Hindawi
Mathematical Problems in Engineering
Volume 2018, Article ID 1438792, 18 pages
https://doi.org/10.1155/2018/1438792

Research Article

Optimizing Job Coscheduling by Adaptive Deadlock-Free Scheduler

Zhishuo Zheng[1,2] **Deyu Qi**[1] **Mincong Yu**[1] **Xinyang Wang**[1] **Naqin Zhou**[3] **Yang Shen**[1] **and Jing Guo**[1]

[1]*School of Computer Science and Engineer, South China University of Technology, Guangzhou, China*
[2]*Department of Physics and Information Engineering, Guangdong University of Education, Guangzhou, China*
[3]*Cyberspace Institute of Advanced technology, Guangzhou University, Guangzhou 510006, Guangdong, China*

Correspondence should be addressed to Zhishuo Zheng; z.zhishuo@mail.scut.edu.cn

Received 20 January 2018; Revised 3 July 2018; Accepted 26 July 2018; Published 15 August 2018

Academic Editor: Emilio Insfran Pelozo

It is ubiquitous that multiple jobs coexist on the same machine, because tens or hundreds of cores are able to reside on the same chip. To run multiple jobs efficiently, the schedulers should provide flexible scheduling logic. Besides, corunning jobs may compete for the shared resources, which may lead to performance degradation. While many scheduling algorithms have been proposed for supporting different scheduling logic schemes and alleviating this contention, job coscheduling without performance degradation on the same machine remains a challenging problem. In this paper, we propose a novel adaptive deadlock-free scheduler, which provides flexible scheduling logic schemes and adopts optimistic lock control mechanism to coordinate resource competition among corunning jobs. This scheduler exposes all underlying resource information to corunning jobs and gives them necessary utensils to make use of that information to compete resource in a free-for-all manner. To further relieve performance degradation of coscheduling, this scheduler enables the automated control over the number of active utensils when frequent conflict becomes the performance bottleneck. We justify our adaptive deadlock-free scheduling and present simulation results for synthetic and real-world workloads, in which we compare our proposed scheduler with two prevalent schedulers. It indicates that our proposed approach outperforms the compared schedulers in scheduling efficiency and scalability. Our results also manifest that the adaptive deadlock-free control facilitates significant improvements on the parallelism of node-level scheduling and the performance for workloads.

1. Introduction

In recent years, the great rapid growth of sensors devices, high-speed search engines, and social networks has produced huge amount of data. Additionally, many scientific simulations in critical areas, such as climate sciences, astrophysics, computational chemistry, computational biology, and high-energy physics, are becoming increasingly data intensive [1]. The above applications manipulate a large amount of data that is compared with the amount of computation they perform and often transfer large volume of data to or from storage systems. Because the value of these applications is determined by the data quantity and the speed of producing results, a number of Data-Intensive Scalable Computing (DISC) [2] systems have been developed. These DISC systems are enabling IT solutions for different fields because of their great potential in reducing the operating expenses and management overheads; i.e., DISC systems provide a shared elastic computing infrastructure to accommodate multiple applications. Although these systems are scalable, DISC systems fail to take full advantage of these resources, which leads to the requirement for larger clusters to improve performance. Such poor efficiency results in the energy inefficient problem that increases DISC system providers' budget and CO2 footprint [3]. As a result, the efficiency of resource utilization in data centers has drawn much attention in recent years.

It is noteworthy that each node has already utilized large scale processors cores, multilane high-speed cards, hundreds of gigabytes of memory, and terabytes of disks [4]. In a nutshell, a single node in DISC systems is increasing reminiscent of a networked distributed system. The key issue is that the traditional node-level operating systems have not

been designed with current hardware trends in minds and they fail to make good use of underlying hardware resources. As a consequence, the individual nodes in the data center tend to run far below their peak performance capabilities [5], which contribute to the poor efficiency of resource usage in DISC systems. Previous study [6] points out that corunning different job together is capable of improving resource utilization and therefore reduces hardware quantity requirement for applications. For example, the current winner of the Indy GraySort sorting benchmark is Tencent Sort [7] that has adopted fewer nodes than previous winners. This sort achieves CPU utilization peak at 70% by overlapping sort and networking stages. Its result has shown that although the CPU utilization typically ranges from 10 to 30% in most of time (similar results have also been demonstrated in other data centers [8]), it is able to achieve better CPU utilization by allowing multiple jobs to coexist on the same node. But coexisting jobs will produce different issues. On one hand, resource sharing among different jobs on the same node will make the node-level scheduling problem more complicated: a wider range of requirements and policies have to be taken into account, as servers terminate user requests or provide infrastructure services like storage, naming, or locking. On the other hand, corunning jobs may compete for the shared resources, which may result in performance degradation.

Existing approaches to address the above issues mainly fall into two categories: (1) architectural-level solutions to provide isolation among applications [9–11] and (2) system-level solutions to partition underlying resources among jobs and provide flexible scheduling logic schemes to accommodate the dynamics of application resource demand [12–17]. Within these two categories, architectural-level solutions remain under development by hardware vendors and the modifications to hardware require many changes to system-level software and therefore incur high implementation cost. The second approach, developing a scheduler to facilitate coscheduling, is regarded as a lightweight approach and has therefore attracted considerable research attention. Our research work mainly focuses on the second category.

The most general choice for a node-level operating system for DISC systems is Linux. Its scheduler applies multipath monolithic scheduling, which schedules a job once a time and supports different scheduling logic schemes by providing multiple code paths. This scheduler runs separate scheduling logic schemes for different task types. It is a tough job to add new policies and specialized implementations to multipath monolithic schedulers. Some researchers advocate that two-level schedulers will be a good choice to support flexible scheduling logic schemes. This type of schedulers has a central coordinator (first-level scheduler) to decide how many underlying resources can be distributed or offered to multiple parallel user-level schedulers (i.e., second-level scheduler that is able to implement distinct scheduling logic independently), as in Tessellation [16, 18] and Akaros [19]. These schedulers allow the allocation of resources to different user-level schedulers dynamically. Two-level schedulers do appear to support flexible scheduling policies. However, each user-level scheduler can only perceive limited resources. Moreover, if multiple user-level schedulers compete for the same shared resources, the two-level scheduling will resolve it with traditional locking concurrency control that is easy to produce deadlock without careful treatment. All the above conditions make two-level approaches hard to place jobs that are difficult to schedule or make decisions that need to access the entire underlying resources. The fact is that resource utilizations (especially for CPU) are still low most of the time [20] and it is reported that one-third of the available CPU cycles remain idle [21].

To support flexible scheduling logic schemes and address competition for the shared resources without performance degradation, we present a novel approach that applies an adaptive optimistic lock mechanism built on shared state of underlying hardware resources. This approach offers both flexibility and parallelism.

Our contributions in this study include the following:

(i) We propose a deadlock-free scheduling for a DISC node. To make it adaptive, we enable the proposed scheduler to periodically adjust the number of running tools for different jobs when frequent conflicts take place. The adjustment provides a trade-off between adaptability and stability;

(ii) We evaluate our scheduling via simulation using synthetic and real-world workloads and compare it with multiple-path monolithic and two-level schedulers;

(iii) We show that our approach comprehensively outperforms these other common schedulers and the simulation results show that the adaptive deadlock-free scheduling performing well in most experimental scenarios.

We structure the remainder of the paper as follows. Section 2 describes related research. Section 3 introduces the design of adaptive deadlock-free scheduling and discusses how its design attacks scalability problems without performance degradation. Section 4 addresses the assumptions and parameters that are used in simulators and explains metrics and workloads for evaluating different schedulers mentioned in Section 1, and Section 5 presents the results through comparatives diagrams. Section 6 concludes the paper with a summary of our results and provides an outlook for future work.

2. Related Work

As reviewed in the previous section, many-core systems have been extensively adopted and the underlying of a node is much more similar to a distributed system than before [4, 22]. Multiple jobs with diverse scheduling objectives are likely to corun on the same node [6, 16, 19]. To run these corunning jobs efficiently, this requires schedulers to provide flexible scheduling policies and to cope with the competition among the shared resources, which are hard to meet with traditional single-path monolithic schedulers that only have a single scheduling logic.

To address the above issues, different schedulers, such as multipath monolithic and two-level schedulers, have been proposed. The relative merits of these approaches have been discussed in Section 1. Our solution is to provide an adaptive deadlock-free scheduler and it builds on many prior ideas. Our deadlock-free approach is an example of optimistic concurrency control mechanism. This mechanism has previously

been studied in the database community for a long time. Recently, it is considered as a new parallel programming mechanism in the transaction memory community [23, 24]. Conventional programming models that apply lock-based concurrency control are known for their conceptual difficulty [25, 26]. In particular, any programming errors that are produced by programmers can easily result in local or distributed deadlock. This places a great burden on programmers, which makes the parallel programming a nightmare. To address such a tough issue, transactional memory programming model has been proposed. This programming model applies a lock-free synchronization that is intended to eliminate the deadlock and make parallel programming easy and efficient. A transaction is a finite sequence of instructions that are executed by a single process and satisfy two properties: serializability and atomicity. When a transaction completes, it will either be committed, making its changes visible to other processes instantaneously, or be aborted, causing its changes to be discarded [25].

Besides, we concur with the principles of Exokernel [27, 28] to expose sufficient information about underlying hardware resource to jobs but suggest that the coordination for competitions of underlying hardware resources should be left to user-level schedulers rather than application writers. Therefore, the adaptive deadlock-free model will expose the entire state to all user-level schedulers and they will implement scheduling by shared state. This will be further discussed in the following sections.

3. Adaptive Deadlock-Free Scheduling

Some researchers advocate revealing very little about the state of the overall system to applications and avoiding a knowledge of the existence of other applications in the system by virtual machine (VM) technology. Parallel applications are sensitive to the underlying state of the system and require information exposure that includes details of the underlying system and the actual resource usage. VMs have been able to meet severe performance penalties [19, 29–31]. Consequently, Exokernel suggests exposing information about the underlying system to the application writers to allow them to make the best decisions, i.e., permitting the application writers to write the application's own scheduling logic to suit the application's needs with the help of library operating system (lib-os for short) [27, 28]. The main problem in Exokernel is that it exposes too much detail, leaving the coordination of resource competition to the application writers, which stresses application writers and diminishes its advantages.

