or decrease accordingly during the entire scheduling processing in dynamic module.

(a)

```
Procedure Initial_Dtimeslice(int i)
    {
1:    io_log = open(iostat_log(task[i])) ; /* open the log from iostat */
2:    SCAN_IDX =0; /*Numbers of scan phases during execution*/
3:    Current_Status = 0; /*the I/O status of the current interval,
4:                            *0 is free and 1 is busy */
5:    Prev_Status = 0;  /* the I/O status of the previous interval */
6:    /* statistics for all scan phases for task[i] */
7:   int Accumulated_Continous_Intervals=0;
8:
9:    for ( each line in io_log) {
10:       /* get the I/O quantities for the current sample interval
11:        * from each line of the log file */
12:       io_quantities = read_io(line) ;
13:
14:       /* 1: decide whether the interval is busy or not */
15:       if ( io_quantities  > IO_BUSY_THRESHOLD)
16:          Current_Status =1;
17:       else
18:          Current_Status = 0;
19:
20:      /* decide whether the current interval has the
21:       * same status with previous intervals. If not,
22:       * a new interval shall be created */
23:       if (Prev_Status != Current_Status) {
24:           /* 2: decide whether  the current intervals
25:            * should be counted in a new scan
26:            * phase */
27:           if (Current_Status ==1) {
28:               scan[i] =io_quantities;
29:              Accumulated_Continous_Interval =1;
30:           }else {
31:               /* 3: if the current scan phase is completed,
32:                * calculate the average I/O quantities for
33:                * this phase */
34:              if (Accumulated_Continous_Interval !=0) {
35:                scan[SCAN_IDX]
36:                     /=Accumulated_Continous_Interval ;
37:               Accumulated_Continous_Interval =0;
38:               SCAN_IDX ++;
39:              }
40:           }
41:       }else {
42:           /* 4: if the current interval is a continuation of the
43:            * previous interval, just increase the number of the
44:            * corresponding record */
45:           if (Current_Status ==1) {
46:               scan[SCAN_IDX]     +=io_quantities;
47:               Accumulated_Continous_Interval ++;
48:           }
49:       }
50:       Prev_Status = Current_Status;
51:   }
52:   /*5: calculate the initial value of dtimeslice according to the
53:    * averaged skip number and scan number*/
54:   for (I =0; i< SCAN_IDX; i++)
55:       average_scan  +=   scan[i] ;
56:   }
57:       average_scan /= SCAN_IDX;
58:       task[i].dtimeslice =  average_scan / IO_BUSY_THRESHOLD;
59:       task[i].initial_dtimeslice = task[i].dtimeslice;
60: }
```

Fig. 16 The initialization and the adaptive tuning algorithms for dtimeslice. **a** Initialization algorithm for *dtimeslice* in the static module. **b** The adaptive self-adjustment algorithm for *dtimeslice* in the dynamic module

(b)

```
Procedure SelfTuning_Adaptive_Dtimeslice
   {
1:    for each task[i] {
2:        if (! Is_Quanlified(task[i].dtimeslice)) continue;
3:
4:        /*1: check the current I/O requests of task[i]
5:         * determine the status according to the requests
6:         */
7:        io_request = read_io(device_io_file(task[i]));
8:         if ( io_request  > IO_BUSY_THRESHOLD)
9:             Current_Status =1;
10:        else
11:            Current_Status = 0;
12:
13:        /* decide whether the dtimeslice should be shortened
14:         * or be extended according to the accumulated
15:         * continous  status */
16:        if (Current_Status == 0) {
17:           Continuous_Busy_Phase =0;
18:           Continuous_Idle_Phase ++;
19:            if (Continuous_Idle_Phase >= TURNABLE^ext) {
20:
21:              /* 2:extension period : if the accumulated idle
22:               * phases are bigger than TURNABLE^ext, then the
23:               * dtimeslice can be enlarged X times. */
24:               task[i]. dtimeslice *=X;
25:            }
26:        }else {
27:              Continuous_Idle_Phase =0;
28:              Continuous_Busy_Phase ++;
29:              if (Continuous_Busy_Phase >= TURNABLE^sht){
30:
31:                /* 3: shortening period: if the accumulated
32:                 * idle phases are bigger than  TURNABLE^sht,
33:                 * then the dtimeslice can be enlarged X times. */
34:                 task[i]. dtimeslice /=X;
35:                 if (task[i]. dtimeslice <task[i].initial_dtimeslice )
36:                    task[i]. dtimeslice = task[i].initial_dtimeslice  ;
37:              }
38:        }
39:    }
40: }
```

Fig. 16 continued

In Fig. 16b, ***SelfTuning_Adaptive_Dtimeslice*** in the dynamic module deals with the adaptive self-tuning process. As in line 19, we use *TURABLE* as a threshold to determine the variation of *dtimeslice*. $TURABLE^{ext}$ stands for the maximum successive intervals that allow none I/O operations for a task. Once a task has no I/O demands

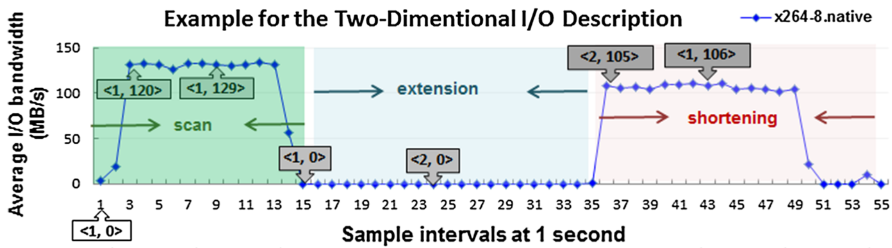


Fig. 17 An example for I/O description

and has not been regulated for successive $TURABLE^{ext}$ intervals, its *dtimeslice* will enlarge X times (e.g., 2 times in our work) as in line 24. We refer to this process as ***extension***. On the contrary, if a task keeps on I/O intensive for $TURABLE^{sht}$ intervals, it *dtimeslice* will decrease and finally return to the initialization value, as in line 34. We refer to this process as ***shortening***. The values of TURABLE in extension and shortening process, denoted with $TURABLE^{ext}$ and $TURABLE^{sht}$, are not necessarily the same. This variation either leads to a sparse scheduling interval or an intensive scheduling interval for a task, which can make a better trade-off between lower scheduling overheads and more accurate scheduling automatically.

4.2.2.2 Sensitivity ***Sensitivity*** describes a task's performance vulnerability to I/O contentions. The key issue is to estimate the potential performance influences from I/O contentions for a task. Since this property only plays a role when mitigating conflicts in real time, we only focus on the adaptive self-tuning process for it. The ***Initial_Sensitivity*** in Fig. 11 in the static module initializes *sensitivity* of a task to be zero.

