

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

Fang Lu<sup>1,2</sup> (吕方), Hui-Min Cui<sup>1</sup> (崔慧敏), Member, CCF, Lei Wang<sup>1</sup> (王蕾), Lei Liu<sup>1,2</sup> (刘磊), Cheng-Gang Wu<sup>1</sup> (武成岗), Member, CCF/ACM/IEEE, Xiao-Bing Feng<sup>1</sup> (冯晓兵), Member, CCF/ACM/IEEE, and Pen-Chung Yew<sup>3,4</sup> (游本中), Fellow, IEEE

<sup>1</sup>*State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China*

<sup>2</sup>*University of Chinese Academy of Sciences, Beijing, China*

<sup>3</sup>*Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, USA*

<sup>4</sup>*Institute of Information Science, Academia Sinica, Taiwan, China*

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

**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 could impede both the co-running performance of batch jobs and the system throughput seriously. In this

---

\* PAPER CLASSIFICATION

Supported by National Basic Research Development Program of China (973 program) under grant 2011CB302504, National High Technology Research and Development Program of China (863 program) under grant 2012AA010902, and National Natural Science Foundation of China under the grant 61202055, 60925009, 60921002, 61100011 contemporarily.

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 setup our environment on an 8-socket, 64-core server in Dawning Linux Cluster. 15 workloads ranging from 8 jobs to 256 jobs are evaluated. Our experimental results showed 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 could 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 (key words)

## 1. Introduction

Cluster, datacenter and cloud computing have emerged as major computing platforms for the ever expanding applications today<sup>[1]</sup>. On such platforms, *batch-mode processing* (or *batch processing*, for short) is still one of the major service patterns<sup>1,2</sup>. It is non-interactive and has very different demands on both performance and QoS<sup>[1]</sup>. 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)<sup>3</sup>, 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

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

<sup>2</sup>Microsoft. Batch Applications—the hidden asset, August 2013.

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

into the computing platforms, 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<sup>[3]</sup>.
- Scalability in Data Sets. due to the development of web applications, big input data sets have become one of the most remarkable characters.

These requirements have created higher demands on the server system capacity, and thus stimulated the development of parallel server systems. Servers have evolved from the former SMP architectures to the current CMP 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 could seriously impede the co-

running performance and the overall system throughput. Hence, resolving such contentions have become one of the most important issues for such systems<sup>[2-13]</sup>.

For many applications, shared I/O-related resource is a significant contention point<sup>[14]</sup>. In fact, I/O bottleneck has been known on parallel computing systems for some time<sup>[14-18,19]</sup>. 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 I/O performance by rescheduling I/O requests<sup>[15-18]</sup>, or using shared memory as disk cache<sup>[20]</sup>. 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, the workload slowdown, and the average runtime for each job. The evaluations are setup on an 8-socket, 64-core CMP server node. 15 workloads, which range from 8 jobs to 256 jobs dynamically, are experimented on this platform. Experimental results show that the proposed scheduler could 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 could be obtained.

Improving the co-running performance of CMP systems has many practical implications for large web applications with expanding data

sets. From this perspective, we made the following contributions in this paper:

- A methodology has been proposed that could 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 summarizes our work and presents some possible future work.

## **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's 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 could 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 lower costs of other resource contentions.
- *Sustainability.* The overall resource requirements of a job would 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) would keep a job's resource consumption as "local", i.e., its memory would be allocated to the local memory of its core(s) on NUMA architecture<sup>[21]</sup>. Furthermore, the allocation strategy also keeps the resource consumption as "local" throughout the job execution. Therefore, the premises of CMP stacking could be satisfied, and consequently, inter-socket I/O contention turns into the critical issue for co-running performance degradation. We would give a further discussion for the leading role of the inter-socket I/O contention in Section 2.3.

CMP stacking is setup to illuminate the negative impacts of I/O contentions on batch processing. However, our solution would target to all concurrent jobs, **which** includes not only inter-socket I/O contentions, but also **that** inside sockets.

**Table 1.** Description for applications

	<b>Application</b>	<b>Type</b>	<b>Description</b>
<b>Real user application</b>	Paper Similar (abbr. PS)	2-threaded	A program which compares a paper with the other K papers concurrently, while K is 2 in our work.
	Kmeans clustering (abbr. KM)	8-threaded	A key algorithm from data mining which partitions n observations into k clusters.
<b>Graph500</b>	graph	1-thread	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.
<b>PARSEC benchmarks</b>	x264	1-thread	encoding video
	vips	1-thread	image processing library
	freqmine	1-thread	data mining problem
	bodytrack	1-thread	tracker of the 3D pose of a human body
	raytrace	1-thread	Tracing the path of light and generating images

## 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  $p$ ,

$T_{alone}^j$  : It is the execution time of  $Job^j$  when it runs alone, i.e. without any resource contention.

$T_{1-c}^j$  : It is 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 other CMPs concurrently. It is different from  $T_{alone}^j$  because of possible contentions within the CMP.

