Lv F, Cui HM, Wang L *et al.* Dynamic I/O-aware scheduling for batch-mode applications on chip multiprocessor systems of cluster platforms. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(1): 21–37 Jan. 2014. DOI 10.1007/s11390-013-1409-2

Dynamic I/O-Aware Scheduling for Batch-Mode Applications on Chip Multiprocessor Systems of Cluster Platforms

Fang Lv[1,2] (吕　方), Hui-Min Cui[1] (崔慧敏), *Member*, *CCF*, Lei Wang[1] (王　蕾), Lei Liu[1,2] (刘　磊)
Cheng-Gang Wu[1] (武成岗), *Member*, *CCF*, *ACM*, *IEEE*, Xiao-Bing Feng[1] (冯晓兵), *Member*, *CCF*, *ACM*, *IEEE*
and Pen-Chung Yew[3,4] (游本中), *Fellow*, *IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Department of Computer Science and Engineering, University of Minnesota at Twin Cities, Minneapolis, MN 55455, U.S.A.*

[4] *Institute of Information Science, Academia Sinica, Taibei 115, China*

E-mail: {flv, cuihm, wlei, liulei2010, wucg, fxb}@ict.ac.cn; yew@cs.umn.edu

Received December 28, 2012; revised August 13, 2013.

Abstract Efficiency of batch processing is becoming increasingly important for many modern commercial service centers, e.g., clusters and cloud computing datacenters. However, periodical resource contentions have become the major performance obstacles for concurrently running applications on mainstream CMP servers. I/O contention is such a kind of obstacle, which may impede both the co-running performance of batch jobs and the system throughput seriously. In this paper, a dynamic I/O-aware scheduling algorithm is proposed to lower the impacts of I/O contention and to enhance the co-running performance in batch processing. We set up our environment on an 8-socket, 64-core server in Dawning Linux Cluster. Fifteen workloads ranging from 8 jobs to 256 jobs are evaluated. Our experimental results show significant improvements on the throughputs of the workloads, which range from 7% to 431%. Meanwhile, noticeable improvements on the slowdown of workloads and the average runtime for each job can be achieved. These results show that a well-tuned dynamic I/O-aware scheduler is beneficial for batch-mode services. It can also enhance the resource utilization via throughput improvement on modern service platforms.

Keywords chip multiprocessor, batch processing, co-running, I/O contention, scheduling

1 Introduction

Cluster, datacenter and cloud computing have emerged as major computing platforms for the ever expanding applications today[1]. On such platforms, batch-mode processing (or batch processing, for short) is still one of the major service patterns①-②. It is non-interactive and has very different demands on both performance and QoS[1]. For example, some inquiry services may have higher demands on the responding time (performance), while services such as offline backup have higher demands on correctness. Harvard-MIT Data Center (HMDC)③, some commercial service providers such as Amazon Elastic Compute Cloud (Amazon EC2), and Google Cloud Platform all offer such services. There are three main requirements in batch processing:

- *Scalability in Pipelining.* Batched jobs are dynamically and continuously pumped into the computing platforms, and some with 300 jobs per night.
- *Multi-Dimension Resource Requirements.* The resource requirements for each job include not only computing cores, but also memory, bandwidth, and I/O related resources[3].

Regular Paper

Supported by the National High Technology Research and Development 863 Program of China under Grant No. 2012AA010902, the National Basic Research 973 Program of China under Grant No. 2011CB302504, and the National Natural Science Foundation of China under Grant Nos. 61202055, 60925009, 60921002, 61100011.

①Migration scenario: Migrating batch processes to the aws cloud. http://d36cz9buwru1tt.cloudfront.net/CloudMigration-scenario-batch-apps.pdf, August 2013

②Microsoft. Batch applications — The hidden asset, August 2013.

③Getting started with batch processing. http://support.hmdc.harvard.edu/book/export/html/402, August 2013.

• *Scalability in Datasets.* Due to the development of web applications, big input datasets have become one of the most remarkable characteristics.

These requirements have created higher demands on the server system capacity, and thus have stimulated the development of parallel server systems. Servers have evolved from the former SMP (Symmetric Multiprocessing) architectures to the current CMP (Chip Multiprocessor) architectures, which house multiple sockets and more computing units. On such multi-socket CMP systems, shared resource contentions become major concerns because, if left unattended, the potential contentions on shared resources among competing jobs running on different cores may seriously impede the co-running performance and the overall system throughput. Hence, resolving such contentions has become one of the most important issues for such systems[2-13].

For many applications, shared I/O-related resource is a significant contention point[14]. In fact, I/O bottleneck has been known on parallel computing systems for some time[14-19]. With decades of technological innovations, improvement on I/O latency still lags significantly behind that of CPU and memory. There have been many techniques proposed to improve the I/O performance by rescheduling I/O requests[15-18], or using shared memory as disk cache[20]. However, no matter for I/O intensive applications, or other types of applications which rely on some data input files, I/O contention is still one of the most harassing problems in batch services. Their co-running performances are much more prone to I/O conflicts because of the concurrent file operations. Therefore, more work still needs to be done to mitigate I/O contention on large-scale multi-socket CMP systems.

In this paper, a new approach using a dynamic timeslice-based (quantum-based) I/O-aware scheduling policy is proposed to enhance the I/O performance on multi-socket CMP systems. It is done through regulating I/O contention dynamically. We evaluate the effectiveness of the scheduler from three aspects: the throughput, workload slowdown, and average runtime for each job. The evaluations are setup on an 8-socket, 64-core CMP server node. Fifteen workloads ranging from 8 jobs to 256 jobs dynamically, are experimented on this platform. Experimental results show that the proposed scheduler can achieve 7% to 431% improvements on the throughput of all workloads. Meanwhile, noticeable improvements on the slowdown of workloads and the average runtime for each job can be obtained.

Improving the co-running performance of CMP systems has many practical implications for large web applications with expanding datasets. From this perspective, we made the following contributions in this paper:

• A methodology is proposed that can isolate the impacts of inter-socket I/O contention from intra-socket resources contentions, such as CPU and memories, and give a more precise qualification of the impacts from global I/O contention on large-scale multi-socket CMP systems.

• An effective dynamic scheduling policy for batch processing is proposed to mitigate global I/O contention. The policy is adaptive to the scalability of batch applications and the dynamic variation of periodical I/O contentions. Through evaluations on the throughput, the slowdown of the workloads, and the average runtime for each user job, the proposed dynamic policy is shown to be effective and beneficial for batch services which are sensitive to I/O contentions.

The rest of the paper is organized as follows. The impacts of global I/O contention on co-located batch-processing jobs are examined in Section 2. Section 3 presents our proposed dynamic I/O-aware scheduling policy. Experiments and evaluations are detailed in Section 4. Related work is covered in Section 5. Section 6 concludes the paper.

2 Conflicts in Co-Location

Shared resource contentions among co-running applications are the major reasons for the performance degradation on CMP systems. However, the effects of contention from various shared resources such as CPU, last-level shared cache (LLC), memories, and I/O systems are all juxtaposed in a very complex way. It is difficult to distinguish one kind of contention from another on such systems. In this section, we use a methodology of CMP stacking to distinguish the impacts of global I/O contention from other shared resources contained within a socket. The following two constraints are useful to isolate such I/O contentions.

• *Confinement.* The overall resource requirements of a batch job, including computing cores, private and shared cache memories, as well as memory bandwidth, are satisfied within each CMP (i.e., confined within each socket). Although there may be multiple concurrent batch jobs sharing the resources of the same CMP (in a socket), the inter-socket I/O contention becomes the most outstanding feature for batch jobs running on different CMPs considering the much higher costs of I/O operations versus the lower costs of other resource contentions.

• *Sustainability.* The overall resource requirements of a job abide by the confinement rule during the execution. It will not ask for other inter-socket resources except I/O demands during its entire execution.

The constraints of "confinement" and "sustainability" can be guaranteed in existing systems with NUMA. In particular, for Linux, the default resource allocation strategy (node-local) keeps a job's resource consumption as "local", i.e., its memory will be allocated to the local memory of its core(s) on NUMA architecture[21]. Furthermore, the allocation strategy also keeps the resource consumption as "local" throughout the job execution. Therefore, the premises of CMP stacking can be satisfied, and consequently, inter-socket I/O contention turns into the critical issue for co-running performance degradation. We will give a further discussion for the leading role of the inter-socket I/O contention in Subsection 2.3.1.

CMP stacking is set up to illuminate the negative impacts of I/O contentions on batch processing. However, our solution targets at all kinds of I/O contentions, which include not only inter-socket I/O contentions but also I/O contentions inside a socket.

2.1 CMP Stacking

In our methodology, we gradually increase the intensity of I/O contention by adding CMPs one by one (i.e., stacking up CMPs). Each CMP is fully loaded with concurrent batch jobs on each core. By the two constraints of "confinement" and "sustainability", although jobs on the same CMP still suffer from resource contentions within the CMP, the global I/O contention becomes the major inter-socket interferences during the process of stacking up CMPs. For the easiness of our presentation, we use the following definitions.