Adaptive value of sensitivity

The adaptive value of sensitivity for a task will be updated in terms of I/O demands at different intervals through ***SelfTuning_Adaptive_Sensitivity***. As shown in Fig. 18a, this interface is embedded in the first step of ***Candidate_Queue_Setup*** in the dynamic module. In practice, higher sensitive tasks will be optimized firstly.

4.2.2.3 An example for I/O description In this subsection, we will illustrate how to determine the I/O descriptions for a task. Figure 17 demonstrates the process from initial value to the adaptive value for x264-8.native (please refer to Sect. 5.1.2 for more details about the applications). The whole execution takes about 55 s, as the x-axis denotes (55 intervals). The y-axis shows the corresponding I/O demands at each interval. Comment boxes are used to show the adaptive tuning process for I/O descriptions.

In the figure, the task goes through three successive periods, which are the scan period, the extension period and the shortening period. As the comment box of $< 1, 0 >$ at timeslice 1 shows, the static module initializes the property of *dtimeslice* to 1, and *sensitivity* to 0. From the first timeslice, the task enters the scan period and it keeps I/O busy from timeslice 1 to timeslice 15. In this period, *dtimeslice* remains intact. Then, the task keeps I/O idle, and thus enters the extension period

from timeslice 15 to timeslice 35 as the figure shows. During this period, when the task has accumulated $TURABLE^{ext}$ ($TURABLE^{ext} = 10$) continuous idle intervals from timeslice 15, the *dtimeslice* is enlarged X (here X is 2) times at timeslice 24, and turns into 2. After that, x264 will be inspected every 2 s. Finally, x264 becomes I/O busy once again with file writeback operation from timeslice 36. From that point on, *dtimelsice* will go through a shortening period. After 5 busy intervals are accumulated (here, $TURABLE^{sht} = 5$), *dtimeslice* will shorten X times at timeslice 44, and return back to the initial value of 1, while *sensitivity* keeps changing with the real-time I/O quantities.

To sum up, WiseThrottling uses a two-dimensional I/O description for each task. Two properties (*dtimeslice* and *sensitivity*) in the I/O resource description need to be initialized in the static module of WiseThrottling. These values are useful for guiding the scheduling at the first beginning of the later dynamic process. After that, these properties will be maintained dynamically according to the real-time I/O demands in the dynamic modules of WiseThrottling.

4.3 WiseThrottling scheduling policy

WiseThrottling is composed of two modules, the off-line static module and the dynamic module. These two modules work collaboratively to make the scheduling more efficiently and accurately.

4.3.1 Static module

As shown in Fig. 11, the major function of the static module is to collect I/O behaviors of user tasks and output the I/O resource description vectors for them in an off-line way. It includes two algorithms to initialize the two properties in I/O descriptions. More details about the initialization are presented in Sect. 4.2.2. The output resource descriptions for tasks will work as inputs of the dynamic module of WiseThrottling.

4.3.2 Dynamic module

The dynamic module plays a more important role for WiseThrottling. It implements the asynchronous scheduling policy to reduce I/O conflicts, and adjusts I/O descriptions during the real-time scheduling.

It should be pointed out that not all tasks could be updated and regulated in each interval. To control the software overhead, the dynamic module only inspects tasks that are quantified with regard to *dtimeslice* because only these tasks will result in I/O conflicts and should be regulated. This is the principle of the dynamic module.

The dynamic module undergoes three stages to fulfill its duty. Figure 18 illustrates the three major algorithms in the dynamic module.

Candidate_Queue_Setup: The first stage in the dynamic module is to create a candidate queue. As illustrated in Fig. 18a, I/O busy tasks at the current interval will enter a candidate queue. Here the busy status is determined by *IO_BUSY_THRESHOLD* (as introduced in Sect. 4.2.2.1). With the real-time I/O information, the property of sensi-