$T_{k-c}^j$  : It is 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 could be measured by the difference between  $T_{1-c}^j$  and  $T_{k-c}^j$ . It could be clearly ascribed to the inter-socket I/O contentions. We use normalized run time for the comparison as in (1).

$$\text{normalized\_runtime} = \frac{T_{k-c}^j}{T_{1-c}^j} \quad (1)$$

## 2.2 Benchmarks and Platforms

Before presenting experiments with our methodology, 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 data sets. In this section, we use duplicated copies of a benchmark with the same input sets to demonstrate I/O contentions. This could 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<sup>4</sup>. 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 would lead to continuous implicit I/O behaviors.
- Benchmarks from the Princeton Application Repository for Shared-Memory Computers (PARSEC 3.1)<sup>5</sup>. 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 and different periodic I/O characteristics. Contentions from explicit I/O APIs could be demonstrated with this package. Among all data sets, the medium data set of *simlarge* and the largest data set 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 would 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 jobs on a server node. Note that our CMP stacking method and our

---

<sup>4</sup> <http://www.graph500.org/>, August 2013.

<sup>5</sup> <http://parsec.cs.princeton.edu/>, August 2013.

scheduling solution themselves do not have these limitations. We would 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 would 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.

**Table 2.** Workloads and Input Set

	Benchmark	Input set
#1	x264	simlarge
#2	x264	native
#3	graph	-s 22 -e 16
#4	graph	-s 22 -e 18

## 2.3 Performance Degradation from I/O

### Contention

#### 2.3.1 Influences from Inter-Socket I/O

##### Contentions

In this section, 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 with either duplicated explicit I/O jobs (x264) or duplicated implicit I/O jobs (graph) in this section. 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 could observe the severe performance impacts from inter-socket I/O contentions.

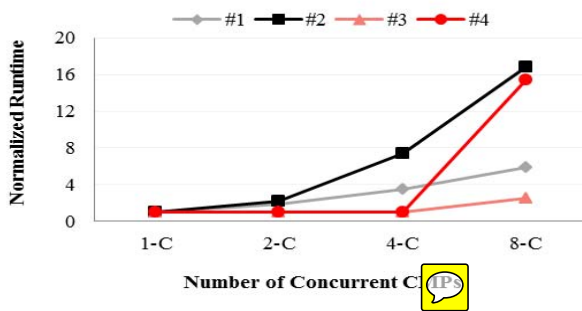


Fig.1. Inter-socket performance impacts from I/O contentions while increasing the number of CMPs.

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

There are four curves in Figure 1. Each curve stands for the normalized runtime of each job in the four steps (denoted as  $k$ -c in Figure 1, or  $k$ -CMP in later figures). Take workload #1 in Table 2 as an example, the average runtime for a x264 job is 7 seconds (denoted as 7s) on 1 CMP, while it degrades to 122s when co-running with other 56 jobs on seven other CMPs. It is about 16x degradation due to the increased I/O contention.

Data in these figures 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 would suffer from I/O contentions.

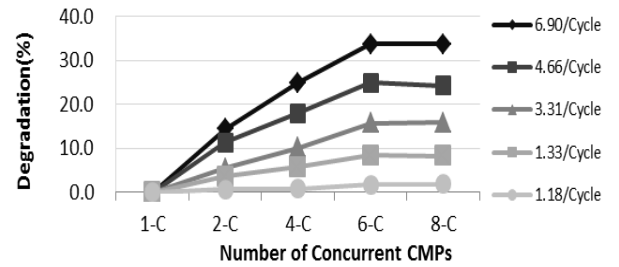


Fig.2. Performance degradation from inter-socket CC costs in  $k$ -CMP configuration,  $k=1, 2, 4, 6, 8$ .

### 2.3.2 Discussion for Other Influences

It is worth noticing that 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.

Figure 2 displays experiments for the inter-socket CC costs on our Intel Nehalem server system, which uses MESIF<sup>6</sup> as its cache protocol. The maximum CC costs is about 33.7%, which are generated from the most serious LLC misses (6.90/Cycle per CMP) during 8-C co-running. This is much more trivial than 2x ~ 16x degradations from I/O contentions in Figure 1.

<sup>6</sup> [http://en.wikipedia.org/wiki/MESIF\\_protocol](http://en.wikipedia.org/wiki/MESIF_protocol), August 2013.

---

Procedure LLCMISS\_Pressure

---

```

1: #define ITERATION 1000 //repeat the experiments
   #define CACHELINE 64 // the cache line size
2: #define COL CACHELINE/sizeof(int)
3: #define ROW (MEM_SIZE/COL*sizeof(int))
4: int a[ROW][COL].b[ROW][COL];
5: #define nops 5000 //intervals between two successive
6: reads
7:
8: //memory allocation
9: Initialization(a, b);
10: for (iter=0; iter<ITERATION; iter++){
11: //LLC cache miss kernel
12:   for (i=0; i<ROW; i++) {
13:     b[i][0] = a[i][0];
14:     //use nops to adjust the density of LLC misses
15:     for (k=0; k<nops;k++) {
16:       asm( "nops" );
17:     }
18:   }
19: }

```

---

Fig.3. LLC-miss kernel.