For a job, Job^j, on CMP p,

T^j_{alone}: the execution time of Job^j when it runs alone, i.e., without any resource contention;

$T^j_{\text{1-C}}$: the execution time of Job^j when it co-runs with other concurrent batch jobs on the same CMP (denoted as 1-c in the subscript of T), while no other jobs are co-running on the other CMPs concurrently. It is different from T^j_{alone} because of possible contentions within the CMP;

$T^j_{k\text{-C}}$: the execution time of Job^j when there are k CMPs running concurrent batch jobs (denoted as k-C in the subscript of T). It will change when the number of concurrent jobs on other CMPs changes.

The performance degradation due to other concurrent jobs running on other k-1 CMPs can be measured by the difference between $T^j_{\text{1-C}}$ and $T^j_{k\text{-C}}$. It can be clearly ascribed to the inter-socket I/O contentions. We use normalized runtime for the comparison as in (1).

$$normalized\_runtime = \frac{T^j_{k\text{-C}}}{T^j_{\text{1-C}}}. \quad (1)$$

2.2 Benchmarks and Platforms

Before presenting experiments with our methodology of CMP stacking, we introduce the benchmarks and the platform as follows.

2.2.1 Benchmarks

More and more applications today become increasingly sensitive to I/O contentions due to their fast expanding input datasets. In this section, we use duplicated copies of a benchmark with the same input sets to demonstrate I/O contentions. This can facilitate our analyses because they have the same demands on all resources. More complicated and randomly generated workload types are covered and examined in Section 4.

Different I/O APIs can lead to different forms of I/O contentions. We have observed two types of I/O contentions from our experiments:

• Explicit I/O. It is caused by the usage of API such as *fread* and *fwrite*, which contends for I/O related resources directly and as a result, suffers from I/O conflicts directly.

• Implicit I/O. It is incurred by the memory associated file operations (e.g., *mmap*), which impose a high pressure on the main memory. Thus, swapping is usually involved in these operations, leading to I/O contentions.

Considering the above differences, we use three kinds of applications to demonstrate the side-effects of I/O contentions in Table 1:

• Real User Application. Two real applications from regular users in Dawning Cluster are adopted in our work, which are paper similarity examination and Kmeans cluster algorithm. These two applications employ explicit I/O APIs in their file operations.

• Benchmarks from Graph500[④]. Graph traversal algorithms with sequential compressed-sparse-row implementation are used. The amount of I/O requests in the benchmark is proportionate to the graph size it traverses. The graph for searching is generated with two parameters, s and e. They correspond to a graph's scale and edge factor, respectively. For example, the graph created with "-s 22 -e 18" is much larger than that with "-s 22 -e 16", so does the number of I/O requests. The usage of *mmap* in the application will lead to continuous implicit I/O behaviors.

• Benchmarks from the Princeton Application Repository for Shared-Memory Computers (PARSEC 3.1)[⑤]. This package is made up of more than ten applications, which have diverse sensitivities to I/O contentions due to their different sizes of data input files

[④]http://www.graph500.org/, August 2013.

[⑤]http://parsec.cs.princeton.edu/, August 2013.

Table 1. Description for Applications

	Application	Type	Description
Real user application	Paper Similar (PS)	2-threaded	A program which compares a paper with the other K papers concurrently, where K is 2 in our work
	Kmeans Clustering (KM)	8-threaded	A key algorithm from data mining which partitions n observations into K clusters
Graph500	Graph	1-threaded	BFS algorithm from Graph500. The graph for searching is generated with two parameters, s and e, which stand for a graph's scale and edge factor, respectively
PARSEC benchmarks	x264	1-threaded	Encoding video
	Vips	1-threaded	Image processing library
	Freqmine	1-threaded	Data mining problem
	Bodytrack	1-threaded	Tracker of the 3D pose of a human body
	Raytrace	1-threaded	Tracing the path of light and generating images

and different periodic I/O characteristics. Contentions from explicit I/O APIs can be demonstrated with this package. Among all datasets, the medium dataset of simlarge and the largest dataset of native are used in our work. We only introduce five benchmarks which are relatively more sensitive to I/O contentions as shown in Table 1. We will include some benchmarks such as swaptions which are less sensitive to I/O contention in Section 4 for more thorough evaluations.

A workload is composed of one or more batch jobs. For a clearer analysis, in this section, we use single-threaded jobs as our examples to demonstrate the I/O contentions, and the number of concurrently running jobs in the workload ranges from 8 to 64 on a server node. Note that our CMP stacking method and our scheduling solution themselves do not have these limitations. We will cover both single-threaded job and multi-threaded jobs in later sections. Dynamically increasing the number of batch jobs for the workload is also permitted. All these issues will be discussed and evaluated in Section 3 and Section 4.

2.2.2 *Platform*

The server node used in our work is a CMP system integrated with Intel® Xeon® X7550 processors in Dawning Linux Cluster. It is based on Nehalem architecture. Most of the state-of-art high-performance CMP systems from Intel® are evolved from this type of architecture. The CMP server is an 8-socket CMP server node with NUMA support. Each of the CMP (socket) has 8 cores and 32GB local memory. It uses Linux OS 2.6.32 for X86-64.

2.3 Performance Degradation from I/O Contention

2.3.1 *Influences from Inter-Socket I/O Contentions*

In this subsection, we use CMP stacking to illustrate the performance degradation from I/O contention. For a clearer description, CMP stacking is experimented with four workloads, which are composed of either duplicated explicit I/O jobs (x264) or duplicated implicit I/O jobs (Graph) in this subsection. CMP stacking for each workload includes four steps: one-CMP running, two-CMP running, four-CMP running, and eight-CMP running. Each step runs 8, 16, 32, and 64 jobs, respectively. During each step, we full-load all cores with 8 jobs on each co-running CMP. Through this process, we can observe the severe performance impacts from inter-socket I/O contentions.

Table 2 lists the detailed information for each workload that is generated from two benchmarks, x264 and Graph.

Table 2. Information of the Four Workloads

Index	Benchmark	Input Set
#1	x264	Simlarge
#2	x264	Native
#3	Graph	-s 22 -e 16
#4	Graph	-s 22 -e 18

There are four curves in Fig.1. Each curve stands for the normalized runtime of each job in the four steps (denoted as k-C or k-CMP in later figures). Take workload #1 in Table 2 as an example, the average runtime for an x264 job is 7 seconds (denoted as 7 s) on one CMP, while it degrades to 122 s when co-running with

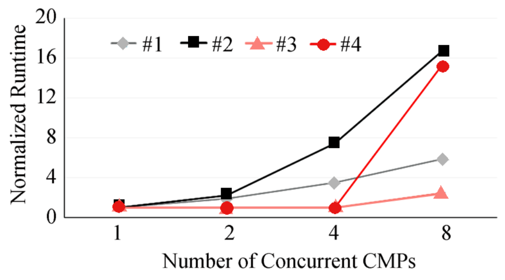


Fig.1. Inter-socket performance impacts from I/O contentions while increasing the number of CMPs.

other 56 jobs on seven other CMPs. It is about 16x degradation due to the increased I/O contention.

Data in the figure display similar trend in performance degradation for all four workloads. That is, each job's performance will degrade with CMP stacking. The more co-runners are, the more they suffer from I/O contentions.

2.3.2 *Discussion for Other Influences*

It is worth noticing that the inter-socket I/O contention is not the only type among different sockets. Cache coherency (CC) still plays a role among CMPs for some cache-miss intensive tasks. However, CC cost is much lower than that from I/O contentions. Therefore, we ignore CC interferences in our work according to the following experiments.

Fig.2 displays experiments for the inter-socket CC costs on our Intel Nehalem server system, which uses MESIF[6] as its cache protocol. The maximum CC cost is about 33.7%, generated from the most serious LLC (last level cache) misses (6.90/cycle per CMP) during 8-CMP co-running. This is much more trivial than 2x∼16x degradations from I/O contentions in Fig.1.

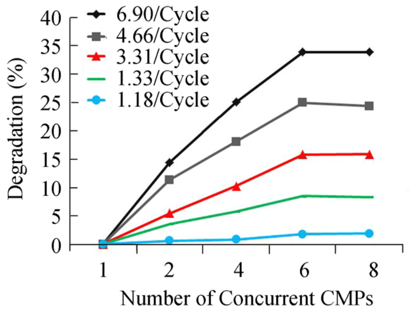


Fig.2. Performance degradation from inter-socket CC costs in k-CMP configuration, $k = 1, 2, 4, 6, 8$.

The experiments in Fig.2 are performed with CMP stacking, while each CMP runs LLC-miss intensive kernels as in Fig.3. All experiments comply with the two constraints of "confinement" and "sustainability". Each data copy in the kernel can produce a read miss and a write miss. Through varying the number of nops, we can get different LLC miss rates inside a CMP. CMP LLC miss rate at 6.90/cycle is the maximum which can be generated on the server. During the process of CMP stacking, intensive LLC misses in a CMP bring forth cache coherency information globally, which results in inter-socket performance influences. In such circumstance, CC cost is the most outstanding inter-socket performance influence. Each curve in the figure stands for the runtime degradation of the kernel during the CMP stacking process. Data in the figure displays that the higher the LLC miss rate is, the more CC cost is. However, the maximum performance influences from CC (by LLC miss rate 6.90/cycle) is only about 33.7% during 8-C running.