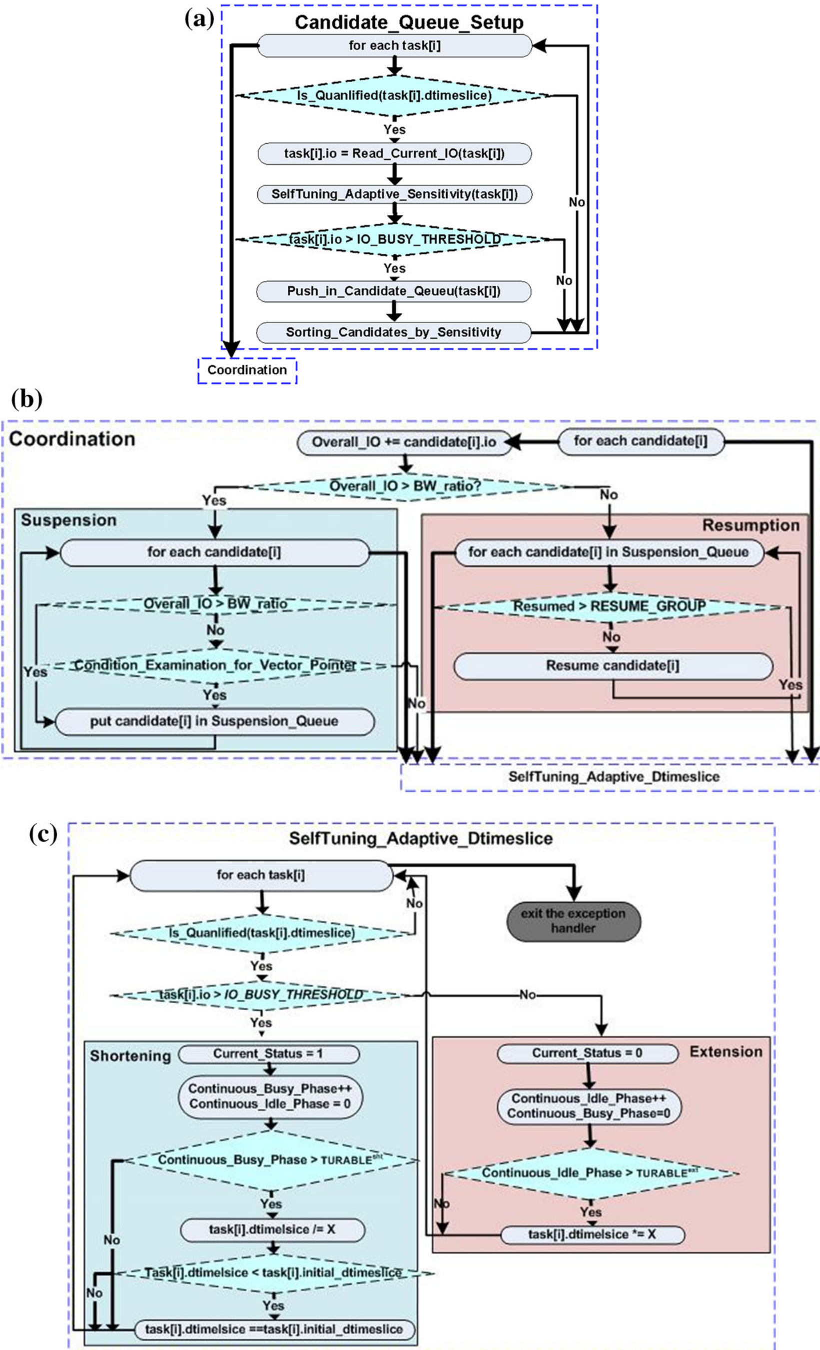


Fig. 18 The dynamic module of WiseThrottling. **a** ***Candidate_Queue_Setup*** is to put qualified tasks in the candidate queue. **b** ***Coordination*** is to detect and regulate I/O contentions. **c** ***SelfTuning_Adaptive_Dtimeslice*** is to adjust *dtimeslice* for all tasks according to their real-time I/O demands

tivity can be updated through ***SelfTuning_Adaptive_Sensitivity***. Tasks in the candidate queue are in descending order in terms of sensitivities. No matter for multi-threaded tasks or multi-programmed tasks, children will have their own I/O information. This makes it possible to mitigate both inter-task and intra-task conflicts in the following steps.

Coordination: After the candidate queue is updated, ***Coordination*** in the dynamic module begins to regulate I/O conflicts in the following steps.

Firstly, the algorithm needs to inspect the current conflicts and decide on whether regulation is needed for the current interval. Two metrics in Sect. 3.4 are used to determine the necessity of the scheduling:

(a) BW_Ratio: Our analyses in Sect. 3.4.2 show that once the total I/O demands approach around 50 % during multi-task competitions, potential performance degradation will occur. We use *BW_Ratio* of 50 % in our work as a heuristics for scheduling. It is an empirical value.
(b) Vector Points: Besides the condition of *BW_Ratio*, we use Vector Points as subsidiary conditions to predict potential conflicts. We obtain the Vector Points via the fine-grain experiments with IO_RANKING (as in Sect. 3.2.1). Each Vector Point corresponds to a condition examination. The dynamic module will go through all condition examinations one by one in order to detect the potential performance influences during competitions.

Secondly, two types of decisions will be made in this procedure, suspension and resumption. As in the left branch of Fig. 18b, after conflict detections, if the BW_Ratio or one of the condition examinations fails to be satisfied, the dynamic module will keep on suspending some tasks or some threads of a task in one or more CMP. On the contrary, as in the right branch of Fig. 18b, if the current I/O demands are below BW_Ratio, the dynamic module will resume some suspended tasks. Those who have been suspended for a longer time are preferred during the process of resumption. In case of the oversubscription of I/O bandwidth, we use a tunable constant, *RESUME_GROUP*, e.g., 1/4 of the co-running tasks, to control the number of the tasks that are resumed in the interval.

In summary, the dynamic module checks the current I/O contentions, and makes decisions to regulate I/O conflicts. If the contentions are heavy, some tasks will be suspended. On the other hand, if the resource contentions are not severe, some suspended tasks will be resumed.

4.4 An instance of WiseThrottling in reality

Figure 19 uses a four-task workload as an example to illustrate the routine behind WiseThrottling in detail. In the figure, Task_0 and Task_1 are single-threaded. Task_2 and Task_3 are dual-threaded. Each thread is mapped to a specific core, and there is no core-sharing between any of the two tasks. All tasks except Task_3 are I/O intensive applications. Task_3 is less I/O intensive. We use the four Vector Points in Fig. 10 and BW_Ratio of 50 % to set up the four examination windows in our example.

The initialization values of the I/O resource description are given in the left of the figure. The x-axis in the figure denotes the timeslice during execution. The comment

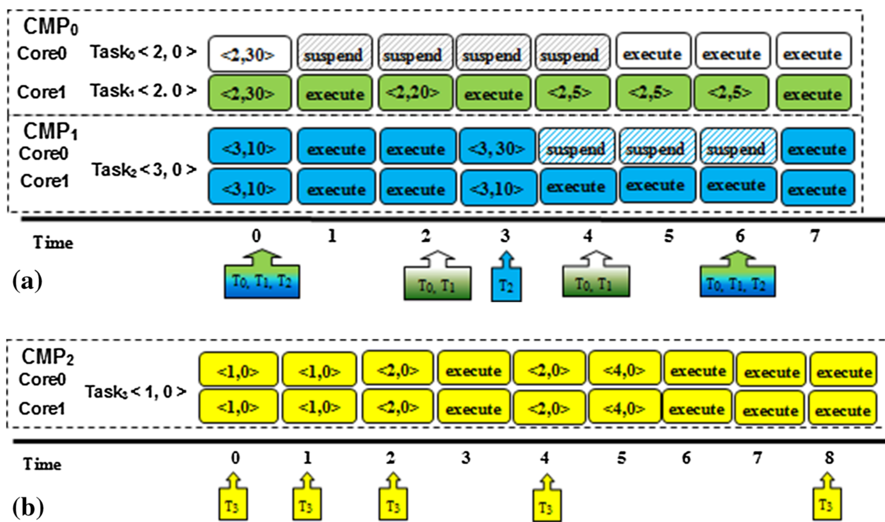


Fig. 19 WiseThrottling scheduling example. Task_0, Task_1, Task_2 and Task_3 are co-located on different cores in CMP_0-CMP_2. The two-dimensional resource description for each task is <dtimeslice, sensitivity>. The *comment boxes* in *different color* at the bottom illustrate timeslice when a task should be checked (in short, T_i), which are indicated by the tasks' adaptive *dtimeslice*. **a** The asynchronous scheduling of WiseThrottling. **b** The adaptive self-adjustment of the I/O description (color figure online)

boxes at the bottom represent the qualified interval (in short, T_i) for a task which should be examined. These are specified by the initial values of tasks' *dtimeslice*. The initial *dtimeslice* for each task specifies that the qualified inspection timeslice for Task_0, Task_1 is on 0, 2, 4, 6, and for Task_2 is on 0, 3, 6. Guided by these non-uniform values, WiseThrottling will initiate asynchronous scheduling and examine a task at its specified intervals. These *dtimeslice* values keep on self-adjusting during the dynamic running, which are reflected through the variation of values in the rectangle for each task.