The experiments in Figure 2 are performed with CMP stacking, while each CMP runs LLC-miss intensive kernels as in Figure 3. All experiments comply with the two constrains 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 could get different LLC miss rate inside a CMP. CMP LLC miss rate at 6.90/cycle is the maximum number which can be generated on the server. During the process of CMP stacking up, intensive LLC misses in a CMP would bring forth cache coherency information globally, which result in inter-socket performance influences. In such circumstance, CC cost is the most outstanding inter-socket performance

hip Mu

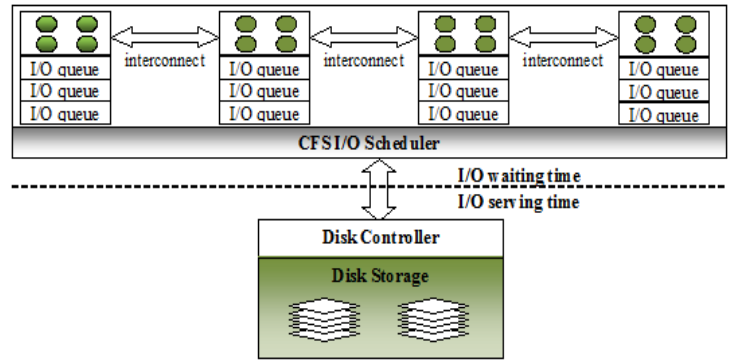


Fig.4. Two parts for the latency of I/O requests.

influences. 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 more 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.

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 analyses of global I/O contention are 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 show in Figure 4.

$$IO\_Latency = Latency_{Serving} + Latency_{Waiting} \quad (2)$$

$Latency_{Serving}$  is the actual service time of a I/O request by the I/O device. This latency is decided by both the decision making of disk controller and the specific I/O devices.

$Latency_{Waiting}$  is the handling time of the software scheduler for a I/O request, and the time costs in I/O queues. The default task and I/O scheduler are **completely fair schedule( CFS ) policies** on our Linux OS<sup>[22]</sup>. It also includes the extra overhead resulted from bursts of I/O requests.

I/O latency is calculated with these two parts as in (2).

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,  $IO\_Latency$  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 service 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.

##### 2.4.2.1 Analyses for Explicit I/O Behavior

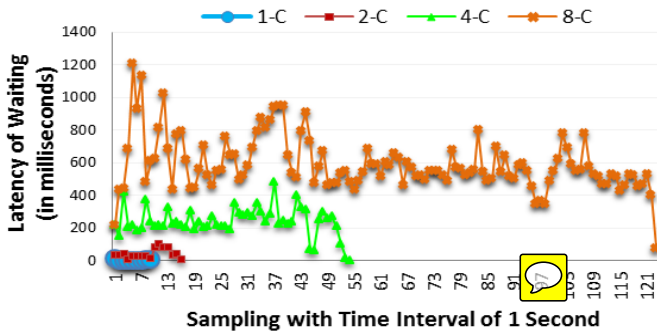


Fig.5. Average waiting time in  $k$ -CMP configuration,  $k=1, 2, 4, 8$ .

Take workload #1 as an example, Figure 5 and Figure 6 show  $Latency_{Waiting}$  and  $Latency_{Serving}$  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 Figure 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. 590ms vs. 4.55ms, respectively.

A more detailed comparison is made between these two kinds of latencies, and the results are shown in Figure 7. The contrast clearly shows

that I/O contention has a much more severe impact on the waiting time than on service time. Bursty I/O requests in a time interval that cannot be handled due to limited I/O resources will

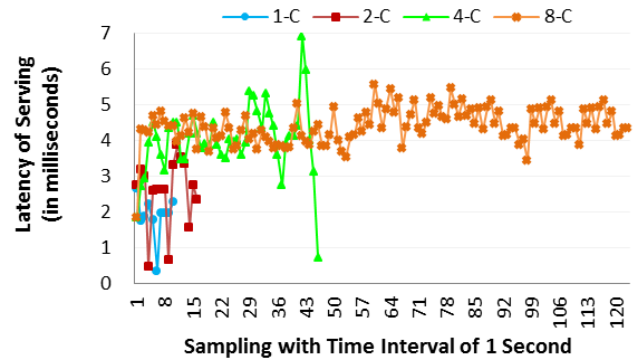


Fig.6. Average serving time in  $k$ -CMP configuration,  $k=1, 2, 4, 8$ .

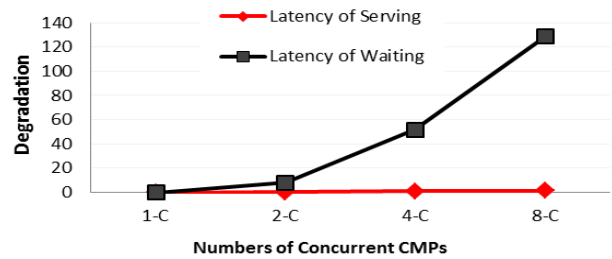


Fig.7. The trend of I/O performance degradation for waiting time and service time in  $k$ -CMP configurations,  $k=1, 2, 4, 8$ .

accumulate, and have a severe impact on other co-running batch jobs in I/O queues.

From these figures we could 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 would suffer.

#### 2.4.2.2 Analyses for Implicit I/O Behavior

From Figure 1, we observed that workload #3 and #4 behave differently to workload #1 and #2.