Procedure LLCMISS_Pressure
```
#define ITERATION 1000 //repeat the experiments
#define CACHELINE 64 //the cache line size
#define COL CACHELINE/sizeof(int)
#define ROW (MEM_SIZE/COL*sizeof(int))
int a[ROW][COL], b[ROW][COL];
#define nops 5000 //intervals between two successive
reads

/*Memory allocation
    Initialization(a, b);
for (iter = 0; iter < ITERATION; iter++) {
/*LLC cache miss kernel
        for (i = 0; i <ROW; i++) {
            b[i][0] = a[i][0];
            //Use nops to adjust the density of LLC misses
            for (k = 0; k <nops; k++) {
                asm ("nops");
            }
        }
}
```

Fig.3. LLC-miss intensive kernel.

Based on above all, we ignore CC costs and only focus on costs from I/O contentions in our paper.

2.4 Analysis for I/O Contention

2.4.1 *Analysis Methodology*

The analysis of global I/O contention is made with the support of Linux OS. For each I/O request serviced by the local storage disk, the latency can be divided into two parts: the I/O waiting time and the hard disk serving time by the I/O devices, as shown in Fig.4.

$$IO\_Latency = Latency_{\rm serving} + Latency_{\rm waiting}. \quad (2)$$

$Latency_{\rm serving}$ is the actual service time of an I/O request by the I/O device. This latency is decided by both the decision making of the disk controller and the specific I/O devices. $Latency_{\rm waiting}$ is the handling time of the software scheduler for an I/O request, and the time cost in I/O queues. The default task scheduler and the default I/O scheduler are all fairness-oriented policies on our Linux OS[22]. $Latency_{\rm waiting}$ also includes the extra overhead resulted from bursts of I/O requests. I/O latency is calculated with these two parts as in (2).

[6]http://en.wikipedia.org/wiki/MESIF_protocol, August 2013.

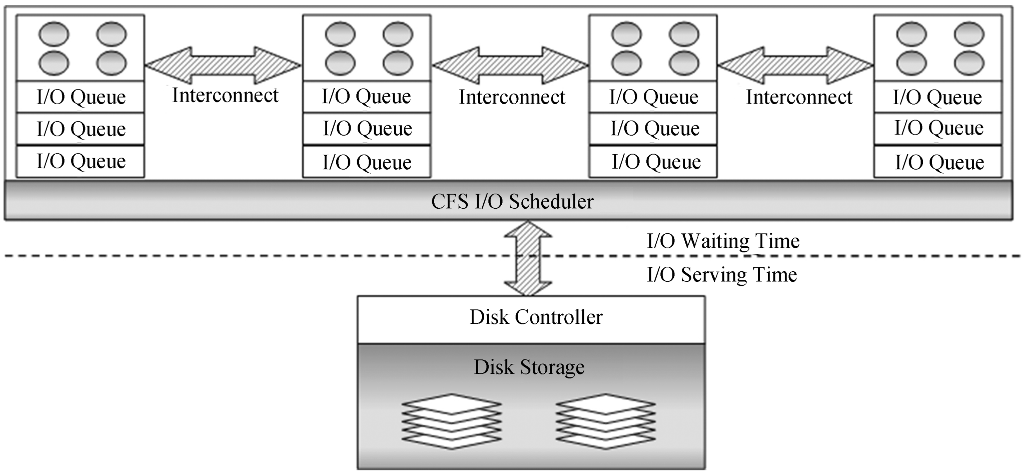


Fig.4. Two parts for the latency of I/O requests.

The analysis is made with a Linux user utility, *iostat*, a statistical tool for I/O devices in Linux. This tool samples the status of I/O devices at a fixed time interval specified by the user. In our experiments, we set the sampling interval to be 1 second.

For each I/O request, *IO_Lantency* can be calculated from the entry *await* in the report generated by *iostat*. It includes both waiting time and hard disk serving time for each I/O device operation. The entry *svctm* in the report stands for the serving time and the software waiting time can be calculated accordingly. By studying changes in these two parts in the process of CMP stacking, I/O bottlenecks exposed by I/O contention can be identified.

2.4.2 *Analysis Results*

We analyze I/O contentions from two aspects: contentions from implicit I/O interfaces and contentions from explicit I/O interfaces.

1) *Analysis for Explicit I/O Behavior.*

Take workload #1 as an example, Fig.5 and Fig.6 show $Latency_{\text{waiting}}$ and $Latency_{\text{serving}}$ in k-CMP configurations, $k = 1, 2, 4, 8$. Data in these two figures show that the scaling during CMP stacking can lead to degradation in both the serving time and the waiting time. However, the increase in the waiting time deserves more attention since it deteriorates much more seriously than the serving time. As shown in Fig.5, when we scale from 1 CMP to 8 CMPs, the average waiting time in 8-CMP configuration shows 128x degradation compared with that in 1-CMP, i.e., 590 ms vs 4.55 ms.

A more detailed comparison is made between these two kinds of latencies, and the results are shown in Fig.7. The contrast clearly shows that I/O contention has a much more severe impact on the waiting time

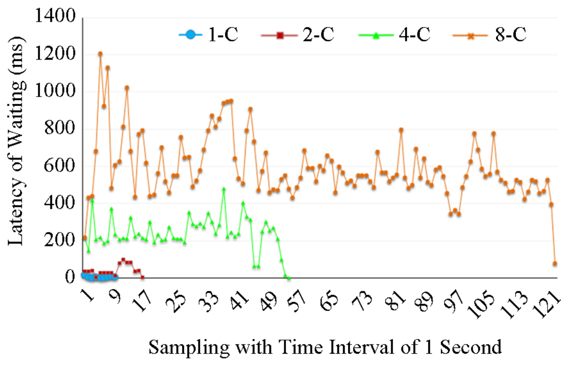


Fig.5. Average waiting time in k-CMP configuration, $k = 1, 2, 4, 8$.

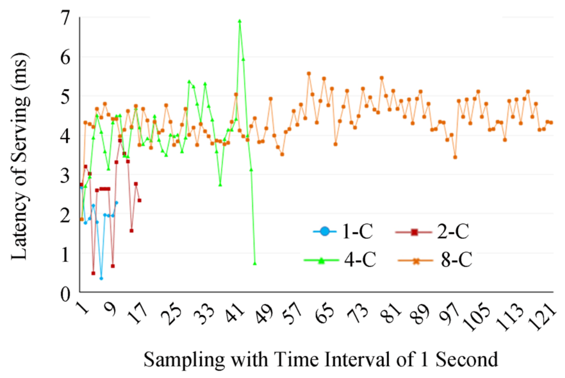


Fig.6. Average serving time in k-CMP configuration, $k = 1, 2, 4, 8$.

than on serving time. Bursty I/O requests in a time interval that cannot be handled due to limited I/O resources will accumulate, and have a severe impact on other co-running batch jobs in I/O queues.

From these figures we can observe that for explicit I/O jobs, I/O quantities from co-running jobs have direct relations with the performance influences. The more co-runners are, the more co-runners suffer.

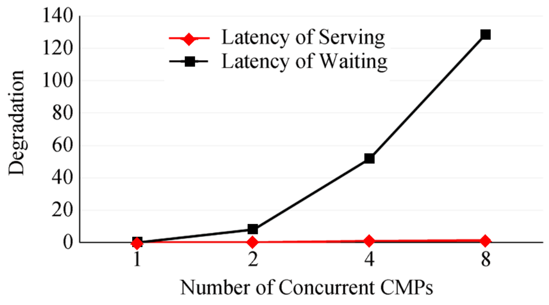


Fig.7. Trend of I/O performance degradation for waiting time and serving time in k-CMP configurations, $k = 1, 2, 4, 8$.

2) *Analysis for Implicit I/O Behavior.*

From Fig.1, we observed that workloads #3 and #4 behave differently from workloads #1 and #2. The average runtimes of #3 and #4 keep stable until the co-runners reach 32 jobs (before 4-C). After that, the average runtimes display sudden degradations. The difference is actually resulted from the usage of implicit I/O interfaces. For memory-associated optimizations such as implicit I/O, paging will put a high pressure on the memory. When the accumulated memory demands reach a certain degree (50% of the whole system memory in our environment), swapping for each job will happen, which is always companied with severe I/O contentions among co-running jobs.

The following experiments are used to study the correlation between the memory utilization ratio and the severity of I/O contentions for implicit I/O workloads. These experiments are implemented with four more Graph workloads and each workload has 64 Graph jobs. Through varying the value of s and e as in Table 1, different graphs can be generated, which will result in different memory sizes to associate file operations.

In our experiment in Fig.8, the memory utilization ratio ranges from 26.9% to 68.6% (the system memory is 256 G in total). The figure draws an interesting conclusion: the more memory we use, the more I/O quantities will be generated and the more contentions we have to suffer. As can be seen from the figure, if the memory utilization is just 48.4% (or less), the I/O contention period is only 1/250 of the entire sampling period, indicating that the overall system performance is not impacted by I/O contention seriously. Nevertheless, on the contrast, when the memory utilization achieves 68.6%, we have 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. Notably, the figure only displays partial data although we have sampled the entire execution period.

