Container-Based Job Management for Fair Resource Sharing

Jue Hong[1,6], Pavan Balaji[2], Gaojin Wen[3], Bibo Tu[4], Junming Yan[5], Chengzhong Xu[6], and Shengzhong Feng[6]

[1]Oracle Corporation
[2]Mathematics and Computer Science Division, Argonne National Laboratory
[3]National Laboratory of Pattern Recognition, Institute of Automation, CAS
[4]Institute of Information Engineering, CAS
[5]Tencent, Inc.
[6]Shenzhen Institutes of Advanced Technology, CAS

Abstract. Achieving fair resource sharing is rapidly becoming an essential requirement in cluster computing systems. Although many fair scheduling algorithms have been proposed in recent decades, controlling resource sharing among jobs on servers remains a challenging problem that, if not handled well, may result in chaotic resource contention and service-level agreement violation of jobs. To address this problem, we propose a resource container–based job management approach for fair resource sharing. In our approach, we first design and implement a general container-based job management module, providing lightweight and fine-grained resource allocation and isolation for job execution. With this module, we propose a resource-aware management scheme to enable fair resource sharing in job scheduling and dispatching. We conduct experiments by implementing the proposed module and applying the scheme on `TCluster`, a self-developed cluster computing system of a worldwide top Internet corporation. Results show that our approach performs well in guaranteeing fair resource sharing with negligible overhead.

1 Introduction

Unlike past decades when batch job submission prevailed, today various types of jobs are being deployed simultaneously by using cluster computing systems. Each job usually has a specific service-level agreement (SLA), which can be mapped to resource requirements such as CPU, memory, or I/O bandwidth. How to fairly partition and share such resources among running jobs in a cluster, is key to guaranteeing SLAs.

Many job-scheduling algorithms based on fair strategies have been proposed in recent decades [9,15,19,25,26,28]; they determine which job should be scheduled to run according to the job's resource requirements and the available quota of job owners. Controlling resource sharing among running jobs on servers remains a challenge, however; and if not handled well, chaotic resource contention

* This work was partially supported by NSFC 61202417, 60903116 and 61003063.

and SLA violation of jobs may result. Two popular approaches for resource sharing are process-level sharing and virtual machine (VM)-level sharing. In a process-level approach, it is difficult to track and control the resources used by programs with multiple processes or with a newly spawned process while running. Moreover, fine-granularity isolation of important resources such as the CPU and network bandwidth cannot be guaranteed [10]. The VM-level approach [16] relies on various virtualization technologies such as XEN [11], VMware [8], and KVM [4]. It typically runs programs inside a VM configured with the required resources. Although the VM-level approach provides good resource-isolation, it usually incurs high overhead for controlling, setting up, and running programs [25], especially in the case of small short-lived programs.

In this paper we propose to use a resource container to build a job management approach for fair resource sharing on cluster computing systems. Resource containers are based on OS-level virtualization that has been popularized in recent decades; examples are LRP [10], VServer [25], OpenVZ [7], and Linux Container (LXC) [5]. Resource containers partition the resources of a single operating system into isolated groups and can give programs the illusion of running on a separate machine. By running instructions native to the core CPU without any special interpretation mechanisms, resource containers introduce little or no overhead. Moreover, modern container technology such as LXC can provide fine-grained resource partitioning, for example, assigning half a CPU to a program. The features of low cost and fine-grained partitioning make resource containers particularly suitable for job execution in cluster computing systems, in which each server hosts a homogeneous operating system. Furthermore, recent container technologies such as LXC provide good support for resource management of multiprocess programs.

The contributions of our paper are twofold:

- We propose a general container-based job management module (CJMM) as the kernel of our fair resource sharing approach, providing resource isolation and a sharing mechanism for running jobs on one server.
- Based on CJMM, we present a resource-aware management scheme, including resource-aware job scheduling and dispatching, that enables fair resource sharing on cluster computers.