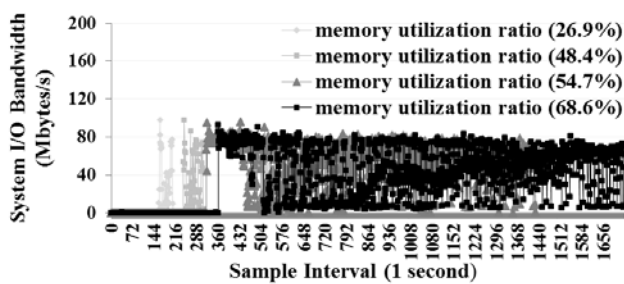


Fig.8. Correlation between memory utilization ratio and the severity of I/O contentions.

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 would happen, which is

always accompanied 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 4 more graph workloads and each workload has 64 graph jobs. Through varying the value of  $s$  and  $e$  as in Table 1, different graphs could be generated, which would result in different memory sizes to associate file operations.

In our experiment in Figure 8, the memory utilization ratio ranges from 26.9% to 68.6% (the system memory is 256G in total). The figure draws an interesting conclusion: the more memory we use, the more I/O quantities would 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 (30000s). 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.

From the above analyses we could learn 2 points:

- The root reason for degradations is thus the much higher I/O demands vs relatively low disk serving capacity, which results in a 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 would 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 could 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<sup>[8]</sup> and LLC-aware scheduling<sup>[9]</sup>. 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 have implemented the proposed dynamic I/O-aware scheduler (called dynamic scheduler in the rest of the paper) as a user-level scheduler



in Linux. However, it can be applied to other OS with a slight modification in its system call interfaces.

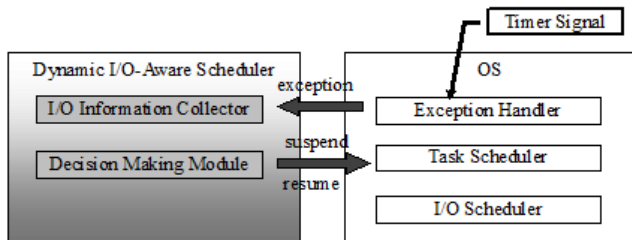


Fig.9. Overall framework for the dynamic I/O-aware scheduler.

The framework of the dynamic scheduler and its interface to OS are shown in Figure 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 could be collected through reading I/O files (under `/proc/pid/io`). After the sampling, I/O related analysis and the scheduling policies could 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

---

**Procedure 1 Dynamic\_IO\_Aware\_Scheduling**

---

```

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

```

---

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

I/O operations once again. In this way, the dynamic scheduling policy still could make use of all existing optimizations for the tasks and I/O operations in OS.

The framework of the user-level dynamic scheduler is also presented as a pseudo code in Figure 10. It is registered as a signal handler, `sigalarm_handler_IO`, in Line 4 of the pseudo code. This program is triggered at fixed interval (timeslice, or quantum), which is defined by a tunable variable, *TIMESLICE*, in Line 3.

### 3.1.1 Overall Approach

As shown in the left of Figure 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 Figure 11. The two major

---

**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:  * would 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.

---

**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.

parts correspond to Line 6 and Line 14 respectively in the figure.

- 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, dynamic scheduler could 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 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 wouldn't 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 Figure 12 parses the device 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 would 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.

### 3.3 Decision-Making Module

The Decision-Making Module targets to two key issues: when to schedule and how to schedule. First, the module would 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 the bandwidth exceeds the boundary, two kinds of measures, either suspension or resumption, are 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* (Mbytes/second), as a guidance for scheduling.

#### 3.3.2 How to Schedule

Different jobs show different sensitivity 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 designed a module, named *Sensitivity\_RANKING*, which is composed with

a file reading kernel. As Figure 13 shows, each

```

Sensitivity_RANKING Module
1: #define block_size 32 //data size for one read
   #define nops 5000 //intervals between two successive
2: reads
3:
4: //read file only for one time in case of page caching
5: while (!eof(file)) {
6:     fgets(file, block_size);
7:     //use nops to adjust the bandwidth density
8:     for (i=0; i<nops; i++) {
9:         asm ( "nop" );
10:    }
11: }

```

Fig.13. Kernel of Sensitivity\_RANKING.

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 reading and the number of concurrent kernels (inserting nops), we could get different ranks of simultaneous I/O requests.

We illustrate the relation between simultaneous I/O quantities and their performance influences through 12 groups of experiments in Figure 14. Tasks in these 12 groups of experiments are composed with duplicated Sensitivity\_RANKING kernels which has 1.5 Mbytes ~ 26.4Mbytes/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

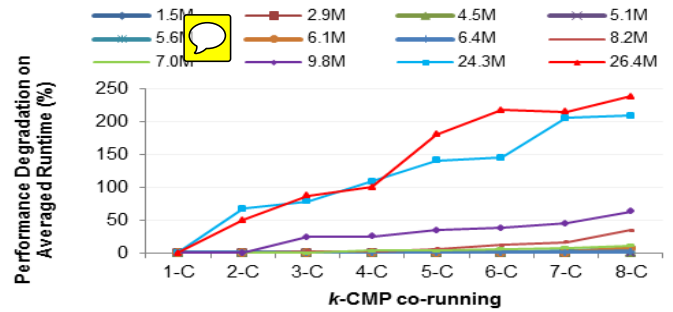


Fig.14. Averaged performance degradation in  $k$ -CMP configuration,  $k = 1 \sim 8$ .