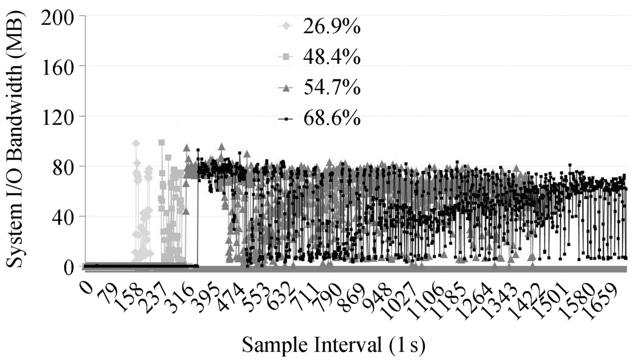


Fig.8. Correlation between memory utilization ratio and the severalty of I/O contentions.

From the above analysis we can learn two points:

• The root reason for the performance degradation is the much higher I/O demands and the relatively lower disk serving capacity. The mismatch between these two aspects results in dramatically longer I/O waiting time in I/O queue.

• No matter for explicit I/O jobs or implicit I/O jobs with high memory demands, they will eventually lead to I/O contentions, and I/O quantities have much relation with the performance influences during the contention.

In this paper, we propose an I/O-aware scheduling policy. The policy uses a dynamic analyzing process for I/O activities at a fixed time interval. Decisions are made during these intervals by regulating the co-running jobs, so that overheated I/O contention can be mitigated.

3 Dynamic I/O-Aware Scheduling Policy

Based on the above analyses, an I/O-aware scheduling policy is proposed, which is implemented as a user-level timeslice-based scheduler. Timeslice-based scheduling is an effective technique to deal with the dynamic variation of resource contention. It has been applied in other contention-aware schedulers, such as bandwidth-aware scheduling[8] and LLC-aware scheduling[9]. In this paper, we apply it with an I/O-aware scheduling policy in order to regulate I/O conflicts.

3.1 Framework of the Dynamic Scheduler

We implement the proposed dynamic I/O-aware scheduler (called dynamic scheduler in the rest of the paper) as a user-level scheduler in Linux. It can also be applied to other OS with a slight modification in its system call interfaces.

The framework of the dynamic scheduler and its interface to OS are shown in Fig.9. The dynamic scheduler is registered as an exception handler in OS as shown in the left part of the figure. Each time OS receives a timer signal which is specified by the user (at least 1 second), the exception handler will find the entry for our dynamic scheduler and transfer the task management control to the dynamic scheduler. The dynamic scheduler takes over all concurrent jobs and samples I/O information for each job. The I/O information for each job can be collected through reading I/O files (under/proc/pid/io). After the sampling, I/O related analysis and the scheduling policies can be applied on these jobs. Two kinds of interventions, job suspension and resumption, are made according to some heuristics. OS then takes over the management of both the jobs and I/O operations once again. In this way, the dynamic scheduling policy still can make use of all existing optimizations for the tasks and I/O operations in OS.

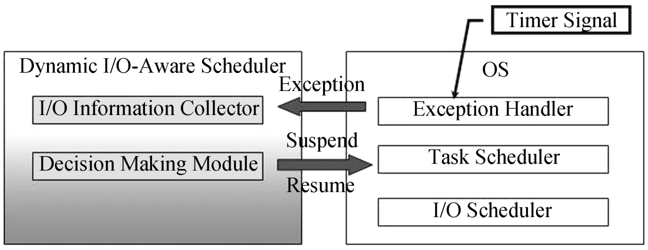


Fig.9. Overall framwork of the dynamic I/O-aware scheduler.

The framework of the user-level dynamic scheduler is also presented as the pseudo-code in Fig.10. It is registered as a signal handler, *sigalarm_handler_IO*, in line 3 of the pseudo-code. This program is triggered at fixed interval (timeslice, or quantum), which is defined by a tunable variable, *TIMESLICE*, in line 2.

Procedure 1. Dynamic IO-Aware Scheduling

```
   {
1:    struct itimerval IO_Interval;
2:    IO_Interval.it_interval.tv_see=TIMESLICE;
3:    Signal(SIGALRM, sigalarm_handler_IO);
4:    int res =setitimer(ITIMER_REAL, & IO_Interval,
5:    NULL);
   }
```

Fig.10. Dynamic scheduling policy is implemented as a user-level timeslice-based scheduler that is triggered by a timer signal.

3.1.1 Overall Approach

As shown in the left of Fig.9, the overall policy in the signal handler is made up of two parts: I/O information collector, and decision-making module. The implementation of the policy is presented in Fig.11. The two major parts correspond to line 6 and line 14 respectively in the figure.

Procedure 2. Sigalarm_Handler_IO

```
1:  {
2:     /*Read IO information for each running job
3:     *from system device file
4:     */
5:
6:     Overall_IO=IO_Information_Collector();
7:
8:     /*For time intervals which cumulated IO requests
9:     *exceed the I/O capacity limit, scheduling policy
10:    *will be applied
11:    */
12:
13:    if (Overall_IO>IO_BOUND_INTERVAL) {
14:       Decision_Making_Module();
15:    }
16: }
```

Fig.11. Major components of the I/O-aware scheduling policy.

• I/O Information Collector. This part mainly collects I/O information for each job at certain intervals under the control of OS. With such information, the dynamic scheduler can regulate the execution of concurrent jobs under the guidance of some heuristics in the decision-making module.

• Decision-Making Module. This module plays a regulatory role to reduce the congestion from bursty I/O without causing excessive idleness in I/O devices. Two kinds of scheduling decisions are made on each job, suspension or resumption, according to the available capacity of I/O devices in each interval, *IO_BOUND_INTERVAL*.

3.1.2 Workload

The workload needs to be scaled with the number of jobs dynamically. Most of the batch jobs have relatively less stringent demands on the QoS and the performance. Newly arrived jobs are appended to the end of the workload queue. Core-sharing for independent jobs is not used for batch jobs on many of the current platforms. At any time, the number of concurrent batch jobs will not exceed the number of cores in the system. All batch jobs are serviced according to their submit order without preemption. A job is scheduled whenever a core becomes available.

3.2 I/O Information Collector

With the support of OS, I/O operations of reads and writes from each batch job are profiled and stored in a device file periodically. *IO_Information_Collector* in Fig.12 parses the devices file (under /proc/pid/io), and determines the I/O requirement for each batch job. It is realized by *Generate*_IO in line 4. Those jobs which have non-zero I/O demands will be put in the candidate queue. Each CMP has its own candidate queue. The total number of I/O requests from all concurrent batch jobs in a CMP is calculated and stored in *Overall*_IO in line 5. This information is useful for later analyses and decisions made by the dynamic scheduler.

```
Procedure 3. IO_Information_Collector
1: {
2:    open(device_io_file);
3:    for each job in CMPs{
4:       io_request = Generate_IO (device_io_file);
5:       Overall_IO+=io_request;
6:    }
7:    return Overall_IO;
8: }
```

Fig.12. Periodical collection of I/O information.

3.3 Decision-Making Module

The decision-making module targets at two key issues: when to schedule and how to schedule. First, the module will supervise the whole I/O bandwidth usage and decide whether or not it needs to interfere with the co-running execution. Second, at the moment that the bandwidth exceeds the boundary, one of the two measures, either suspension or resumption, is taken for candidate jobs. Because the entire bandwidth is amortized equally into each CMP, the dynamic module only cares jobs on the CMP that exceeds its portion.

3.3.1 When to Schedule

Similar to other shared resources, the available capacity of I/O devices in a time interval is limited. We use a threshold value, *IO_BOUND_INTERVAL* (MB/s), as a guidance for scheduling.

3.3.2 How to Schedule

Different jobs show different sensitivities to I/O contentions. This sensitivity has much relation with I/O quantities (I/O bandwidth requirements) of each job. We demonstrate this relation with the following experiments.

We design a module, named *Sensitivity_RANKING*, which is composed of a file reading kernel. As Fig.13 shows, each kernel performs file reading only for one time in case of page caching. Through varying the block size for each read, the interval between two successive readings and the number of concurrent kernels (inserting nops), we can get different ranks of simultaneous I/O requests.

```
#define block_size 32 //data size for one read
#define nops 5000 //intervals between two successive
reads

//Read file only for one time in case of page caching
while (!eof(file)) {
        fgets(file, block_size);
        //Use nops to adjust the bandwidth density
        for (i = 0; i <nops; i++) {
            asm ("nop");
        }
}
```

Fig.13. Kernel of *Sensitivity_RANKING*.