Figure 19a illustrates the ***Candidate_Queue_Setup*** and ***Coordination*** steps of the asynchronous scheduling. In the figure, at timeslice 0, only three tasks, Task_0, Task_1 and Task_2, are qualified for the candidates because resource description indicates that timeslice 0 is a common check point for all of them. WiseThrottling only checks these 3 tasks and the dynamic module fails to satisfy both the *BW_Ratio* and the condition examinations for Vector Point <24.3, 2>. Thus, WiseThrottling decides to suspend Task_0 due to its higher *sensitivity* (resource description shows Task_0's *sensitivity* at this interval is 30), and let Task_1 and Task_2 continue executing. At timeslice 2, WiseThrottling only checks Task_1 (because resource description indicates that timeslice 2 is a check point for Task_0 and Task_1). Since there is only one task, Task_1, in the candidate queue, WiseThrottling does nothing and Task_1 need not be suspended. At timeslice 3, only Task_2 is checked (because resource description indicates timeslice 3 is the check point for it). WiseThrottling is aware of that *BW_Ratio* still cannot be catered after Task_0 is suspended. Therefore, one thread of Task_2 is suspended by WiseThrottling. At this moment, Task_0 and one thread of Task_2 are both suspended, and Task_1 and the

other thread of $Task_2$ are left running. At timeslice 4, $Task_1$ is checked since $Task_1$ is still in suspension. WiseThrottling is aware of that the two conflict detections are luckily fulfilled, so it resumes $Task_0$ at this moment (because $Task_0$ is the longest one which has been suspended). At timeslice 6, one of the threads of $Task_2$ is resumed by WiseThrottling finally. The above steps show the general process of WiseThrottling, and the essence of the whole process is the two-dimensional resource description, which obtained from static module.

Figure 19b illustrates the step of ***SelfTuning_Adaptive_Dtimeslice***. In the figure, we use $Task_3$ to demonstrate how *dtimeslice* adapts to the I/O behaviors dynamically. For brevity, we set $TURABLE^{ext}$ to 2 and X to 2. At timeslice 0, WiseThrottling checks $Task_3$ (because its resource description specifies the initialization value of *dtimeslice* is 1). Since it has no I/O demands, the scheduler does nothing and just moves on. At timeslice 1, WiseThrottling still checks $Task_3$, and finds that it still has no I/O demands. At this moment, $Task_3$ has accumulated 2 successive idle intervals and reaches the upper bound specified by $TURABLE^{ext}$ value. Its *dtimeslice* enlarges 2 times and is updated to 2 due to our adaptive strategy. Thus, the next check points for $Task_3$ are timeslice 2 and 4. From timeslice 2 to timeslice 4, $Task_3$accumulates 2 successive idle intervals once again, and its *dtimeslice* will be extended to 4. With this automatic tuning process, the dynamic module will skip other intervals to reduce the overhead of unnecessary examination for $Task_3$.

5 Experiments and evaluations

In this section, we evaluate the performance of WiseThrottling from two aspects:

(1) We evaluate and examine the effectiveness of WiseThrottling via two sorts of typical services:

- Similar-pattern service: This kind of service is made up of duplicated applications, while either same or different data input sets.
- Compounded-pattern service: This kind of service is composed of randomly selected tasks from the 7 applications (in Table 2).

(2) We evaluate the scalability of WiseThrottling with batch-mode service, which is a common service type in modern datacenter.

- Batch-mode service: The key feature of batch-mode service includes the dynamic participation of new tasks and dynamic extinction of old tasks.

During the batch-mode serving process, concurrent running tasks will not exceed the socket number at any moment. Whenever a task is completed and its socket turns into idle, a new task will start to execute.

Table 2 Applications drawn from user applications on Dawning Cluster

Application	Type	I/O implementation	Abbr.	Description
Parallel building	8-programmed	Explicit	PB	Each application is an 8-programmed building process for "open64" compiler with specified optimization flags
Paper similar	8-programmed	Explicit	PS	A concurrently similarity checking program. Each application compares a paper with the other K papers, while K is 8 in our work
Kmeans clustering	8-threaded	Explicit	KM	A key algorithm from data mining and currently be widely applied in web applications. The program partitions n observations into k clusters
BFS tree traversal	8-threaded	Implicit	BFS	BFS algorithm from Graph500. The graph for searching is generated with two parameters, s and e (denoted with BFS.sMeN, M and N are values of s and e), which stand for a graph's scale and edge factor, respectively
X264	8-threaded	Explicit	X264	An encoder program from PARSEC 3.0
Swaptions	8-threaded	Explicit	SW	An application of pricing a portfolio of swaptions from PARSEC 3.0
Wordcount	Mapreduce	Explicit	WD	A map/reduce program that counts the words in the input files

5.1 Platform and workloads

5.1.1 Platform

The evaluation for WiseThrottling is set up in Dawning Cluster. Currently, Dawning Cluster datacenter has more than 56 nodes ranging from 8-core to 64-core. We perform our experiments on an 8-socket 64-core Intel® Xeon® E7-8830 server. The OS is Linux CentOS 6.3 with kernel 2.6.32. The server has 256 GB memory in total.

5.1.2 Workloads

A notable factor is that diversified I/O interfaces correspond to different I/O implementation mechanisms. And the different implementations make the forms of I/O contentions different. We take this factor into account. From this perspective, we classify I/O operations into the following two types:

- ***Explicit I/O***: It is caused by the usage of API such as ***fread*** and ***fwrite***, which contends for I/O related resources and as a result suffers from I/O conflicts.
- ***Implicit I/O***: Different from explicit I/O, implicit I/O is incurred by the memory-associated file operations (e.g., ***mmap***), imposing a high pressure on main memory. Thus, swapping is usually involved in these operations, leading to I/O contentions. This is so-called implicit I/O.

We use 7 applications to generate more than 20 workloads in our experiments.

- MapReduce application: One MapReduce application is wordcount from Hadoop-1.1.2 package [38]. It suffers from explicit I/O contentions between the concurrent mappers.
- Real application: Three applications are real programs from regular users in Dawning Cluster, which are parallel building compiler, paper similarity examination and Kmeans clustering algorithm.
- Benchmark application: Three of the applications are BFS graph traversal algorithm from graph500 [39], X264 encoder and Swaptions from PARSEC 3.0 [40]. These applications are used to represent some big data processing applications, which we could not obtain due to privacy. Swaptions is a less I/O sensitive application. We include it in Sect. 5 for evaluations on workloads mixed with both I/O sensitive and I/O insensitive applications.

Detailed information for these applications is listed in Table 2. For brevity, we use the abbreviations for each application in later sections and figures. In this section, a workload is composed with either multiple multi-programmed tasks or multi-threaded tasks. A task is either an explicit I/O or an implicit I/O application.