8 CMPs. From the figure we could learn that, the higher the averaged I/O quantity is, the easier co-running performance degradation happens. For the group with 26.4Mbytes/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 lower bandwidth than 5.63Mbytes/s, I/O contentions will not result in co-running performance influences.

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. Kmeans) is worth more attention than those tasks of lower-I/O quantity (e.g. graph).

Regulation on high-I/O quantity tasks would reduce the I/O bandwidth pressure, which are beneficial for more low-I/O quantity jobs.

The pseudo code for Decision-Making Module is shown in Figure 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.

In an interval, if I/O requests exceed the upper bound of the I/O capacity, the policy would 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 16 to Line 29 in Figure 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*, would be made. Its pseudo code is shown in Line 31 to Line 41. Jobs that were suspended will be resumed for better utilization of I/O capacity. In case of too aggressive contentions from resumption, we'll let go a job at a time.

Moreover, to avoid excessive idleness in a

---

**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 would
7:  * pick out jobs which has the most I/O qualities
8:  * on each CMP separately
9:  */
10: for ith 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 exceed the
15:   * upper capacity of I/O, suspending part of the jobs
16:   */
17:   while (Overall_IO > IO_BOUND_INTERVAL) {
18:     for ith 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]=head[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 ith CMP in CMPs {
37:     while (head[i] != NULL) {
38:       if ( Status(head[i]) is SUSPENDED) {
39:         Status(head[i]) = RESUMING;
40:         break;
41:       }
42:       head[i]= head[i]->next;
43:     }
44:   }
45: }

```

---

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

CMP, a procedure, *Sorting\_Jobs\_in\_Each\_CMP* is used for making such a decision. It is shown in Line 2. It sorts all jobs according to their I/O requirements in a descending order. The policy

would 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 Decision-Making Module.

#### 3.4.1 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 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 could 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 1s. For jobs with longer execution time (more than 1000s), e.g. benchmarks in *graph500*, we use a coarse-grained *TIMESLICE*, and it is set at 20s.

#### 3.4.2 Setting of *IO\_BOUND\_INTERVAL*

Since 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 *40Mbytes/s*.

### 3.5 Scheduling Overheads

Overheads of our proposed dynamic I/O-aware scheduling not only determine 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: 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 would not exceed 1% when *TIMESLICE* varies from 1s to 20s.

### 3.6 Discussions for Insufficiencies

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 would 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. Our future work would consider more issues including fairness in our scheduling policy.

## 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 with and without our scheduling policy.

To study its efficiency more comprehensively, performance improvement on the throughput of the workloads are examined. The throughput is calculated by the **number of jobs in a workload**, and the execution time of the workload,  $T_{workload}$ , as shown in (3). For a fairer comparison of the optimization effects on all concurrent jobs, *slowdown* and *average runtime* are evaluated either. The *slowdown* of a job,  $Job^j$  is calculated by the ratio of the runtime when it runs alone to

**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	parsec+graph	mixed	128
	#12	parsec+graph	mixed	128
	#13	parsec + real	mixed	128
	#14	parsec + real	mixed	128
	#15	parsec	simlarge	64

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* is calculated with all jobs'  $T_{k-c}^j$  values as in (5).

$$\text{Throughput} = \frac{\text{Jobs}}{T_{\text{workload}}} \quad (3)$$