We illustrate the relation between simultaneous I/O quantities and their performance influences through 12 groups of experiments in Fig.14. Tasks in these 12 groups of experiments are composed of duplicated *Sensitivity_RANKING* kernels which has 1.5 MB/s$\sim$26.4 MB/s I/O demands per CMP respectively. Each curve in the figure stands for the averaged runtime degradation for a task when increasing co-running tasks from 1 to 8 CMPs. From the figure we can learn that, the higher the averaged I/O quantity is, the easier co-running performance degradation happens. For the group with 26.4 MB/s bandwidth requirement, it starts to degrade when there are only two co-runners on an 8-socket, 64-core server. For the group which has bandwidth lower than 5.63 MB/s, I/O contentions will not result in co-running performance influences.

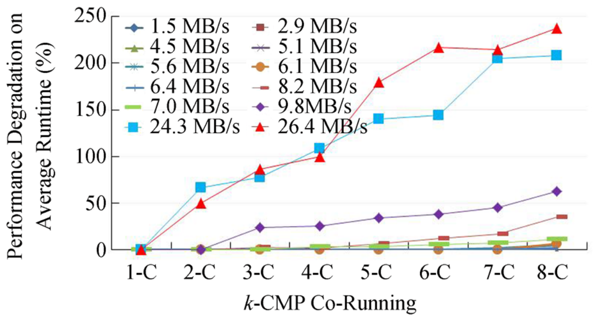


Fig.14. Averaged performance degradation in k-CMP configuration, $k = 1 \sim 8$.

Therefore, the sensitivity of a task to I/O contentions has much relation with its averaged I/O bandwidth demands. This inspires us that, to mitigate the I/O contention problem in large-scale platforms, the sort of high-I/O quantity jobs (e.g., *K*means) is worth more attention than those tasks of lower-I/O quantity (e.g., Graph). Regulation on high-I/O quantity tasks will reduce the I/O bandwidth pressure, which are beneficial for more low-I/O quantity jobs.

The pseudo-code for the decision-making module is shown in Fig.15. It intends to control the total number of I/O requests in the system so that they will not result in severe congestion and long waiting time.

```
Procedure 4. Decision_Making_Module
    {
 1:     /*Sorting jobs descendingly according to
        *I/O qualities on each CMP
 2:     */
 3:     Sorting_Jobs_on_Each_CMP();
 4:
 5:     /*Major part for decision maker */
 6:     /*In case of idleness of a CMP, the policy will
 7:     *pick out jobs which has the most I/O qualities
 8:     *on each CMP separately
 9:     */
10:     for the i-th CMP in CMPs {
11:         head[i]=the first job in the job set of the CMP;
12:     }
13:     if (Overall_IO>IO_BOUND_INTERVAL) {
14:         /*For a quantum in which Overall_IO exceeds the
15:         *upper capacity of the I/O device, the dynamic sche-
            *duler suspends partial candidate jobs
16:         */
17:         while (Overall_IO>IO_BOUND_INTERVAL) {
18:             for the i-th CMP in CMPs {
19:                 if (Status (head[i]) is RUNNING) {
20:                     Overall_IO-=head[i].IO;
21:                     Status(head[i])=SUSPENDED;
22:                 }
23:                 if (Overall_IO<=IO_BOUND_INTERVAL)
24:                     break;
25:                 }
26:                 head[i] = bead[i]->next;
27:             }
28:             if (Overall_IO<=IO_BOUND_INTERVAL)
29:                         break;
30:         }
31:     } else {
32:         /*For a quantum in which Overall_IO is below the
33:         *upper capacity of I/O, resuming those jobs which
34:         *are suspended
35:         */
36:         for the i-th CMP in CMPs {
35:             while (head[i]!=NULL) {
36:                 if (Status(head[i]) is SUSPENDED) {
37:                     Status(head[i])=RESUMING;
38:                     break;
39:                 }
40:                 head[i] = head[i]->next;
41:             }
42:         }
        }
    }
```

Fig.15. Two different scheduling decisions according to the number of I/O requests.

In an interval, if I/O requests exceed the upper bound of the I/O capacity, the policy will start to suspend some of the jobs until the total number of I/O requests drops below *IO_BOUND_INTERVAL*. The pseudo-code to make a decision on suspending a job is presented in line 17 to line 30 in Fig.15. In an interval, if the total number of I/O requests drops below the upper bound of I/O capacity, another kind of decision, *resumption*, will be made. Its pseudo-code is shown in line 32 to line 42. Jobs that were suspended will be resumed for better utilization of I/O capacity. In case of too aggressive contentions from resumption, we will let go a job at a time.

Moreover, to avoid excessive idleness in a CMP, a procedure, *Sorting_Jobs_in_Each_CMP* is used for making such a decision. It is shown in line 3. It sorts all jobs according to their I/O requirements in a descending order. The policy will select the jobs that currently have the most I/O demands pending on each CMP for resumption.

3.4 Parameter Setting

There are two tunable variables in our dynamic scheduler: *TIMESLICE* in I/O information collector and *IO_BOUND_INTERVAL* in the decision-making module.

Setting of TIMESLICE. The signal handler is triggered at fixed time intervals. The number of time quantum is defined by *TIMESLICE*. It is used as the granularity of time intervals for job scheduling. This value is similar to the timeslice used in the Linux scheduler. Since the average I/O latency is much higher than that of memory operations and the algorithm is assisted with periodic analysis of system I/O files, the cumulative time overheads of these components can result in substantial total time overhead. Therefore, the value of *TIMESLICE* should be carefully selected. Two different values for *TIMESLICE* are adopted in our evaluation in Section 4. For jobs with relatively shorter execution time, e.g., benchmarks in PARSEC with simlarge, we use a fine-grained *TIMESLICE*, set at 1 s. For jobs with longer execution time (more than 1 000 s), e.g., benchmarks in Graph500, we use a coarse-grained *TIMESLICE*, and it is set at 20 s.

Setting of IO_BOUND_INTERVAL. Since the theoretical optimal value is always difficult to obtain in real world, this threshold value for a specific CMP system can be obtained through experimental results (e.g., via experiments with *Sensitivity_RANKING* module) or some empirical values. The value in our current policy is set at 40 MB/s.

3.5 Scheduling Overheads

Overheads of our proposed dynamic I/O-aware scheduling determine not only the overall performance of the workloads, but also the practicability of such a policy. The total overheads are the sum of those incurred in each time quantum. The overheads in each quantum depend on two major components: the I/O information collector and decision-making module. The time complexity is $O\ (N^2)$ for both of them, where N is the number of batch jobs.

Since our policy is implemented as a user-level scheduler, the overheads of context switching due to system calls are the most time-consuming part. This is mainly due to the current implementation of Linux that allows certain time delay after it receives the signal before carrying out job suspension. Nevertheless, the overheads will not exceed 1% when *TIMESLICE* varies from 1s to 20 s.

4 Evaluations

Our dynamic I/O-aware scheduling policy is evaluated on an 8-socket 64-core X7550 server node with benchmarks as introduced in Table 1.

Since our dynamic scheduler is implemented as a user-level scheduler, it will be taken over by Linux scheduler eventually. The efficiency of our dynamic scheduler can be evaluated through a comparison between Linux scheduler without and with our scheduling policy.

To study its efficiency more comprehensively, performance improvement on the throughput of the workloads is examined. The throughput is calculated by the number of jobs in a workload (N), and the execution time of the workload, T_{workload}, as shown in (3). For a fair comparison we evaluate our scheduler from the other two aspects, slowdown and the average runtime. The slowdown of a job, Job^j, is calculated by the ratio of the runtime when it runs alone to that when it runs in a k-CMP configuration. The slowdown for a workload is the sum of slowdown for all jobs as shown in (4). The average runtime of a workload (*Aver-runtime*) is calculated with all jobs' $T^j_{k\text{-C}}$ values as in (5), where T^j_{alone} $T^j_{k\text{-C}}$, are defined in Subsection 2.1.

$$throughput = \frac{N}{T_{\text{workload}}}, \quad (3)$$

$$slowdown = \sum_{j=1}^{N} \frac{T^j_{\text{alone}}}{T^j_{k\text{-C}}}, \quad (4)$$

$$Aver\text{-}runtime = \frac{\sum_{j=1}^{N} T^j_{k\text{-C}}}{N}. \quad (5)$$

4.1 Workloads

Two different types of workloads are studied in this subsection:

• *Duplication Type* (*D-type*). The workload is composed of duplicated I/O sensitive jobs that have almost the same behavior and will suffer serious contention when they are running concurrently.

• *Mixed Behaviors* (*M-type*). The workload is composed of different jobs that are combined randomly with all applications in Table 1.

Table 3 lists detailed information for 15 workloads, which includes benchmark name, input dataset, and the corresponding number of jobs in each workload, denoted as batch length. At the beginning of the execution for each workload, the server full-loads 64 single-threaded jobs on all 64 cores or 8 multi-threaded jobs on all 8 CMPs at most for each batch. No core-sharing is permitted in our experiment. For M-type workloads, each job takes a separate CMP and at most 8 jobs can be handled in a batch. More jobs will be served whenever there are cores released and become idle.