5.2 Experimental metrics

Three metrics are used in our evaluations. Firstly, we use ***Weighted Speedup*** (***WS***) [33] to measure the overall system performance improvements brought by WiseThrottling.

For either a multi-programmed task or a multi-threaded task, $Task_i$,

$runtime^i_{alone}$ is the execution time of $Task_i$ when it runs alone, i.e., without any resource contention.

$runtime^i_{shared}$ is the execution time of $Task_i$ when it co-runs with other task on the same CMP server.

$runtime_{workload}$ is the execution time of the whole workload.

With the above definitions, WS for an N-task workload is calculated with the following equation:

$$WS = \sum_{i=1}^{N} \frac{runtime^i_{alone}}{runtime^i_{shared}} \tag{3}$$

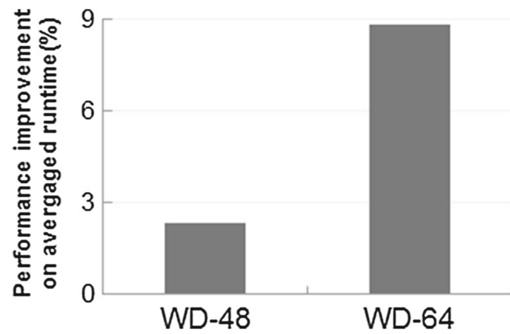


Fig. 20 Evaluations for MapReduce application of wordcount. Each application includes 48 mappers or 64 mappers

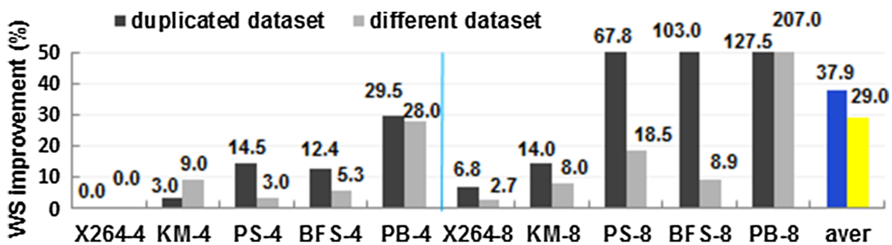


Fig. 21 Evaluations for similar-pattern service. Each workload contains a duplicated data set testing in *dark gray bar* and a different data set setting in *light gray bar*, respectively. The *blue line* represents a "watershed" for application-8 workloads and application-4 workloads (color figure online)

The second metrics is ***throughput (TP)***. We use TP for batch-mode service aside from WS. TP for an N-task workload is calculated as in Eq. (4):

$$TP = \frac{N}{runtime_{workload}} \tag{4}$$

For MapReduce workloads, we use average runtime to measure the optimization effects of WiseThrottling.

All evaluations in our work are made between the default OS scheduler (cfs task scheduler + cfq I/O scheduler) with and without optimization of WiseThrottling.

5.3 Evaluations

5.3.1 Evaluations for similar-/compounded-pattern service

In this section, WiseThrottling is evaluated across more than 10 workloads of similar-pattern service which are denoted in the x-axis of Figs. 20 and 21.

In Fig. 20, the MapReduce application of wordcount are experimented with 48 mappers and 64 mappers, respectively, that are named with WD-N (N stands for the number of mappers). The performance improvement on average runtime is evaluated for wordcount in Fig. 20. WiseThrottling performs intra-task scheduling between

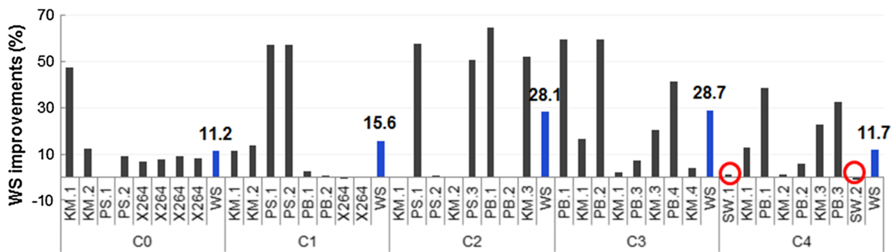


Fig. 22 Evaluations for compounded-mode services, ranging from C0 to C4. Each *gray bar* stands for the WS improvement for a task in a workload. The *blue bar* at the end stands for the WS improvement for the workload (color figure online)

the 48 mappers or 64 mappers, and improves the performance by 2.3 and 8.8 %, respectively.

In Fig. 21, each workload contains 4 tasks or 8 tasks that are named with "application-N" (N stands for the number of concurrent tasks). The WS improvements across the 10 workloads in the figure illuminate the effectiveness of WiseThrottling for similar-pattern services. With duplicated input data sets, WiseThrottling improves the overall system performance by 37.9 % on average across all workloads. With different input sets, it can achieve 29.0 % on these workloads. WiseThrottling boosts 8-task workloads much more than 4-task workloads on average. And the maximum improvement of 8-task workload achieves 127.5/207.0 % in duplicated/different data set separately, while the maximum improvement of 4-task workload is 29.5/28.0 %.

The potential optimization space of WiseThrottling closely relates to the size of working data set, which determines the amount of I/O demands. For example, PS-8 processes similarity examination for 8 files concurrently while KM-8 only handles one file. Therefore, PS-8 obtains much more favorable improvements from WiseThrottling than KM-8, which is 67.8 %/18.5 % vs 14.0/8.0 %, respectively, for duplicate data set and different data set. Similarly, it is reasonable that WiseThrottling performs much better in 8-task condition than that in 4-task condition, because WiseThrottling is able to potentially reduce more I/O contentions in 8-task case. And Fig. 21 also proves that WiseThrottling is to work well in large-scale multi-/many-core platforms, which would inevitably generate more I/O contentions.

WiseThrottling is also evaluated across 5 compounded workloads, which are denoted by C0–C4 in the x-axis of Fig. 22. Each workload is composed of 8 tasks, which are randomly generated from the 7 applications (in Table 2). X264 uses native data set. Other tasks adopt different data set in each workload. We use "Application.N" for an application and N stands for a different data set indexes. For compounded-pattern service, shown in Fig. 22, WiseThrottling improves the overall system performance, ranging from 11.2 to 28.7 %. C4 is a workload, which is mixed with both I/O sensitive applications, such as X264, KM and PB, and I/O insensitive applications, such as SW (circled in red). We can see that the heuristic of WiseThrottling can ensure the performance of non-I/O intensive applications.