Therefore, we target a node and assume that each type of applications will be submitted to user-level schedulers and implemented in its own VM [19, 32] that is built by a user-level scheduler with the help of lib-os. We use physical names of underlying resources so that each user-level scheduler is capable of organizing resource lists for a node. However in contrast to Exokernel, each user-level scheduler will include a shared state list recording the current state of all underlying resources rather than the resources it owns; thus, the coordination of resource competition can be left to user-level schedulers rather to application writers.

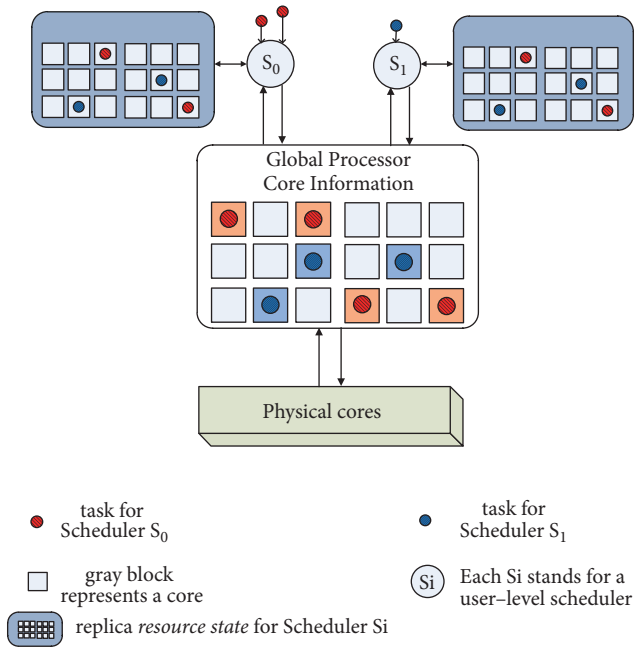


Figure 1: Schematic overview of deadlock-free scheduler.

Based on the above premises, we begin with a deadlock-free scheduling and later equip it with an adaptive control. Our goal is twofold: first, to develop a scheduler that addresses the scalability problem and, second, to deliver good scheduling performance. The proposed schedulers support multiple user-level schedulers in a composable way and allow them to compete for underlying hardware resources in an uncoordinated way. Our hope is that the resulting performance degradation caused by uncoordinated user-level schedulers will be compensated for by the benefits introduced by adaptive control.

3.1. Deadlock-Free Scheduling. Our proposed scheduler is deadlock-free because it is based on the shared state approach. We grant each user-level scheduler full access to the entire node and allow them to compete in a free-for-all manner. This approach utilizes optimistic concurrency control to mediate conflicts when user-level schedulers update the state. As shown in Figure 1, each user-level scheduler (Si) is given a resilient master copy of the resources allocation, which is called the *resource state* and maintained by user-level schedulers. At this stage, we assume that resources represent CPU cores; later, they can be extended to physical memory pages, I/O bandwidth, and network. Each user-level scheduler makes its own scheduling decision based on its *resource state*. Therefore, user-level schedulers have complete autonomy to claim any available resources, or even ones that have been acquired by another scheduler, provided they have the appropriate permissions and priority. This approach thus immediately removes two issues in the two-level scheduler approach: limited parallelism due to pessimistic concurrency control and restricted visibility of resources in a user-level scheduler.

As mentioned above, once a user-level scheduler makes a placement decision, the shared copy of the *resource state* is updated in an atomic commit similar to the transactional memory mechanism. In reality, the time between resource state synchronization and the commit attempt is a transaction defined in the transactional memory mechanism. This transaction could fail because another user-level scheduler simultaneously made a conflicting change. The conflict resolution mechanism of transactional memory guarantees that at most one of these commits will succeed if a conflict occurs. Whether or not the transaction succeeds, each user-level scheduler will resynchronize its local copy of the *resource state* and then try to run its scheduling algorithm again. This repetition of work can weaken the advantages produced by using optimistic locking.

3.2. Adaptive Control for Deadlock-Free Scheduler. The *resource state* is frequently updated by each user-level scheduler, which makes a local placement decision and attempts to commit changes back to the shared copy with an atomic commit. Where there are conflicting updates, the scheduler's transaction will be aborted and the scheduler must retry with a new *resource state*.

If conflicts and synchronizing the *resource state* among all the user-level schedulers form a bottleneck, the performance of the deadlock-free scheduler will dramatically degrade. This situation is liable to occur when the numbers of running user-level schedulers increase rapidly. The number of user-level schedulers will increase with the types and numbers of jobs. This increase causes the average number of cores that can be allocated by a user-level scheduler to decrease; i.e., there is a decrease in the exploitable parallelism that a node can provide for each user-level scheduler. A similar issue has been reported and studied in Software Transactional Memory (STM) [26, 33, 34]. The performance of a transactional system can be strongly affected by the conflict rate of transactions running concurrently, because a high degree of transactional concurrency may lead to an unacceptably high rate of aborts. The resolution of the above issues falls into two categories: performance model-based approaches [26, 35–37] and heuristic-based schemes [34, 38–40], which are summarized in Table 1.

Different from the above approaches from Table 1, our proposed scheduler tries to apply heuristic-based methods to the case of resource acquisition for their straightforward and easiness. We optimize the scheduling phase and throughput on the basis of dynamically increasing or decreasing the number of concurrent user-level schedulers in response to the fluctuating exploitable parallelism.

Our proposed scheduler should improve resource usage when exploitable parallelism is low and improve execution time when exploitable parallelism is high. We investigate the *Transaction Conflict Fraction* (TCF) as a suitable measure of exploitable parallelism; the TCF is the percentage of aborted scheduler's transactions out of the total number of all scheduler's transactions in a sample period. The TCF increases during phases with high contention, thus suggesting that the number of active user-level schedulers can be reduced and vice versa.

Based on the above idea, we apply proportional control to improve our deadlock-free scheduler; this concept has been applied in a diverse range of fields to maintain some variable within a bounded range. According to control theory terminology, the objective of our proportional control is to maintain the *process variable* TCF at a set point, despite unmeasured disturbance from the fluctuating exploitable parallelism restricted by the available hardware resources. The TCF is defined as follows:

$$TCF = \frac{\sum_{i=0}^{n-1} A_i}{\sum_{i=0}^{n-1} T_i} \tag{1}$$

The symbol Ai denotes the number of aborted transactions for user-level scheduler Si, and Ti is the total number of transactions executed over a sample period for Si. These data range between 0.0 (no conflicting transactions) and 1.0 (all conflicting transactions) and will be used to determine whether the number of active user-level schedulers (i.e., *numActiveULS*) should be increased or decreased.

To realize control of the number of user-level schedulers that can be run concurrently in responsive to automatic changes in *TCF*, we define the control algorithm as follows:

$$f(t) = \begin{cases} numActiveULS + \dfrac{numOfULS \times \Delta TCF}{\text{order of magnitude for } numOfULS}, & \Delta TCF \le 0, \\ numActiveULS - \dfrac{numOfULS \times \Delta TCF}{\text{order of magnitude for } numOfULS}, & \Delta TCF > 0 \end{cases} \tag{2}$$

where $\Delta TCF = TCF - SP$ and SP denotes a user-defined set point that is a target *TCF* that our deadlock-free scheduler tries to maintain. If the *TCF* falls below SP due to an increase in the number of transactions being committed, our model will activate user-level schedulers that have been deactivated and improve the parallelism of the deadlock-free scheduler by attempting to execute more user-schedulers concurrently. If the *TCF* increases above SP due to an increase in the number of aborted user-level scheduler (ULS) transactions, our scheduler will deactivate some active ULSs, reducing the number of concurrent ULSs competing for the underlying resources. The parameter *numberOfRunningULS* denotes the number of ULSs that have been developed in the deadlock-free scheduler. Algorithm 1 shows the process of adaptive control for the number of actively running ULSs for short).

Table 1: Comparisons between different scheduling schemes in STM.

Schemes	Definitions	Strengths	Drawback
performance model-based	describe the system performance as a function of the degree of concurrency among transactions	Accurate and be a more general one	requiring off-line workload profiling of the system for collecting measurements and parameter values to instantiate the performance model
heuristic-based methods	require the user to configure scheduler parameters, such as conflict rate thresholds	Simple and straightforward	The scheduler parameters of heuristic-based ones are required to be configured by the users. It is hard to guarantee convergence to the optimal solution

The constant *SP* determines how conservatively adaptive control treats resource usage efficiency. A low *SP*, e.g., 10%, will rapidly reduce the number of active ULSs when *TCF* increases but will cause a slower response to sudden large decreases in *TCF* and vice versa. The evaluation in Section 6 demonstrates that a low *SP* of 30% will not result in performance degradation.

Conversely, the controller of our deadlock-free scheduler calculates ΔULS using the relative gain formula described in Steps 12, 16, and 17 (together, these steps are equivalent to (2)), which allows *SP* to be a value, rather than a range. Moreover, *samplePeriod* is determined based on the overhead of executing the control loop.

4. Experiment Setups and Parameter Assumptions

We compare our proposed schedulers with two commonly used node-level schedulers, multipath monolithic and two-level. To understand the tradeoffs between the multipath monolithic, two-level, deadlock-free, and adaptive deadlock-free schedulers, we construct a simulator that is based on a many-core system. This simulator is driven by synthetic workloads, and its parameters are drawn from empirical workload distributions [41–44]. We use this approach to compare the behavior of four schedulers under the same conditions with identical workloads. The remainder of this section describes the assumptions and parameters that are used in our simulators and explains the metrics and workloads we use to evaluate the schedulers.