We note that our resource-aware management scheme provides only a framework for implementing resource fairness using the proposed module. The job scheduling and dispatching algorithms can be chosen as needed and thus are not the focus of this paper. As the underlying container technology, we use LXC, which is a recent implementation of OS-level virtualization and has been included in the mainstream Linux kernel. Several other container technologies (e.g., OpenVZ and VServer) are also popular and have performance comparable to that of LXC. However, they all require customized Linux kernels. Moreover, issues of security, stability, and maintenance make them less competitive, compared to LXC, in production environments. The proposed module and scheme are implemented on `TCluster`, a self-developed cluster computing system of a

worldwide top Internet corporation. Experiment results show that our approach performs well in enabling fair resource sharing among running jobs. Also, the proposed scheme helps improve the resource utilization of the whole cluster.

The rest of this paper is organized as follows. We introduce related work in Section 2. The detailed design and implementation of the job management module are discussed in Section 3. We present the resource-aware management scheme in Section 4. In Section 5 we present our experimental results. We conclude in Section 6 with a brief look at future work.

2 Related Work

This section briefly discusses related work in two areas: fair scheduling algorithms and resource containers.

2.1 Fair Scheduling Algorithms

Fair job scheduling is an important research problem for cluster computing systems. One approach to fair scheduling is lottery and stride scheduling [26], proposed by Waldspurger, using both random and deterministic manner to allocate resource for jobs. In [25], Soltesz et al. proposed using a hierarchical token bucket to assign quotas among jobs in order to achieve fairness. Recently, with the popularization of Hadoop, some simple yet effective fair scheduling algorithms based on a round-robin approach have been proposed [9, 28]. Isard et al. proposed a fair scheduling named Quincy [19] for the Microsoft Dryad cluster by modeling the scheduling as a network flow problem. For multiple resources cases, Ghodsi et al. proposed a dominant resource fairness (DRF) scheduling algorithm [15]. However, all these algorithms focus only on how to determine a fair order of jobs running in a cluster. They do not provide a mechanism for controlling resource sharing of running jobs on servers.

2.2 Resource Containers

The concept of resource containers was first proposed by Gaurav Banga et al. [10]. Similar concepts on non-Linux system are Solaris Zone and FreeBSD Jail. Early resource containers on Linux systems are OpenVZ, VServer, and FreeVPS. Most of these technologies require customized Linux kernels, however, and thus are unacceptable in many product scenarios, especially for large corporations. On the other hand, LXC has been combined into the mainstream Linux kernel; and, different from traditional machine-level virtualization, LXC requires neither instruction-level emulation nor just-in-time compilation.

The building block of LXC's resource sharing is the `cgroup` framework and various subsystems [2, 22]. The `cgroup` framework allows LXC to track, control, and audit the resources used by process groups. Subsystems in current mainstream Linux kernels, such as cpu, cpuset, cpuacct, memory, and net_cls, enable LXC to support sharing and accounting on such resources. For resource isolation,

LXC employs the kernel namespace [6] and the `pivot_root` system call. Additionally, the LXC toolkit provides a `liblxc` library and a series of userspace tools for container management. Given the advantages and the popularity of LXC, we use it as the underlying container technology in this paper.

3 Container-Based Job Management

In this section we describe the design and implementation of the container-based job management module (CJMM).

3.1 Design of Job Management Module

The architecture of a typical cluster computing system can be separated into two parts: a central manager that responsible for global control, and an execution engine that is responsible for running and managing jobs on cluster servers. The CJMM is plugged into the execution engine, taking over the job execution, resource provisioning, and isolation. More specifically, the execution engine passes the job information to our job management module. The module then creates a container, assigns resources, and starts the job inside it. A handler of the running job (e.g., the PID) will be returned to the execution engine for monitoring and controlling. Each job runs in its own container, unaware of jobs in other containers.

The CJMM consists of two components: `JobManager` exposes a simple, high-level, container-based job management interface to the execution engine; and `Container` represents the data structure and operations of a real container.

JobManager `JobManager` starts jobs and manages their containers. Job's data (e.g., executing function and arguments) is passed in via a `JobPtr` object when starting. The PID of the job is stored and returned for monitoring and control. While the configuration of a container (CPU shares, memory limits, etc.), is stored in a `ContainerConf` object.

In addition, `JobManager` assigns and accounts for the resource usage on the server. When a job is being started, `JobManager` checks whether the required resource can be allocated according to the available amount. If allocatable, a container is created and the required resources are deducted from the available amount; they will be returned when the job finishes. Otherwise, the request is declined. The resource requirement is stored in a `ResInfo` entity, an example of which is "resinfo.cpus = 0.5; resinfo.memory = 3GB...".