$$\text{Slowdown} = \sum_{j=1}^N \frac{T_{\text{alone}}^j}{T_{k-c}^j} \quad (4)$$

$$\text{Aver - runtime} = \frac{\sum_{j=1}^N T_{k-c}^j}{N} \quad (5)$$

#### 4.1 Workload

Two different types of workloads are studied in this section:

- *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 data set, and the corresponding number of jobs in each workload, denoted as batch length. At the beginning of execution for each workload, the server would full-load 64 single-threaded jobs on all 64 cores or 8 multi-threaded jobs on



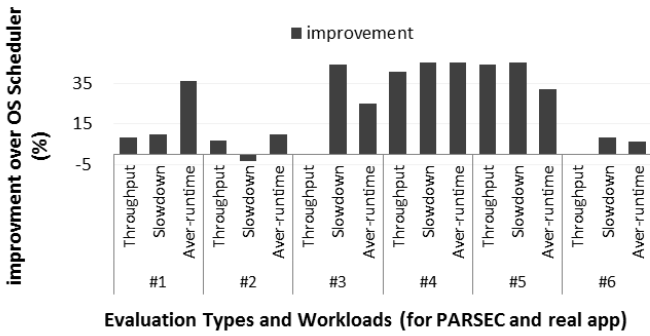


Fig.16. Performance improvement for D-TYPE workloads (explicit I/O jobs: PARSEC and real user app).

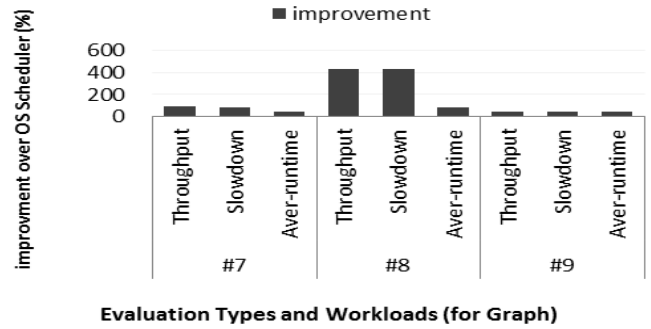


Fig.17. Performance improvement for D-TYPE workloads (implicit I/O jobs: Graph).

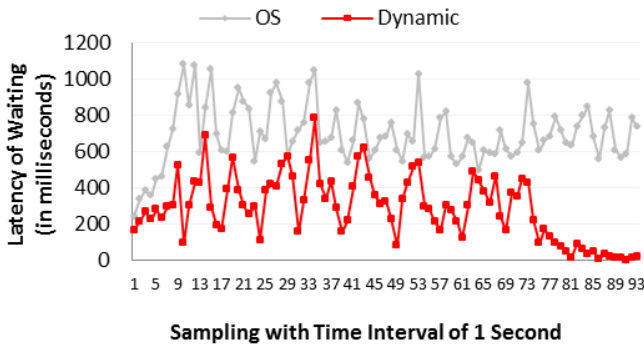


Fig.18. Improvement on waiting time for workload #1.

all 8 CMPs at most for each batch. No core-sharing is permitted in our experiment. For M-type workloads, each job would take a separate CMP and at most 8 jobs could be handled in a batch. More jobs will be served whenever there are cores released and become idle.

As in Table 3, experiments with D-type are composed with 9 workloads. Explicit I/O interfaces are evaluated through PARSEC benchmarks (workload #1 ~ #3) and real user

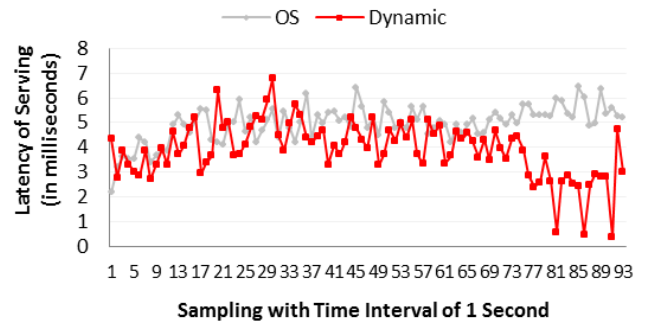


Fig.19. Improvement on serving time for workload #1.

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 could 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 Evaluation for D-TYPE workloads

### 4.2.1 Performance Results

Figure 16 shows the performance improvement for 6 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 could 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.

Compared with workloads of PARSEC, workloads with concurrent graph traversal algorithms have obtained much more improvements from the dynamic scheduling in Figure 17. The throughputs for them range from 45% to 431%. The improvements on average

slowdown range from 40% to 433%. The average runtime could be improved by 45% to 82% with our dynamic scheduler. For #8 which includes 64 jobs, our dynamic scheduler could obtain the most improvements on the throughput by 431%.

### 4.2.2 Performance Analysis

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

$Latency_{Serving}$  could also benefit from the dynamic scheduler as shown in Figure 19. It achieves 21% improvement with the dynamic scheduler, from 4.95ms to 3.9ms.

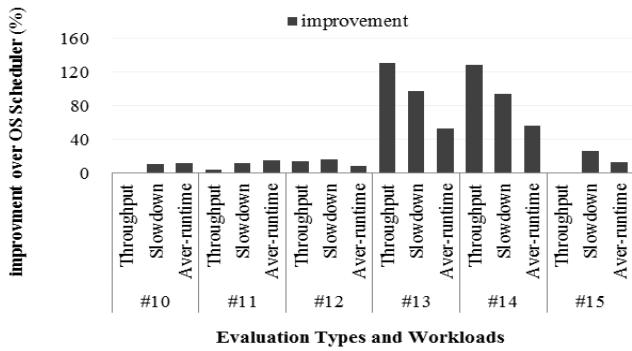


Fig.20. Performance improvement on M-TYPE workloads.

### 4.3 Evaluations for M-TYPE workloads

#### 4.3.1 Performance Results

In this section, 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.

Figure 20 shows the improvement for M-TYPE workloads. All these workloads could achieve improvement on the slowdown, ranging from 10.8%-97.7%. The improvement on average runtime ranges from 9.3% to 56.2%.

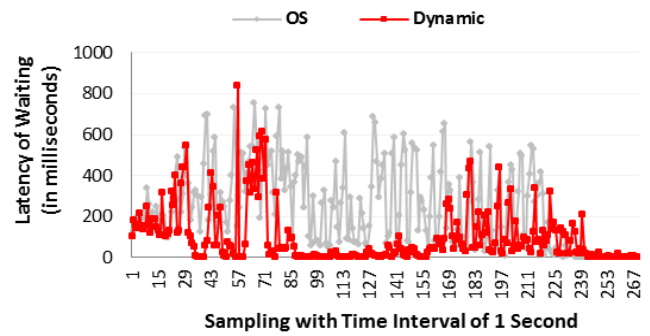


Fig.21. Improvement on waiting time for workload #11.

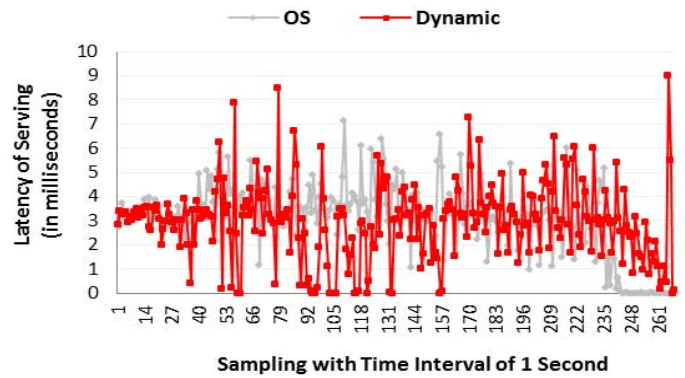


Fig.22. Improvement on service time for workload #11.

Except workload #10, all other workloads see improvements in throughput at 1.5% ~ 131.2%, respectively. Although workload #10 doesn't see obvious improvement in throughput, neither does it suffer with the dynamic scheduler.

#### 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,

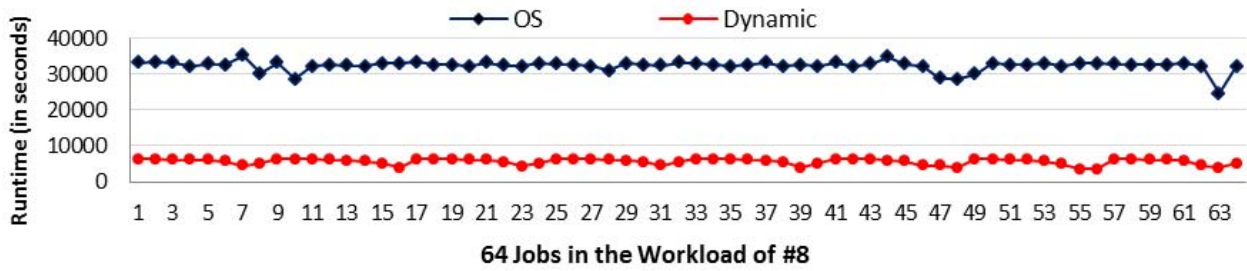


Fig.23. Improvement on the runtime for workload #8.

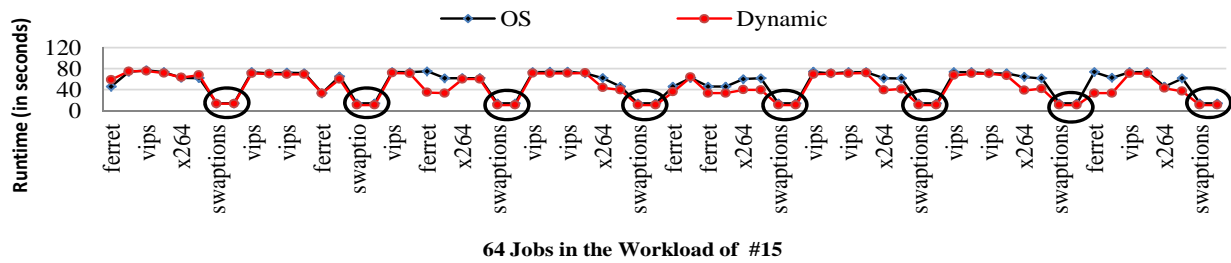


Fig.24. Improvement on the runtime for workload #15.

$Latency_{Serving}$  and  $Latency_{Waiting}$ , are made between Linux scheduler with and w/o optimization of our dynamic scheduler. Data sampled at interval of 1s are collected for these two kinds of latency and the improvements are shown in Figure 21 and Figure 22.

Figure 21 shows the effect of the dynamic scheduler on  $Latency_{Waiting}$ , average waiting time from all sampled intervals is improved by 60% compared with that with Linux scheduler, from 243.6ms to 98ms.

$Latency_{Serving}$  also could benefit slightly from the dynamic scheduler as shown in Figure 22. The average service time is improved by 4.6% compared with that with Linux scheduler, from 3.05ms to 2.91ms.