4.1. Model Assumption. Various scheduling algorithms have been proposed to permit node sharing between jobs. Many-core systems are used extensively, making the underlying operation of a node much more similar to a distributed system than it was previously [4]. In other words, different types of jobs are likely to share the same node [45]. This sharing amounts to the realization of supporting multiple scheduling logic schemes at the node level. The most common technique used in current node-level scheduling is multipath monolithic scheduling. It lacks parallelism and supports a large number of scheduling logic schemes at the price of adding a large amount of extra code. Because of these problems, some studies have advocated using a two-level scheduling, which lacks control of the underlying resources and loses the global knowledge of a node for determining priority. In this work, we opt for supporting multiple scheduling logic schemes in a composable and uncoordinated manner, which is enabled by Exokernel, user-level schedulers, and our proposed *resource state*. We anticipate that this approach will address the problems of a monolithic scheduler without suffering from the known drawbacks of the multipath monolithic and two-level schedulers.

System Overview and Job Models. Because modern machines that can easily apply 48 or more CPU cores to process a 100-GB dataset entirely in memory are already used in many practical applications, we assume that each node is a system with multilane high-speed networking cards, hundreds of gigabytes of memory, and terabytes of disk space. As discussed in previous studies [6, 46], jobs with different resource demands and distinct scheduling logic schemes tend to coexist on the same node. Each job consists of one or more tasks to be executed in parallel. Our goal is to design a reasonable mechanism for implementing rapid and sound resource allocation at the node-level.

Table 2: Design issues in single-path and multipath monolithic, two-level, and adaptive deadlock-free scheduling.

	Synthetic workload	Real-life workload
CPU architecture	many-core	many-core
Cores and memory request size	sampled	actual data
Initial cores and memory state	sampled	actual data
Tasks per job	sampled	actual data
Job arrival rate λ_{job}	sampled	actual data
Task duration	sampled	actual data

As the trend of BIG DATA that are reported in studies [47, 48], we target a mixture of two different types of workloads in this work: HPC and data-intensive, which will be discussed detailedly in Section 5.3. HPC workloads mostly comprise regular parallel applications, which are mainly compute-bound. Data-intensive workloads are dominated by I/O bound applications. As studied in literature [49], most large distributed scientific applications fall under the Single Program Multiple Data (SPMD) programming model, which consists of multiple identical or nearly identical tasks that make very similar demands on the system. We suppose that all tasks in a job have the same memory requirements and CPUs need and that they must progress at the same rate. The memory and CPU requirements for a task are expressed as a fraction of the total node memory and a fraction of the available CPU cores for a node that allows the task to run at maximum speed. For instance, a task might require 30% of a node's memory and use 50% of the node's CPU resources in dedicated mode. We also assume that such proportions are publicly known and will not be changed throughout the entire job execution process.

4.2. Discrete-Event Simulator. We have developed a discrete-event simulator that implements five schedulers, including the following types: single-path monolithic, two prevalent schedulers: single-path monolithic, the two most common schedulers (multipath monolithic and two-level), deadlock-free, and adaptive deadlock-free scheduler. Our simulator primarily accepts a list of jobs as the input that is likely to be scheduled on the same node. Each job is described by a submit time, a required number of tasks, one CPU requirement, one memory requirement specification, and an execution time.

Our simulator incorporates some simplification for the targeted schedulers that are summarized in Table 2, allowing us to include a broad range of operating points within a reasonable runtime. Thus, we are able to compare the behavior

```
1. If currentTime – lastSampleTime < samplePeriod then
2. goto Step 1
3. Sampling number of existing ULS, numOfULS = sampling
   // miniULS is the minimum number of user-level scheduler, for this paper one
4. while numOfULS <= miniULS
5.   goto Step 1
6. endwhile
7. if numOfULS > miniULS then
8. Calcualte TCF = numAborteds/numTransactions ×100
9.     Calculate ΔTCF = TCF – SP
10.    Get the number of active ULS, numActiveULS = getActiveULS( )
11.    Calculate the order of magnitude for numOfULS, order = log10(numOfULS)
12.    Calculate ΔULS = ΔTCF×( numOfULS/order)
13.    while ΔULS < 0
14.        activateULS(ULS[numActiveULS])
15.        numActiveULS = numActiveULS + 1
16.        ΔULS = ΔULS + 1
17.    endwhile
18.    while ΔULS > 0
19.       deactivateULS(ULS[numActiveULS])
20.        numActiveULS = numActiveULS – 1
21.        ΔULS = ΔULS - 1
22.    endwhile
23. end if
```

Algorithm 1: Process of adaptive controlling active user-level schedulers (ULS).

of all these schedulers under the same conditions and with identical workloads. The rest of this section describes our simulators.

Parameters for Simulation. The scheduler decision time ($t_{decision}$) for assigning cores and memory pages in a user-level scheduler is modeled as a linear function as follows:

$$t_{decision} = o_{job} + o_{task} \times \text{tasks per job} \quad (3)$$

where o_{job} represents the per-job overhead due to the coordination needs of cores and memory pages [50, 51] and o_{task} is the incremental cost to place each task. The values for o_{job} and o_{task} are based on estimates from real-world data-intensive workloads in the real world: o_{job} = 0.1m s and o_{task} = 50ns.

Since the dominant application in data-intensive workloads is compute-bound, we are interested in how the added I/O bound applications will influence the performance of DISC node. Our experiments in Section 5 will explore the effects of varying o_{job} (I/O bound) for I/O bound user-level scheduler, which aims to investigate how the proposed deadlock-free and adaptive deadlock-free schedulers will be affected by the longer decision times needed for a more complicated placement algorithm for I/O applications.

Poisson distribution has been adopted to simulate job arrival rate at the node level, denoted as λ_{job}. We also vary the job arrival rate to a 32-cores node to evaluate how it affects scheduling performance.

Metrics in Simulation. In this paper, we implement our evaluation at two levels: first we compare different schedulers and then we compare our proposed schedulers.

At the first level, we primarily use three metrics to evaluate the efficiency and scalability of different schedulers: *per-job wait time* speedup, *job execution time* speedup, and *busyness of scheduler*. These metrics are defined and explained below.

The perceived quality of resource allocation is evaluated by the *per-job wait time* and *job runtime to completion*. The *per-job wait time* is defined as the time until jobs begin running, i.e., the difference between the submission time of a job and the beginning of the job's first scheduling attempt. The *job runtime to completion* is defined as the difference between the beginning of the job's first runtime and its completion time. We assume that each scheduler can process one request at a time, so a busy scheduler will cause a delay in the jobs that want to queue. Consequently, the *per-job wait time* measures the depth of scheduler queues and will increase as the scheduler becomes busy, either because it receives more jobs or because its jobs take longer to schedule. Compared with single-path monolithic scheduler, the multipath monolithic, two-level, and our deadlock-free and adaptive deadlock-free schedulers achieve higher job throughput by parallelizing the process of resource allocation in different ways and executing multiple jobs in parallel if there are sufficient resources. According to Amdahl's law, we must use the speedup in *per-job wait time* and *job runtime to completion* to compare the multipath monolithic, two-level, and our deadlock-free and adaptive deadlock-free schedulers.

More notable is that *per-job wait time* depends upon the *busyness of scheduler*, which is the amount of time that the scheduler is busy making scheduling decisions. This metric increases with the *per-job decision time* (*i.e.*, o_{job}). Thus, it will also be used to evaluate different schedulers.

Several second-level metrics are used to evaluate our proposed schedulers: the deadlock-free and adaptive deadlock-free schedulers. The *busyness of scheduler* for the proposed schedulers will also increase if scheduling work must be redone because of conflicts in the deadlock-free approaches. To evaluate the occurrence probability of the latter, we utilize two more metrics. One is the average number of conflicts per overall committed transactions, called the *fraction of conflict*. A value of 0 for the *fraction of conflict* means no conflicts occurred, whereas a value of 3 indicates that the average workloads experienced three conflicts and thus requires four scheduling attempts. Another metric that we will evaluate is *waste time* in *the busyness of scheduler*, which is the fraction of time during which the scheduler is rescheduling a redo job or becomes idle. Obviously, as these two values increase, the performance of the scheduler decreases.

The values for the *busyness of scheduler* and the *fraction of conflict* are the medians of the generated traces. These traces will be described in Section 4.3. In addition, the values for *per-job wait time speedup* and *waste time* in *the busyness of scheduler* are overall averages. In the following experiments, the error bars for the *busyness of scheduler* or the *fraction of conflict* show the median absolute deviation (MAD) from the median value of the per-trace averages.

4.3. Workloads. Workload heterogeneity [42, 45] is commonplace in current data-intensive workloads. Since the data-intensive applications will converge with HPC applications in the future, our simulators aim at two main applications: data-intensive ones and traditional MPI ones [48]. There are many approaches to partitioning workloads among clusters [42, 43, 52, 53]. To simplify and obtain the critical behavior characteristics of different schedulers, we select a simple two-way split on top of our simulators between I/O bound jobs that provide end-user operations (e.g., web services), internal infrastructure services (e.g., BigTable), and compute-bound jobs that perform a computation.

We adopt synthetic workloads for part of the study, using parameters drawn from empirical workload distributions from previous studies [43, 45]. There are a number of reasons to use synthetic workloads. Real workloads are often of poor quality and may not contain all information that we require. In addition, real workloads are for specific systems, while synthetic workloads are generated using a model instantiated from multiple systems and thus can be more representative.

Through the use of Exokernel, page coloring [10, 54, 55], and other techniques, CPU cores and memory pages of a many-core node can be precisely shared, partitioned, and distributed to different applications. Thus, the total amount of allocated CPU resources in a node is constrained not to exceed 100%. CPU resource utilization of 100% can only be reached by a single job if that job is CPU-intensive and implemented using multiple threads. A CPU-intensive sequential job can only use $100/n\%$ of the node's CPU resources, where n is the number of processor cores for that node. In our synthetic trace experiments, we arbitrarily assume a 64-core node, which means that a sequential task would use at most 2% of the node's CPU resource.