Container The `Container` represents the data structure and the operations of a real container. For the sake of extensibility, `Container` is designed as a virtual class. Two key operations of `Container` are task execution and resource usage information retrieval. The execution command of a job will be passed to the real container's executing API via `RunTask` method. Real-time resource usage information can be obtained via `GetUsage`. We have designed a class `LXCContainer` inherited from `Container` using the LXC technology.

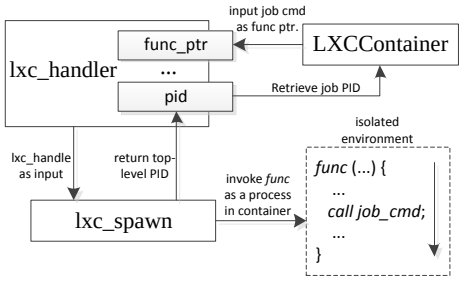


Fig. 1. Modified job-startup mechanism in LXCContainer.

3.2 Implementation Issues

We have implemented a prototype of the job management module using C++ based on the LXC 0.7.2. In this version, the low-level control of programs outside their containers is obscured. And the LXC toolkit has no direct method to obtain the real-time resource usage. Next we describe our solutions to these two issues, including the job-startup mechanism and resource usage information retrieval for a container.

Job-Startup Mechanism LXC's application execution mode allows starting a program in a container through the `lxc-execute` command. However, because of the hierarchical PID namespace, the PID of a running program in a container is usually different from that outside. Neither the LXC userspace tools nor the open API of `liblxc` provides a direct way to get the outer-layer PID. Moreover, in LXC's application mode, a process called `lxc-init` will be started with PID 1, acting as the parent of all other processes in the container. As a result, the ending signal and exit code of a job cannot be captured by the execution engine, which is undesirable for the JobManager.

To handle these problems, we hacked the program-starting mechanism of LXC's application mode in the `LXCContainer` as in Figure 1. We directly use the `lxc_handler` structure in LXC's source code as well as the open API of `liblxc`. We first replace `lxc-init` with a function executing the job's commands directly (e.g., `func`), which makes the job itself the root process of a container. Then we store a pointer of the function in the `lxc_handler` and invoke the `lxc_spawn` method in `liblxc`'s open API, which will set up a container and start the job finally. The job's PID in the top-level namespace is stored in the `lxc_handler` and passed to the execution engine.

Usage Information Retrieval We use the statistical functions of subsystems with the `cgroup` framework to obtain real-time usage information of containers.

For CPU usage, we employ the subsystem cpuacct, which accumulates the CPU time of all tasks in a group in cpuacct.usage. Given a timing period, we can calculate the real-time CPU usage of a container by dividing its CPU time by the

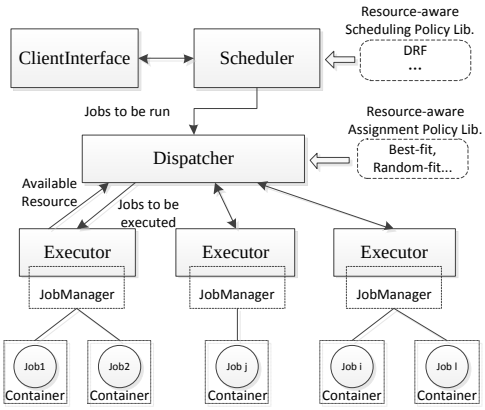


Fig. 2. Architecture of resource-aware TCluster.

elapsed time. For simplicity, we let the timing period be the time between the most recent two calls of the `GetUsage` method. For memory usage, we employ the memory subsystem, and access the memory.usage_in_bytes to obtain memory usage of all tasks in a group.

4 Resource-Aware Management Scheme

Based on the CJMM, we propose a management scheme and show its application on the `TCluster` system.

4.1 TCluster

`TCluster` is a typical cluster computing system for job processing and cluster resource management (Figure 2), which consists of four main modules: a *ClientInterface* for submitting jobs, a *scheduler* for scheduling jobs, a *dispatcher* for assigning jobs to servers, and an *executor* for running jobs on each server. Using a process-level isolation and job-number based scheduling, the original `TCluster` is not resource-aware and is unable to conduct SLA-guaranteed job processing. To remedy this situation, we apply our resource-aware management scheme on `TCluster`.