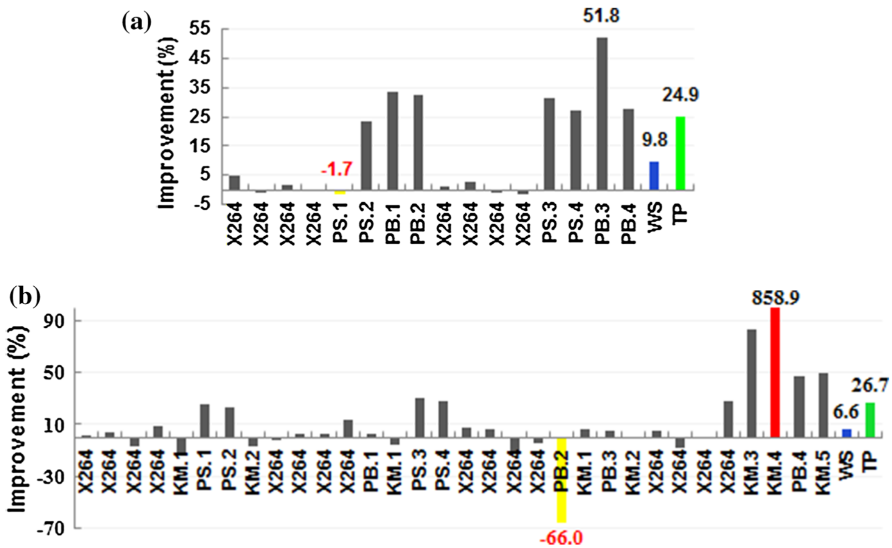


Fig. 23 Evaluations for batch-mode service. Each *bar* represents the WS improvements for every task. The *blue WS bar* stands for the WS improvement for the workload. The *green TP bar* represents the TP improvement for the workload. **a** Evaluations for 16-task batch-mode workload. **b** Evaluations for 32-task batch-mode workload (color figure online)

5.3.2 Evaluations for batch-mode service

WiseThrottling is evaluated across two workloads of batch-mode service. The experimental results are shown in Fig. 23. Figure 23a illuminates detailed WS improvement for a 16-task workload (denoted in the x-axis of the figure). The WS improvement for these tasks ranges from −1.7 to 51.8 %. The WS improvement for this workload is 9.8 %, and the TP improvement is 24.9 %. In this workload, the majority tasks benefit from WiseThrottling, though some tasks suffer a very trivial degradation. In Fig. 23b, the WS improvement for a 32-task workload is 6.6 % and TP improvement is 26.7 %.

WiseThrottling performs asynchronous and self-regulating scheduling for concurrent tasks. It brings performance improvements through taking different I/O characteristics and sensitivities of co-runners into its consideration. Somehow, it may focus on those who are more sensitive to I/O contentions. This may result in some unfair phenomena as illustrated in Fig. 23. As shown in Fig. 23a, PS.1 suffers a trivial degradation by 1.7 %. In Fig. 23b, it shows an obvious difference on WS improvements across these tasks (denoted in the x-axis of the figure). Although the workload could achieve WS improvement by 6.6 % and TP improvement by 26.7 %, we also notice that the worst case (PB.2 in yellow bar) suffers a serious degradation by 66.0 % (the best case (KM.4 in red bar) benefits a significant improvement by 858.9 %). WiseThrottling seems not fair enough for all the tasks even in the same workload. Our future work will study it more deeply and address the unfairness problem.

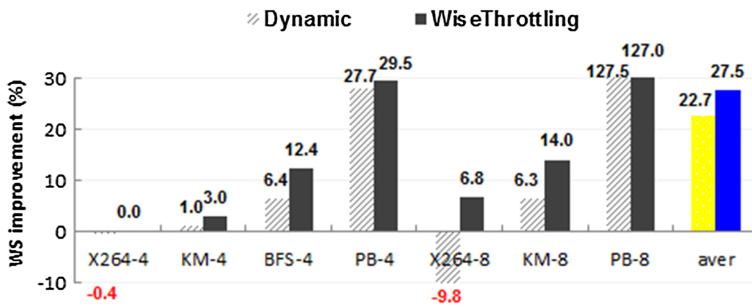


Fig. 24 Contrasts between the dynamic scheduler and WiseThrottling

5.3.3 Contrasts between preliminary work

The major difference between WiseThrottling and the dynamic scheduler in our preliminary work [21] is the self-adjustment of dtimeslice and sensitivities. The self-adjustment can reduce unnecessary examinations and make the scheduling more exact.

Figure 24 contrasts the different optimization effects between the dynamic scheduler and WiseThrottling. In the figure, the self-adjustment of I/O resource descriptions makes WiseThrottling outperform the dynamic scheduler on workloads of x264, KM, and BFS. These workloads only suffer from partial-time I/O contentions, i.e., I/O contentions only happen during file reading at the first beginning. For this kind of workloads, the fixed scheduling intervals and parameters of the dynamic scheduler will lead to unbalanced regulation effects. It may benefit first during I/O intensive phases of file reading, while lead to interferences during non-I/O intensive phases when the file reading operation finish. On the contrary, for WiseThrottling, the self-adjustment of dtimeslice and sensitivities can automatically extend and shorten the inspection intervals, thus unnecessary interferences for co-running workloads can be inhibited effectively. Moreover, the self-adjustment of I/O resource descriptions can reflect I/O demands more exactly, which makes the scheduling more precise.

5.4 Analysis

The goal of WiseThrottling is to reduce the negative impacts of I/O conflicts through coordinating I/O behaviors and throttling amount of I/O requests. In this section, detailed analysis is made via iostat.

5.4.1 Correlation between memory utilization and WiseThrottling optimization

Performance degradation is caused by different I/O types through different ways. Most workloads with explicit I/O such as application PS, etc., always show obvious performance degradation. The performance degradation relates much to the severity of I/O contentions. On the contrast, the workloads with implicit I/O applications such as BFS are much more different. Their performance degradation correlates with the fast-growing memory utilization (e.g., via usage of mmap). Implicit I/O contention

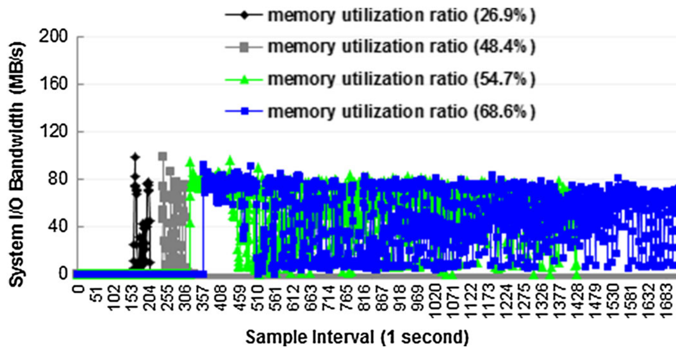


Fig. 25 Correlation between memory utilization ratio and the severity of I/O contentions [21]

that can potentially be mitigated by WiseThrottling has its root in memory cost. (1) Memory initialization costs: the allocation and initialization for memory space can lead to I/O bursts. And when the concurrently running tasks initiate together, the intensive I/O bursts will degrade the system performance. (2) Memory swapping costs: optimizations such as mmap may incur the usage of memory swapping, destructing the co-running performance severely.