As discussed [52, 53], we assume that all tasks of jobs from HPC workloads are CPU-intensive and the tasks in a one-task job are sequential. For data-intensive jobs [43], we assume that a job is made up of one or more tasks (occasionally thousands of tasks), which are mostly CPU-intensive (> 80%) but with a fast turnaround, whereas the rest are I/O-intensive (< 20%) and consume the majority of resources. The latter typically run for much longer and have fewer tasks than CPU-intensive jobs.

One thousand distinct traces of 2000 jobs were generated according to parameters drawn from an empirical distribution of workloads [43, 45] and annotated with CPU and memory requirements as described. The generated traces assume a 64-core node and thus contain jobs with between 1 and 64 tasks. Suppose that all the jobs only submit to the 64-core node.

Also, real workloads [56] from Google cluster have also been used to test our simulator. The information of workloads is collected during a 29-day period in the month of May 2011 in one of Google's production cluster cell. The cell size is more than 12,477 distinct machines, including 55,272 jobs and 1,405,572 tasks. Many jobs have a small number of tasks (less than 100); in fact, a very large number of jobs have a single task. A few jobs have over 2000 tasks, which is consistent with our assumptions for synthetic workloads.

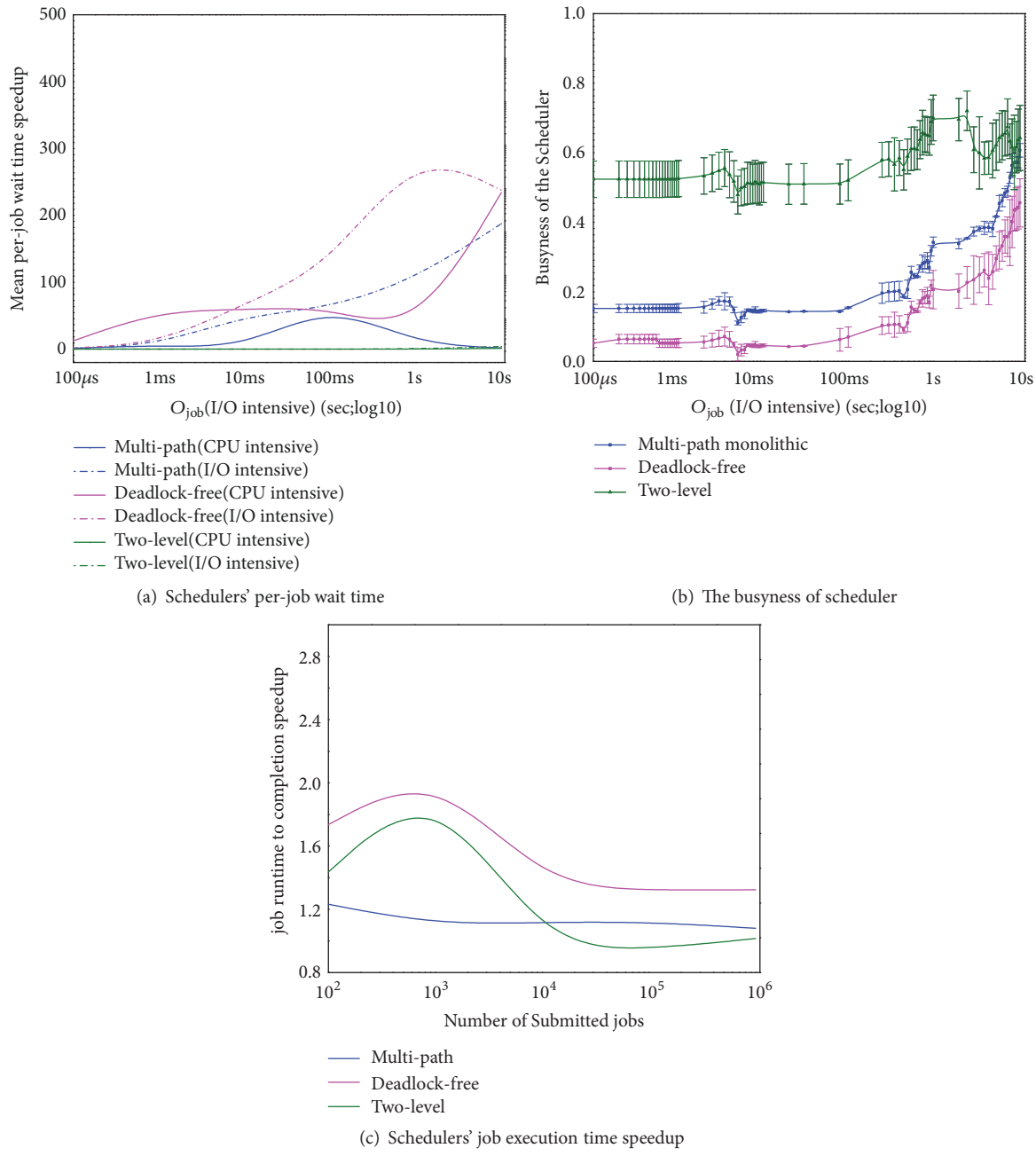
In summary, our experiments require a scheduling that can accommodate both types of jobs, flexibly support job-specific policies, and scale to an ever-growing amount of scheduling work.

5. Experiment Result Comparison and Evaluation

In this section, we construct simulators and compare the behavior of all schedulers (single-path monolithic, multipath monolithic, two-level scheduling, deadlock-free, and adaptive deadlock-free scheduling), under the same conditions and with identical workloads, as described in Section 4. Our simulator runs on a laptop with an Intel processor i7 4710MQ with 4 physical cores (8 logic cores) and 8 GB memory. Each core has a dedicated 256 KB L1 cache and a dedicated 1024 KB L2 cache. There is a 6 MB L3 cache shared by four cores.

Our baseline for comparison is a single-path monolithic scheduler. Its successor is a multipath monolithic scheduler with a fast code path for CPU-intensive jobs; we refer to this scheduler as a multipath monolithic scheduler because it still schedules only one job at a time.

The two-level scheduling is modeled on the offer-based design. We simulate a first-level scheduler that can distribute CPU cores and memory pages to user-level schedulers, one for CPU-intensive jobs and one for I/O-intensive jobs. For simplicity, we assume that a user-level schedulers can only manage the set of resources available to it when it begins a scheduling attempt for a job. Subsequently, any offers that arrive during the attempt are ignored. Resources unused at the end of a job-scheduling activity are returned to the first-level scheduler; they can be reoffered if the user-level scheduler is the furthest one below its fair share. The gang



(a) Schedulers' per-job wait time

(b) The busyness of scheduler

(c) Schedulers' job execution time speedup

Figure 2: The performance of schedulers.

scheduling used in this first-level scheduling is very efficient. Thus, we assume that it takes 100 ns to make a resource offer.

We use the simulator to explore our adaptive deadlock-free approach. First, however, we begin with its predecessor, the deadlock-free approach. We again simulate two user-level schedulers, one addressing CPU-intensive job and the other handling I/O-intensive jobs. Each user-level scheduler is given a private, local, frequently updated copy, i.e., the *resource state*. Once a user-level scheduler makes a scheduling decision for a job, it will refresh its local copy of the *resource state* and synchronize it with the states of other user-level schedulers. We define the operation of updating a *resource state* as a transaction. If there are no conflicts, then the entire transaction is accepted; otherwise, only those changes that do not result in an overcommitted core or memory page are accepted. All these changes materialize the *resource state* in a single core for a computing node; the *resource state* in our deadlock-free scheduling can be set to read or write and the scheduling transactions are submitted directly to the different cores. The cores themselves check for conflicts and accept or reject the changes. This checking allows our deadlock-free scheduler to make progress even when the shared state is temporarily unavailable. A problem arises because we use resource-fit in our deadlock-free scheduling. Obviously, if

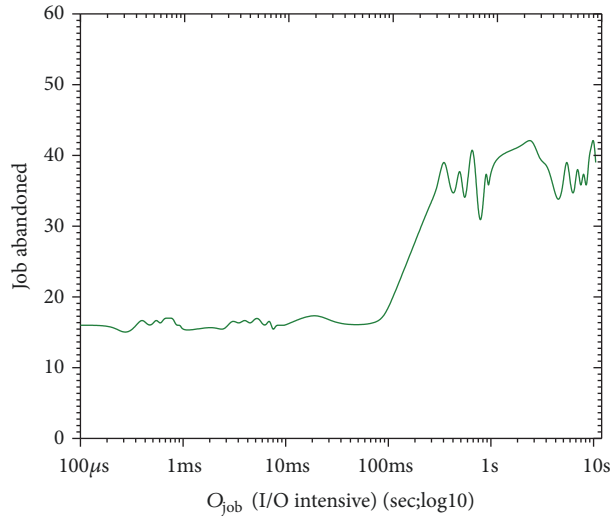


Figure 3: Two-level approach: unscheduled job while varying o_{job} (I/O-intensive).