4.2 Resource-Aware Scheme on TCluster

Our proposed scheme consists of a resource-aware scheduler, a resource-aware dispatcher, and a container-based execution engine, which can be applied on any cluster computing system.

Resource-Aware Scheduler The resource-aware scheduler employs the required resources of each job as a key metric while scheduling. We require users to declare the resource requirements for their submitted jobs and then employ the DRF scheduling algorithm in `TCluster`'s scheduler, which works well in multiresource scenarios. This renders `TCluster`'s scheduler able to generate fair scheduling results in terms of multiple resources (e.g., both CPU and memory).

Resource-Aware Dispatcher The resource-aware dispatcher tries to find a good match between jobs' required resources and the available resources on servers in assignment. We use a simple matching policy as blow among various candidates (e.g., [12–14, 17, 18, 21, 23, 24, 27]).

Let $\overrightarrow{R_{req}} = \{x_1, x_2, \ldots, x_i\}$ be the required resources for a job, where x_i is the amount of the ith resource. Also let $\overrightarrow{R_k} = \{y_1, y_2, \ldots, y_i\}$ be the available resources on server k. Both x_i and y_i are normalized values. We then find the server m with the minimum Euclidean distance (Affinity Number) of $\overrightarrow{R_{req}}$ and $\overrightarrow{R_k}$ for the given job. This simple policy works well in scenarios where short-lived jobs dominate and the scale of cluster is large (e.g., over 5,000 servers), reducing both computation complexity (e.g. the backfilling with online bin-packing [20]) and resource fragments .

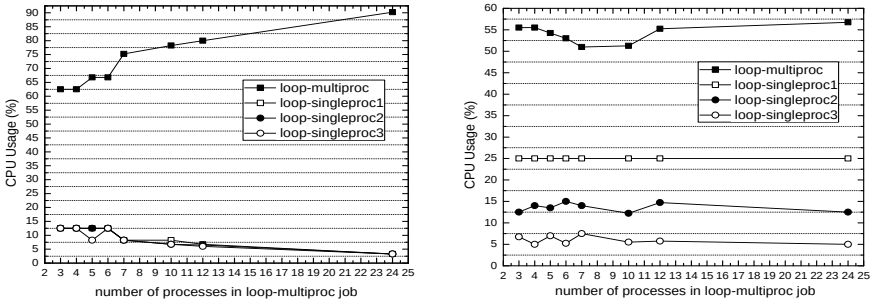
Container-Based Execution Engine We modified the executor of `TCluster` by merging the `JobManager` class, taking over the job execution, and providing resource guarantees and isolation. The executor reports the available resources to the dispatcher in each heartbeat to facilitate the resource-aware assignment.

Additionally, resource requirements of each job is expressed using XML and passed via the ClientInterface. An example is like "cpu_num=0.5; memory=0.5GB".

5 Performance Evaluation

In this section we evaluate our approach via experiments. The OS used on each server is SUSE Linux Enterprise 11-sp1 with kernel version of 2.6.32.29-x86_64. The version of the LXC toolkit used is 0.7.2.

To evaluate resource sharing, isolation, and utilization performance, we set up a cluster consisting of six servers on the same rack and connected with 1G Ethernet, and deploy `TCluster` on them. Each server is equipped with four Intel 3 GHz Xeon CPUs and 2 GB memory. One server hosts the ClientInterface, the scheduler, and the dispatcher, and the other five servers run the container-based executor. Here we focus on the CPU and memory resources. To see the CPU, we employ two CPU-intensive programs: "loop-singleproc" and "loop-multiproc". Both programs repeatedly execute some increment instruction (e.g.,"i++") with single and multiple process respectively. And we increase the number of processes in "loop-multiproc" gradually (i.e., 3, 4, 5, 6, 7, 10, 12, and 24). For each version of `TCluster`, one job of "loop-multiproc" and three jobs of "loop-singleproc" are



(a) In original TCluster. (b) In modified TCluster.