The relation between the memory utilization ratio and the severity of I/O contentions for implicit I/O applications has been studied in our previous work [21]. In Fig. 25 [21], the experiments are implemented with BFS-8 and different graphs (generated with different s and e as in Table 2). Through varying the value of s and e, different graphs can be generated, which would result in different memory sizes for file operations. In the figure, the memory utilization ratio ranges from 26.9 to 68.6 % (the system memory is 256 G in total). As can be seen from the figure, if the memory utilization is just 48.4 % (or less), the I/O contention period is very short, which indicates that the overall system performance is not impacted by I/O contention seriously. Nevertheless, on the contrast, when the memory utilization reaches 68.6 %, the system performance has to suffer the I/O contentions during nearly 3/4 of our sampling period (30,000 s). This significant difference is caused by I/O swapping for each job and the corresponding I/O contentions. We have provided more details about this correlation in our previous paper [21].

Our further experiment with BFS correlates the memory utilization with the performance improvement brought by WiseThrottling. As displayed in Fig. 26, with the increasing memory utilization ($26.9 - 77.1\%$), the I/O contentions become more serious, causing more severe performance degradation (denoted in dark columns). However, using our approach, WiseThrottling can offset these negative impacts. Through carefully scheduling, WiseThrottling can avoid the I/O conflicts effectively and shorten the I/O contention period (in Fig. 5) dramatically. As seen from the blue curve in Fig. 26, it thus improves the overall system performance.

5.4.2 Essence of WiseThrottling optimization

To show how the reduced software service time contributes to the improvement of system performance, we conduct the following experiments (shown in Fig. 27a, b).

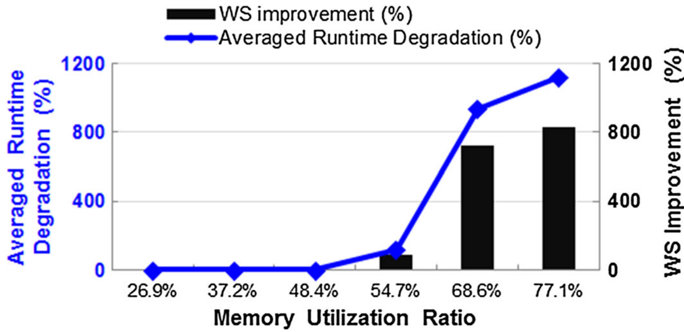


Fig. 26 Correlation between memory utilization ratio and the effectiveness of WiseThrottling

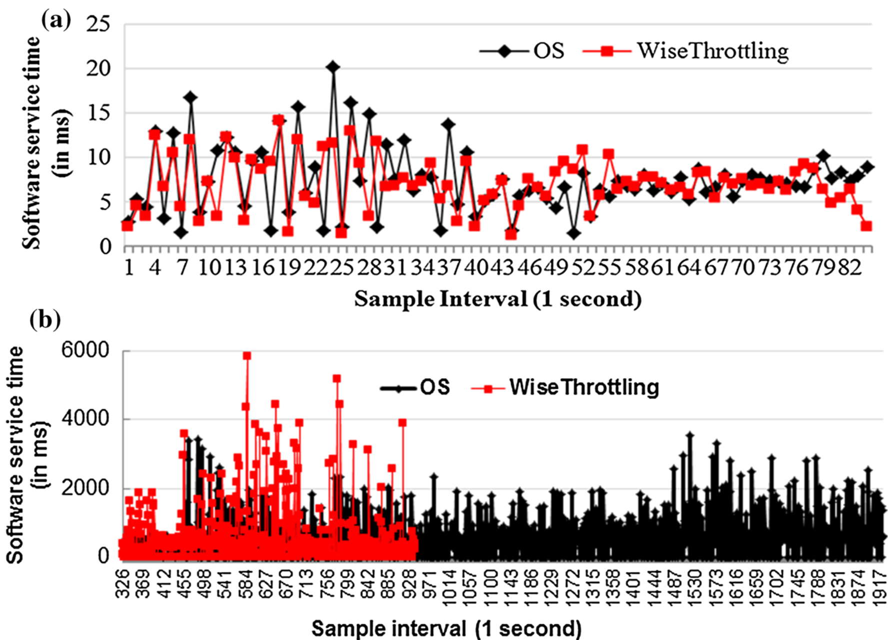


Fig. 27 Sample result of software service time with *iostat* at 1 s. **a** Samples of software service time for the experiment of Kmeans-4 with duplicated data. **b** Samples of software service time for the experiment of BFS-8.s25e16 with different data

We use two workloads, KM-4 and BFS-8 from similar-pattern service, for detailed studies. The reasons why we use them are (1) the two workloads stand for different degree of I/O contentions, and they could obtain different performance benefits from WiseThrottling; (2) they represent two different I/O implementation mechanisms (explicit and implicit I/O).

Figure 27a shows the comparison between the software service time of default OS scheduler and WiseThrottling. The workload is KM-4 with duplicated data set, which has less intensive I/O competitions. We observe that for most of the sample

intervals, WiseThrottling is able to reduce the software service time, contributing to the system performance. Figure 27b shows the same comparison. But the workload is BFS-8 with a different data set, and is selected as a representative for implicit I/O contentions. Figure 27b reveals the essence why WiseThrottling is able to improve system performance. As shown in this figure, using WiseThrottling (represented by red curve), the software service time (dominate factor in I/O latency) is reduced by more than half compared with the default OS scheduler. Same to the above experiments, we also sample the entire execution process. This is root of our WiseThrottling solution, which reduces the software service time and thus contributes to the improvement of the overall system performance significantly.

6 Conclusions and future work

We conclude that although the CMP platforms can benefit the increasingly complex computing requirements, they pose new challenges on the resource utility and management (such as I/O problem). Some prior work in this area mainly focus on how to improve the I/O hardware speed to meet the CPU requirement, but the software layer (the default OS scheduling) plays a more critical role in modern systems. Our work takes an important step in analyzing the performance of software scheduler, and shows that it has been a new performance bottleneck in large-scale CMP systems. Hence, we propose WiseThrottling (a user-level software component) to deal with the I/O problem. WiseThrottling performs asynchronous and self-regulating scheduling for concurrent tasks. It achieves a wise scheduling through taking different I/O characteristics and sensitivities of co-runners into its consideration. In our future work, we will apply WiseThrottling in more datacenter environments, and further optimize the scheduling process.