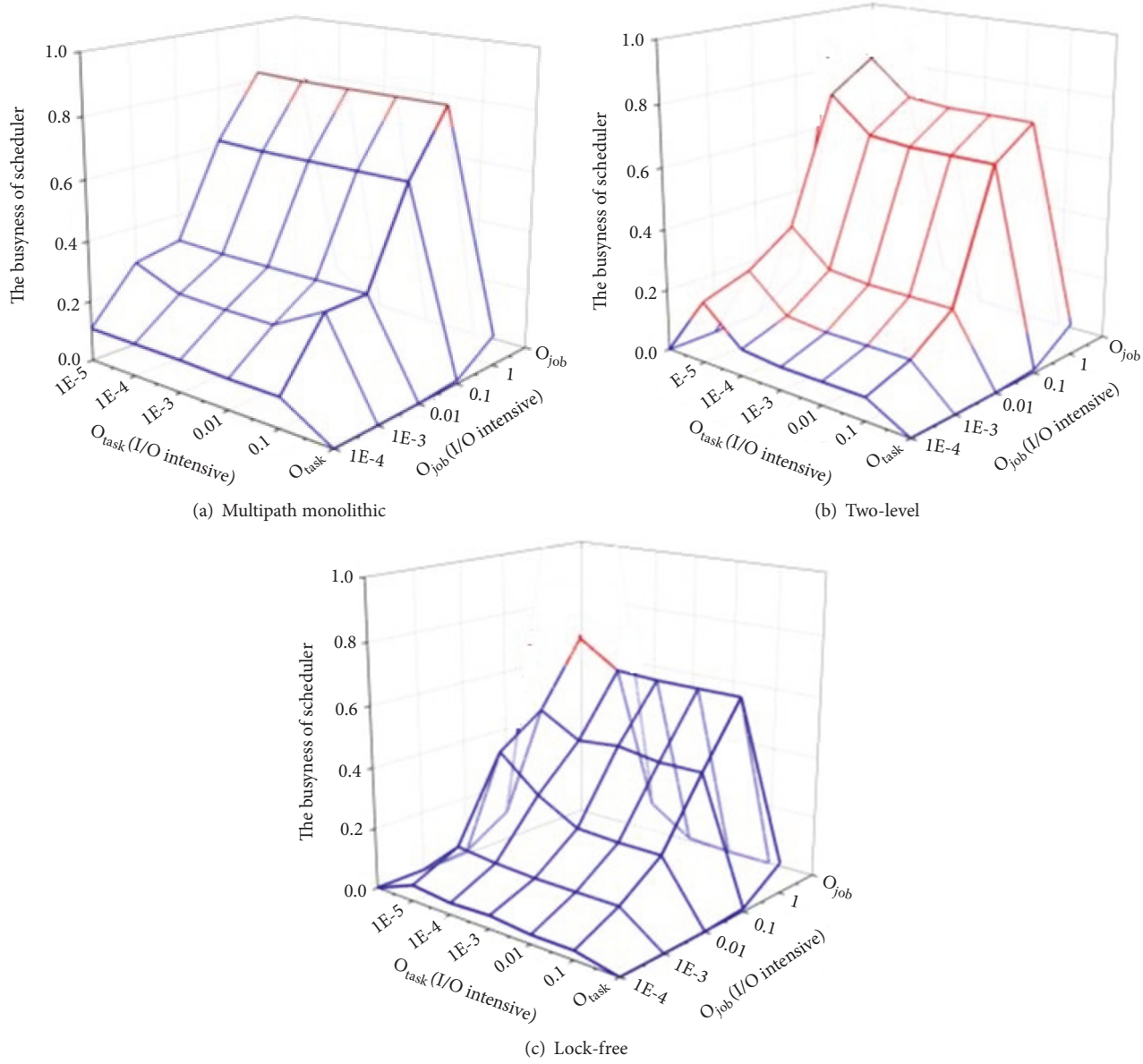
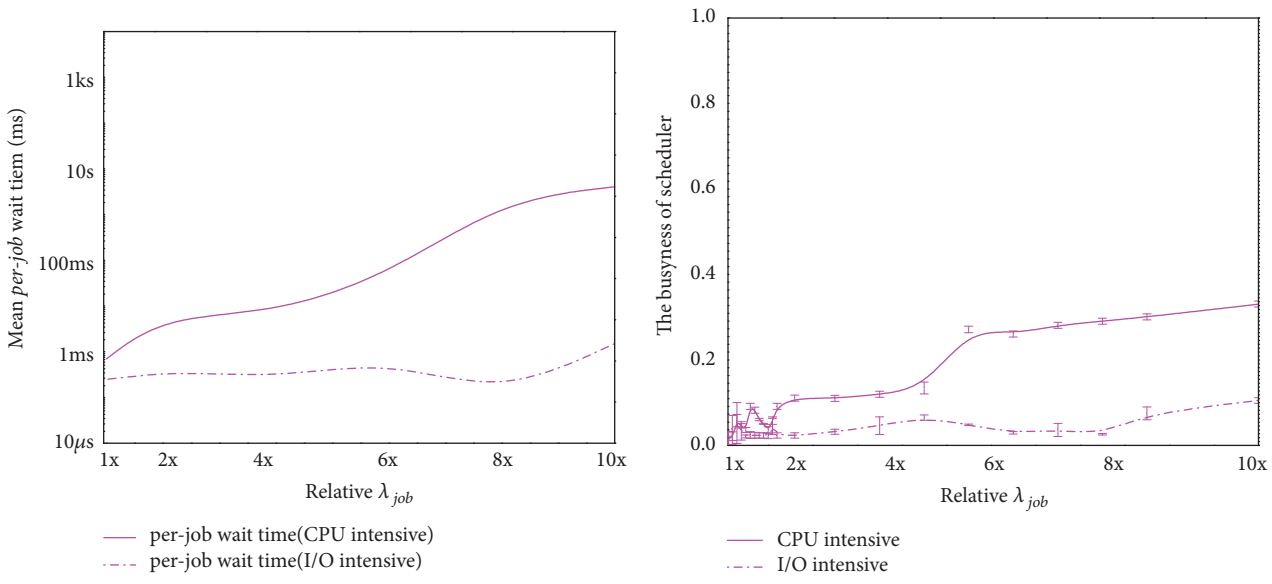


Figure 4: Impact of varying the job overhead o_{job} (I/O-intensive) (right axis) and cost to place task o_{task} (I/O-intensive) (left axis) on scheduler busyness (z-axis) in different scheduling schemes. Red shading of a 3D graph means that part of the jobs remained unscheduled.



(a) Per-job wait time

(b) Scheduler busyness

FIGURE 5: Performance of deadlock-free scheduling by varying the arrival rate for the CPU-intensive job λ_{job} (CPU-intensive).

there is no feedback control, the conflict fraction is likely to rapidly increase with an increasing number of user-level schedulers. The situation could worsen because starvation might occur. To overcome this problem with deadlock-free scheduling, we use the global TCF (discussed in Section 4.2) as an index to control the number of running user-level schedulers. This approach will be further explored in the Section 5.2.

Because all the schedulers support two scheduling logic schemes, CPU-intensive and I/O-intensive, we divide the jobs into CPU-intensive and I/O-intensive throughout the experiments.

5.1. Comparison by Varying Parameters. In this experiment, we keep the decision time for the CPU-intensive user-level scheduler constant and vary the decision time for the I/O-intensive user-level scheduler by adjusting o_{job}(I/O-intensive). Figure 2 shows the performance of the multipath monolithic, two-level and deadlock-free schedulers.

Multipath Monolithic Model. With a fast path for CPU-intensive jobs in the multipath monolithic model, both the average *per-job wait time* and the *busyness of scheduler* increase significantly even with long decision times for I/O-intensive jobs. Although the majority of jobs are CPU-intensive, these jobs can still become stuck in a queue behind the slow-to-schedule I/O-intensive jobs or face head-of-line blocking. Scalability remains limited by the processing capacity of the multipath monolithic scheduler (Figures 2(a) and 2(b)). Parallel processing is required to mitigate this limitation.

Two-Level Model. Figure 2(b) illustrates that the busyness of the CPU-intensive user-level scheduler is much higher than in the monolithic multipath case. This difference is a consequence of the interaction between the offer-based two-level model and the I/O user-level scheduler's long scheduling decision time. The offer-based two-level model achieves fairness by alternately offering all the available computing resources to different user-level schedulers, predicated on the assumptions that resources will become available frequently, and scheduler decisions will be rapid. Thus, a long scheduler decision time indicates that nearly all many-core resources have been in use for a long time, making them inaccessible to other user-level schedulers. These resources are often insufficient to schedule CPU-intensive job of above-average size, thus, the CPU-intensive user-level scheduler will not make progress, while the I/O-intensive user-level scheduler will hold an offer. The I/O-intensive user-level scheduler will keep trying; consequently, a number of tasks will be abandoned because they were not scheduled before the retry limit defined in Section 4.2 in the offer-based two-level mode case (Figure 3).

This problem occurs because of the two-level scheduler's assumption of quick scheduling decisions, small tasks, and sufficient resources, which does not hold for I/O-intensive jobs. The two-level scheduler could be extended to make only fair-share offers; this would complicate the first-level scheduler logic. In reality, each user-level scheduler can only perceive a small fraction of the available resources, and the quality of the placement decisions for large or fastidious jobs might decrease.

Deadlock-Free. In the deadlock-free approach, the speedup in the average *per-job wait time* (Figure 2(b)) and the schedulers' job execution time (Figure 2(c)) is comparable to the times for the multipath monolithic and two-level schedulers. This similarity suggests that conflicts and interference are comparatively rare. This suggestion is confirmed by the graph of the *busyness of scheduler* (Figure 2(b)). Unlike the offer-based