Table 3. Information for the Workloads

Type	Index	Benchmark	Input Set	Batch Length
D-type	#1	x264	Simlarge	128
	#2	x264	Simlarge	256
	#3	Freqmine	Simlarge	64
	#4	PS	Duplicated	8
	#5	PS	Different	24
	#6	KM	Duplicated	8
	#7	Graph	-s 22 -e 16	64
	#8	Graph	-s 22 -e 18	64
	#9	Graph	-s 22 -e 16	128
M-type	#10	Raytrace + x264	Simlarge	128
	#11	Rarsec + Graph	Mixed	128
	#12	Parsec + Graph	Mixed	128
	#13	Parsec + real	Mixed	128
	#14	Parsec + real	Mixed	128
	#15	Parsec	Simlarge	64

As in Table 3, experiments with D-type are composed of 9 workloads. Explicit I/O interfaces are evaluated through PARSEC benchmarks (workloads #1~#3) and real user applications (workloads #4~#6). Implicit I/O interfaces are evaluated through Graph applications (workload #7~#9). Workloads #10~#15 are M-type, which are mixed with PARSEC benchmarks, Graph and real user applications. Applications in a workload can be single-threaded or multi-threaded. Moreover, these workloads are mixed with applications which are either sensitive to I/O contentions or not (from PARSEC or Graph applications with lower memory demands).

4.2 Evaluations for D-Type Workloads

4.2.1 Performance Results

Fig.16 shows the performance improvement for six workloads using our dynamic scheduler. All workloads have achieved notable improvement in average runtime, ranging from 6.7% to 67.9%, respectively. As for the slowdown, most of the workloads obtain improvement ranging from 8.3% to 73.3%, respectively, while #2 suffers a slight degradation by 2.9%. Moreover, most of the workloads can benefit 7.09%∼40.4% on the system throughput, while #3 and #6 do not obtain obvious benefits from this scheduler in their throughputs.

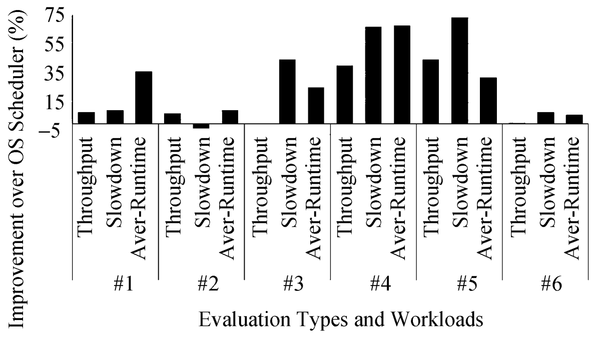


Fig.16. Performance improvement for D-type workloads (explict I/O jobs: PARSEC and real user applications).

Compared with workloads of PARSEC, workloads with concurrent Graph traversal algorithms have obtained much more improvements from the dynamic scheduling in Fig.17. The throughputs for them range from 45% to 431%. The improvements on average slowdown range from 40% to 433%. The average runtime is improved by 45% to 82% with our dynamic scheduler. For #8 which includes 64 jobs, our dynamic scheduler obtains the most improvements on the throughput by 431%.

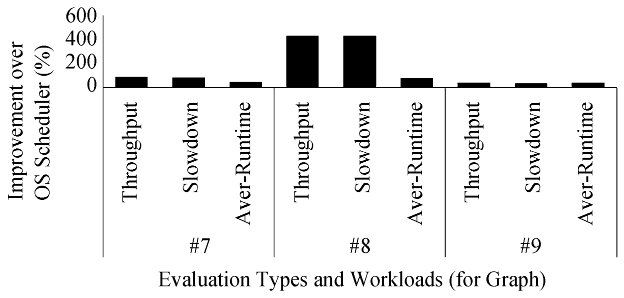


Fig.17. Performance improvement for D-type workloads (implicit I/O jobs: Graph).

4.2.2 Performance Analysis

We take #1 for further analysis. Comparisons for two kinds of latency, $Latency_{\text{waiting}}$ and $Latency_{\text{serving}}$, are made between Linux scheduler without and with the optimization of the dynamic scheduler, denoted as OS and Dynamic, in Fig.18 and Fig.19, respectively. Latency data are sampled at intervals of 1 second. Fig.18 illustrates the effects on $Latency_{\text{waiting}}$ through inhibition of bursty I/O requests. With the dynamic scheduler, the average waiting time is improved by 59%, from 696ms to 286ms.

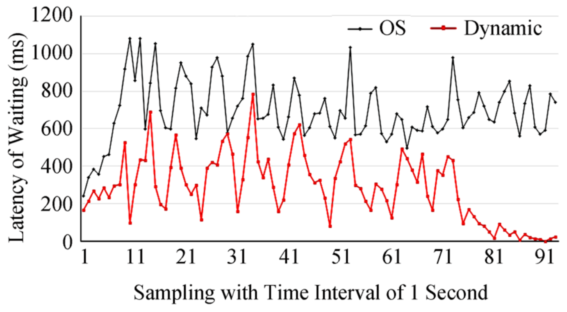


Fig.18. Improvement on waiting time for workload #1.

$Latency_{\text{serving}}$ can also benefit from the dynamic scheduler as shown in Fig.19. It achieves 21% improvement with the dynamic scheduler, from 4.95 ms to 3.9 ms.

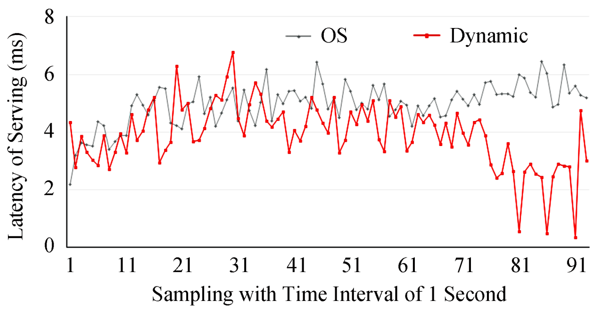


Fig.19. Improvement on serving time for workload #1.

4.3 Evaluations for M-Type Workloads

4.3.1 Performance Results

In this subsection, we evaluate six other workloads which are mixed with the three types of applications in Table 1.

Since the workloads are composed of benchmarks with different execution time, the throughput is influenced by the jobs that have the longest runtime. The evaluation of slowdown can reflect the effect of optimization more properly.

Fig.20 shows the improvement for M-type workloads. All these workloads achieve improvement on the slowdown, ranging from 10.8% to 97.7%. The improvement on average runtime ranges from 9.3% to 56.2%. Except workload #10, all the other workloads see improvements in throughput at 1.5%∼131.2%, respectively. Although workload #10 does not see obvious improvement in throughput, neither does it suffer with the dynamic scheduler.

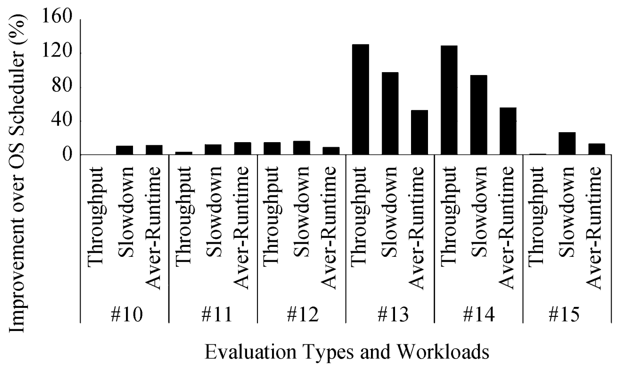


Fig.20. Performance improvement on M-type workloads.

4.3.2 *Performance Analysis*

We also perform analysis on the effect of the dynamic scheduler for M-type workload #11. The comparisons of two kinds of latency, $Latency_{\text{serving}}$ and $Latency_{\text{waiting}}$, are made between Linux scheduler without and with optimization of our dynamic scheduler. Data sampled at interval of 1 s are collected for these two kinds of latency and the improvements are shown in Fig.21 and Fig.22.

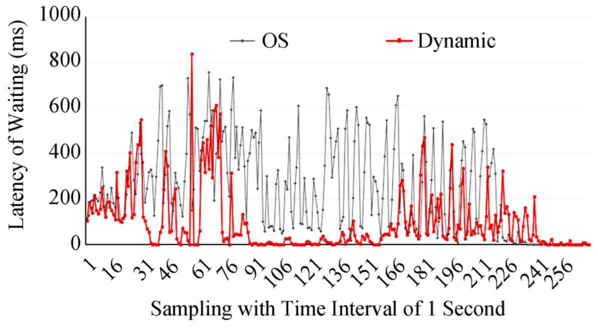


Fig.21. Improvement on waiting time for workload #11.

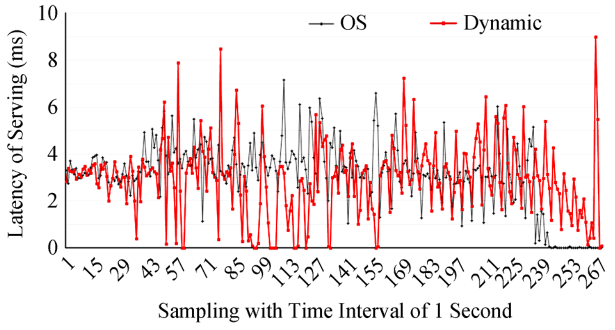


Fig.22. Improvement on serving time for workload #11.