Fig. 3. CPU usage of each job in TCluster (ratio 8:4:2:1).

submitted simultaneously. In the modified `TCluster`, the CPU resource ratio set to the "loop-multiproc" job, and the other three "loop-singleproc" jobs is 8:4:2:1. For memory testing, we use a memory-intensive program that continuously allocates and touches memory.

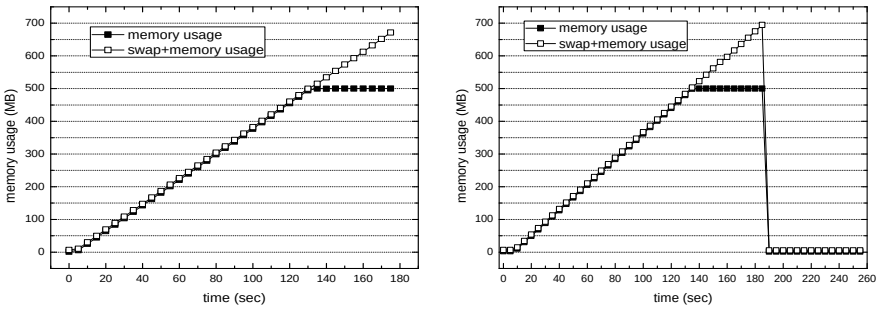
In the overhead evaluation, we use an experimental IBM x3550 server. The server is equipped with a quad-core Xeon E5504 2 GHz CPU and 15 GB memory. We use GeekBench [3] and UnixBench [1] to evaluate the performance of CPU, memory, disk I/O, and system operations.

5.1 Resource Sharing and Isolation

Figure 3(a) shows that the "loop-multiproc" job consumes a significant portion of CPU, indicating that the original `TCluster` cannot ensure sharing and isolation of CPU resource among jobs. While in the modified container-based `TCluster`, the CPU times of the four jobs are approximately in accordance with the preset ratio of 8:4:2:1 (Figure 3(b)), which means the container-based approach helps guarantee fair sharing of the CPU.

Since no memory control mechanism was in original `TCluster`, we only evaluate the modified `TCluster` with the memory-intensive benchmark. We first set the memory limit of the job to 500 MB with unlimited swap space, and then limit the total space of swap and memory to 700 MB. In both cases the used physical memory never exceed 500 MB (Figure 4(a) and Figure 4(b)). In former the job can still obtain the memory since the swap space is unlimited, while in latter the amount of used swap and physical memory drops to zero quickly once the limit is reached. The reason is that in the memory subsystem, a default out-of-memory (OOM) behavior is to kill the programs.

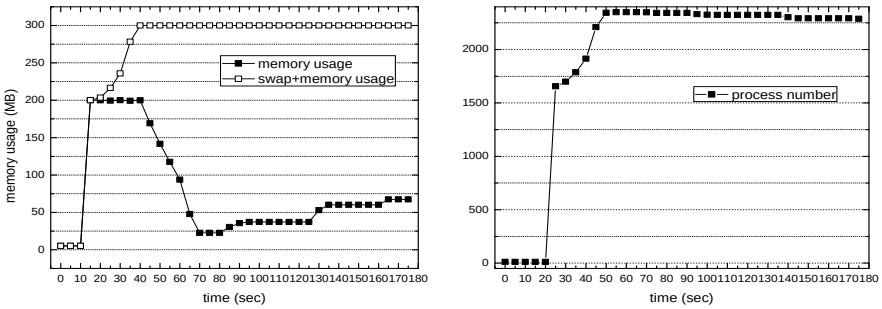
Next we show with our approach, the famous "forking attack" can be alleviated. We implement a program called "bomb," which keeps the forking process using a nonpaused and infinite loop. We submit "bomb" to both versions of `TCluster`, and establish SSH sessions to the server on which the "bomb" job is assigned to see the isolation effect. On the original `TCluster`, when the "bomb"



(a) Limited memory with unlimited swap.

(b) Limited memory and limited swap.

Fig. 4. Memory usage of the experimental job in modified TCluster.



(a) Memory usage of the bomb job.

(b) Process number in system.

Fig. 5. Memory usage of bomb job and process number in system.

begins running, the SSH session quickly becomes unresponsive. On the modified `TCluster` with memory limit of 200 MB and a total limit for swap and memory of 300 MB, both memory usage and process number are well controlled (Figure 5(a) and Figure 5(b)), and the SSH session is completely responsive. The reason is that the default OOM killer of the memory subsystem keeps killing the processes forked by "bomb" and releases system resource.