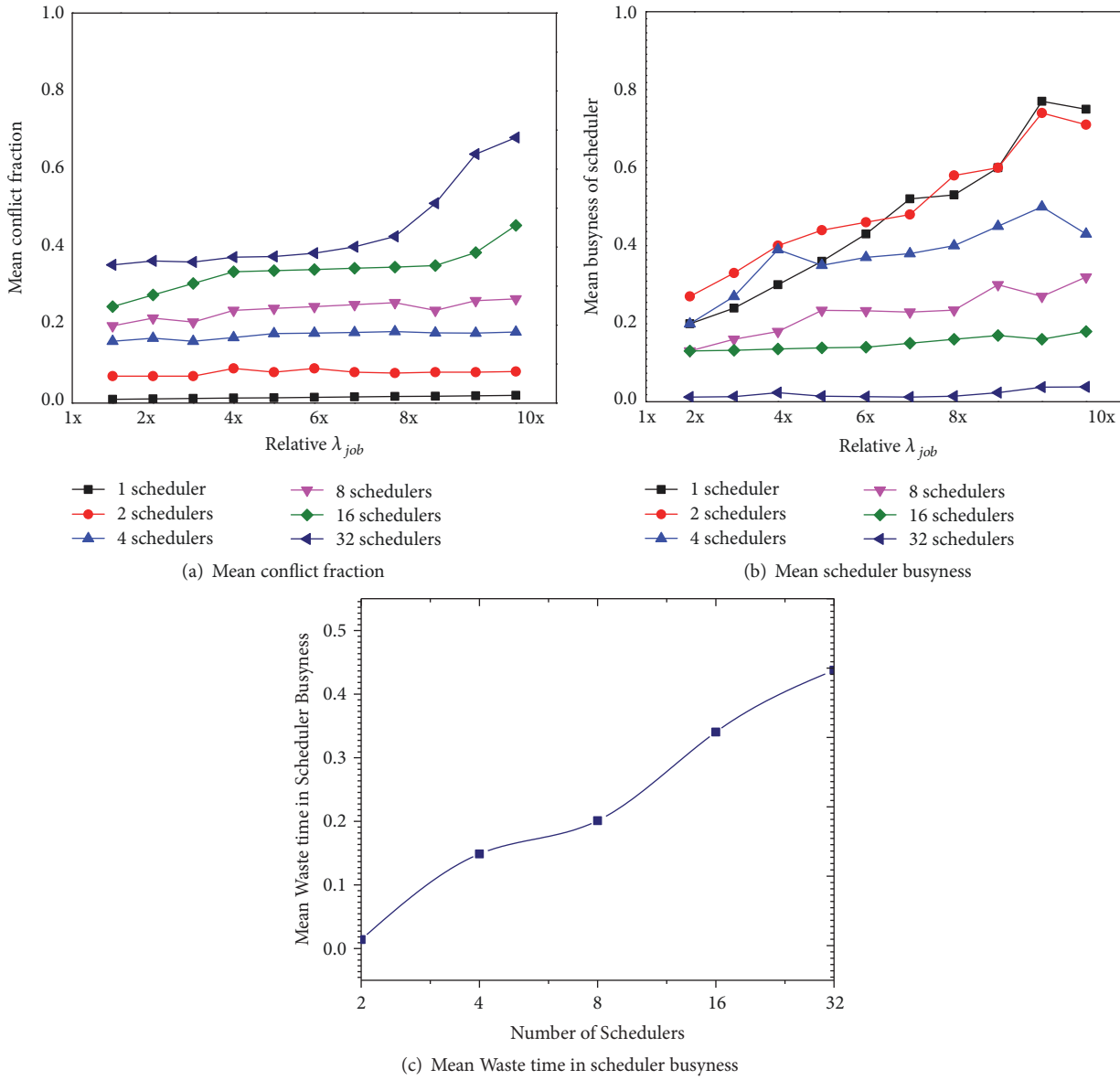


Figure 6: Additional metrics to evaluate deadlock-free scheduling by varying the arrival rate for the CPU-intensive job, λ_{job} (CPU-intensive)(1.0 is the default rate.).

two-level mode (Figure 3), the deadlock-free scheduler manages to schedule all tasks in the job. Furthermore, this scheduler does not suffer from the head-of-line blocking that occurs in monolithic multipath implementation; the lines for CPU-intensive and I/O-intensive jobs are independent.

We also evaluate the effect of scaling o_{task} as an additional dimension, and the results are shown in Figure 4. The results demonstrate that the multipath monolithic scheduler is not scalable. Even when adding the multipath feature to support multiple scheduling logic schemes, head-of-line blocking remains a problem for CPU-intensive jobs. In other words, a multipath monolithic scheduling might not be able to scale to the workloads that we project for large nodes. Composable scheduler implementations are supported by the offer-based two-level scheduler, but the advantages of this technique are diminished by its use of a pessimistic concurrency control mechanism, which fails to handle jobs with long decision times. Therefore, the scheduler could not handle much of the heterogeneous load offered to it.

The deadlock-free approach offers competitive, scalable performance, supports independent scheduler implementations, and exposes all allocation state to the schedulers. Our results indicate that the deadlock-free approach can scale to tens of user-level schedulers and to challenging workloads, which will be further discussed in the following Section 5.2.

5.2. Scalability Exploration for Adaptive Deadlock-Free Scheduling. In this section, we examine how the deadlock-free model scales as the workload changes. For this purpose, we increase the job arrival rate λ_{job} of the CPU-intensive

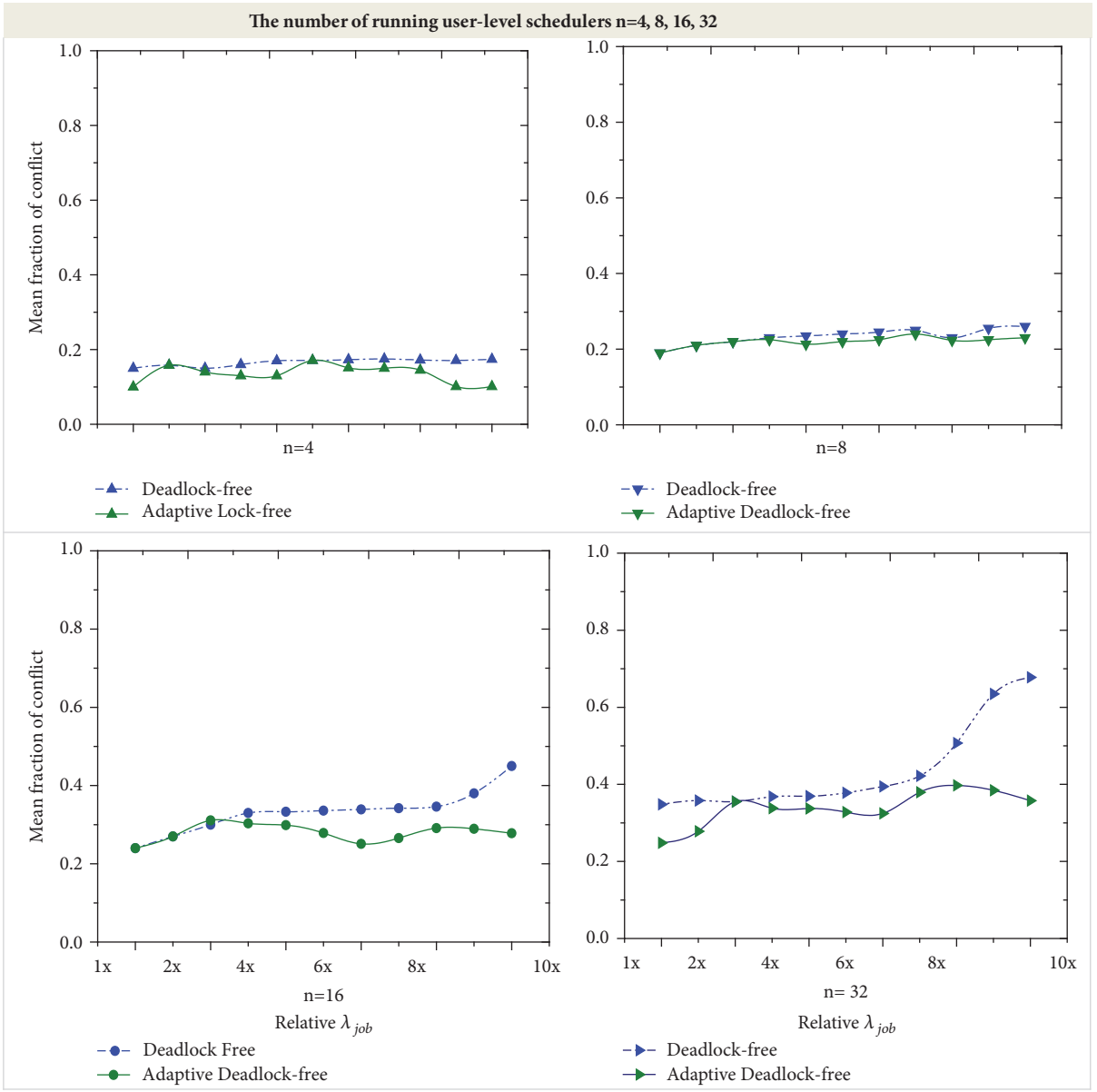


Figure 7: Comparison of fraction of conflict between adaptive deadlock-free scheduling and Deadlock-free scheduling by varying the arrival rate for the CPU-intensive job, λ_{job} (CPU-intensive) (1.0 is the default rate.).

schedulers. As demonstrated in Figure 5, both the *per-job wait time* and the *busyness of scheduler* increase. In the CPU-intensive case, this is due to the higher job arrival rate, whereas in the I/O-intensive case, it is due to additional conflicts.

There are two parameters of our adaptive schemes needed to be set: sample interval and target TCF range. Through experimentation these were set to a sample interval of 10 seconds, with lower TCF threshold of 40% and upper threshold of 70%. Because the CPU-intensive user-level scheduler represents the main scalability bottleneck, we repeat the same scaling experiment with multiple CPU-intensive user-level schedulers in order to test the ability of the deadlock-free model to scale to larger loads. The CPU-intensive scheduling work is load-balanced across the schedulers using a simple hashing function. As expected, the *fraction of conflict* increases with the number of user-level schedulers, because a greater possibility of conflict exists (Figure 6(a)). However, the *busyness of scheduler* decreases with the addition of more user-level schedulers (Figure 6(b)), which is not an encouraging. By exploring the wasted time in scheduler busyness shown in Figure 6(c), we infer that the decreased busyness of the scheduler observed in Figure 6(b) might result from the high conflict fraction, which leaves some of the schedulers idle, awaiting sufficient resources for the next schedule.