#### 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 with and w/o the optimization of dynamic scheduler for workloads #8 and #15 (in Figure 23 and Figure 24), we give a more detailed analysis on the efficiency of the dynamic scheduler.

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 could play a better role for mitigating the I/O contention. All the jobs show visible improvement in Figure 23.

Figure 24 displays contrasts for 64 jobs in workload #15. All jobs have relatively shorter execution time of about 30s-200s. Jobs in this

workload have diverse I/O characteristics and with different sensitivities to I/O contention. For example, swaptions (in black circle) from PARSEC has 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 runtime using Linux scheduler and that optimized by the dynamic scheduler.

For workloads with more than 64 jobs, I/O contention would decrease 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)<sup>[23-25]</sup>, memory bandwidth<sup>[8]</sup>, and memory subsystem<sup>[9]</sup> on SMP or CMP have also been studied

extensively. Some of the researches put more focuses on optimizations of contentions for multithreaded applications<sup>[7,10]</sup>. 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<sup>[2,4]</sup>. These work tries to enhance 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<sup>[26][27]</sup> combines page coloring and XOR cache mapping to reduce row buffer conflicts due to inter-thread inference. A recent work<sup>[28]</sup> 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<sup>[4][29]</sup>, or some domain-specific

optimizations<sup>[30][31][32]</sup>, 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 work on mitigating I/O bottlenecks have concentrated on the performance of I/O devices. Under such premises, *disk scanning* is considered the core reason for the 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<sup>[15][16]</sup>. Those schemes did 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* could improve such I/O performance<sup>[17][18]</sup>.

Disk caching in memory is an effective technique to speedup I/O performance. The work in [20] demonstrated 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 could lead to poor performance due to the differences in I/O requests. The work in [34] pointed out a key shortcoming in the scheduler of current virtual

machine monitors (VMM) that may lead to communication behavior of applications. Solutions include techniques such as booking pages for communication, anticipatory scheduling for sender, etc. 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 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 both for better performance with a preference to reads using timeslice-based scheduling. This could do well to some applications that 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<sup>[36]</sup>. This technique always is applied in services which are comprised by tasks of different QoS. The study in [37] 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.

## 5.2 Fairness-based Scheduling

Software scheduling policies are always a better choice 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 policies in mainstream OS such as Linux<sup>[22,38]</sup>. 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, contention of shared resources are difficult to resolve using fairness-oriented policies.

## 6. Conclusions

The efficiency of batch processing is attracting renewed interests on many modern service platforms such as clouds and clusters because of the massive data sets 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 could 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 are identified and studied. A dynamic I/O-aware scheduling strategy is proposed to deal with those issues. It could improve 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 could also benefit from such a strategy.

Fairness is a very important and practical issue on cluster platforms. Further researches on this problem are our next step and would be covered in our future work.