Fig.21 shows the effect of the dynamic scheduler on $Latency_{\text{waiting}}$. Average waiting time from all sampled intervals is improved by 60% compared with that with Linux scheduler, from 243.6 ms to 98 ms. $Latency_{\text{serving}}$ also benefits slightly from the dynamic scheduler as shown in Fig.22. The average serving time is improved by 4.6% compared with that with Linux scheduler, from 3.05 ms to 2.91 ms.

4.4 Discussion on Efficiency and Inefficiency

In the experiments, we observed that workloads that have a higher sensitivity to I/O contention would also show a sensitivity to the dynamic scheduler. For example, jobs of x264, raytrace and Graph with "-s 22 -e 18" suffer a lot from I/O contention when there are multiple co-running jobs. They are also much easier to achieve more improvement from optimizations that deal with I/O contention.

Through detailed comparisons between Linux scheduler without and with the optimization of dynamic scheduler for workloads #8 and #15 (in Fig.23 and Fig.24), we give a more detailed analysis on the efficiency of the dynamic scheduler.

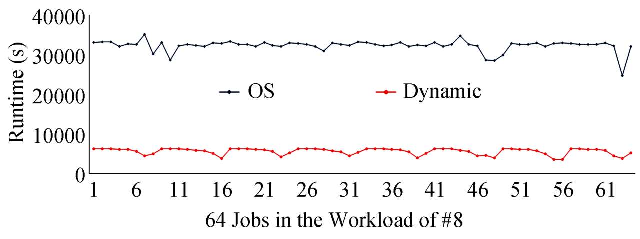


Fig.23. Improvement on the runtime for workload #8.

Workload #8 is the one that benefits the most from the dynamic scheduler. The workload is generated by duplicating Graph jobs that have relatively higher I/O activities and longer execution time of more than 2000s. Due to the calling of *mmap*, severe implicit I/O contention is observed throughout the entire job execution. For this kind of workloads, the dynamic scheduler plays a better role for mitigating the I/O contention. All the jobs show visible improvement in Fig.23.

Fig.24 displays contrasts for 64 jobs in workload #15. All jobs have relatively shorter execution time of about 30 s~200 s. Jobs in this workload have diverse I/O characteristics and with different sensitivities to I/O contentions. For example, swaptions (in black circle) from PARSEC have less I/O contentions and correspondingly they appear more stable in the experiments. For this kind of workload, relatively few options for optimizations are there. The comparisons among jobs in this workload reveal a very narrow gap between the

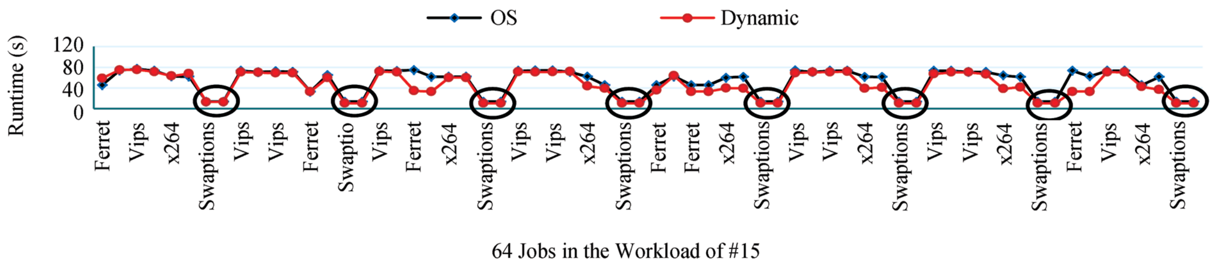


Fig.24. Improvement on the runtime for workload #15.

runtime using Linux scheduler and that optimized by the dynamic scheduler.

For workloads with more than 64 jobs, I/O contention decreases gradually after the first 64 jobs. If the workload is mostly composed of shorter jobs, trivial degradation may occur for this kind of workloads, e.g., in workload #2.

5 Related Work

Contention-aware scheduling has been investigated ever since last century. Performance degradation caused by contention in shared resources, such as last level cache (LLC)[23-25], memory bandwidth[8], and memory subsystem[9] on SMP or CMP has also been studied extensively. Some of the researches put more focuses on the optimizations of contentions for multithreaded applications[7,10]. Recent work in this area has started to focus on more practical issues of resource utilization on modern service platforms such as CMP and cloud computing[2,4]. These researches try to enhance the resource utilization by co-locating applications with complementary demand on system resources, e.g., one is CPU-intensive and the other is memory-intensive. Through estimation or mitigation of interference from shared-memory contention, these techniques can improve resource utilization without losing QoS. The research efforts[26-27] combine page coloring and XOR cache mapping to reduce row buffer conflicts due to inter-thread inference. A recent work[28] introduces an empirical model for predicting cross-core performance interference on multicore processors, which can further be used to guide co-runner-aware compiler optimizations[4,29], or some domain-specific optimizations[30-32], to make datacenter applications co-locate better.

Performance bottlenecks in I/O continue to be one of the hot research topics since last century. There have been many solutions proposed from different perspectives for better I/O performance. As an effective optimization, techniques using I/O scheduling policies can be divided into two major types: performance-oriented scheduling and fairness-oriented scheduling.

5.1 Performance-Oriented Scheduling

A lot of previous studies on mitigating I/O bottlenecks have concentrated on the performance of I/O devices. Under such premises, disk scanning is considered as the core reason for low I/O performance. Scheduling I/O operations to improve disk scanning is a kind of optimization that benefits from high concurrency among I/O operations[15-16]. Those schemes do not take I/O contention into consideration.

Longer disk scanning by noncontiguous I/O requests is one of the main reasons that cause poor disk performance. Optimization on the sequence I/O operations by data sieving can improve such I/O performance[17-18].

Disk caching in memory is an effective technique to speedup I/O performance. The work in [20] demonstrates several I/O optimizations with shared memory for specific languages, e.g., MPI-IO applications. Since optimization using shared memory will take some memory resources away from regular memory operations, a careful trade-off needs to be made. Workloads with Graph500 in our work are also optimized with disk caching for I/O operations. However, background implicit I/O activities still can cause severe I/O contention.

Research on I/O activities in virtual machines has also become a hot topic. The work in [33] focuses on I/O contention among multiple guest domains. The work points out that the fairness in I/O resource allocation may lead to poor performance due to the differences in I/O requests. The work in [34] points out a key shortcoming in the scheduler of current virtual machine monitors (VMMs) that may lead to communication behavior of applications. Solutions include techniques such as booking pages for communication, anticipatory scheduling for sender, in order to make VMM more aware of the characteristics of applications.

Most I/O schedulers focus on scheduling algorithms without taking the characteristics of applications into consideration. In fact, the applications may show different sensitivities to I/O performance. FIOS in [35] is a flash I/O scheduler that targets at solid-state drives (SSD) and takes both fairness and performance into consideration using timeslice-based heuristic. 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 can do well to some applications used to stall by writes.

I/O throttling is a kind of optimization for higher resource utilization, which is the most similar to our work from the perspective of coordinating I/O demands[36]. This technique always is applied in services which are comprised of tasks of different QoS. The study in [37] is a very recent work which exploits I/O throttling in MapReduce. However, it sacrifices low-QoS tasks to ensure the performance of high-QoS tasks.

5.2 Fairness-Based Scheduling

Software scheduling policies are always better choices for mitigating resource conflict, including I/O contention. Among all I/O schedulers, fairness-oriented I/O schedulers are 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[22,38]. The work in [22] gives a comparative study on all these policies. However, fairness-based schedulers often take little or no consideration in performance. Due to the lack of knowledge in the characteristics of applications, contentions of shared resources are difficult to resolve using fairness-oriented policies.

6 Conclusions

The efficiency of batch processing is attracting renewed interest on many modern service platforms such as clouds and clusters because the massive datasets need to be processed by many new applications. Multi-socket CMPs on those platforms also have created new challenges and opportunities for batch processing, for example, shared resources contentions such as I/O contentions that can lower the resource utilization of the platforms and the QoS for batch-mode services running in concurrent mode.

In this paper, the major causes of performance degradation due to I/O contention were identified and studied. A dynamic I/O-aware scheduling strategy was proposed to deal with those issues. It can improve the performance by regulating I/O contention and reducing the overhead caused by bursty I/O requests. The experimental results on the large-scale server of Dawning Linux Cluster show the effectiveness of such a strategy in improving the throughput of the I/O sensitive batch-mode workloads. Meanwhile, the slowdown of workloads and the average runtime of each user job can also benefit from such strategy.

Different applications have their different sensitivities to shared resource contention. Accommodating such sensitivities can make the scheduler more adaptive. This part of work is still in progress, and will be covered in our future work.

Our current scheduling policy aims primarily at co-running performance. Applications with higher I/O demands are more prone to be suspended. This tends to hurt the fairness, which is a very important and practical issue of cluster platforms. Our future work will consider more issues including fairness in our scheduling policy.