References

1. Alvarez GA, Chambliss DD, Jadav D et al (2009) Utilizing informed throttling to guarantee quality of service to I/O streams. US Patent, Google Patents
2. Armbrust M, Fox A, Griffith R et al (2009) Above the clouds: a berkeley view of cloud computing. Technical Report UCB/EECS-2009-28
3. Barroso L, Holzle U (2007) The case for energy-proportional computing. IEEE Comput 40(12):33–37
4. Bienia C (2011) Benchmarking Modern Multiprocessors. Princeton University. http://parsec.cs.princeton.edu/publications/bienia11benchmarking.pdf
5. Boneti C, Cazorla FJ, Gioiosa R, Buyuktosunoglu A, Cher C-Y, Valero M (2008) Software-controlled priority characterization of POWER5 processor. In: Proceedings of the 35th international symposium on computer architecture, June 21–25, pp 415–426
6. Bordawekar R, Rosario JM, Choudhary AN (1993) Design and evaluation of primitives for parallel I/O. In: Proceedings of SC'93, pp 452–461
7. Ching A, Choudhary A, Coloma K, Liao WK, Ross R, Gropp W (2003) Noncontiguous access through MPI-IO. In: Proceedings of CCGrid'03. pp 104–111
8. Das R, Ausavarungnirun R, Mutlu O, Kumar A et al (Feb 2013) Application-to-core mapping policies to reduce memory interference in multi-core systems. In: Proceedings of PACT'13
9. Dhodapkar A, Smith J (2003) Comparing program phase detection techniques [C]. In: Proceedings of the 36th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, Los Alamitos, pp 217–217

10. Ding C, Dwarkadas S, Huang MC et al (2006) Program phase detection and exploitation. In: Proceedings of the 20th international conference on parallel and distributed processing. IEEE Computer Society, Los Alamitos, pp 279–279
11. Durand D, Jain R, Tseytlin D et al (2003) Parallel I/O scheduling using randomized, distributed edge coloring algorithms. J Parallel Distrib Comput 63(6):611–618
12. Govindan S, Nath AR, Das A et al (2007) Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms. In: Proceedings of VEE'07, pp 126–136
13. Hastings A, Choudhary A (Sep 2006) Exploiting shared memory to improve parallel i/o performance. In: EuroPVM/MPI'06, pp 212–221
14. Jain R, Somalwar K, Werth J et al (1992) Scheduling parallel I/O operations in multiple-bus systems. IEEE Trans Parallel Distrib Syst 16(4):352–362
15. Jain R, Somalwar K, Werth J et al (1997) Heuristics for scheduling I/O operations. IEEE Trans Parallel Distrib Syst 8(3):310–320
16. Jiang Y, Tian K, Shen X (2010) Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In: Proceedings of the 5th international conference on high performance embedded architectures and compilers. Springer, Berlin, pp 201–215
17. Kambadur M, Moseley T, Hank R, Kim Martha A (2012) Measuring interference between live datacenter applications. In: IEEE/ACM SC'12, pp 51
18. Lin Z, Zhou S (1993) Parallelizing I/O intensive applications for a workstation cluster: a case study. SIGARCH Comput Arch News 21(5):15–22
19. Ling X, Jin H, Ibrahim S et al (2012) Efficient Disk I/O scheduling with QoS guarantee for Xen-based hosting platforms. In: Proceedings of CCGRID '12, pp 81–89
20. Lu Y, Chen Y, Amritkar P, Thakur R et al (2012) A new data sieving approach for high performance I/O. In: Proceedings of the 7th international conference on future information technology (FutureTech'12)
21. Lv F, Cui H-M, Wang L, Liu L, Wu CG, Feng X-B, Yew PC (2014) Dynamic I/O-aware scheduling for batch-mode applications on chip multiprocessor systems of cluster platforms. J Comput Sci Technol 29(1):21–37
22. Ma S, Sun X-H, Ioan R (2012) I/O throttling and coordination for MapReduce. Technical Report, Illinois Institute of Technology
23. Mars J, Tang L, Hundt R et al (2011) Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of Micro'11, pp 248–259
24. Mishra AK, Hellerstein JL, Cirne W, Das CR (2010) Towards characterizing cloud backend workloads: insights from google compute clusters. SIGMETRICS Perform Eval Rev 37(4):34–41
25. Moreira JE, Franke H, Chan W et al (1999) A gang-scheduling system for ASCI Blue-Pacific. In: HPCN'99, pp 831–840
26. Ma L, Chamberlain R, Agrawal K (2014) Performance modeling for highly-threaded many-core GPUs. In: Proceedings of IEEE ASAP'14, pp 84–91
27. Ma L, Agrawal K, Chamberlain RD (2014) A memory access model for highly-threaded many-core architectures. Future Gener Comput Syst 30:202–215
28. Ongaro D, Cox AL, Rixner S (2018) Scheduling I/O in virtual machine monitors. In: Proceedings of VEE'08, pp 1–10
29. Park S, Shen K (2012) FIOS: a fair, efficient flash i/o scheduler. In: FAST'12
30. Ryu KD, Hollingsworth JK, Keleher PJ (2001) Efficient network and I/O throttling for fine-grain cycle stealing. In: Proceedings of SC'01, pp 3–3 (CDROM)
31. Schulz G (2006) Data center I/O performance issues and impacts a look at I/O performance bottlenecks and their impact on time sensitive applications. White paper
32. Shakshober DJ (2015) Choosing an I/O Scheduler for Red Hat ® Enterprise Linux ® 4 and the 2.6 Kernel. http://www.redhat.com/magazine/008jun05/features/schedulers/
33. Snavely A, Tullsen D (2000) Symbiotic jobscheduling for a simultaneous multithreaded processor. In: Proc of ASPLOS'00, pp 234–244
34. Sun N-H, Meng D (2007) Dawning4000A high performance computer. Front Comput Sci China 1(1):20–25
35. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in romio. In: Frontiers'99, pp 182–189
36. Thakur R, Ross R, Lusk E, Gropp W, Latham R (2004) Users guide for ROMIO: a high-performance, portable MPI-IO implementation. Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division. Argonne National Laboratory (revised)

37. Zhang Y, Yang A, Sivasubramaniam A et al (2003) Gang scheduling extensions for I/O intensive workloads. In: JSSPP'03, pp 183–207
38. http://hadoop.apache.org/releases.html. Accessed Apr 2015
39. http://www.graph500.org. Accessed Apr 2015
40. http://parsec.cs.princeton.edu/. Accessed Apr 2015