### Acknowledgement

We thank anonymous reviewers for their constructive and valuable comments. Our thanks also go to Professor Shiguang Shan, Ying Liu at ICT 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*, February 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*, March 2010, 37(4):34-41.
- [4] Tang L, Mars J, and Soffa M L, Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proc. 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] Hoelzle U, Barroso L A. The Datacenter as a computer: An introduction to the design of warehouse-scale machines. Morgan and

- Claypool Publishers, 2009.
- [7] Snaveley A, Tullsen D. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proc. the 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, November 2000. 28(5): 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, 38(1): 129–142.
- [10] Gao L, Nguyen Q H, Li L, Xue J 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 and Ngai T F. Loop recreation for thread-level speculation on multicore processors. *Software -- Practice and Engineering (SPE)*, John Wiley & Sons, 2010, 40(1):45 – 72.
- [12] Gao L, Li L, Xue J L and Ngai T F. Loop recreation for thread-level speculation. In *Proc. the 13th Int. Conf. Parallel and Distributed Systems*, December 2007, pp.1-10.
- [13] Gao L, Li L, Xue J L and Ngai T F. Exploiting speculative TLP in recursive programs by dynamic thread prediction. In *Proc. the 18th Int. Conf. Compiler Construction*, March 2009, pp. 78 – 93.
- [14] Ghoshal D, Canon R S, Ramakrishnan L. I/O performance of virtualized cloud environments. In *Proc. the 2th 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. *IEEE Trans. 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. the 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*, 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.
- [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/>, June 2005.
- [23] Jiang Y L, Shen X P 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*, November 2011. 45(1): Article No. 4.
- [25] Majo Z et al. 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 and 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, Chen L and Xue J L. 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, Feng X B 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.
- [29] Bao B and 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, Yang Y, and Feng X B. 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, Yang Y, Feng X B et al. Extendable pattern-oriented optimization directives. In *Proc. the 9th Annual IEEE/ACM Int. Symp. on 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*, January 2013, 9(4): Article 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*, March 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 and Shen K. Fios: A fair, efficient flash I/O scheduler. In *Proc. the 10th USENIX Conf. File and Storage Technologies*, February 2012. pp. 13-13.
- [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, p.3.
- [37] Ma Siyuan, Sun Xian-He et al. 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://www.docstoc.com/docs/111590044/red-hat-enterprise-linux-5-io-tuning-guide>, August 2013.



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



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

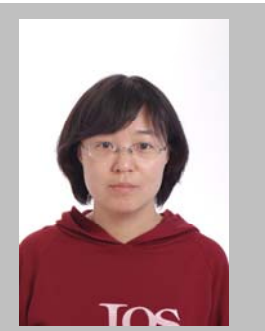


**Hui-Min Cui** participated in the Advanced Compiler Group in 2003. She is now an Associate Professor. Her research interests include programming language and optimization.

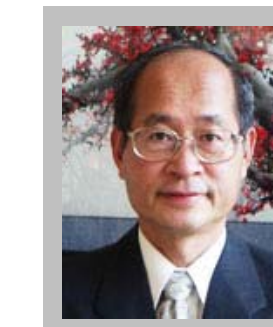


**Cheng-Gang Wu** received his Ph.D. degree in computer architecture from ICT in 2001. Now he is an Associate Professor and PhD supervisor. His research interests include compiler optimization and

binary translation.



**Lei Wang** participated in the Advanced Compiler Group in 2002. Her research interests include programming language and optimization.



**Pen-Chung Yew** received his Ph.D. degree in computer science from University of Illinois at Urbana-Champaign, Computer Science in 1981. He is a Professor of



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

the Department of Computer Science and Engineering, University of Minnesota at Twin-Cities. 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.