5.2 Resource Utilization

Here we compare the resource utilization under the FF, RF in original `TCluster` and affinity number-based best-fit (ANBF) in our resource-aware scheme, described in Section 4.2.

The total available resources (CPU (cores) and memory (MB)) of the experimental cluster are shown in Table 1. For each server, we reserve 0.2 CPU for the executor process itself. The workload we use consists of a series of production jobs with different resource requirements (Table 2).

Table 1. Available resources of cluster.

Server ID	CPU(s)	Memory
1	3.8	1847
2	3.8	1851
3	3.8	1858
4	3.8	1859
5	3.8	1855

Table 2. Information of experimental jobs.

Job ID	CPU(s)	Memory
1	1.2	1000
2	2.5	900
3	2.5	1200
4	3.0	800
5	3.0	1500
6	0.7	300
7	1.0	600
8	1.7	800
9	1.1	900
10	0.5	800

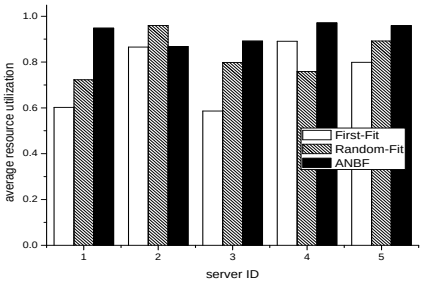


Fig. 6. Average resource utilization of each server with different policy.

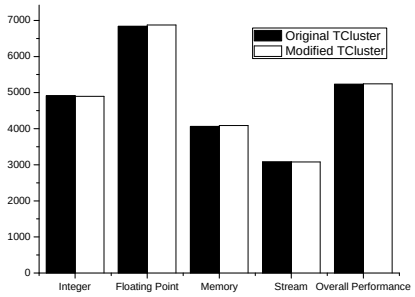


Fig. 7. CPU and memory overhead (Higher score is better).

We submit all ten jobs to the modified `TCluster` with FF, RF, and ANBF, and the available resources of each server is shown in Table 3. And the status of submitted jobs in `TCluster` with each assignment policy is: FF {running 8, pending Jobs (8,9)}, RF {running 9, pending Job (9)}, ANBF {running 10, no pending}. This result indicates that with our resource-aware scheme, `TCluster` can produce fewer resource fragments and thus improve resource utilization. We also calculate the average resource utilization of each server with $avg\_util = 0.5 * cpu\_util + 0.5 * mem\_util$, and the results are shown in Figure 6. We can see that in most cases, our approach with the ANBF policy can produce higher resource utilization on cluster servers (over 85%).

5.3 Overhead

First we analyze the overhead of CPU and memory. From Figure 7 we see no noticeable disparity between the two versions of `TCluster`. Hence our LXC-based approach has negligible overhead for CPU or memory-intensive programs. Then we observe the disk I/O overhead by comparing the data from UnixBench. From Table 4 (higher score is better) we can see that our approach causes at most 1.78% degradation compared with that of the original `TCluster`. Finally we observe the overhead on microsystem through UnixBench as well. The observed items and scores are shown in Table 5. The modified `TCluster` incurs minor

Table 3. Free resources on each server.

Server ID	First-Fit		Random-Fit		ANBF	
	CPU(s)	Mem(MB)	CPU(s)	Mem(MB)	CPU(s)	Mem(MB)
1	1.9	547	1.6	247	0.3	47
2	0.3	351	0.2	51	0.9	51
3	1.8	658	0.8	358	0.3	258
4	0.3	259	1.1	359	0.1	59
5	0.8	355	0.3	255	0.2	55

Table 4. Disk I/O overhead.

	Original TCluster	Modified TCluster	Gain
File Copy 1024 bufsize 2000 maxblocks	766476.3	761287.11	-0.68%
File Copy 256 bufsize 500 maxblocks	226551.2	222517.2	-1.78%
File Copy 4096 bufsize 8000 maxblocks	1609916.2	1606169	-0.23%

Table 5. System operation overhead.