Therefore, we attempt to improve the deadlock-free model with automated control, as described in Section 4. The resulting adaptive deadlock-free scheduling can not only scale to highly CPU-intensive jobs but also provide good behavior for I/O-intensive jobs. We repeat the same input

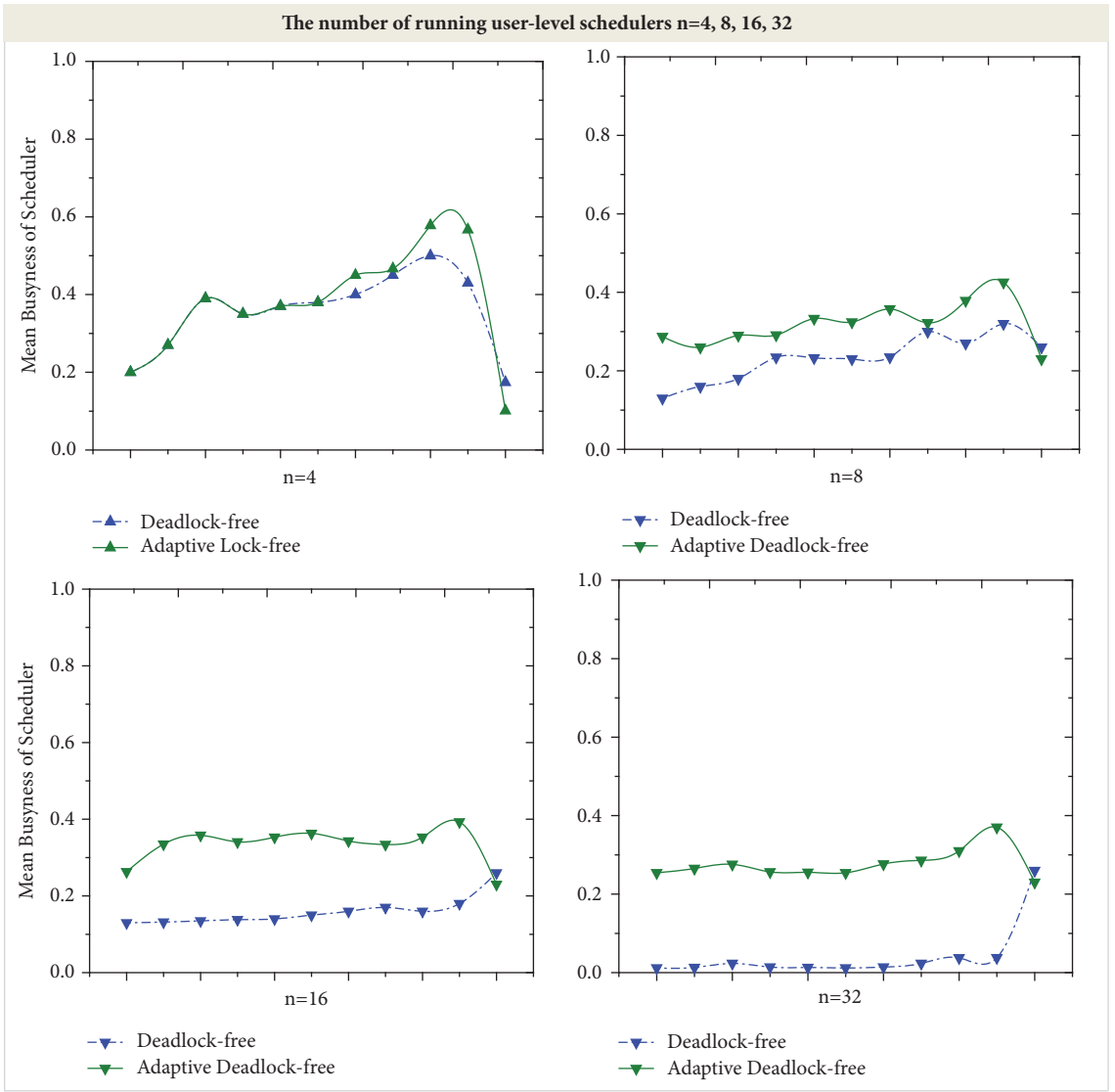


Figure 8: Comparison of busyness of scheduler between adaptive deadlock-free scheduling and deadlock-free scheduling by varying the arrival rate for the CPU-intensive job, λ_{job} (CPU-intensive) (1.0 is the default rate.).

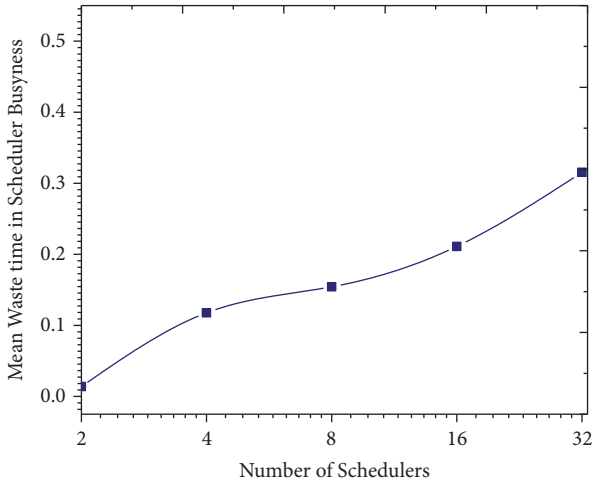
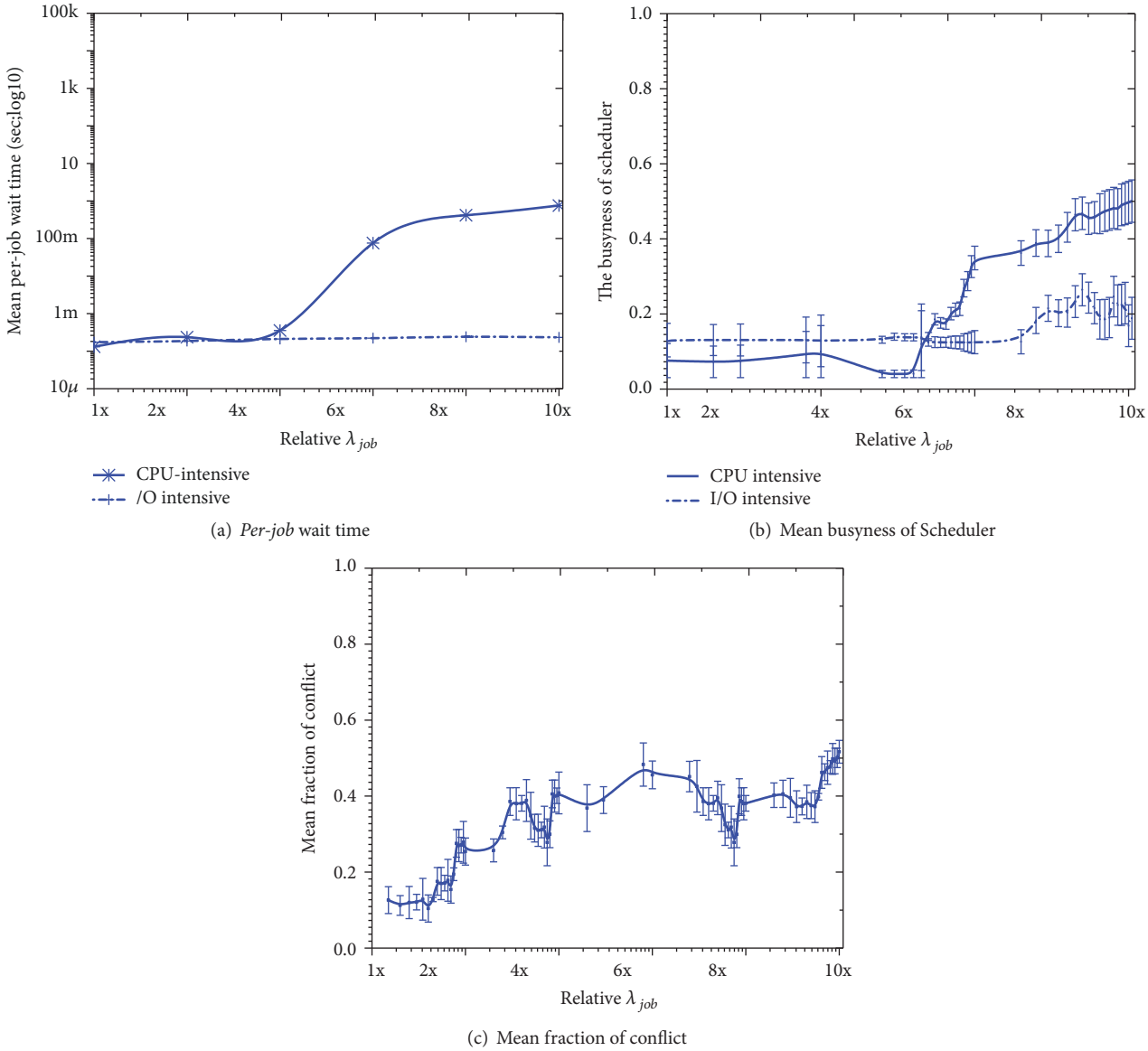


Figure 9: Mean waste time in scheduler busyness.



(a) *Per-job* wait time

(b) Mean busyness of Scheduler

(c) Mean fraction of conflict

Figure 10: Results for usage of real-life workload to evaluate adaptive deadlock-free scheduling.

for adaptive deadlock-free scheduling, and the results are shown in Figures 7, 8, and 9. Encouragingly, these results demonstrate a large improvement in the *fraction of conflict*, while moderate *busyness of scheduler* is maintained, which is consistent with our assumptions. Compared to Figure 6(c), the mean waste time in scheduler busyness shown in Figure 9 is greatly reduced. Importantly, these results suggest that the adaptive deadlock-free approach can scale to tens or even hundreds of user-level schedulers and to more challenging workloads.

5.3. Improvement to Deadlock-Free Scheduling with Automated Control. Having compared the different schedulers using the simulator with synthetic workload, we use real-life workload to test our adaptive deadlock-free scheduling approach.

We repeat the process similar to that described in Section 5.2 with the real-life workload. Similar results to those shown in Figure 10 were produced which gives us confidence in our deadlock-free scheduling method.