Acknowledgement We thank anonymous reviewers for their constructive and valuable comments. Our thanks also go to Ying Liu and Professor Shi-Guang Shan at Institute of Computing Technology, Chinese Academy of Sciences, for their technical feedback on earlier drafts of this paper.

References

[1] Armbrust M, Fox A, Griffith R *et al.* Above the clouds: A Berkeley view of cloud computing. Technical Report, UCB/EECS-2009-28, University of California, Berkeley, February 10, 2009.

[2] Mars J, Tang L, Hundt R *et al.* Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. the 44th Int. Symp. Microarchitecture*, December 2011, pp.248-259.

[3] Mishra A K, Hellerstein J L, Cirne W *et al.* Towards characterizing cloud backend workloads: Insights from Google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 2010, 37(4): 34-41.

[4] Tang L, Mars J, Soffa M L. Compiling for niceness: Mitigating contention for QoS in warehouse scale computers. In *Proc. the 10th Int. Symp. Code Generation and Optimization*, March 31-April 4, 2012, pp.1-12.

[5] Barroso L, Holzle U. The case for energy-proportional computing. *IEEE Trans. Computer*, 2007, 40(12): 33–37.

[6] Höelzle U, Barroso L A. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers, 2009.

[7] Snavely A, Tullsen D. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proc. the 9th ASPLOS*, November 2000, pp.234–244.

[8] Xu D, Wu C G, Yew P C. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proc. the 19th Int. Conf. Parallel Architectures and Compilation Techniques*, September 2010, pp.237–248.

[9] Zhuravlev S, Blagodurov S, Fedorova A. Addressing shared resource contention in multicore processors via scheduling. In *Proc. the 15th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, March 2010, pp.129–142.

[10] Gao L, Nguyen Q H, Li L *et al.* Thread-sensitive modulo scheduling for multicore processors. In *Proc. the 37th Int. Conf. Parallel Processing*, September 2008, pp.132-140.

[11] Gao L, Xue J L, Ngai T F. Loop recreation for thread-level speculation on multicore processors. *Software: Practice and Engineering* (*SPE*), 2010, 40(1): 45-72.

[12] Gao L, Li L, Xue J L *et al.* Loop recreation for thread-level speculation. In *Proc. the 13th Int. Conf. Parallel and Distributed Systems*, Dec, 2007, pp.1-10.

[13] Gao L, Li L, Xue J L *et al.* Exploiting speculative TLP in recursive programs by dynamic thread prediction. In *Proc. the 18th Int. Conf. Compiler Construction*, Mar, 2009, pp.78-93.

[14] Ghoshal D, Canon R S, Ramakrishnan L. I/O performance of virtualized cloud environments. In *Proc. the 2nd Int. Workshop on Data Intensive Computing in the Clouds*, November 2011, pp.71-80.

[15] Jain R, Somalwar K, Werth J *et al.* Scheduling parallel I/O operations in multiple bus systems. *Journal of Parallel and Distributed Systems*, 1992, 16(4): 352-362.

[16] Jain R, Somalwar K, Werth J *et al.* Heuristics for scheduling I/O operations. *IEEE Trans. Parallel and Distributed Systems*, 1997, 8(3): 310-320.

[17] Thakur R, Gropp W, Lusk E. Data sieving and collective I/O in ROMIO. In *Proc. the 7th Symp. Frontiers of Massively Parallel Computation*, February 1999, pp.182-189.

[18] Acharya A, Uysal M, Bennett R *et al.* Tuning the performance of I/O-intensive parallel applications. In *Proc. the 4th Workshop on I/O in Parallel and Distributed Systems*, May 1996, pp.15-27.

[19] Lin Z, Zhou S. Parallelizing I/O intensive applications for a workstation cluster: A case study. *ACM SIGARCH Computer Architecture News*, 1993, 21(5): 15-22.

[20] Hastings A, Choudhary A. Exploiting shared memory to improve parallel I/O performance. In*Proc. the 13th European PVM/MPI User's Group Conf. Recent Advances in Parallel Virtual Machine and Message Passing Interface*, September 2006, pp.212-221.

[21] Lameter C. Local and remote memory: Memory in a Linux/NUMA system. In *Linux Symp.*, July 2006, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.7986&rep=rep1&type=pdf, Nov. 2013.

[22] Shakshober D J. Choosing an I/O scheduler for Red Hat® Enterprise Linux® 4 and the 2.6 kernel. http://www.redhat.com/magazine/008jun05/features/schedulers/, Dec. 2012.

[23] Jiang Y L, Shen X P, Chen J *et al.* Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proc. the 17th Int. Conf. Parallel Architectures and Compilation Techniques*, October 2008, pp.220-229.

[24] Zhuravlev S, Saez J C, Blagodurov S *et al.* Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 2011, 45(1): Article No.4.

[25] Majo Z, Gross T R. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *Proc. the Int. Symp. Memory Management*, June 2011, pp.11-20.

[26] Mi W, Feng X B, Xue J L, Jia Y C. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. In *Proc. the 7th Int. Conf. Network and Parallel Computing*, September 2010, pp.329-343.

[27] Mi W, Feng X B, Jia Y C *et al.* PARBLO: Page-allocation-based DRAM row buffer locality optimization. *Journal of Computer Science and Technology*, 2009, 24(6): 1086-1097.

[28] Zhao J C, Cui H M, Xue J L *et al.* An empirical model for predicting cross-core performance interference on multicore processors. In *Proc. the 22nd Int. Conf. Parallel Architectures and Compilation Techniques*, September 2013, pp.201-212.

[29] Bao B, Ding C. Defensive loop tiling for shared cache. In *Proc. the IEEE/ACM Int. Symp. Code Generation and Optimization*, February 2013, pp.1-11.

[30] Cui H M, Wang L, Xue J L *et al.* Automatic library generation for BLAS3 on GPUs. In Proc. *the 25th IEEE Int. Symp. Parallel and Distributed Processing*, May 2011, pp.255-265.

[31] Cui H M, Xue J L, Wang L *et al.* Extendable pattern-oriented optimization directives. In *Proc. the 9th Annual IEEE/ACM Int. Symp. Code Generation and Optimization*, April 2011, pp.107-118.

[32] Cui H M, Yi Q, Xue J L, Feng X B. Layout-oblivious compiler optimization for matrix computations. *ACM Trans. Architecture and Code Optimization*, 2013, 9(4): Article No.35.

[33] Ongaro D, Cox A L, Rixner S. Scheduling I/O in virtual machine monitors. In *Proc. the 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Mar. 2008, pp.1-10.

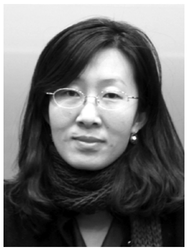
[34] Govindan S, Nath A R, Das A *et al.* Xen and co.: Communication-aware CPU scheduling for consolidated xen-based hosting platforms. In *Proc. the 3rd ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, June 2007, pp.126-136.

[35] Park S, Shen K. FIOS: A fair, efficient flash I/O scheduler. In *Proc. the 10th USENIX Conf. File and Storage Technologies*, February 2012, Article No.13.

[36] Ryu K D, Hollingsworth J K, Keleher P J. Efficient network and I/O throttling for fine-grain cycle stealing. In *Proc. the 2001 ACM/IEEE Conf. Supercomputing*, November 2001, Article No.3.

[37] Ma S, Sun X H, Raicu I. I/O throttling and coordination for MapReduce. Technical Report, Illinois Institute of Technology, 2012.

[38] Domingo D. Linux 5 IO tuning guide-performance tuning whitepaper for Red Hat Enterprise Linux 5.2. http://wenku.baidu.com/view/b7fc01ee5ef7ba0d4a733b74.html, August 2013.



Fang Lv participated in the Advanced Compiler Technology Laboratory (ACT) of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in 2001. She is now a Ph.D. candidate of ICT, CAS. Her research interests include performance analysis, compiler optimizations, and resource utilization for large-scale servers.

Hui-Min Cui participated in the Advanced Compiler Technology Laboratory (ACT) of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 2003. She is now an associate professor of ICT, CAS. Her research interests include programming language and optimization.

Lei Wang participated in the Advanced Compiler Technology Laboratory (ACT) of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 2002. Her research interests include programming language and optimization.

Lei Liu is now a Ph.D. candidate of ICT, CAS. His research interests include operating system and memory system design and implementation.

Cheng-Gang Wu received his Ph.D. degree in computer architecture from ICT, CAS, in 2001. Now he is an associate professor and Ph.D. supervisor of ICT. His research interests include compiler optimization and binary translation.

Xiao-Bing Feng received his Ph.D. degree in computer architecture from ICT, CAS, in 1999. Now he is a professor and Ph.D. supervisor of ICT. His research interests include compiler optimization and binary translation.

Pen-Chung Yew received his Ph.D. degree in computer science from University of Illinois at Urbana-Champaign, in 1981. He is a professor of the Department of Computer Science and Engineering, University of Minnesota at Twin Cities, USA. His research interests include high-performance and low-power multi-core architectures, compilation techniques that support multi-threading and speculation, dynamic compilation, binary translation, parallel machine organizations, and OS for multi-core embedded systems.