	Original TCluster	Modified TCluster	Gain
Pipe Throughput	1642770.8	1646751.1	0.24%
System Call Overhead	2771390	2771562.9	0.01%
Process Creation	14374.9	13785	-4.10%
Pipe-based Context Switching	259403.2	226531.7	-12.67%
Shell Scripts (1 concurrent)	6270.2	6345.1	1.19%
Shell Scripts (8 concurrent)	2153.7	2171	0.80%

overheads in almost all items except the process creation and the pipe-based context switching.

6 Conclusion and Future Work

Although many fair scheduling algorithms have been proposed for cluster computing systems, few suitable mechanisms exist that control resource sharing among jobs on servers. To address this problem, we introduced a container-based job management approach. We first designed and implemented the CJMM to control the resource sharing and isolation for jobs on servers. Based on CJMM, we then proposed a resource-aware management scheme to enable fair resource sharing. We experimented with our approach on `TCluster`, a self-developed cluster computing system for a worldwide top Internet corporation. Results show that our approach performs well in providing fair resource sharing at a very low overhead, as well as a higher resource utilization of above 85%. An adaptive and automatic reconfiguration mechanisms and strategies will be our future work.

References

1. byte-unixbench. http://code.google.com/p/byte-unixbench/
2. Cgroup. http://www.kernel.org/doc/documentation/cgroups/cgroups.txt
3. Geekbench. http://www.primatelabs.ca/geekbench/
4. KVM. http://www.linux-kvm.org/

5. Linux container. http://lxc.sourceforge.net/
6. Linux kernel namespace. http://lxc.sourceforge.net/index.php/about/kernel-namespaces/
7. OpenVZ. http://download.openvz.org/doc/openvz-intro.pdf
8. Vmware. http://www.vmware.com/
9. Hadoop fair scheduler. http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html (2010)
10. Banga, G., Druschel, P., Mogul, J.C.: Resource containers: A new facility for resource management in server systems. In: OSDI. pp. 45–58 (1999)
11. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP. pp. 164–177 (2003)
12. Chang, F., Ren, J., Viswanathan, R.: Optimal resource allocation in clouds. In: ICCC (2010)
13. Diaz, C.O., Guzek, M., Pecero, J.E., Danoy, G., Bouvry, P., Khan, S.U.: Energy-aware fast scheduling heuristics in heterogeneous computing systems. In: HPCS. 2011
14. Gambosi, G., Postiglione, A., Talamo, M.: Algorithms for the relaxed online bin-packing model. SIAM Journal on Computing 30(5), 1532–1551 (2000)
15. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI (2011)
16. Grit, L., Irwin, D., Marupadi, V., Shivam, P., Yumerefendi, A., Chase, J., Albrecht, J.: Harnessing virtual machine resource control for job management. In: VTDC (2007)
17. He, Y., Elnikety, S., Sun, H.: Tians scheduling: Using partial processing in best-effort applications. In: ICDCS (2011)
18. Huang, Q., Huang, T.: An optimistic job scheduling strategy based on QoS for cloud computing. In: ICISS (2010)
19. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: SOSP. pp. 261–276 (2009)
20. Lee, C.C., Lee, D.T.: A simple on-line bin-packing algorithm. J. ACM 32, 562–572 (Jul 1985)
21. Liu, H., Abraham, A., Hassanien, A.E.: Scheduling jobs on computational grids using a fuzzy particle swarm optimization algorithm. In: FGCS (2009)
22. Menage, P.B.: Adding generic process containers to the linux kernel. In: OLS (2007)
23. Pinel, F., Pecero, J.E., Bouvry, P., Khan, S.U.: A two-phase heuristic for the scheduling of independent tasks on computational grids. In: HPCS. 2011
24. Sanders, P., Sivadasan, N., Skutella, M.: Online scheduling with bounded migration. Journal of Mathematics of Operations Research 34(2) (2009)
25. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: EuroSys. pp. 275–288 (2007)
26. Waldspurger, C.A.: Lottery and stride scheduling: Flexible proportional-share resource management (1995)
27. Wang, M., Meng, X., Zhang, L.: Consolidating virtual machines with dynamic bandwidth demand in data centers. In: INFOCOM (2011)
28. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user mapreduce clusters. Tech. Rep. UCB/EECS-2009-55, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html (2009)