6. Conclusion and Future Work

In this study, we proposed a novel adaptive deadlock-free scheduling that can be applied to a single node in a DISC system. This proposed scheduling solves the scalability problem for current node-level schedules through adaptive optimistic locking control. We compared the adaptive deadlock-free model and its predecessor with two commonly used node-level schedulers, multipath monolithic and two-level scheduling, using both synthetic and real-world workloads. In our simulations, all the targeted schedulers were simplified to

examine abroad range of operating points within a reasonable runtime, allowing us to compare the behavior of all the targeted schedulers under the same conditions and with identical workloads.

The results show that our approach outperforms the other schedulers with the respect to scheduling efficiency and scalability without performance degradation. Our approach also results in large improvements in the parallelism and scalability of the node-level scheduling, because providing the independent scheduler implementations with a knowledge of the underlying resource allows them to compete for resources in an uncoordinated manner and adjusts active scheduler implementations dynamically when conflicts occur.

Our future work will fall into three categories. First, we will focus on approaches that provide global guarantees (fairness, starvation avoidance, etc.) in the adaptive deadlock-free scheduler. Second, our current approach simply uses the global TCF as an indicator to adjust the deadlock-free scheduling, which will randomly deactivate any user-level scheduler. Therefore, as a next step, our works will investigate more complex indicators to ascertain how they influence our adaptive deadlock-free approach. Finally, it is not easy to set the optimal parameter configuration that may depend on both workload and system-level settings. Our proposed heuristic-based approach is hard to guarantee convergence to the optimal solution for relieving performance degradation of coscheduling. We will explore and implement different approaches to optimize the scheduling phase of our proposed scheduler in the future study, such as the Markov Chain method proposed by previous study [26] and machine learning-based approach proposed by study [35].

Data Availability

(1) The original data used to support the findings of this study have been deposited in the baidu yunpan repository https://pan.baidu.com/s/1GQx__u0jAjB_hJgTy9S5Q. (2) The simulator used to support the findings of this study is available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the Foundation of China (Grant nos. 61402183 and 61070015), the Guangdong Provincial Science and Technology Projects (Grant nos. 2014B010110004, 2014B010117001, 2014A010103022, and 2014A010103008), and the Foundation of Guangdong (Grant no. 10351806001000000), and Guangzhou major research collaborative innovation projects (Grant no. 201604016074).

Supplementary Materials

The specific data for Figures 2–5 is available free of charge. *(Supplementary Materials)*

References

[1] Y. Chen, C. Chen, Y. Yin, X. Sun, R. Thakur, and W. Gropp, "Rethinking High performance computing system architecture for scientific big data applications," in *Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 1605–1612, Tianjin, China, August 2016.

[2] E. Anderson and J. Tucek, "Efficiency matters!," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, p. 40, 2010.

[3] B. Wadhwa and A. Verma, "Energy and carbon efficient VM placement and migration technique for green cloud datacenters," in *Proceedings of the 2014 7th International Conference on Contemporary Computing, IC3 2014*, pp. 189–193, India, August 2014.

[4] A. Baumann, S. Peter, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. why isn't your OS?" in *Proceedings of the in Conference on Hot Topics in Operating Systems*, pp. 12-12, 2009.

[5] A. Rasmussen, G. Porter, M. Conley et al., "TritonSort: a balanced large-scale sorting system," in *Proceedings of the Usenix Conference on Networked Systems Design and Implementation*, pp. 29–42, 2011.

[6] A. D. Breslow, L. Porter, A. Tiwari et al., "The case for colocation of high performance computing workloads," *Concurrency and Computation: Practice and Experience*, vol. 00, pp. 1–20, 2012.

[7] J. Gray, "Sort benchmark home page," http://sortbenchmark.org/.

[8] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014*, pp. 127–143, USA, March 2014.

[9] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *Proceedings of the Usenix Technical Conference*, pp. 29–42, Boston, MA, USA, 2010.

[10] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the the fourth ACM european conference*, p. 89, Nuremberg, Germany, April 2009.

[11] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel et al., "Concurrent direct network access for virtual machine monitors," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 306–317, 2007.

[12] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM Sigops Operating Systems Review*, vol. 43, pp. 76–85, 2009.

[13] H. Pan, B. Hindman, and K. Asanovic, "Lithe: enabling efficient composition of parallel libraries," in *Proceedings of the Hotpar*, pp. 11-11, 2009.

[14] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, and R. Morris, "Corey: an operating system for many cores," in *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation, OSDI 2008*, pp. 43–57, San Diego, Calif, USA, 2008.

[15] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe, "Decoupling cores, kernels, and operating systems," in *Proceedings of the in Usenix Conference on Operating Systems Design and Implementation*, pp. 17–31, 2014.

[16] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," in *Proceedings of the 9th ACM European Conference on Computer Systems, EuroSys 2014*, Netherlands, April 2014.

[17] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," in *Proceedings of the 11th European Conference on Computer Systems, EuroSys 2016*, UK, April 2016.

[18] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf et al., "Resource management in the tessellation manycore OS," in *Proceedings of the Usenix Workshop on Hot Topics in Parallelism*, 2010.

[19] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new OS abstractions," in *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011*, Portugal, October 2011.

[20] X. Sun, N. Ansari, and R. Wang, "Optimizing resource utilization of a data center," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2822–2846, 2016.

[21] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *The Computer Journal*, vol. 40, no. 12, pp. 33–37, 2007.

[22] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto, "Maximizing power efficiency with asymmetric multicore systems," *Communications of the ACM*, vol. 52, no. 12, p. 48, 2009.

[23] M. Herlihy and J. E. Moss, "Transactional memory," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 289–300, 1993.

[24] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[25] K. Asanovic, B. Author, and J. Demmel, *The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View*, 2008.

[26] P. D. Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, "Markov chain-based adaptive scheduling in software transactional memory," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 373–382, USA, May 2016.

[27] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.

[28] Chen and Benjie, "Multiprocessing with the exokernel operating system," *Massachusetts Institute of Technology*, vol. 13, pp. 18–24, 2000.

[29] I. Pietri and R. Sakellariou, "Mapping virtual machines onto physical machines in cloud computing: a survey," *ACM Computing Surveys*, vol. 49, no. 3, 2016.

[30] R. J. Creasy, "Origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

[31] R. P. Goldberg, "Survey of virtual machine research," *The Computer Journal*, vol. 7, no. 6, pp. 34–45, 1974.

[32] V. Uhlig, "The mechanics of in-kernel synchronization for a scalable microkernel," *ACM Sigops Operating Systems Review*, vol. 41, pp. 49–58, 2007.

[33] I. Watson, C. Kirkham, and M. Lujan, "A study of a transactional parallel routing algorithm," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 388–400, Brasov, Romania, September 2007.

[34] M. Ansari, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proceedings of the International Euro-Par Conference on Parallel Processing*, pp. 719–728, 2008.

[35] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2012*, pp. 278–285, USA, August 2012.

[36] A. Dragojevic and R. Guerraoui, "Predicting the scalability of an STM: a pragmatic approach," in *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[37] P. D. Sanzo, F. D. Re, D. Rughetti, B. Ciciani, and F. Quaglia, "Regulating concurrency in software transactional memory: an effective model-based approach," in *Proceedings of the 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013*, pp. 31–40, USA, September 2013.

[38] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'08*, pp. 169–178, Germany, June 2008.

[39] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Proceedings of the Revised Selected Papers of the First International Conference on Networked Systems*, pp. 233–247, 2013.

[40] A. Dragojevi, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pp. 7–16, 2009.

[41] S. Di, D. Kondo, and W. Cirne, "Characterization and Comparison of Google Cloud Load versus Grids," http://hal.archives-ouvertes.fr/hal-00705858, 2012.

[42] M. Alam, K. A. Shakil, and S. Sethi, "Analysis and clustering of workload in google cluster trace based on resource usage," in *Proceedings of the 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pp. 740–747, Paris, France, August 2016.

[43] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013*, pp. 351–364, Czech Republic, April 2013.

[44] H. Li, D. Groep, and L. Wolters, "Workload characteristics of a multi-cluster supercomputer," in *Proceedings of the International Conference on Job Scheduling Strategies for Parallel Processing*, pp. 176–193, 2004.

[45] G. P. Rodrigo, P.-O. Östberg, E. Elmroth, K. Antypas, R. Gerber, and L. Ramakrishnan, "Towards understanding HPC users and systems: A NERSC case study," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 206–221, 2018.

[46] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, "Xen and the art of cluster scheduling," in *Proceedings of the VTDC 2006—2nd International Workshop on Virtualization Technology in Distributed Computing; Held in Conjunction with SC06*, USA, November 2006.

[47] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[48] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and HPC convergence," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 10044, pp. 3–17, 2016.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, and R. Katz, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pp. 429–483, 2011.

[50] J. T. Havill, W. Mao, and V. Dimitrov, "Improved parallel job scheduling with overhead," in *Proceedings of the in Joint Conference on Information Sciences*, 2008.

[51] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure, "On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems," in *Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER 2015*, pp. 428–437, USA, September 2015.

[52] H. You and H. Zhang, "Comprehensive workload analysis and modeling of a petascale supercomputer," in *Job Scheduling Strategies for Parallel Processing*, vol. 7698 of *Lecture Notes in Computer Science*, pp. 253–271, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[53] J. Emeras, S. Varrette, M. Guzek, and P. Bouvry, "Evalix: classification and prediction of job resource consumption on HPC platforms," in *Proceedings of the Intl. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 15)*, pp. 102–122, 2015.

[54] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems*, vol. 28, no. 4, 2010.

[55] S. Ahrndt, J. Fähndrich, and S. Albayrak, "Cache-hierarchy contention-aware scheduling in CMPs," *IEEE Transactions on Parallel & Distributed Systems*, vol. 25, pp. 581–590, 2014.

[56] "Traces of Google workloads," http://code.google.com/p/googleclusterdata/.